1. Einleitung

1.1. Das Thema des Buches

Dieses Buch handelt von *modellgetriebener Softwareentwicklung* oder *Model-Driven Software Development*, kurz *MDSD*. Eine etwas ungenauere, aber gängige Bezeichnung für diese Disziplin lautet *Model-Driven Development (MDD)*

Vielleicht werden Sie sich als Leser fragen, warum wir ein solches Buch schreiben. Wir denken, dass modellgetriebene Softwareentwicklung eine große Bedeutung hat und in Zukunft eine noch viel größere haben wird. Es handelt sich im Grunde um die natürliche Fortsetzung der Programmierung, wie wir sie heute kennen.

Die Verwendung von Modellen in der Softwareentwicklung hat zwar eine lange Tradition und spätestens seit der Definition der Unified Modeling Language (UML) auch eine zunehmende Verbreitung, doch handelt es sich dabei in aller Regel „nur“ um eine *Dokumentation* von Softwaresystemen, insofern als dass es lediglich eine gedankliche Verbindung zwischen Modell und Softwareimplementierung gibt. Diese Form der Modellnutzung in einem Entwicklungsprozess nennen wir *modellbasiert.* Sie birgt zwei gravierende Nachteile in sich: Zum einen sind Softwaresysteme nicht statisch, sondern insbesondere während der ersten Phasen ihres Lebenszyklus zum Teil starken Änderungen unterworfen. Die Dokumentation muss daher sorgfältig angepasst werden – dies kann je nach Detaillierungsgrad sehr aufwändig sein – oder sie wird inkonsistent. Zum anderen tragen solche Modelle nur mittelbar etwas zum Fortschritt bei, da ja erst durch die Interpretation der Softwareentwickler daraus Code wird. Diese Gründe veranlassen nicht ganz zu unrecht viele Programmierer, Modelle als Overhead zu betrachten und sie allenfalls als Zwischenergebnisse anzusehen.

Die *modellgetriebene* Softwareentwicklung geht einen anderen Weg: Modelle sind hier nicht nur Dokumentation, sondern sie sind

gleichzusetzen mit Code – die Umsetzung ist automatisiert. Ein Vergleich mit ausgereiften Ingenieursdisziplinen wie dem Maschinenbau lässt die Idee plastisch werden: Stellen Sie sich zum Beispiel eine CNC-Fräse vor, die mit CAD[1]-Daten gespeist wird und dieses Modell vollautomatisch in ein entsprechendes Werkstück umsetzt. Oder nehmen wir eine Produktionsstraße im Automobilbau: Sie geben Ihre Bestellung inklusive Ausstattungsmerkmalen (das „Modell“ Ihres Autos) auf, die eigentliche Herstellung ist zu einem Großteil automatisiert.

Diese Beispiele zeigen bereits, dass der fachliche Bereich sowohl für die Modelle als auch für die Herstellungsautomation essenziell ist: Weder die kundenorientierte „Modellierungssprache“ (Bestellformulare) eines Automobilherstellers noch seine Produktionsstraße sind z.B. dazu geeignet, Fertighäuser herzustellen. Diesen fachlichen Geltungsbereich bezeichnet man als *Domäne*.

Das Ziel von MDSD ist es daher, domänenspezifische Abstraktionen zu finden und diese einer formalen Modellierung zugänglich zu machen. Dadurch entsteht ein hohes Automationspotenzial für die Fertigung von Software und dies wiederum lässt sich in den allermeisten Fällen in eine deutliche Produktivitätssteigerung umsetzen.

Auch verbessern sich dadurch Qualität und Wartbarkeit von Softwaresystemen. Zudem können die Modelle von Domänenexperten verstanden werden. Es handelt sich um einen ähnlichen Evolutionsschritt wie bei der Einführung der ersten Hochsprachen in der Ära der Assemblerprogrammierung. Das Adjektiv „getrieben“ („driven“) in MDSD soll im Gegensatz zu „basiert“ deutlich machen, dass Modelle bei diesem Paradigma eine zentrale, treibende Position einnehmen und mindestens den gleichen Stellenwert wie Quellcode besitzen.

Um das Konzept der „domänenspezifischen Modelle“ erfolgreich umsetzen zu können, werden vor allem drei Dinge benötigt:

- Domänenspezifische Sprachen, um die Modelle überhaupt formulieren zu können
- Sprachen, in welchen die notwendigen Modell-zu-Code-Transformationen ausgedrückt werden können
- Compiler, Generatoren oder Transformatoren, welche in der Lage sind, die Transformationsdefinitionen anzuwenden, um aus domänenspezifischen Modellen Programme zu erzeugen, die auf vorhandenen Plattformen ausgeführt werden können.

---

[1] CAD = Computer Aided Design

Im Rahmen der modellgetriebenen Entwicklung werden zwar oft grafische Modelle verwendet, dies ist jedoch keineswegs zwingend und auch nicht immer opportun – textuelle Modelle sind genauso gut möglich. Typischerweise werden diese Modelle dann in programmiersprachlichen Quellcode übersetzt, um anschließend kompiliert und ausgeführt zu werden.

1.1.1. MDA

Wenn Sie bereits mit der Model Driven Architecture (MDA™) der Object Management Group (OMG) vertraut sind, werden Sie nun vielleicht sagen: „Das klingt wie MDA“. Zum Teil ist das auch richtig. MDA hat prinzipiell ähnliche Ansätze, unterscheidet sich aber in verschiedenen Details – zum Teil auch in der Motivation – und nimmt tendenziell Einschränkungen vor, wie z.B. die Fokussierung auf UML-basierte Modellierungssprachen. MDSD im Allgemeinen macht diese Einschränkungen nicht. Das Ziel von MDA ist in erster Linie Interoperabilität zwischen Werkzeugen und auf längere Sicht die Standardisierung von Modellen für populäre Anwendungsbereiche. MDSD hingegen zielt auf die Bereitstellung von praktisch einsetzbaren Methodenbausteinen ab, welche heute im Zusammenhang mit modellgetriebenen Ansätzen im Allgemeinen einsetzbar sind, und zwar unabhängig von der Werkzeugwahl oder dem Reifegrad der OMG-MDA-Standards.

Zum aktuellen Zeitpunkt (Ende 2004) befindet sich der Standardisierungsprozess der MDA zum Teil noch in den Kinderschuhen. Zum einen muss man daher von der originären MDA-Vision einiges weglassen und andere Dinge recht pragmatisch interpretieren, um einen in der Praxis wirklich tauglichen Ansatz zu erhalten. Zum anderen ist eine praktische *Methoden*unterstützung für MDA nicht unbedingt im Fokus der OMG. Dies wird zum Teil sicherlich auch durch die Zielsetzung der MDA widergespiegelt. Die Relation zwischen MDA und MDSD werden wir im Rahmen dieses Buches noch genauer betrachten, im Prinzip kann man aber sagen, dass die MDA eine Standardisierungsinitiative der OMG zum Thema MDSD ist.

1.2. Zielgruppen

Bestimmte Konzepte, Begrifflichkeiten und Grundlagen müssen von allen Projektbeteiligten verstanden werden, sonst lässt sich ein MDSD-

Projekt nicht erfolgreich abschließen. Das Buch beschreibt diese Dinge vor allem in den einführenden Kapiteln.

1.2.1. Architekten

Modellgetriebene Entwicklung ist in dreierlei Hinsicht von Bedeutung für Software-Architekten: Zum einen erfordert der Ansatz eine klare und formale Definition der architektonischen Konzepte einer Anwendung. Des Weiteren findet MDSD oft nicht nur im Rahmen der Entwicklung einer einzelnen Anwendung statt, sondern im Kontext von Produktlinien/Software-Systemfamilien. Diese besitzen ihre eigenen architektonischen Anforderungen, die von den Architekten adressiert werden müssen. Des Weiteren kommt eine ganz neue Sichtweise und Art der Entwicklung ins Projekt, bedingt durch die Trennung von Modellen und Modellierungssprachen, Programmen und Generatoren, entsprechenden Tools sowie spezifische prozessuale Aspekte. All diese Themen werden im vorliegenden Buch behandelt.

1.2.2. Entwickler

Wenn es um ein Softwareentwicklungs-Paradigma geht, ist es nahezu selbstverständlich, dass die Rolle des Softwareentwicklers eine zentrale Bedeutung hat. MDSD impliziert zum Teil schärfere und klarere Sichtweisen auf Dinge wie die Bedeutung von Modellen, Trennung von fachlichem und technischem Code, das Verhältnis zwischen Design und Implementierung, Roundtrip-Problematiken, Architektur und Generierung, Framework-Entwicklung, Versionierung und Tests. MDSD richtig eingesetzt erleichtert die Arbeit des Softwareentwicklers deutlich, vermeidet redundanten Code und steigert die Softwarequalität durch formalisierte Strukturen.

1.2.3. Entscheider/Projektleiter

Die Entscheidung, modellgetrieben zu arbeiten, fußt nicht zuletzt auf betriebswirtschaftlichen Fragestellungen wie dem Kosten-Nutzen-Verhältnis oder der Gewinnschwelle. There is no free lunch; auch modellgetriebene Entwicklung hat ihre Kosten. Unter vielen Umständen rechnet sich ein modellgetriebener Ansatz; unter einigen Umständen ist eher davon abzuraten. Auch wenn dieses Buch einen technischen Schwerpunkt hat, so betrachten wir durchaus auch organisatorische und

ökonomische Aspekte, die aus Sicht eines Projektes oder des ganzen Unternehmens relevant sind.

Des Weiteren hat ein modellgetriebenes Vorgehen Auswirkungen auf Projektorganisation, Teamstruktur und den Softwareentwicklungsprozess. Auch darauf gehen wir in diesem Buch ausführlich ein.

1.3. Ziele des Buches

Dieses Buch verfolgt primär das Ziel, Sie, den Leser, davon zu überzeugen, dass MDSD ein praktikabler Ansatz ist – und in der Tat in vielen Fällen der traditionellen Entwicklung überlegen ist. Wir wollen Sie motivieren, diesen Ansatz alsbald einzusetzen, denn es handelt sich weder um bloße Visionen noch um graue Theorie. Dazu möchten wir Ihnen alles an die Hand geben, was Sie benötigen. Falls Sie MDSD bereits praktizieren, soll dieses Buch ggf. eine Hilfestellung oder Vertiefung für spezifische Fragen oder Bereiche darstellen.

Im Detail verfolgen wir dementsprechend eine ganze Reihe von „Unterzielen" – unabhängig vom Aufbau des Buches –, die wir hier kurz und knapp erläutern möchten:

Zunächst wollen wir den theoretischen Rahmen, die Grundkonzepte und die Grundbegriffe einführen. Dabei werden wir auch auf andere, verwandte Themen und Ansätze eingehen, wie z.B. die Model Driven Architecture (MDA) der OMG.

Des Weiteren wollen wir konkrete Hilfestellung für spezifische, MDSD-relevante Themen geben. Dazu gehört Metamodellierung (mit UML und MOF[2]), Codegenerierung, Versionierung, Testen sowie die Unterstützung bei der Toolauswahl auf technischer Seite. Sehr wichtig sind uns aber auch organisatorische und prozessbezogene Hilfestellungen. Zusätzlich wollen wir Argumentationen für MDSD aus ökonomischer Sicht liefern.

Auch wenn man ohne einige theoretische Grundlagen nicht auskommen kann, so soll das Buch vor allem praktische Hilfestellungen geben bzw. praxisrelevante Teilmengen der angesprochenen Themen näher beleuchten. Best Practices sowie die Weitergabe von konkreten Erfahrungen (und teils auch persönlichen Meinungen) sind uns wichtig. Im Zentrum der praktischen Sicht steht die Fallstudie im ersten Teil des

[2] MOF = Meta Object Facility, Standard der OMG

Buches. Mehrere sekundäre Fallbeispiele aus verschiedenen Domänen begleiten die vertiefenden Buchteile.

Wir wollen auf aktuelle Fragestellungen Antworten geben sowie aktuelle Diskussionen aufgreifen. Ein Ausblick auf Trends und Visionen im Kontext von MDSD sollen das Buch abrunden.

1.4. Abgrenzung

Dies ist kein MDA-Buch. Wir beschreiben zwar die Grundlagen und die Terminologie dieses OMG-Standards sowie die zugrunde liegende Vision, auch geben wir in Kapitel <Ref MDA> im Anhang einen Abriss über den Stand der Standardisierung, aber wir vertreten in diesem Buch eben durchaus unsere eigenen Ansichten und Erfahrungen. Sekundär-Literatur (neben der Spezifikation) zum Thema MDA bietet z.B. [Fra02].

Dieses Buch hat nicht zum Ziel, einen geschlossenen, schwergewichtigen MDSD-Entwicklungsprozess zu definieren. Stattdessen beschränken wir uns auf die Vermittlung von Best Practices, welche sich im Zusammenspiel mit agilen Ansätzen wie Crystal [Coc01] und Methodenbausteinen der Produktlinienentwicklung <Ref PLE> zur Konstruktion eines auf den jeweiligen Kontext maßgeschneiderten Entwicklungsprozesses anbieten.

1.5. Struktur des Buches und Readers' Guide

Das Buch beschreibt modellgetriebene Ansätze, wie sie u.a. von den Autoren in der Praxis seit vielen Jahren erfolgreich eingesetzt wurden und werden. Wir beleuchten das Thema im Wesentlichen aus den Perspektiven *Technik*, *Engineering* und *Management*, welche sich in der Aufteilung des Buches wiederfinden:

- **Teil 1** – **Einführung:** Dieser Teil enthält die Einleitung, die Sie gerade lesen, sowie eine Erläuterung der wichtigsten MDSD-Grundideen und der zugehörigen Basisterminologie, die der Model Driven Architecture (MDA) der OMG entnommen ist. Wir widmen uns dann der architekturzentrierten Ausprägung von MDSD, welche ideal als Einstieg in die Praxis geeignet ist. Wir vermitteln die konkreten Techniken anhand einer ausführlichen Fallstudie aus dem Bereich E-Business/Web-Applikationen. Anschließend erfolgt auf der Grundlage der erworbenen Kenntnisse eine umfassendere MDSD-Begriffsbildung. Dieses

Kapitel ist sehr wichtig, zumal der Rest des Buches auf der dort definierten Terminologie aufbaut. Hier findet sich die konzeptionelle, artefaktbezogene Definition von MDSD. Abgerundet wird der erste Teil durch eine Einordnung respektive Abgrenzung verwandter Themen, wie agiler Softwareentwicklung.

- **Teil 2 – Domänenarchitekturen:** Die Domänenarchitektur ist das MDSD-Schlüsselkonzept. Sie beinhaltet unter anderem die Modellierungssprache für die Domäne und die Generierungsvorschriften, welche die Modelle auf eine konkrete Plattform abbilden sollen. Dieser Teil des Buches vermittelt schrittweise Best Practices zur Konstruktion solcher Domänenarchitekturen. Das Kapitel über Metamodellierung bildet das Fundament. Es folgt eine detaillierte Betrachtung der besonderen Rolle der Zielarchitektur im Kontext von MDSD. Das zentrale Kapitel, welches sich mit der Konstruktion von Generierungsvorschriften (MDSD-Transformationen) beschäftigt, schließt sich an. Zum Aufbau von Domänenarchitekturen stützt man sich zweckmäßigerweise auf generische Werkzeuge. Die letzten beiden Kapitel dieses Teils vermitteln diesbezüglich allgemeine technische Grundlagen über Codegenerierung sowie Konstruktions- bzw. Auswahlkriterien für MDSD-Werkzeuge.

- **Teil 3 – Prozesse und Engineering:** In diesem Teil behandeln wir die prozessualen MDSD-Aspekte und Engineering-Themen, die durch MDSD eine spezifische Ausprägung erhalten. Spätestens hier wird deutlich, dass MDSD nicht einfach nur eine Technik ist. Wir stellen eine Reihe von Best Practices auf, die zu einem praktischen und pragmatischen Entwicklungsprozess kombiniert werden können. Anschließend gehen wir auf die Themen Test und Versionierung ein. Es folgt ein Abriss über Productline Engineering, der durch ein reales Projektbeispiel begleitet und durch eine ebenso reale Fallstudie im Anschluss konkretisiert wird. Zum Schluss geht es um den Übergang von objektorientierter Geschäftsprozessanalyse zu modellgetriebener Softwareentwicklung.

- **Teil 4 – Management:** Dieser Teil des Buches wendet sich primär an IT-Manager und Projektleiter und kann weit gehend unabhängig vom Rest des Buches gelesen werden. Wir gehen dabei auf betriebswirtschaftliche und organisatorische Aspekte ein

und behandeln Adaptionsstrategien. Das erste Kapitel dieses Teils enthält einen FAQ[3]-Abschnitt zum Thema MDSD.

- **Anhang:** Der Anhang enthält eine ausführlichere Darstellung des MDA-Standards der OMG sowie Index und Literaturverzeichnis.

Grundsätzlich haben wir uns beim Aufbau des Buches größte Mühe gegeben, dass es beim sequentiellen Lesen – trotz der inhaltlich unvermeidlichen zyklischen Abhängigkeiten – eine optimale Didaktik entfaltet. Da wir jedoch zum Teil stark unterschiedliche Zielgruppen ansprechen, wird sicherlich bei dem einen oder anderen Leser der Wunsch bestehen, (zunächst) selektiv zu lesen. Dazu ein paar Hinweise:

Der primär technisch interessierte Leser, der bereits über MDA-Kenntnisse verfügt, kann ggf. direkt mit der Fallstudie in Teil 1 beginnen, mit Kapitel <Ref Begr> (ebenfalls in Teil 1) fortfahren und sogleich die technischen Fragestellungen durch die Lektüre ausgewählter Kapitel von Teil 2 oder 3 vertiefen, bevor er sich dem Rest des Buches widmet.

Falls Sie sich fragen, welche ökonomischen Vorteile MDSD eigentlich konkret bietet, bevor Sie sich im Detail mit der Frage auseinander setzen wollen, was MDSD ist, können Sie das Kapitel <Ref Oeko> vorziehen. Als Verständnisgrundlage empfehlen wir ggf. das Kapitel <Ref Basis>.

1.6. Über die Autoren

Thomas Stahl ist Diplom-Informatiker und arbeitet seit 1994 bei der b+m Informatik AG in Melsdorf bei Kiel. Als Chefarchitekt ist er verantwortlich für Architekturmanagement, Framework-Entwicklung und Software Engineering. Außerdem ist er als Berater tätig. Sein inhaltlicher Arbeitsschwerpunkt ist derzeit die modellgetriebene Softwareentwicklung. Neben der Projektarbeit verfasst er Artikel für Fachzeitschriften und spricht auf IT-Konferenzen. Er ist einer der „geistigen Väter“ des Open-Source Generators *openArchitectureWare*. Seine Freizeit widmet er unter anderem der Musik. Sie erreichen ihn unter t.stahl@bmiag.de.

[3] FAQ = Frequently Asked Questions

Markus Völter arbeitet als freiberuflicher Berater zum Thema Softwaretechnologie und -engineering. Seine Schwerpunkte liegen dabei auf Softwarearchitektur, Middleware und Modellgetriebener Entwicklung. Er hat langjährige Erfahrung in den verschiedensten Domänen, darunter Banken, E-Business, Gesundheitswesen, Astronomie, Telematik und Automotive Embedded. Neben seiner Tätigkeit in Projekten ist Markus regelmäßig als Sprecher auf den einschlägigen nationalen und internationalen Konferenzen anzutreffen. Er ist außerdem Autor mehrerer Bücher, Patterns und Artikel zu Themen, die sich aus seiner Tätigkeit als Berater ergeben. In seiner Freizeit ist er vorzugsweise mit seinem Segelflugzeug am Himmel über Süddeutschland anzutreffen. Sie erreichen ihn unter voelter@acm.org oder www.voelter.de.

1.7. Danksagungen

Ein Buch, wie Sie es hier vor sich haben, ist mit viel mehr Aufwand verbunden, als man es sich vorstellen mag. Ohne die Unterstützung einer ganzen Reihe von Leuten wäre die Fertigstellung noch deutlich schwieriger gewesen. Deshalb ist es an der Zeit, dass wir unseren Dank aussprechen.

Da wären zunächst unsere Gastautoren Jorn Bettin, Wolfgang Lorenz und Tim Weilkiens. Jorn hat vor allem zum Management-Teil des Buches sehr viel beigesteuert, uns bei der Begriffsbildung unterstützt und wichtige Hinweise zur Struktur des Buches gegeben. Wolfgang und Tim haben das Kapitel „Objektorientierte Geschäftsprozessmodellierung und MDSD“ beigesteuert.

Wir bedanken uns bei Bernd Oestereich, Peter Roßbach und Wolfgang Neuhaus für fruchtbare Diskussionen und Material, welches Eingang in die Fallstudie im ersten Teil des Buches gefunden hat.

Weiterhin möchten wir uns sehr bei unseren Reviewern bedanken, die nützliches Feedback zu vielen Details, aber vor allem auch zur Struktur des Buches gegeben haben. Die Reviewer waren (in alphabetischer Reihenfolge): Wolfgang Görigk, Michael Kircher, Michael Kunz, Andreas Mohnberg, Wolfgang Neuhaus, Jürgen Rühle, Martin Schepe, Klaus-Dieter Schmatz, Eberhard Wolff und Ghica van Emde Boas.

Last – but not least – möchten wir auch dem dpunkt.verlag danken, insbesondere René Schönfeldt, unserem Lektor. Er hat nicht nur Verständnis für unsere mehrfachen Terminverschiebungen aufgebracht,

sondern ist uns vor allem gegen Ende mit der einen oder anderen „unüblichen“ Vorgehensweise entgegengekommen.

Markus möchte sich bei dieser Gelegenheit auch noch einmal explizit bei Tom Stahl bedanken, dass er – vor allem im letzten Drittel der Schreibphase – bei diesem Buchprojekt die Koordination übernommen und viel Arbeit in Details investiert hat.
Thomas bedankt sich bei der b+m Informatik AG für die sehr kulante Schaffung des nötigen Freiraums für dieses Projekt. Sein größter Dank gilt jedoch seiner Frau Anja, welche ihn trotz schwieriger Phasen kontinuierlich unterstützt hat.

2. MDSD – Grundideen und Basisterminologie

Dieses Kapitel dient dazu, die Motivation und die wichtigsten Grundkonzepte modellgetriebener Softwareentwicklung aufzuzeigen. Wir bevorzugen die Abkürzung MDSD (Model-Driven Software Development) gegenüber der etwas ungenaueren Variante MDD (Model-Driven Development). Erstere hat sich über die letzten ein bis zwei Jahre in der entsprechenden Community etabliert.

Die Model Driven Architecture (MDA) der Object Management Group (OMG) ist eine Ausprägung respektive eine Standardisierungsinitiative zu diesem Thema. Unser Fokus liegt auf der praktischen Umsetzbarkeit in Softwareprojekten. An vielen Stellen decken sich unsere Konzepte und Erfahrungen mit der MDA-Vision der OMG – an einigen nicht. Wir werden diese Stellen benennen und diskutieren. Unabhängig davon ist die MDA-Terminologie (eben weil sie standardisiert und mittlerweile recht verbreitet ist) sehr nützlich, um einen Einstieg in die Materie zu ermöglichen – und genau in diesem Sinne ist der zweite Teil des Kapitels zu verstehen: Die MDA liefert die Basisterminologie für MDSD. Der dritte Teil des Kapitels führt dann die noch fehlenden Konzepte ein, die zum Verständnis der ausführlichen Fallstudie notwendig sind.

2.1. Herausforderung

Im 21. Jahrhundert ist Software überall. Die Softwareindustrie hat sich zu einer der größten Industrien auf dem Planeten entwickelt, und viele der erfolgreichsten Unternehmen heutzutage sind Softwareproduzenten beziehungsweise bieten Leistungen im Softwareumfeld an.

Heute ist Software ein kritischer Bestandteil im „Maschinenraum“ aller technologiebasierter und vieler servicebasierter Unternehmen. Hohe Softwareentwicklungskosten haben eine erhebliche ökonomische

Auswirkung, und schlechtes Softwaredesign, welches die Produktivität von Anwendern beeinträchtigt, hat womöglich eine noch größere Auswirkung.

Viele Business-Softwarehersteller sind so sehr mit den sich ständig verändernden Implementierungstechnologien beschäftigt, dass Produktivitätsüberlegungen und Risikomanagement auf der Strecke bleiben. Da hilft weder Offshoring noch die neueste Generation von Infrastruktursoftware wie integrierte Entwicklungsumgebungen, EAI-Tools, BPM[4]-Tools, Middleware etc. Produktivitätsprobleme sind meist das Resultat von mangelnder Konsistenz oder Offenheit der Anwendungsarchitektur, unzureichendem Management von Abhängigkeiten zwischen den verschiedenen Systemkomponenten und untauglichen Softwareentwicklungsprozessen.

2.1.1. Historischer Rückblick

Die 90er Jahre wurden hauptsächlich von zwei Softwareentwicklungsparadigmen beeinflusst: am Anfang der 90er waren es Computer Aided Software Engineering (CASE) und Sprachen der „vierten" Generation (4GLs), und in der zweiten Hälfte des Jahrzehnts hielt die Objektorientierung Einzug in den Mainstream.

CASE-Methoden und entsprechende Werkzeuge waren teuer, und die proprietären Ansätze kollidierten mit dem wachsenden Bewusstsein für offene Standards. Viele Unternehmen machten schlechte Erfahrungen mit unterschiedlichen Herstellern – letztendlich wanderten nicht nur die Werkzeuge, sondern auch der Ansatz der modellbasierten Softwareentwicklung in den Papierkorb. Objektorientierung erfüllte auch nicht alle Versprechungen, aber letzten Endes wurde damit der Grundstein für Komponententechnologien gelegt, und objektorientierte Sprachen haben erfolgreich die vorherige Generation von Programmiersprachen abgelöst.

Mit dem Abgang von 4GLs und CASE haben sich Werkzeughersteller auf OO-Modellierungswerkzeuge konzentriert; dies führte zum Unified Modeling Language (UML)-Notationsstandard und zu Werkzeugen die auf einer „Round-Trip"-Philosophie basieren. Diese ermöglicht es, reibungslos zwischen UML-Modellen und dem korrespondierenden Implementierungscode zu wechseln. Oberflächlich betrachtet beeindrucken die UML-Werkzeuge durch die Fähigkeit, Modelle und Code im Einklang zu halten. Allerdings ergibt eine

[4] BPM = Business Process Management

genauere Betrachtung, dass solche Werkzeuge keineswegs unmittelbar die Produktivität erhöhen, sondern bestenfalls eine effiziente Methode bieten, gut aussehende Dokumentation[5] zu erzeugen, bzw. für das Verständnis großer Mengen bereits vorhandenen Codes hilfreich sein können.

2.1.2. Aktuelle Bilanz

Die Grenzen zwischen modernen UML-Werkzeugen und integrierten Entwicklungsumgebungen (IDEs) verschwinden mehr und mehr. UML-Werkzeuge beinhalten teilweise komfortable Code-Editoren und Compiler, und traditionelle IDEs werden mit UML-Modellierungskomponenten ausgestattet.

Softwareentwicklungswerkzeuge stellen zunehmend intelligente Wizards zur Verfügung, die den Benutzer bei der Anwendung von Entwurfsmustern, beim Erstellen von Benutzeroberflächen und durch die Generierung von Codeskeletten für die Benutzung von populären Frameworks unterstützen.

Obwohl dieser Ansatz eine Verbesserung gegenüber älteren UML-Werkzeugen darstellt, welche nur in der Lage waren, leere Klassenskelette zu generieren, sind starke Ähnlichkeiten mit der Inflexibilität von CASE-Werkzeugen festzustellen. Wenn sich zum Beispiel ein Designmuster ändert, sind die heutigen UML-Werkzeuge nicht in der Lage, die Auswirkungen automatisch und vor allem iterativ auf den gesamten Quellcode eines Anwendungssystems zu übertragen, ohne dass die Abstraktion verloren geht.

Die Schwächen von Mainstream-IDEs und UML-Werkzeugen führte letztlich zur MDA-Initiative der OMG. Entsprechende Werkzeuge ermöglichen es den Benutzern, genau festzulegen, wie UML-Modelle auf unternehmensspezifische Implementierungstechnologie-Kombinationen abzubilden sind. In diesem Kontext wittern allerdings leider auch einige traditionelle CASE-Tool-Hersteller eine zweite Chance, ihre Werkzeuge in neuer Verpackung als kommerzielle MDA-Produkte anzubieten. In diesem Fall sind die Werkzeuge nicht auf individuelle Gegebenheiten oder Anforderungen des Kunden anpassbar, sondern vertreten immer noch den „One-Size-Fits-All“-Ansatz. Die meisten der auf den OMG-Webseiten gelisteten Werkzeuge verdienen jedoch durchaus die Bezeichnung „MDA-Tool“.

---

[5] Sie bieten nur eine andere, grafische Sicht auf den Code, aber keine echte Abstraktion.

Parallel zu dem Fortschritt bei Softwareentwicklungswerkzeugen hat allerdings auch eine signifikante Verschiebung bei den verwendeten Softwareentwicklungsmethoden stattgefunden. Dieses Feld adressiert die MDA bisher so gut wie nicht.

Die rasche Verbreitung agiler Ansätze zeigt, dass traditionelle Methoden, die in aller Regel ein hohes Maß an manuell erstellten Fließtextdokumenten erfordern, auf zunehmenden Widerstand stoßen. Es wird jetzt offen zugegeben, dass die Produktion großer Mengen Dokumentation zwar durch die traditionellen Methoden gefordert wurde, aber in der Praxis nicht mit dem Marktdruck hin zu niedrigeren Softwareentwicklungskosten vereinbar war. Allerdings bieten agile Methoden wie z.B. Extreme Programming (XP, vgl. [Bec00]) alleine nicht genügend Hilfsmittel zur Erstellung qualitativ hochwertiger Software, und sie skalieren nicht ohne weiteres für größere Projekte. Problematisch ist insbesondere der gelegentlich anzutreffende Irrglaube, dass sie fehlende analytische Fähigkeiten oder mangelnde Softwaredesignerfahrung innerhalb eines Entwicklerteams kompensieren könnten.

2.2. MDSD-Ziele

Bevor wir uns den Konzepten und der Terminologie zuwenden, wollen wir kurz etwas über die Ziele von MDSD sagen, wobei wir die Mittel zur Erreichung derselben zunächst natürlich nur andeuten können:

- Mit MDSD kann eine Steigerung der *Entwicklungsgeschwindigkeit* erreicht werden. Das Mittel dazu heißt Automation: Aus formalen Modellen kann durch einen oder mehrere aufeinander folgende Transformationsschritte letztendlich lauffähiger Code erzeugt werden.

Durch den Einsatz automatisierter Transformationen und formal definierter Modellierungssprachen kann darüber hinaus die *Softwarequalität* gesteigert werden, zumal sich eine einmal definierte Softwarearchitektur gleichförmig in der Implementierung wiederfindet.

Querschnittlich verteilte Implementierungsaspekte können an einer Stelle – in den Transformationsvorschriften – verändert werden. Dasselbe gilt für die Fehlerbeseitigung in generierten Codeanteilen. Diese so genannte *Trennung von Verantwortlichkeiten* (Separation of Concerns [Lad03]) verspricht nicht zuletzt auch durch *Redundanzvermeidung* eine

verbesserte *Wartbarkeit* von Softwaresystemen sowie die *Handhabbarkeit von Technologiewandel*

- Einmal definierte Architekturen, Modellierungssprachen und Transformationen können im Sinne einer Software-Produktionsstraße zur Herstellung diverser Softwaresysteme verwendet werden. Dies führt zu einem *höheren Grad der Wiederverwendung* und macht *Expertenwissen* in Softwareform in der Breite verfügbar.

- Ein weiteres wesentliches Potenzial ist die bessere *Handhabbarkeit von Komplexität durch Abstraktion.* Mit den Modellierungssprachen soll „Programmierung" oder Konfiguration auf einer abstrakteren Ebene möglich werden. Die Modelle müssen dazu in einer möglichst problemorientierten Modellierungssprache ausgedrückt werden.

- MDSD bietet durch Prozessbausteine und Best Practices in den Bereichen Technik, Engineering und Management ein *produktives Umfeld*, welches die Erreichung der bisher genannten Ziele unterstützt.

- Zum Schluss noch zwei Ziele, die erst durch eine Standardisierung, wie sie die OMG mit der MDA anstrebt, wirklich gelingen können: Gemäß der Ausrichtung der OMG lauten die beiden primären Motivationen für MDA *Interoperabilität* (Herstellerunabhängigkeit durch Standardisierung) und *Portabilität* (Plattformunabhängigkeit) von Softwaresystemen – dieselbe Motivation also, die bereits zur Definition von CORBA[6] geführt hat. Dazu postuliert die OMG, die Spezifikation von Systemfunktionalität von der Spezifikation der Implementierung dieser Funktionalität auf einer gegebenen Plattform zu trennen. Die MDA hat den Zweck, Richtlinien und Standards zur Verfügung zu stellen, die auf eine entsprechende Strukturierung von Systemspezifikationen in Form von Modellen führen.

Die meisten der hier vorgestellten Ziele sind natürlich nichts Neues. Im Gegenteil, sie stellen so etwas wie den „heiligen Gral" der IT-Branche dar und heilsbringerischen Versprechungen glaubt mittlerweile zu Recht kaum noch jemand. Doch betrachtet man die Historie der IT bzw. der

[6] CORBA = Common Object Request Broker Architecture (Standard der OMG).

Informatik, so wird deutlich, dass eine kontinuierliche Evolution stattfindet. Hochsprachen, Objektorientierung und Komponentensysteme waren Meilensteine auf dem Weg zur Erreichung dieser Ziele – MDSD ist ein weiterer. Das Paradigma hilft bei allen genannten Punkten, dem Ziel ein kleines oder sogar ein großes Stück näher zu kommen.

2.3. Der MDSD-Ansatz

Jede Software trägt „unter dem Strich", d.h. auf Quelltextebene, Konstruktionsparadigmen in sich – eine innere Struktur. Wie gut und konsequent ausgeprägt diese Struktur ist, beeinflusst in direkter Weise die Aspekte Entwicklungsgeschwindigkeit, Qualität, Performance, Wartbarkeit, Interoperabilität und Portabilität der Software, also extrem wichtige ökonomische Schlüsselfaktoren.

Das Problem ist, dass auf dem Niveau von Programmiersprachen die eigentlichen Konstruktionsparadigmen nur noch schwer zu erkennen sind, weil das Abstraktionsniveau von Programmiersprachen viel niedriger liegt. Anders ausgedrückt: Das kostbare Gut der inneren Struktur liegt letztlich in verschleierter, verteilter und natürlich auch stark individualisierter Form vor, im System selber ist es nicht mehr direkt repräsentiert. Die Qualität schwankt mit den Fertigkeiten und Interpretationen der Entwickler.

Nun ist die Idee der Modellierung natürlich nicht neu und gerade die schwergewichtigen Entwicklungsprozesse setzen auf sie, um die innere Struktur einer Software zu *dokumentieren*. Den zwangsläufigen Konsistenzproblemen wird versucht, mit aufwändigen Reviews zu begegnen. In der Praxis unter Zeitdruck sind diese Reviews und auch die Modelle stets das erste Opfer – und das zumindest aus pragmatischer Sicht zu Recht. Ein anderer Ansatz ist wie gesagt das so genannte Roundtrip- bzw. Reverse Engineering, wie es von den meisten UML-Tools angeboten wird: Es handelt sich lediglich um Sourcecode-Visualisierung in der Syntax von UML, d.h. das Abstraktionsniveau dieser Modelle ist das gleiche, wie das des Sourcecodes selbst[7]. Die Übersichtlichkeit mag etwas besser sein, doch das Grundproblem bleibt.

---

[7] Inzwischen beherrschen UML-Tools zwar auch das J2EE-Programmiermodell und können so ein EJB-Bean durch eine UML-Klasse repräsentieren, doch weiter kann die Abstraktion nicht gelingen, weil die Konzepte der Anwendungsarchitektur dem Tool schlicht nicht

Die modellgetriebene Softwareentwicklung bietet einen deutlich effektiveren Ansatz: Modelle sind abstrakt *und zugleich formal*. Abstraktheit ist dabei keineswegs gleichzusetzen mit Vagheit, sondern vielmehr mit Kompaktheit und Reduktion auf das Wesentliche. MDSD-Modelle erhalten die exakte Bedeutung von Programmcode, in dem Sinne, dass aus ihnen z.B. ein Großteil der letztendlichen Implementierung (und nicht nur Klassen- und Methodenrümpfe) generiert werden kann. Modelle sind dann nicht mehr nur Dokumentation, sondern *Bestandteil der Software* und stellen einen ganz entscheidenden Beschleunigungs- und Qualitätsfaktor für die Softwareentwicklung dar. Mit der Betonung auf „modell*getrieben*“ im Gegensatz zu „modell*basiert*“ versuchen wir verbal diesem Unterschied Rechnung zu tragen.

Die Modelle orientieren sich bezüglich der verwendeten Ausdrucksmittel am *Problemraum* der jeweiligen Domäne, wodurch die Abstraktion vom Niveau der Programmiersprachen und eine entsprechende Kompaktheit überhaupt erst möglich werden. Dieses Prinzip ist allen modellgetriebenen Ansätzen gemein, egal ob die Domäne nun „Softwarearchitektur“, „Finanzdienstleistungssysteme“, „Versicherungen“ oder „Embedded-Systeme“ lautet. Zur Formalisierung der Modelle wird also offenbar eine höhere, domänenspezifische Modellierungssprache (DSL = Domain-Specific Language) benötigt, und es ist zumindest aus dieser „Vogelperspektive“ unerheblich, ob es sich dabei um eine UML-basierte Sprache handelt oder nicht.

Den zweiten Grundpfeiler neben formalen, abstrakten Modellen bilden „semantisch reiche“, domänenspezifische Plattformen: Vorgefertigte, wiederverwendbare Komponenten und Frameworks bieten eine ungleich mächtigere Basis als eine „nackte“ Programmiersprache oder eine technische Plattform wie J2EE. Das bedeutet insbesondere, dass der Generator, der das formale Modell transformieren soll, deutlich einfacher wird, wenn sich das Generat auf wesentlich höherwertigere APIs[8] stützen kann. Die Einführung wiederverwendbarer Frameworks, Basisklassen, Komponenten etc. zur Vermeidung von Coderedundanz ist keine neue Idee, aber im Kontext der modellgetriebenen Softwareentwicklung dienen sie überdies dazu, den Modelltransformationen in Gestalt einer wohldefinierten Plattform

---

bekannt sind und es zudem keine eindeutige Abbildung auf den Sourcecode gibt.

[8] API = Application Programming Interface

sozusagen „auf halbem Wege entgegenzukommen“. Dies bewirkt eine deutliche Komplexitätsreduktion der Generatoren.

Abb. 1 zeigt zur Verdeutlichung die Zusammenhänge bei der Anwendungsentwicklung mittels MDSD.

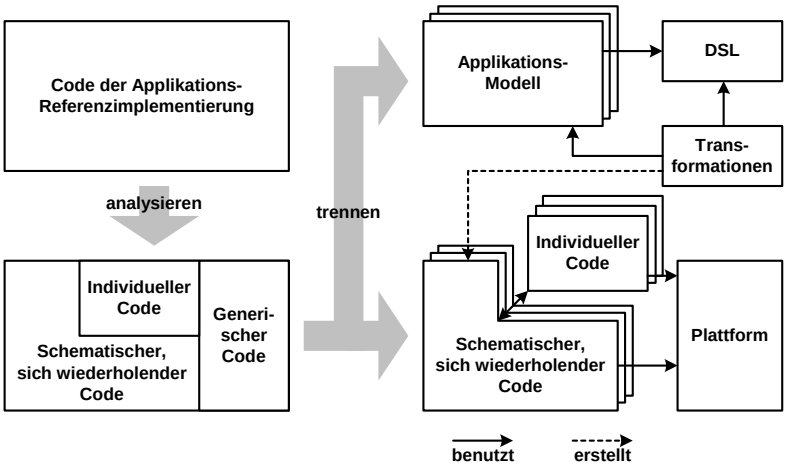


Abb. 1 *Grundidee modellgetriebener Softwareentwicklung*

Betrachten wir eine existierende Anwendung oder eine Referenzimplementierung. (oben links). Es handelt sich um Unikate mit individueller Struktur. Den Code jeder dieser (Teil-)Anwendungen können wir gedanklich so umstrukturieren, dass drei Teile separierbar sind[9] (unten links): Einen generischen Anteil, der für alle zukünftigen (Teil-)Anwendungen identisch ist, einen schematischen Anteil, der für alle Anwendungen zwar nicht identisch, aber von gleicher Systematik ist (d.h. zum Beispiel auf den gleichen Entwurfsmustern basiert), und zu guter Letzt einen anwendungsspezifischen Anteil, welcher nicht verallgemeinerbar ist. Über die Gewichtung der Anteile sagen wir zunächst nichts aus (im Extremfall kann der anwendungsspezifische Anteil auch leer sein). Die modellgetriebene Softwareentwicklung hat zum Ziel, den schematischen Anteil für alle (Teil-)Anwendungen jeweils aus einem Anwendungsmodell abzuleiten. Es kann bei der Transformation auch Zwischenstufen geben, jedoch sind DSL, Transformation und Plattform in jedem Falle die Schlüsselelemente. Sie müssen für die jeweilige Domäne (z.B. „Enterprise Softwarearchitektur“ oder „Bestandssysteme für Versicherungen“) nur einmal erstellt werden (unten rechts).

[9] ggf. durch Refactoring

2.4. Basisterminologie

In diesem Abschnitt werden wir die wichtigsten Konzepte bzw. Begriffe des MDA-Standards darstellen, um damit wie gesagt die MDSD-Basisterminologie aufzubauen.

Fachliche Spezifikationen werden in pattformunabhängigen Modellen (*PIM = Platform Independent Model*) definiert. Dazu wird eine formale Modellierungssprache verwendet, die möglichst gut an die zu modellierende Fachlichkeit (Domäne) angepasst ist. In den meisten Fällen wird man – schon des Toolsupports wegen – eine mittels Profilen an die Domäne angepasste UML verwenden (vgl. <Ref MM::Prof>). Die damit spezifizierte Fachlichkeit ist vollständig unabhängig von der Implementierung auf der späteren Zielplattform. Solche Zielplattformen können zum Beispiel CORBA, J2EE, .Net oder proprietäre Frameworks/Plattformen sein. Abb. 2 zeigt dieses Grundprinzip.

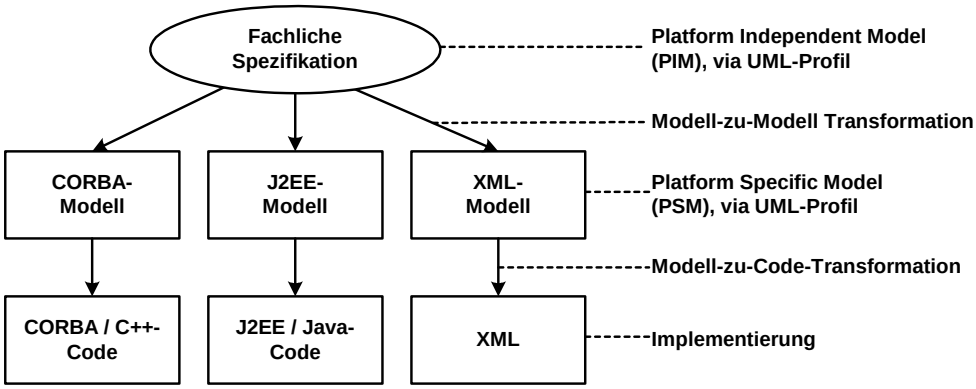


Abb. 2 *MDA-Grundprinzip*

Durch eine in der Regel mit Werkzeugen automatisierte Modell-Transformation werden aus den plattformunabhängigen fachlichen Spezifikationen plattformabhängige Modelle (*PSM = Platform Specific Model*) gewonnen. Diese plattformabhängigen Modelle enthalten die spezifischen Konzepte der Zielplattform. Mit einer weiteren werkzeugunterstützten Transformation auf der Basis eines oder mehrerer PSM wird dann für eine konkrete Zielplattform die Implementierung erzeugt (siehe Abb. 3).

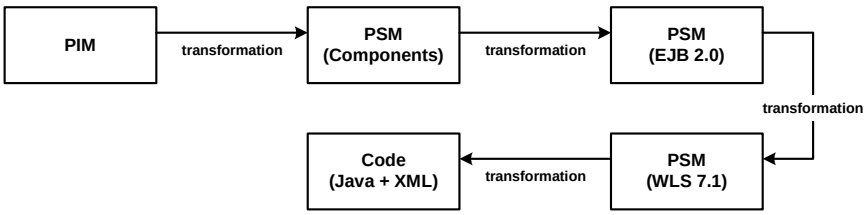


Abb. 3 *PIM, PSM und Transformationen*

Zu beachten ist hierbei noch, dass ein PIM und PSM relative Konzepte sind – relativ zur Plattform. Im obigen Beispiel ist EJB 2.0 PSM spezifisch für die EJB 2.0-Plattform, aber noch unabhängig bzgl. der konkreten, Applikationsserver-spezifischen Umsetzung.

Hier ein Beispiel: In Abb. 4 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.

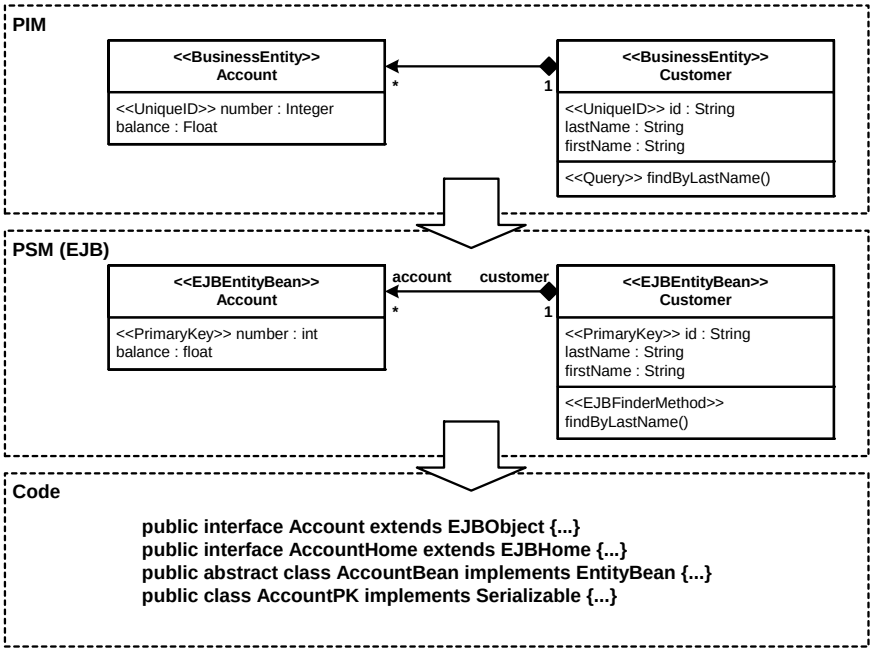


Abb. 4 *Beispiel*

Durch das Notieren von Stereotypen an UML-Modellelementen kann die Bedeutung eines Elements geändert bzw. spezialisiert werden. Eine

Klasse mit dem Stereotyp *<<BusinessEntity>>* ist eben nicht einfach nur eine Klasse, sondern eine eigenständige Entität in Geschäftsanwendungen. Was dies nun *konkret* bedeutet, wird durch Transformationen festgelegt, die definieren, auf welche Art ein Stereotyp wie z.B. *<<BusinessEntity>>* auf eine gegebenen Plattform wie z.B. J2EE abgebildet wird.

Eine solche Erweiterung des Standardsprachumfangs der UML durch Stereotypen wird als (UML-)Profil bezeichnet und ist ein Standardmechanismus, der zum Zweck der Offenheit der UML von der OMG spezifiziert wurde. Hier wird er genutzt, um eine formale Modellierungssprache zu definieren. Erst diese Formalisierung 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 Stereotypen *<<EJBEntityBean>>*, *<<PrimaryKeyField>>* und *<<EJBFinderMethod>>*. Auch dies sind zunächst wieder Konzepte, die erst durch Transformationen (z.B. in die Programmiersprache Java) ihre Bedeutung erhalten.

Durch die Transformation wird aus dem PSM letztlich also Sourcecode, in dem sich die aufgeführten Konzepte dann in ihrer konkreten Ausprägung wiederfinden.

2.4.1. 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[10]. Im Prinzip sieht die MDA auch klassische Programmiersprachen formal als MDA-Modellierungssprachen an, die wiederum in Relation zu einer Plattform stehen. Dies ist zweifelsohne richtig, jedoch sind wir der Ansicht, dass diese Betrachtungsweise der Erläuterung von Konzepten zuweilen im Weg steht, so dass wir die Begriffe „Modell“ und „Modellierungssprache“ im Folgenden klar von den Begriffen „Programm“ und „Programmiersprache“ separieren werden.

UML-Diagramme sind jedoch nicht per se MDA-Modelle. Der wichtigste Unterschied zwischen allgemeinen UML-Diagrammen (z.B.

[10] Laut Standard mit MOF-basierten Modellen – vgl. <Ref MM>

Analyse-Modellen) und MDA-Modellen liegt darin, dass die Bedeutung (Semantik) von MDA-Modellen formal definiert ist. Dies wird durch die Verwendung einer entsprechenden Modellierungssprache sichergestellt, die typischerweise durch ein UML-Profil und daran angeknüpfte Transformationsvorschriften gegeben ist. Auf diese Mechanismen gehen wir später in diesem Kapitel noch ausführlicher ein. Letztlich bedeutet es, dass die Abbildung eines Modelles auf eine gegebene Plattform eindeutig definiert ist.

Plattform

MDA sagt (zunächst) nichts über den Abstraktionsgrad von Plattformen aus. Plattformen können aufeinander aufbauen. Beispielsweise ist Intel PC eine Plattform für Linux. CORBA, J2EE oder WebServices sind mögliche Plattformen für ein E-Business-System und C++ ist eine mögliche Plattform für CORBA. Auch eine wohldefinierte Anwendungsarchitektur mit dazugehörigem Laufzeitsystem kann eine Plattform für Applikationen sein. Gerade Letzteres sehen wir als eines der Schlüsselkonzepte für modellgetriebene Softwareentwicklung an und wir werden später ausführlich darauf eingehen.

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 (siehe Abb. 5)

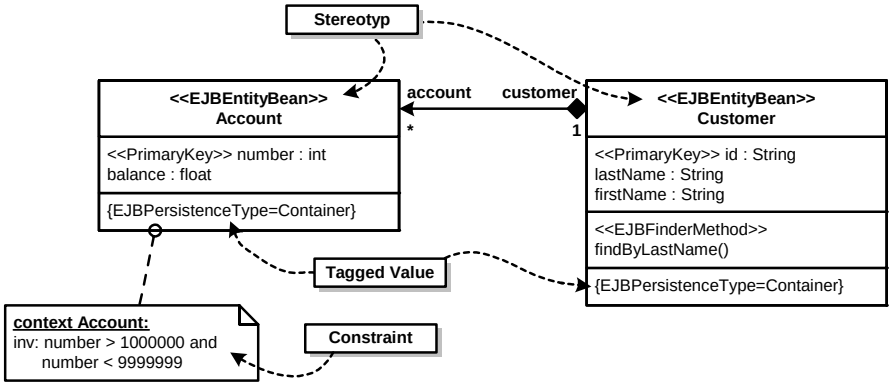


Abb. 5 *Verwendung eines UML-Profils*

Definiert wird ein UML-Profil als Erweiterung des UML-Metamodells. Ein Metamodell definiert u.a. die Basiskonstrukte, die in einem

konkreten Modell vorkommen dürfen. Konzeptionell ist ein Modell eine „Instanz“ eines Metamodelles. Das UML-Metamodell enthält dementsprechend Elemente wie *Class*, *Operation*, *Attribute* oder *Association*. Das Konzept des Metamodelles ist eines der wichtigsten im Kontext modellgetriebener Softwareentwicklung – aus diesem Grund haben wir ihm ein eigenes Kapitel gewidmet (siehe <Ref MM>). An dieser Stelle soll uns aber zunächst ein intuitives Verständnis genügen. Der Zusammenhang zwischen Metamodell und Profil wird in Abb. 6 anhand eines vereinfachten Beispiels – ein UML-Profil für Enterprise Java Beans (EJB) – verdeutlicht.

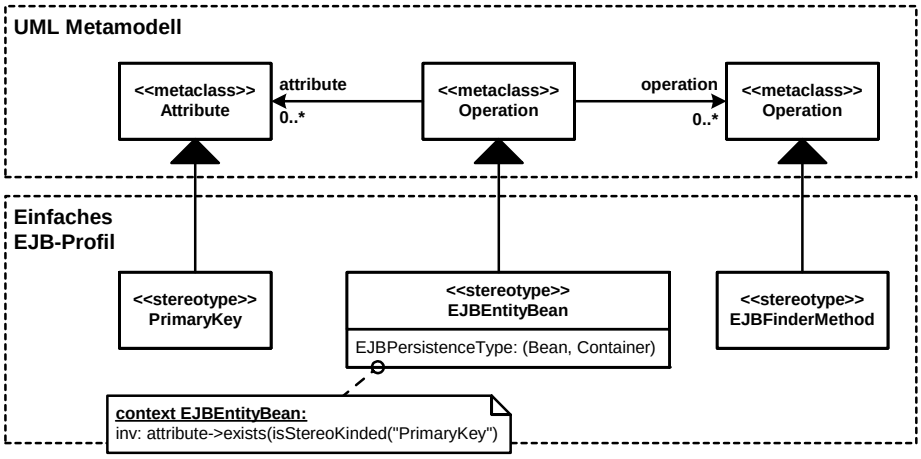


Abb. 6 *UML Metamodell und UML-Profil für EJB (Ausschnitt)*

Die Standard-UML-Konzepte *Attribute*, *Class* und *Operation* werden im UML-Profil um die spezifischen Konzepte *PrimaryKeyField*, *EJBEntityBean* und *EJBFinderMethod* erweitert.

Dazu kommt ein neues UML 2.0-Sprachkonstrukt (*Extension*) zum Einsatz. Es wird durch den ausgefüllten Vererbungspfeil dargestellt (wir haben ihn zur Vermeidung von Verwechslungen extra etwas größer gezeichnet).

Zusätzliche Erweiterungen werden durch Tagged Values und Modellierungsrichtlinien in Form von Constraints definiert. Ein Constraint wird üblicherweise als Kommentar zu dem betreffenden Modellelement notiert (wir verwenden hier die formale Constraint-Sprache OCL). Tagged Values werden als Attribute des Stereotyps notiert.

Ein UML-Profil bietet also eine konkrete Notation für die Bezugnahme von Modellen auf ein Metamodell und bestimmt, ob ein konkretes Modell „wohlgeformt“ (d.h. gültig) ist oder nicht. Kurz, es definiert eine formale Modellierungssprache als Erweiterung der UML.

Die genauen Zusammenhänge werden später im Kapitel <Ref MM> erläutert.

PIM und PSM

Die Trennung von Platform Independent Model (PIM) und Platform Specific Model (PSM) ist ein Schlüsselkonzept der 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 (siehe Abb. 7). Der „Rückweg" – die Gewinnung eines PIMs aus einem PSM – ist kaum automatisierbar. Dieser Weg erfordert in aller Regel manuelle, intellektuelle Arbeit, welche in der MDA-Spezifikation – etwas unglücklich – als Refactoring bezeichnet wird (Refactoring hat ja eher die Bedeutung von Äquivalenzumformungen, siehe [Fow99]).

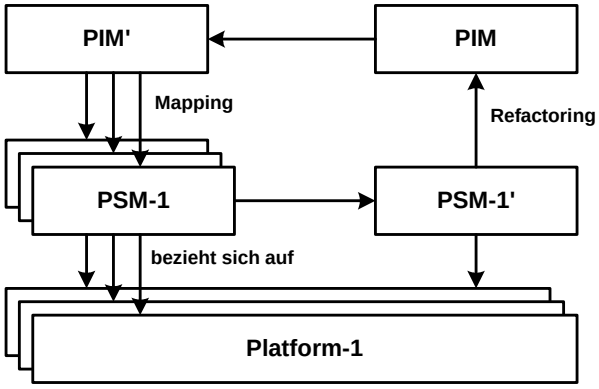


Abb. 7 *Zusammenhang PIM, PSM, Plattform*

Transformationen

Transformationen bilden Modelle auf die jeweils nächste Ebene – weitere Modelle oder Sourcecode – ab. Im Sinne der 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 Request für Proposal dazu vor („MOF 2.0 Query/Views/Transformations RFP", OMG Document ad/2002-04-10). Die Vorstellung besteht darin, Transformationen auf

Basis der Metamodelle von Quelle und Ziel zu beschreiben. Kapitel <Ref MDA> enthält dazu ausführlichere Informationen.

Die meisten derzeit verfügbaren MDA-/MDSD-Tools definieren ihre Transformationsvorschriften nicht zwischen zwei Metamodellen, sondern verwenden z.B. Templates zur direkten Generierung von Sourcecode (ohne dass das Metamodell der Programmiersprache dem Generator bekannt wäre). Es gibt aber durchaus Generatoren, welche diese Transformationsvorschriften am UML-Profil respektive dem dazugehörigen Metamodell festmachen. Solche Ansätze sind absolut praxistauglich (vgl. <Ref CSI>, <Ref Code>). Der Vorteil einer Transformation auf der Basis zweier Metamodelle (Quelle und Ziel) liegt hauptsächlich darin, elegant ein Modell in ein anderes überführen zu können. Ob das Paradigma zur Erzeugung von Sourcecode allerdings überhaupt praxistauglich ist, halten wir für fraglich.

Heutige Generatoren lösen dieses Problem wie gesagt auf andere Weise, indem sie proprietäre Transformationssprachen verwenden. Hier kommen beispielsweise JPython, TCL, JSP, XSLT oder zweckoptimierte Skriptsprachen zum Einsatz[11]. Die mit diesen Sprachen definierten Generator-Templates funktionieren nach dem Prinzip von Makros und verwenden die Modelle als Eingabeparameter. Derzeit ist demzufolge (noch) keine Interoperabilität für Modell-Transformationen gegeben. Hier bleibt die Standardisierung abzuwarten.

Eine weiterführende Betrachtung des MDA-Standards liefert Kapitel <Ref MDA>.

2.5. Architekturzentrierte MDSD

In diesem Abschnitt wollen wir die noch fehlenden Grundlagen zum Verständnis der anschließenden Fallstudie vermitteln. Es handelt sich dabei um eine Ausprägung modellgetriebener Softwareentwicklung, die wir *architekturzentriert* nennen. (AC-MDSD = Architecture Centric MDSD). Die beschriebenen Ansätze sind aus einer mittlerweile über sechsjährigen Praxis zahlreicher Projekte entstanden und insbesondere auf praktische Handhabbarkeit ausgelegt.

---

[11] Diese sind selbst domänenspezifische Sprachen; Sprachen für die Domäne der Definition von Codegenerierungs-Templates.

2.5.1. Motivation

Im Gegensatz zu den primären Zielen der OMG bezüglich MDA, nämlich Interoperabilität und Portabilität von Software, geht es bei architekturzentrierter MDSD zunächst einmal um die Steigerung der Entwicklungseffizienz sowie die Verbesserung von Softwarequalität und Wiederverwendbarkeit. Dies bedeutet insbesondere, den Software-Entwickler von lästiger und fehleranfälliger Routinearbeit zu befreien. Entwickler sind heutzutage sehr komplexen Software-Infrastrukturen ausgesetzt: Application-Server, Datenbanken, Open-Source-Frameworks, Protokollen, Schnittstellentechnologien etc., die alle in geeigneter Weise in Zusammenhang gebracht werden müssen, so dass letztlich eine performante, robuste und wartbare Software entsteht. Mit steigender Komplexität gewinnt daher die Disziplin der Softwarearchitektur zunehmend an Bedeutung.

Eine gegebene Software-Infrastruktur impliziert entsprechenden Infrastrukturcode in den nutzenden Softwaresystemen. Dabei handelt es sich um Sourcecode, der im Wesentlichen dazu da ist, die technische Kopplung zwischen Infrastruktur und Anwendung herzustellen, so dass auf dieser Basis Fachlogik erstellt werden kann. Das J2EE/EJB-Programmiermodell dient hier als Paradebeispiel: Home- und Remote-Interfaces, Bean-Klassen, Deskriptoren etc., technischer Code, der zwar durchaus fachliche Information enthält (z.B. Methodensignaturen), aber eine sehr hohe Redundanz auf Code-Ebene aufweist. Spätestens nach drei bis fünf manuell erstellten Enterprise-Beans wünscht sich ein J2EE-Entwickler einen Generator für diese Art von Infrastrukturcode – und erhält diese Unterstützung typischerweise in Form eines Preprozessors bzw. Wizards einer IDE. Bestenfalls bringen gewisse Infrastrukturkomponenten also „Helfer“ zur Erzeugung *ihres* Infrastrukturcodes bereits mit[12]. Das Problem ist nur, dass diese Werkzeuge nichts voneinander „wissen“ und sie daher weit hinter den Möglichkeiten eines ganzheitlichen – eben architekturzentrierten - Ansatzes zurückbleiben, wie wir spätestens im Zuge der Fallstudie sehen werden.

Das Ziel architekturzentrierter, modellgetriebener Softwareentwicklung lautet also:

---

[12] Im Falle von EJB z.B. wird dies mit Version 3.0 des Standards gegeben sein.

Ganzheitliche Automation bei der Erstellung von Infrastrukturcode bzw. Minimierung von redundantem, technischen Code in der Anwendungsentwicklung.

Wenn wir über Infrastrukturcode reden, so ist das im Übrigen keine Kleinigkeit: Messungen (vgl. <Ref Oeko>) zeigen, dass moderne E-Business-Anwendungen typischerweise zwischen 60 und 70 Prozent Infrastrukturcode enthalten.

2.5.2. Generative Softwarearchitekturen

Wie das Adjektiv *architekturzentriert* bereits vermuten lässt, spielt die Softwarearchitektur die Schlüsselrolle der hier diskutierten MDSD-Ausprägung. Tatsächlich kann ein ganzheitlicher, generativer Ansatz zur Erzeugung von Infrastrukturcode nur auf Basis einer durchgängig ausgearbeiteten und formalisierten Softwarearchitektur funktionieren.

Man kann sich das folgendermaßen vorstellen: Je besser und weiter eine Softwarearchitektur ausgearbeitet ist, desto schematischer wird der Sourcecode der Anwendungen, die diese Architektur benutzen. Besteht die Architekturdefinition nur aus Folien, wo lediglich die System-Infrastruktur (Datenbanken, Application Server, Mainframes, Netzwerke etc.) und vielleicht noch die wichtigsten Schichten aufgezeichnet sind, so werden zwei Entwicklerteams ein und dieselbe Anwendung mit großer Wahrscheinlichkeit auf vollkommen verschiedene Art und Weise umsetzen – und zwar auch hinsichtlich der Implementierung einer Softwarearchitektur. Es entstehen sozusagen Unikate. Angenommen aber, es hätte ein Architekturteam vorgearbeitet und eine Art technische Referenzimplementierung erstellt, die die konkrete Umsetzung der wichtigsten Softwarearchitekturaspekte auf Sourcecode-Ebene zeigt und damit vorgibt, so könnten die Anwendungsentwickler diese Referenz gleichsam als Blaupause verwenden. Da in der konkreten Anwendung die gleichen technischen Umsetzungen (z.B. Nutzung einer Schnittstellentechnologie oder ein MVC-Pattern) immer wieder auftreten – allerdings in fachlicher Variation – wäre dann ein Großteil der Tätigkeit „Copy & Paste“-Programmierung. Diese Art der Programmierung ist natürlich deutlich effizienter als stets von neuem individuell erdachter Code. Zusammengefasst: Je weiter eine Softwarearchitekturdefinition in Richtung Sourcecode ausgearbeitet ist, desto schematischer wird die Anwendungsprogrammierung. Schematische Programmierung bedeutet zu einem Großteil Copy & Paste mit anschließendem Modifizieren je nach fachlichem Kontext. Dieser Teil der Arbeit ist aber offensichtlich eine nicht-intellektuelle Tätigkeit. Führt man den Gedanken nun weiter, so liegt die Idee nicht mehr fern,

die lästige und fehleranfällige Copy/Paste/Modify-Arbeit einem Generator zu überlassen. Genau dies führt in letzter Konsequenz zu einer *generativen Softwarearchitektur*. In ihr sind sämtliche Implementierungsdetails der Architekturdefinition – also alle architektonischen Schemata – in Form von Software enthalten. Als Eingabe benötigt sie ein fachliches Modell der Anwendung und als Ausgabe erzeugt sie den kompletten Infrastrukturcode der Anwendung (eben alles, was sonst per Copy/Paste/Modify erstellt werden müsste). Das Modell muss dazu lediglich gewisse Annotationen erhalten, die auf die zur Verfügung gestellten architektonischen Konzepte der generativen Softwarearchitektur verweisen.

Zur Modellierung wird bei AC-MDSD in der Regel ein architekturzentriertes UML-Profil (vgl. Abschnitt 2.4.1) verwendet. Es entsteht damit also ein formales, architekturzentriertes Anwendungs-*Design*. Die Modell-zu-Code-Transformationen werden üblicherweise in Form von Generator-Templates festgelegt und typischerweise von einem Generator-Framework verarbeitet, so dass der komplette Infrastrukturcode automatisch aus dem architekturzentrierten Design-Modell erzeugt werden kann. Wichtig dabei ist die Tatsache, dass das Modell bereits alle relevanten Informationen zur Generierung des Infrastrukturcodes enthalten muss – es ist eben nur um ein Vielfaches abstrakter und kompakter als der expandierte Code. Die Templates können sich die Mächtigkeit der gesamten Infrastruktur zunutze machen und den generierten Code auf diese Plattform (vgl. Abschnitt 2.3) abstützen, wodurch dieser entsprechend einfacher wird. Da der erzeugte Infrastrukturcode technisch/architektonisch motiviert ist, bleibt eine „semantische Lücke“, d.h. die Entwickler müssen die eigentliche Fachlogik der Anwendung manuell erstellen. Zur Integration von generiertem und manuell erstelltem Code gibt es wiederum verschiedene Techniken, die wir in den Kapiteln <Ref Trans> und <Ref Code> im Detail beleuchten. Abb. 8 zeigt die Zusammenhänge.

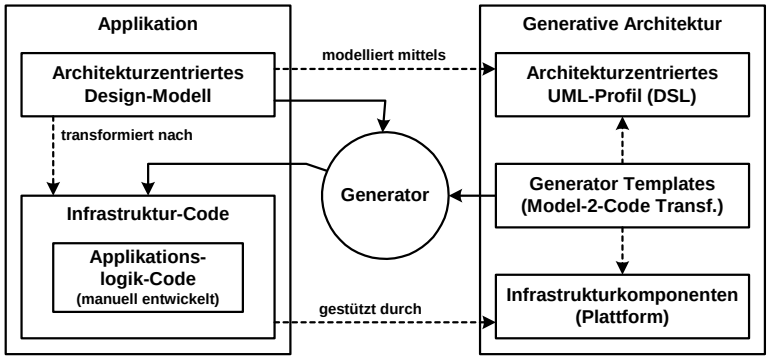


Abb. 8 *Prinzip architekturzentrierte MDSD*

Die Fallstudie im nächsten Kapitel wird dieses Bild im Detail durch ein praktisches und realistisches Beispiel konkretisieren.

Eine generative Softwarearchitektur ist ein mächtiges Mittel zur Erreichung der in Abschnitt 2.2 genannten Ziele. Die wichtigsten Vorteile sind erhöhte Entwicklungsgeschwindigkeit, verbesserte Wartbarkeit und Softwarequalität – und praktische Wiederverwendbarkeit. Wiederverwendung innerhalb einer Anwendung, aber natürlich auch (oder erst recht) über die Grenzen von Anwendungen hinweg: Eine generative Softwarearchitektur kann eine ganze Gruppe oder *Familie* von architektonisch gleichartigen Anwendungen tragen. Man nennt diesen Wirkungsbereich daher auch *Software-Systemfamilie*. Statt der Erstellung von Unikaten widmet sich AC-MDSD also der Erstellung von generativen Softwarearchitekturen für Software-Systemfamilien.

2.5.3. Architekturzentriertes Design

Die definierte Designsprache (typischerweise ein UML-Profil) enthält die Architekturkonzepte der Software-Systemfamilie in einer „plattformunabhängigen“[13] Abstraktion. Mit dieser Designsprache erstellen die Designer das fachliche Anwendungsdesign in Form von PIMs. Auf die Transformation dieser PIMs in explizit sichtbare, plattformabhängige UML-Modelle (PSMs) wird im Unterschied zur OMG-MDA-Vision bei AC-MDSD meist bewusst verzichtet.

Die Praxis in bisherigen Projekten hat gezeigt, dass diese Vereinfachung in der Regel einen höheren Nutzen hat als die

[13] Plattformunabhängigkeit ist ein relativer Begriff. Gemeint ist hier die Unabhängigkeit von Standard-Softwareplattformen wie J2EE.

zusätzlichen Freiheitsgrade beim Einsatz von PSMs. Das liegt daran, dass man nicht mehr viele sukzessive Transformationsschritte hat, deren Ergebnisse dann logischerweise auch einzeln kontrolliert, manipuliert und mit spezifischer Information angereichert werden müssen, sondern nur noch einen einzigen Schritt[14]. Dies gestattet nicht nur eine effizientere Entwicklung, sondern vermeidet auch das Konsistenzproblem der einzelnen Modelle: Ein manueller Eingriff in einem Zwischen-Modell bedeutet potenziell eine nicht automatisch korrigierbare Inkonsistenz zu den höheren Abstraktionsebenen.

Ebenso wird auf ein Reverse Engineering vom Sourcecode zum PIM verzichtet, was im Allgemeinen auch gar nicht möglich wäre: Mit dem PIM wird eine echte Abstraktion erreicht. Ein „rückwärts" aus Sourcecode gebildetes Modell ist naturgemäß so (wenig) abstrakt wie der Sourcecode selbst, nur anders (möglicherweise für bestimmte Zwecke besser verständlich) dargestellt. Für einen *beliebigen* Sourcecode existiert aber evtl. gar kein PIM-Modell, aus dem sich das Programm per Transformation ableiten ließe[15] – insbesondere dann nicht, wenn die PIM-Modellierungssprache eine spezifische Domäne wie „Softwarearchitektur für E-Business-Systeme" fokussiert. Dieser Umstand wird im Rahmen der MDA-Spezifikation seitens der OMG allerdings mehr oder weniger ignoriert. Einige Vertreter antizipieren wiederum toolgestützte Wizards o.Ä., die wenigstens ein halbautomatisches Reverse Engineering ermöglichen. Unserer Auffassung nach ist dies aber zumindest bei neu zu entwickelnder Software eher ein Zugeständnis als ein zielführendes Konzept.[16] Zugegebenermaßen mag diese Sichtweise, je nach persönlicher Arbeitsweise, zunächst als Nachteil empfunden werden, doch in Wahrheit ist es gerade eine Stärke, wie wir später noch genauer sehen werden. AC-MDSD setzt also im Grundsatz auf Forward Engineering.

Der vorwärtsgerichtete, generative Ansatz erlaubt dabei die Übertragung harter Aussagen über die architekturzentrierten Modelle auf die generierten Anwendungen. So kann beispielsweise

---

[14] Wir wenden uns dabei *nicht* gegen die Modularisierung von Transformationen durch Hintereinanderausführung, sondern nur gegen explizit sichtbare und manipulierbare Zwischenergebnisse.

[15] Mathematiker würden sagen, die Abbildung vom PIM-Modell auf eine Programmiersprache ist nicht surjektiv.

[16] Im Rahmen der Adaption von Legacy-Software für MDSD kann Reverse Engineering quasi als Bootstrapping Sinn machen.

die generative Architektur eine lose Kopplung von Komponenten oder die Nichtexistenz von Zugriffspfaden zwischen Anwendungsschichten garantieren. Zum Beispiel kann sichergestellt werden, dass die Präsentationsschicht (Web-Oberfläche) nicht direkt auf die SQL-Schnittstelle der Datenbank zugreifen kann.

Einen wichtigen Hinweis möchten wir an dieser Stelle noch geben: Forward Engineering ist nicht etwa mit einem Wasserfall-Vorgehensmodell zu verwechseln! Es bedeutet lediglich, dass Designänderungen im Modell gemacht werden und nicht direkt im Sourcecode, was natürlich nicht bedeutet, dass man erst die gesamte zu erstellende Anwendung zunächst ausmodelliert, um sie dann schrittweise zu Sourcecode zu transformieren, in der Hoffnung, dass man auf Anhieb alles richtig modelliert hat. Forward Engineering schließt ein derartiges Vorgehen zugegebenermaßen nicht aus, aber es erzwingt es keineswegs. Wir (und auch die MDA) proklamieren selbstverständlich ein iterativ, inkrementelles Vorgehen (vgl. [Oes01]).

Betrachten wir nun ein Beispiel eines solchen PIM (Abb. 9). Das Modell verrät nichts über verwendete Basistechnologien – die technologische Umsetzung solcher Modelle ist erst im Kontext einer konkreten Plattformbindung definiert. Durch die semantische Anreicherung des Modells mit Stereotypen, Tagged Values und Constraints entsteht eine formale UML-Designsprache. Bei AC-MDSD liegt das Abstraktionsniveau dieser Sprache auf der Ebene der Architekturkonzepte, weshalb wir auch von architekturzentriertem Design sprechen. Anders ausgedrückt: Die Domäne bei AC-MDSD lautet „Softwarearchitektur“.

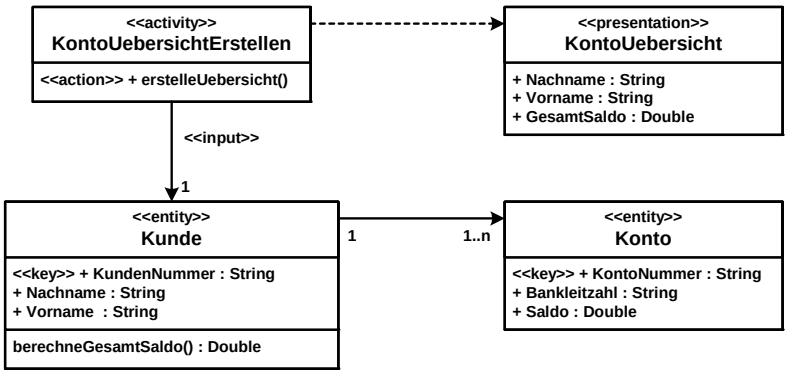


Abb. 9 *Beispiel für architekturzentriertes Design*

Die fachliche Bedeutung des Diagramms in Abb. 9 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. Bei AC-MDSD wird die programmiersprachliche Umsetzung des Modelles aber erst durch die Bindung an eine konkrete Plattform definiert. 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 Local 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 entsprechenden Deskriptoren 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. Die *presentation-Implementierungen werden von* einem FrontController-Framework aktiviert.

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 *activity*-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 Java-Swing GUIs, welche die Schnittstellen eines spezifischen Client-Frameworks implementieren.

Anhand dieses einfachen Modell-Beispiels lassen sich gut die wesentlichen Vorteile des Ansatzes erkennen: architekturzentrierte

Modelle sind kompakt, ausreichend informationsreich und enthalten keine unnötigen Details, welche die Portabilität behindern und den Abstraktionsgrad senken.

Sie sind dadurch übersichtlicher und leichter zu warten. Außerdem eignen sie sich besser zur Diskussion mit anderen Projektbeteiligten, da sie nicht mit technischen Details „verschmutzt“ sind.

2.5.4. Vorgehensweisen

Generative Softwarearchitekturen und architekturzentrierte Designs lassen sich nur effektiv einsetzen, wenn auch die Entwicklungsmethodik adäquat angepasst wird. Dieses sehr wichtige Thema liegt weit gehend außerhalb des Fokus der MDA. Wir haben ihm ein eigenes Kapitel gewidmet, das MDSD ganz allgemein aus prozessualer Sicht beleuchtet (siehe <Ref Proz>). Da wir an dieser Stelle allerdings noch den architekturzentrierten Spezialfall behandeln und im Wesentlichen den roten Faden für die nachfolgende Fallstudie vorbereiten, wollen wir uns zunächst nur mit einigen wenigen Schlaglichtern befassen:

Trennung von Architektur- und Anwendungsentwicklung

Wir haben gesehen, dass eine generative Softwarearchitektur zu einer Modularisierung der Anwendungsentwicklung führt: UML-Profil, Generator-Templates und Infrastrukturkomponenten auf der einen Seite, architekturzentriertes Design, generierter Infrastrukturcode und manuell kodierte Fachlogik auf der anderen Seite.

Offensichtlich gibt es dabei eine Abhängigkeit von den nutzenden Anwendungen auf die generative Softwarearchitektur, aber nicht umgekehrt. Dies führt zu der Überlegung, auch die *Erstellung* dieser Artefakte in zwei Stränge zu teilen. Ähnlich wie in der Framework-Entwicklung kann sich ein Team mit der Erstellung der generativen Softwarearchitektur (*Architektur-Entwicklungsstrang*) befassen, während sich ein anderes Team der Anwendungsentwicklung (*Anwendungs-Entwicklungsstrang*) widmet. Die Abhängigkeiten müssen dann durch entsprechende Synchronisation der Iterationen bzw. durch ein Release-Management aufgefangen werden – dazu, wie gesagt, mehr in Kapitel <Ref Proz>. Doch ganz unabhängig von der Frage, ob man eine personelle Trennung anstrebt oder nicht, handelt es sich ganz offensichtlich um inhaltlich unterschiedliche Tätigkeiten, so dass eine rollenbezogene Sichtweise Sinn macht:

- *Architekten* entwickeln die generative Softwarearchitektur.
- *Designer* erstellen das architekturzentrierte Modell der Anwendung.

- *Entwickler* kodieren die Fachlogik und integrieren sie in den generierten Infrastrukturcode.

Natürlich können diese Rollen prinzipiell auch in Personalunion ausgefüllt werden. Wir werden in der Fallstudie allein zur Verdeutlichung der unterschiedlichen Tätigkeiten allerdings diese Sichtweisen einnehmen und uns auf die beiden genannten Stränge begeben. Eine ausführliche Betrachtung von Rollen und Teamstrukturen enthält Kapitel <Ref Orga>.

Bedeutung der Referenzimplementierung

Eine praxistaugliche, generative Softwarearchitektur entsteht nicht „aus der hohlen Hand“ – man benötigt eine Vorlage für das spätere Generat. Diese nennen wir *Referenzimplementierung*. Es handelt sich dabei um ein möglichst schlankes, lauffähiges Beispiel, welches die Bedeutung der Konstrukte des architekturzentrierten UML-Profils auf Sourcecode-Ebene zeigt. Aus einer solchen Referenzimplementierung lassen sich dann im nächsten Schritt Generator-Templates ableiten. Auch dies werden wir im Verlauf der Fallstudie konkretisieren und in Kapitel <Ref Proz> vertiefen.

2.5.5. Merkmale architekturzentrierter MDSD

Bevor wir uns nun endgültig der Fallstudie widmen, wollen wir kurz zusammenfassen, welche Eigenschaften AC-MDSD besitzt:

- Methodische Aspekte rücken in den Vordergrund: AC-MDSD unterstützt individuelle architektonische Anforderungen. Es steht also klar das Engineering-Prinzip und nicht eine integrierte Entwicklungsumgebung (CASE- oder MDA-Tool/IDE) im Zentrum. 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 Generatorschablonen ableitet.
- Software-Systemfamilien statt Unikate: AC-MDSD zielt nicht nur auf die Effizienz- und Qualitätssteigerung einer Anwendung, sondern darüber hinaus auf die Wiederverwendung von generativen Softwarearchitekturen im Rahmen architektonisch

gleichartig gelagerter Anwendungen, die damit eine Software-Systemfamilie bilden. Dieser Aspekt steht nicht explizit im Fokus der MDA.

- Architekturzentriertes Design: Wir verzichten im Gegensatz zur MDA auf den expliziten Einsatz von plattformspezifischen Modellen. 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 für Zwischenergebnisse werden reduziert und Konsistenzprobleme vermieden.

- Forward Engineering: Der Verzicht auf Roundtrip Engineering entgegen der MDA-Vision geschieht bewusst. Da in architekturzentrierten MDSD-Modellen echte Abstraktionen vorgenommen werden, ist ein Reverse Engineering nicht sinnvoll bzw. möglich. Design-Änderungen müssen im Design vorgenommen werden. Dadurch ist das Modell stets konsistent mit dem Sourcecode (Generat).

- Modell-zu-Modell-Transformationen nur zur Modularisierung: Wir verwenden ein möglichst abstraktes PIM, das dann idealerweise direkt (aber natürlich iterativ) in Sourcecode überführt wird. Die „Transformationsstrecke“ kann zwar durchaus mittels Modell-zu-Modell-Transformationen modularisiert sein, auf dem Weg entstehende „Zwischenmodelle“ sind jedoch für den Anwendungsentwickler nicht sichtbare Implementierungsdetails.

- Quelltextgenerierung ohne Verwendung des Zielmetamodells: Die Generierung von programmiersprachlichem Quelltext ist essenziell für AC-MDSD. Die Vorteile dieser Technik werden damit zu Vorteilen von MDSD (vgl. <Ref Code>). Die Beschreibung von Abbildungen zwischen Metamodellen zur Definition von Modelltransformationen hingegen, wie sie derzeit im Rahmen der MDA-Standardisierung diskutiert wird, halten wir nur für Modell-zu-Modell-Transformationen für sinnvoll. Architektonisch motivierter Infrastruktur-Sourcecode lässt sich auf diese Art nur sehr umständlich generieren. Der Einsatz von Generatorschablonen (Templates) hingegen hat sich in der Praxis bewährt und ist sehr intuitiv handhabbar. Das Quellmetamodell (also das der Designsprache) ist im Gegensatz zum Zielmetamodell für die Sourcecode-Generierung äußerst nützlich, um daran die Transformationsvorschriften festzumachen, wie unsere Fallstudie zeigen wird.

- Keine 100%-Generierung: Es wird aus *architekturzentrierten* Modellen in der Regel „nur“ 60–80% einer Software generiert. Wir halten eine 100%-Generierung auch nur in Ausnahmefällen für möglich und sinnvoll[17]. Generiert wird der architektonische Infrastrukturcode einer Anwendung. Die individuellen bzw. fachlichen Aspekte werden in der Zielsprache „manuell“ ergänzt. Dennoch findet iterative Generierung statt, wobei bereits vorgenommene manuelle Ergänzungen natürlich erhalten bleiben.

- Softwarearchitektur wird handhabbar: Die generative Softwarearchitektur ist formal und per se aktuell. Die Entwickler können den gesteckten Infrastrukturcode-Rahmen weder unabsichtlich noch absichtlich verlassen. Dies ist ein klarer Qualitätsvorteil. Entwickler und Designer stellen andererseits notwendige Architekturänderungen sofort fest und können diese an der richtigen Stelle (nämlich zentral in der generativen Softwarearchitektur und nicht verteilt in der Anwendung) durchführen. Technische und fachliche Aspekte sind damit eindeutig voneinander getrennt. Damit unterstützt AC-MDSD die Durchgängigkeit der Architektur und Machbarkeit von architektonischen Änderungen in der Breite. Schematische Sourcecode-Anteile werden hundertprozentig sicher umgesetzt, und auch die Änderungen an der Architektur fließen mit gleicher Sicherheit sofort in die Umsetzung ein. Dies wiederum unterstützt die Skalierbarkeit des Entwicklungsprozesses. Mit anderen Worten: AC-MDSD ist ein sehr wertvolles und operatives Instrument für das Architekturmanagement.

[17] Diese Aussage gilt wohlgemerkt nur für AC-MDSD, nicht für MDSD im Allgemeinen.

3. Fallstudie: Eine typische Web-Anwendung

3.1. Einleitung

Nachdem wir in den vorangegangenen Kapiteln die notwendigen Grundlagen bezüglich MDSD im Allgemeinen und architekturzentrierter, modellgetriebener Softwareentwicklung (AC-MDSD) im Besonderen gelegt haben, können wir uns nun einer konkreten Fallstudie widmen, um AC-MDSD in der Praxis kennen zu lernen.

3.2. Anwendungs-Entwicklungsstrang

Zunächst nehmen wir die Sicht des Anwendungsentwicklers ein, d.h. wir setzen eine existierende generative Softwarearchitektur voraus (vgl. <Ref Basis::AC>). Diese wird (wie im letzten Kapitel bereits erwähnt) typischerweise iterativ und inkrementell *parallel* zur Anwendungsentwicklung erstellt. Die dazu notwendigen Methoden werden wir im Kapitel <Ref Proz> noch ausführlich diskutieren.

Es sei noch einmal darauf hingewiesen, dass es sich bei dieser Sichtweise um eine rollenbezogene Darstellung handelt. Über die Zuordnung von Rollen zu Personen sagen wir zunächst nichts aus. Diese Balancierung ist eine Frage der Projektorganisation (siehe <Ref Orga>). Es geht uns hier primär darum, die unterschiedlichen Tätigkeiten zu kategorisieren, was das Verstehen der Inhalte erleichtern sollte.

Anhand eines Anwendungsbeispiels werden wir die wesentlichen wiederkehrenden Schritte exemplarisch erläutern und abschließend die Beziehung zwischen Anwendungs-Entwicklung und generativer Architektur beschreiben.

Eine Iteration im Anwendungs-Entwicklungsstrang beginnt mit der initialen Erstellung oder Erweiterung des Anwendungsdesigns unter Zuhilfenahme eines UML-Tools. Die aus dem UML-Tool exportierte XMI[18]-Repräsentation des Anwendungsdesigns wird durch einen MDSD- Generator in einen Implementierungsrahmen transformiert. Die eigentliche Fachlogik – der fachliche Code – wird manuell programmiert und in den generierten Infrastrukturrahmen integriert. In dieser Fallstudie verwenden wir dazu exemplarisch das Mittel der geschützten Bereiche (*Protected Regions* oder *Protected Areas*). Dabei handelt es sich syntaktisch gesehen um Kommentare der Zielsprache, die aber vom MDSD-Generator interpretiert werden. Jede Protected-Region im Generat besitzt (eben als Kommentar „getarnt“) einen global eindeutigen Bezeichner und ist eindeutig mit einem Modellelement verknüpft. Dadurch kann der Generator die Inhalte dieser Bereiche insofern schützen, als dass er sie bei iterativer Generierung an den richtigen Stellen im neuen Generat wieder einsetzt. Das Verfahren ist auch robust gegen Umbenennungen im Modell, da die IDs der Protected Regions aus UUIDs[19] des Modells (genauer des XMI-Formats) erzeugt werden und nicht aus Klassennamen oder Ähnlichem. Inhalte von geschützten Bereichen (d.h. Fachlogik) können also (je nach Build-Prozess) allenfalls verloren gehen, wenn die entsprechenden Modellelemente tatsächlich gelöscht werden – und in diesem Fall ist das in der Regel natürlich auch das erwünschte Verhalten.

Protected Regions sind nicht immer das beste Mittel, um generierten und manuell programmierten Code zu integrieren, doch darauf wollen wir an dieser Stelle noch nicht näher eingehen (siehe <Ref Trans::BP-Tech::IntCode>).

3.2.1. Das Anwendungsbeispiel

Das folgende Beispiel ist einem MDA/MDSD-Tutorial entnommen, welches im Rahmen der Arbeitsgemeinschaft der *Architecture Management Group* [AMG] erstellt und bereits auf diversen

---

[18] XMI = XML Metadata Interchange. Ein MOF-XML-Mapping, das meist dazu verwendet wird, UML-Modelle in XML-Form zu serialisieren. Nahezu alle UML-Tools unterstützen XMI als interoperables Exportformat.

[19] Global eindeutige Identifikatoren für Modellelemente

Konferenzen (JAX 2003/4, OOP 2004 u.a.) mit großem Erfolg als Hands-On-Session vorgestellt wurde. Die Anwendung wurde geschaffen, um einen ganzheitlichen Ansatz zur Entwicklung von Software, angefangen bei der Geschäftsprozessanalyse über Systemanalyse, architekturzentriertes Design und modellgetriebene Codegenerierung bis hin zur Implementierung der Fachlogik anhand eines einfachen aber nicht trivialen Beispiels demonstrieren zu können. Das Analysemodell entstammt den Schulungsunterlagen der Firma oose.de, die generative Softwarearchitektur wurde von der b+m Informatik AG erstellt. Zum jetzigen Zeitpunkt existiert noch kein öffentlicher Zugang zu der Software – dies ist jedoch im Rahmen des Open Source Projektes *openArchitectureWare* [OAW] geplant.

Der fachliche Hintergrund der Anwendung basiert auf einem Fallbeispiel zur Erstellung eines Informationssystems für ein Car-Sharing-Unternehmen. Abb. 10 zeigt die Use-Case-Übersicht der fiktiven Anwendung „Car-Sharing 1.0“.

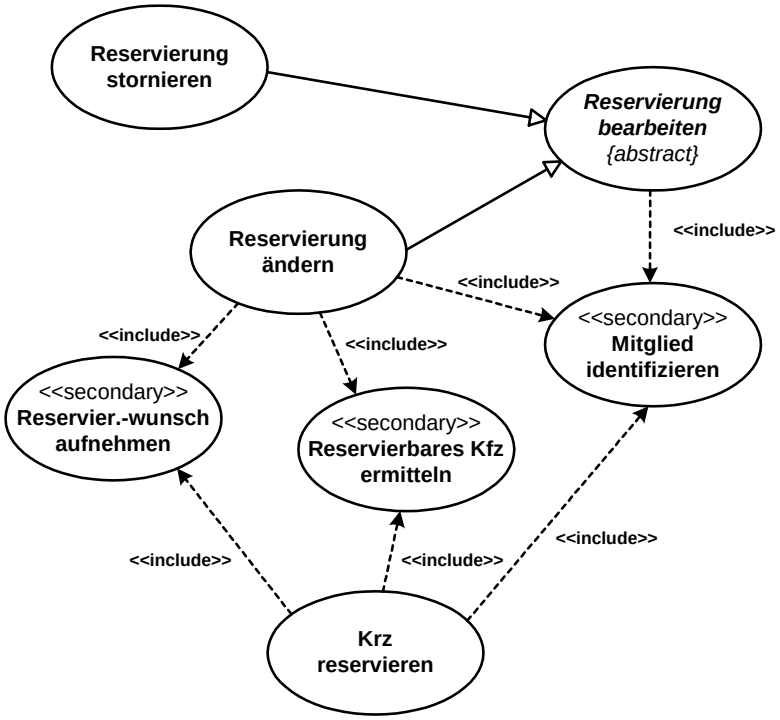


Abb. 10 *Use-Case-Übersicht der Car-Sharing-Anwendung*

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 Datenhaltungs-Schicht (siehe Abb. 11). Sie basiert auf der J2EE-Rahmenarchitektur.

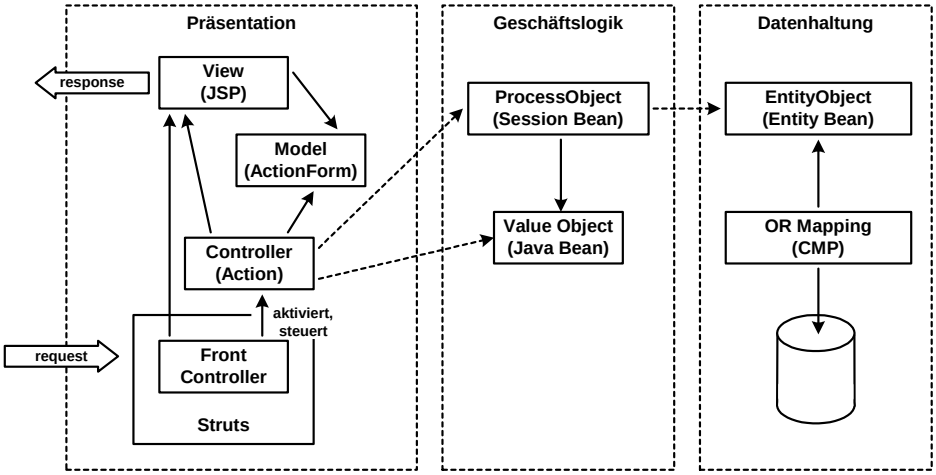


Abb. 11 *Car-Sharing-Basisarchitektur*

Die Präsentationsschicht verwendet das MVC Pattern gemäß der Servlet-Model 2-Architektur à la Struts [STRT]. Alle vom Browser kommenden HTTP-Anfragen werden zentral von einem *FrontController* entgegengenommen, 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* zur Verfügung. Zur Ablaufsteuerung stützt sich die Schicht auf das 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 Prozessobjekte zur Anzeige relevante Daten lesen und neu erfasste Daten persistieren. Gleichzeitig werden die in der Datenhaltungsschicht liegenden Entitäten gemäß des Process-Entity-Patterns vor direkten Zugriffen durch Objekte der Präsentationsschicht geschützt.

Die Datenhaltungsschicht verfügt über ein persistentes Business Object Model (BOM), welches durch Java 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 [TOMC], der EJB 2.0 konforme Applikationsserver JBOSS [JBOS] und die Datenbank HyperSonic SQL [HSQL]. 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.

Die Erstellung des für Car-Sharing verwendeten Designs in Form eines PIMs erfolgt mit Hilfe einer Designsprache (UML-Profil), 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*, ...). Auf die genaue Profildefinition gehen wir später ein. Die Transformation (Plattformbindung) auf die Zielplattform erfolgt über einen Satz von Generator-Templates, welche aus den Modellinformationen den entsprechenden Sourcecode für die oben skizzierte Zielumgebung generieren. Die Designsprache und die Plattformbindung in Form der Templates bilden die generative Softwarearchitektur für Car-Sharing 1.0 (siehe <Ref Basis::AC>), die in Abb. 12 gezeigt ist.

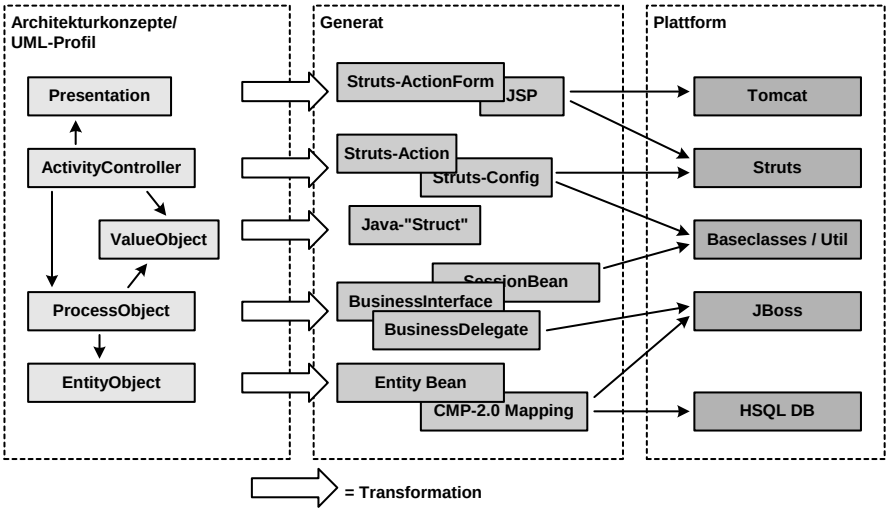


Abb. 12 *Generative Softwarearchitektur und Laufzeitumgebung*

3.2.2. MDSD-Werkzeuge

Um AC-MDSD praktisch zu nutzen, benötigen wir ein UML-Modellierungswerkzeug und einen MDSD-Generator. Das UML-Modellierungswerkzeug muss in der Lage sein, mit UML-Profilen zur UML-Spracherweiterung zu arbeiten. Zur Zeit findet sich kein UML-Tool, das Modellierungs-Constraints, d.h. die Prüfung von Zusicherungen, die auf der Metaebene in Form von OCL-Ausdrücken[20] festgelegt worden sind, auswerten kann. Die Prüfung der Zusicherungen sollte daher durch den MDSD-Generator unterstützt werden.

Das Generator-Werkzeug muss die vom ausgewählten UML-Werkzeug erstellten Modelle lesen und als Eingabe für die Generierung nutzen können. Die meisten UML-Tools sind heute in der Lage, Modelle im XMI-Format abzuspeichern. Allerdings ist XMI noch lange nicht gleich XMI. Hier sollte der MDSD-Generator vorgefertigte oder anpassbare Adapter für die unterschiedlichen Modellierungswerkzeuge bieten.

Eine genauere Betrachtung von Tools und Anforderungen findet sich in Kapitel <Ref Tool>.

In unserem Falle kommt als UML-Modellierungswerkzeug Poseidon UML Community Edition von Gentleware zum Einsatz [POSE]. Dessen XMI-Output wird vom Generator-Framework openArchitectureWare [OAW] über Generator-Templates in Sourcecode transformiert. Dieser Sourcecode wird dann in der integrierten Entwicklungsumgebung Eclipse weiterbearbeitet [ECLI]. Für das Generator-Framework liegt ein Eclipe-Plugin vor, so dass auch hier eine Verwendung in einer integrierten Entwicklungsumgebung stattfinden kann. Zudem erfüllt es die oben genannten Anforderungen.

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

3.2.3. 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

[20] OCL = Object Constraint Language, Teil der UML

generiert werden, bietet sich eine Änderung auf Ebene einer Präsentation an (siehe Abb. 13).

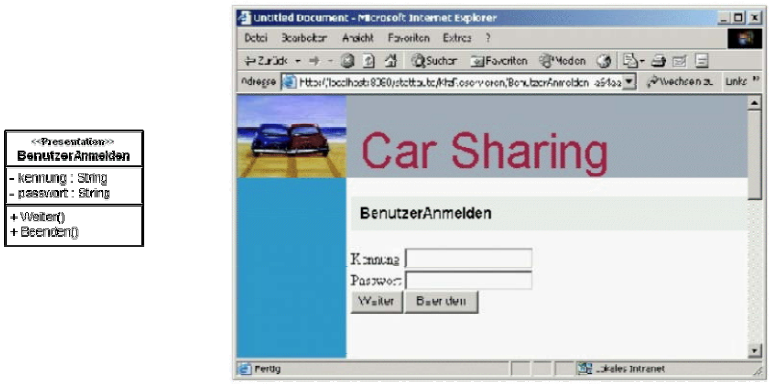


Abb. 13 *Transformation vom Modell BenutzerAnmelden auf den konkreten Dialog BenutzerAnmeldenView*

Der linke Bereich von Abb. 13 zeigt die Präsentations-Klasse *BenutzerAnmelden; der rechte* den aus ihr entsprechend generierten Dialog als HTML 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 generiert. Beide Artefakte sind Ergebnisse der Transformation. Der folgende Code zeigt die generierte JSP *BenutzerAnmelden.jsp*.

```
...
<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.BenutzerAnmeld
en.kennung"/> </td>
    <td>
      <html:text property="kennung"/>
    </td>
    </tr>
  <tr>
    <td><bean:message
key="de.amg.stattauto.benutzer.presentation.BenutzerAnmeld
en.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>
...
```

Das folgende Listing zeigt die dahinter liegende Formklasse.

```
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;
  }
}
```

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 findet anschließend über die Entwicklungsumgebung bzw. über Ant [ANT] statt.

Hier wird erkennbar, dass das Modell die Stellung von Sourcecode einnimmt. Alle Informationen über die durchgeführte Änderung werden im Modell gehalten. So ist es nahe liegend, dass das Modell neben den eigentlichen Sourcen in das Release-Management der Anwendung einbezogen wird.

3.2.4. 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 Reservierung notwendigen Parameter in der Prozessschicht bestimmt und der Präsentationsschicht zur Verfügung gestellt werden.

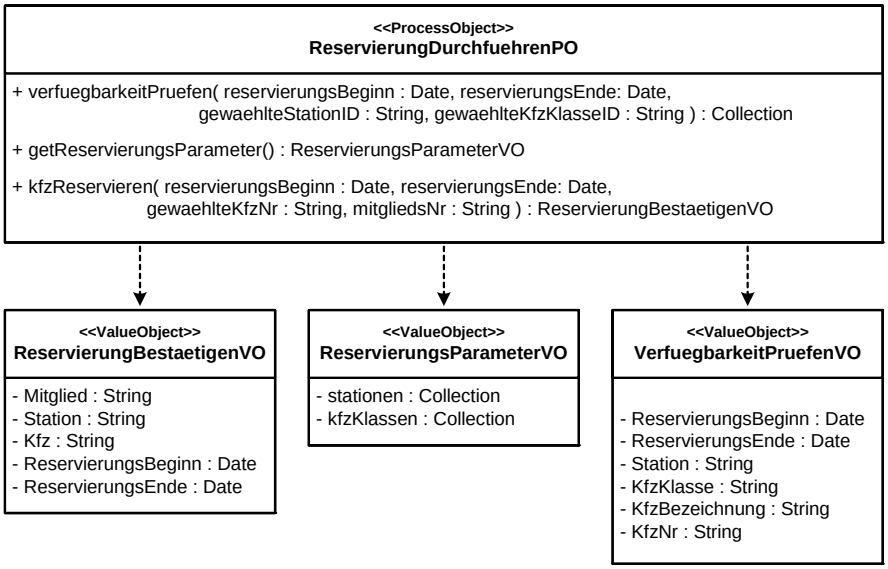


Abb. 14 *ReservierungDurchfuehrenProcessView*

Abb. 14 zeigt den in der Prozessschicht benötigten Modellausschnitt. Das *ReservierungDurchfuehrenPO* erhält dazu die *getReservierungsParameter()*,-Methode, die ein *ReservierungsParameterVO* zurückliefert. Das *ReservierungsParameterVO* 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 *ReservierungDurchfuehrenPO* in ein SessionBean notwendig sind. Zusätzlich wird ein *ReservierungDurchfuehrenBusinessDelegate* gemäß des Business Delegate Patterns aus den J2EE Core Patterns generiert [CORE]. Das *ReservierungsParameterVO* wird zu 100% generiert. Für die Methode *getReservierungsParameter()* wird jedoch „nur“ die Methodensignatur generiert. Die Implementierung muss manuell in der IDE ergänzt

werden. Dies geschieht in geschützten Code-Bereichen, wie im folgenden Codeausschnitt gezeigt.

```
public ReservierungsParameterValueObject
       getReservierungsParameter()
    throws RemoteException {
    // PROTECTED REGION ID(12Operation_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
  }
```

Die Entscheidung über mögliche geschützte Code-Bereiche ist auf Architekturebene bei der Erstellung der Generator-Templates zu treffen (siehe Abschnitt 3.3). Ergänzungen oder Änderungen außerhalb dieser geschützten Bereiche sind bei MDSD nicht erlaubt, weil sie der klaren Trennung zwischen Modellierung und Programmierung auf der einen Seite und zwischen Anwendungs- und Architektur-Entwicklung auf der anderen Seite zuwiderlaufen würden: Design-Änderungen müssen im Design (Anwendungsmodell) und Architekturänderungen (d.h. systematische Änderungen am generierten Code) in der generativen Architektur erfolgen. Das Generator-Framework stellt dies sicher, indem nicht erlaubte Änderungen bei erneuter (iterativer) Generierung verloren

gehen. Dies ist nicht etwa eine Gängelung der Anwendungsentwickler, sondern sorgt für Konsistenz und einen aktiven Austausch zwischen Anwendungs- und Architektur-Entwicklung. Die Festlegung dieser Grenze zwischen Generat und Fachlogik ist von zentraler Bedeutung und erfordert einiges an Fingerspitzengefühl (vgl. <Ref Archi>).

3.2.5. Beispiel 3: Arbeiten mit dynamischen Modellen

Neben der in den vorangegangenen Beispielen aufgezeigten Möglichkeit der Generierung von Sourcecode auf Basis statischer Modelle, können auch dynamische Modelle wie Aktivitätsdiagramme und Zustandsdiagramme zur Codegenerierung genutzt werden (vgl. <Ref GPM>). Wie so etwas aussehen kann, wollen wir mit diesem Beispiel aufzeigen. Abb. 15 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).

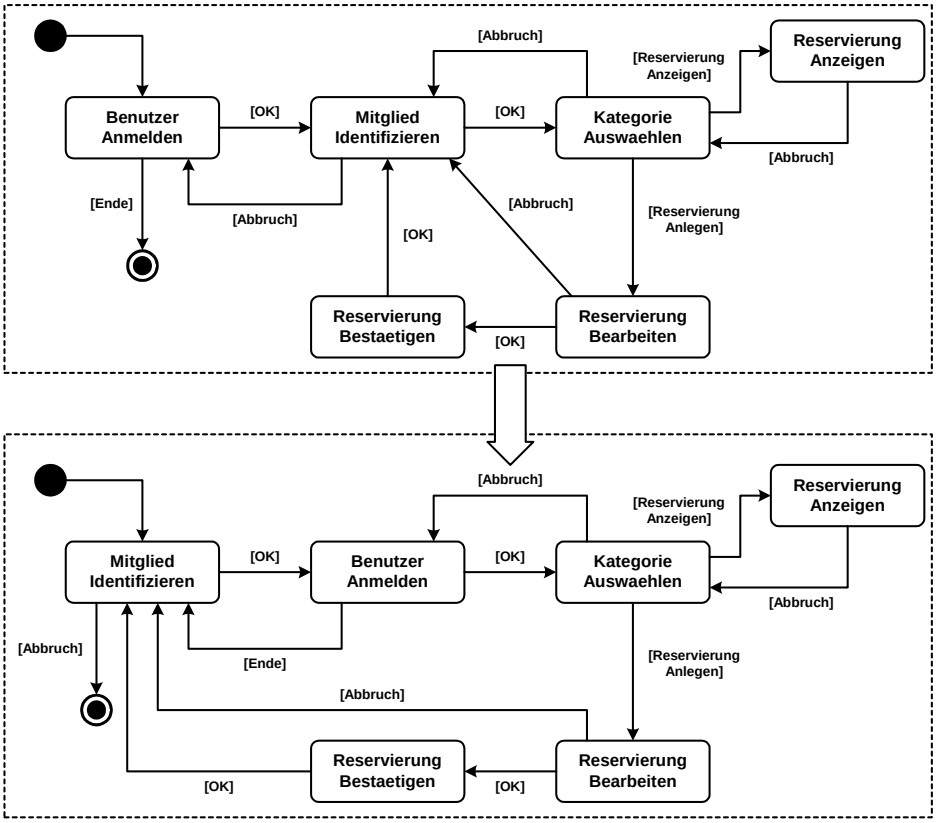


Abb. 15 *Änderung der Navigationsfolge*

Da wir uns für Struts als Framework 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:

```
<!-- ControllerState "BenutzerAnmelden" -->
<action
 path="/ BenutzerAnmelden_a64aa2a7d0162ba7ffb_Init"
 type="de.amg.stattauto.benutzer.
            presentation.BenutzerAnmeldenController"
 name="BenutzerAnmeldenForm"
 input="/BenutzerAnmelden.jsp"
 scope="request"
 parameter="BenutzerAnmelden_Init,a6488aa27d162ba7ffb">
   <forward name="Ok"
            path="/BenutzerAnmelden.jsp"
            contextRelative="true"
   />
</action>

<action
```

```
 path="/BenutzerAnmelden_a64aa2a7d0162ba7ffb_Exit"
 type="de.amg.stattauto.benutzer.
             presentation.BenutzerAnmeldenController"
 name="BenutzerAnmeldenForm"
 input="/BenutzerAnmelden.jsp"
 scope="request"
 parameter="BenutzerAnmelden_Exit, a6488aa27d162ba7ffb ">
   <forward name="BenutzerAnmelden_To_Beenden"
            path="/MitgliedsnummerAngeben_a6affa_Init.do"
    />
   <forward name="Weiter"
            path="/KategorieAuswaehlen_a64aac9_Init.do"
   />
   <forward name="Fehler"
            path="/BenutzerAnmelden_a64aa30f0fbfb_Init.do"
   />
</action>
```

Das folgende XML-Fragment ist ein Auszug aus der Struts config.xml, der das Angeben der Mitgliedsnummer steuert.

```
<!-- ControllerState "MitgliedsnummerAngeben" -->
<action
 path="/MitgliedsnummerAngeben_a64aa3a062ba7ffa_Init"
 type="de.amg.stattauto.mitglied.
       presentation.MitgliedIdentifizierenController"
 name="MitgliedIdentifizierenForm"
 input="/MitgliedIdentifizieren.jsp"
 scope="request"
 parameter="MitgliedsnummerAngeben_Init,a60f06b7f">
   <forward name="Ok"
            path="/MitgliedIdentifizieren.jsp"
            contextRelative="true"
   />
</action>
```

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 (vgl. <Ref GPM>).

3.2.6. Zusammenspiel Entwicklung und Architektur

Ein funktionierendes Zusammenspiel zwischen Entwicklung und Architektur ist ein wichtiger Schlüssel zum Erfolg in MDSD-Projekten. Denn mit der Bereitstellung der generativen Softwarearchitektur 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 Softwarearchitektur. 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, das aus der Entwicklung stammende Feedback aufzunehmen und in die generative Softwarearchitektur einfließen zu lassen, entsteht am Ende eine maximal tragfähige Lösung zur Erfüllung der gestellten Anforderungen. Somit entwickelt sich die generative Softwarearchitektur ähnlich einem Framework evolutionär weiter, muss dementsprechend versioniert und den Projekten, in denen sie verwendet wird, zur Verfügung gestellt werden. Diese Themen betrachten wir ausführlicher im dritten Teil des Buches.

Um bei der Entwicklung eigene Ideen zur Verbesserung der verwendeten generativen Softwarearchitektur 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 gewünschten Ergebnis, kann sie dem gesamten Projekt durch Anpassung der generativen Softwarearchitektur zur Verfügung gestellt werden. Dies hat den Vorteil, dass die generative Softwarearchitektur innerhalb eines Projekts immer in einem klar definierten und konsistenten Zustand vorliegt.

3.2.7. Zwischenfazit

In der Rolle des Entwicklers gewinnt man mehr Zeit für die „wesentlichen Aufgaben“ – 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 MDSD-Generator bzw. der generativen Softwarearchitektur übernommen.

Die Fehlerbehebung im technischen Code ist 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 der generativen Softwarearchitektur behoben werden (ähnlich dem Fehlerfixing in einem Framework). Nach erneuter Generierung werden sämtliche fehlerhaften Codefragmente durch korrigierte ersetzt.

Durch die Tatsache, dass wir manuell kodierte Fachlogik mit einem generierten Infrastrukturcode-Rahmen integrieren, verlieren wir

allerdings die vollkommene Plattformunabhängigkeit und automatische Portabilität unserer Anwendung. Ganz offensichtlich besitzen die Inhalte der geschützten Bereiche Abhängigkeiten auf die Programmiersprache Java, das Framework Struts usw. Es gibt zwar Pattern (wie z.B. das verwendete *BusinessDelegate*), die helfen, diese Abhängigkeiten zu verringern, doch an die OMG-Vision der ausführbaren Modelle kommen wir nicht heran. Hier treten die unterschiedlichen Zielsetzungen zu Tage (vgl. <Basis::AC>): AC-MDSD ist pragmatisch und stellt eben die Steigerung von Entwicklungseffizienz, Qualität, Wartbarkeit und Wiederverwendung in den Vordergrund, MDA Portabilität und Interoperabilität.

3.3. Architektur-Entwicklungsstrang

Im letzten Abschnitt wurde AC-MDSD aus Sicht der Modellierung und der eigentlichen Programmentwicklung betrachtet. Dabei wurden Komponenten und Elemente wie ein UML-Profil (Designsprache), eine Plattform (J2EE, Struts, Persistence-Layer etc.) und eine generative Softwarearchitektur, die die Arbeit erledigt, vorausgesetzt. Wir beleuchten nun, wie diese Artefakte im Detail aussehen können.

Ein Schlüsselkonzept heißt, wie bereits in Kapitel <Ref Basis::AC> ausgeführt, Redundanzvermeidung. Redundanz (d.h. Schemata, die in unterschiedlicher Ausprägung mehrfach vorkommen) ist nicht nur auf der Ebene von EJBs zu finden, sondern in allen Schichten moderner Softwarearchitekturen: Ablaufsteuerung (z.B. Struts), Präsentation (z.B. JSPs und ActionForms), Controllern, Anbindung von Host-Systemen etc. Unser Ziel ist es, diese Redundanz möglichst komplett in eine generative Softwarearchitektur zu verlagern, die sozusagen alle Konstruktionsprinzipien und die Programmiermodelle in den verschiedenen Schichten kennt, nicht nur einzelne Facetten daraus. Ein Nutzen, der sich daraus ergibt, ist ein enormer Produktivitätsschub in der gesamten Anwendungsentwicklung, wie die Beispiele im Abschnitt <Ref Proz::Strg::AWStrg> zeigen.

Eine solche generative Softwarearchitektur geht über einfaches XSLT, XDoclet o.Ä. natürlich weit hinaus und besitzt offensichtlich keine Abhängigkeit zu einer speziellen Anwendung, sondern ist wiederverwendbar. Sie trägt eine ganze Software-Systemfamilie mit gleichen technologischen Eigenschaften. Letztere kann also sowohl die Car-Sharing-Anwendung als auch eine Versicherungsanwendung beinhalten, eben wenn die gleichen technologischen Prinzipien zugrunde liegen. Wir schaffen uns einen Fertigungsprozess für Anwendungen, mit

dem diese aus Modellen zu einem Großteil automatisch gefertigt werden können, analog zu Produktionsstraßen im Automobilbau. Der Architektur-Entwicklungsstrang unseres MDSD-Prozesses beschäftigt sich mit der Erstellung solcher „Produktionsstraßen“ (siehe auch <Ref Proz> und <Ref PLE>).

3.3.1. UML-Profil

Zunächst einmal benötigen wir ein architekturzentriertes UML-Profil, mit dem wir formale MDSD-Modelle erstellen können. Betrachten wir dazu das Modell in Abb. 16.

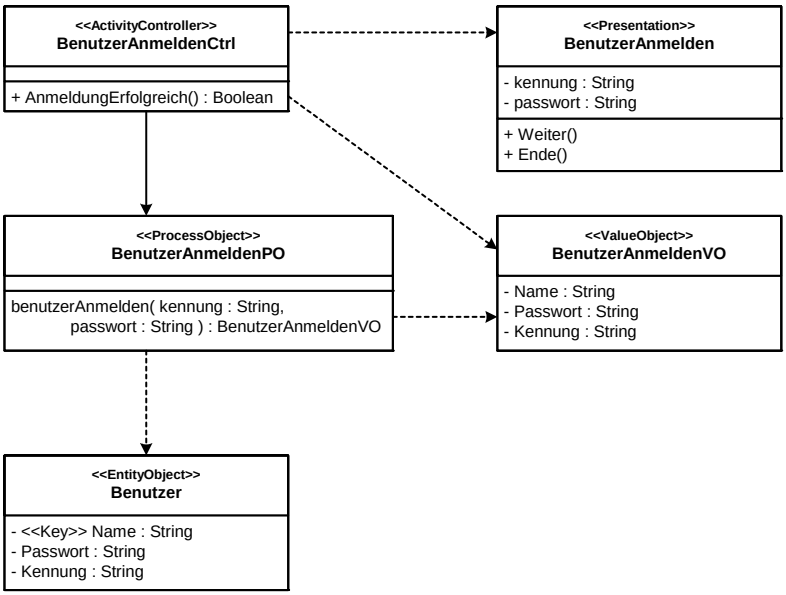


Abb. 16 *Architekturzentriertes Design BenutzerAnmelden*

Die fachliche Bedeutung des Modells liegt auf der Hand: Die Klasse *BenutzerAnmeldenCtrl* ist als Schritt der Ablaufsteuerung in der Lage, eine Präsentation zur Eingabe und Übernahme der Kennung und Passwort eines Benutzers zu aktivieren. Zur Authentifizierung am System verwendet der Controller den Service *benutzerAnmelden()* des *BenutzerAnmeldenPO*.

Das ProcessObject erhält anhand der Kennung, die das identifizierende Merkmal eines Benutzers darstellt, die passende Entität des Benutzers und validiert das eingegebene Passwort. Bei erfolgreicher Anmeldung erhält der Controller vom Service ein *BenutzerAnmeldenVO* mit den entsprechenden Zugangsdaten.

Zur Modellierung wird eine Designsprache verwendet, welche auf die in der Anwendung verwendeten Architekturkonzepte abzielt. Der Fokus der Designsprache liegt auf der architektonischen Wiederverwendung und mündet in einem architekturzentrierten Design. Es abstrahiert vollständig von technologischen Details. Das vorliegende Modell lässt sich durch entsprechende Vorschriften in den Sourcecode für diverse Zielplattformen transformieren.

Die wichtigsten konzeptionellen Elemente des UML-Profils haben wir bereits in Abb. 12 kennen gelernt (linke Spalte). Über die entsprechenden Transformationen des Generators erfolgt die Bindung auf die in der Car-Sharing-Architektur verwendete Plattform (rechte Spalte).

Abb. 17 zeigt nun einen Ausschnitt aus der formalen UML-Profildefinition:

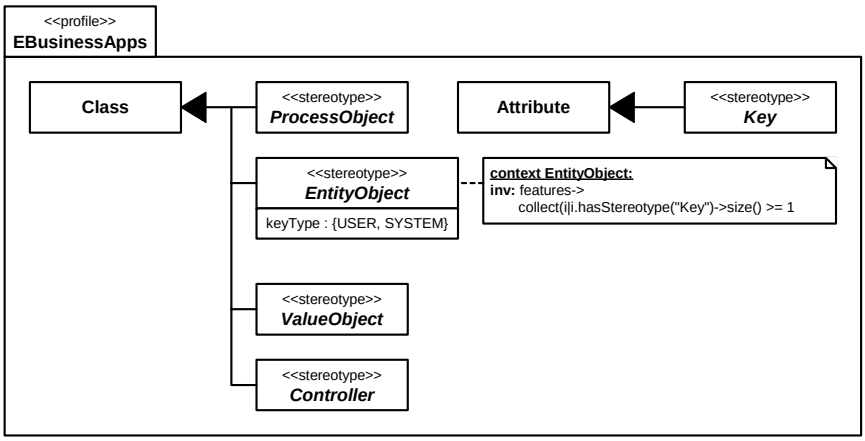


Abb. 17 *UML Profildefinition für unsere Designsprache*

Das Profil stellt eine Spezialisierung des Standard-UML-Metamodells (vgl. <Ref MM>) auf Klassen-, Attribut- und Operations-Ebene dar, wodurch ein spezielles Sprachprofil für die konkreten Anforderungen im Kontext der 3-Schichten-Architektur von Car-Sharing ausgeprägt wird.

Es werden die UML-Erweiterungsmechanismen Stereotypen und Tagged Values verwendet. So sind mit dem Stereotypen <<*Presentation*>> versehene Klassen, wie z.B. BenutzerAnmelden, für die Anzeige und Eingabe von Daten zuständig. Als <<*EntityObject*>> gekennzeichnete Klassen, wie z.B. Benutzer, bilden die persistenten Geschäftsklassen der Anwendung und bieten Mechanismen zur eindeutigen Identifizierung und zur Abfrage. Design Constraints (Modellierungsregeln) sind ebenfalls wichtige Teile des Profils und können mit Hilfe der Object Constraint Language formuliert werden.

Idealerweise würde die Profildefinition inklusive der Constraints über das UML-Tool vorgenommen und auch gleich von diesem interpretiert

werden, so dass beim Anwendungsdesign z.B. genau die für ein Stereotyp zulässigen Tagged Values zur Verfügung stehen und die Modellierungsregeln (Constraints) des Profils überprüft würden. Leider besitzen die gängigen UML-Tools diese Fähigkeiten noch nicht, so dass die UML-Profildefinition (noch) eher Dokumentations-Charakter hat.

3.3.2. Transformationen

Nachdem wir ein UML-Profil definiert haben, können wir uns an die eigentliche Codegenerierung heranwagen. Diese Aufgabe ist als Ganzes nicht leicht zu erledigen. Glücklicherweise sind aber diverse Teilaufgaben allgemeiner Natur, und es gibt MDSD-Tools, die sie einem abnehmen. Dazu gehören das Neutralisieren des XMI-Outputs der UML-Tools, die Expansionssteuerung, das Input/Output-Stream-Handling, das Scannen und Erhalten geschützter Codebereiche für die Fachlogik. Die meisten MDSD-Generatoren sind Frameworks und verwenden eine templateartige Sprache, um die Transformationen vom Modell zum Code einfach beschreiben zu können.

Mit XSLT beispielsweise kommt man hier schnell an die Grenzen, insbesondere dann, wenn man direkt auf dem XMI-Output des UML-Tools aufsetzt. Die XML-Strukturen sind dort tief verschachtelt und indirekt referenziert, so dass XSLT-Stylesheets schnell absolut unleserlich und damit unwartbar würden. Außerdem fehlt die Mächtigkeit einer Programmiersprache wie Java und auch geschützte Codesegmente lassen sich nur sehr schwer und unhandlich realisieren.

Leider gibt es wie gesagt noch keinen Standard für MDA- (oder MDSD-)Generatoren bzw. Transformationen. Im Markt sind dementsprechend viele verschiedene Ansätze mit unterschiedlichen Leistungsmerkmalen vorhanden. Im Folgenden greifen wir uns das Open Source Generator-Framework *openArchitectureWare* heraus, um anhand des Car-Sharing-Beispiels zu zeigen, wie die Metaprogrammierung im Architektur-Strang im Detail aussieht.

Metamodell/Profil-Implementierung

Damit unser Generator-Framework aus dem Anwendungsdesign den Implementierungsrahmen für die Zielplattform generieren kann, benötigt es eine Java-Implementierung des verwendeten UML-Profils (siehe Abschnitt <Ref CSI::Archi::GenFW>). Das *openArchitectureWare*-Framework bringt zur Vereinfachung dieser Aufgabe bereits eine Java-Implementierung des UML-Class- und Activity-Core-Metamodelles mit, die einfach durch Java-Vererbung spezialisiert werden kann. Diese

Spezialisierungen entsprechen dann gerade den Stereotypen des UML-Profils. Wir müssen also zunächst einmal lediglich für jedes Stereotyp im Profil eine gleichnamige Java-Klasse anlegen. Diese muss dann von derjenigen mitgelieferten Metaklasse erben, die auch im UML-Profil als Scope für den Stereotyp angegeben ist. Also z.B.

```
public class EntityObject extends Class
{}

public class Key extends Attribute
{}
```

Class und *Attribute* sind hier nicht etwa die Klassen aus dem *java.lang.reflect*-Package, sondern wie gesagt Metaklassen, die das *openArchitectureWare*-Framework mitbringt.

Der Clou ist nun, dass das Generator-Framework das so spezialisierte Metamodell *instanziieren* kann, d.h. zu Beginn eines Generatorlaufs erzeugt es für jedes Modellelement, das mit dem Stereotyp <<*EntityObject*>> gekennzeichnet ist, eine Instanz der in Java implementierten Metaklasse *EntityObject*. Jedes einzelne Element des Eingabe-Designmodells (Klassen, Assoziationen, Attribute, Operationen, Parameter, Aktivitäten, Transitionen usw.) wird also durch genau ein Java-Objekt des entsprechenden Typs in der JVM des Generators repräsentiert. Nicht-stereotypisierte Elemente führen dabei natürlich zur Instanziierung der jeweiligen Core-Metaklasse. Die Metamodell-Implementierung wird also instanziiert, d.h. ein Designmodell wird in ein Java-Objekt-Geflecht überführt und steht damit dem Generator zum Zugriff bereit.

Die spezialisierten Klassen haben neben der Repräsentation von Stereotypen bei *openArchitectureWare* noch weitere wichtige Aufgaben:

Abbildung von Tagged Values: Ein stereotypspezifisches Tagged Value im UML-Profil wird einfach auf ein gleichnamiges String-Attribut der zugehörigen Java-Metaklasse abgebildet.

Service-Methoden für die Generierung: Um die Template-Programmierung einfach zu machen und die Scriptsprache nicht zur Programmiersprache auswachsen zu lassen, werden Hilfsmethoden, die zur Generierung benötigt werden, eben einfach in Java programmiert und zwar als *public*-Methoden der Metaklassen. Diese können dann von den Templates aus einfach als *Properties* der Metaklassen aufgerufen werden.

Das folgende Listing zeigt diese nächste Stufe[21]:

```
public class EntityObject extends Class
{
  /** Set mit allen Key-Attributen der Klasse */
  protected ElementSet KeyList = null;

  /** Liefert Set mit allen Key-Attributen der Klasse */
  public ElementSet Key() throws  DesignException
  {
    if (KeyList == null) {
      KeyList = new ElementSet();

      for (int i=0; i < Attribute.size(); i++){
        if (Attribute().get(i) instanceof Key) {
          KeyList.add(Attribute().get(i));
        }
      }
    }
    return KeyList;
  }
}
```

Attribute() wird dabei von *Class* vererbt und ist die Java-Repräsentation der Metabeziehung zwischen Klassen und ihren Attributen – oder anders ausgedrückt: Über diese API ist abfragbar, welche Attribute der Designer an der aktuellen Klasse (in diesem Fall sogar ein *EntityObject*) modelliert hat.

Die Template-Sprache von *openArchitectureWare* macht keinen Unterschied zwischen dem Zugriff auf Attribute und Methoden einer Metaklasse, es sind einfach Properties. Methoden verschatten gleichnamige Attribute. Das hier definierte *Key*-Property liefert einem Template alle mit dem Stereotyp *<<Key>>* versehenen Attribute des *EntityObjects*, um etwa elegant im Template die *PrimaryKey*-Klasse generieren zu können.

Unterstützt das verwendete UML-Tool keine Design-Constraints, kann deren Prüfung in der Metamodell-Implementierung vorgenommen werden und vom Generator vor der eigentlichen Generierung automatisch aufgerufen werden. Das Generator-Framework stellt hierfür eine Schnittstelle zur Verfügung, über die eine Prüfung von Zusicherungen vor der Transformation durchgeführt wird. Der Generator prüft nach der Instanziierung der Metamodell-Implementierung alle

---

[21] Für übliche Constraints, wie zum Beispiel auch die im folgenden Listing, bietet openArchitectureWare bereits vorgefertigte Hilfsfunktionen, die wir hier aber aus Gründen der Einfachheit zunächst nicht verwenden. Die zweite Fallstudie in Kapitel <Ref CSII> verdeutlicht diesen Weg.

Zusicherungen durch Aufruf der *CheckConstraints()*-Methode auf den Modellelementen. Wie im nächsten Listing dargestellt, kann so beispielsweise sichergestellt werden, dass für jedes *EntityObject* mindestens ein *Key* definiert ist.

```
// EntityObject.CheckConstraints() definiert die
// DesignConstraints für Elemente mit dem
// Stereotypen <<EntityObject>>
public String CheckConstraints() throws  DesignException {
  if(Key().isEmpty())
    throw new DesignException("Constraint violation:
          "+No Key found for EntityObject '" +
          this.Name() + "'");
  return "";
}
```

So erhält man bei einem Modellierungsfehler aussagekräftige Fehlermeldungen und wird nicht mit einer unverständlichen Generator-Exception konfrontiert. Ein solches Feature ist für den produktiven Einsatz in echten Projekten unerlässlich.

Template-Programmierung

Der plattformspezifische Implementierungsrahmen wird durch so genannte Templates erzeugt. Templates besitzen eine große Ähnlichkeit zum Generat und lassen sich daher leicht aus einer Referenzimplementierung ableiten. Die konstanten Anteile der Referenzimplementierung werden bei der Erstellung von Templates mit Copy-Paste-Charakter als Text erfasst und mit Hilfe von Kontrollstrukturen der Template-Sprache mit den direkt aus dem Metamodell gelesenen Properties kombiniert.

Auf diese Weise werden z.B. sämtliche Klassen und Deskriptoren (mit den Eigenschaften für Deployment, Persistenz, Relationen etc.) der EJB EntityBeans der Car-Sharing-Anwendung zum Ablauf im Container des JBoss bis auf die EQL für fachlich motivierte Finder vollständig aus den mit *<<EntityObject>>* gekennzeichneten Klassen generiert.

Als einfaches Beispiel ist das Template zur Generierung des Naming-Eintrags für ein Entity Bean in den plattformspezifischen Deployment-Deskriptor *jbossDD.xml* aufgeführt:

```
«DEFINE DeplDescr FOR EntityObject»
  «FILE FullPathName"/"Name"jbossDD.xml"»
    <entity>
      <ejb-name>«Name»EJB</ejb-name>
       «IF needsRemote»
         <jndi-name>
           «FullPackageName».«Name»RemoteHome
         </jndi-name>
       «ENDIF»
```

```
      <local-jndi-name>
        «FullPackageName».«Name»Home
      </local-jndi-name>
    </entity>
  «ENDFILE»
«ENDDEFINE»
```

Der Output für die Klasse Benutzer aus unserem Beispiel wird in die Datei *BenutzerjbossDD.xml* geschrieben und sieht wie folgt aus:

```
<entity>
  <ejb-name>BenutzerEJB</ejb-name>
  <local-jndi-name>
    de.amg.stattauto.benutzer.entity.BenutzerHome
  </local-jndi-name>
</entity>
```

Dieses kurze Beispiel-Template zeigt bereits die Einfachheit und Prägnanz der Template-Sprache. Lediglich die durchgängig groß geschriebenen Bezeichner sind Elemente der Template-Sprache. Die anderen Bezeichner innerhalb der doppelten spitzen Klammern sind Properties im Metamodell. Der restliche Teil (außerhalb der doppelten spitzen Klammern) wird als konstanter Text in die Zieldatei expandiert. Der für dieses Template relevante Ausschnitt des in Java implementierten Metamodells, inklusive Properties, ist für das Car-Sharing-Beispiel vereinfacht in Abb. 18 dargestellt:

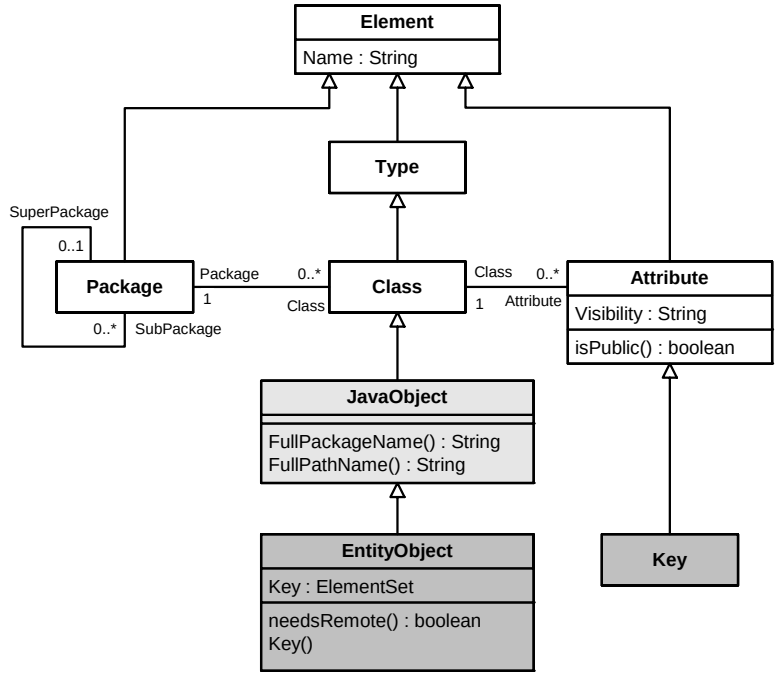


Abb. 18 *Relevanter Ausschnitt des in Java implementierten Metamodells*

Die Klasse *EntityObject* entspricht dem gleichnamigen Stereotypen der Designsprache. Die anderen Klassen sind Teil des Kern-Metamodells für Klassendiagramme (siehe Abschnitt 3.3.3). Das Design des Beispiels aus Abb. 16 würde in unserem Fall genau eine Instanz der Klasse *EntityObject* liefern. Die Entity-Instanz mit dem Namen Benutzer hätte drei assoziierte Attribut-Instanzen mit den Bezeichnungen *Name*, *Passwort* und *Kennung*.

Das Set-Property *Key* liefert, wie wir gesehen haben, eine Collection mit Key-Instanzen der assoziierten Attribute. Über das Bool'sche Property *needsRemote* lässt sich abfragen, ob die Entität remote aufrufbar sein soll und somit für die vorliegende Plattform entsprechende Remote-Interfaces generiert werden müssen. Die *String*-Properties der Basisklasse *JavaObject*, *FullPackageName* und *FullPathName* traversieren die Package-Hierarchie des Designs und bilden zielsprachenkonforme Strings zur Erzeugung von Java-Import-Anweisungen bzw. Dateinamen inklusive Pfad. Es sind aus Sicht der Templates also wiederverwendbare Services. Die neue Basisklasse wurde zwischengeschoben, weil die Properties eben nicht nur für *EntityObject*, sondern auch für *ProcessObject*, *ValueObject* und andere Metaklassen Sinn machen. Sie ist aber abstrakt und kann daher nicht direkt instanziiert werden.

Bereits an diesem einfachen Beispiel sieht man deutlich, dass Metamodellierung ein zentrales MDSD-Thema ist. Wir haben ihm daher ein eigenes Kapitel gewidmet (siehe <Ref MM>).

Das Template mit dem Namen *DeplDescr* wird mit Hilfe eines *DEFINE*-Blocks (*DEFINE ... ENDDEFINE*) in einer speziellen Template-Datei (einfache Textdatei mit dem Suffix .tpl) definiert. Dabei bezieht es sich mit *FOR EntityObject* auf eine Klasse des Metamodells (*EntityObject*). Entsprechend ist unser Beispiel-Template nur für Entities expandierbar und darf somit eigene Properties (wie z.B. *needsRemote*) direkt nutzen. Beim Property-Zugriff kann natürlich die Java-seitige Vererbung genutzt werden, so dass alle Properties der Basisklassen von *JavaObject* (z.B. *FullPackageName*), *Class*, *Type* und *Element* (wie z.B. *Name*) zur Verfügung stehen.

Der *FILE*-Block (*FILE ... ENDFILE*) ermöglicht die direkte Expansion von Templates in eine Datei, wobei der Dateiname in unserem Beispiel durch Zusammenfügung von String-Properties und String-Konstanten dynamisch entsteht. *FullPathName* ist in der Metamodell-Implementierung eine Methode, *Name* ein Attribut. Aus Template-Sicht besteht dabei hinsichtlich der Art des Zugriffs wie gesagt kein Unterschied.

Über den *IF*-Block (*IF ... ENDIF*) sind bedingte Expansionen von Template-Teilen durchführbar. So kann im vorliegenden Beispiel alternativ ein entferntes oder ein lokales Home-Interface in den *jboss.xml* Deployment-Deskriptor eingetragen werden.

Eine besonders nützliche Eigenschaft dieser Template-Sprache ist die Unterstützung von Polymorphismus auf Template-Ebene: Gleichnamige Template-Definitionen an verschiedenen Metaklassen innerhalb eines Vererbungsbaumes werden zur Laufzeit des Generators wie Methoden in Java über den *dynamischen* Typ gebunden. Dies ist eines der wichtigsten OO-Konzepte und dient der Vermeidung schwer wartbarer und im ganzen Code verstreuter Typ-Abfragen (*instanceof*).

Die gesamte Template-Sprache[22] des *openArchitectureWare*-Frameworks umfasst weniger als 30 Konstrukte und ist sofort intuitiv beherrschbar.

[22] Der Name der *openArchitectureWare* Template-Sprache lautet *XPand*.

3.3.3. Funktionsweise des MDSD-Generators

Werfen wir einen Blick auf Abb. 19, die zeigt, wie das *openArchitectureWare*-Framework eine generative Softwarearchitektur verarbeitet.

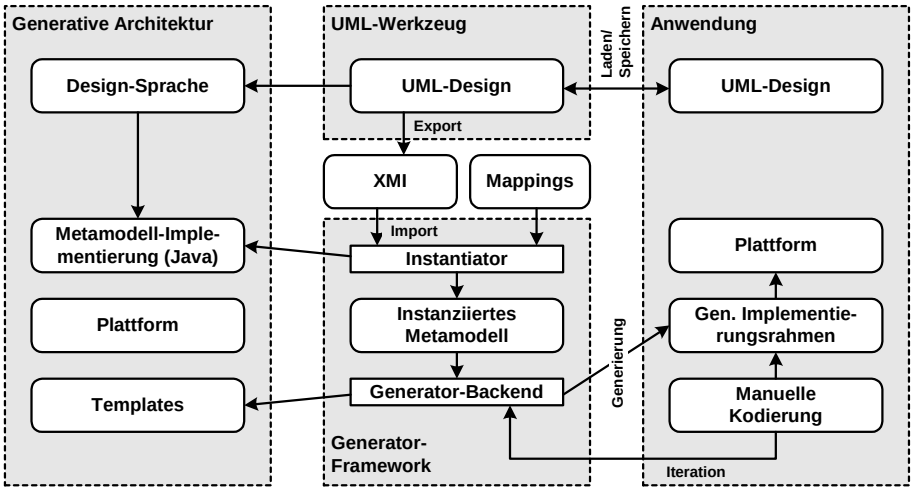


Abb. 19 *Funktionsweise des openArchitectureWare Generator-Frameworks*

Die Komponenten im Einzelnen:

Die generative Softwarearchitektur beinhaltet alle für den Generator-Einsatz notwendigen Bausteine.

Designsprache: Als Designsprache wird in oft ein UML-Profil verwendet, d.h. Stereotypen, Tagged Values und Constraints, mit denen die Standard-UML um spezifische Konzepte erweitert wird[23].

UML-Design: Das UML-Design ist das Modell einer konkreten Anwendung der Software-Systemfamilie. Zur Modellierung wird dementsprechend die definierte Designsprache verwendet.

XMI-Input: Das UML-Design wird über das Modellierungswerkzeug in eine XMI-Repräsentation exportiert. Diese XMI-Repräsentation des Designs kann vom Generator weiterverarbeitet werden. Jedes Modellelement muss im XMI mit einer UUID (global eindeutige ID) versehen sein.

---

[23] Beliebige andere Modellierungssprachen können auch verwendet werden.

Metamodell-Implementierung (in Java): Dieser MDSD-Generator besitzt ein frei konfigurierbares Metamodell. Es wird in Java implementiert. Das heißt, für jedes benötigte Standard UML-Element und jeden Stereotyp in der Designsprache gibt es genau eine Java-Klasse im Metamodell. Auf diese Weise kann der Instantiator des Generator-Frameworks die Informationen im XMI-Input verwenden, um das Metamodell per Reflection-API zu instanziieren. Dazu verwendet er eine Instanziierungsvorschrift, die angibt, welches XMI-Element auf welche Metamodellklasse abgebildet wird. Das UML-Design liegt ab diesem Zeitpunkt als Java-Objektgeflecht im Heap der JVM des Generators vor. Die Objekte des Geflechts sind gerade Instanzen der Klassen aus dem Metamodell (diese Technik ist vergleichbar mit dem DOM-Tree, der beim Parsen von XML-Dokumenten entsteht – Compiler-Bauer sprechen von einer abstrakten Syntax). Das instanziierte Metamodell ist die Schnittstelle zum Generator-Backend. Es kapselt damit die unübersichtlichen XMI-Strukturen vor den Templates. Gleichzeitig ermöglicht es die Programmierung von Hilfsmethoden für die Generierung und die Prüfung von Modellierungsregeln des UML Profils in Java.

Templates: Das hier vorgestellte Framework besitzt eine Template-Sprache, die zusammen mit dem in Java implementierten Metamodell einen objektorientierten Generator bildet (die Konstrukte des Metamodelles übersetzen sich selbst). Die Template-Sprache erlaubt eine einfache und elegante Formulierung der gewünschten Transformationen auf Basis des Metamodells (siehe Beispiele in diesem Abschnitt). Die Templates werden vom Generator-Backend dynamisch an das instanziierte Java-Metamodell gebunden. Sie steuern die eigentliche Sourcecode-Expansion.

Generator-Backend: Das Backend interpretiert die Templates, übernimmt das Datei-Handling sowie das Scannen der geschützten Bereiche und die Übernahme der Inhalte (schon vorhandene Fachlogik) in den neu generierten Architekturrahmen. Damit hier z.B. beim Umbenennen von Klassen im Design nichts verloren geht, verwendet der Generator die UUIDs in der XMI-Repräsentation, um die geschützten Bereiche zu identifizieren.

Instanziierungsvorschrift: Bei diesem Generator kann man das Mapping zwischen der XMI-Repräsentation und dem Metamodell in Form einer XML-Datei deklarieren. Damit können XMI-Formate (z.B. verschiedener UML-Tools) und das Metamodell

unabhängig voneinander variieren. Es ist dadurch im Prinzip sogar möglich, beliebige (non-UML) XML-Inputs zu verarbeiten. Durch das Konzept der abstrakten Syntax (d.h. das Metamodell) bleiben die Templates von einer Änderung des konkreten Input-Formats unberührt.

Laufzeitsystem: Das Laufzeitsystem (bzw. die Plattform) gehört logisch zur generativen Softwarearchitektur, da es nicht von einer konkreten Anwendung abhängt. Aber jede generierte Anwendung benutzt natürlich das Laufzeitsystem. Anders ausgedrückt: Der generierte Code enthält Abhängigkeiten (z.B. Methodenaufrufe, *extends*- oder *implements*-Statements etc.) auf die Plattform.

3.3.4. Bootstrapping

Um die soeben beschriebenen Artefakte initial zu erstellen, wird, wie schon in Kapitel <Ref Basis::AC> angedeutet, eine Art Bootstrapping-Prozess benötigt. Es ist schwierig und auch nicht sinnvoll, im Projekt direkt mit der Template-Programmierung zu beginnen, wenn noch keine generative Softwarearchitektur vorliegt. Stattdessen sollte der später generierte Code zunächst einmal „von Hand“ beispielhaft ausgearbeitet und erst anschließend daraus die Systematik in Form von Templates abgeleitet werden. Eine lauffähige Referenzimplementierung stellt dafür die Basis. Statische Codefragmente können 1:1 in die Templates übertragen werden, die variablen Codeanteile werden mit Hilfe der Template-Sprache auf Basis des Metamodells formuliert. Dadurch wird die Template-Erstellung zu einer Aufgabe, die sich im Wesentlichen mit eleganter Textersetzung befasst und nicht mehr mit der Ausarbeitung von architektonischen Inhalten belastet ist. Die Trennung vereinfacht also beide Teilaufgaben.

Im Kapitel <Ref Proz> werden wir die prozessualen Aspekte modellgetriebener Softwareentwicklung weiter vertiefen.

3.3.5. Anpassungen der generativen Softwarearchitektur

Im Laufe des Lebenszyklus einer Applikation ergeben sich in der Regel nicht nur Änderungen und Erweiterungen an funktionalen Anforderungen, sondern es ändern sich auch die Anforderungen an architektonische Aspekte der Applikation durch z.B. Einsatz der Software auf einem anderen Applikationsserver, Migration auf eine neue EJB- oder Struts-Version. Wo normalerweise alle betroffenen Klassen

auf die neue Struktur manuell angepasst werden müssten, kann beim Einsatz eines MDSD-Generators an einer zentralen Stelle die Änderungen durchgeführt werden. Die Transformation wird in den Templates entsprechend angepasst und der neue Infrastrukturcode durch eine Neugenerierung erzeugt. Manuelle Anpassungen sind ausschließlich innerhalb der geschützten Bereiche im Sourcecode vonnöten, und auch nur dann, wenn sich die strukturelle Änderung auf das Programmiermodell auswirkt.

Allein anhand eines kleinen Ausschnitts des hier verwendeten Beispiels ist die Arbeitserleichterung sehr gut darzustellen. Ändert sich z.B. aufgrund einer neuen Version des EJB-Komponentenmodells oder des verwendeten Applikationsservers die Struktur der Deskriptoren, ist nur das bereits dargestellte Template anzupassen und nicht alle **jbossDD.xml*-Dateien aller *EntityObject*-Klassen der Anwendung. Eine angenommene Migration vom JBoss-Container auf den Container des Bea Weblogic Servers würde das *<<EntityObject>>* Benutzer im Beispiel u.a. den folgenden Deployment-Deskriptor notwendig machen:

```
<weblogic-enterprise-bean>
  <ejb-name>BenutzerEJB</ejb-name>
  <entity-descriptor>
    <persistence>
      <persistence-use>
        <type-identifier>
          WebLogic_CMP_RDBMS
        </type-identifier>
        <type-version>6.0</type-version>
        <type-storage>
          META-INF/weblogic-cmp-rdbms-jar.xml
        </type-storage>
      </persistence-use>
    </persistence>
  </entity-descriptor>
  <local-jndi-name>
    de.amg.stattauto.benutzer.entity.BenutzerHome
  </local-jndi-name>
</weblogic-enterprise-bean>
```

Um diese Änderung für alle Klassen des Typs *EntityObject* zu erreichen, wird das Template für die Generierung der Deskriptor-Dateien, wie im nachfolgenden Listing angedeutet, entsprechend geändert und die Anwendung neu generiert.

```
«DEFINE DeplDescr FOR EntityObject»
«FILE FullPathName"/"Name"weblogic-ejb-jarDD.xml"»
  <weblogic-enterprise-bean>
    <ejb-name>«Name»EJB</ejb-name>
    <entity-descriptor>
      <persistence>
        <persistence-use>
          <type-identifier>
```

```
            WebLogic_CMP_RDBMS
          </type-identifier>
          <type-version>6.0</type-version>
          <type-storage>
            META-INF/weblogic-cmp-rdbms-jar.xml
          </type-storage>
        </persistence-use>
      </persistence>
    </entity-descriptor>
    «IF needsRemote»
      <jndi-name>
        «FullPackageName».«Name»RemoteHome
      </jndi-name>
    «ENDIF»
    <local-jndi-name>
      «FullPackageName».«Name»Home
    </local-jndi-name>
  </weblogic-enterprise-bean>
«ENDFILE»
«ENDDEFINE»
```

Die Anforderungen an architektonische Aspekte können natürlich nicht nur Änderungen von bestehenden Strukturen, sondern auch Erweiterungen erforderlich machen. Um die notwendigen Schritte zur Erweiterung einer generativen Softwarearchitektur zu erläutern, soll ein Tagged Value für Geschäftsklassen eingeführt werden. Das Tagged Value soll mit *KeyType* bezeichnet werden und kann den Wert *USER* oder *SYSTEM* annehmen. Über *KeyType* kann die Art des eindeutigen Schlüssels der Geschäftsklasse bestimmt werden. Im Falle des *KeyType == SYSTEM* wird ein Attribut und der eindeutige Schlüssel generiert, andernfalls (*KeyType == USER*) wird der Schlüssel durch Kennzeichnung eines Attributs mit den Stereotypen *<<Key>>* festgelegt. Im Beispiel in Abb. 20 besitzt die Geschäftsklasse Benutzer den *KeyType == USER* und das als *<<Key>>* gekennzeichnete Attribut *Name*.

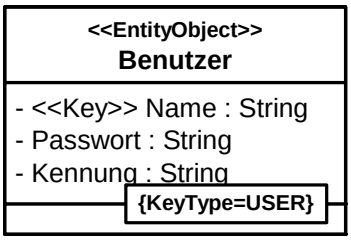


Abb. 20 *Erweitertes EntityObject Benutzer*

Durch die Erweiterung des Profilelementes *<<EntityObject>>* muss auch das Modellierungs-Constraint entsprechend angepasst werden: *<<Key>>*-Attribute müssen und dürfen nur noch im Falle *KeyType ==*

USER angegeben werden. Abb. 21 zeigt die entsprechend geänderte formale Profildefinition mit Tagged Value und OCL-Constraint.

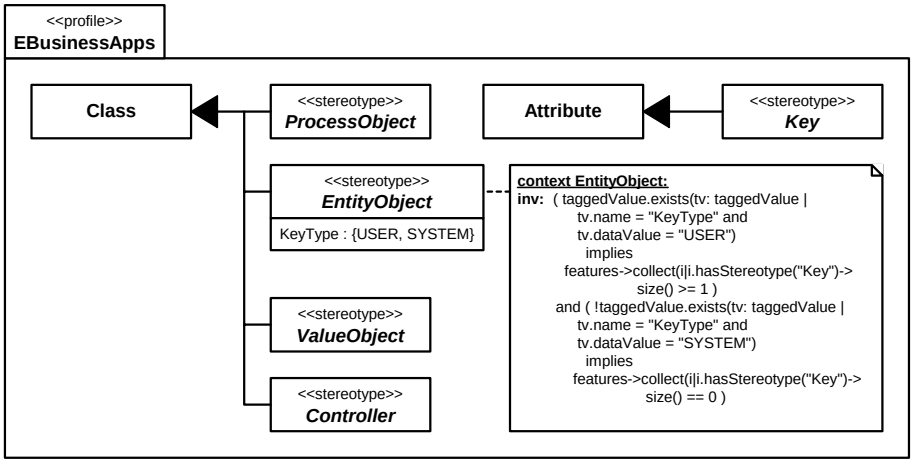


Abb. 21 *Profildefinition mit Constraints*

In der Implementierung unserer generativen Softwarearchitektur wird das Tagged Value im Metamodell in der Klasse *EntityObject* als Property eingeführt, welches vom Generator-Framework beim Instanziieren des Metamodells gesetzt wird. Die Implementierung der *CheckConstraints()*-Methode muss entsprechend erweitert werden:

```
public class EntityObject extends JavaObject {
  public String KeyType = "USER"; //TaggedValue Default
  ...
  // EntityObject.CheckConstraints()
  // definiert die DesignConstraints für
  // Elemente mit dem Stereotypen <<EntityObject>>
  public String CheckConstraints()throws DesignException {
    if( Key().isEmpty() &&
        KeyType.equals("USER")) {
      throw new DesignException("Constraint "+
            +"violation: No Key found for "+
            +"EntityObject '" + this.Name() + "'");
    }
    return "";
  }
  ...
}
```

Die erweiterte Mächtigkeit des UML-Profils schlägt sich natürlich auch in den Templates nieder. Hier müssen, wie unten für die Entity-Bean-Klasse angedeutet, abhängig vom *KeyType* unterschiedliche Transformationen erfolgen:

```
«IF KeyType == "SYSTEM"»
  // init-method
```

```
  private void init() {
    long time;
    time = System.currentTimeMillis();
    setImplId(String.valueOf(time) + "+" +
              System.identityHashCode(this));
  }

  public «Name»PK ejbCreate() throws CreateException {
    init();
    ...

«ELSE»«REM KeyType==”USER”»
   public «Name»PK ejbCreate(
          «EXPAND Attribute::Signature FOREACH Key
                              USING SEPARATOR ", "»)
           throws CreateException {
     «FOREACH Key AS CurKey EXPAND USING SEPARATOR "\n"»
       setImpl«CurKey»(«CurKey.asPARA»);
     «ENDFOREACH»
     ...

«ENDIF»
```

Aus dem Template ist direkt ersichtlich, wie die generierte Entity-Bean-Klasse in der Implementierung aussieht. Ja nach *KeyType* werden im Konstruktor entweder die <<*Key*>>-Attribute gesetzt oder systemseitig eine ID vergeben.

Wie gezeigt, sind Änderungen und Erweiterungen an architektonischen Aspekten hier an einer zentralen Stelle, nämlich an der generativen Softwarearchitektur, durchzuführen und nicht an vielen verteilten Stellen im Code der Anwendung(en).

3.3.6. Der Rand des Infrastrukturcode-Rahmens

Bis hierhin wurde gezeigt, welche Aufgaben im Architektur-Entwicklungsstrang liegen und wie der Infrastrukturcode-Rahmen definiert wird. Aber wie werden nun die fachlichen Inhalte dieses Rahmens implementiert und wie bleiben diese bei Neugenerierung und strukturellen Änderungen erhalten? Zum Integrieren von generiertem Infrastrukturcode und manuell erstellten Businesscode gibt es wie gesagt verschiedene Ansätze (vgl. <Ref Code>).

Der hier verwendete MDSD-Generator bietet den Ansatz der geschützten Bereiche (Protected Regions), d.h. in den Templates können Codebereiche ausgezeichnet werden, in denen die Entwickler die Fachlogik implementieren. Um den Code bei erneuten Generierungsläufen zu erhalten, ist es notwendig, die Bereiche eindeutig auszuzeichnen. Hierzu stehen den Templates konstante IDs im Java-Metamodell (die UUID aus dem XMI-Output des UML-Tools) zur

Verfügung. Die Definition eines geschützten Bereichs im Template könnte wie folgt aussehen:

```
«PROTECT CSTART "//" CEND "" ID Id"Operation_MethodBody"»
  //add custom initialization here ...
«ENDPROTECT»
```

Im Code ergibt sich daraus der im nächsten Listing gezeigte eindeutig ausgezeichnete Bereich mit der entsprechenden UUID.

```
// PROTECTED REGION ID(12aaaeOperation_MethodBody) START
  ReservierungsParameterValueObject vo = null;
  try {
    StattAutoModuleComponent component =
        new StattAutoModuleComponentImpl();
     ...
  } catch ( … ) { … }
  return vo;
// PROTECTED REGION END
```

3.3.7. Strukturierung von Metaprogrammen

Die vorgestellten Templates bilden zusammen mit den Properties des in Java implementierten Metamodells eine mögliche Implementierungstechnik für MDSD-Transformationen – in diesem Fall auf zwei Sprachen verteilt. Es handelt sich dabei um Metaprogramme, denn sie dienen der Erzeugung von Programmen. Metaprogramme sind aber insbesondere auch Programme. Das heißt, auch auf dieser Ebene entsteht in realen Projekten Software, die strukturiert werden muss, damit sie iterativ und inkrementell wachsen kann. Auch hier werden also Mechanismen, wie sie aus der Objektorientierung bekannt sind, benötigt. Zum Beispiel ist eine Komponentenbildung wünschenswert. Eventuell entsteht der Bedarf, die Komponente zur Generierung der Entitätsschicht zu wechseln, um eine Migration von EJB 1.1 auf EJB 2.0 zu automatisieren. Auch Vererbung und Polymorphismus sind auf dieser Ebene nützliche Verbündete. Hier zeigen sich nicht zuletzt die Qualitäten des jeweiligen MDSD-Tools. Mehr dazu im Kapitel über Tools sowie in der zweiten Fallstudie in Kapitel <Ref CSII>.

3.4. Fazit und Ausblick

Die Praktikabilität des OMG/MDA-Ansatzes stößt zum Teil auf Skepsis, die möglicherweise nicht ganz unbegründet ist. Es gibt nicht wenige, die MDA für eine „Disziplin für Theoretiker“ halten. Die hier von uns vorgestellte pragmatische Version der architekturzentrierten,

modellgetriebenen Softwareentwicklung (AC-MDSD) hingegen hat sich in der Praxis bereits seit mehreren Jahren und in diversen Projekten unterschiedlicher Größe bewährt. Der Ansatz wird mittlerweile von den so genannten „Early-Adopters“ selbstständig produktiv eingesetzt.

Mancher verbindet mit der Einführung generativer Ansätze eine Einschränkung persönlicher Freiheiten oder gar die „Gängelung“ durch die „Inhaber“ der Generatoren. Solche Vorbehalte entstehen typischerweise auf Grund schlechter Erfahrungen mit CASE-Ansätzen oder durch fehlende oder falsche Informationen. Das Vorgehen selbst erwartet wie gesagt 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. Die Rollenbesetzung selbst liegt in der Verantwortung des Teams oder des Projektmanagements.

Maßgeblich für den Erfolg des Einsatzes von AC-MDSD ist neben der geeigneten Methodik die Verfügbarkeit von Werkzeugen, welche die Verwendung der benötigten Konzepte unterstützen. Diese Unterstützung ist aus unserer Sicht noch nicht optimal. Insbesondere eine bessere Unterstützung seitens der UML-Tools durch verteilte Modellierung, Profilierung, Generator-Integration und OCL-Constraint-Unterstützung auf der Metaebene wären wünschenswert. Viel versprechend aber sind Bestrebungen, z.B. Debugging oder Traceability auf Metaebene durch MDSD-Generatoren bereitzustellen.

Architekturzentriertheit ist nicht die einzige MDSD-Variante. So fokussieren Profile für spezifische fachliche Domänen zwar sehr viel kleinere Anwendungsgebiete, ihr Generierungspotenzial ist dafür aber in der Regel deutlich höher (oft 100%). Diesem weiter gefassten Thema widmen wir uns unter anderem in den weiteren Teilen des Buches. Es sei aber schon an dieser Stelle darauf hingewiesen, dass die in diesem Kapitel vorgestellten Tools natürlich auch dafür einsetzbar sind, zumal sie keine Abhängigkeit auf eine konkrete Domäne besitzen.

4. Begriffsbildung

Wie wir zum Teil bereits gesehen haben und in diesem Kapitel noch sehen werden, gibt es verschiedene Ansätze, Software modellgetrieben zu entwickeln. Jeder Ansatz bringt auch seine eigene Terminologie mit, was zu einem Großteil durch die unterschiedlichen Intentionen bedingt ist. Dieser unvermeidliche Umstand ist zwar für die Praxis nicht unbedingt kritisch, aber er kann zu Verwirrung führen und die Kommunikation erschweren. Wir wollen in diesem Kapitel daher einen gemeinsamen, konzeptionellen Überbau entwickeln, also eine vereinheitlichte MDSD-Terminologie. Anschließend führen wir die konzeptionelle Abbildung der drei wichtigsten MDSD-Varianten durch. Wir sind der Auffassung, dass diese Begriffsbildung für ein tieferes Verständnis der Materie und insbesondere als Fundament für die weiteren Kapitel dieses Buches mindestens hilfreich ist.

4.1. Allgemeine MDSD-Konzepte und Terminologie

Einzelne Techniken, Teilbereiche oder spezifische Ausprägungen modellgetriebener Softwareentwicklung sind keineswegs neu. Begriffe wie Generative Programming, Domain-Specific Modeling, Product Line Engineering und insbesondere natürlich Codegenerierung sind schon lange etabliert – allerdings mit stark unterschiedlichem Bekanntheitsgrad. Mit der MDA-Initiative startete die OMG einen Standardisierungsprozess für gewisse Kernkonzepte, allerdings mit einem Fokus, der primär auf Interoperabilität von Portabilität zielt. Die MDA erlangte schnell einen vergleichsweise hohen Bekanntheitsgrad und *überschattete* damit die erwähnten Techniken in gewisser Weise – ohne sie aber inhaltlich vollständig zu *überdecken*. Wir, die Autoren, sehen daher die Notwendigkeit und unternehmen den Versuch, einen einheitlichen Gesamtkontext inklusive Terminologie herzustellen. Diesen

konzeptionellen Überbau bezeichnen wir eben als *Model-Driven Software Development (MDSD)* – modellgetriebene Softwareentwicklung. Dabei fußen wir soweit es uns möglich und sinnvoll erscheint auf der Standard-Nomenklatur der OMG.

Wir entwickeln die allgemeinen Konzepte und ihre Beziehungen untereinander (d.h. den MDSD-Konzeptraum) in Form eines statischen UML-Modelles, welches schrittweise erweitert und verfeinert wird.

4.1.1. Modellierung

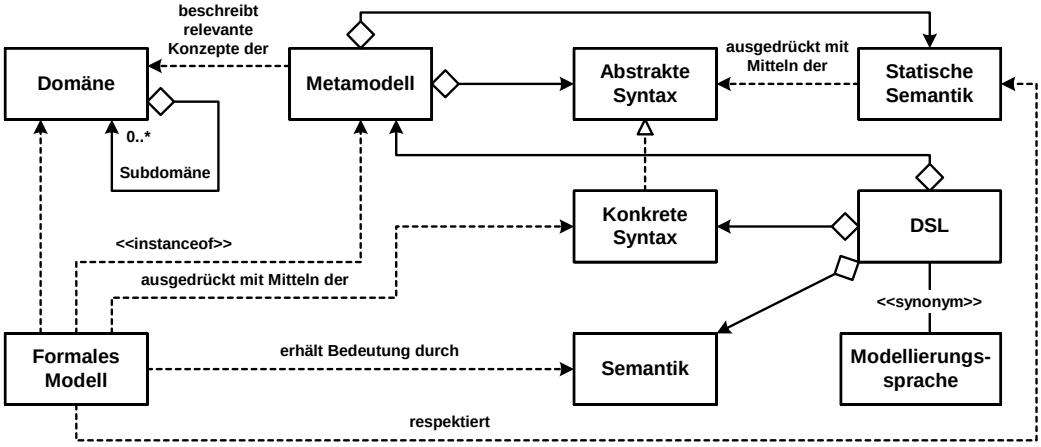


Abb. 22 *Begriffsbildung: Modellierung und DSLs*

Domäne

Ausgangspunkt der Modellierung bei MDSD ist stets eine *Domäne*. Es handelt sich dabei um ein *begrenztes Interessen- oder Wissensgebiet.* Um dieses Wissen aufnehmen und verarbeiten zu können, ist es nützlich, eine Ontologie der Konzepte einer Domäne zu erstellen. Domänen können sowohl technisch als auch fachlich motiviert sein, wenn man diese Unterscheidung überhaupt treffen mag. Das Fallbeispiel aus Kapitel <Ref CSI> beispielsweise liegt in der Domäne „Architektur für Business-Software“, denn sie enthält Konzepte wie *Entity, SystemUseCase, Controller* und *Presentation*. Eine „fachliche“ Domäne könnte „Versicherungen“ sein, mit Konzepten wie *Versicherungsprodukt, Tarif, Schaden, Leistung, Versicherungsnehmer, Vertrag* usw. Weitere Beispiele für Domänen wären „Embedded-Systeme“, „EAI“ oder „Astronomie“. Selbstverständlich können sich Domänen aus kleineren *Subdomänen* zusammensetzen. Diese lassen sich in zwei Arten unterscheiden:

Technische Subdomänen beschreiben einzelne Teile oder Aspekte des Gesamtsystems, für die eine spezialisierte Modellierungssprache nützlich ist. Typische Beispiele wären GUI-Layout oder Persistenz.

Des Weiteren lässt sich ein großes System natürlich in Partitionen, also inhaltliche Teile, zerlegen. Im obigen Versicherungs-Beispiel könnte man Partitionen für einzelne Sparten (Produkttypen) definieren wie „Leben“, „Kfz“, „Haftpflicht“ usw.

Metamodell

Im Kontext von MDSD ist es zwingend notwendig, sich über die Struktur der Domäne (eben die Ontologie) soweit im Klaren zu sein oder zu werden, dass man diese Struktur – respektive den relevanten Teil – formalisieren kann. Denn dies bildet die Grundlage für jegliche Automation. Die Formalisierung findet dann in Form eines *Metamodelles* statt. Das UML-Profil in <Ref CSI::Profil> ist ein Beispiel für ein solches Metamodell. Es spezialisiert das Basis-UML-Metamodell bezüglich der relevanten Konzepte der Domäne. Das Metamodell ist jedoch im allgemeinen Fall nicht notwendigerweise UML-basiert (vgl. <Ref PLE>).

Das Metamodell umfasst die *abstrakte Syntax* und die *statische Semantik* einer Sprache (s.u.)

Abstrakte und konkrete Syntax

Während die *konkrete Syntax* einer Sprache wie z.B. Java spezifiziert, welche Texte (oder ganz allgemein „Eingaben“) ein Parser für diese Sprache akzeptiert, so spezifiziert die *abstrakte Syntax* lediglich, wie die Struktur der Sprache aussieht. Von Details wie z.B. der Schreibweise von Schlüsselwörtern wird dabei eben abstrahiert.

Daher kann man sagen, dass eine konkrete Syntax die Realisierung einer abstrakten Syntax ist. Interessant dabei ist die Tatsache, dass verschiedene konkrete Syntaxformen eine gemeinsame abstrakte Syntax besitzen können. Andersherum: Man kann das Metamodell einer Domäne in verschiedenen Notationen ausdrücken, also zum Beispiel eine grafische auf UML basierende und eine textuelle.

Technisch gesehen wird die abstrakte Syntax einer Sprache typischerweise vom Parser instanziiert, d.h. sie wird vom Compiler verwendet, um die Eingabe (d.h. den Programm-Quelltext) im Heap des Compile-Prozesses zu repräsentieren, um sie weiterverarbeiten zu können. Dieses Paradigma kennen wir aus der XML-Welt: Ein XML-Dokument ist in der konkreten Syntax von XML formuliert, ein

generischer XML-Parser erzeugt daraus eine Repräsentation im Hauptspeicher – den DOM-Tree (DOM = Document Object Model). Das DOM selbst ist also die abstrakte Syntax von XML. Ein anderes Beispiel ist die UML: Sie besitzt eine grafische Notation mit Kästchen und Pfeilen als konkrete Syntax, die abstrakte Syntax enthält Konstrukte wie *Class, Attribute, Operation, Association, Dependency* usw. und Beziehungen zwischen diesen Konstrukten.

Nun stellt sich natürlich die Frage, wie man denn die abstrakte Syntax bzw. das Metamodell einer Domäne konkret spezifizieren bzw. notieren kann. Zu diesem Zweck existiert üblicherweise ein Metametamodell. Im Kontext der OMG-Standards ist dieses die MOF (vgl. <Ref MM>), die Meta Object Facility. Metamodelle lassen sich daher mit dieser „Urform" beschreiben. UML-Profile stellen in diesem Sinne also lediglich einen Spezialfall dar. Anders ausgedrückt: MOF bietet *eine mögliche* konkrete Syntax zur Spezifikation von Metamodellen.

Statische Semantik

Die *statische Semantik* einer Sprache legt die Wohlgeformtheitskriterien fest. Ein typisches Beispiel aus der Welt der Programmiersprachen ist die Regel, dass Variablen deklariert werden müssen. Die Syntax der Sprache (abstrakt und konkret) kann dies typischerweise nicht festlegen, d.h. der Parser erkennt einen diesbezüglichen Fehler nicht, erst die statische Analyse des Compilers schlägt fehl[24].

In Kapitel <Ref CSI> haben wir bereits gesehen, wie die statische Semantik von UML-Profilen formal definiert werden kann: Mit OCL-Ausdrücken, die sich auf die abstrakte Syntax der Sprache (d.h. die Klassenstruktur des Profils) abstützen.

Im Kontext von MDSD ist die statische Semantik von besonderer Bedeutung, denn sie dient der Detektierung von Modellierungsfehlern im Sinne der formalisierten Domäne. Idealerweise wird sie also bereits zur Modellierungszeit geprüft.

Domänenspezifische Sprache

Nun besitzen wir die Konzepte, um den Begriff der *domänenspezifischen Sprache (DSL = Domain Specific Language)* zu fassen: Eine DSL dient dem Zweck, die Schlüsselaspekte einer Domäne – nicht etwa sämtliche denkbaren Inhalte – formal ausdrückbar (modellierbar) zu machen. Dazu

[24] Konzeptionell gehört die statische Semantik eigentlich eher zur Syntax als zur Semantik einer Sprache.

besitzt sie ein Metamodell (inkl. statischer Semantik) und eine korrespondierende konkrete Syntax. Das allein genügt natürlich nicht, denn es fehlt noch die (dynamische) *Semantik* der Sprache, um den Konstrukten des Metamodelles auch eine Bedeutung zu geben. Die Semantik der DSL ist aus verschiedenen Perspektiven relevant: Zum einen muss der Modellierer die Bedeutung der ihm zur Verfügung stehenden Sprachmittel kennen, um sinnvolle Modelle erzeugen zu können, zum anderen müssen automatische Transformationen auf den Modellen eben diese Semantik umsetzen. Doch dazu später mehr.

Die Semantik der DSL muss für den Modellierer entweder gut dokumentiert oder intuitiv klar sein. Dies wird dadurch erleichtert, dass die DSL Konzepte des Problemraums aufgreift, so dass ein Domänenexperte in ihnen seine Sprachwelt wiederfindet[25].

Als Synonym zu DSL wird oftmals der Begriff der Modellierungssprache verwendet. Wir ziehen jedoch den Begriff der DSL vor, weil hier deutlicher wird, dass wir uns immer im Kontext einer Domäne bewegen.

DSLs können verschieden mächtig und auch verschieden komplex sein. Einfache textuelle Konfigurationsoptionen mit Gültigkeitsprüfungen können eine DSL bilden. Am anderen Ende des Spektrums stehen z.B. grafische DSLs mit entsprechenden domänenspezifischen Editoren.

Es gibt zwei Klassen von DSL-Editoren: Generische Werkzeuge wie z.B. UML-Tools, die mit einem Profil konfiguriert werden, und domänenspezifische Spezial-Werkzeuge.

Formale Modelle

Schließlich kommen wir zum Begriff des *formalen Modells* im Kontext von MDSD. Formale Modelle bilden wie gesagt den Ausgangspunkt für automatisierte Transformationen. Auch ein rein interpretativer Ansatz erfordert ein formales Modell. Dieses benötigt eine DSL und steht damit offensichtlich im Zusammenhang mit der entsprechenden Domäne. Es ist in der gegebenen konkreten Syntax formuliert und ist mindestens konzeptionell (meist auch technisch) eine Instanz des gegebenen Metamodelles.

Ein Modell ist also ein in der DSL formulierter „Satz“. Seine Bedeutung erhält es durch die Semantik der DSL. Wir sehen, der Kontext der Domäne ist bei MDSD von zentraler Bedeutung.

---

[25] Im architekturzentrierten Fall ist der Domänenexperte also ein Softwarearchitekt, da die Domäne „Softwarearchitektur“ lautet.

Betrachten wir nun ein paar Beispiele:

- Die architekturzentrierten Designs aus der vorangegangenen Fallstudie sind formale Modelle im Sinne von MDSD. Die Domäne lautet „Architektur für Business-Software“.
- Ein Java-Programm ist eine Instanz der Programmiersprache Java bzw. des zugehörigen Metamodelles. Java besitzt auch eine Semantik. Doch was ist die Domäne von Java? Man kann sagen „Turing-berechenbare Funktionen“, also im Prinzip „alles was man mit Computern machen kann“. Dasselbe gilt für Executable UML (eine direkt ausführbare Variante von UML, siehe <Ref MDA>). Im Kontext von MDSD ist eine solche Domäne aber wenig hilfreich, weil der Ansatz ja gerade darin besteht, Konzepte eines höherwertigen Problemraums zu formalisieren und sich damit vom Abstraktionsniveau einer Programmiersprache abzuheben.
- Eine Powerpoint™-Folie ist per se kein formales Modell im Sinne von MDSD, denn es gibt zwar ein sehr generisches Metamodell (*Rechteck, Pfeil, Ellipse, Text* etc.), aber eben keine Semantik für solche Modelle. In dem Moment allerdings, wo man eine echte DSL mit einer Powerpoint-basierten, konkreten Syntax definiert, wäre es durchaus möglich, entsprechend wohlgeformte Powerpoint-Folien als formale Modelle weiterzuverarbeiten und daraus z.B. Code für grafische Benutzeroberflächen (GUI) zu erzeugen.

4.1.2. Plattformen

Wir wollen unsere Ontologie der Domäne MDSD nun um die nächste Partition erweitern: Mit Hilfe der DSL bilden wir wie gesagt den Problemraum ab. Formale Modelle im Sinne der DSL können dann weiterverarbeitet (transformiert) oder interpretiert werden. Dazu benötigen wir auf der anderen Seite – im Lösungsraum – etwas, das die Transformation respektive die Interpretierung stützt, also etwas, auf das der durch die Transformation erzeugte Code aufbaut.

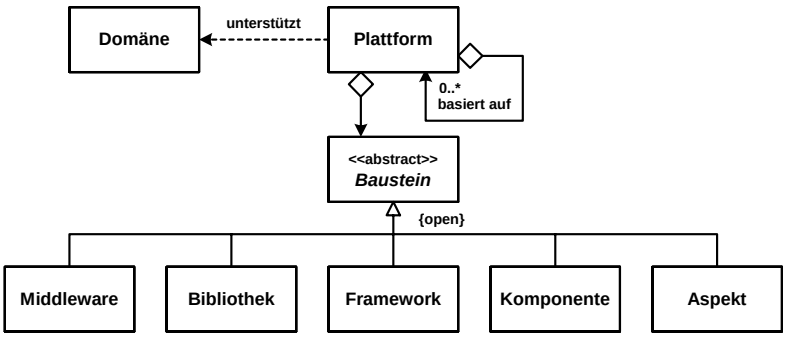


Abb. 23 *Begriffsbildung: Plattformen*

Plattform

Der Begriff der *Plattform* wird sowohl im MDA-Kontext (vgl. Abschnitt <Ref Basis> und <Ref MDA>) als auch beim Software Product Line Engineering (vgl. Kapitel <Ref PLE>) verwendet. Er ist allgemein genug, um bei der Beschreibung von MDSD nützlich zu sein. Die Plattform hat die Aufgabe, die Umsetzung der Domäne zu „stützen“. Das heißt, die Transformation von formalen Modellen soll möglichst einfach werden. In der Fallstudie (vgl. Kapitel <Ref CSI>) haben wir als Plattform J2EE mit Apache-Struts sowie einigen selbst erstellten Basis- und Hilfsklassen verwendet. Die DSL der Domäne (also das verwendete architekturzentrierte UML-Profil) beschreibt den Problemraum (*Entity*, *Controller*, *Presentation* etc.), aber nicht den Lösungsraum, d.h. die Plattform. Offensichtlich sind die Transformationen umso einfacher herzustellen, je mächtiger die Plattform ist. Würden wir Struts und unsere Hilfsklassen aus der Plattform entfernen, so wäre der Aufwand, die dynamischen Konstrukte unserer DSL (z.B. Aktivitätsdiagramme) im Rahmen der Codegenerierung abzubilden, um ein Vielfaches höher.

Plattformen können kaskadieren (vgl. Kapitel <Ref CSII>).

Im Extremfall eines interpretativen Ansatzes nimmt die Plattform die Rolle einer virtuellen Maschine (d.h. eines Interpreters) für ausführbare Modelle ein, so dass die Modell-Transformation trivial ist.

Baustein

Eine Plattform kann sich auf vorhandene *Bausteine (Building Block)* stützen. Dies können z.B. Middleware, Bibliotheken (Library), Frameworks, Komponenten (Component) oder Aspekte (Aspect) im Sinne von AOSD/AOP (Aspect-Oriented Programming [Lad03]) sein.

4.1.3. Transformationen

Nachdem wir Modellierung und Plattformen betrachtet haben, können wir diese beiden Partitionen in unserem Konzeptraum verbinden:

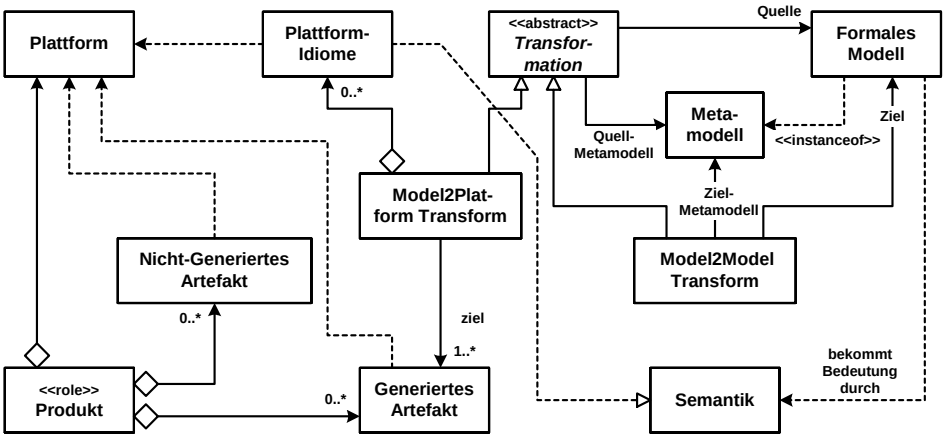


Abb. 24 *Begriffsbildung: Transformationen*

Transformation

MDSD-*Transformationen (Transformation)* basieren immer auf einem Quell-Metamodell, da das zu transformierende Quell-Modell eben gerade eine Instanz dieses Metamodelles ist. Die Transformationsvorschriften können daher nur an den Konstrukten des Metamodelles festgemacht sein, und genau darin liegt sein Hauptzweck. Letztlich implementieren die Transformationen die Semantik der DSL.

Wir unterscheiden zwischen Modell-zu-Modell-Transformationen (*Model2ModelTransform*) und Modell-zu-Plattform-Transformationen (*Model2PlatformTransform*), oft auch als Modell-zu-Code-Transformation bezeichnet.

Eine Modell-zu-Modell-Transformation erzeugt wiederum ein Modell. Dieses kann allerdings auf Basis eines anderen Metamodelles formuliert sein als das Quell-Modell. Derartige Transformationen beschreiben also im Allgemeinen, wie die Konstrukte des Quell-Metamodelles auf die Konstrukte des Ziel-Metamodelles abzubilden sind. Die MDA beschreitet mit der Query/Views/Transformations-Spezifikation diesen Weg (vgl. Abschnitt <Ref MDA>).

Eine Modell-zu-Plattform-Transformation „kennt“ die Plattform und generiert Artefakte *(Generiertes Artefakt)*, die sich auf die Plattform stützen können. Generierter Sourcecode, der sich in ein gegebenes Framework einfügt, wäre ein Beispiel. Bei dieser Klasse von

Transformationen ist ein Ziel-Metamodell nicht notwendig, da es sich hier in der Regel um reine Makroexpansionen handelt. Die Template-Definitionen in Kapitel <Ref CSI> fallen in diese Kategorie.

Plattform-Idiome

Die Tatsache, dass Modell-zu-Plattform-Transformationen das komplette Wissen über die Plattform verwenden können, gibt ihnen ein mächtiges Mittel an die Hand. *Plattformspezifische Idiome (Platform Idioms)* können von den Transformationen vollkommen transparent verwendet werden. In der Fallstudie (vgl. Kapitel <Ref CSI>) beispielsweise haben wir Code generiert, der das *Business Delegate Pattern* verwendet, um die Fachlogik frei vom EJB-Programmiermodell zu halten. Die Verwendung des Patterns musste allerdings nirgends im Design notiert werden – das Wissen darum, an welchen Stellen und wie das Pattern anzuwenden ist, steckt allein in den Transformationen.

Produkt

MDSD hat das Ziel, aus formalen Modellen durch eine oder mehrere Transformationen ein Software-*Produkt* ganz oder teilweise herzustellen. Das Produkt kann eine ganze Anwendung oder auch nur eine Komponente, die möglicherweise an anderer Stelle wiederum als Baustein Verwendung findet, sein. Ein solches Produkt im Sinne von MDSD aggregiert die Plattform, generierte und gegebenenfalls auch *nicht-generierte Artefakte*. Nicht-generierte Artefakte können z.B. anwendungsspezifische Hilfsklassen oder manuell programmierte Fachlogik sein.

4.1.4. Software-Systemfamilien

Die nächste MDSD-Partition betrachtet die Zusammenhänge zwischen den Produkten einer Domäne und bringt den Wiederverwendungsaspekt ins Spiel.

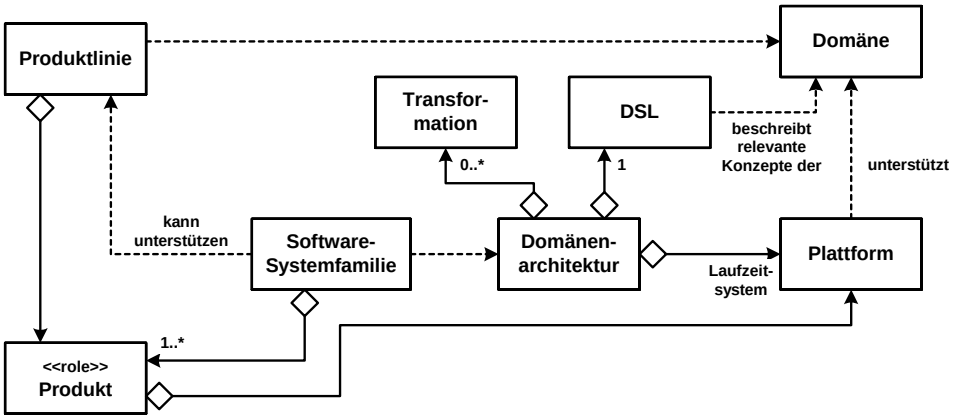


Abb. 25 *Begriffsbildung: Domäne, Produktlinie, Software-Systemfamilie*

Domänenarchitektur

Das Metamodell einer Domäne, eine Plattform und dazu passende Transformationen inklusive der zugehörigen Idiome stellen das notwendige Werkzeug, um vom Modell zum Produkt zu kommen – und zwar ganz oder teilweise automatisiert. Die Aggregation dieser Dinge bezeichnen wir allgemein als *Domänenarchitektur* - das zentrale MDSD-Konzept. Im Gegensatz zur Architektur der Plattform legt eine Domänenarchitektur also fest, welche Konzepte formal unterstützt werden sollen (nicht unbedingt die konkrete Syntax) und wie diese auf eine gegebene Plattform abzubilden sind. Die Plattform spielt dabei die Rolle des Laufzeitsystems. Eine Domänenarchitektur ist damit immer relativ zu einer Plattform. Die generative Softwarearchitektur der Fallstudie (vgl. Kapitel <Ref CSI>) ist ein Beispiel für eine Domänenarchitektur.

Software-Systemfamilie

Offensichtlich ist eine Domänenarchitektur zur Herstellung aller Produkte geeignet, die sich mit dem gegebenen Metamodell ausdrücken lassen und auf derselben Plattform realisiert sind.

Die Menge aller Produkte, die sich mit einer bestimmten Domänenarchitektur erstellen lässt, bezeichnet man allgemein als *Software-Systemfamilie (Software System Family)*. Anders ausgedrückt: Die Software-Systemfamilie nutzt für ihre Umsetzung eine Domänenarchitektur und die Domänenarchitektur ist für alle Produkte der Software-Systemfamilie wiederverwendbar. Die Domänenarchitektur

muss also so flexibel sein, dass sie die Unterschiede (Variabilitäten) der verschiedenen Produkte der Software-Systemfamilie abbilden kann.

Produktlinie

Eine Produktlinie ist eine Menge fachlich aufeinander abgestimmter Einzelprodukte. Die Produkte der Produktlinie können aus Sicht des Anwenders Alternativen darstellen (also in verschiedenen, aber verwandten Kontexten einsetzbar sein) oder sich inhaltlich ergänzen (und damit praktisch eine „Suite“ definieren). Wichtig ist dabei zu beachten, dass die Produkte einer Produktlinie nicht zwangsläufig irgendwelche technische Gemeinsamkeiten haben müssen. Einer Produktlinie kann eine Software-Systemfamilie zu Grunde liegen, muss es aber nicht.

4.2. Abbildung der MDA-Konzepte

Den Inhalten, der Ausrichtung und den Trends der MDA ist das Kapitel <Ref MDA> gewidmet. In diesem Abschnitt geht es vielmehr darum, diesen Standard nun konzeptionell in den zuvor dargestellten allgemeinen Rahmen der MDSD einzuordnen.

Abb. 26 zeigt die Einordnung von MDA in den Konzeptraum von MDSD:

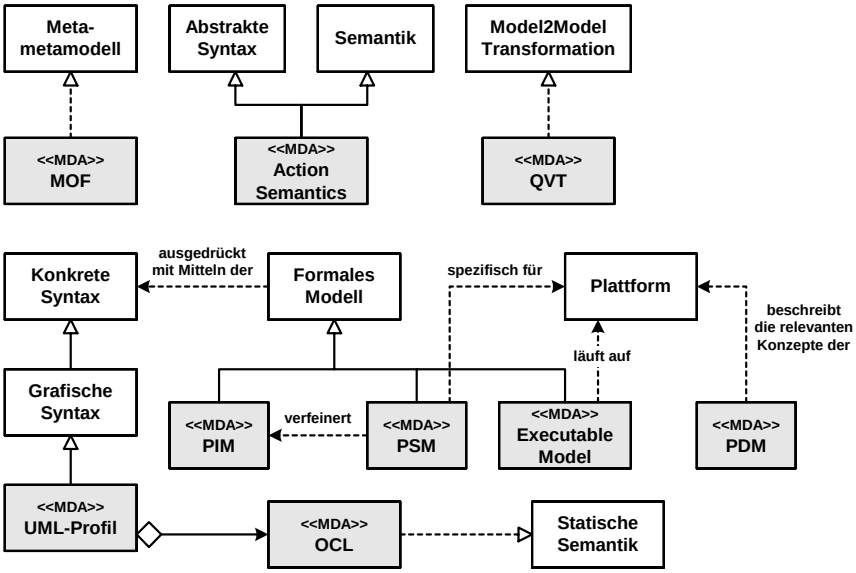


Abb. 26 *Begriffsbildung: Einordnung der MDA-Konzepte*

MDA ist bezüglich der Ontologie eine Spezialisierung von MDSD mit folgenden Eigenschaften:

- Software-Systemfamilien und Produktlinien haben in der MDA-Terminologie keine direkte Entsprechung und stehen auch inhaltlich nicht im Fokus.
- MDA verwendet MOF als das Metametamodell (d.h. als Mittel, um Metamodelle zu definieren).
- MDA sieht vor, dass die DSLs auf Basis der MOF definiert werden. Es sind also beliebige Notationen und Metamodelle möglich, solange sie mit Hilfe des OMG-Metametamodells definiert wurden. In der Praxis regt MDA die Verwendung von UML-Profilen als konkrete Syntax für eine DSL an. Damit ist die DSL im Kern dann auf UML festgelegt. Die statische Semantik wird dementsprechend mit OCL-Ausdrücken spezifiziert.
- Es werden verschiedene Sichtwinkel auf formale Modelle definiert: Ein fachliches Modell kann relativ zu einer Plattform spezifisch (PSM) oder unspezifisch (PIM) sein. Die MDA legt mehrschrittige Transformationen zwischen Modellen nah, verbietet aber auch eine direkte PIM-zu-Code-Transformation nicht.
- Um auch die „letzte“ Transformation, also die zur Plattform hin, als Modell-zu-Modell-Transformation notieren zu können, muss auch die Plattform mittels eines Metamodells beschrieben werden. Dazu werden die PDMs, die Platform Description Models, verwendet.
- Es gibt derzeit noch keine standardisierte Transformationssprache, aber ein Request for Proposal für Query/Views/Transformations (QVT). Ziel ist es, Transformationen zwischen Quell- und Ziel-Metamodell für Modell-zu-Modell-Transformationen zu beschreiben.
- Ausführbare UML-Modelle (Executable UML, siehe Kapitel <Ref MDA>) stellen eine Besonderheit und für viele MDA-Vertreter eine Zielvorstellung von MDA dar: Sie sind entweder mehr oder weniger direkt auf einer entsprechend mächtigen und generischen Plattform ausführbar – d.h. sie werden von einer virtuellen UML-Maschine interpretiert oder mittels Transformationen vollständig kompiliert, so dass sie auf einer „niedrigeren“ Plattform ausführbar werden. Im Gegensatz zu einer fachlich fokussierten Domäne wie „Versicherungswesen“ haben wir es in diesem Falle

also mit einer „Domäne“ zu tun, die der Ausdrucksmächtigkeit einer Programmiersprache entspricht. Ein UML-Profil ist daher nicht notwendig. Offensichtlich ist dadurch aber auch die Distanz zu einem fachlichen Problemraum deutlich größer.

- Die Action Semantics der UML (vgl. <Ref MDA>) ist ein essenzieller Baustein für Executable UML, denn sie bietet die Möglichkeit, in abstrakter Form Algorithmen zu spezifizieren. Unter Verwendung einer (toolspezifischen, weil nicht standardisierten) konkreten Syntax kann man damit programmieren wie in anderen Programmiersprachen auch, allerdings unter Einbeziehung der Informationen im statischen Modell <vgl. Ref MDA>.

4.3. Abbildung der AC-MDSD-Konzepte

Architekturzentrierte, modellgetriebene Softwareentwicklung (AC-MDSD = Architecture Centric MDSD) ist ein praktisches Schwerpunktthema dieses Buches, und wir sind nun in der Lage, die Terminologie einzuordnen.

Abb. 27 zeigt die Einordnung des Ansatzes in den zuvor erarbeiteten allgemeinen Kontext von MDSD:

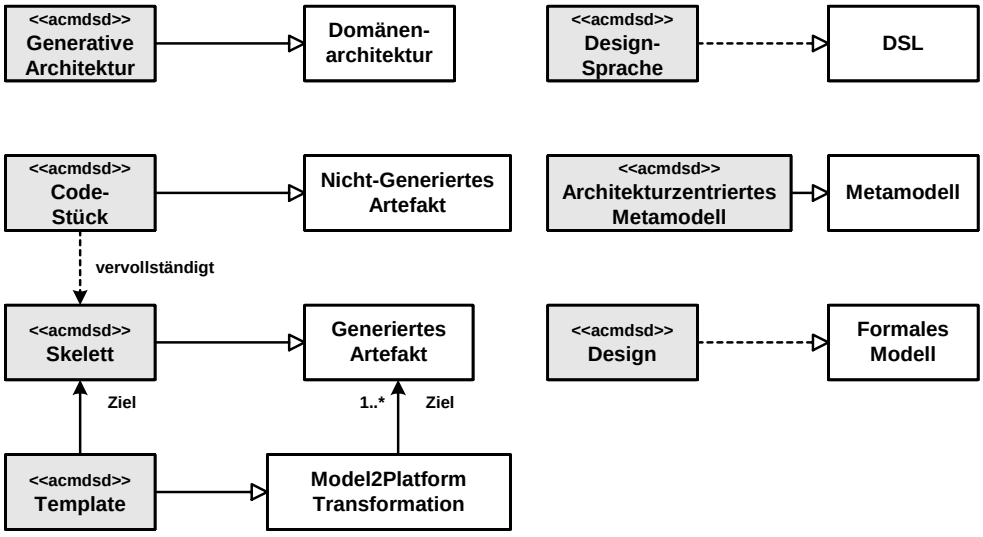


Abb. 27 *Begriffsbildung: Einordnung der AC-MDSD-Konzepte*

AC-MDSD ist eine Spezialisierung von MDSD, die konzeptionell auch einen gemeinsamen Schnitt mit MDA besitzt. Diese Variante hat folgende Eckpfeiler:

- Die Domäne ist architektonisch motiviert – z.B. „Architektur für Business Software“ oder Komponenteninfrastrukturen für Embedded-Systeme.
- Die zu erstellenden Produkte sind in der Regel ganze Anwendungen.
- Es gibt in der Außensicht (Black Box View) nur einstufige Modell-zu-Plattform-Transformationen (genauer: Modell-zu-Code-Transformationen). Diese können allerdings intern (White Box View) zwecks Modularisierung als Hintereinanderausführung mehrerer Transformationen strukturiert sein.
- Das Metamodell der DSL enthält dementsprechend möglichst abstrakte Architekturkonzepte (vgl. Abschnitt <Ref CSI>).
- Die DSL wird auch *Designsprache* genannt. Typischerweise werden UML-Profile verwendet, teils in Kombination mit weiteren, textuellen Spezifikationen.
- Die formalen Modelle werden auch *Designs* genannt.
- Die Modell-zu-Plattform-Transformation liegt typischerweise in Form von Templates vor, die eine große Ähnlichkeit zum Generat aufweisen und daher bequem aus einer Referenzimplementierung extrahiert werden können (vgl. Abschnitt <Basis::AC >).
- Die Transformation hat nicht den Anspruch, die gesamte Anwendung zu erzeugen, sondern nur einen Implementierungsrahmen, der den architektonischen Infrastrukturcode enthält (*Skelett*).
- Der nicht generierte fachliche Implementierungscode wird manuell in der Zielsprache implementiert (*Codestück*). Dazu kann der generierte Implementierungsrahmen geschützte Bereiche zur Ergänzung der Fachlogik enthalten, die bei iterativer Generierung bestehen bleibt. Alternativ dazu werden generierter und nicht-generierter Code mittels geeigneter Design-Patterns integriert <Ref Code>.
- Designsprache, Templates und Plattform bilden eine *generative Architektur*. Dabei handelt es sich offensichtlich um eine spezielle Domänenarchitektur. Diese stützt die Software-Systemfamilie.

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

4.4. Abbildung der Generative Programming-Konzepte

Da wir diesen Ansatz hier zum ersten Mal diskutieren, wollen wir natürlich nicht nur auf den Bezug zur MDSD-Ontologie eingehen, sondern auch einen kurzen Überblick liefern über die Motivation und Historie sowie den primären Fokus.

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

> *Generative Programming is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.*

Die treibenden Faktoren von GP sind:

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

- Herstellung bzgl. bestimmter Aspekte (z.B. Performance und Codegröße) optimierter Produkte

Ziel ist es also, aus einem Modell einer (formalen) Anforderungsspezifikation möglichst automatisiert genau passende und optimierte Produkte zu erstellen. Um das Ganze etwas zu verdeutlichen, eignet sich ein Blick auf das generative Domänenmodell:

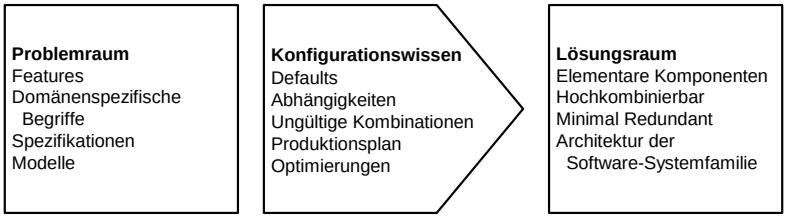


Abb. 28 *Domänenmodell der Generativen Programmierung*

Innerhalb des Problemraumes (der Domäne) wird die formale Anforderung an eine Anwendung definiert. Das kann geschehen durch verschiedene Modelle und Spezifikationen, darunter auch Feature-Modelle. Im Lösungsraum wird die betreffende Anwendung, das Produkt, durch elementare Komponenten realisiert. Diese sind möglichst gut kombinierbar und nicht redundant, was ihre Funktionalitäten angeht. Komponenten in diesem Sinne können durchaus auch Aspekte im Sinne von AOSD sein. Den Zusammenhang zwischen den beiden Bereichen stellt das Konfigurationswissen dar. Dieses umfasst unter anderem nützliche Defaults, Abhängigkeiten und illegale Kombinationen – falsch spezifizierte Produkte werden also nicht hergestellt. Außerdem enthält das Konfigurationswissen den Produktionsplan sowie mögliche Optimierungen. Es beinhaltet damit also auch den Generator.

Betrachten wir nun den Ansatz und seine Terminologie im Hinblick auf die Einordnung in MDSD in Abb. 29:

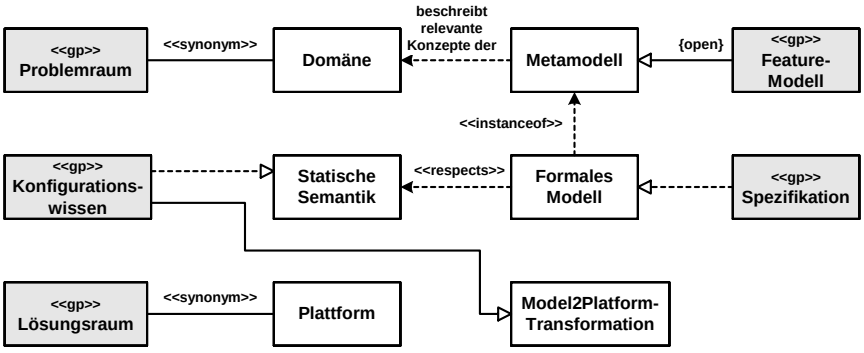


Abb. 29 *Begriffsbildung: Einordnung der Konzepte der GP*

Auch Generative Programming ist bezüglich der Ontologie eine Spezialform von MDSD mit folgenden Eigenschaften:

- Bei GP spielt der Gedanke der Software-Systemfamilie eine zentrale Rolle. Man geht davon aus, dass man eine Domäne z.B. mittels Feature-Modellen (vgl. <Ref PLE::PLE::DomA>) modelliert und darauf basierend dann einzelne Produkte erzeugen kann.
- Traditionell ist der Gedanke der (UML-)Modellierung weniger stark vorhanden. Viel mehr definiert man basierend auf der Domänenanalyse eine oft textuelle DSL, mit der Produkte der Familie beschreibbar werden.
- Oft werden Feature-Modelle als Basis für die DSL bzw. das Metamodell verwendet. Allerdings ist dies keine zwingende Vorgabe. Prinzipiell kann auch in GP jede Art von Metamodell bzw. DSL verwendet werden.
- Die konkrete Ausprägung eines Feature-Modells, also die Spezifikation des Produkts, spielt dann die Rolle des formalen Modells im MDSD-Sinne.
- Die Domäne wird auch als Problemraum bezeichnet, die Plattform und die Komponenten, aus denen das „Produkt“ zusammengesetzt wird, als Lösungsraum.
- Das Konfigurationswissen ist in einen Generator gegossen, der ähnlich wie bei AC-MDSD eine einschrittige Modell-zu-Code-Transformation durchführt. Die statische Semantik (Erkennung ungültiger Produktkonfigurationen) wird ebenfalls durch das Konfigurationswissen realisiert.
- Die Plattform besteht typischerweise aus maximal kombinierbaren und minimal redundanten Komponenten, welche letztlich die Variabilität der DSL, d.h. deren Ausdrucksmöglichkeiten abbilden.
- Oft kommen im Rahmen von GP Feature-Modellierungswerkzeuge als Modellierungswerkzeug zum Einsatz. Natürlich ist auch dieser Sachverhalt nicht zwangsläufig, je nach DSL können auch andere Tools verwendet werden. Beispielsweise ist im Rahmen von C++ Template-Metaprogrammierung die C++-IDE das Modellierungswerkzeug.

Auch wenn es durch die Definition von GP nicht zwangsläufig so vorgegeben ist, trifft man auf Grund der Betonung von „optimiert“

oftmals auf statische Generierungstechniken. Die Konfiguration von Frameworks oder das Erstellen einer virtuellen Maschine ist eher unüblich. Trotzdem ist es wichtig zu verstehen, dass GP nicht einfach nur Codegenerierung bedeutet. Auch ist GP nicht gleichzusetzen mit C++ Template-Metaprogrammierung, welches lediglich *eine* Implementierungstechnologie für GP darstellt.

Traditionell hat sich GP eher auf die Erstellung kleiner, aber hoch effizienter Produkte konzentriert. Große, verteilte Enterprise-Anwendungen bzw. -Familien standen weniger im Fokus.

5. Einordnung und Abgrenzung

Nachdem wir in Kapitel <Ref Begr> eine einheitliche MDSD-Terminologie erarbeitet haben, wollen wir in diesem Kapitel MDSD nun eine inhaltliche Einordnung respektive Abgrenzung verwandter Themen vornehmen.

5.1. MDSD vs. CASE, 4GL, Wizards

Eine bemerkenswerte Eigenschaft modellgetriebener Softwareentwicklung ist, dass nicht etwa generative und starre Entwicklungsumgebungen verwendet werden, sondern *beliebige* Ziel-Architekturen, Modellierungs- und Zielsprachen, Schnittstellen und Laufzeitkomponenten unterstützt werden können.

Ein CASE- oder 4GL-Tool hingegen gibt mindestens einen Bestandteil einer Domänenarchitektur fest vor (meist sogar alle):

- DSL (Modellierungssprache)
- Transformationen
- Plattform (und Zielarchitektur)

Auf der anderen Seite fokussieren diese Tools aber gerade *nicht* eine spezifische Domäne. Daraus ergibt sich automatisch der Anspruch „One size fits all“: Eine vorgedachte Kombination passt global für alle Eventualitäten (Anwendungen). Genau diese Annahme ist aber in der Praxis vollkommen unrealistisch und bereitet erhebliche Probleme. Typischerweise lassen sich 80% einer Anwendung so recht schnell erstellen, die letzten 20% verschlingen dafür 80% des Aufwandes, weil man auf Grund der Inflexibilität gezwungen wird, Workarounds „gegen“ das Werkzeug zu erfinden. Eigene Architekturanforderungen, Schnittstellen oder gar Domänenwissen können nicht eingebracht werden.

MDSD bedeutet die explizite Abkehr von „One size fits all"-Ansätzen. Es steht also klar die Entwicklungs*methodik* und nicht eine Entwicklungs*umgebung* im Vordergrund.

In aller Regel wird man die Dinge, die man generieren will, vorher mindestens einmal von Hand implementieren und verifizieren. Erst in einem zweiten Schritt wird man daraus eine Domänenarchitektur ableiten, die aufgrund eines Eingabemodells bestimmte Dinge automatisch generiert. Damit erübrigen sich insbesondere die mit traditionellen, generativen Ansätzen zu Recht verbundenen Fragen:

- Wie gut ist die Laufzeit-Performance des Generats?
- Welche Qualität/Lesbarkeit hat der generierte Sourcecode?
- etc.

All dies ist genauso gut (oder schlecht) wie die Referenzimplementierung, aus der man die Transformationen ableitet.

Natürlich muss nicht unbedingt bei Null angefangen werden: Sobald eine Software-Systemfamilie generativ einsetzbar ist, multipliziert sich der Nutzen mit jeder Anwendung, welche diese technologische Basis verwenden kann – d.h. ein „Mitglied" der Familie ist.

MDSD ist auch nicht zu vergleichen mit einem Code-Wizard oder Pattern-Expander. Die nützlichen Helfer in den gängigen Entwicklungsumgebungen oder die Musterexpansion einiger UML-Tools erlauben die automatische Erzeugung von Klassenskeletten z.B. für EJBs bzw. die Erzeugung von Klassenverbänden, je nach Gestalt des Musters. Dieser Schritt ist im Unterschied zu MDSD jedoch in der Regel einmalig. Die mit MDSD erreichbare wiederholte Transformation bei Erhaltung der vorgenommenen Individualisierungen fehlt. Außerdem geht die Abstraktion „verloren", die mit MDSD im Modell vorgenommen wird.

5.2. MDSD vs. Roundtrip Engineering

Unter Roundtrip Engineering (siehe Abb. 30) versteht man die Möglichkeit, sowohl in einem Modell als auch in dem daraus generierten Code nach Belieben ändern zu können, wobei die Änderungen in beide Richtungen propagiert werden und beide Artefakte immer konsistent sind. Vor allem der Weg vom Code zum Modell (Reverse Engineering) ist dabei interessant.

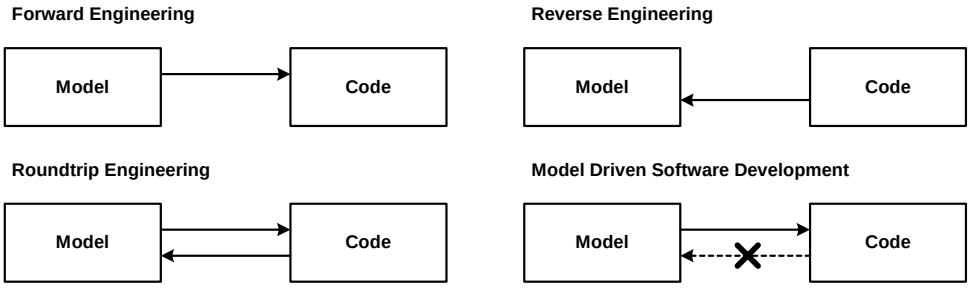


Abb. 30 *Forward/Reverse/Roundtrip Engineering und MDSD*

Typischerweise steht das Modell bei diesen Ansätzen auf demselben Abstraktionsniveau wie der Code („ein Rechteck pro Klasse“). Es handelt sich praktisch um die Visualisierung der Struktur eines Programmes. In einem solchen Szenario ist es nützlich (und praktikabel), Änderungen am Code in dem Modell automatisiert nachzuvollziehen.

MDSD hat einen anderen Ansatzpunkt: Das Modell ist deutlich abstrakter als der daraus generierte Code. Es ist daher im Allgemeinen nicht möglich, nach einer manuellen Änderung am generierten Code das Modell automatisch konsistent zu halten.

Es sollte daher vermieden werden, generierten Code manuell zu verändern. Eine klare Definition, welche Anteile generiert und welche manuell implementiert werden, ist somit nötig. Um auch ohne Roundtrip Engineering den gewünschten Code erzeugen zu können, gibt es mehrere Möglichkeiten (siehe [Fra02]):

1. Abstraktion: Das Heben des Abstraktionsniveaus von Entscheidungen auf Modellebene. Dies ist allerdings nur sinnvoll, wenn eine entsprechende Abstraktion auf Modellebene identifizierbar ist.

2. Tagging des Modells: Die Übernahme von Entscheidungen vom Code ins Modell, ohne das Abstraktionsniveau zu erhöhen. Dies wird auch als „tagging“ des Modells mit Implementierungsentscheidungen bezeichnet. Dieses Verfahren führt schnell zur Verunreinigung der Modelle mit Implementierungskonzepten, welche nicht aus dem Sprachraum (der Domäne) der Modellierer oder Fachexperten stammen, und somit zu einer potenziellen Fehlerquelle werden. Wenn Tagging des Modells verwendet wird, dann sollte dies bevorzugt durch Einführung einer technischen Subdomäne erfolgen <vgl. Trans::SubDom>, so dass Modellierer und Fachexperten von den Implementierungskonzepten verschont bleiben.

3. Code-Klassentrennung: Anpassung der Zielarchitektur, so dass handgeschriebener Code in speziell zu diesem Zweck zu erstellenden Klassen geschrieben werden muss.

4. Tagging des Codes: Der Einbau von geschützten Bereichen im Code durch die Benutzung von speziellen Tags, welche bei der Neugenerierung den zwischen den Tags liegenden Code vor Überschreiben schützen. Dies ist ein pragmatisches Verfahren zum Mischen von generiertem und handgeschriebenem Code zur Generierungszeit. Es gibt verschiedene Varianten dieses Verfahrens, bis hin zu Verfahren, welche nicht nur Einschübe von handgeschriebenem Code erlauben, sondern auch optionales Ersetzen von generierten Anweisungen ermöglichen.

Die Aufzählung spiegelt die Eleganz der Lösung wider – in strikt absteigender Reihenfolge. *Tagging des Modells* erlaubt noch eine sehr saubere Trennung von Verantwortlichkeiten und ermöglicht eine vollautomatische Neugenerierung ohne manuelle Nachbearbeitung. *Tagging des Codes* hingegen sollte mit Vorsicht eingesetzt werden, da es z.B. die Versionierung erschwert.

5.3. MDSD und Patterns

Patterns (Architektur-Patterns, Design Patterns, Idiome) haben zunächst einmal nichts mit MDSD zu tun. Patterns sind dokumentierte Best Practices zur Lösung bestimmter wiederkehrender Probleme.

Die Beziehung zu MDSD rührt daher, dass die Transformationen ja eine Form von „formalisierten Best Practices“ sind. Insofern werden durch Transformationen oft Strukturen im Zielmodell (bzw. dem generierten Code) erstellt, die der Struktur der Lösung eines Patterns entsprechen. In der Abbildung unten wird die Abhängigkeit zwischen den GUIs und den Entitäten beispielsweise auf die Implementierung des Observer Patterns abgebildet (siehe Abb. 31).

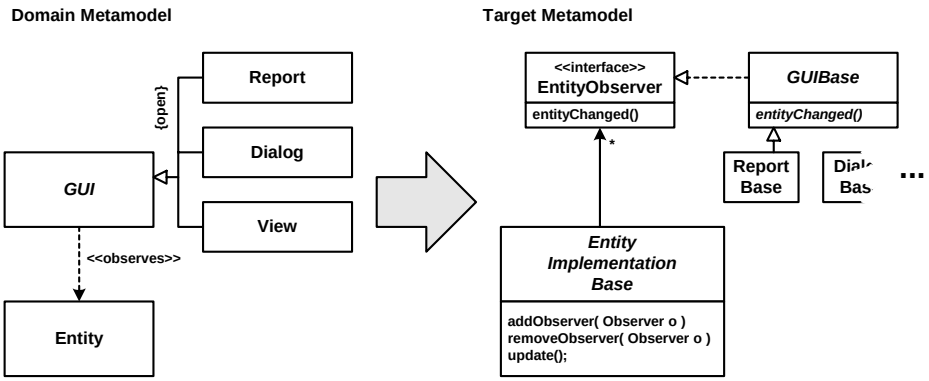


Abb. 31 *Verwendung von Patterns in Transformationen*

Es ist dabei allerdings wichtig zu verstehen, dass ein Pattern nicht nur aus dem UML-Diagramm der Lösung besteht! Wesentliche Bestandteile eines Patterns sind die Erläuterung, welche Randbedingungen die Lösung eines Patterns beeinflussen, wann ein Pattern einzusetzen ist (und wann nicht), sowie die Konsequenzen der Verwendung des Patterns. Auch gibt es bei vielen Patterns diverse Variationen, die alle andere Vor- und Nachteile haben. Ein im Rahmen einer Transformation implementiertes Pattern berücksichtigt diese Aspekte nicht – diese müssen vom Entwickler der Transformationen berücksichtigt, abgewogen und entschieden werden.

Eine MDSD-Transformation kann also dazu dienen, Lösungsstruktur (sowie das damit verbundene Verhalten) in ein Modell oder Code hineinzugenerieren. Die Abwägung, ob und wie ein Pattern eingesetzt wird, obliegt weiterhin dem Entwickler – allerdings dem Entwickler der Transformation, nicht dem Entwickler, der die Transformation im Rahmen der Anwendungsentwicklung einsetzt.

Einige UML-Tools und auch die MDA (genauer das EDOC-Pattern Profile [<Lit EDOC>]) bezeichnen toolgestützte „Makrodefinitionen“ auf UML-Ebene zur Kompaktifizierung von Modellen als „Pattern“. Dies ist allerdings irreführend, zumal ein echtes Pattern, wie gerade erläutert, eben weit mehr ist als nur ein UML-Makro. Zudem können diese „Pattern“ bei vielen Tools auch nur einmal expandiert werden, so dass die Kompaktifizierung anschließend verloren ist.

5.4. MDSD und Agile Softwareentwicklung

Ein iterativ-inkrementelles Vorgehen ist ein starker Verbündeter für MDSD, und striktes Timeboxing hilft, die Feedback-Schleifen zwischen dem Architektur-Entwicklungsstrang und dem Anwendungs-

Entwicklungsstrang reibungslos zu implementieren. Eine der höchsten Prioritäten – auch bei MDSD – ist die Entwicklung lauffähiger Software, welche von Stakeholdern und Endbenutzern validiert werden kann. MDSD beinhaltet eine Reihe von Techniken und Verfahren, welche es ermöglichen, die Prinzipien agiler Softwareentwicklung in großen Industrieprojekten einzusetzen. Diese Techniken stützen agiles Requirements Management und die regelmäßige Validierung von Software-Under-Construction. Diese Themen werden in Kapitel <Ref Proz> vertieft.

Es ist nicht das Ziel von MDSD, eine bestimmte (agile) Methode vorzuschreiben. Solange die wenigen, aber strikten MDSD-Regeln für iterative Softwareentwicklung eingehalten werden, können die Mikroaktivitäten des Entwicklungsprozesses durch eine beliebige agile Methodik bestimmt werden. In der Praxis ergibt sich die Rollenverteilung in agilen Teams aus den Stärken und Fähigkeiten der Einzelpersonen und nicht aufgrund fest vorgegebenen starren Job-Beschreibungen.

Die Betonung von MDSD liegt auf Modellen, diese haben den Stellenwert von Quellcode und nicht etwa den Stellenwert von optionaler Dokumentation. Die „Produktion“ oder Generierung eines Systems mittels einer Domänenarchitektur ist genauso automatisiert wie die Kompilierung einer 3GL-Sprache. Die Frage nach Agilität erstreckt sich also auf das Erstellen der Domänenarchitektur sowie der fachlichen Modellierung und Implementierung einer Anwendung.

5.4.1. Das Agile Manifest und MDSD

Im Folgenden wollen wir sehen, wie MDSD und Agile Entwicklung zusammenpassen. Dazu zunächst das agile Manifest, wie es unter http://agilemanifesto.org zu finden ist.

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> *- Individuals and interactions over processes and tools*
>
> *- Working software over comprehensive documentation*
>
> *- Customer collaboration over contract negotiation*
>
> *- Responding to change over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more.*

Analysieren wir nun die einzelnen Aussagen im Kontext von MDSD.

Individuals and Interactions vs. Processes and Tools

Dieses Statement ist in erster Linie Ausdruck einer Wertschätzung des Menschen. Es sagt letztendlich, dass keine allzu formalen Prozesse etabliert werden sollen, die die Menschen nicht mitnehmen. Ein Team soll seinen eigenen, für sich passenden Entwicklungsprozess definieren und diesen auch im Laufe der Zeit weiterentwickeln. Die Interaktion zwischen den Teammitgliedern hat Vorrang vor formalen (dokumentzentrierten) Prozessen.

Der Einsatz von Tools wie Versionierungssysteme oder Compiler wird selbstredend nicht kritisiert. Unter der Prämisse, dass bei MDSD ein Teil der Programmierung mittels der DSL geschieht, ist der Generator der Ersatz für den Compiler und es ergibt sich kein Widerspruch zu agiler Entwicklung.

Working Software vs. Comprehensive Documentation

In der Projektpraxis ist es wichtiger, lauffähige Software abzuliefern anstatt gut aussehender Dokumente (Anforderungen, Konzepte, Architektur, Design). Bei MDSD ist das Modell der Quellcode. Diagramme sind nicht schmuckes Beiwerk, sondern zentrales Artefakt. Diagramme und „die Software“ werden auch nicht auseinander laufen, sondern sind schon deshalb immer aktuell, weil die Anwendung direkt aus dem Modell generiert wird.

Das Erstellen lauffähiger Software wird durch MDSD deutlich beschleunigt, weil lästige, wiederkehrende Implementierungsaufgaben automatisiert sind. Wie oben erläutert, propagieren wir keinesfalls ein wasserfallartiges Vorgehen, wo zunächst die Domänenarchitektur implementiert wird und erst in einer zweiten Projektphase die darauf basierende Anwendung entwickelt wird. Eine enge Verzahnung der beiden Aspekte erlaubt die frühe Erstellung lauffähiger, wenn auch nicht vollständiger Anwendungen.

Customer Collaboration vs. Contract Negotiation

Dieser Aspekt besagt, dass wir den Kunden möglichst gut in die Anwendungsentwicklung einbinden sollen; insbesondere die Reaktion auf die sich ändernden Anforderungen des Kunden soll im Projekt gut möglich sein, statt am Anfang einen Vertrag aufzusetzen, wo alles festgezurrt ist (siehe auch nächster Punkt).

MDSD kann hier Vorteile gegenüber traditioneller iterativer, inkrementeller Entwicklung haben. Dies gilt insbesondere, wenn eine

fachlich motivierte DSL verwendet wird, die sich zur Kommunikation mit dem Kunden eignet und sich dadurch Feedback-Schleifen verkürzen lassen. Ansonsten ist dieser Punkt eher unabhängig von der Frage, ob modellgetrieben entwickelt wird oder nicht.

Responding to change over following a plan

Bei dieser Wertung geht es darum, die (sich ändernden) Anforderungen eines Kunden im Rahmen eines Projektes flexibel berücksichtigen zu können und sich nicht auf am Anfang formal festgelegte Anforderungen zu berufen, die für den Kunden nun möglicherweise nicht mehr relevant sind.

Modellgetriebene Softwareentwicklung erleichtert dieses Vorgehen deutlich:

- Wenn sich fachliche Anforderungen ändern, so können diese durch den generativen Ansatz schneller und konsistent umgesetzt werden als bei traditioneller Softwareentwicklung.
- Implementierungsaspekte, die durch Transformationen abgebildet sind, lassen sich an einer Stelle ändern. Die Änderung wird automatisch in die gesamte Anwendung propagiert.

5.4.2. Agile Techniken

In diesem Abschnitt wollen wir kurz die Wechselwirkung von MDSD mit agilen Techniken ansprechen.

Eine Technik, die vor allem aus XP bekannt ist, ist *Pair Programming*. Hier sitzen zwei Entwickler vor einem Rechner, die gemeinsam die Anwendung implementieren. Vorteil: Fehler werden schneller erkannt, weil einer die Details implementiert und der andere den Überblick behält und Fehler erkennt. Dies funktioniert natürlich auch bei MDSD. Im Rahmen der Modellierung können (je nach DSL) möglicherweise sogar Entwickler und Fachexperten gemeinsam vor dem Rechner sitzen.

Eine andere, wichtige Technik ist *Test-Driven Development*, auch als *Test First* bekannt.: Hier ist die Idee, zunächst Tests zu implementieren und dann die Anwendung „dagegen“ zu entwickeln, bis alle Tests laufen. Im Kontext von MDSD ist dieses Vorgehen natürlich im Prinzip genauso möglich. Durch die zusätzliche Spezifikationsebene der Modelle eröffnen sich allerdings sogar weitere Möglichkeiten. In Kapitel <Ref Test> gehen wir näher darauf ein.

Oft wird die Tatsache, dass das Design einer Anwendung in eine standardisierte Implementierung überführt wird, als Einschränkung der Freiheit des Entwicklers gesehen und auch als Einschränkung seiner Möglichkeiten, auf die Wünsche des Kunden zu reagieren. Auf der anderen Seite wird gerade in agilen Projekten ein signifikanter Anteil der Refactoring-Aufwände darauf verwendet, die Implementierung der Soll-Architektur anzunähern. Insofern ist die „Kodierung der Architektur" in den Transformationen eigentlich nur die konsequente Fortführung des Gedankens. Refactoring als agile Technik kann prinzipiell bei Modellen, Plattformen, Transformationen und natürlich manuell implementiertem Code angewendet werden.

Wir sind nicht nur der Auffassung, dass MDSD und Agilität in keinem Widerspruch zueinander stehen, sondern dass MDSD helfen kann, durch die explizite Kodierung von Architektur- bzw. Domänen-Wissen sowie die Trennung von Domänenarchitektur- und Anwendungs-Entwicklung, agile Techniken zu skalieren.

5.5. MDSD und Domain-Specific Modeling

Unter Domain-Specific Modeling (oder DSM) versteht man primär die Idee, Modelle für eine Domäne in einer für die Domäne geeigneten DSL zu erstellen. Insofern befasst sich DSM vor allem mit dem Modellierungsanteil von MDSD. Allerdings werden auch in der DSM-Community seit einiger Zeit Generierungstechniken verwendet. Es ist festzustellen, dass die Unterschiede zu MDSD zu zerfließen beginnen.

5.6. MDSD und Domain-Driven Design

Der Begriff des Domain-Driven Design (DDD) wurde vor allem durch das gleichnamige Buch von Eric Evans ([Eva03]) bekannter. Dieser Ansatz hat nur in Bezug auf die Entwicklung einer domänenspezifischen Plattform etwas mit MDSD gemeinsam: Bei DDD werden weder DSL definiert noch Code generiert. Evans beschreibt Techniken, Patterns und Prozesselemente, die zu „guten", meist UML-basierten Modellen einer Domäne führen sollen, sowie zu Code, der das Design der Anwendung möglichst gut erhält bzw. zum Ausdruck bringt.

Obwohl es keinen starken Zusammenhang zwischen MDSD und DDD gibt, ist es nichtsdestotrotz sehr nützlich, sich damit im Sinne einer Qualitätssteigerung bei der Modellierung, sowie des generierten Code und des Programmiermodells zu beschäftigen.

6. Teil 2: Domänenarchitekturen

Im vorangegangenen Teil des Buches haben wir MDSD von der praktischen Seite kennen gelernt und anschließend im Rahmen der Begriffsbildung das für MDSD zentrale Konzept der Domänenarchitektur definiert. In Form der Fallstudie haben wir gesehen, wie eine Domänenarchitektur konkret aussehen kann.

In diesem Teil des Buches wollen wir uns der *Konstruktion* von Domänenarchitekturen widmen. Damit meinen wir zunächst weniger prozessbezogene Fragestellungen als solche bezüglich der Beschaffenheit. Erstere werden wir im dritten Teil des Buches behandeln. Dazu zählt insbesondere auch die Frage, welche Engineering-Ansätze sinnvoll zur Findung von DSLs einsetzbar sind.

Die folgenden Kapitel sollen technisches Handwerkszeug und allgemeine Best Practices vermitteln, die zum Erstellen von Domänenarchitekturen wichtig sind. Angefangen von der Metamodellierung als Schlüssel zur DSL-Definition, über die besondere Rolle der Ziel-Softwarearchitektur (d.h. die der Referenzimplementierung) im MDSD-Kontext, bis hin zur Extraktion von Generierungsvorschriften aus der Referenzimplementierung. Viele der eher technischen Fragestellungen (z.B. aus dem Bereich der Codegenerierung), die einem Domänenarchitekten begegnen, sind generisch, d.h. weit gehend unabhängig von der Domäne. Oder anders ausgedrückt: Viele Problemstellungen lassen sich durch generische Werkzeuge lösen, die in beliebigen oder zumindest verwandten Domänen wiederverwendet werden können. Diese Themen sind in den letzten beiden Kapiteln dieses Teils konzentriert. Die dort vermittelten Grundlagen und Best Practices sind sowohl zur Konstruktion als auch zur Auswahl von MDSD-Werkzeugen geeignet und daher nicht auf die Zielgruppe der Werkzeugbauer eingeschränkt.

7. Metamodellierung

Metamodellierung ist einer der wichtigsten Bestandteile modellgetriebener Softwareentwicklung. Kenntnisse über Metamodellierung werden für folgende MDSD-Problemstellungen benötigt:

- Konstruktion von domänenspezifischen Modellierungssprachen (DSL): Das Metamodell beschreibt die abstrakte Syntax dieser Sprache (vgl. <Ref Begr>).
- Modellvalidierung: Modelle werden gegen die Constraints des Metamodells validiert.
- Modell-zu-Modell-Transformationen: Solche Transformationen werden als Abbildungsvorschrift zwischen zwei Metamodellen definiert.
- Codegenerierung: Die Generierungs-Templates beziehen sich auf das Metamodell der DSL.
- Werkzeugintegration: Anhand des Metamodells lassen sich Modellierungswerkzeuge an die betreffende Domäne anpassen.

Diese Liste macht sicherlich deutlich, warum wir diesem Thema ein eigenes Kapitel widmen.

7.1. Was ist Metamodellierung?

Die griechische Vorsilbe *meta* bedeutet soviel wie *über*. Metamodelle sind Modelle, die etwas *über* Modellierung aussagen. Genauer: Ein Metamodell beschreibt die mögliche Struktur von Modellen – es definiert damit in abstrakter Weise die Konstrukte einer Modellierungssprache, ihre Beziehungen untereinander sowie Einschränkungen bzw. Modellierungsregeln, nicht jedoch die konkrete Syntax dieser Sprache. Man sagt daher auch, dass ein Metamodell die *abstrakte Syntax* und die

statische Semantik einer (Modellierungs-)Sprache definiert (vgl. <Ref Begr>). Umgekehrt besitzt jede formale Sprache (wie z.B. Java oder UML) auch ein Metamodell.

Metamodell und Modell stehen in einer Klasse-Instanz-Beziehung zueinander: Jedes Modell ist die Instanz eines Metamodelles.

Um ein Metamodell zu definieren, benötigt man konsequenterweise wiederum eine (Meta-)Modellierungssprache, die ihrerseits wieder durch ein (Meta-)Metamodell beschrieben ist. Theoretisch lässt sich diese Kaskade unendlich weit fortsetzen, praktisch geht man zweckmäßigerweise andere Wege, wie wir noch sehen werden.

Im Kontext von MDSD wird die DSL der Domäne durch ein Metamodell definiert. Die konkrete Syntax (d.h. die konkrete Ausprägung der textuellen oder grafischen Konstrukte, mit denen modelliert wird) ist dabei konzeptionell irrelevant – sie muss das Metamodell lediglich eindeutig abbilden. Diese Trennung von abstrakter und konkreter Syntax ist deshalb so wichtig, weil das Metamodell (und *nicht* die konkrete Syntax) die Basis für die automatisierte, werkzeuggestützte Verarbeitung von Modellen ist. Auf der anderen Seite bildet eine passende konkrete Syntax die Schnittstelle zum Modellierer – ohne sie könnten keine Modelle erstellt werden –, und ihre Qualität entscheidet z.B. über die Lesbarkeit von Modellen[26].

Metamodell und konkrete Syntax einer DSL können durch die Entkopplung in einer 1:n-Beziehung zueinander stehen: Es kann z.B. sowohl eine grafische als auch eine textuelle Syntax geben, die beide dasselbe Metamodell realisieren.

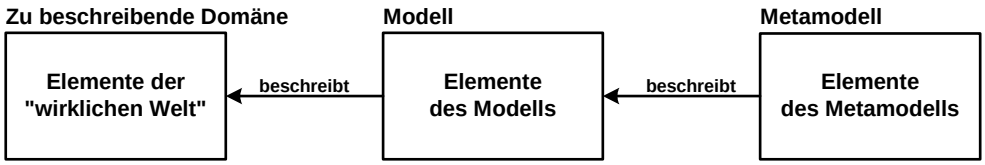


Abb. 32 *Beziehung wirkliche Welt/Modell/Metamodell*

Modelle können prinzipiell in einer beliebigen Modellierungssprache beschrieben werden. Die Auswahl der Sprache sollte sich idealerweise nach ihrer Eignung für die zu beschreibende Domäne richten. In der Praxis wird diese Wahl aber oft von der Frage bestimmt, ob für die Modellierungssprache geeignete Werkzeuge zur Verfügung stehen. Das wiederum bedeutet, dass heutzutage primär UML zur Modellierung

[26] Das bisher Gesagte gilt nicht nur für Modelle/Modellierungssprachen, sondern analog für auch Programme/Programmiersprachen.

eingesetzt wird. Insofern ist es besonders relevant, Metamodellierung im Zusammenhang mit der UML zu betrachten.

Wie bereits erwähnt, ist die „meta“-Beziehung immer relativ zu einem Modell zu sehen. Eine absolute Definition des Begriffs „Metamodell“ macht also theoretisch keinen Sinn, ist aber in der Praxis dennoch nützlich. Aus diesem Grunde definiert die OMG vier Metaschichten. Diese sind aus Abb. 33 ersichtlich und werden im Folgenden näher beschrieben.

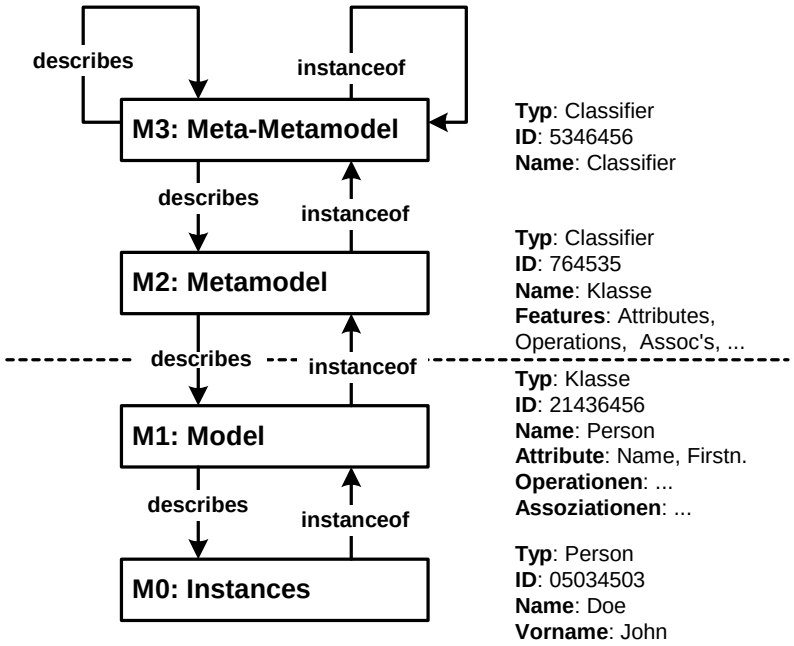


Abb. 33 *Die vier Metaschichten der OMG*

Unterhalb der gestrichelten Linie befinden wir uns in für Softwareentwickler gewohntem Fahrwasser. In M1, im Modell, wird eine Klasse definiert. Diese bekommt einen Namen, *Person*, und eine Reihe von Attributen, hier *Name* und *Vorname*. In M0 werden nun (üblicherweise zur Laufzeit des Programms) *Instanzen dieser Klasse* angelegt, hier im Beispiel die Person mit der (internen) *ID 05034503* sowie dem Namen *Doe* und dem Vornamen *John* (präziser: die Attribute *Name* und *Vorname* der Klasse Person haben bei dieser Instanz den Wert *Doe* bzw. *John*). Im Rahmen einer Instanziierung einer Klasse werden also für die Attribute der Klasse Werte festgelegt. Man beachte weiterhin, dass es von einer Klasse mehrere Instanzen geben kann. Das Modell (hier die Klasse *Person*) wird mittels einer Sprache definiert – in

unserem Fall UML –, auch wenn dies in Abb. 33 grafisch nicht dargestellt ist.

Wir bewegen uns nun eine Meta-Ebene höher. In M2, dem Metamodell, werden nun die Konstrukte definiert, die im M1-Modell Verwendung finden. Die Elemente des M1-Modells sind damit Instanzen der Elemente des M2-Metamodells. Nachdem wir nun im M1-Modell Klassen verwenden, muss das Konstrukt *Klasse* in M2 definiert werden. Dies ist im UML-Metamodell auch tatsächlich der Fall[27].

Das Konstrukt *Klasse* im UML-Metamodell ist nun eine Instanz des Metametaelements *MOF-Klasse*. MOF-Klassen sind in M3 definiert. Die *Meta Object Facility* (kurz *MOF*) ist das Metametamodell der OMG (siehe <Ref MDA>). Die MOF dient der Definition von Modellierungssprachen auf M2, wie z.B. der UML. Dahinter steht die Idee, dass UML eben nicht die einzige Modellierungssprache bleiben wird, sondern dass weitere – domänenspezifische, ggf. standardisierte – Modellierungssprachen auf Basis der MOF definiert werden. Die MOF erhebt dabei auch den Anspruch, nicht-OO-Modellierungssprachen definieren zu können. Wir werden weiter unten ein Beispiel dafür vorstellen.

Oberhalb der MOF gibt es im OMG-Modell keine weitere Meta-Ebene. Damit beschreibt sich die MOF letztendlich selbst.

Abb. 34 zeigt einen (vereinfachten und unvollständigen) Ausschnitt der MOF.

[27] Wir verwenden in M1 UML als Sprache, also muss M2 die Sprache UML definieren – als M2 kommt also hier das UML-Metamodell zum Einsatz.

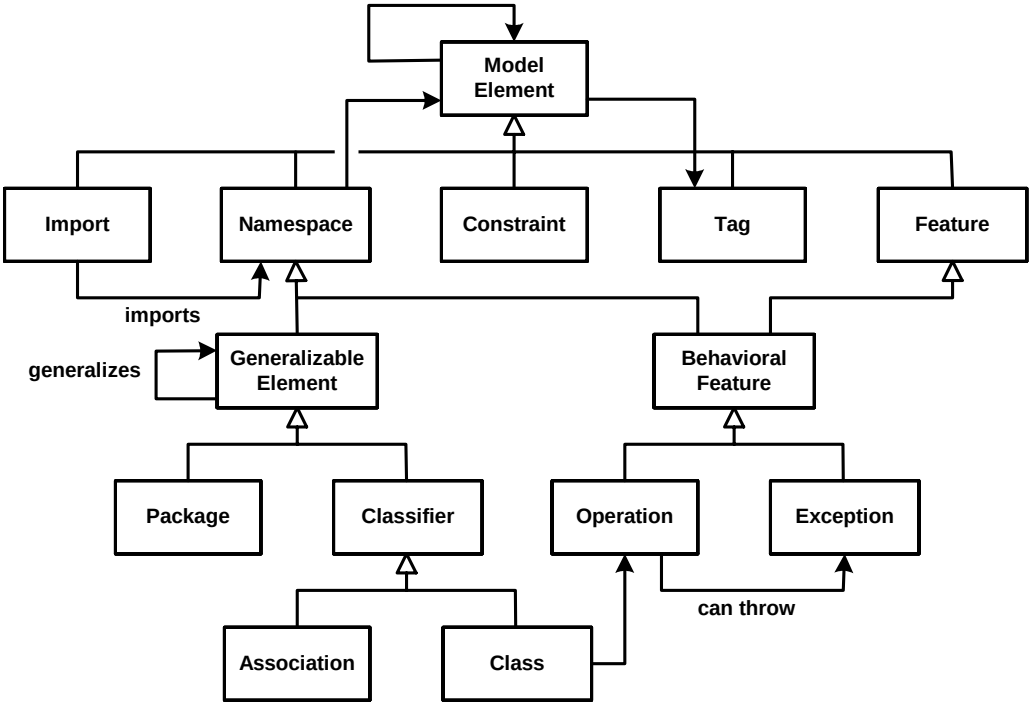


Abb. 34 *Ausschnitt der MOF*

Wie der Name MOF nahelegt, ist sie eine auf dem objektorientierten Paradigma basierende Metameta-Sprache. Zu diesem Zweck „leiht“ sich die MOF den Klassenkern der UML, verwendet also die gleichen Konzepte sowie die gleiche konkrete Syntax.

Wann immer wir beispielsweise das UML-Metamodell erweitern, beispielsweise durch Ableitung einer Metaklasse *MyMetaClass* von *UML::Class*, verwenden wir Sprachmittel der MOF. Die Vererbungsbeziehung zwischen den beiden Metaklassen ist die Vererbungsbeziehung der MOF, wie sie im *MOF::Classifier* bzw. in dessen Oberklasse *MOF::GeneralizableElement* definiert ist.

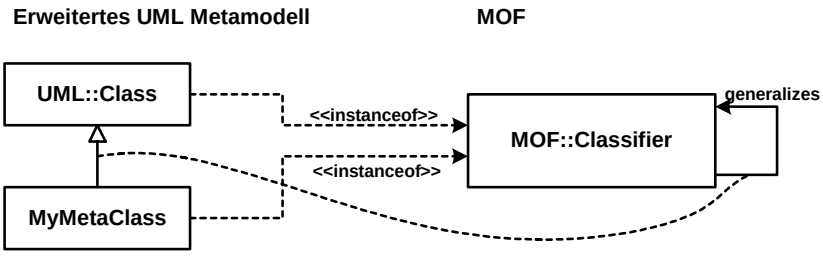


Abb. 35 *Metamodell-Erweiterung in Bezug zur MOF*

7.2. Meta-Ebenen vs. Abstraktheit

Modelle können zueinander in verschiedenen Beziehungen stehen. Dieses Kapitel beleuchtet die „meta“-Beziehung, welche aussagt, dass das Metamodell die Konzepte definiert, mit denen ein Modell erstellt werden kann.

Andererseits können sich Modelle durchaus auf verschiedenen Abstraktionsniveaus befinden, obwohl sie auf ein und derselben Meta-Ebene liegen. Typischerweise werden Transformationen verwendet, um Modelle eines höheren Abstraktionsniveaus auf ein solches mit einem niedrigeren abzubilden. Jedes der Modelle ist wie gesagt (zwangsläufig) eine Instanz eines Metamodells. Die Metamodelle der beiden Modelle sind dann zwar verschieden, trotzdem befinden sich sowohl die Modelle als auch die Metamodelle jeweils auf derselben Meta-Ebene. Abb. 36 verdeutlicht dies.

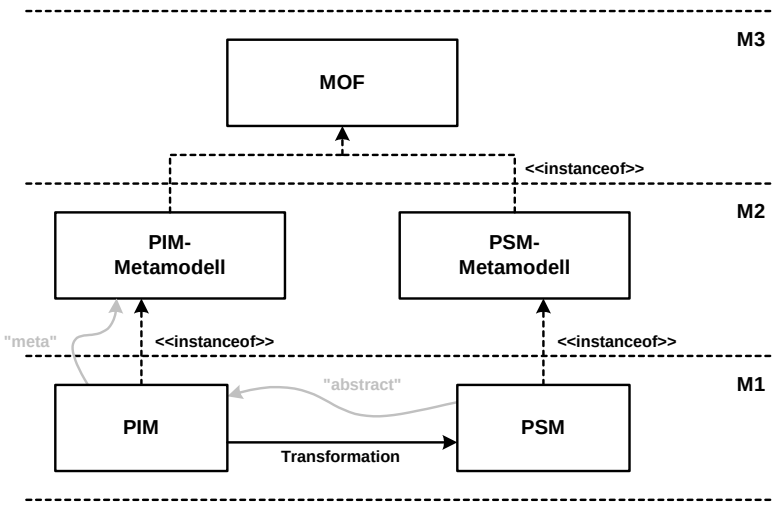


Abb. 36 *„Meta“ vs. „Abstrakt“*

7.3. MOF und UML

Die UML ist wie gesagt eine Instanz, eine Anwendung, der MOF. Dabei gibt es allerdings einige beachtenswerte Details .

Zunächst einmal gab es die UML *vor* der MOF. UML wurde zunächst informell, also rein verbal definiert, die MOF wurde im Nachhinein definiert, um die UML auf Basis der MOF formal zu spezifizieren. Die dabei entstandenen Probleme wurden in späteren

UML-Revisionen behoben, so dass man die UML nun guten Gewissens als MOF-Sprache bezeichnen kann.

Als Notation für MOF-Modelle wird die konkrete Syntax der UML benutzt. Dies kann gelegentlich zu Verwirrungen führen. Formal lässt sich dieses Problem durch die Angabe von Namespace/Packages bei Modellelementen lösen – Verwechslungspotenzial besteht trotzdem.

Des Weiteren ist zu beachten, dass es in der MOF eine ganze Reihe von Modellelementen gibt, die auch in der UML vorhanden sind. Beispielsweise gibt es in beiden Sprachen ein Element namens *Class*. Obwohl die Elemente gleich heißen und oft oberflächlich auch dasselbe bedeuten, sind sie nicht identisch – schon deshalb nicht, weil sie auf verschiedenen Meta-Ebenen angesiedelt sind.

7.4. Erweiterung der UML

Im Rahmen der Softwareentwicklung wird man in aller Regel nicht damit beginnen, eine komplett neue M2-Sprache auf Basis der MOF zu definieren. Vielmehr wird man typischerweise mit dem UML-Metamodell beginnen und dieses passend erweitern.

Um diese Erweiterung durchzuführen, gibt es drei Möglichkeiten:

- Erweiterung auf Basis des formalen Metamodells der UML
- Erweiterung mittels Stereotypen (mit Mitteln von UML 1.x)
- Erweiterung mittels Profilen (mit Mitteln von UML 2)

Wir werden jede dieser Varianten im Folgenden betrachten. In der Praxis wird man auf Grund der vorhandenen Werkzeuge zur Definition von UML-basierten Metamodellen primär den Stereotyp- bzw. Profilmechanismus verwenden.

7.4.1. Erweiterung auf Basis des Metamodells

Bei dieser Art der Erweiterung wird das Metamodell der UML erweitert. Dabei verwenden wir (wie immer bei der Modellierung) die Sprachmittel der nächsthöheren Meta-Ebene, hier also der MOF. Eine solche Erweiterung kann also innerhalb eines Werkzeuges nur erfolgen, wenn es ein explizit repräsentiertes, MOF-basiertes Metamodell besitzt und dies auch offenlegt.

Um z.B. eine „eigene Art von Klasse“ zu definieren, erstellt man eine neue M2-Klasse, die von der UML-Metaklasse *UML::Class* erbt. Abb. 37 zeigt diesen Vorgang.

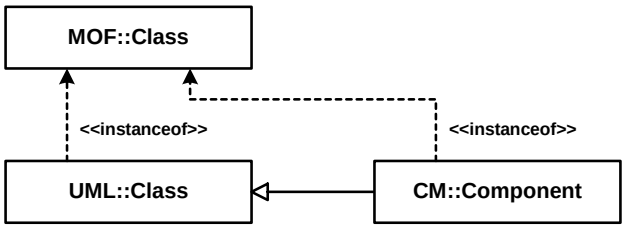


Abb. 37 *UML-Anpassung durch Erweiterung des UM- Metamodells*

Hier wird also ein neues Sprachkonstrukt definiert, die *CM::Component*. Diese ist eine Unterklasse des Elements *Class* aus der UML. Wie im vorigen Abschnitt erläutert, kommt auch hier der Vererbungsmechanismus der MOF zum Einsatz, da es sich ja hier um ein MOF-Modell (der von uns erweiterten UML) handelt.

Theoretisch kann man jedem selbstdefinierten Sprachelement (hier also *CM::Component*) eine eigene grafische Repräsentation – d.h. eine konkrete Syntax – zuweisen (siehe (e) in Abb. 38). Dies ist allerdings in der Praxis oft nicht möglich, weil das Werkzeug dies nicht unterstützt. Es bieten sich daher andere Darstellungsmöglichkeiten an, die meisten basieren auf Stereotypen (siehe Abb. 38).

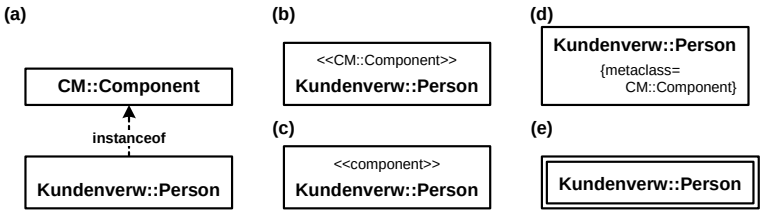


Abb. 38 *Abbildung der Metamodellerweiterung durch Stereotypen*

Abb. 38 (a) zeigt die Klasse *Kundenverwaltung::Person* direkt als Instanz der Metaklasse *CM::Component*. (b) verwendet den Namen der Metaklasse als Stereotyp; (c) eine per Konvention verabredete Kurzform, (d) einen Tagged Value, der die Metaklasse angibt und (e) zu guter Letzt eine eigene grafische Notation. Ansatz (c) hat sich in der Praxis als der praktikabelste erwiesen, (e) ist eine gute Alternative, wenn das Werkzeug diese Möglichkeit bietet.

Die *CM::Component* unterscheidet sich zunächst – abgesehen vom formalen Typ – nicht von einer UML-Klasse, da sie ja keine Attribute und Operationen hinzufügt bzw. überschreibt und auch keine Constraints definiert. Dies muss allerdings nicht so sein; man kann bei einer eigenen

Metaklasse auch neue Attribute definieren. Diese werden dann in den verwendenden Modellen üblicherweise mittels eines Tagged Value dargestellt wie in Abb. 39 gezeigt.

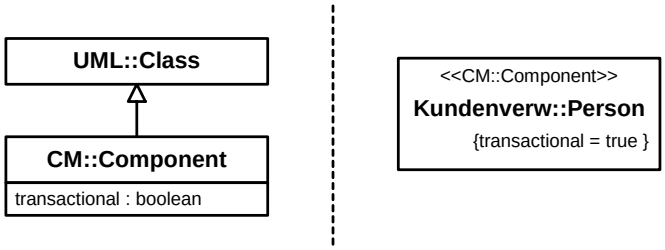


Abb. 39 *Tagged Values als konkrete Syntax für Metamodellattribute*

Die hier vorgestellte Art der Anpassung von Modellierungssprachen (also die Erweiterung des Metamodells mittels der MOF) funktioniert nicht nur im Rahmen der UML, sondern mit allen MOF-basierten Modellierungssprachen. Der weiter unten vorgestellte, auf Profilen basierende Mechanismus ist dagegen auf die UML beschränkt.

Es ist wichtig, darauf hinzuweisen, dass man durch Erweiterung des Metamodells durch Vererbung nicht die Meta-Ebene wechselt. Abb. 40 zeigt dies.

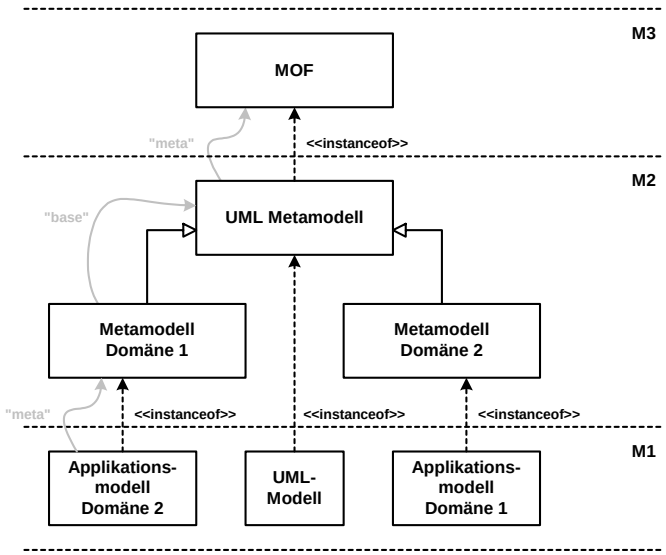


Abb. 40 *Vererbung innerhalb von M2*

Abb. 40 zeigt auch, dass die Vorsilbe„meta“ prinzipiell immer relativ zu einem Modell ist[28]. Bei Erweiterung eines Metamodells spricht man bei dem Ausgangspunkt vom *Basismetamodell*.

7.4.2. Erweiterung mittels Stereotypen in UML 1.x

Die Erweiterung mittels Stereotypen ist eine UML-spezifische Funktionalität im Rahmen des Profil-Mechanismus. Das heißt, die UML selbst stellt Sprachmittel zur Verfügung, mit denen sich das UML-Metamodell in eingeschränkter Weise erweitern, genauer gesagt spezialisieren lässt, ohne dass auf die Sprachmittel der MOF zurückgegriffen werden muss. Insofern funktioniert dieser Mechanismus der Erweiterung auch nur mit der UML, andere MOF-basierte Sprachen müssten ihre eigenen Erweiterungsmechanismen definieren. Abb. 41 zeigt die Definition des Stereotypen *CM::Component* inkl. des Tagged Value *transactional*.

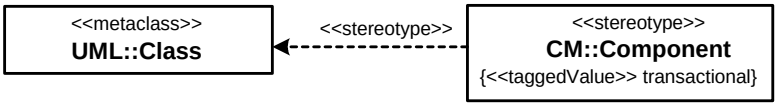


Abb. 41 *Definition eines Stereotypen in UML 1.x*

Hierbei ist zu bemerken, dass das Diagramm in Abb. 41 formal eigentlich ein M1-Modell der MOF-Hierarchie ist, denn es ist ein UML-Modell (und nicht etwa ein Ausschnitt des Metamodells der UML). Semantisch befindet es sich jedoch auf M2, denn ganz offensichtlich wird eine UML-Metaklasse (*UML::Class*) spezialisiert.

Gravierende Einschränkungen gegenüber der Erweiterung des Metamodelles via MOF besteht darin, dass Tagged Values nicht typisiert sind und keine neuen Meta-Assoziationen zwischen bestehenden Metamodellklassen oder Stereotypen definiert werden können. Der Vorteil ist wie gesagt die Nutzbarkeit im Rahmen generischer UML-Werkzeuge.

---

[28] Lediglich im Rahmen der OMG haben die Ebenen 0–3 feste Begriffe, sind also quasi absolut definiert.

7.4.3. Erweiterung mittels Profilen in UML 2.0

Mit der Definition der UML 2.0 wurde der Stereotyp-Mechanismus erweitert und in den Kontext eines umfassenderen Profil-Mechanismus gestellt (siehe auch Abschnitt 7.5 und <Ref MDA>). Zentral ist dabei das Konzept der Extensions. Eine Extension ist ein neues Symbol (und damit ein neues Sprachkonstrukt) der UML. Es wird als ausgefüllter Vererbungspfeil notiert (siehe Abb. 42).

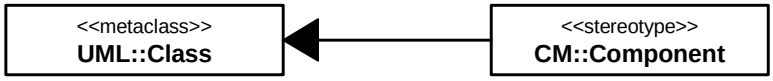


Abb. 42 *Definition eines Stereotyps in UML 2.x*

Es sei darauf hingewiesen, dass es sich hierbei nicht um Vererbung, Implementierung oder eine stereotypisierte Abhängigkeit oder Assoziation handelt, sondern wie gesagt um ein komplett neues UML-Sprachmittel. Dieses ist im UML-Metamodell auch formal definiert.

Ein Stereotyp kann Attribute haben; diese werden dann (wie schon in UML 1.x) im Modell, in dem der Stereotyp verwendet wird, als Tagged Values dargestellt (siehe Abb. 43). Ab UML 2 kann einem Tagged Value ein Typ zugeordnet werden, es sind also nicht mehr alle Tagged Values per se Strings.

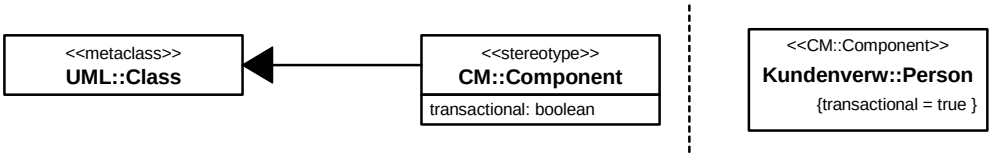


Abb. 43 *Tagged Values in UML 2.0*

Ein weiterer Unterschied zu UML 1.x besteht darin, dass ein Modellelement nun mehrere Stereotypen gleichzeitig haben kann. Es besitzt dann die Attribute aller Stereotypen als Tagged Values[29].

[29] Noch eine Formalität: Genau genommen sind die Tagged Values keine Tagged Values mehr, sondern die Darstellung der Attribute der Stereotypen. Sie werden aber – weil sie genauso aussehen – weiterhin als Tagged Values bezeichnet.

7.5. UML-Profile

Profile dienen der Anpassung bzw. Erweiterung der UML an fachliche oder technische Domänen. Man könnte auch sagen, UML ist keine Sprache, sondern eine Sprachfamilie: UML-Profile sind dann Elemente – also konkrete Sprachen – dieser Familie. Die Zielvorstellung ist die, dass UML-Werkzeuge und Generatoren Profile wie Plugins verarbeiten können, d.h. man „lädt" zunächst ein spezielles fachliches oder technisches Profil und dazugehörige Transformationen. Anschließend kann man auf Basis des Profils modellieren und dann z.B. Sourcecode generieren. Damit dies in der Praxis gut funktioniert, benötigt man offensichtlich eine klare Trennung von Modell, Profil, Transformationen und Werkzeug. Zu diesem Zweck definiert die OMG einen Profilmechanismus für die UML. Auch hierbei handelt es sich um einen UML-spezifischen Mechanismus.

UML-Profile bestehen prinzipiell aus drei Kategorien: Stereotypen, Tagged Values und Constraints[30]. Dabei können Profile die gültigen Constraints der UML nur erweitern (also weiter einschränken), nicht aber lockern. In UML 1.x war das Konstrukt des Profils im Prinzip nur verbal definiert. In UML 2.0 ist das Konzept des Profils basierend auf dem UML-Metamodell formal definiert. Dabei findet sich auch eine Definition des Extension-Konzeptes, welches im vorherigen Abschnitt erwähnt wurde.

Abb. 44 zeigt das Metamodell der Profildefinition aus der UML 2.0-Spezifikation (welches nebenbei auch als Beispiel für Metamodellierung dienen kann). Namespace-Angaben fehlen, weil alles Teil des UML-Metamodells ist[31].

[30] UML 2.0 definiert das formal etwas anders – siehe unten.

[31] Weiterhin fehlen der Einfachheit halber auch einige Constraints.

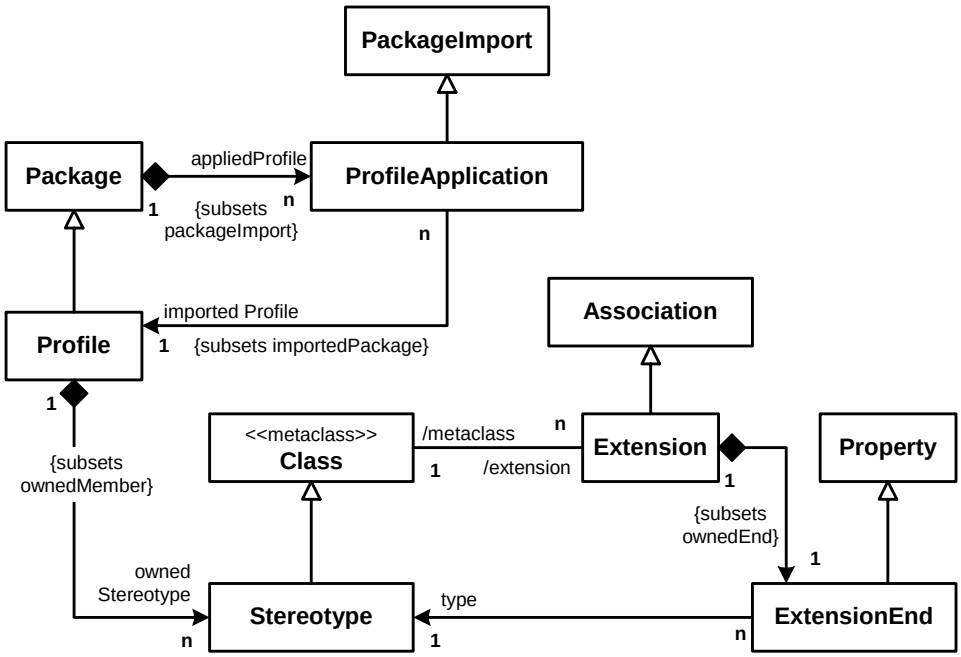


Abb. 44 *Metamodell des Profilkonzeptes*

Laut Abb. 44 wird ein *Profile* zunächst als Spezialisierung eines *UML::Package* definiert. Packages können „profiliert" werden, indem eine *ProfileApplication* angewandt wird, eine Spezialisierung von *PackageImport*. Etwas unpräzise ausgedrückt: Wenn ein Package ein Profil-Package importiert, dann bedeutet dies, dass das Profil auf das importierende Package angewandt wird. Ein Profil enthält dabei eine Menge von Stereotypen – ein Stereotyp ist dabei eine Spezialisierung von *UML::Class*. Die Extension (siehe Abschnitt zu Stereotypen) ist dabei eine Spezialisierung der *UML::Association*, deren eines Ende einen Stereotyp referenzieren muss.

UML bietet nun also die sprachlichen Möglichkeiten, Profile mittels der UML auszudrücken sowie deren Anwendung auf Anwendungsmodelle zu notieren. Abb. 45 zeigt die Definition eines (extrem vereinfachten) Profiles für EJB[32].

[32] Die OCL, die in diesem Diagramm für die Constraints verwendet wird, wird später erläutert.

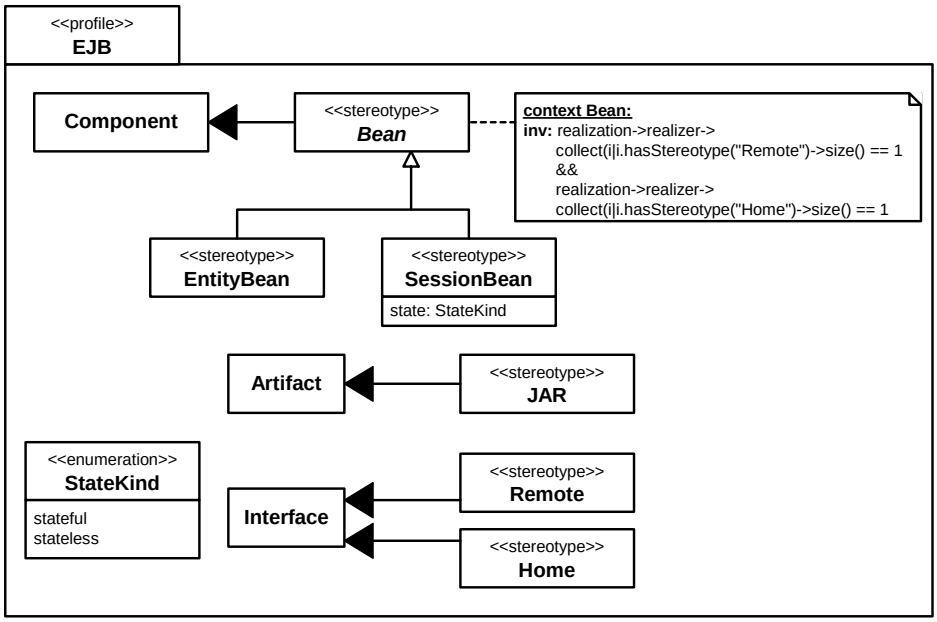


Abb. 45 *Ein einfaches EJB-Profil*

Das Diagramm sollte (nach obigen Erläuterungen) mehr oder weniger selbsterklärend sein. Einige interessante Aspekte sind dabei aber doch noch zu erwähnen: Zum einen können Stereotypen *abstrakt* sein, was konzeptionell dasselbe bedeutet wie bei abstrakten Klassen: Man kann sie nicht direkt an Modellelementen notieren, sie können lediglich als Basis(meta-)klasse für weitere Stereotypen dienen; Stereotypen können auch voneinander erben. Constraints, die für Stereotypen definiert sind, bedeuten, dass sie für die Klassen auf, die der Stereotyp angewandt wird, gelten müssen. In unserem Beispiel heißt dies, dass eine *Bean* genau ein *Remote*- und ein *HomeInterface* implementieren darf (und muss).

Auch ist an diesem Beispiel zu sehen, wie *Enumerations* (in diesem Fall zur Typisierung eines Tagged Values) modelliert werden.

Ein Profil ist nicht eigenständig, sondern es referenziert und verwendet immer ein so genanntes Referenz-Metamodell. Dies kann entweder das UML-Metamodell oder ein bestehendes Profil sein. Das Profil kann die bestehenden Definitionen im Referenz-Metamodell nicht verändern oder entfernen, aber der Profilmechanismus ist sozusagen eine wohldefinierte Hintertür, durch die man neue Konstrukte (Stereotypen) und deren Eigenschaften (Tagged Values) hinzufügen kann. Das Gleiche gilt für zusätzliche Modellierungsregeln (Constraints), die das Zusammenspiel der Konstrukte weiter einschränken und damit die Wohlgeformtheit von Modellen der spezifischen Sprache formalisieren können.

Dies kann dann auch als Basis z.B. für die Anpassung von UML-Werkzeugen dienen, so dass der Entwickler schon während der Modellierung auf profilspezifische Modellierungsfehler aufmerksam gemacht werden kann. Die derzeit gängigen UML-Werkzeuge sind allerdings in der Regel noch nicht so weit fortgeschritten. Modellierungsregeln werden – wenn überhaupt – in vielen Tools noch durch proprietäre Mechanismen unterstützt (Skripte oder Plugins). In der Praxis bieten sich (bis auf weiteres) daher folgende Möglichkeiten im Umgang mit UML-Profilen an:

- Die Constraints im Profil dienen nur zu Dokumentationszwecken, sie werden ggf. auch nur informal notiert.
- Die Formalisierung (Implementierung) der Profil-Constraints erfolgt mittels spezifischer UML-Tool-Mechanismen.
- Die Prüfung der Wohlgeformtheit wird dem MDA/MDSD-Generator überlassen, der ggf. ein profiliertes Modell unabhängig vom UML-Tool validieren kann. Die Modellierungsregeln müssen dann dementsprechend dem Generator „beigebracht“ werden. Verwendet man dazu einen OCL-Interpreter, kann sogar eine OMG-konforme, formale Profildefinition evaluiert werden.

7.6. Metamodellierung und OCL

OCL ist die Abkürzung für *Object Constraint Language.* Es handelt sich um eine seiteneffektfreie, deklarative Sprache zur Definition von Constraints (also Einschränkungen), wie z.B. Modellierungsregeln für MOF-basierte Modellierungssprachen. Constraints reichern Modelle um zusätzliche Informationen hinsichtlich der Gültigkeit von Modell-Instanzen an. Constraints lassen sich sowohl auf M1 als auch auf M2 sinnvoll einsetzen.

Angenommen, wir haben ein UML-Modell, welches eine Assoziation zwischen Personen und Autos enthält (siehe Abb. 46); die Person kann die Rolle des Fahrers sowie die eines Fahrgastes spielen. Während alle Personen Fahrgäste sein können, müssen Fahrer per Definition 18 Jahre alt sein (und einen Führerschein besitzen). Wie bringen wir dies in UML zum Ausdruck?

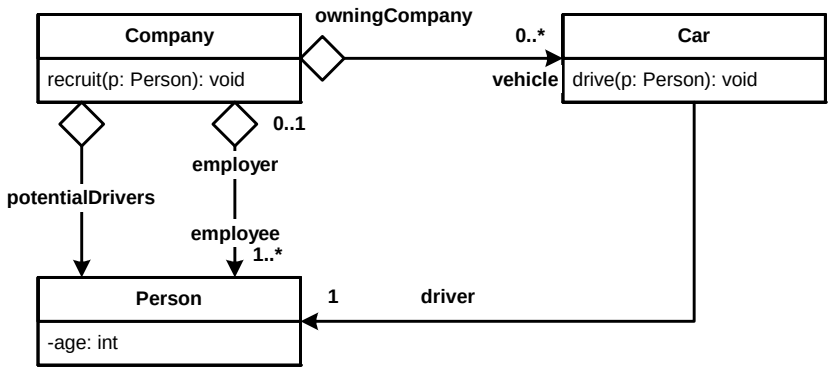


Abb. 46 *Beispielmodell für die Erläuterungen zu OCL*

Abgesehen von der suboptimalen Möglichkeit, eine Unterklasse von Person namens *VolljährigePersonMitFuehrerschein* zu definieren und die Fahrer-Assoziation zu dieser zu definieren, bleibt nur die Verwendung eines Constraints. In den folgenden Beispielen sind einige Constraints verbal und mittels OCL beschrieben.

Für alle Instanzen von *Car* gilt: Fahrer eines Autos müssen mindestens 18 Jahre alt sein (Invariante)

```
Car
driver.age >= 18
```

Für alle Instanzen von *Company* gilt: Die möglichen Fahrer der Firma sind all die Mitarbeiter, die älter als 18 Jahre sind.

```
Company
potentialDrivers = employees->select( age >= 18 )
```

Für die Operation *drive()* der Klasse *Car* gilt: Sie kann nur aufgerufen werden, wenn noch kein Fahrer im Fahrzeug sitzt und die übergebene Person älter ist als 18 (Precondition). Nach der Ausführung der Operation spielt die übergebene Person die Rolle des Fahrers (Postcondition):

```
Car::drive( p : Person )
pre : ( driver == null ) &&
      ( p.age >= 18 )
post: driver = p
```

Für die Operation *recruit()* der Klasse *Company* gilt: Nach der Ausführung der Operation ist die Liste der Mitarbeiter um eines größer als vorher, und die eingestellte Person ist Teil dieser Liste.

```
Company::recruit( p : Person )
```

```
pre : -- none
post: (employees.size =
       employees @pre.size+1 ) &&
       (employees.includes( p ) )
```

Wie anhand obiger Beispiele ersichtlich sein sollte, sind per OCL geschriebene Constraints kürzer und exakter als freier Text. Insbesondere sind sie formal an das Modell angekoppelt.

OCL hat nun im Zusammenhang mit Metamodellierung eine besondere Bedeutung, und zwar deshalb, weil Metamodelle sehr präzise und idealerweise maschinell verarbeitbar sein sollen. Ein Constraint in englischer oder deutscher Sprache kann von keinem Verifikationstool verarbeitet werden.

OCL-Constraints sind zunächst einmal modellierungssprachenunabhängig. Dies bedeutet insbesondere, dass man OCL-Constraints auf verschiedenen Meta-Ebenen anwenden kann. Das obige Beispiel verwendet OCL in einem konkreten UML-Modell, also auf M1. Dabei wirkt es auf die Instanzen der Modellelemente dieses Modells (allgemein: Ein Constraint in M_n wirkt auf M_{n-1}). Im Kontext modellgetriebener Entwicklung ist die OCL insbesondere wichtig, weil sie auch auf M2 (also beispielsweise im Rahmen einer Metamodellerweiterung) angewandt werden kann. Abb. 47 zeigt eine Erweiterung des UML-Metamodells mit einem OCL-Constraint:

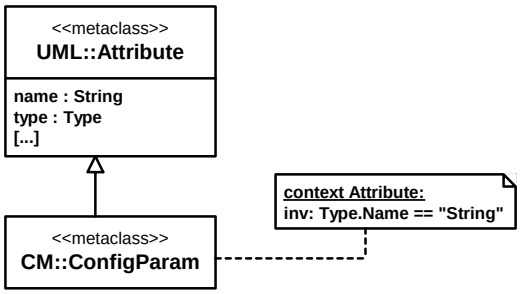


Abb. 47 *OCL-Constraints auf Ebene des Metamodells*

7.7. Metamodellierung: Beispiel 1

Im Folgenden soll zu Illustrationszwecken ein eigenes Metamodell entwickelt werden, welches nichts mit der UML zu tun hat, also das UML-Metamodell nicht erweitert. Als Beispiel sollen die aus der generativen Programmierung sowie der FODA-Methode (siehe [EC00] und [FODA]) bekannten Feature-Modelle dienen. Dieses Beispiel wird im Kapitel <Ref PLE> noch genauer eingeführt.

Zu Illustrationszwecken ist nun in Abb. 48 das Metamodell solcher Featuremodelle gezeigt. Zu beachten ist dabei, dass es sich um das Metamodell des Feature*modells* handelt. Auf die grafische Repräsentation als Diagramm wird dabei nicht eingegangen.

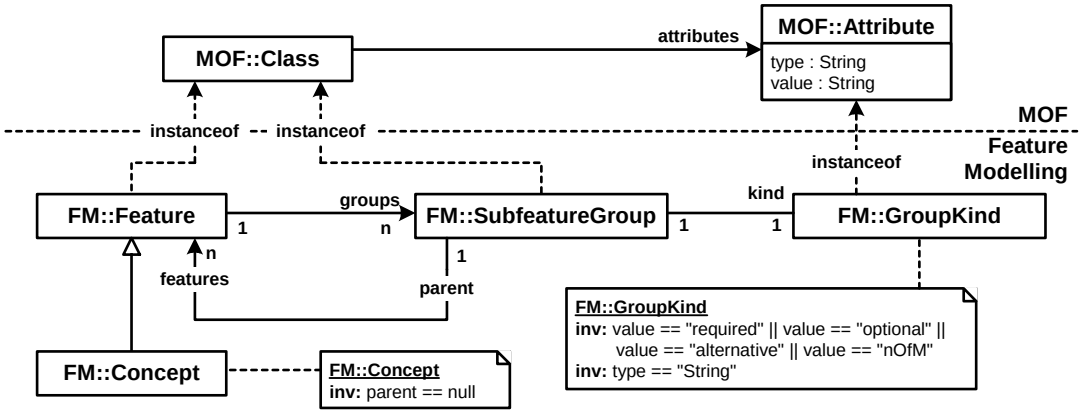


Abb. 48 *Metamodell für Featuremodelle*

Wir definieren zunächst ein Feature als Instanz von *MOF::Class*. Ein *Feature* kann dabei eine Menge von Subfeature-Gruppen haben. Eine *Subfeaturegroup* ist auch eine *MOF::Class* und fasst verschiedene *Subfeatures* zusammen. Diese Zusammenfassung hat einen Typ *verpflichtend*, *optional*, *alternativ*, *n-aus-m* in Form des Attributs *kind*. Die Attribute *type* und *value* erhält *GroupKind* dabei durch Vererbung aus der Ober-Metaklasse *MOF::Attribute*. Alternativ hätte man auch *SubfeatureGroup* als abstrakte Metaklasse definieren können und die verschiedenen Unterarten als konkrete Unterklassen.

Das Diagramm in Abb. 49 zeigt einen Ausschnitt des Beispiel-Featuremodells in <Ref PLE::PLE> als UML-Objektdiagramm, basierend auf dem eben definierten Metamodell.

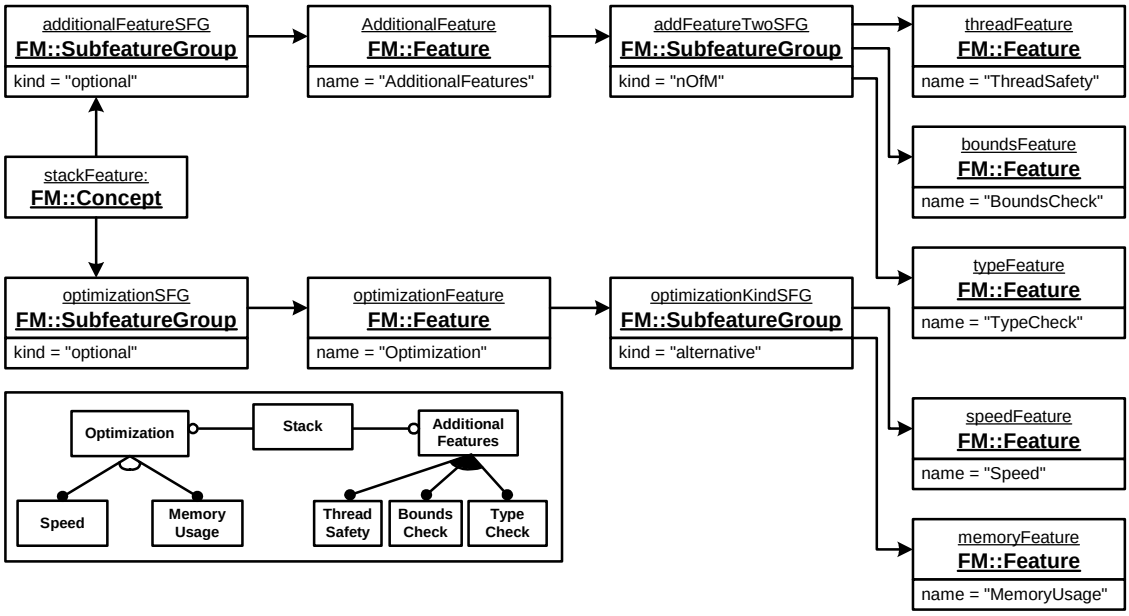


Abb. 49 *Featuremodell dargestellt mittels der konkreten Syntax von UML-Objektdiagrammen (links unten der betreffende Ausschnitt als Featurediagramm)*

Anhand dieses Diagramms ist auch sehr schön erkennbar, warum es wichtig ist, bei einer Anpassung des Metamodells eine spezifische grafische Notation (in diesem Falle eben die der Featurediagramme) zu verwenden. Diese sind erheblich leichter les- und erstellbar als ein (theoretisch ausreichendes) UML-Objektdiagramm. Die Akzeptanz domänenspezifischer Modellierung ist oft eine Frage einer passenden grafischen Notation (und der passenden Werkzeugunterstützung).

Man kann Featuremodelle noch mit weiteren Informationen anreichern. Zum Beispiel kann angegeben werden, ob ein Feature als endgültig anzusehen ist oder ob zukünftig möglicherweise zusätzliche Features (ggf. im Rahmen einer neuen *SubfeatureGroup*) dazukommen können. Im letzteren Fall nennt man ein solches Feature *offen*. Das Metamodell kann sehr einfach erweitert werden; auch die Darstellung eines Instanzdiagrammes ist naheliegend. Abb. 50 zeigt dies.

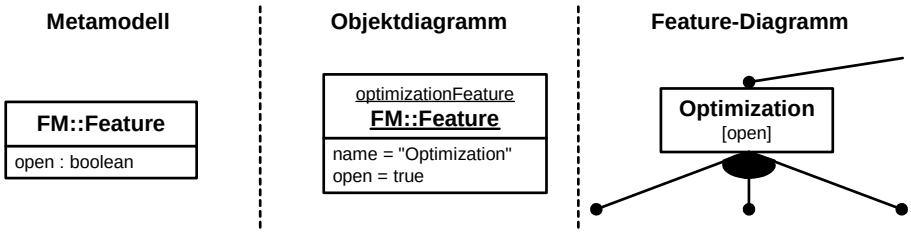


Abb. 50 *Metamodell und konkrete Syntax*

Die Verwendung von Featuremodellen wird in Kapitel <Ref PLE> noch genauer betrachtet.

7.8. Metamodellierung: Beispiel 2

Als weiteres Beispiel für Metamodellierung soll eine sehr stark vereinfachte Komponenteninfrastruktur dienen [VSW02], die für Kleingeräte und eingebettete Systeme anwendbar ist (siehe <Ref CSII> sowie [Voe02]). Zentraler Bestandteil von Anwendungen, die auf dieser Infrastruktur beruhen, sind – naheliegenderweise – Komponenten. Es macht also Sinn, im Rahmen der Architekturdefinition festzulegen, was eine Komponente ist. Wir beginnen also mit der Definition eines Metamodells für Komponenten dieser Infrastruktur[33].

Abb. 51 zeigt ein einfaches Beispiel für ein konkretes Modell, welches das Konzept der Komponenten verwendet. Zunächst wollen wir zum Ausdruck bringen, dass eine Komponente eine Reihe von Diensten anbieten kann, die im Rahmen so genannter *Provided Ports* definiert werden. Ein Provided Port ist mit einem *Interface* assoziiert, welches die angebotenen Operationen definiert. Des Weiteren soll die Komponente angeben, welche Ressourcen eine Komponente benötigt. Dies wird definiert, indem der Komponente ein so genannter *Required Port* zugeordnet wird. Auch dieser hat ein *Interface*, in diesem Falle gibt es allerdings an, welche Operationen die Komponente von anderen Komponenten *benötigt*.

Zusätzlich soll eine Komponente noch eine Reihe so genannter Konfigurationsparameter haben. Dies sind der Einfachheit halber Attribute der Komponentenklasse, die vom Typ *String* sein müssen, da

[33] Da wir im Metamodell die Konzepte der Zielarchitektur abbilden, handelt es sich hier um ein Beispiel architekturzentrierter, modellgetriebener Entwicklung.

sie beim Starten des Systems aus einer Konfigurations-Datei geladen werden sollen.

Zu guter Letzt gibt es noch einen speziellen Typ von Komponenten, die nur Dienste verwenden, aber keine anbieten: Applikationen.

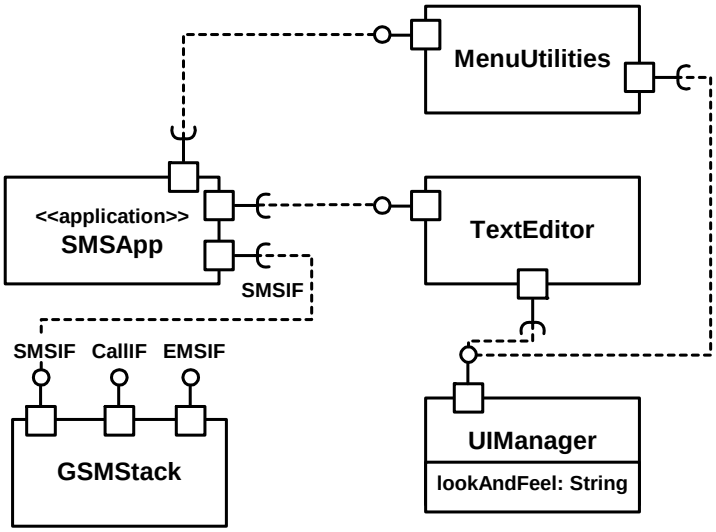


Abb. 51 *Beispiel eines einfachen, komponentenbasierten Systems*

Im Beispiel der Abb. 51 haben wir eine Applikation *SMSApp*, welche drei Required Ports definiert; diese beziehen sich auf Interfaces, welche die entsprechenden anderen Ports definieren. Beispielsweise wird das Dienst-Interface der *TextEditor*-Komponente benötigt, um eine SMS einzugeben. Sowohl der *TextEditor* als auch die *MenuUtilities* benötigen den *UIManager*, um tatsächlich auf den Bildschirm zuzugreifen.

Das Diagramm in Abb. 52 zeigt nun das Metamodell dieser Architektur.

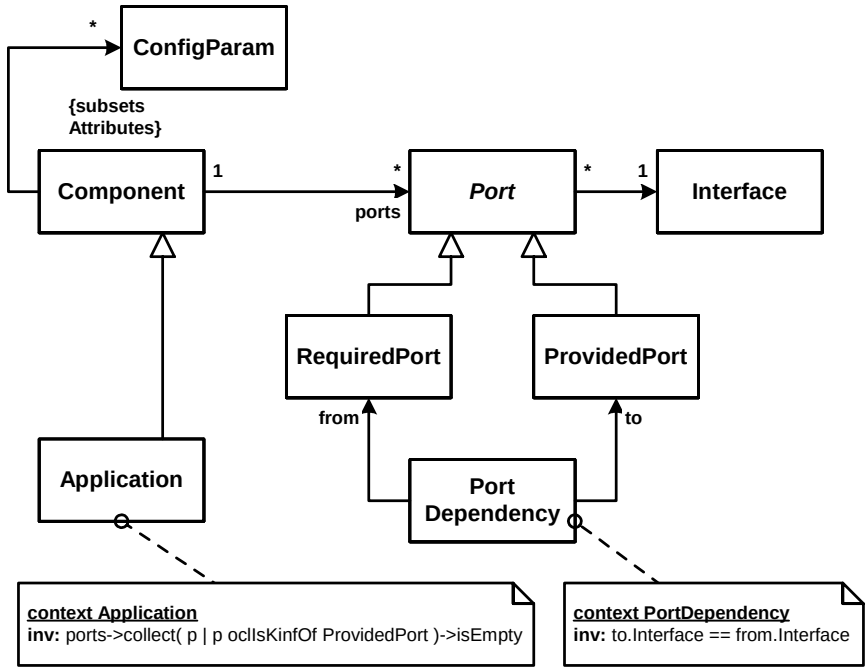


Abb. 52 *Metamodell zur Beschreibung von Komponenten*

Es bringt formal zum Ausdruck, was wir oben verbal beschrieben haben – zumindest zum Teil. Die Ankopplung eines Instanzdiagrammes (obige SMS-Anwendung) an das Metamodell passiert mittels Stereotypen und grafischen Notationen: Die Ports werden – in Anlehnung an UML 2.0 – als kleine Rechtecke auf der Kante der Komponente modelliert, Attribute solcher Komponenten sind per Definition Konfigurationsparameter. Applikationen werden als Komponente mit dem Stereotyp *Application* gekennzeichnet.

Um nicht ein komplett eigenes Metamodell entwickeln zu müssen, verwenden wir das UML-Metamodell als Basis. Wir erinnern uns an die Aussage „Ein Konfigurationsparameter ist ein Attribut vom Typ *String*“. Etwas anders ausgedrückt: Es gibt im UML-Metamodell die (Meta-)Klasse *Attribute*. Diese hat ein Attribut namens *Type*. Wir sagen hier nichts anderes, als dass die (Meta-)Klasse *ConfigParam* eine Unterklasse von *Attribute* ist, wobei gilt, dass deren Attribut *Type* den Wert *String* haben muss. Das Diagramm in Abb. 53 bringt dies zum Ausdruck. Man beachte dabei die Verwendung von OCL zur Definition des entsprechenden Constraints. Wir können nun analog bei den anderen Elementen unseres Metamodells vorgehen. Abb. 53 zeigt das Ergebnis,

wobei wir für unsere eigenen Metaklassen den Namespace bzw. das Package *CM* (für *Component Model)* einführen.

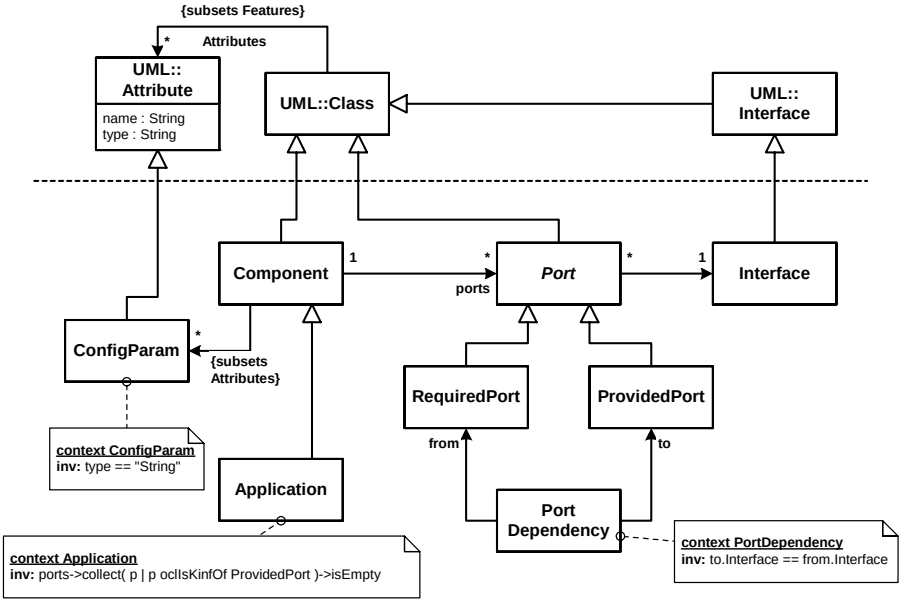


Abb. 53 *Komponentenmetamodell angebunden an das UML-Metamodell*

Worin liegt nun eigentlich der Nutzen dieser expliziten Metamodellierung? Wie immer macht Modellierung nur dann wirklich Sinn, wenn die Modelle nicht nutzlos im Schrank verstauben: Die Modelle müssen getreu dem MDSD-Prinzip im Rahmen der Softwareentwicklung sinnvoll weiterverwendbar sein. Diese Forderung gilt nun natürlich auch für Metamodelle. Diese sollen „implementierbar" sein und den weiteren Entwicklungsprozess unterstützen. Es ist also nicht essenziell, ein Metamodell als Diagramme abzuheften, sondern die Entwicklungswerkzeuge mittels des Metamodells anzupassen.

Effektive domänenspezifische Modellierung kann nur funktionieren, wenn für die zu modellierende Domäne eine passende Modellierungssprache vorliegt und diese von den verwendeten Tools „verstanden" wird.

Dabei sind die zu Beginn dieses Kapitels genannten Aspekte *Modellvalidierung, Transformation, Codegenerierung und Werkzeugadaption relevant.*

Im Folgenden sei nun der erste dieser Aspekte näher beleuchtet. Für die anderen verweisen wir insbesondere auf die Fallstudie in <Ref CSII>, die obiges Komponentenbeispiel weiterführt.

7.9. Toolgestützte Modellvalidierung

Es gibt bezüglich Metamodellierung sehr stark variierende Werkzeugunterstützung. Man kann dabei zwischen folgenden Varianten unterscheiden:

- *Keine Unterstützung:* Die meisten UML-Modellierungstools bieten kaum eine Unterstützung für Metamodelle. Dies ist nicht als negative Kritik zu sehen – sie sind schlicht nicht dafür gedacht. Sie basieren implizit auf dem UML-Metamodell, und dies ist nicht änderbar. Unbenommen bleibt natürlich die Möglichkeit, per Stereotypen ein Modell an ein Metamodell zu koppeln. Es findet jedoch keine weitere Unterstützung (bzgl. Validierung etc.) statt. In diese Kategorie fallen praktisch alle am Markt gängigen UML-Tools – wobei zu erkennen ist, dass sich dies langsam in Richtung Unterstützung für UML-Profile ändert.

- *Separate Tools:* Einige Tools kommen zum Einsatz, *nachdem* ein Modell in einem normalen UML-Werkzeug erstellt wurde. Üblicherweise wird das Modell zu diesem Zweck mittels XMI aus einem herkömmlichen UML-Werkzeug exportiert (XMI ist ein XML-Mapping für MOF, siehe <Ref MDA>) und dann auf dieser Basis weiterbearbeitet. Derartige Tools umfassen Modellvalidierer, Transformatoren und Codegeneratoren, also fast das volle Spektrum der oben erwähnten Aufgaben (und sind oft nicht auf UML/MOF beschränkt, sondern können mit beliebigen Modellierungssprachen umgehen). Ein Beispiel für diese Kategorie ist dar in <Ref CSI> verwendete Open-Source-Generator *openArchitectureWare*.

- *Integrierte (Meta-)Modellierungstools:* Im Gegensatz zu normalen UML-Modellierungswerkzeugen beruhen integrierte (Meta-)Modellierungswerkzeuge intern wirklich auf einem Metamodell. Der Anwender kann dabei mit Hilfe des Tools sowohl das Metamodell anpassen, als auch unter Verwendung dieses Metamodells Modelle erstellen; das Tool wird dann seine Oberfläche anpassen und dafür sorgen, dass nur gültige Modelle erstellt werden können. Dabei findet die Validierung meist in Echtzeit, also bereits bei der Eingabe, statt. Beispiele für solche Tools sind MetaEdit+ [MC04] oder GME [VU04].

Am verbreitetsten ist die Kombination aus UML-Werkzeug und nachgelagertem Generator-/Validierungswerkzeug. Integrierte Metamodellierungstools fristen leider immer noch ein Nischendasein

Betrachten wir nun exemplarisch die Modellvalidierung mittels *openArchitectureWare*. Das Funktionsprinzip ist in Abb. 54 dargestellt.

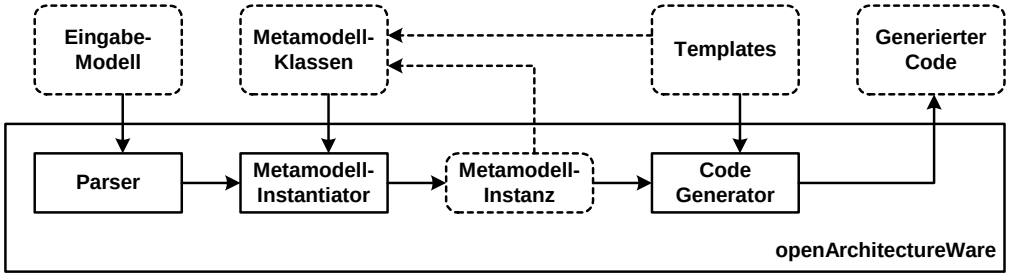


Abb. 54 *Funktionsweise des openArchitectureWare-Generator*

Als Eingabedaten verwendet der Generator ein beliebiges Modell. Dieses wird vom Parser geparst; der daraus entstehende Parse-Baum wird dann vom Metamodell-Instantiator unter Verwendung des konfigurierten Metamodells instanziiert. Das Eingabe-(Modell-)Format ist dabei austauschbar, da verschiedene Parser im Generator verwendet werden können. Im Beispiel hier kommt (wie in vielen Fällen) XMI zum Einsatz. Nach der Instanziierung des Metamodells liegt das Modell als Objektgraph (Java-Objekte) im Speicher vor. Die (Java-)Klassen der Objekte entsprechen den Metaklassen des Metamodells. Nun kann, wie bereits in der ersten Fallstudie erläutert, die eigentliche Codegenerierung mittels Templates stattfinden.

Wir haben oben bereits festgestellt, dass Metamodellierung letztlich ein Mittel ist, um die „Sprache“, die dem Modellierer zur Verfügung steht, zu definieren. Dazu gehört insbesondere die Definition von Modellierungsregeln und die diesbezügliche Validierung der konkreten Modelle.

Wenden wir uns wieder dem obigen Komponentenbeispiel zu:

Der von uns verwendete Generator besitzt ein explizites, konfigurierbares Metamodell. Dieses ist in Java implementiert. Das Prinzip haben wir bereits in <Ref CS1::Archi> ausführlich erläutert.. Damit sollte klar sein, wie eine Metamodellanpassung aussieht: Wir erstellen eine Unterklasse der entsprechenden Metaklasse und konfigurieren im Generator, dass Instanzen dieser neuen Metaklasse im Modell auf die neu implementierte Metaklasse abzubilden sind. Hier das Beispiel für den *ConfigParam*:

```
package cm;
public class ConfigParam extends Attribute {
}
```

Über eine (hier nicht gezeigte) Konfigurationsdatei sagen wir dem Generator, dass alle UML-Attribute, die den Stereotyp *ConfigParam*

haben, in Wirklichkeit Konfigurationsparameter sind, er also deshalb die Unterklasse *ConfigParam* statt *Attribute* instanziieren soll. Aus Sicht des Generators ist dies kein Problem, denn wie immer in der OO-Programmierung kann ja auch eine Instanz einer Unterklasse verwendet werden, wenn eine Variable mit der Oberklasse typisiert ist (Polymorphismus).

Nun bietet uns diese Klasse *ConfigParam* noch nicht viel Nutzen. Insbesondere haben wir noch nichts zum Thema Modellvalidierung beigetragen. Bei *ConfigParams* fehlt z.B. noch der Constraint, dass der Typ eines *ConfigParams* immer *String* sein muss. Um derartige Constraints zu überprüfen, besitzen alle Klassen des Metamodells eine Operation *CheckConstraints*, die vom Generator aufgerufen wird, wenn das *gesamte* Metamodell instanziiert ist – dies ist also die primäre Stelle, an der die Modellvalidierung stattfindet. Stellt diese Operation ein Problem fest, so wirft sie eine *DesignError-Exception*, die nachfolgend dem Entwickler gemeldet wird – das verarbeitete Modell ist nicht konform zum verwendeten Metamodell. Im Folgenden der Code für *CheckConstraints()* der Klasse *ConfigParam*[34]:

```
public String CheckConstraints()
              throws DesignError {
  if ( !Type().Name().toString().equals("String") ) {
   throw new DesignException(
             "ConfigParam Type not String" );
  }
  return super.CheckConstraints();
}
```

Um genauer zu verstehen, was hier passiert, ist ein Blick auf das vom Generator verwendete und von uns durch *ConfigParam* erweiterte UML-Metamodell hilfreich (siehe Abb. 55).

[34] *Type* und *Name* sind in diesem Beispiel Attribute der Metaklasse *Attribute*. Unglücklicherweise werden hier vom Generator Attribute mit großem Anfangsbuchstaben verwendet.

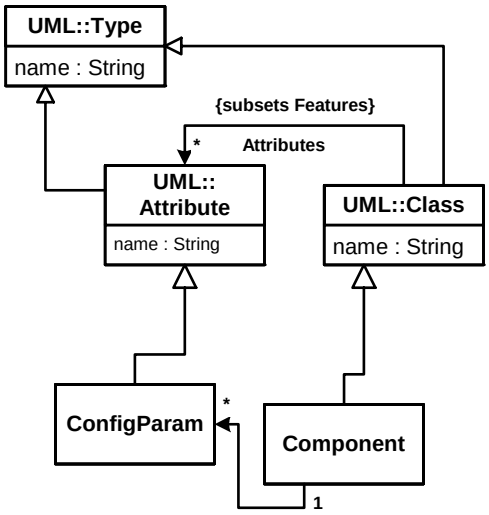


Abb. 55 *ConfigParam-Ausschnitt des Metamodells*

Da wir uns im Kontext der Klasse *ConfigParam* befinden, liefert der Ausdruck *Type()* die Instanz des *UML::Type*-Objektes durch Verfolgen der geerbten *Type*-Assoziation der Klasse *Attribute*. Dieser besitzt ein Attribut *Name* vom Typ *String*, welches den Namen des Typs enthält. Zu beachten ist hier, dass die Implementierung des Constraints nicht deklarativ per OCL, sondern operational mittels Java erfolgt. Die Integration eines OCL/Java-Übersetzers ist hier möglich und wird sicherlich im Rahmen des Open-Source-Projektes in naher Zukunft geschehen.

Auf die gleiche Art und Weise erstellen wir nun Metaklassen für *Component*, *ProvidedPort* und *RequiredPort*. Im Folgenden noch einige Beispiele. Die Metaklasse für *Component* kann zum Beispiel ungefähr folgendermaßen aussehen:

```
public class Component extends Class {

  public ElementSet Port() {
    // return all ports of the component
  }

  public ElementSet RequiredPort() {
    return Util.filter( Port, RequiredPort.class );
  }

  public ElementSet ProvidedPort() {
    return Util.filter( Port, ProvidedPort.class );
  }

  public void CheckConstraints() {
       Util.assert( Operation().size() == 0,
```

```
          "Component must not define operations by itself"
);
  }
}
```

Die Hilfsfunktion *Util.filter()* filtert dabei eine Menge von Objekten (hier: die Ports) bzgl. einer bestimmten Metaklasse. Beispielsweise liefert die Operation *ProvidedPort()* alle Ports zurück, die tatsächlich Provided Ports sind. Man beachte auch die Operation *CheckConstraints()*, die die Möglichkeit bietet, Invarianten bezüglich des Metamodells zu implementieren.

Wir können uns nun die Metaklasse *Application* ansehen. Diese ist wie oben erwähnt eine spezielle Art von Komponente, welche keine *ProvidedPorts* haben darf.

```
public class Application extends Component {
  public void CheckConstraints() {
    Util.assert( ProvidedPort().size() == 0,
      "Application must not have any provided"+
      "ports, only required ports are allowed." );
  }
}
```

Auch hier wird *CheckConstraints()* verwendet, um sicherzustellen, dass eine Applikation keine Provided Ports haben hat.

7.10. Metamodellierung und Verhalten

Verhalten ist im Zusammenhang mit Metamodellierung in zweierlei Hinsicht interessant. Zum einen kann man Verhalten in der Bedeutung des Metamodells „verstecken", zum anderen kann man mittels Metamodellierung die Verhaltensmodellierung explizit zugänglich machen (z.B. in Form von Aktivitäts- oder Zustandsdiagrammen). Den zweiten Weg verfolgen wir in <Ref GPM>. An dieser Stelle wollen wir uns auf den ersten Punkt konzentrieren.

Illustriert sei dies am wohlbekannten Komponentenbeispiel: Angenommen, wir verlangen, dass jede Komponente eine Operation *init()* haben muss. Dies lässt sich am einfachsten so realisieren, dass wir ein Interface definieren, welches diese Operation enthält, und wir weiterhin festlegen, dass alle Instanzen der Metaklasse *Component* dieses Interface implementieren müssen. Ein einfacher *Calculator* dient als Beispiel und ist in Abb. 56 dargestellt.

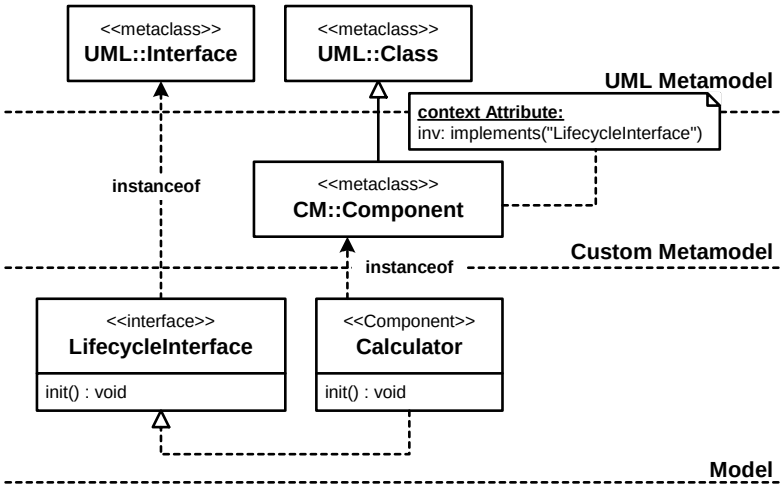


Abb. 56 *Beispiel für: „Alle Komponenten müssen ein bestimmtes Interface implementieren“*

Nun ist die Frage, was innerhalb der *init()*-Operation passiert. Es kann zum Beispiel überprüft werden, dass Links für alle *RequiredPorts* vorhanden sind. Diese Links[35] werden vom *Container* aufgebaut. Dazu bietet die Komponentenimplementierung für jeden *RequiredPort* eine entsprechende *set*-Operation, die vom Container aufgerufen wird. Die Implementierung dieser Operationen speichert die Referenz zu der Komponente, die den *ProvidedPort* für den *RequiredPort* zur Verfügung stellt, in einem Attribut. Wenn *init()* vom Container aufgerufen wird, erwartet die Komponenteninstanz, dass diese Links bereits gesetzt sind; die entsprechenden Attribute dürfen also nicht mehr *null* sein. Der Algorithmus, um dies zu überprüfen, ist dabei folgender:

```
foreach r:RequiredPort {
  if ( Attribut mit dem Namen
       des Ports  == null ) {
    FEHLER!
  }
}
```

Dies ist Verhalten, welches nicht vom Benutzer „programmiert“ wird, sondern implizit aufgrund von Richtlinien der Architektur bestimmt ist. Dies hat folgende Konsequenzen:

- Der Programmierer, der eine Anwendung erstellt, muss sich darum nicht kümmern.

[35] Referenzen sind Instanzen von Assoziationen.

- Die Modellvalidierung beschränkt sich in dem Falle darauf, dass sichergestellt werden muss, dass jede Komponente das *LifecycleInterface* implementiert. Dies geschieht wie oben erläutert. Das Verhalten innerhalb der Methode wird nicht validiert, weil der Implementierungscode (siehe unten) automatisch generiert werden kann.
- Wenn man möchte, kann man dieses Verhalten auch auf Metamodellebene spezifizieren, beispielsweise durch Sequenzdiagramme, Action Semantics (siehe <Ref MDA>) oder – in diesem Falle – auch einfach durch eine Constraint, die besagt, dass nach Ausführung der Operation alle Ressourcen-Attribute ungleich *null* sein müssen.
- Im Rahmen der Codegenerierung kann der Implementierungscode für solche Operationen direkt generiert werden. Alle Informationen, die für die Generierung vonnöten sind, sind zur Generierungszeit vorhanden. Der folgende Abschnitt zeigt ein Beispiel (die prozedurale Umsetzung des obigen Constraints):

```
«DEFINE InitOperation FOR Component»
  public void init() throws IllegalConfiguration {
  «FOREACH Operation o IN RessourceInterface»
    if ( «o.NameWithoutSet» == null ) {
      throw ( new IllegalConfiguration(
        “Ressource «o.NameWithoutSet» not set!” ) );
    }
  «ENDFOREACH »
«ENDDEFINE»
```

Es ist übrigens bemerkenswert, dass oft große Anteile des Verhaltens von Anwendungen in Wirklichkeit solches architekturdefiniertes Verhalten sind. Dazu gehören beispielsweise Persistenz, Workflow oder Remote-Proxys. Alle diese Aspekte können sehr leicht vollständig generiert werden.

7.11. Ein ausführlicheres Beispiel

Dieser Abschnitt enthält ein etwas ausführlicheres Beispiel zum Thema Metamodellierung. Dabei handelt es sich um einen Teil des ALMA-Teleskops[36]. ALMA ist ein internationales Astronomie-Projekt, welches

[36] Vielen Dank der Europäischen Südsternwarte ESO für die Erlaubnis, das Beispiel hier zu verwenden.

zum Ziel hat, ein Array von 50 Radioantennen in der Atacama-Wüste in Chile zu erbauen. Es sind verschiedene internationale Organisationen am Bau beteiligt (ESO, IRAM, MPI, NRAO). Die 50 Antennen werden rechnerisch zusammengeschaltet (Interferometrie), um bessere Auflösungen zu erhalten. Um diese zu variieren, können alle 50 Antennen per „Gabelstapler“ in ihrer Position verändert werden.

Natürlich hat ein solches Projekt eine recht umfangreiche Software-Infrastruktur. Diese besteht aus

- Echtzeitbestandteilen zur Antennensteuerung, implementiert in C++ und CORBA.
- Skripts zur Jobdefinition, implementiert in Python.
- High-Performance-Rechenmodulen zur Korrelation und Bildnachbearbeitung, implementiert in C++.
- Klassischer „IT-Infrastruktur“ (implementiert in Java) zur Definition der Projekte, der Datenverwaltung sowie dem Fernzugriff auf die Teleskopinfrastruktur (das Teleskop steht auf 5000m in Chile – die Wissenschaftler machen ihre Arbeit „von zu Hause aus“ über das Web).

Viele, wenn auch nicht alle Daten des Systems werden in mehreren dieser Subsysteme benötigt. Aufgrund der verschiedenen nicht-funktionalen Anforderungen müssen die Daten in verschiedenen Repräsentationen vorliegen: XML für Speicherung und Remote-Transport, CORBA-Strukturen im Teleskopsteuerungssystem sowie einige proprietäre, Astronomie/spezifische Formate für die effiziente Verarbeitung der Rohdaten.

Es wurde daher beschlossen, die Datenstrukturen[37] mittels UML zu definieren und verschiedene andere Artefakte daraus zu generieren:

- XML Schema
- Wrapper-Klassen für XML sowie (De-)Marshaller in verschiedenen Sprachen (C++, Java, Python)
- Konverter für die proprietären Datenformate
- HTML-Dokumentation für das Datenmodell

[37] Für Astronomen: Das Beispiel ist natürlich etwas vereinfacht...

7.11.1.Grundlegendes

Es wird zunächst unterschieden zwischen so genannten Entitäten und abhängigen Objekten. *Entities* haben eine eigene ID und können aufgrund verschiedener Merkmale gesucht werden. Eine Entity kann nun untergliedert werden. Die Teile sind *DependentObjects*. Diese haben keine eigene Identität und können nicht gesucht werden – nur die sie besitzende Entität kennt sie und kann auf sie verweisen. Teile können weitere Teile enthalten. Abb. 57 zeigt zwei Beispiele:

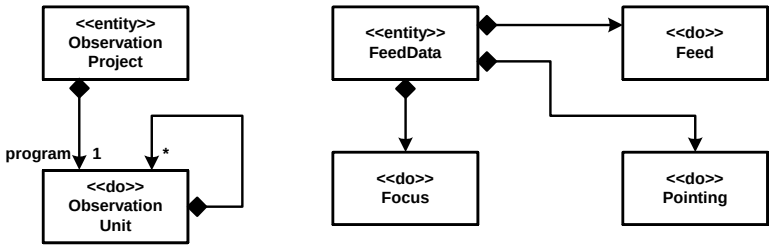


Abb. 57 *Beispiel für Entitäten und DependentObjects*

Ein *ObservationProject* enthält einzelne Abschnitte (*ObservationUnits*). Diese bilden einen Baum, der Wurzelknoten wird vom *ObservationProject* aus mit *program* referenziert. Im anderen Beispiel sind Beobachtungsdaten abgebildet. Ohne auf die Details eingehen zu wollen: Auch hier kann man erkennen, dass die *FeedData* aus verschiedenen Unterstrukturen bestehen.

Das Metamodell für solche Modelle sieht, nach Ankoppelung an die UML, wie in Abb. 58 gezeigt aus:

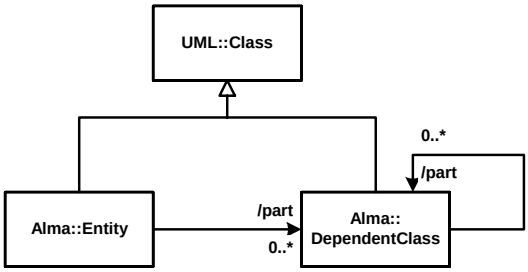


Abb. 58 *Metamodell für Entitäten und DependentClasses, angekoppelt an das UML-Metamodell*

7.11.2. Value Types

Es gibt im Datenmodell noch weitere Unterscheidungen bezüglich der Daten. Bestimmte Informationen, wie z.B. die Position eines Sternes am Himmel, sind zwar keine *DependentObjects*, aber auch keine primitiven Typen. Aus diesem Grunde haben wir *ValueTypes* eingeführt. *ValueTypes* haben auch keine Identität; sie bestehen praktisch nur aus ihrem Wert. Zwei *ValueType*-Instanzen, die denselben Wert haben, werden als identisch betrachtet. Als Konvention wird definiert, dass die Attribute von *Entities* oder *DependentObjects* nur primitive Typen oder *ValueTypes* sein dürfen. Hintergrund ist, dass diese *ValueTypes* überall im System immer wieder vorkommen. Ein Beispiel für die Verwendung von ValueTypes ist in Abb. 59 gegeben:

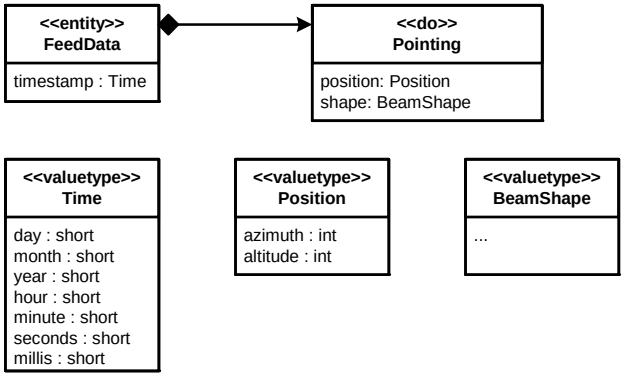


Abb. 59 *Beispiel für die Modellierung und die Verwendung von ValueTypes*

Das Metamodell wird daher entsprechend erweitert. Da sowohl *Entities* als auch *DependentObjects* und *ValueTypes* die gleiche Einschränkung bzgl. ihrer Attribute haben, wird eine entsprechende abstrakte Metaklasse *AlmaAbstractClass* eingeführt. Dies ist in der Objektorientierung üblich und wird hier nun auf der Meta-Ebene verwendet, wie Abb. 60 zeigt.

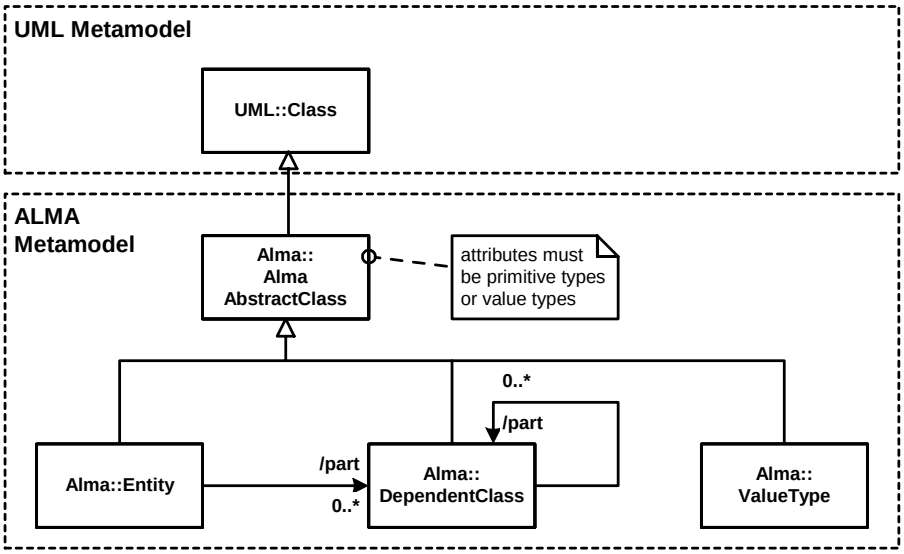


Abb. 60 *Metamodell mit herausfaktorisierter AlmaAbstractClass*

Wir haben uns hier entschieden, die Constraints als englischen Text zu beschreiben und nicht als OCL, da der Generator sowieso eine „manuelle Programmierung“ der Constraints in Java erfordert. Die Constraint bzgl. der Attributtypen ließe sich folgendermaßen als OCL-Constraint schreiben:

```
context AlmaAbstractClass
inv: attribute->forAll( a |
          (a.oclIsKindOf(ValueType) ||
           a.oclIsKindOf(primitiveType) ) )
```

7.11.3. Physikalische Größen

Nachdem es sich bei ALMA letztlich um ein physikalisches “Messinstrument” handelt, wird dort sehr viel mit physikalischen Größen gearbeitet. Es ist daher sinnvoll, physikalische Größen explizit als solche im Metamodell zur Verfügung zu stellen. Physikalische Größen haben einen Wert und eine Einheit. Verschiedene Größen haben bestimmte Einheiten und Wertebereiche. Beispiel: *Winkel* haben die Einheit *Grad* oder *arcsec* (Bogensekunden). Längen werden gemessen in *mm*, *cm*, *m*, *km*, und *pc* (Parsec). All diese Dinge sollen im Modell festgehalten werden. Es gibt nun verschiedene Möglichkeiten, diese Informationen im Modell darzustellen; wir haben uns für die in Abb. 61 gezeigte entschieden. Als Beispiel dienen wieder Winkel und Länge.

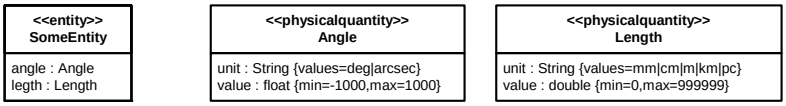


Abb. 61 *Definition und Verwendung von physikalischen Größen*

Die Einheit wird in einem Attribut *unit* gehalten, welches vom Typ *String* sein muss. Der Wert (*value*) muss vom Typ *int*, *long*, *float* oder *double* sein. Für die Einheit wird mittels eines Tagged Values die Menge von gültigen Einheiten angegeben (Liste getrennt durch |). Physikalische Größen sind verwendbar wie *ValueTypes*, müssen also auch als Attribute von *AlmaAbstractClasses* auftauchen. Im Folgenden möchten wir das entsprechende Metamodell entwickeln.

Zunächst (das dürfte nach obiger Diskussion klar sein) ist *PhysicalQuantity* eine Unterklasse von *ValueType* – physikalische Größen sind eine spezielle Art von *ValueType*. In Abb. 62 haben wir die Constraints der Einfachheit halber wieder in Englisch formuliert.

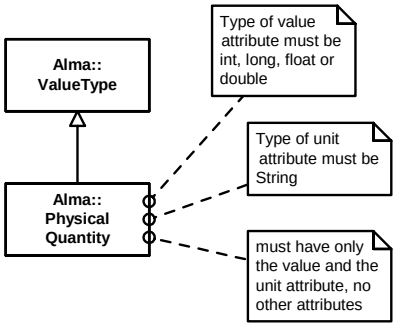


Abb. 62 *Metamodell für physikalische Größen*

Was jetzt noch fehlt, ist die Möglichkeit, Wertebereiche und die möglichen Einheiten modellieren zu können. Dazu werden neue Metaklassen eingeführt, und zwar spezielle Arten von *Attribute (siehe Abb. 63)*.

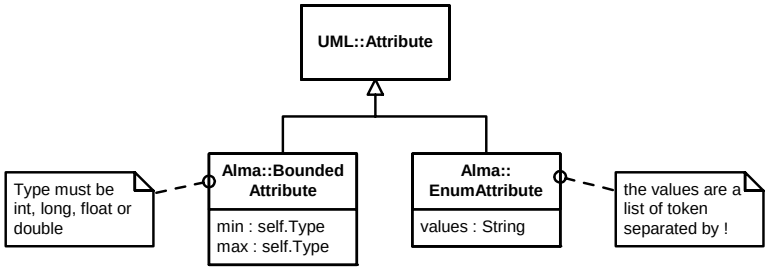


Abb. 63 *Spezielle Arten von Attributen*

BoundedAttribute muss vom Typ *int, long, float* oder *double* sein. Weiterhin werden zwei Attribute (*min* und *max*) definiert. Diese Attribute der Metaklasse tauchen nachher im Modell als Tagged Value auf, sind also letztendlich Eigenschaften der im Modell definierten physikalischen Größe! Der Minimalwert und der Maximalwert haben denselben Typ wie das Attribut selbst.

EnumAttribute kann einen beliebigen Typ haben. Der Tagged Value *values* definiert die gültigen Werte des zu definierenden Typs.

Was jetzt noch fehlt, ist der Zusammenhang zwischen der physikalischen Größe und den beiden neuen Metatypen. Das folgende Diagramm in Abb. 64 zeigt diesen.

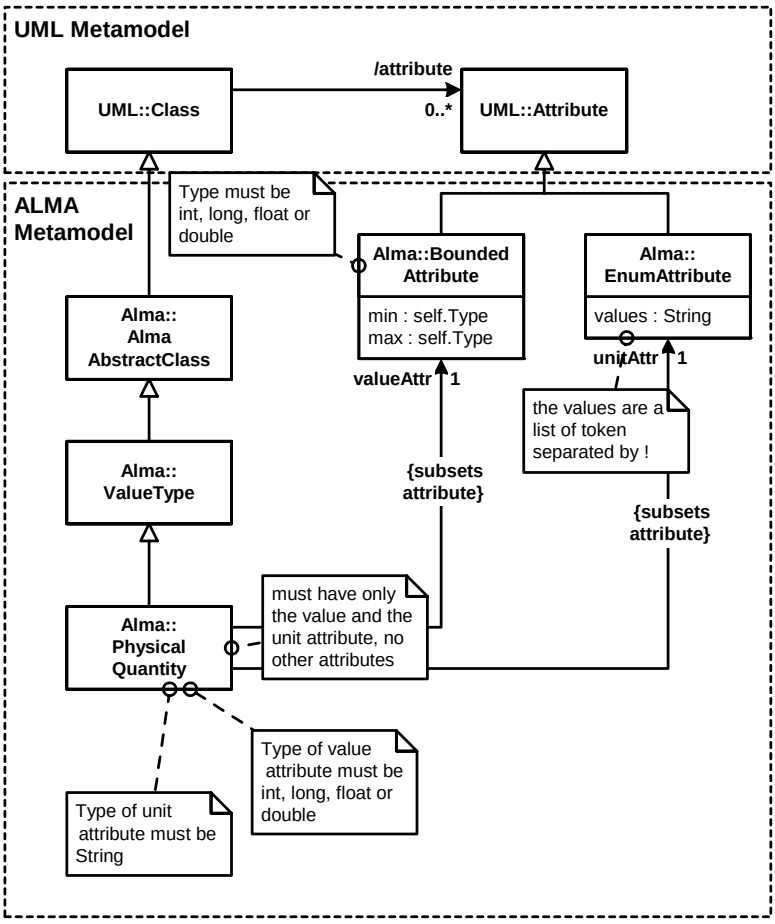


Abb. 64 *Auszug aus dem vollständigen ALMA-Metamodell*

7.12. Fallstricke bei der Metamodellierung

Dieser Abschnitt zeigt einige Tipps, Tricks und Fallstricke bezüglich Metamodellierung, insbesondere in Bezug auf die UML: Man erreicht im Rahmen der Metamodellierung zuweilen einen Punkt, an dem nicht mehr offensichtlich ist, welche Notation zu verwenden ist. Eine andere Gefahr ist die, „aus Versehen“ auf die falsche Meta-Ebene zu geraten.

Generell ist vielleicht die Leitfrage hilfreich, wie das Metamodell in einer Programmiersprache zu implementieren wäre. Diese Sichtweise kann umgekehrt einen Hinweis darauf geben, welches die richtige Notation darstellt.

7.12.1.Interfaces

Problem: Sie möchten zum Ausdruck bringen, dass Instanzen einer Metaklasse *Entity* (also alle Entities) ein bestimmtes Interface implementieren müssen.

Richtige Lösung: Die Menge der implementierten Interfaces einer Entity muss das *SomeInterface* enthalten. Dies kann entweder per OCL-Constraint ausgedrückt werden oder durch ein *Subsetting* der betreffenden Metassoziation (siehe Abb. 65).

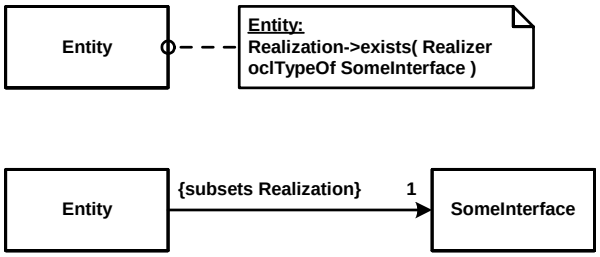


Abb. 65 *„Alle Entitäten müssen ein bestimmtes Interface implementieren.“ (Richtig)*

Falsch: Abb. 66 bringt zum Ausdruck, dass die Metaklasse *Entity* das Interface *SomeInterface* implementiert. Dies ist nicht die Aussage von oben.

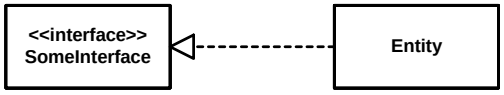


Abb. 66 *„Alle Entitäten müssen ein bestimmtes Interface implementieren.“ (Falsch)*

Allerdings kann auch dies manchmal gewollt sein. Angenommen, es gibt eine Reihe von Metamodellelementen, deren Instanzen alle einen Namen haben sollen. Es macht dann möglicherweise Sinn, auf der Meta-Ebene ein Interface zu definieren, welches die Operation *Name()* enthält. Abb. 67 zeigt dies.

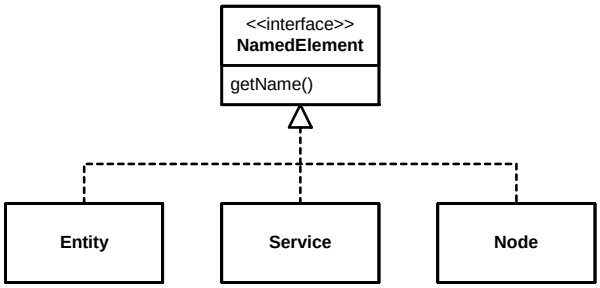


Abb. 67 *Nutzung von Interfaces und der „Implements"-Beziehung auf Ebene des Metamodells*

7.12.2. Abhängigkeiten

Problem: Sie möchten zum Ausdruck bringen, dass Komponenten von Interfaces abhängen können, weil sie deren Operationen aufrufen.

Richtige Lösung: Sie definieren eine Assoziation von Komponente zu Interface und benennen diese *uses*. Abb. 68 sagt aus, dass eine Komponente viele Interfaces verwenden kann, und dass ein Interface von verschiedenen Komponenten verwendet werden kann.

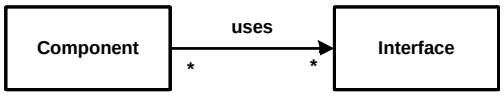


Abb. 68 *Abhängigkeiten (Richtig)*

Falsch: Das Modell in Abb. 69 hingegen sagt aus, dass die Metaklasse Component in irgendeiner Art und Weise von der Metaklasse Interface abhängt.

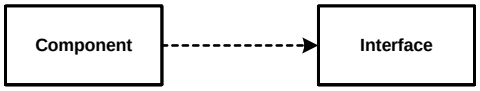


Abb. 69 *Abbildungen (Falsch)*

Man beachte, dass eine *Dependency* wie Abb. 69 nie Kardinalitäten haben kann. Man kann die Aussage „hängt von mehreren Interfaces ab“ also nicht abbilden.

7.12.3.IDs

Problem: Entities müssen genau ein Attribut mit dem Namen *ID* vom Typ *String* haben. Dieses stellt das identifizierende Attribut oder den Primärschlüssel dar. Dabei gehen wir davon aus, dass die Metaklasse *Entity* von *UML::Class* erbt und damit die ererbte Assoziation Attribute (welche die Attribute der Klasse definiert) besitzt.

Richtige Lösung: Die richtige Lösung in Abb. 70 verwendet eine OCL-Constraint, welche aussagt, dass es unter den Attributen der Entitäten genau eines geben soll, dessen Name *ID* ist und dessen Typ *String*.

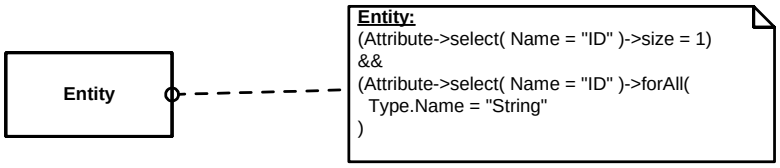


Abb. 70 *„Entities müssen genau ein Attribut mit dem Namen ID vom Typ String haben.“ (Richtig)*

Falsch: Die Definition eines Entity-Attributes mit dem Namen ID, wie in Abb. 71, führt nicht zum Ziel. Dies entspricht vielmehr der Definition eines Tagged Values für die Metaklasse Entity.

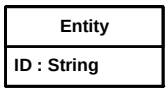


Abb. 71 *„Entities müssen genau ein Attribut mit dem Namen ID vom Typ String haben.“ (Falsch)*

Übrigens wäre auch folgende Constraint im richtigen Modell falsch:

```
context Entity inv:
  Attribute->select(
      (Name = „ID“) && (Type.Name = „String“)
    )->size = 1
```

Diese Constraint würde es erlauben, dass es mehrere Attribute mit dem Namen *ID* gibt – allerdings nur eines mit dem Typ *String*.

7.12.4.Primärschlüssel

Problem: Alle Instanzen von *Entity* müssen unter ihren Attributen genau eines haben, welches vom Typ *EntityPK* ist. Dabei ist *EntityPK* eine Spezialisierung der Metaklasse *Attribute*.

Richtige Lösung: Abb. 72 zeigt das richtige Metamodell.

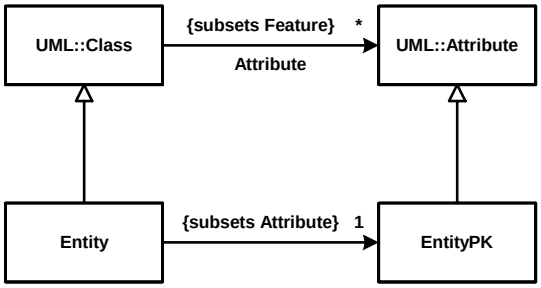


Abb. 72 *„Alle Instanzen von Entity müssen unter ihren Attributen genau eines haben, welches vom Typ EntityPK ist.“ (Richtig)*

Falsch: Abb. 73 hat das gleiche Problem wie das vorige Beispiel: Definition eines Tagged Value.

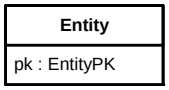


Abb. 73 *„Alle Instanzen von Entity müssen unter ihren Attributen genau eines haben, welches vom Typ EntityPK ist.“ (Falsch)*

7.12.5.Meta-Ebenen und instanceOf

Dieses Beispiel soll einen Stolperstein bei der Verwendung von komplexeren Modellierungssprachen erläutern. Als Beispiel kann hier die UML herhalten.

Abb. 74 zeigt ein UML-Klassendiagramm (M1) sowie ein UML-Objektdiagramm (M0). Objekte sind Instanzen von Klassen; diese sind im Klassendiagramm definiert. Insofern ist die Objekt-Klasse-Beziehung ganz klar eine *instanceof*-Beziehung, erkennbar auch an der Tatsache, dass es mehrere Objekte von derselben Klasse geben kann. Gleiches gilt für die Beziehung zwischen Link und Assoziation.

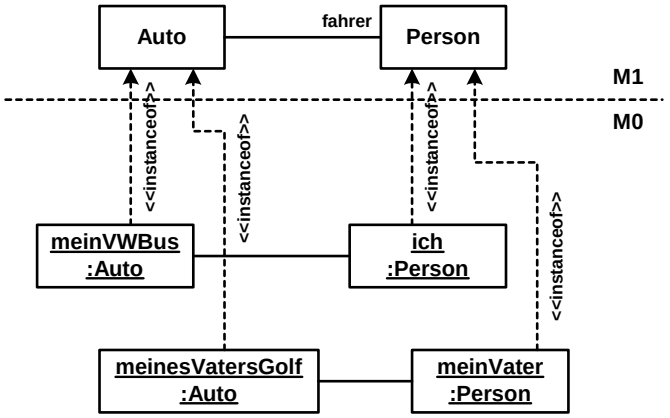


Abb. 74 *Objekte als Instanzen von Klassen*

Wie in Abb. 74 ersichtlich, sind Objekte und Klassen auf verschiedenen Meta-Ebenen angesiedelt. Auf der anderen Seite liegen sie bezüglich der UML auf *derselben* Meta-Ebene: Sowohl Klassen- als auch Objekt-Modelle sind Instanzen des UML-Metamodells. Abb. 75 zeigt dies:

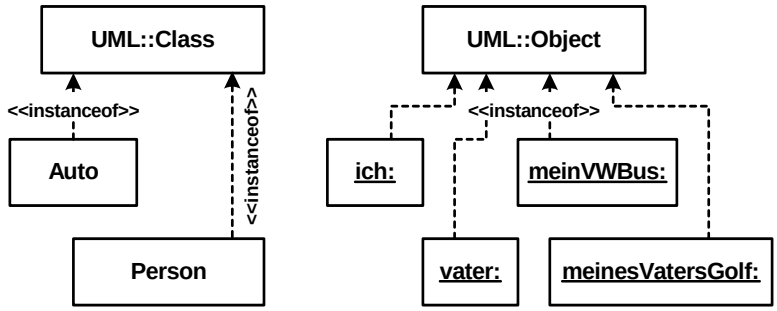


Abb. 75 *Modellelemente als Instanzen von Metamodellelementen*

Die Auflösung dieses scheinbaren Widerspruchs ist bei genauem Hinsehen aber eigentlich ganz einfach: Die beiden *instanceof* sind nicht dasselbe Sprachkonstrukt. Im ersten Beispiel ist das *instanceof* Teil der UML und im Rahmen dieser definiert (siehe Abb. 76)

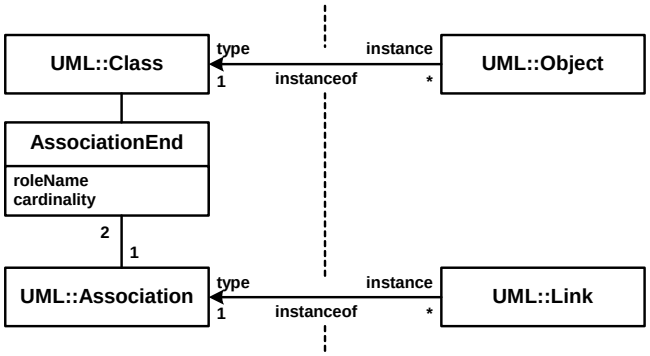


Abb. 76 *Die in UML definierte instanceof-Beziehung*

Die Beziehung zwischen *UML::Class* und *UML::Object* ist eine *MOF::Association*. Diese definiert die *instanceof*-Beziehung zwischen Instanzen von *UML::Class* und *UML::Object* in Instanzen dieses (Meta-) Modells, in UML-Modellen also. Unabhängig davon sind natürlich alle Modellelemente (*ich*, *meinVater*, *meinVSBus*, *meinesVatersGolf*) Instanzen von UML-Metaklassen (*UML::Class* bzw. *UML::Object* in diesem Fall).

8. Konstruktion MDSD-tauglicher Zielarchitekturen

8.1. Softwarearchitektur im Kontext von MDSD

So wichtig der Begriff Softwarearchitektur ist, so überbelegt und unscharf ist er leider auch. Wir wollen und müssen hier nicht den Versuch unternehmen, eine allgemein gültige und tief greifende Definition zu liefern – dazu möchten wir auf die einschlägige Literatur verweisen (z.B. [BCK98], [POSA1], [JB00], [PBG04]). Es genügt für die weiteren Betrachtungen vielmehr, die im MDSD-Kontext relevanten Sichtwinkel und Besonderheiten herauszuarbeiten. Als Basis kann uns dazu folgende, einfache „Definition" des Begriffs Softwarearchitektur genügen:

> *Die Softwarearchitektur beschreibt bis zu einem gewissen Detaillierungsgrad die Struktur (Schichtung, Modularisierung etc.) und die Systematik (Pattern, Konventionen etc.) eines Softwaresystems.*

Das Thema Softwarearchitektur spielt bei MDSD in verschiedenen Subkontexten eine Rolle:

- Zunächst einmal dient Softwarearchitektur dazu, die zu generierenden bzw. zu erstellenden Softwaresysteme insgesamt zu strukturieren. Hier spielt die Referenzimplementierung die zentrale Rolle: Die Softwarearchitektur der generativ zu erstellenden Anwendungen ist in ihr bereits vollständig zu sehen – allerdings nur exemplarisch. Aber jedes fertige Exemplar der Software-Systemfamilie besitzt genau dieselbe Softwarearchitektur. Diese Sichtweise auf das Thema ist also die klassische, allgemein übliche: Wie strukturiert man Anwendungen bzw. Softwaresysteme? Die Frage ist zunächst einmal unabhängig

davon, ob modellgetrieben entwickelt wird oder nicht, und daher kann man auch den gesamten bekannten Werkzeugkasten zur Beantwortung heranziehen. Im Kontext von MDSD gilt es darüber hinaus allerdings, zusätzliche Anforderungen zu berücksichtigen. Dieser MDSD-Sichtwinkel auf das Thema Softwarearchitektur führt uns auf die Bezeichnung *Zielarchitektur*. Die Zielarchitektur umfasst die Plattformarchitektur (s.u.).

- Auch eine MDSD-*Domänenarchitektur* (siehe <Ref Begr>) ist eine Softwarearchitektur. Sie definiert die Gesamtheit aus Metamodell (bzw. DSL), der Plattform sowie den Transformationen. Die Domänenarchitektur stellt die Basis für die Produkte einer Software-Systemfamilie dar. Hier befinden wir uns in gewisser Hinsicht also auf der Meta-Ebene, denn eine Domänenarchitektur dient der Erzeugung von Software. Die Domänenarchitektur legt weite Teile der Zielarchitektur fest.

- Des Weiteren ist Softwarearchitektur im Zusammenhang mit der MDSD-Plattform relevant. Dort beschreibt sie die wichtigsten Plattformkomponenten, ihre Interaktionen sowie deren nicht-funktionale Eigenschaften, weshalb wir sie in diesem Kontext als *Plattformarchitektur* bezeichnen. Die Plattform befindet sich sowohl auf der Meta-Ebene, denn die Plattform ist Teil der Domänenarchitektur, als auch auf der konkreten Ebene, denn die Plattform ist ebenso Teil jeder Software[38], die mit Hilfe der Domänenarchitektur erzeugt wurde – sonst wäre die Software ja nicht vollständig bzw. lauffähig.

- Softwarearchitektur spielt auch eine Rolle für die MDSD-Transformationen, denn durch sie wird ja gerade die Softwarearchitektur des Generats (Teil der Zielarchitektur – s.o.) festgelegt und ggf. konkrete Integrationspunkte für manuell zu programmierende Fachlogik definiert. Außerdem sind Transformationen selbst auch wiederum Software und sollten daher durch eine Softwarearchitektur strukturiert werden. Letzteres bezeichnen wir als *Transformationsarchitektur*.

[38] Dies ist eine *uses*-Beziehung: Die Plattform wird von den (generierten) Softwareprodukten *verwendet*, sie *gehört* ihnen aber nicht.

- Schließlich müssen auch generische MDSD-Werkzeuge gewissen architektonischen Anforderungen genügen. Ihre Softwarearchitektur bezeichnen wir als *Werkzeugarchitektur*.

Diese Kategorisierung liefert uns eine thematische Zerlegung, die wir zur Strukturierung der weiteren Kapitel dieses Teils des Buches verwenden können:

- Dieses Kapitel befasst sich mit dem Thema Zielarchitektur und deckt dabei „Plattformarchitektur“ mit ab.
- Das Kapitel <Ref Trans> behandelt „Transformationsarchitektur“ und deckt „Domänenarchitektur“ mit ab.
- Die Kapitel <Ref Code> und <Ref Tool> stellen Grundlagen und Auswahlkriterien für Werkzeugarchitekturen vor.

Im Zusammenhang mit MDSD hat die Zielarchitektur einen hohen Stellenwert; nur wenn ihre Konzepte wohldefiniert sind, kann man die Erstellung von Teilen dieser Architektur überhaupt automatisieren. Wenn sich das zu Generierende nicht systematisch beschreiben lässt, ist die Erstellung von Generierungsvorschriften (Transformationen) und damit einer Domänenarchitektur nicht möglich. Alle folgenden Hilfestellungen in diesem Kapitel sind daher insbesondere für die Erstellung von MDSD-Referenzimplementierungen relevant.

8.2. Was ist eine gute Architektur?

Wie in Abschnitt 8.1 erläutert, muss die zu erstellende Anwendung eine „gute Architektur“ aufweisen. Aus unserer Sicht zeichnet sich eine „gute Architektur“ durch die folgenden Eigenschaften aus:

- Zunächst muss die Architektur die Fachlichkeit der Anwendung, für die sie gebaut wird, gebührend unterstützen. Ohne diese Eigenschaft ist eine Architektur nutzlos.
- Des Weiteren muss sie die geforderten nicht-funktionalen Anforderungen umsetzen. Dazu gehören dynamische Aspekte wie Performance oder Skalierbarkeit, aber auch Faktoren wie Verfügbarkeit, Testbarkeit und Wartbarkeit.
- Die Architektur sollte eine möglichst kleine Menge klar definierter Konstrukte umfassen. Die Architektur wird damit einfach, leichter zu verstehen und damit praxistauglich.

- Die Architektur sollte auch gewisse Wachstums-/Entwicklungspfade für die Anwendung ermöglichen. Es geht hier nicht um eine „eierlegende Wollmilchsau“, sondern um eine in klar definierter Hinsicht erweiterbare Architektur.
- Eine gute Architektur ist auch gut dokumentiert. Dazu gehört die kurze und prägnante Dokumentation aller oben genannten Punkte, ein Programmiermodell, welches erläutert, wie man basierend auf der Architektur Anwendungen implementiert, sowie eine Rationale, die erläutert, warum die Architektur so ist wie sie ist, und warum mögliche Alternativen nicht umgesetzt wurden.

Ein Zeichen für eine gute Architektur ist, wenn sie in der Praxis des Projektgeschäftes – auch unter Zeitdruck – tatsächlich so umgesetzt wird, wie sie sich ein Architekt, das Team oder wer auch immer einmal ausgedacht hat, sie sich also im Projektalltag bewährt.

Im Zusammenhang mit MDSD gilt es nun noch zwei weitere wichtiger Punkte zu beachten:

- Zunächst muss die Architektur in der Lage sein, alle Produkte der Software-Systemfamilie zu „tragen“. Insbesondere sollte sie auch in der Lage sein, neue Produkte in der betreffenden Familie abzubilden, die bei der Definition der Domänenarchitektur noch nicht im Einzelnen bekannt waren (Details hierzu im Kapitel <Ref PLE>).
- Die Konzepte der Architektur müssen noch schärfer definiert sein, sonst lassen sie sich nicht automatisiert per Transformation aus Modellen erstellen.

Interessant ist vor allem der zweite Punkt: Die Konzepte/Konstrukte einer Architektur, die als Plattform für MDSD dienen soll, müssen sehr genau definiert sein. Nachdem wir diese Wohldefiniertheit bereits generell als Qualitätsmerkmal einer guten Architektur identifiziert haben, kann man sagen, dass die Anwendung von MDSD eine gute Softwarearchitektur fördert, ja regelrecht erzwingt.

8.3. Wie kommt man zu einer guten Architektur?

Die Frage, wie man zu einer guten Architektur kommt – generativ oder nicht –, füllt ganze Bücher und wir wollen das hier daher nicht komplett erörtern. Wir möchten jedoch trotzdem auf einige Aspekte eingehen.

8.3.1. Architekturmuster & -stile

Es gibt in der Softwaretechnologie nur eine begrenzte Menge gut funktionierender Architektur-Blaupausen. Diese wurden mit verschiedenen Mitteln beschrieben, unter anderem als Patterns [POSA1] oder als Stile ([BCK98]). Abb. 77 zeigt einige typische Architekturstile.

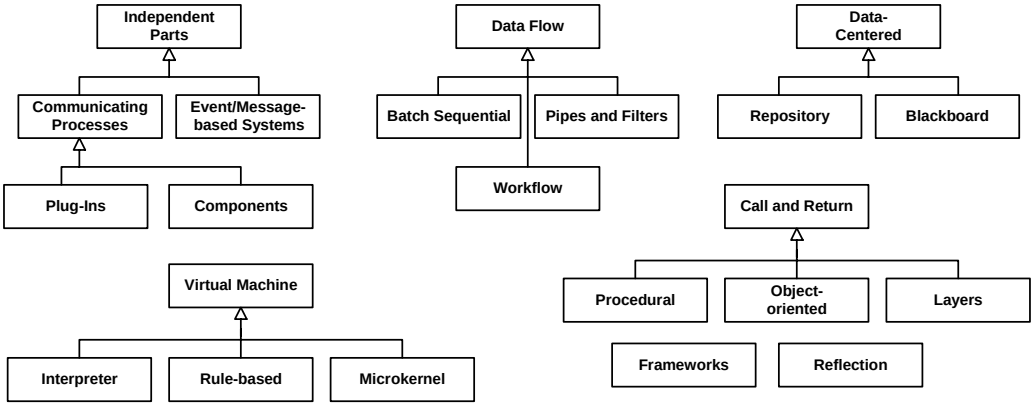


***Abb. 77** Einige bekannte Architekturstile*

Eine bewährte Möglichkeit, um zu einer guten Architektur zu gelangen, ist daher die Verwendung eines bewährten Architekturmusters/-stiles als Basis für die Architektur. [POSA1] beschreibt die grundlegenden Architektur-Patterns für Softwaresysteme recht gut und ausführlich. Zusätzlich gibt es eine ganze Reihe von Büchern, die die Architektur bestimmter Arten von Systemen beschreiben. Dazu einige Beispiele:

- POSA 2 beschreibt verteilte, multi-threading Systeme [POSA2].
- POSA 3, Resource Management Patterns kümmert sich um den architekturell extrem wichtigen Aspekt der Ressourcenverwaltung [POSA3].
- Server Component Patterns beschreibt die interne Architektur von Komponenteninfrastrukturen wie z.B. EJB, CCM oder COM+ [VSW02].
- Remoting Patterns beschreibt die interne Architektur von Remoting-Middleware wie beispielsweise CORBA, .NET Remoting oder Webservices [VKW04].
- Patterns of Enterprise Architecture beschreibt allgemein die Architektur großer Enterprise-Systeme [Fow04].
- Enterprise Integration Patterns beschreibt die Architektur von EAI-Systemen und Messaging Middleware [Hor04].

Heutzutage werden auch Referenzarchitekturen und -Plattformen wie J2EE oder .NET gerne als Basis für Architekturen verwendet. Die Verwendung einer solchen Plattform macht zwar noch keine Architektur, aber sie kann als solide Basis dienen. Man muss dabei immer noch entscheiden, welche von der Plattform angebotenen Konzepte man für seine Anwendung(en) verwenden möchte, und wie.

Eine bewährte Art und Weise, zu einer guten Architektur zu kommen, ist insbesondere, eine Architektur im Rahmen mehrerer Anwendungen (idealerweise einer Software-Systemfamilie) weiterzuentwickeln. Die Erfahrung, die man mit einer Architektur sammelt, kann in die nächsten Versionen der Architektur bzw. in die weiteren Mitglieder der Systemfamilie einfließen. Auch in dieser Hinsicht besitzt MDSD einen positiven Effekt auf die Softwarearchitektur der erstellten Anwendung.

8.4. Bausteine für Softwarearchitektur

Im Folgenden wollen wir nun auf einige technische Aspekte von Softwarearchitekturen eingehen und deren Bedeutung im Kontext modellgetriebener Entwicklung erläutern.

8.4.1. Frameworks

Als Frameworks bezeichnen wir hier alles, was mittels systematischer Erweiterung oder Konfiguration angepasst oder erweitert werden kann. Zum Beispiel müssen Entwickler, die ein Framework benutzen, bestimmte Konfigurationsparameter spezifizieren, von Basisklassen erben oder Callbacks implementieren. In aller Regel müssen mehrere solche Anpassungen vorgenommen werden, um eine bestimmte Funktionalität (ein bestimmtes Feature) zu realisieren. Dabei ist es wichtig, dass diese Änderungen zusammenpassen. Genau dieses sicherzustellen ist bei vielen Frameworks nicht immer einfach. Dies ist einer der Hauptgründe, warum Frameworks teilweise schwer zu benutzen sind und daher einen zweifelhaften Ruf genießen.

MDSD kann insofern helfen, als dass man die benötigten Features mittels einer geeigneten DSL spezifiziert und dann die verschiedenen nötigen Anpassungen des Frameworks daraus generiert. Frameworks und DSLs passen daher ideal zusammen. Anders ausgedrückt: MDSD-Plattformen lassen sich oft sehr gut mit Hilfe von Frameworks implementieren.

8.4.2. Middleware

Eine Middleware kann durchaus als ein spezielles Framework angesehen werden. Sie ist meist spezifisch für eine technische Domäne (Distributed Systems, Messaging, Transaktionen) und stellt die technische Basis für eine Zielarchitektur dar. Aufgrund ihrer Fokussierung auf technische Aspekte ist eine Middleware in vielen (fachlichen) Domänen einsetzbar und daher oft standardisiert. Wohlbekannte Beispiele sind CORBA, DCOM, MQSeries oder CICS.

8.4.3. Komponenten

Eine besonders mächtige und weit verbreitete Art von Middleware sind Komponenteninfrastrukturen. Ohne jetzt in einer Diskussion über die Definition des Begriffs „Komponente“ versinken zu wollen – eine knappe Erläuterung, was wir *hier* darunter verstehen, ist angebracht:

> *Eine Komponente ist ein in sich abgeschlossenes Stück Software mit klar definierten Interfaces und explizit deklarierten Kontextabhängigkeiten.*

Komponenten sind damit die Basis für sauber modularisierte, und „zusammenbaubare“ Systeme. Viele Domänenarchitekturen dienen dazu, Komponenten zu definieren oder vorgefertigte Komponenten zu einer Anwendung zusammenzusetzen.

Ein weiterer wichtiger Aspekt von Komponenten kann darin bestehen, dass sie der „kleinste gemeinsame Nenner“ für die Komposition von Systemen sind, die mittels verschiedener DSLs spezifiziert werden (aufgrund der verschiedenen fachlichen Subdomänen in einem System). Idealerweise sollte dieses Zusammenführen zwar auf Modellebene passieren (siehe <Ref Vers>), aber die dafür benötigten Tools (Modelltransformatoren) sind nicht immer verfügbar oder praxistauglich. Deshalb wird die Zusammenführung verschiedener Subsysteme auf Implementierungsebene erreicht. Dies wird in Abb. 78 gezeigt.

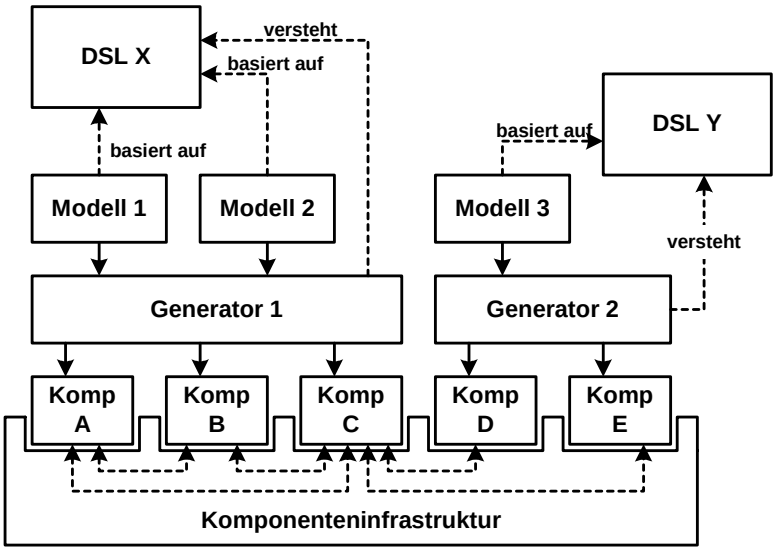


Abb. 78 *Integration auf Ebene des Generats*

Container-Infrastrukturen wie EJB oder COM+ (bzw. .NET Enterprise Services) stellen eine wichtige Grundlage für MDSD dar. Sie bieten letztendlich eine technische Plattform für Komponenten, die idealerweise nur noch Fachlichkeit enthalten. Container faktorisieren dabei die technischen Aspekte aus Komponenten heraus und stellen diese standardisiert und wiederverwendbar zur Verfügung. Derartige Container werden meist nicht generiert (eine Ausnahme stellen die Komponenteninfrastrukturen für eingebettete Systeme dar, wie sie in Kapitel <Ref CSII> gezeigt werden), sondern durch MDSD nur konfiguriert, indem beispielsweise wie in EJB Deployment-Deskriptoren aus dem Modell generiert werden.

8.5. Architektur-Referenzmodell

Für Softwarearchitekturen hat sich in der Praxis ein Schichtenmodell bewährt. Diese Struktur findet sich in mehr oder weniger ausgeprägter Form in fast allen (gut strukturierten) Softwaresystemen. Abb. 79 zeigt dieses Schichtenmodell; im Folgenden wird dies erläutert.

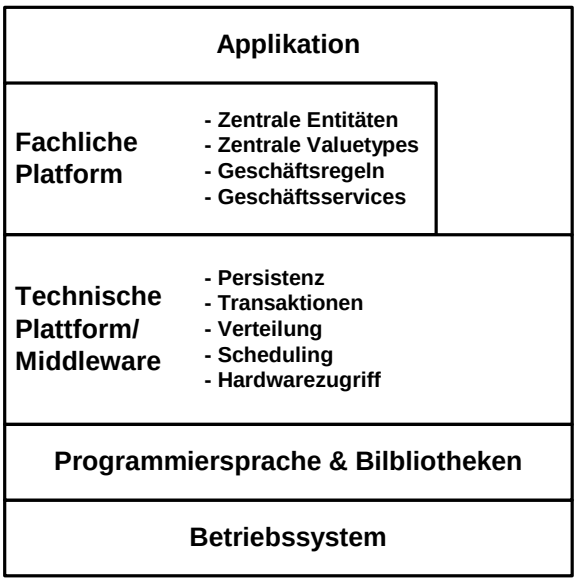


Abb. 79 *Architektur-Referenzmodell*

Die Basis jeder Architektur bieten zunächst einmal das Betriebssystem und die Programmiersprache. Darauf basierend wird üblicherweise ein technisches Basis-Framework verwendet, welches grundlegende technische Dienste anbietet (oft eine oben erwähnte Middleware). Dazu gehören zum Beispiel Persistenz, Transaktionen, Verteilung, Workflow, GUIs, Scheduling oder Hardware-Zugriff. Die Frage, welche Dienste diese Schicht tatsächlich anbietet, hängt von der technischen Domäne ab (Realtime-Embedded, Business oder Peer-to-Peer). Typische Beispiele für derartige Frameworks wären dabei J2EE und .NET (im Enterprise-Umfeld) oder im Osek + Standardcore (im Bereich eingebetteter Systeme, vor allem im Automotive-Bereich).

Basierend auf dieser Schicht findet sich üblicherweise ein fachliches Framework. Dieses enthält Konzepte aus der Anwendungsdomäne. Auch wenn sich das Folgende sehr stark nach Enterprise/Business-Systemen anhört, so gelten die folgenden Aussagen auch für technische oder eingebettete Systeme. Die Begrifflichkeiten sowie die softwaretechnischen Ausprägungen sind dort natürlich anders:

- *Entitäten* repräsentieren Konzepte, die eine Identität und einen Lebenszyklus besitzen (jeder „Kunde“ hat eine Identität, die im Rahmen seines Lebenszyklus erhalten bleiben muss).
- *Value Objects* repräsentieren Werte bzw. deren Typen. Geldbeträge (im Bankenumfeld) oder Positionen (in einer GPS-

Anwendung) sind gute Beispiele für Value Objects. Sie haben keine Identität, ihr Wert ist ausschlaggebend.

- *Business Rules* und *Constraints:* Hier werden grundlegende Regeln der Domäne festgehalten, beispielsweise die, dass Fahrer von Autos immer älter als 18 Jahre sein müssen oder dass der Wert eines Geldbetrags einer Transaktion nie negativ sein darf.
- *Services:* Hier werden grundlegende Dienste festgehalten, die nicht einer Entität zugeordnet werden können. Beispielsweise das Durchführen einer Transaktion oder das Validieren einer komplexen Dokumentenstruktur in Redaktionssystemen.

Auf dieses fachliche Framework stützt sich nun die eigentliche Anwendung, also die Implementierung der Use-Cases.

8.6. Ausbalancierung der MDSD-Plattform

Im Rahmen modellgetriebener Softwareentwicklung ist obiges Referenzmodell sehr wichtig – nicht nur zur Strukturierung der Zielarchitektur, sondern insbesondere auch zur Abgrenzung der MDSD-Plattform, die ja ein Teil der Zielarchitektur darstellt. Letztendlich müssen die Anwendungsmodelle mittels Transformationen auf die MDSD-Plattform abgebildet werden, um ausgeführt werden zu können. Je größer der Unterschied zwischen den Konzepten der MDSD-Domäne und den Konzepten der MDSD-Plattform, desto komplexer sind die notwendigen Transformationen. Dies sollte vermieden werden, zumal Komplexität auf der Meta-Ebene schwerer zu beherrschen ist als auf der konkreten Ebene der Zielarchitektur. Konkret wäre dies der Fall, wenn die MDSD-Domäne der fachlichen Anwendungsdomäne aus dem Referenzmodell entspräche und die MDSD-Plattform der technischen Plattform aus dem Referenzmodell. Um die Komplexität zu verringern, sollten MDSD-Domäne und MDSD-Plattform möglichst „nah“ beieinander liegen – genauer gesagt, die MDSD-Plattform sollte der MDSD-Domäne möglichst „entgegenkommen“. Wir nennen dies auch *reichhaltige, domänenspezifische Plattform*.

Mit Blick auf das Referenzmodell gibt es nun verschiedene grundsätzliche Möglichkeiten, den Abstand zwischen Domäne und Plattform zu verringern:

- MDSD-Domäne und -Plattform liegen auf der Ebene der technischen Plattform des Referenzmodells. Dies würde z.B. zu einem UML-Profil für J2EE als DSL und J2EE als MDSD-

Plattform führen. Eine solche Domänenarchitektur ist natürlich möglich, schöpft aber wegen des geringen Abstraktionsniveaus das Automationspotenzial typischerweise kaum aus: Der Großteil einer modellierten Anwendung muss manuell programmiert werden. Dafür ist die MDSD-Domäne recht „breit“, d.h. es kann mittels der Domänenarchitektur die Produktion fachlich sehr verschiedener Anwendungen unterstützt werden.

- MDSD-Domäne und -Plattform liegen auf der Ebene der Konzepte der Zielarchitektur (d.h. AC-MDSD, wie in der Fallstudie in <Ref CSI>). Hier werden die architektonischen Realisierungsmuster der fachlichen Plattform aus obiger Referenzarchitektur sowohl für die DSL-Definition als auch für die MDSD-Plattform aufgegriffen. In diesem Fall ist die Domäne etwas „schmaler“, aber der Abstraktionsgrad ist gestiegen und damit das Automationspotenzial.
- MDSD-Domäne und -Plattform befinden sich auf der Ebene der fachlichen Plattform der Referenzarchitektur, d.h. die fachliche Plattform der Referenzarchitektur wird zur MDSD-Plattform und die DSL ist entsprechend fachlich motiviert. Hier ist die Domäne jetzt deutlich „schmaler“ als im architekturzentrierten Fall, dafür kann die Automation ohne weiteres 100% erreichen.

Wir empfehlen im Grundsatz die letzten beiden Varianten – oder sogar eine Kaskadierung derselben, d.h. die Herstellung einer fachlichen MDSD-Plattform mittels einer untergeordneten architekturzentrierten Domänenarchitektur (siehe <Ref CSII>). Die Reichhaltigkeit der MDSD-Plattform ist jedoch in allen Fällen relevant.

8.6.1. Beispiele

Wo die Grenze der MDSD-Plattform konkret und im Einzelnen verläuft, hängt typischerweise davon ab, wie viel Flexibilität an der betreffenden Stelle vonnöten ist. Beispiele:

- Typische Bestandteile einer architekturzentrierten MDSD-Plattform für E-Business-Systeme sind: Ablaufsteuerung oder Workflow-Engine, Persistenz-Framework, Basisklassen für GUIs, Aktivitäten und Entitäten etc., technische Standard-Infrastrukturen (z.B. J2EE-Container und relationale Datenbank). All diese Artefakte sind in jedem Softwaresystem der architekturzentrierten MDSD-Domäne gleich.

- Im Rahmen einer Systemfamilie für Radioteleskope (Astronomie) sind Sterne, Galaxien oder Planeten wichtige Entitäten. Deren Eigenschaften ändern sich typischerweise nicht sehr stark. Diese Entitäten sind daher Teil einer fachlichen MDSD-Plattform. Auch viele „Business Rules“ sind fix, da sie auf physikalischen Gesetzen beruhen.
- Für Versicherungen sind Versicherungsprodukte wichtige Entitäten. Diese sind allerdings sehr unterschiedlich und sie ändern sich auch regelmäßig. Diese Entitäten werden zweckmäßigerweise über die DSL beschrieben, und deren Implementierung wird generiert. Andere zentrale Entitäten wie *Person* oder *Konto* können Teil der MDSD-Plattform sein.
- Im Rahmen einer Komponenteninfrastruktur für verteilte Embedded-Systeme wird selbst die technische Plattform, die Middleware, basierend auf vordefinierten Systemconstraints und Topologiedefinitionen generiert (siehe Kapitel <Ref CSII>) und ist damit nicht Teil der MDSD-Plattform. Selbst wenn alle Basisentitäten oder sogar die technische Infrastruktur generiert werden, gibt es üblicherweise trotzdem ein domänenspezifisches Basis-Framework als Teil der MDSD-Plattform. Im Embedded-Beispiel finden sich dort Bustreiber sowie Marshaller zur Serialisierung der Datenstrukturen für den Transport über das Netzwerk.

Vergrößern Sie inkrementell im Laufe des Projektes die Mächtigkeit der MDSD-Plattform in dem Maße, in dem Ihr Verständnis über die Domäne wächst. Dies reduziert die Größe und Komplexität des „Framework Completion Code“, welcher generiert oder teilweise sogar von Hand geschrieben werden muss.

Generell gilt: Generische und generalisierbare Code-Anteile wie z.B. Basisklassen sollten Teil der MDSD-Plattform sein. Auch bestehende Frameworks lassen sich in der Regel gut in die MDSD-Plattform integrieren.

8.6.2. Integration von Frameworks

Die Benutzung komplexer Frameworks lässt sich durch darauf abgestimmte DSLs erheblich vereinfachen und beschleunigen. Dies wird von Framework-Puristen leicht übersehen. Oft kann nur durch eine DSL und ein modellgetriebenes Verfahren gewährleistet werden, dass ein Framework auch korrekt (also „im Sinne des Erfinders“) benutzt wird.

Die heutigen Implementierungssprachen besitzen keine besonders mächtigen Mechanismen, um falsche Framework-Benutzung zu verhindern. Frameworks bedingen per Definition eine starke Verzahnung von Framework-Implementierung und Framework-Benutzung, welche dem Prinzip der Kapselung oft widerspricht. Nur mittels einer DSL kann die Domäne sauber von der benutzten Implementierungsplattform abgeschirmt werden.

Auf der anderen Seite bergen hochkonfigurierbare, generische Frameworks auch die Gefahr der Überfrachtung ihrer Implementierung. Sie sind dann nur noch schwer wartbar und schlecht zu debuggen. Dies gilt es bei der Konstruktion der MDSD-Plattform abzuwägen.

Der Schlüssel zum erfolgreichen Entwurf einer MDSD-Plattform liegt in der iterativen, inkrementellen Vorgehensweise, wie im Kapitel <Ref Proz> beschrieben. Die Entwicklung mächtiger Frameworks in groß angelegten eigenständigen Projekten ohne direkte, iterative Bindung an reale Applikationsentwicklungsprojekte führt unweigerlich zum Misserfolg. Anstelle dessen bieten kleine Frameworks in Kombination mit Codegenerierung eine solide Basis für iterative Weiterentwicklung. Wenn die Generierung (bzw. das Schreiben von Templates) zu schwierig wird, dann lohnt es sich meist, an den Frameworks zu arbeiten und mehr Features mit deren Hilfe zu implementieren. Umgekehrt gilt: Wenn die Implementierung von Framework-Funktionalität übermäßig kompliziert (oder die Performance zu schlecht) wird, können oft mittels generativer Techniken elegante Lösungen gefunden werden.

8.7. Architekturkonformität

Eine gute Zielarchitektur kann ihre Vorteile nur ausspielen, wenn sie im Projektalltag nicht ignoriert oder umgangen wird. Traditionelle Methoden wie Reviews und exzessive Dokumentation skalieren bei größeren Teams sehr schlecht. Für generierbaren Code bietet MDSD per se die Lösung, zumal die dort vorhandenen Aspekte der Zielarchitektur ja in Transformationsvorschriften gegossen sind.

Das Komponenten-Beispiel aus <Ref MM> und <Ref CS II> soll nun als Beispiel dienen, wie man mit Hilfe von MDSD die Einhaltung bestimmter Eigenschaften der Zielarchitektur auch in manuell programmierten Systemteilen erzwingen bzw. kontrollieren kann. Auch wollen wir daran zeigen, welche Auswirkungen der Einsatz dieser Möglichkeiten wiederum auf die Zielarchitektur haben kann. Abb. 80 soll das Beispiel nochmals vergegenwärtigen:

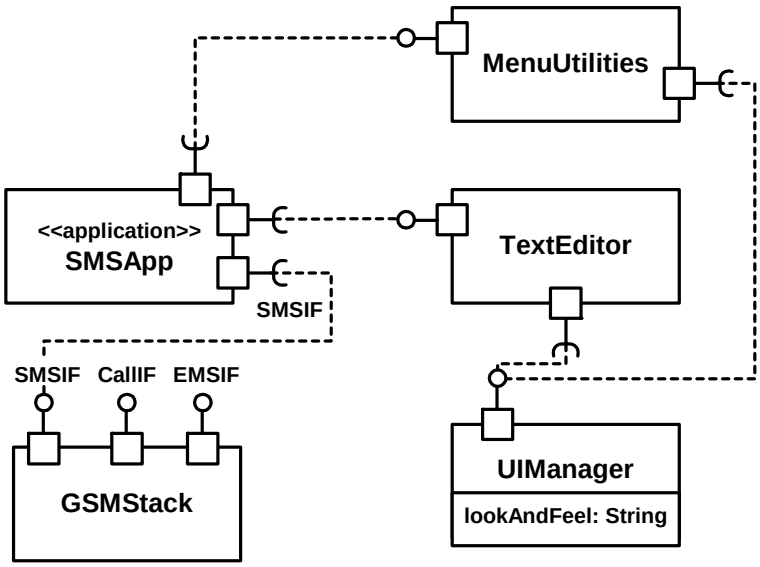


Abb. 80 *Beispiel für Abhängigkeiten zwischen Komponenten*

Abb. 80 bringt unter anderem zum Ausdruck, dass beispielsweise die Komponente *SMSApp* von den Komponenten *MenuUtilities*, *TextEditor* und *GSMStack* (bzw. deren Interfaces) abhängt. Die Offenlegung solcher Abhängigkeiten und deren Management ist in großen Projekten ein essenzieller Aspekt der Architektur. Insofern sollte es nicht passieren, dass eine Komponente (bzw. deren Implementierungscode) auf Interfaces bzw. Komponenten zugreift, für die keine Abhängigkeit im Modell definiert ist. Dies muss in großen Projekten tatsächlich sichergestellt werden, idealerweise automatisiert. Dies kann mittels MDSD, insbesondere durch den Aspekt der Codegenerierung, erreicht werden. Der folgende Codeausschnitt zeigt eine „klassische“ Implementierung einer Komponente; der Zugriff auf andere Komponenten erfolgt, indem sie sich die betreffende Referenz von einer zentralen *Factory* besorgt:

```
public class SMSAppImpl {
  public void tueWas() {
    TextEditor editor = (TextEditor)
            Factory.getComponent(“TextEditor”);
    editor.setText( someText );
    editor.show();
  }
}
```

Es ist nicht ohne weiteres möglich sicherzustellen, dass der Entwickler sich nicht „unerlaubt“ andere Referenzen von der Factory abholt. Dies

lässt sich nur durch Reviews (oder andere Tools, bspw. AspectJ) erreichen.

Wenn man dagegen modellgetrieben arbeitet, bietet sich eine andere Möglichkeit: Für jede Komponente kann ein so genannter Komponentenkontext ([VSW02]) generiert werden, welcher nur auf die Komponenten bzw. Interfaces Zugriff erlaubt, die im Modell definiert sind. Folgender Code zeigt ein Beispiel:

```
public interface SMSAppContext
         extends ComponentContext {
  public TextEditorIF getTextEditorIF();
  public SMSIF getSMSIF();
  public MenuIF getMenuIF();
}

public class SMSAppImpl implements Component {
  private SMSAppContext context = null;
  public void init( ComponentContext ctx) {
    this.context = (SMSAppContext)ctx;
  }
  public void tueWas() {
    TextEditor editor =
            context.getTextEditorIF();
    editor.setText( someText );
    editor.show();
  }
}
```

Hier kann sich der Entwickler nun nicht mehr selbstständig Referenzen organisieren – er kann nur noch auf die zugreifen, für die es im Kontext entsprechende Zugriffsoperationen gibt. Und diese werden aus dem Modell generiert. Will er auf weitere Interfaces zugreifen, so muss er dies im Modell modellieren, anders bekommt er die betreffende Zugriffsmethode nicht in seinen Code. Es ist damit sichergestellt, dass eine Komponente tatsächlich nur die Abhängigkeiten hat, die sie im Modell auch explizit angibt. Natürlich benötigt man dafür noch etwas mehr Infrastruktur, beispielsweise muss „irgendjemand“ die Operation *init()* aufrufen und das richtige Kontextobjekt zur Verfügung stellen. Dies ist die Aufgabe eines so genannten Containers, also dem Laufzeitsystem, welches sich um den Lebenszyklus der Komponente kümmert.

Übrigens hat obiges Vorgehen noch den angenehmen Nebeneffekt, dass man in modernen IDEs per Code Completion sehr schön angezeigt bekommt, welche Operationen im Kontext vorhanden sind – damit hat man die Information über die „erlaubten“ Abhängigkeiten direkt im Code –, dem Entwickler wird es also einfach gemacht, die Regeln der Architektur zu respektieren.

Obiges Beispiel zeigt auch, wie die Tatsache, dass MDSD verwendet wird, Auswirkungen auf die Architektur hat. Man würde nie auf die Idee kommen, für jede Komponente eine extra Kontextklasse zur Verfügung zu stellen, wenn man diese von Hand implementieren müsste. Die Verwendung von MDSD eröffnet also andere architektonische Alternativen. Es ist daher entscheidend, bei der Definition der Zielarchitektur diese Möglichkeiten zu kennen und auch auszunutzen.

9. Konstruktion von MDSD-Transformationen

In diesem Kapitel wollen wir uns nun damit befassen, wie zu einer gegebenen Ziel- und Plattformarchitektur (vgl. letztes Kapitel) eine passende Domänenarchitektur, insbesondere die MDSD-Transformationen, konstruiert werden können. Betrachten Sie dies aber als thematische Gliederung, nicht als prozessuale, denn die hier vorgestellten Best Practices können durchaus erwünschte Auswirkungen auf die Zielarchitektur bzw. die Referenzimplementierung haben. Fragestellungen bezüglich des Entwicklungsprozesses behandeln wir in Teil 3 des Buches.

Die Best Practices sind in zwei Gruppen untergliedert. Zunächst erläutern wir generelle konstruktive Aspekte, im zweiten Teil gehen wir dann näher auf technische Details ein.

9.1. Best Practices – Vorgehen

9.1.1. Welche Teile der Zielarchitektur sollen generiert werden?

Den Zusammenhang und die Balance zwischen MDSD-Domäne, -Plattform und -Zielarchitektur haben wir im letzten Kapitel ausführlich diskutiert. Jetzt muss noch die Trennlinie zwischen generiertem und manuell zu programmierendem Code gezogen werden.

Im Rahmen einer Domänenarchitektur wird man immer die Dinge generieren, die einerseits nicht schon durch die Plattform abgedeckt werden und sich andererseits gut und kompakt durch eine DSL beschreiben lassen.

Ein ziemlich sicheres Indiz dafür, dass man die Generierung zu weit treibt, ist in aller Regel die Einführung typischer programmiersprachlicher Konstrukte wie Schleifen o.Ä. in die DSL[39]. Diese sollte weit gehend deklarativ sein und nicht zur klassischen Programmiersprache mutieren – stattdessen sollte man dann besser eine solche verwenden und den so erstellten Code in das Generat integrieren.

9.1.2. Glaube an die Wiedergeburt

Die endgültige implementierte Anwendung sollte mit einem Build-Prozess erstellt werden, welcher die Neugenerierung von allen generierten/transformierten Artefakten umfasst. Sobald auch nur ein einziger manueller Schritt als Teil des Build-Prozesses erforderlich wird, oder auch nur eine einzige Zeile Quellcode nach der Neugenerierung manuell angepasst werden muss, wird früher oder später (früher ist die Regel!) der Generator links liegen gelassen, und die traditionelle Softwareentwicklung erlebt eine Renaissance.

Dies heißt (wie bereits anderweitig erwähnt) nicht, dass Sie 100% einer Anwendung generieren müssen/sollen. Es ist völlig in Ordnung, Teile weiterhin in einer 3GL zu implementieren, wenn die 3GL sich dafür eignet. Diese Best Practice sagt nur, dass die Teile, die generiert werden, zu 100% generiert werden, so dass man „in einem Rutsch“ das komplette System neu erstellen kann. Nachträgliche Anpassungen an generierten Artefakten sind tabu!

9.1.3. Beuten Sie das Modell aus

Die Informationen, die in einem Modell enthalten sind, sollten soweit wie möglich ausgeschlachtet werden, um Duplizierung zu vermeiden und um manuelle Arbeit zu minimieren. Das heißt, Sie werden üblicherweise weit mehr als nur Quellcode generieren: Rümpfe für Komponententests, ggf. sogar Implementierungen, Testdaten und Mock-Objekte, Datenbankerstellungs-, Migrations- und Befüllungsskripte, einfache Oberflächen zur Stamm- oder Testdatenpflege oder Teile der Dokumentation. All dies kann im Projektalltag erheblichen Ärger und Aufwand sparen. Die richtige Balance zwischen dem Aufwand zur Automatisierung und dem Aufwand für wiederholte Durchführung manueller Tätigkeiten sollten aufgrund von „sustainability“-

[39] Eine Ausnahme bildet hier die Executable UML (siehe <Ref MDA>).

Überlegungen (also über lange Zeiträume zumutbar und ökonomisch) im Sinne von Extreme Programming erfolgen. Als Faustregel gilt: Vergleichen Sie eine grobe Abschätzung der benötigten Tastaturanschläge und Benutzergesten, um ein bestimmtes Ziel zu erreichen, und wählen Sie dann die Methode, welche am wenigsten Arbeit verursacht. Beachten Sie dabei, dass sie manuell erstellte Artefakte oft neu erstellen oder anpassen müssen!

Verstehen Sie diesen Ratschlag nicht falsch und generieren Sie nicht Dinge, die es bereits am Markt gibt. Wählen Sie aus zwischen Kaufen, Erstellen und der Verwendung von Open-Source-Software. Oft haben andere schon die notwendige Arbeit geleistet, und Sie können eine Menge Tipp- und Denkarbeit sparen, die bei der Neuerfindung des sprichwörtlichen Rades anfallen würde.

Hier zwei konkrete Beispiele für diese Best Practice:

Generierung von Komponenten-Konfigurationen

Im letzten Kapitel wurde bereits erläutert, dass Frameworks und DSLs insofern sehr gut zusammenpassen, als man DSLs als „Konfigurationssprache“ für Frameworks verwenden kann. Diese Konfigurationen sind bei mächtigen Plattformen oft kein Quellcode, sondern Konfigurationsfiles im weitesten Sinne, heutzutage oft XML. Derartige Konfigurationen lassen sich meist sehr einfach aus dem Modell generieren, da die nötigen Informationen im Modell oft sowieso schon vorliegen. Beispiele:

- Die EJB-Deployment-Deskriptoren lassen sich aus dem Modell erstellen, wenn man einige zusätzliche Annotationen definiert, die z.B. das Transaktionsverhalten von Operationen definieren.
- Die Ablaufdefinitionen für das Web-Ablaufsteuerungs-Framework Struts lassen sich aus Zustandsdiagrammen oder Aktivitätsdiagrammen generieren. Auch diese Konfigurationen werden in XML notiert.
- Hibernate, ein Java Persistenz-Framework, arbeitet auch mit XML-basierten Konfigurationsdateien, welche die zu persistierenden Klassen und Attribute angeben. Auch dies lässt sich leicht generieren.
- CORBA IDL ist ein weiterer Kandidat. Die Erzeugung von IDL-Definitionen für Modellelemente kann auch leicht automatisiert werden. Aus diesen werden dann – in einem weiteren Generierungsschritt – die CORBA-Artefakte erstellt.

Das CORBA-Beispiel ist auch insofern interessant, als es zeigt, dass die generierten Artefakte selbst wieder als Eingabeformat in weitere Generatoren dienen können (Kaskadierung). Natürlich könnten wie die CORBA-Stubs und -Skeletons auch direkt aus unserem Modell erstellen. Der Gedanke der Wiederverwendung verbietet diesen Ansatz allerdings. Dies zeigt, dass das Aneinanderreihen von mehreren Transformationen (mit wohldefinierten Zwischenformaten, hier: CORBA IDL) ein sehr nützlicher Ansatz ist (ein Beispiel für solche kaskadierte Domänenarchitekturen findet sich am Ende von <Ref CS II>).

Unterstützung der Systemarchitektur

Kein System lässt sich nur unter Betrachtung der *Software*architektur bauen. Es ist immer auch eine Systemarchitektur vonnöten, die etwas über vorhandene Maschinen, Prozesse und die Verteilung der Software auf diese aussagt. Wir sind zwar trotz der Fortschritte im Bereich MDSD noch nicht in der Lage, Hardware zu generieren, aber wir können das Deployment maßgeblich unterstützen. Angenommen wir wollen ein komplexes System mit vielen Komponenten auf einer Rechnerfarm ausführen. Dazu müssen die folgenden Dinge erledigt werden:

- Die Komponenten müssen auf den entsprechenden Maschinen installiert werden.
- Ggfs. müssen auf den richtigen Datenbankinstanzen die richtigen Tabellen angelegt und initialisiert werden.
- Ggfs. muss die Infrastruktur konfiguriert werden, ein Beispiel sind Load-Balancer.

Auch die betreffenden Systemstrukturen können einfach modelliert werden, beispielsweise mit UML-Deployment-Diagrammen; man kann dann auf der Basis dieser Modelle die nötigen Artefakte (bspw. Installations- und Konfigurationsskripte) generieren.

Ein Beispiel: In der Infrastruktur für verteilte eingebettete Systeme (siehe Kapitel <Ref CSII>) wird die Topologie des verteilten Systems in einem Modell definiert; des Weiteren werden die Netzwerkverbindungen zwischen den einzelnen Knoten angegeben. Der Generator erstellt dann die kompletten Images für die entsprechenden Knoten. Sie können direkt auf dem entsprechenden Systemknoten deployed und ausgeführt werden.

9.1.4. Generiere gut aussehenden Code – wann immer möglich

In vielen Fällen ist es unrealistisch, anzunehmen, dass Entwickler den generierten Code nie zu sehen bekommen. Selbst wenn Anwendungsentwickler den generierten Code nicht ändern (oder manuellen Code einfügen) müssen, so werden sie doch mit dem generierten Code konfrontiert, z.B. wenn sie mit herkömmlichen Werkzeugen die generierten Anwendungen debuggen oder wenn sie die Generatorkonfiguration überprüfen möchten. Wie kann sichergestellt werden, dass Anwendungsentwickler den generierten Code verstehen und keine Angst vor der Arbeit mit generiertem Code haben?

Das Vorurteil, dass man generierten Code nicht vernünftig lesen, verstehen und debuggen kann, sitzt tief. In einigen Fällen wird dieses Vorurteil zum Anlass genommen, ganz auf modellgetriebene Entwicklung zu verzichten.

Dem wirkt man dadurch entgegen, dass man, wann immer möglich, „guten“ Code generiert. Denken Sie bei der Erstellung von Templates an die Anwendungsentwickler, die mit dem generierten Code – zumindest in gewissen Situationen – arbeiten müssen.

Sie können verschiedene Dinge tun, um „guten“ Code zu generieren, unter anderem:

- Sie können Kommentare generieren. In den Templates haben Sie die meisten, wenn nicht gar alle notwendigen Informationen zur Hand, um sinnvolle Kommentare zu erzeugen. Üblicherweise sind generierte Kommentare nicht starre Texte, sondern basieren, wie der Rest der Templates auch, auf Informationen, die dem Modell entnommen wurden.

- Wegen der oftmals mangelhaften Möglichkeit, in Templates mit Whitespace umzugehen, müssen Sie entscheiden, ob die Templates oder der generierte Code richtig eingerückt werden sollen (der Generator kann oft nicht unterscheiden, ob eine Einrückung zur Strukturierung der Templates oder zur Strukturierung des generierten Codes dienen soll). Oft ist daher die beste Lösung, sich auf die Lesbarkeit der Templates zu konzentrieren und einen nachgelagerten Pretty Printer/Beautifier zu verwenden, welcher den generierten Code korrekt formatiert. Pretty Printer sind für die allermeisten gängigen Programmiersprachen sowie XML erhältlich.

- Eine dritte und sehr nützliche Technik ist die Verwendung von so genannten „Location Strings“, die im generierten Code die

verwendete Transformation bzw. das verwendete Template und die zugrunde liegenden Modellelemente identifizieren. Es ist ebenfalls speziell für Debugging-Zwecke anzuraten, einen Zeitstempel mit Information über die Version des verwendeten Templates mit einzubauen. Ein „Location String“ könnte zum Beispiel wie folgt aussehen:

```
[2003-10-04 17:05:36]
GENERATED FROM TEMPLATE SomeTemplate
MODEL ELEMENT aPackage::aClass::SomeOperation().
```

Die Verwendung dieser Best Practice ist vor allem (aber nicht nur) wichtig, um die Akzeptanz der Codegenerierung bei den Entwicklern zu erhöhen. Im Wesentlichen läuft es darauf hinaus, an generierten Code die gleichen Qualitätsmaßstäbe und Stilrichtlinien wie an nicht-generierten Code anzulegen. Insbesondere ist die logisch korrekte Einrückung in Bezug auf Kontrollstrukturen wichtig. Wenn Sie einen qualitativ hochwertigen Anwendungsprototypen „templatisieren“, dann sollten Sie bereits alle notwendigen Kommentare zur Hand haben, um sie in die Templates zu integrieren.

Die einzige Einschränkung/Ausnahme zu dieser Best Practice besteht in Bezug auf die Generierung von Performance-optimiertem Code. In solchen Fällen muss man oft auf Konstrukte zurückgreifen, unter denen die Lesbarkeit des Codes leidet. Diese Fälle sollten explizit identifiziert und beschrieben werden, und der generierte Code sollte getrennt von anderem Code verwaltet werden. Übrigens gilt die Tatsache, dass man Struktur für Performance opfert, ja nicht nur für generierten Code, sondern ist auch bei manuell geschriebenem Code gängig.

Die eigentliche Aussage hinter dieser Best Practice beschränkt sich nicht nur auf generierten, sondern auch auf handgeschriebenen Code: Quellcode wird längst nicht mehr primär für Maschinen geschrieben, sondern vielmehr zum Lesen durch andere Benutzer des Quellcodes.

9.1.5. Modellgetriebene Integration

In vielen Fällen müssen Sie Ihr mittels MDSD entwickeltes System mit bestehenden Systemen und Infrastrukturen integrieren. Softwareprojekte, die auf der grünen Wiese stattfinden, sind bekanntlich eher selten, meistens wird Software im Kontext von existierenden Systemlandschaften entwickelt, welche noch einiges an Lebensdauer vor sich haben. Zusätzlich besteht oft der Wunsch, einige Legacy-Systeme zu ersetzen, sie im Laufe der Zeit inkrementell durch neue, besser passende Funktionalität abzulösen, die dann mittels modernerer Technologien realisiert wird.

Integration umfasst typischerweise die systematische Abbildung verschiedener APIs sowie die notwendigen Daten- und Protokolltransformationen zwischen diesen APIs. Je nach Integrationsstrategie muss in die neu zu erstellende Anwendung oder in die zu integrierenden Anwendungen Integrationscode generiert werden. Generierte Artefakte gehen oft über reinen Integrationsquellcode hinaus und beinhalten auch entsprechende Datenkonvertierungsskripte zur einmaligen Benutzung.

Daten- und Schnittstellenabbildungen zwischen Systemen sind am wertvollsten, wenn sie in einem Modell abgebildet sind. Gehen Sie das Thema Integration also als Teil von MDSD an und lassen Sie es nicht etwa außen vor. Definieren Sie eine technische Subdomäne (also eine spezifische DSL) für modellgetriebene Integration. Wenn es komplizierter wird, erwägen Sie, eine technische Subdomäne für jedes der zu integrierenden Systeme zu benutzen. Definieren Sie DSLs in den entsprechenden Subdomänen, die es ermöglichen, die Abbildungsvorschriften zwischen Modellelementen und bestehenden Legacy-Anwendungen genau zu beschreiben. Benutzen Sie Automatisierung, um das Abschalten von Legacy-Systemen so zu vereinfachen, dass dafür kein Expertenwissen nötig ist. Integration mit existierenden Systemen ist eine Stärke und nicht (wie gelegentlich behauptet wird) eine Schwäche modellgetriebener Softwareentwicklung.

Im Falle der Integration zweier unterschiedlicher modellgetriebener Systeme kann es sinnvoll sein, den Integrationscode so zu strukturieren, dass das notwendige Wissen über Implementierungstechnologien nicht unnötig in Template-Quellcode in beiden Systemen dupliziert wird.

Für einfache Integrationsaufgaben kann eine technische Subdomäne unnötig kompliziert sein, und es kann ausreichen, UML Tagged Values oder äquivalente Konzepte in den Modellen zu benutzen, die für die Modellierung der eigentlichen Anwendung verwendet werden. Benutzen Sie diesen Ansatz aber nur, wenn die Integrationsinformation nicht unnötig die Übersichtlichkeit des Modells beeinträchtigt und wenn die Integration von permanenter Natur ist und sich nicht etwa auf ein Legacy-System mit begrenzter Lebensdauer bezieht.

In letzterem Fall sollte sichergestellt sein, dass Integrationscode möglichst einfach wieder entfernt werden kann, sobald er nicht mehr benötigt wird. Ansonsten trägt toter Code nur zur Verunreinigung der Architektur bei. Benutzen Sie eine Anti-Corruption Layer [Eva03]. Spezifizieren Sie Abbildungen mittels externen Modellmarkierungen.

Versuchen Sie, die schrittweise Abschaltung von Legacy-Systemen so weit zu automatisieren, dass in einer DSL einfach die abzuschaltenden Teile gekennzeichnet werden, damit erleichtern Sie das Leben

zukünftiger (Entwickler-)Generationen – denn die Personen, die in drei Jahren die letzten Teile eines Legacy-Systems abschalten, wissen womöglich nur sehr wenig über die Details des Integrationscodes.

9.1.6. Trennen von generiertem und nicht-generiertem Code

Wenn nur Teile der Anwendung generiert werden (was heutzutage in vielen Szenarien der Fall ist), dann müssen die Lücken mit manuell erstelltem Code ausgefüllt werden. Allerdings beinhalt die Modifikation von generiertem Code eine Fülle an Problemen in Sachen Konsistenz, Build-Management, Versionierung und Konsistenz zwischen Modell und generiertem Quellcode, vor allem bei der Neugenerierung (auch wenn der letztere Aspekt von modernen Generatoren rein *technisch* beherrscht wird).

Wenn die Dateien mit generiertem Code nie modifiziert werden, dann kann bei Bedarf der gesamte generierte Code einfach gelöscht werden und eine Neugenerierung erfolgen. Wenn generierter Code manuell geändert werden muss, so darf dies nur innerhalb speziell dafür vorgesehener geschützter Bereiche erfolgen, die bei der Neugenerierung nicht überschrieben werden.

Halten Sie daher generierten und nicht-generierten Code in separaten Dateien (meist ist es sogar sinnvoll, getrennte Verzeichnisbäume zu verwenden!). Modifizieren Sie niemals generierten Code. Entwerfen Sie eine Architektur, welche klar definiert, welche Artefakte generiert werden und welche manuell erstellt werden müssen. Im Rahmen dieser Architektur wird auch definiert, wie generierter und nicht-generierter Code im Rahmen der Zielarchitektur kombiniert werden. In der OO-Welt sind Interfaces, abstrakte Klassen, Delegation sowie Entwurfsmuster wie Factory, Strategy, Bridge oder Template Method (siehe [GHJ+94]) geeignete Mittel. In nicht-OO-Sprachen kann man z.B. mit Includes arbeiten.

Die Trennung von generiertem und nicht-generiertem Code zwingt die Architekten, ein Design zu wählen, welches unterschiedliche Aspekte sehr sauber trennt – in unsern Augen eine sehr erstrebenswerte „Nebenwirkung“.

Als Folge dieser Best Practice kann generierter Quellcode somit als Wegwerfprodukt angesehen werden, welches noch nicht einmal versioniert werden muss, wodurch Konsistenzprobleme reduziert werden können. Allerdings gilt, dass in allen Fällen, bei denen manuell implementierter Code mit generiertem Code zusammentrifft, Inkonsistenzen auftreten können, falls das Modell so geändert wurde,

dass es mit dem handgeschriebenen Code strukturell oder semantisch nicht mehr kompatibel ist – selbst wenn der manuelle Code vollständig in separate Klassen ausgelagert ist.

Ein weiterer wichtiger Grund für die Trennung von generiertem und nicht-generiertem Code ist die Tatsache, dass manuell geschriebener Code üblicherweise der Code ist, welcher bei der Erzeugung von neuen Varianten eines Produkts manuell angepasst werden muss – produktübergreifend identischer Code ist Teil der Applikationsplattform, und der aus dem Modell ableitbare Code wird generiert. Somit ist die Trennung von generiertem und nicht-generiertem Code wichtig für die effiziente Verwaltung von Varianten und hilft bei der Identifizierung von Variationspunkten in der Software-Systemfamilie.

Falls aus Performance-Gründen, oder weil die Zielsprache keine Möglichkeiten bietet, verschiedene Artefakte zusammenzuführen, manueller Code direkt in generierten Code eingefügt werden muss, dann ist die Einführung von geschützten Bereichen unvermeidbar. Versuchen Sie, dies aber wirklich nur im Rahmen dieser Ausnahmen zu tun!

Die vorliegende Best Practice kann verallgemeinert werden, in dem Sinne, dass verschiedene Generatoren verwendet werden können, um verschiedene Teile des Systems zu generieren, wie zum Beispiel im Zusammenhang mit verschiedenen technischen Subdomänen. Manuell geschriebener Code kann dann als Resultat eines sehr speziellen „Generators“ angesehen werden (des Programmierers). Die Architektur muss natürlich all diese verschiedenen Aspekte entsprechend berücksichtigen.

Der folgende zweite Teil des Kapitels beginnt mit einer Betrachtung, wie man generierten und nicht-generierten Code konkret integrieren kann.

9.2. Best Practices – Technologie

9.2.1. Explizite Integration von Generat und manuellen Anteilen

Explizite Integration bedeutet, dass der generierte Code zunächst völlig unabhängig von manuell erstelltem Code ist. Es liegt nun am Entwickler, die Artefakte zu integrieren. In den seltensten Fällen sind die beiden Arten von Code wirklich völlig unabhängig (es gibt oft Abhängigkeiten von nicht-generiertem zu generiertem Code), sie sollen ja üblicherweise im Rahmen eines Systems gemeinsam verwendet werden.

Die einfachste Methode der Integration besteht darin, dass im generierten Code „reservierte Bereiche“ vorgesehen sind, in die der Entwickler dann manuellen Code einfügen kann. Die Bereiche sind speziell, vom Generator lesbar, gekennzeichnet, so dass bei folgenden Generatorläufen der manuell programmierte Code nicht überschrieben wird.

UML-Werkzeuge arbeiten üblicherweise so. Hier werden aus Modelldaten oft Klassenrümpfe generiert, in die der Entwickler dann das „Verhalten“ einbaut. Abb. 81 zeigt dies.

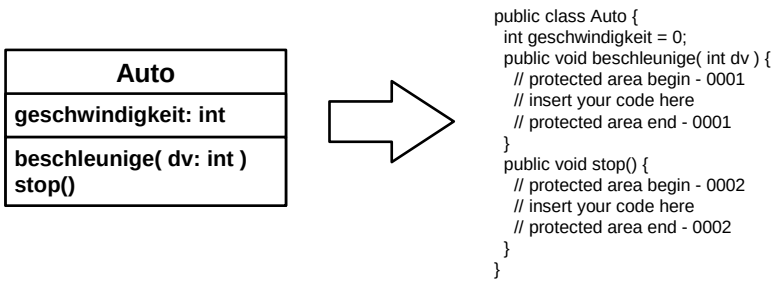


Abb. 81 *Generierter Code mit Protected Areas*

Wie weiter oben beschrieben, hat dieses Vorgehen allerdings einige Nachteile:

- Der Generator ist komplexer, weil er die Verwaltung, Erkennung und Erhaltung der geschützten Bereiche bewältigen muss.
- Es ist nicht immer einfach möglich, den Inhalt der geschützten Bereiche zu erhalten. Es geht in der Praxis immer wieder Code verloren.
- Die Trennung von generiertem und manuell erstelltem Code wird aufgeweicht, da beide in derselben Datei/Klasse stehen.

Der letzte Punkt ist dabei der problematischste, da der Entwickler im generierten Code arbeiten muss. Dazu muss er diesen zunächst verstehen, dies ist nicht immer einfach. Es ist insbesondere auch gar nicht immer sinnvoll. Außerdem werden andere Aspekte der Softwareentwicklung erschwert, zum Beispiel die Versionsverwaltung.

Daher sollten andere Mechanismen der Integration in Betracht gezogen werden. Eine Lösung, die oft zum Einsatz kommt (und auch die Integration mit der Plattform besorgt) ist die so genannte 3-schichtige Implementierung. In vielen Fällen bestehen zu erstellende Systemkomponenten aus drei Arten von Funktionalität:

- Funktionalität, die für alle Komponenten einer bestimmten Art gleich ist.

- Funktionalität, die für jede Komponente anders ist, aber aus dem Modell generiert werden kann.
- Funktionalität, die vom Entwickler von Hand implementiert werden muss.

Eine bewährte Technik, dies in objektorientierten Sprachen zu implementieren, besteht aus dem in Abb. 82 gezeigten Ansatz:

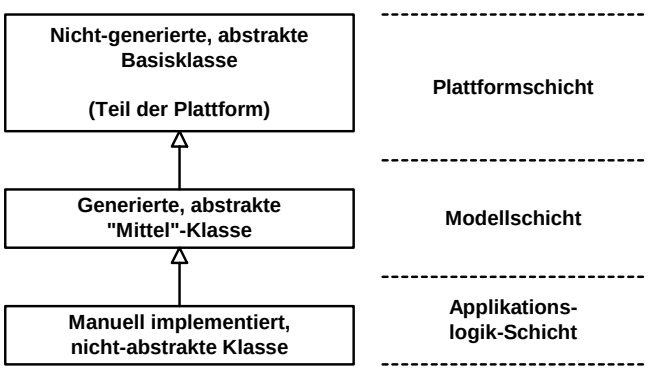


Abb. 82 *Die drei Schichten der dreischichtigen Implementierung*

Als Teil der Plattform wird eine abstrakte Basisklasse für alle Komponenten eines bestimmten Typs implementiert. Der Generator generiert für jede Komponente eine von dieser Klasse erbende abstrakte Zwischenklasse, welche alle Aspekte realisiert, die aus dem Modell abgeleitet (und damit generiert) werden können. Last but not least, erstellt der Entwickler eine nicht-abstrakte Implementierungsklasse, die von der generierten erbt und dann im System verwendet (instanziiert) wird. Diese manuell implementierte Klasse „füllt die Löcher“ in der generierten Mittelklasse auf. Dies kann z.B. recht elegant mit Hilfe der Gang-of-Four Patterns [GHJ+94] geschehen. Der folgende Abschnitt sowie Abb. 83 sollen dies erläutern:

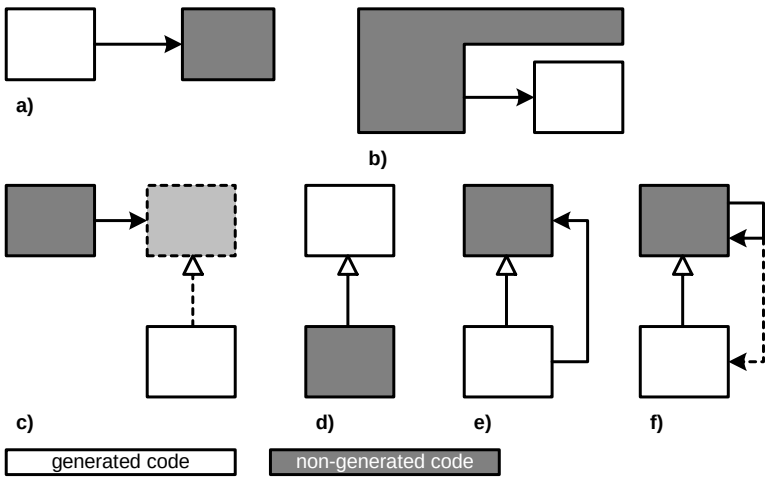


Abb. 83 *Pattern-basierte Integration von generiertem und nicht-generiertem Code*

In Abb. 83, Fall a) ruft generierter Code nicht-generierten Code auf. Dies ist natürlich naheliegend bis trivial. Man wird auch in generiertem Code immer auf nicht-generierten Code zugreifen, man verwendet ja weiterhin bewährte Bibliotheken. Man sollte diesen Fall aber noch etwas umfassender interpretieren: Man sollte immer nur so wenig Code wie möglich generieren und, wann immer möglich, auf fertigen, getesteten Code zurückgreifen, der in Form der Plattform in der Domänenarchitektur seinen Platz findet.

Der Fall b) ist schon etwas weniger offensichtlich. Hier ruft manuell implementierter Code den generierten Code auf. Dazu muss der manuell geschriebene Code den generierten kennen, was ggfs. zu unangenehmen Abhängigkeiten im Build-Prozess führen kann. Abhilfe kann der Fall c) schaffen. Der generierte Code kann von einer manuell erstellten Klasse erben, bzw. ein manuell erstelltes Interface implementieren. Der manuell erstellte Code kann damit gegen die Schnittstelle programmiert werden. Zur Laufzeit wird dann – z.B. mittels einer Factory-Operation – eine Instanz einer generierten Klasse instanziiert.

Um das obige Beispiel mit den geschützten Bereichen nochmals aufzugreifen: Eine Möglichkeit, dies zu umgehen ist, Vererbung zu verwenden, wie im Fall d) dargestellt. Hier erbt eine Implementierungsklasse von der generierten Basisklasse. Die Implementierungsklasse überschreibt die generierten Operationen und stellt damit das Verhalten zur Verfügung. Eine Factory kann wieder bei der Instanziierung helfen. Natürlich kann – wie im Falle e) gezeigt – generierter Code von einer nicht-generierten Klasse erben und ggfs. deren Operationen aufrufen.

Interessant ist auch Fall f), wo eine manuell implementierte Klasse, bzw. deren Operationen, die Operationen einer generierten Unterklasse aufrufen. Das ist praktisch eine Anwendung des Template-Method Patterns. Die nicht-generierte Oberklasse definiert einige abstrakte Operationen, die die generierte Klasse überschreibt. Andere Operationen der Basisklasse rufen die abstrakten Operationen auf. Eine Factory hilft wieder bei der Instanziierung.

Dieses Vorgehen der Integration von generierten und nicht-generierten Codes erfordert oft eine mehrstufige Generierung. Der Anwendungsentwickler kann „seine“ Anwendungslogik in den untersten Ebene natürlich erst implementieren, wenn die Mittelschicht generiert wurde. Andererseits wird es oft noch einen weiteren Generierungs-, Kompilierungs- und Build-Schritt geben, welcher die generierten Artefakte, die der Plattform und die manuell implementierten gemeinsam verwendet und zu einer vollständigen Applikation zusammenstellt.

Man kommt also oft nicht darum herum, zwei Generatorläufe auszuführen:

- Der Erste liest bestimmte Modellelemente und generiert eine Reihe von Basisklassen; also eine Art API, gegen die der Entwickler dann seine manuell zu schreibende Anwendungslogik implementiert.
- Der zweite Lauf des Generators generiert dann alle Modellelemente und verwendet diese sowie die manuell vom Entwickler erstellten, um die vollständige Anwendung zu erstellen (je nach Plattform enthält dieser Schritt auch Dinge wie Kompilieren, Linken, Packen etc.)

Um zu erzwingen, dass der Entwickler auch tatsächlich von der generierten Klasse erbt, kann man Dummy Code (siehe nächster Abschnitt) verwenden.

Bei Verwendung der richtigen Features der Zielprogrammiersprache kann dieser Ansatz auch dazu dienen, bestimmte architekturelle Constraints zu erzwingen. Betrachten wir das Diagramm in Abb. 84.:

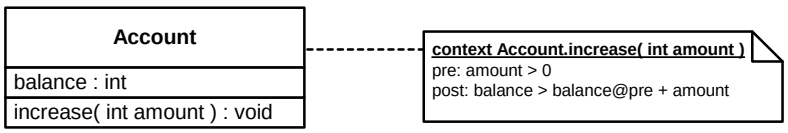


Abb. 84 *Einfaches UML-Modell mit Constraint*

In diesem Fall geht es darum, die im Modell als OCL definierte Pre- und Postcondition im generierten Code zu verankern, und zwar so, dass es nicht möglich ist, dass der Programmierer diese umgeht. Unter

Verwendung von geschützten Bereichen könnte ungefähr folgender Code generiert werden:

```
// generated
class Account {
  int balance;
  public void increase( int amount ) {
    assert( amount > 0 ); // precondition
    int balance_atPre = balance; // postcondition
    // --- protected area begin ---

    // --- protected area end ---
    assert( balance = balance_atPre + bamount );
                  // postcond.
  }
}
```

Das funktioniert natürlich so nicht wirklich. Der Entwickler kann ja jederzeit den Pre- und Postcondition-Code löschen. Die Variante mit einfacher Vererbung ist auch nicht praktikabel. Der Entwickler würde die Operation *increase()* überschreiben und damit würde die Überprüfung der Constraints umgangen. Alternativ könnte man zusätzliche Operationen definieren: *increase_pre()* und *increase_post()*, die die Pre- und Postcondition überprüfen. Die Implementierungsoperation in der Unterklasse müsste dann diese Operationen aufrufen. Allerdings kann der Entwickler auch dies vergessen. Es gibt eine recht elegante Lösung dieser Probleme unter Verwendung des Template-Method Patterns:

```
// generated
public abstract class Account {
  int balance;
  public final void increase( int amount ) {
    assert( amount > 0 ); // precondition
    int balance_atPre = balance; // postcondition
    increase_internal(amount)
    assert( balance = balance_atPre + amount );
                  // postcond.
  }

  protected abstract void increase_internal(int amount);
}
```

Hier wird die Operation *increase()* komplett generiert. Sie enthält Pre- und Postcondition-Code und ruft intern, zur Durchführung des eigentlichen Verhaltens, eine Operation *increase_internal()* auf. Diese ist als abstrakte Methode definiert, die vom Entwickler manuell zu schreibende Unterklasse muss die Operation implementieren.

```
// manually written code
class AccountImpl extends Account {
  protected void increase_internal(int amount) {
    balance += amount;
```

```
  }
}
```

Da das externe Interface (die *public*-Methode) immer noch *increase()* ist, muss ein Client der Klasse immer diese Operation aufrufen und kann die Checks damit nicht umgehen (*increase_internal()* ist *protected*). Ein Entwickler kann die *increase()*-Operation in der Unterklasse auch nicht überschreiben (um die Checks zu umgehen), da diese *final* ist. Das Problem ist damit bombensicher und elegant gelöst.

9.2.2. Dummy Code

Oft muss man, um ein konsistentes System zu bekommen, den Entwickler dazu zwingen, bestimmte Dinge zu tun. Im Rahmen von dreischichtiger Implementierung ist es beispielsweise nötig, dass der Entwickler eine eigene Klasse *manuell* implementiert, die bestimmte Anforderungen erfüllen muss:

- Sie muss einer bestimmten Namenkonvention folgen (z.B. genauso heißen wie die Oberklasse, nur mit dem Suffix „Impl“).
- Sie muss von einer bestimmten (oft generierten) Klasse erben und ggfs. bestimmte Operationen überschreiben.
- Sie muss vielleicht zusätzlich noch ein bestimmtes Interface implementieren oder anderweitig bestimmte Operationen zur Verfügung stellen.

Da es sich hierbei um manuell entwickelte Klassen handelt, kann man diese Dinge nicht erzwingen, indem man sie einfach in Templates schreibt.

Was man allerdings tun kann, ist, Code zu generieren, der die entsprechenden Eigenschaften mit Hilfe des Compilers überprüft. Angenommen wir haben folgende generierte Klasse:

```
public abstract SomeGeneratedBaseClass
       extends SomePlatformClass {
  protected abstract void someOperation();
  public void someOtherOp() {
    // stuff
    someOperation();
  }
}
```

Weiterhin sei angenommen, dass der Entwickler von dieser Klasse erben, die Operation *someOperation()* überschreiben sowie die Klasse …*Impl* nennen und ein Interface *IExampleInterface* implementieren muss. Die korrekte Implementierung sieht dann folgendermaßen aus:

```
public abstract SomeGeneratedBaseClassImpl
       extends SomeGeneratedBaseClass
       implements IExampleInterface {
  protected void someOperation() {
    // do something
  }
  public void anOperationFromExampleInterface() {
    // stuff
  }
}
```

Wie kann man nur erzwingen, dass der Entwickler diese Klasse korrekt implementiert, und zwar selbst dann, wenn das Framework (wie es ja oft vorkommt) die Klasse nur im Rahmen einer Factory dynamisch instanziiert und damit keine Compiler-Prüfung möglich ist?

Die Lösung besteht darin, Dummy Code zu generieren, der nur zum Zwecke der Verifikation der beschriebenen Richtlinien dient:

```
public abstract SomeGeneratedBaseClass
       extends SomePlatformClass {
  // as before
  static {
    if ( false ) {
      new SomeGeneratedBaseClassImpl();
        // verifies that the class is present,
        // and that it is not abstract
      SomeGeneratedBaseClass a =
         (SomeGeneratedBaseClass)
            new SomeGeneratedBaseClassImpl();
        // verifies that the implemented class is
        // actually a subclass of SomeGeneratedBaseClass
      IExampleInterface x = new
        SomeGeneratedBaseClassImpl();
        // verifies that it implements the
        // IExampleInterface
      new SomeGeneratedBaseClassImpl().xyzOperation();
        // this would verify that the operation
        // xyzOperation is impemented in the class
    }
  }
}
```

Dieser Mechanismus stellt sicher, dass der Compiler entsprechende Fehlermeldungen ausgibt, *sobald die generierte Klasse generiert wurde*. Mit diesem Mechanismus wird der Entwickler gezwungen, die Subklasse richtig zu implementieren, Flüchtigkeitsfehler, die später ggfs. schwer zu finden sind, werden vermieden. Man beachte übrigens das *if (false)*, welches sicherstellt, dass der Code nie ausgeführt wird[40].

[40] Manche IDEs sind so clever, dass sie merken, dass der Code in dem *if (false)*-Zweig nie ausgeführt werden wird und „beklagen“ sich dann über

9.2.3. Technische Subdomänen

Große Systeme umfassen typischerweise eine Vielzahl von (technischen) Aspekten. Die Beschreibung all dieser Aspekte in einem einzigen Modell (und demnach mit einer einzigen DSL) ist eine sehr komplizierte Angelegenheit. Das Modell droht mit Details der verschiedenen Aspekte überladen zu werden und in seiner eigenen Komplexität zu versinken. Dazu kommt, dass eine bestimmte DSL (bzw. deren Notation) meist gut zur Beschreibung *eines* bestimmten Aspekts geeignet ist – aber meist nicht für die anderen Aspekte im System.

Betrachten wir als Beispiel eine UML-basierte DSL zur Beschreibung von Geschäftsprozesslogik. Zusätzlich müssen die Persistenz von Modellelementen sowie das GUI-Design/Layout beschrieben werden.

- Sie müssen persistente Modellelemente in der DSL entsprechend kennzeichnen, so dass entsprechender Code und RDBMS-Schemata (SQL DDL) generiert werden können. Es ist schwer, dies alles in einem einzigen Modell unterzubringen. Eine UML-basierte Sprache ist oft nicht für die Erfassung von solchen Aspekten geeignet.

- Ein GUI-Layout im Detail mit UML zu modellieren ist zwar theoretisch vielleicht denkbar, jedoch völlig unpraktisch, insbesondere da heutige GUI-Designwerkzeuge bereits sehr gute grafische DSLs im WYSIWYG-Stil und entsprechende Wizards zur Verfügung stellen. GUI-Design lässt sich nur durch echte Abstraktion weiter automatisieren, zum Beispiel indem die Plattform das Layout und Design stark standardisiert und dem Entwickler viele Entscheidungen abnimmt. Weitreichende Standardisierung ist in einigen Anwendungsbereichen sinnvoll (Geschäftsanwendungen, Administrationsoberflächen), während in anderen Bereichen zu starke Standardisierung als Einschränkung des notwendigen Designspielraums gesehen wird (Webseiten).

Zu viele Aspekte in einem einzigen Modell abzubilden, erschwert die Wartbarkeit und verhindert außerdem eine effiziente Aufgabenverteilung für verschiedene (Teil-)Teams.

Um derartige Probleme zu vermeiden, führen Sie eine Struktur von Technischen Subdomänen ein. Jede Subdomäne sollte mittels einer dafür

---

„unreachable Code“. In diesem Falle muss man die Bedingung eben etwas weniger offensichtlich formulieren.

passenden DSL modelliert werden. Modellieren Sie die verschiedenen Subdomänen mittels getrennter Modelle und führen Sie die verschiedenen Modelle im Generator zusammen. Um dies möglich zu machen, definieren Sie eine kleine Zahl von Gateway-Metaklassen, d.h. Metamodellelemente, welche in den Metamodellen mehrerer DSLs benutzt werden. Diese dienen zur Verknüpfung der Modelle der verschiedenen DSLs, wie in Abb. 85 gezeigt.

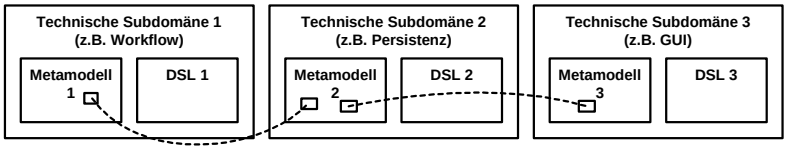


Abb. 85 *Verknüpfung verschiedener Technischer Subdomänen mittels Gateway-Metaklassen*

Dieses Vorgehen ist besonders nützlich, wenn Sie im Transformator/Generator die konkrete Syntax ignorieren (siehe <Ref Tool>, denn dann können die Gateway-Metamodellelemente in verschiedener konkreter Syntax in den unterschiedlichen Subdomänen abgebildet werden, während der Transformator/Generator allein mit der abstrakten Syntax arbeitet. Dadurch wird eine einfache und natürliche Integration der Subdomänen erzielt.

Bitte beachten Sie, dass das hier beschriebene Vorgehen das System in technische Subdomänen partitioniert, und nicht das System in funktionale Subsysteme strukturiert. Letzteres ist auch wichtig und notwendig, jedoch unabhängig von der Struktur der hier beschriebenen Subdomänen.

Eine sehr spezielle technische Subdomäne ist die modellgetriebene Integration. Abbildungsregeln und Wrapping-Regeln können sehr schön in einer maßgeschneiderten DSL definiert werden.

Generator-basiertes AOP ist eine alternative Möglichkeit, um querliegende technische Subdomänen abzubilden.

9.2.4. Proxy-Elemente

Die Integration verschiedener Subdomänen mittels Gateway-Metaklassen funktioniert prinzipiell, hat aber zur Folge, dass die entsprechenden Modellelemente in mehreren Modellen vorkommen (egal ob dies nun die Modelle verschiedener Subdomänen sind oder gemeinsame Elemente, beispielsweise Interfaces, in verschiedenen Partitionen (Teilmodellen)).

Um Duplizierung von Information zu vermeiden, ist es oft praktischer, mit Proxys oder Referenzen zu arbeiten. Man betrachte als Beispiel Abb. 86:

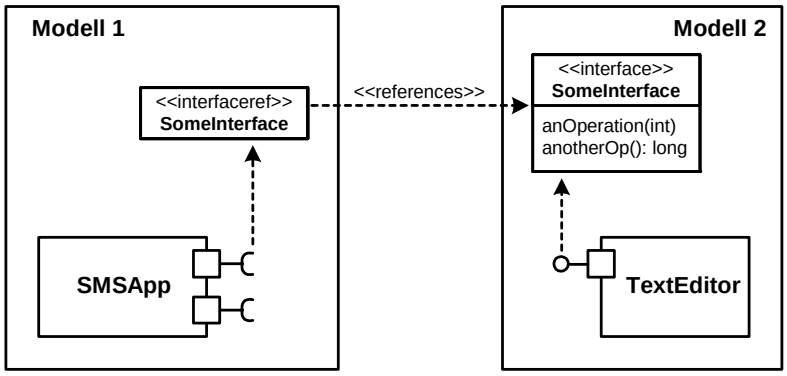


Abb. 86 *Beispiel für die Verwendung von Proxy-Elementen im Modell*

In Abb. 86 bezieht sich die Interface-Referenz in Modell 1 auf das gleichnamige Interface in Modell 2 – das Matching geschieht per Namensgleichheit. Natürlich muss dazu das Metamodell entsprechend erweitert werden; in Abb. 87 der betreffende Ausschnitt.

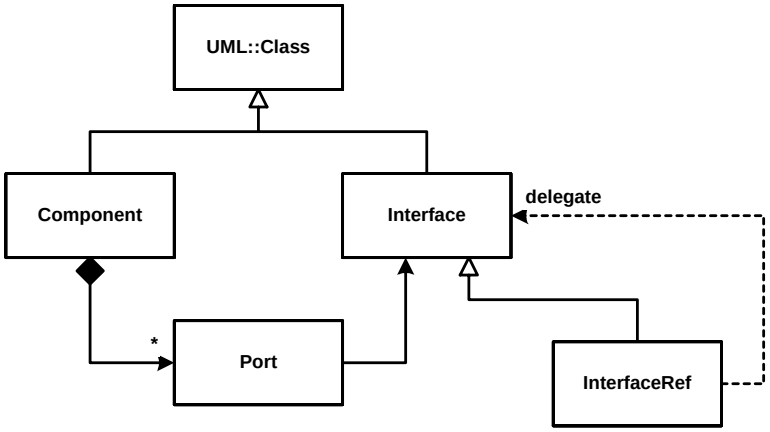


Abb. 87 *Metamodell mit Interface-Proxy*

Ein Port implementiert ein Interface; eine Interface-Referenz ist eine Unterklasse von Interface (entsprechend dem Proxy Pattern in [GHJ+94]) und referenziert den *Delegate*, also das Objekt, für welches der Proxy einsteht. Man beachte, dass diese Referenz zum Delegate im Rahmen des Metamodells (also im Generator, wenn alle Teilmodelle geladen sind) tatsächlich durch eine Assoziation realisiert wird; in den Modellen referenziert die Referenz das tatsächliche Objekt über andere Eigenschaften, hier eben über den Namen.

Dieses Vorgehen bedingt, dass der Generator bei der Generierung des Modelles 1 im obigen Beispiel das Modell 2 zur Verfügung hat. Der Generator muss die Referenz zu dem tatsächlichen Objekt auflösen.

Idealerweise ersetzt er die Assoziation des Ports auf die Referenz direkt durch eine Assoziation zum tatsächlichen Objekt, ansonsten kann der Proxy jegliche Operationsaufrufe weiterdelegieren. Aufgrund des Subtypverhältnisses ist dies technisch kein Problem.

Man beachte, dass man mittels dieses Prinzips beliebige Teilmodelle (seien dies nun Partitionen oder technische Subdomänen) unabhängig von den Fähigkeiten des Modellierungswerkzeuges sowie der konkreten Syntax zusammenführen kann. Dies ist in der Praxis eine äußerst nützliche Technik.

9.2.5. Externe Modellmarkierungen

Um die Transformation eines Ausgangsmodells in ein Zielmodell zu ermöglichen (beziehungsweise um Code zu generieren) ist es manchmal erforderlich, zur Generierungszeit Hilfsinformationen zur Verfügung zu stellen, welche auf das spezifische Zielmetamodell zugeschnitten sind. Das Hinzufügen dieser Informationen zum Quellmodell würde dieses unnötig mit Zielmodellkonzepten „verschmutzen“. Die OMG schlägt den Einsatz von Modellmarkierungen zwar vor, aber nur sehr wenige Werkzeuge unterstützen dieses Konzept ausreichend. Anstelle dessen empfehlen wir, diese Information außerhalb des Modells (z.B. in einer XML-Datei) zu beschreiben und dann diese externen Informationen zur Generierungszeit dem Generator/Transformator zur Verfügung zu stellen.

Wenn man es richtig angeht, kommt es dadurch nicht zu Inkonsistenzen, da der Generator Fehlermeldungen ausgeben kann, wenn für ein Modellelement die Informationen in den externen Modellmarkierungen fehlen oder Informationen für nicht existierende Modellelemente angegeben werden. Im Prinzip ist dieses Vorgehen ein Spezialfall der technischen Subdomänen.

9.2.6. Aspektorientierung und MDSD

Dieser Abschnitt setzt Kenntnisse über Aspektorientierte Programmierung (AOP) voraus. Einführendes Material bieten [Lad04] sowie [AOSD].

Als Aspekte im Sinne der AOP bezeichnet man querschnittliche Belange, die den Code einer Anwendung „horizontal“ durchziehen.

Typische Beispiele sind oft technischer Natur[41]: Transaktionalität, Persistenz, Logging.

Aspektorientierung hat zum Ziel, solche Aspekte im Rahmen einer Software-Systemfamilie innerhalb eines Moduls zu lokalisieren und damit leichter änderbar (bzw. konfigurierbar) zu machen. Querschnittliche Belange können nun auf verschiedenen Ebenen im Rahmen von MDSD adressiert werden:

- Der Generator kann verschiedene Modelle, die verschiedene Aspekte des Gesamtsystems repräsentieren (technische Subdomänen) lesen, auf Modellebene verweben und daraus dann Code generieren, der alle Aspekte gebührend berücksichtigt.
- Der Generator an sich adressiert in gewisser Hinsicht querschnittliche Belange automatisch. Wenn verschiedenste Artefakte aus einer Transformationsvorschrift oder einem Codegenerierungstemplate erstellt werden, muss man nur an dieser einen Stelle (in dem Template) eine Änderung vornehmen, um diese Änderung querschnittlich durch das ganze System (eben all die durch diese Transformation erstellten Artefakte) zu implementieren.
- Man kann durch bestimmte architekturelle Konstrukte querschnittliche Belange adressieren. Das klassische Beispiel sind Proxys + Interceptoren. In diesem Fall kann der Generator automatisch die benötigten Proxys erstellen; man kann dann zur Laufzeit nach entsprechender Konfiguration die benötigten Interceptoren einhängen.
- Schlussendlich kann man natürlich auch die Aspektorientierte Programmierung integrieren, indem man Aspekte (beispielsweise in Form von AspectJ) generiert, oder zumindest die Pointcuts[42] (welche definieren, wo der Aspekt im System eingreift). Ein Beispiel dafür findet sich am Ende von Kapitel <Ref CSII>.

Konkrete Aspekte, die man sehr einfach per Generator realisieren kann seien im Folgenden erläutert:

---

[41] Es gibt auch funktionale querschnittliche Belange, diese wollen wir hier allerdings der Einfachheit halber nicht betrachten.

[42] Definition der Stellen im Programmablauf, an denen ein bestimmter Aspekt eingewoben werden soll, bspw. alle Aufrufe von Methoden einer bestimmten Klasse.

- Threadsynchronisation: Wenn der Generator die Implementierung einer Queue erstellt, kann er den Sychronisierungscode automatisch einfügen, wenn die Queue aus verschiedenen Threads verwendet wird (was im Modell angegeben werden muss).
- Ressourcenallokation: Der Generator kann die Implementierung einer Factory generieren, die verschiedene Ressourcenallokationsstrategien unterstützt (Eager Acquisition, Lazy Acquisition, Pooling, siehe [POSA3]).
- Security: Der Generator kann für alle Komponenten im System Proxys erzeugen, die Berechtigungsüberprüfungen durchführen. Die Instanziierung von Komponenten kann dann über eine Factory passieren, die diese Proxys einfügt. Zu beachten ist dabei, dass der Generator zur Generierzeit introspektiven Zugriff auf die Komponenten hat: Er kennt ihr Interface, dessen Methoden, deren Parameter etc., da diese Dinge im Modell bzw. zumindest im Generator explizit vorliegen.

Eine weitere Alternative zur Behandlung von Aspekten besteht in der Verwendung einer entsprechenden (technischen) Plattform. Der Generator wird dann ggfs. nur die Konfiguration des entsprechenden Frameworks generieren.

- Einige Plattformen erlauben nur die Behandlung einiger bestimmter vordefinierter technischer Aspekte. Beispiel EJB: EJB Container sorgen für Security Checks, Transaktionen, Ressourcenallokation. Der Generator erstellt hier entsprechende Deployment-Deskriptoren.
- Andere Plattformen erlauben, bzgl. bestimmter vordefinierter Join Points beliebige Aspekte einzuhängen. CORBAs Portable Interceptors sind ein Beispiel dafür. Hier würde der Generator den Code zur Instanziierung eines entsprechend konfigurierten POAs bzw. ORB Cores durchführen.

Es stellt sich die Frage, welche dieser Möglichkeiten nun Verwendung finden soll. Diese Frage lässt sich nicht allgemein beantworten. Einige zu beachtende Einflussfaktoren kann man jedoch schon geben:

- Auf welcher Granularitätsebene müssen Aspekte behandelt werden? Wenn auf Plattformebene entsprechende Hooks vorhanden sind, so ist die Nutzung dieser sicherlich das einfachste Verfahren.

- Wenn die vom Aspekt beeinflussten Join Points[43] sowieso vom Generator erstellt werden, dann kann sich dieser um das Einflechten der Aspekte kümmern. In diesem Fall wird aus einem Aspekt des Generats praktisch automatisch ein „Modul“ im Generator (Strategies [GHJ+94] machen hier sehr viel Sinn!)
- Wenn Aspekte auf Stellen wirken, die nicht generiert werden, sondern manuell erstellt sind, und das Plattform-Framework keinen Zugriff auf die entsprechenden Join Points bietet, ist eine Aspektsprache wie AspectJ sicherlich die beste Wahl. In diesem Falle würde der Generator aufgrund des Modells nur noch entscheiden, welcher Aspekt auf welche Codeteile angewandt wird. Der Generator erstellt also die Konfiguration für den Aspectweaver, die Pointcuts. Ein Beispiel hierfür findet sich im hinteren Teil der zweiten Fallstudie (siehe <Ref CSII>).

Generell gilt wie immer: Zusätzliche Tools wenn möglich vermeiden! Eine Ausführlichere Abhandlung wie MDSD und AOSD/AOP miteinander in Bezug stehen, und wie beide gemeinsam verwendet werden können findet sich in [Voe04].

9.2.7. Beschreibende Metaobjekte

Wenn eine reichhaltige domänenspezifische Plattform für MDSD verwendet wird, dann braucht die Anwendung oft Informationen über Modellelemente zur Laufzeit, um die Plattform dynamisch zu steuern. Wie können Informationen aus dem Modell der generierten Anwendung zur Laufzeit zur Verfügung gestellt und mit generierten Artefakten assoziiert werden? Wie kann eine Brücke zwischen generiertem Code und Framework-Teilen gebaut werden?

Nehmen wir als Beispiel an, Sie wollen eine Anwendung mit einem fachlich aussagekräftigen Logging-Mechanismus ausstatten. Die Anwendung soll die Werte von Attributen von Instanzen generierter Klassen in eine Logdatei eintragen. Dazu werden sowohl die Werte der Attribute als auch die Attributnamen benötigt. Insbesondere in Sprachen, die über keinen Reflection-Mechanismus verfügen, ist die Realisierung einer solchen Funktionalität nicht ganz einfach (wenn man sie nicht von Hand implementieren will).

---

[43] Stelle im Programmfluss, an der mittels des betreffenden Aspektwerkzeugs potenziell ein Aspekt eingewoben werden kann.

Ein anderer Fall ist die Anreicherung von Objektattributen im Modell mit nützlichen Zusatzinformationen, wie zum Beispiel mehrsprachige Bezeichnungen für die Verwendung in GUIs oder reguläre Ausdrücke zur Überprüfung von einfachen Wertebereichseinschränkungen für die Werte von Attributen. Zur Laufzeit müssen Sie in der Lage sein, bequem auf diese Informationen zuzugreifen, um zum Beispiel GUIs dynamisch aufbauen zu können. Oft können Sie diese Art von Informationen nicht einfach in den bestehenden Klassen einbetten, sei es aus Gründen der Performance, der Vermeidung von Abhängigkeiten oder aufgrund anderer architektureller Einschränkungen.

Um dieses Problem zu lösen, benutzen Sie die zur Generierungszeit aus dem Modell verfügbaren Informationen, um Metaobjekte zu generieren, welche die generierten Artefakte beschreiben. Stellen Sie einen Mechanismus zur Verfügung, über den die generierten Artefakte auf ihre zugehörigen Metaobjekte zugreifen können. Stellen Sie sicher, dass die Metaobjekte ein generisches Interface besitzen, welches der domänenspezifischen Plattform zugänglich ist. Abb. 88 illustriert dieses Vorgehen.

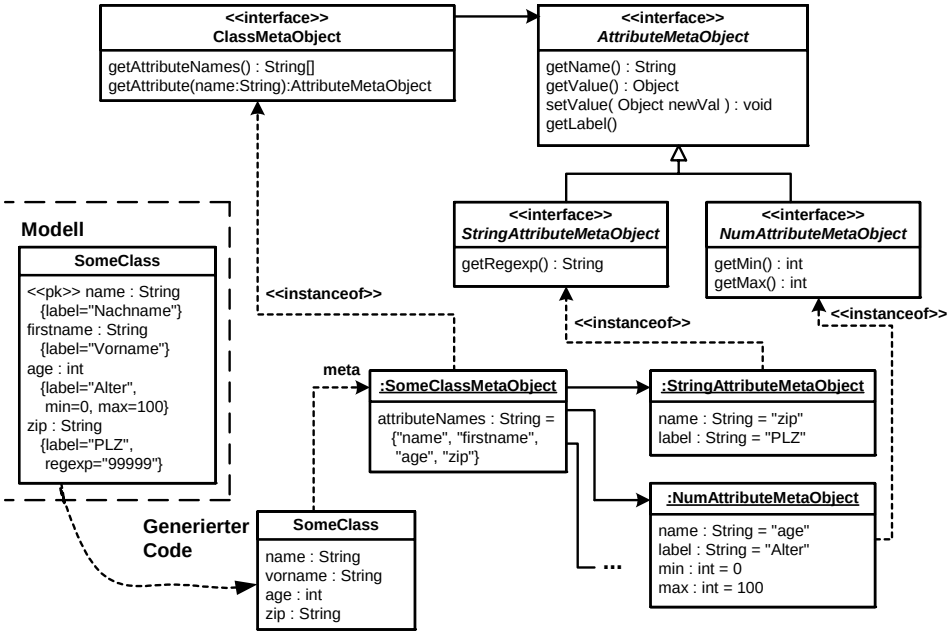


Abb. 88 *Prinzip der Beschreibenden Metaobjekte*

Dieses Vorgehen macht ausgewählte Teile des Modells innerhalb der Anwendung in einer effizienten Art und Weise zugänglich. Eine theoretische Alternative wäre die Speicherung von Teilen des Modells

mit der Anwendung, allerdings ist dann der Zugang zu Modellinformationen typischerweise langsam und umständlich, weswegen dieser Ansatz für die Praxis meist untauglich ist. Eine Ausnahme stellen kleine textuelle Modellfragmente dar (beispielsweise Werteinschränkungen), die durchaus zur Laufzeit mittels eines Interpreters ausgewertet werden können.

Es gibt verschiedene Wege, wie die Metaobjekte mit den Artefakten, die sie beschreiben, assoziiert werden können. Wenn das Artefakt vollständig generiert ist, kann oft eine *getMetaObject()*-Operation direkt in das Artefakt hineingeneriert werden. Wenn dies unpraktisch ist, kann anstelle dessen eine zentrale *Registry* verwendet werden, welche den Zugang zu Metaobjekten über eine Lookup-Funktion wie z.B. *MetaRegistry.getMetaObjectFor(anArtifact)* bieten kann. Die Implementierung dieser Operationen wird natürlich generiert.

Die Metaobjekte können nicht nur zur Beschreibung von Programmelementen benutzt werden, sie können auch dazu benutzt werden, mit Programmelementen zu arbeiten. Dies führt dann zu einem Generierten Reflection-Layer (s.u.)

9.2.8. Generierter Reflection-Layer

Metaobjekt-Protokolle wie zum Beispiel in [KRB91] sind eine Methode, um Programmiersprachenobjekte zur Laufzeit zu inspizieren, zu verändern oder „umzuinterpretieren“. Dies erfolgt typischerweise dynamisch (z.B. in Sprachen wie CLOS [Kos90]). Im Kontext von MDSD können sie zumindest ein *readonly*-Metaobjekt-Protokoll zur Verfügung stellen, so dass Klassen introspiziert oder Operationen dynamisch aufgerufen werden können. Dies funktioniert unabhängig davon, ob die zugrunde liegende Programmiersprache Reflection oder Ähnliches unterstützt, Sie können das auch in C/C++ umsetzen! Ein generisches Interface erlaubt Clients den Zugang zu allen Arten von generierten Klassen; der Einfachheit halber illustrieren wir das hier mittels Java:

```
public interface RClass {
    // initializer – associates with base-level object
  public setObject( Object o );
    // retrieve information about the object
  public ROperation[] getOperations();
  public RAttribute[] getAttributes();
    // create new instance
  public Object newInstance();
}
public interface ROperation {
    // retrieve information about op
```

```
  public RParameter[] getParams();
  public String getReturnType();
    // invoke
  public Object invoke(Object[] params)
}
public interface RAttribute {
    // retrieve information about op
  public String getName();
  public String getType();
    // set / get
  public Object get();
  public void set( Object data );
}
```

Die Implementierung dieses Interfaces für die betreffenden Klassen wird generiert. Während der Generierung haben Sie Zugriff auf alle relevanten Informationen aus dem Modell.

Da obige Interfaces generisch sind, können die Plattform oder andere dynamische Werkzeuge über diese Art von reflektivem Interface mit den Daten arbeiten.

9.2.9. Modulare Transformationen

Um die Wiederverwendung von Transformationsteilen zu ermöglichen, ist es ratsam, Transformationen zu modularisieren. Dies kann (und sollte) je nach Transformationssprache wie in klassischen Programmiersprachen durch die Konzepte der strukturierten oder objektorientierten Programmierung erfolgen. Dazu zählen Subroutinen/Prozeduren, Klassen oder lose gekoppelte Komponenten (oft als *Cartridges* bezeichnet), die dann z.B. für die Generierung verschiedener Schichten der Zielarchitektur zuständig sind und einzeln ausgetauscht werden können. Solche Strukturierungsmittel sind durchweg werkzeugspezifisch (vgl. Kapitel <Ref Tool>), aber sehr wichtig – nutzen Sie sie.

Abb. 89 zeigt, wie so etwas bei einem Template-basierten Ansatz aussehen kann. In dem Beispiel geht es darum, das Standard UML-Java Mapping so zu erweitern, dass JavaBeans generiert werden können. Dabei soll eben für jedes Attribut im Modell nicht nur ein Attribut in der Java-Klasse generiert werden, sondern auch passende Getter und Setter.

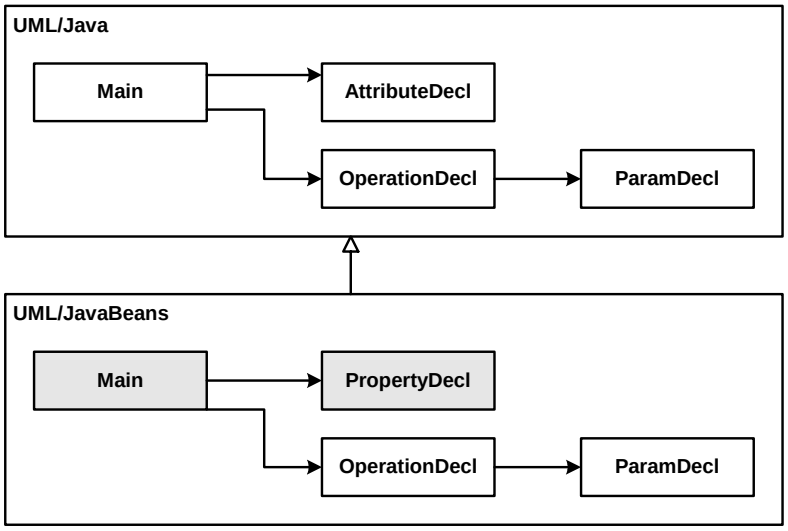


Abb. 89 *Template-Modularisierung*

Im Beispiel wird das *Main*-Template für die Bean neu definiert, so dass es für jedes Attribut die neu definierte *PropertyDecl* statt der regulären *AttributeDecl* verwendet. Die anderen Templates werden unverändert weiterverwendet.

Eine darüber hinausgehende Form der Modularisierung ist die Zerlegung einer Transformation in die Hintereinanderausführung mehrerer Transformationen. Betrachten wir das Beispiel in Abb. 90.

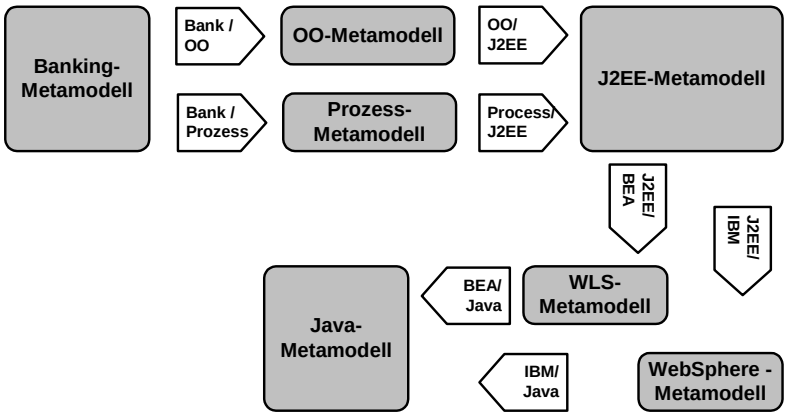


Abb. 90 *Modulare Transformation*

In dieser Abbildung wird das Modell einer Bankanwendung in eine auf J2EE ausführbare Applikation transformiert, und zwar für zwei verschiedene Applikationsserver, *BEA WLS* und *IBM Websphere*. Die Transformation erfolgt allerdings nicht in einem Schritt, sondern in mehreren Phasen:

- Zunächst werden die Prozessaspekte der Bankfachlichkeit auf ein Prozessmodell abgebildet, die restlichen Aspekte auf ein OO-Modell.
- Im folgenden Schritt werden diese beiden Modelle auf J2EE abgebildet.
- Danach wird dieses J2EE-Modell auf ein applikationsserverspezifisches Modell abgebildet.
- Im letzten Schritt werden diese beiden Modelle in Code überführt.

Stellen Sie sich vor, dass wir jetzt J2EE-Code für Callcenter-Applikationen erzeugen möchten. Da die Transformationen modular aufgebaut sind, müssen wir nur den ersten Teil der Transformation austauschen. Die nachfolgenden Transformationen können unverändert wiederverwendet werden, und uns bleibt eine Menge Arbeit erspart (siehe Abb. 91).

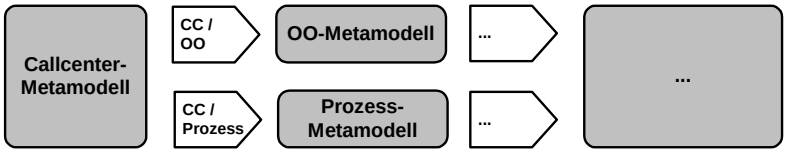


Abb. 91 *Anpassung des Transformationsprozesses an andere Fachlichkeit*

Wenn wir nun beide Software-Systemfamilien nach .NET portieren möchten, müssen wir nur den zweiten Teil der Transformation austauschen, wie in Abb. 92 gezeigt

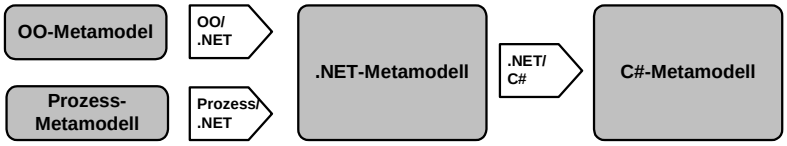


Abb. 92 *Anpassung des Transformationsprozesses an andere Technologie*

Hätten wir nur eine einzige, direkte Transformation verwendet, wäre diese Art der Wiederverwendung undenkbar.

Im Gegensatz zur OMG empfehlen wir allerdings, die Zwischenmodelle nicht von Hand zu modifizieren oder mit weiteren Markierungen zu versehen. Sie sind lediglich als „Schnittstelle“ zwischen den verschiedenen Transformationsschritten nötig.

Wenn wir in der Tat weitere Informationen benötigen, um spätere Transformationen zu steuern oder zu konfigurieren, dann sollten diese Markierungen in Form von Externen Modellmarkierungen realisiert werden. Dieses Vorgehen ist nicht nur sauberer im Sinne der Trennung

von Belangen („Aspektorientierte Modellierung“), sondern mit den heute verfügbaren Werkzeuge auch deutlich praktikabler.

Dieses Vorgehen zeigt auch noch einmal deutlich, dass das Zielmodell des einen das Quellmodell des anderen ist. Mehrstufige Transformationen stellen damit die Basis für Wiederverwendung von Transformationsschritten dar. Die kaskadierte modellgetriebene Entwicklung weist in dieselbe Richtung – wie am Ende von Kapitel <Ref CSII> illustriert.

10. Techniken der Codegenerierung

Entsprechend der in der Einleitung dieses Buchteils vorgestellten Struktur wollen wir uns nun mit allgemeinen Techniken befassen, die die Grundlage für den Bau oder die Auswahl von MDSD-Werkzeugen sind – also jene Aspekte, die man aus Domänenarchitekturen herausfaktorisieren kann, weil sie diesbezüglich unspezifisch sind – ohne die eine Domänenarchitektur aber nicht funktioniert. Die Basis bilden hierbei vor allem Techniken der Codegenerierung.

10.1. Codegenerierung – warum?

Wir haben bereits mehrfach erwähnt, dass Modellierung, Codegenerierung und Framework-Entwicklung gerade bei MDSD in einem sehr engen Zusammenhang stehen – z.B. kann aus einer DSL Konfigurationscode für ein in der MDSD-Plattform befindliches Framework generiert werden. Eine reiche, domänenspezifische Plattform wie in Kapitel <Ref Archi> beschrieben, erleichtert die Codegenerierung und verringert den Bedarf. Auf der anderen Seite bietet Codegenerierung auch Vorteile gegenüber rein generischen Ansätzen bzw. ergänzt diese.

10.1.1. Performance

In vielen Fällen wird Codegenerierung verwendet, weil man eine bestimmte Performance erzielen will, aber trotzdem eine gewisse Flexibilität erhalten möchte. Traditionelle OO-Methoden wie Frameworks, Polymorphismus oder Reflection sind bzgl. der erreichbaren Performance nicht immer ausreichend. Es wird dann die Konfiguration abstrakt angegeben (hier liegt auch die Flexibilität) und effizienter Code generiert.

10.1.2. Codegröße

Ein anderer Grund ist die Codegröße: Wenn man bereits zur Compile-Zeit weiß, welche Features zur Laufzeit benötigt werden, braucht der Generator nur die entsprechenden Teile in den Code zu integrieren. Dadurch kann das Image kleiner werden.

Umgekehrt kann die übermäßige Expansion von Konstrukten auf Quellcode-Ebene auch das Image deutlich vergrößern. Ein Beispiel dafür sind C++-Template-Instanziierungen.

10.1.3. Analysierbarkeit

Komplexe, generische Frameworks oder Laufzeitsysteme neigen dazu, programmiersprachliche Komplexität auf eine proprietäre Konfigurationsebene zu verlagern. Sie arbeiten stark interpretativ, was die statische Analyse von Programmeigenschaften und gelegentlich die Fehlersuche erschwert. Generierter Quelltext einer Standard-Programmiersprache hingegen besitzt diesbezüglich die Qualität von klassisch programmiertem Code. Ideal ist wie gesagt eine sinnvolle Balance beider Ansätze.

10.1.4. Fehlerfrüherkennung

Flexible Systeme verwenden oft schwache Typisierung, um Entscheidungen noch zur Laufzeit treffen zu können (*Object* in Java, *void** in C/C++). Damit wird die Fehlererkennung natürlich auch auf die Laufzeit verlegt, was oft nicht wünschenswert ist (dies ist auch ein Grund, warum diese Art von Programmierung in eingebetteten Systemen nicht besonders verbreitet ist). Durch die Verwendung von „statischen Frameworks“ können einige dieser Nachteile aufgehoben werden. Konfigurationen können bereits vor der Kompilierung als fehlerhaft erkannt werden; auch der Compiler hat üblicherweise mehr Informationen, so dass dieser bereits Fehlermeldungen ausgeben kann.

10.1.5. Plattformportabilität

Der klassische Anwendungsfall der MDA besteht darin, dass man Anwendungslogik unabhängig von der Implementierungsplattform „programmieren“ kann. Dies ermöglicht einen leichteren Umstieg auf neuere, potenziell bessere Plattformen.

10.1.6. Einschränkungen der (Programmier-)Sprache

Die meisten (Programmier-)Sprachen besitzen zum Teil unangenehme Einschränkungen in ihrer Ausdruckskraft, die sich aber mittels Codegenerierung umgehen lassen. Beispiele dafür wären in Java z.B. Typgenerizität (zumindest vor Version 1.5) oder der Downcast auf eine variable Klasse. Ein anderes Beispiel ist die Einführung objektorientierter Konzepte in eine nicht-objektorientierte Sprache.

10.1.7. Aspekte

Querschnittliche Merkmale wie Logging oder Persistenz eines Systems lassen sich mittels Codegenerierung üblicherweise lokalisiert (d.h. nicht-„verstreut“) implementieren. Wir werden dies im Zuge des Kapitels noch ausführlich beleuchten.

10.1.8. Introspection

Zu guter Letzt sei noch der Punkt Introspection erwähnt. Introspection beschreibt den (lesenden) Zugriff eines Programmes auf sich selber. Es erlaubt dem Programm, Informationen beispielsweise über Klassen, deren Attribute und Operationen zu ermitteln. In bestimmten Programmiersprachen (z.B. Java) wird dieser Mechanismus dynamisch unterstützt. Andere Sprachen, bspw. C++, bieten diesen Mechanismus nicht. Codegenerierung kann hier (statisch) Ersatz schaffen: Anstatt zur Laufzeit die Programmstruktur zu analysieren und daran eine bestimmte Funktionalität zu knüpfen, wird die Struktur zur Generierungszeit analysiert und auf dieser Basis Code erzeugt, der die jeweilige Funktionalität ansteuert.

10.2. Kategorisierung

10.2.1. Metaprogrammierung

Metaprogramme sind Programme, die, wenn sie ausgeführt werden, andere Programme (oft als Basisprogramme bezeichnet) erzeugen. Code-Generatoren sind also Metaprogramme, die Spezifikationen (oder Modelle) als Eingabeparameter besitzen und Quelltext als Ausgabe erzeugen.

Metaprogramme können bezüglich des erzeugten Programmes zu verschiedenen Zeiten ausgeführt werden:

- Völlig unabhängig vom Basisprogramm (d.h. vorher)
- Während der Erstellung (Kompilierung) des Basisprogrammes
- Zur Laufzeit des Basisprogrammes

Typische MDA/MDSD-Generatoren verwenden den ersten Ansatz. Dabei sind der manuell zu erstellende Teil des Basisprogramms sowie das Metaprogramm üblicherweise auch getrennt voneinander notiert. Auch ist das Generat vom manuell erstellen Code getrennt und muss vom Entwickler integriert werden (siehe <Ref Trans::IntCode>).

Systeme wie der C++-Präprozessor oder der C++-Template-Mechanismus bieten auch Möglichkeiten zur Metaprogrammierung. Hierbei sind Basisprogramm und Metaprogramm vermischt; auch das Ergebnis des Generierungsvorgangs enthält bereits manuell erstellten Code und Generat. Das erstellte Programm „weiß“ allerdings nichts mehr vom Metaprogramm. Man spricht hier von statischer Metaprogrammierung.

Lisp und CLOS [Kos90] erlauben die Ausführung von Metaprogrammen zur Laufzeit mittels eines so genannten Metaobjektprotokolls. Dies funktioniert deshalb, weil in Lisp Programme als Daten (Listen) repräsentiert werden. Metaprogramme können diese Listen modifizieren und damit zur Laufzeit Basisprogramme erstellen. Durch Änderung des Metaprogrammes aus dem Basisprogramm heraus kann außerdem die Semantik des Basisprogrammes angepasst werden.

10.2.2. Trennung/Verquickung von Programm und Metaprogramm

Im Falle der Vermischung von Metaprogrammen und Programmen gibt es eine gemeinsame (oder zumindest integrierte) Sprache für Programmierung und Metaprogrammierung, und die Quellcode-Anteile sind diesbezüglich nicht separiert, sondern vermischt. Dadurch können auch Aufrufbeziehungen zwischen den beiden konzeptionellen Ebenen bestehen, prinzipiell in beide Richtungen. C++-Template-Metaprogramming kann in diese Kategorie fallen, Lisp und CLOS auch.

Vermischung von Programmen und Metaprogrammen ist ein sehr mächtiger Ansatz. Allerdings kann das resultierende System schnell sehr komplex werden. Die Anwendungsgebiete sind daher in der Praxis relativ beschränkt. Eine weitere Implikation ist natürlich auch, dass die Zielsprache kein Parameter der Codegenerierung mehr ist.

Werden Programm und Metaprogramm getrennt, so geschieht die Erstellung des Systems in zwei klar getrennten Phasen. Das Metaprogramm wird ausgeführt und erzeugt als Output das Programm (oder Teile davon) und ist dann fertig. Das Programm „weiß" nichts vom Metaprogramm[44]! Die Trennung zieht sich also durch von der (Meta-)Programmierung bis zum Build-Prozess.

Die Frage, ob Programm und Metaprogramm in derselben Sprache geschrieben sind, ist hier irrelevant. Man kann z.B. in Java ohne weiteres ein Metaprogramm schreiben, welches dann Java, C++ oder C# als Ausgabe erzeugt.

Man hat in diesem Ansatz zwar kein Metaobjektprotokoll zur Verfügung; für typische architekturzentrierte, generative Programmierung eignet sich dieser Ansatz aber aufgrund der geringeren Komplexität besser. Der im Buch verwendete Generator (*openArchitectureWare* [oAW]) ist dabei bzgl. der Meta-Programmiersprache hybrid: Teile sind in Java implementiert (Metamodell der Anwendungsfamilie) und Teile in der Template-Sprache *Xpand*. Man kann damit die Vorteile einer ausdrucksstarken Template-Sprache mit der Mächtigkeit einer „richtigen" Programmiersprache kombinieren, ohne die Templates mit zu viel „Metaprogrammlogik" zu überfrachten.

10.2.3. Implizite oder explizite Integration des generierten mit dem nicht-generierten Code

Implizite Integration der beiden Codearten bedeutet, dass das Funktionsprinzip des Generators Code erstellt, der bereits eine Vermischung von generiertem und nicht-generiertem Code darstellt. Man braucht sich also über die Integration der beiden Kategorien keine Gedanken mehr zu machen.

Im anderen Falle ist das Generat zunächst einmal unabhängig von manuell erstellten Codeanteilen. Hier müssen die beiden Codearten dann geeignet zusammengeführt werden – dies wurde im vorigen Kapitel bereits erläutert.

---

[44] Das Metaprogramm kann durch beschreibende Metaobjekte teilweise über das Metaprogramm in Kenntnis gesetzt werden, siehe <Ref Trans::BP-Tech::DescrMO>.

10.2.4. Beziehungen

Im Allgemeinen kann man feststellen, dass es durchaus eine Beziehung zwischen den beiden Aspekten der Kategorisierung von Generatoren gibt:

- Systeme, in denen Programm und Metaprogramm getrennt sind, erstellen üblicherweise auch ein Generat, welches von manuell erstelltem Code getrennt ist und daher manuell integriert werden muss.
- Systeme mit einer Vermischung von Programm und Metaprogramm erfordern diese manuelle Integration nicht. Der Generator erstellt bereits das „kombinierte" System.

Wie bereits erwähnt, ist im Zusammenhang mit MDSD die Trennung von Programm und Metaprogramm sowie die explizite Integration von Generat und manuell erstellten Anteilen angeraten. Wir werden entsprechende Techniken später noch im Detail vorstellen Zunächst jedoch einige Beispiele für die andere Kategorie.

10.2.5. Beispiele für Verquickung von Programm und Metaprogramm

Der C++-Präprozessor ist ein System, das Programm und Metaprogramm vermischt. Die zum Einsatz kommenden Sprachen sind dabei unabhängig voneinander (man kann den C++-Präprozessor auch mit anderen Programmiersprachen einsetzen, er arbeitet rein textbasiert auf dem Quellcode). Da das System auf Makroexpansion basiert, erstellt der Präprozessor bereits Quellcode, der Generat und manuell erzeugten Code integriert. Das Folgende ist ein einfaches C++-Makro:

```
#define MAX(x,y) (x<y ? y : x)
```

Wird dieses Makro im Quellcode verwendet, so wird es vom Präprozessor expandiert, anhand der Regel oben. So wird aus

```
int a,b;
int greaterValue = MAX(a,b);
```

der folgende Ausdruck:

```
int greaterValue = (a<b ? a : b);
```

Der generierte Code ist also direkt an der Stelle eingesetzt worden, wo das zu expandierende Makro ursprünglich stand.

Ein anderes Beispiel (wieder in C++) sind Templates. Templates sind zunächst eine Möglichkeit, um in C++ Typgenerizität zu schaffen; Funktionen oder Klassen können mit Typen parametrisiert werden. Aufgrund der Tatsache, dass dieses Feature in C++ durch statische Codegenerierung erfolgt, kann dieses Verfahren auch zur so genannten Template-Metaprogrammierung verwendet werden [Ale01]. Es wird dabei durch die Auswertung der Templates Code generiert[45]. Verwendet wird dieses Vorgehen unter anderem zur Performance-Optimierung, Optimierungen der Codegröße, statische Optimierung des Programms für bestimmte Sonderfälle, zur Adaption von Interfaces, zur generativen Programmierung oder für andere interessante Dinge, die in [Ale01] beschrieben sind.

Auch hier sind Programm und Metaprogramm vermischt, die Integration ist deutlich enger als beim Präprozessor, da der Template-Mechanismus das Typsystem von C++ kennt und verwendet. Das resultierende System besteht auch hier bereits aus generiertem und manuell erstelltem Code.

Als Beispiel für Template-Metaprogrammierung soll die Anpassung der Codegröße dienen. Speziell geht es darum, abhängig von dem zur Verwendung kommenden Maximalwert einer Variable den kleinstmöglichen Datentyp (*short, int*) zu verwenden. Dies verringert den zur Laufzeit benötigten Speicherplatz. Das folgende Code-Fragment zeigt das Beispiel:

```
#define MAXVALUE 200
IF<(MAXVALUE<255), short, int>::RET i; // type of i is
short
```

Das *IF* im obigen Beispiel ist ein If-Statement, welches zur Compile-Zeit ausgewertet wird. Realisiert wird dies durch Templates (wie an den spitzen Klammern zweifelsfrei erkennbar ist). Spezifisch verwenden wir hier die so genannte partielle Template-Spezialisierung. Im Folgenden erläutern wir die Implementierung. Zunächst definieren wir ein Template, das drei Parameter besitzt: einen *bool*-Wert sowie Klassen (Typen) für den *true*-Fall sowie für den *false*-Fall.

```
template<bool condition, class Then, class Else>
struct IF {
```

---

[45] Da dieser Schritt im Rahmen des C++-Kompilierungsvorganges geschieht, wird nicht unbedingt Quellcode erzeugt, sondern direkt Maschinencode oder entsprechende Zwischenrepräsentationen im Compiler.

```
  typedef Then RET;
};
```

Im Rahmen der C++-Template-Definition definieren wir nun einen neuen Typ mittels *typedef*. Nach einer allgemeinen Konvention heißt dieser Typ *RET*. *RET* dient als „Rückgabewert" der Template-Evaluierung. Im Default-Fall definieren wir den Rückgabetyp, den „Wert" der Template-Instanz, als den Typ, der oben als *true*-Fall definiert wurde (der Template-Parameter mit dem Namen *Then*).

Nun wird das Template partiell spezialisiert, für den Fall, wo der boolesche Ausdruck *false* ist. Das Template hat nun nur noch zwei Parameter, da der boolesche Ausdruck nun auf *false* festgelegt ist.

```
//specialization for condition==false
template<class Then, class Else>
struct IF<false, Then, Else> {
  typedef Else RET;
};
```

Der Rückgabewert *RET* ist nun der Parameter, der für den *false*-Fall übergeben wurde (der Parameter *Else*). Wenn nun der Compiler über eine Instanz dieses Template findet, wie im obigen short/int Beispiel, so verwendet er das am besten passende, speziellste Template – hier also den *false*-Fall. *RET* wird damit auf den Typ *Else* definiert und ist damit *short*.

10.3. Generierungstechniken

Im Folgenden wollen wir verbreitete Codegenerierungstechniken vorstellen, inklusive einiger Codebeispiele. Dabei haben wir die Generierungstechniken in verschiedene Kategorien gegliedert.

- Templates + Filtering
- Template + Metamodell
- Frame-Prozessoren
- API-basierte Generatoren
- Inline-Generierung
- Code-Attribute
- Code-Weaving

Wir werden all diese Techniken kurz vorstellen und mit Beispielen illustrieren. Bis auf Inline-Generierung (und in gewisser Weise Codeattribute) erfordern alle Verfahren explizite Integration.

Unabhängig von den teils eklatanten Unterschieden zwischen den verschiedenen Generierungstechniken haben sie doch alle Folgendes gemeinsam:

Es gibt immer (mindestens implizit) ein Metamodell bzw. eine abstrakte Syntax.

- Es gibt immer Transformationen, die auf dem Metamodell aufbauen.
- Es gibt immer irgendeine Art von Frontend, welches das Modell (die Spezifikation) liest und den Transformationen zuführt.

10.3.1. Templates + Filtering

Diese Generierungstechnik beschreibt den wohl einfachsten Fall von Codegenerierung. Wie in Abb. 93 gezeigt, wird mit Hilfe von Templates über die relevanten Teile eines textuell repräsentierten Modells iteriert (z.B. mit XSLT über XML).

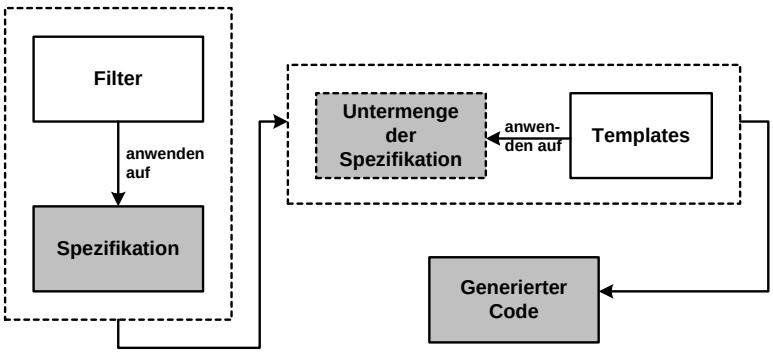


Abb. 93 *Funktionsprinzip Templates + Filtering*

Der zu generierende Code steht in den Templates. Variablen in den Templates können an Werte aus dem Modell gebunden werden. Es folgt ein einfaches Beispiel, bei dem eine Personen-JavaBean aus einer XML-Spezifikation generiert wird (der Einfachheit halber verwenden wir hier nicht XMI zur Repräsentation des Modells, sondern ein eigenes Schema). Hier die Spezifikation:

```
<class name="Person" package="com.mycompany">
```

```
  <attribute name="name" type="String"/>
  <attribute name="age" type="int"/>
</class>
```

Der generierte Code soll z.B. folgendermaßen aussehen:

```
package com.mycompany;
public class Person {
  private String name;
  private int age;
  public String get_name() {return name;}
  public void set_name( String name) {this.name = name;}
  public int get_age() {return age;}
  public void set_age( int age ) {this.age = age;}
}
```

Das XSLT-Stylesheet, um dies zu erreichen, sieht dabei ungefähr so aus:

```
<xsl:template match="/class">
  package <xsl:value-of select="@package"/>;
  public class <xsl:value-of select="@name"/> {
    <xsl:apply-templates select="attribute"/>
  }
</xsl:template>

<xsl:template match="attribute">

  private <xsl:value-of select="@type"/>
    <xsl:value-of select="@name"/>;

  public <xsl:value-of select="@type"/>
    get_<xsl:value-of select="@name"/>() {
       return <xsl:value-of select="@name"/>;
  }

  public void set_<xsl:value-of select="@name"/> (
    <xsl:value-of select="@type"/>
     <xsl:value-of select="@name"/>) {
       this.<xsl:value-of select="@name"/> =
         <xsl:value-of select="@name"/>;
  }
</xsl:template>
```

Das Generieren mittels Templates + Filtering funktioniert recht einfach und portabel, allerdings werden die Stylesheets schnell unübersichtlich und komplex. Für größere Systeme ist dieser Ansatz daher völlig ungeeignet – insbesondere, wenn die Spezifikation auf XMI beruht.

Das XMI-Problem lässt sich etwas entschärfen, indem man in mehreren Schritten arbeitet: Eine erste Transformation transformiert das XMI in ein konkretes, domänenspezifisches XML-Schema. Weitere Transformationsschritte können nun basierend auf diesem Schema den eigentlichen Code erzeugen. Man erhält dadurch eine gewisse Entkopplung der eigentlichen Templates von der konkreten XMI-Syntax, und die eigentliche Codegenerierung ist sehr viel einfacher. Man arbeitet

allerdings immer noch auf der Abstraktionsebene des XML-Metamodells. Der folgende Ansatz kann dieses Problem deutlich lösen.

10.3.2.Templates + Metamodell

Um die Probleme mit der direkten Generierung aus Code aus (XML-)Modellen zu umgehen, kann man einen mehrstufigen Generator implementieren, der zunächst das XML parst, dann ein (vom Benutzer anpassbares) Metamodell instanziiert und dieses dann zusammen mit Templates zur Generierung verwendet. Abb. 94 zeigt das Prinzip.

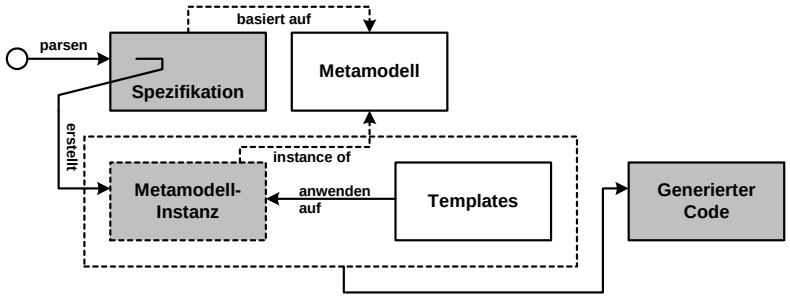


Abb. 94 *Funktionsprinzip Templates + Metamodell*

Der Vorteil dieses Vorgehens ist, dass man dadurch zum einen unabhängiger von der konkreten Syntax des Modells wird (z.B. UML oder den verschiedenen XMI-Versionen). Zum anderen kann komplexere Logik zur Verifikation des Modells im Metamodell untergebracht werden. Dies kann – im Gegensatz zu den Templates – in einer „richtigen“ Programmiersprache, also z.B. Java, implementiert werden. Diese Art der Codegenerierung hat im Zusammenhang mit modellgetriebener Softwareentwicklung eine besondere Bedeutung, wie bereits in Kapitel <Ref MM> beschrieben.

Einen interessanten Implementierungsaspekt des *openArchitectureWare* Generators, der in diese Kategorie fällt, möchten wir nicht unerwähnt lassen: Aus Sicht des Compiler-Baus sind Metamodell-Implementierung (z.B. in Java) *und* Templates Teil der Transformation. Das Metamodell spielt andererseits auch die Rolle der abstrakten Syntax. Da die abstrakte Syntax (das Metamodell) und die Transformation Parameter des Compilers sind, handelt es sich um ein offenes Compiler-Framework. Der Clou besteht nun darin, dass die Konstrukte der Syntax (d.h. die Metamodell-Elemente) sich hier selbst übersetzen, d.h. es ist ein objektorientierter Compiler (das vermeidet u.a. lästige *switch*es). Die Templates sind konzeptionell gesehen genauso

Übersetzungsmethoden am Metamodell, wie die dort in Java implementierten Hilfsmethoden, was man an den Template-Definitionen *<<DEFINE Method FOR Metaclass>>*) sehen kann. Und genau wie Java unterstützt die Template-Sprache auch Polymorphismus und Overwriting – eben um einen objektorientierten Compiler bauen zu können – nur die Definition von Klassen ist auf den Java-Teil verlagert. Deswegen nennen wir die Template Sprache auch gelegentlich objektorientiert, obwohl man eben in ihr keine Klassen definieren kann.

10.3.3. Frame-Prozessoren

Frames – als zentrales Element von Frame-Prozessoren – sind letztendlich Spezifikationen von zu generierendem Code. Wie Klassen in objektorientierten Sprachen können Frames (mehrmals) instanziiert werden. Bei dieser Instanziierung werden die Variablen (genannt Slots) an konkrete Werte gebunden. Jede Instanz kann –wie auch bei Klassen – eigene Werte für ihre Slots besitzen. In einem weiteren Schritt können die Frame-Instanzen dann generiert werden; es wird also konkreter Quellcode erzeugt.

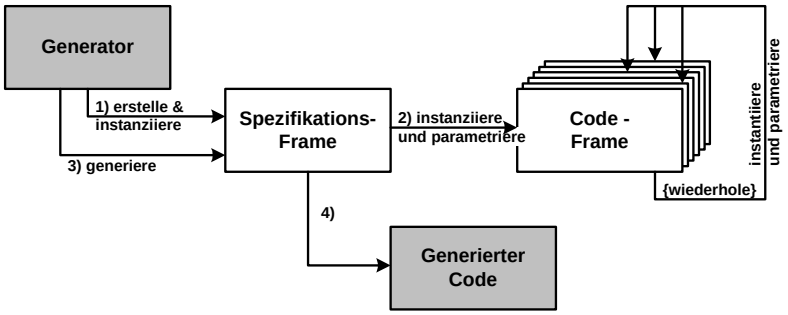


Abb. 95 *Funktionsprinzip Frame-Prozessoren*

Frames können, wie oben erwähnt, Slots besitzen. Die Werte, die Slots zugewiesen werden, können reichen von Strings bis zu weiteren Frame-Instanzen. Dies resultiert zur Laufzeit in einer Baumstruktur von Frame-Instanzen, die letztendlich die Struktur des zu generierenden Programmes repräsentieren. Abb. 96 zeigt ein Beispiel.

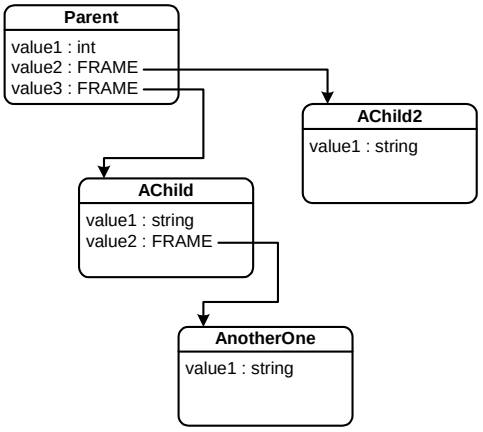


Abb. 96 *Beispielhafte Frame-Hierarchie*

Das folgende Beispiel verwendet den ANGIE-Prozessor [DSTG]. Wir wollen hier zunächst eine einfache Member-Deklaration generieren, in der folgenden Art:

```
short int aShortNumber = 100;
```

Das obige Stückchen Code enthält schon eine ganze Reihe variabler Aspekte: den Namen der Variable, ihren Typ sowie einen optionalen Initialisierungsparameter. Der folgende Frame generiert dieses Stückchen Code:

```
.Frame GenNumberElement(Name, MaxValue)
  .Dim vIntQual = (MaxValue > 32767) ? "long“ : "short"
  .Dim sNumbersInitVal
  <!vIntQual!> int <!Name!> <? = <!sNumbersInitVal!>?>;
.End Frame
```

Die erste Codezeile deklariert den Frame – prinzipiell ein Konstruktor mit zwei Parametern, dem Namen des *NumberElements* und dem maximalen Wert. Basierend auf diesem Maximalwert entscheidet die zweite Zeile, ob wir einen *short int*, *long int* oder einfach einen gewöhnlichen *int* benötigen. Zeile vier definiert den so genannten Host-Code, der dann im Rahmen der Codegenerierung erstellt wird: Die <! ... !>-Syntax greift dabei auf den Wert eines Slots der Frame-Instanz zu, der Code zwischen den <? ... ?> wird dabei nur generiert, wenn der Wert des darin enthaltenen Slots nicht undefiniert ist. Das folgende Stück Code instanziiert den Frame:

```
.myNumbElm =
CreateFrame("GenNumberElement","aShortNumber",100)
```

Zu beachten ist dabei, dass diese Instanziierung noch keinen Code generiert, es wird lediglich eine Frame-Instanz angelegt und der Variable *.myNumbElm* zugewiesen. Die Instanz wird im Generator-internen Instanz-Repository gehalten. Wenn man schließlich

```
.Export(myNumbElm)
```

ausführt, wird die Instanz „ausgeführt" und der Code generiert. Statt die Instanz direkt zu exportieren (und damit den Code zu generieren), kann man sie auch als Wert einem Slot einer anderen Frame-Instanz zuweisen, um damit komplexere Strukturen aufzubauen. Als Beispiel kann der nächste Frame dienen, der eine simple Java-Klasse generiert.

```
.Frame ClassGenerator(fvClassName)
  .Dim fvMembers = CreateCollection()
  public class <!fvClassName!> {
    <!fvMembers!>
  }
.End Frame
```

Dieser Frame akzeptiert den Namen der zu generierenden Klasse als Parameter. Des Weiteren wird ein mehrwertiger Slot (eine *Collection*) angelegt. Ein externes Skript (oder ein anderer Frame) kann nun Werte (z.B. andere Frame-Instanzen) setzen. Beispielsweise können die *NumberElements* von oben gesetzt werden.

```
.Static myGeneratedClass As ClassGenerator
.Function Main(className)
  .myGeneratedClass =
        CreateFrame(„ClassGenerator",className)
     .Add(myGeneratedClass.fvMembers,
        CreateFrame(„GenNumberElement",“i“, 1000))
     .Add(myGeneratedClass.fvMembers,
        CreateFrame(„GenNumberElement",“j“, 1000000))
.End Function
```

Wenn *myGeneratedClass* exportiert wird, wird eine einfache Java-Klasse generiert, die die beiden Member *i* und *j* enthält.

10.3.4. API-basierte Generatoren

Die wahrscheinlich bekannteste Art von Code-Generatoren sind die API-basierten. Diese stellen einfach eine API zur Verfügung, mit der Elemente der Zielplattform oder Sprache erzeugt werden können. Konzeptionell arbeiten diese Generatoren auf der abstrakten Syntax (dem Metamodell) der Zielsprache und sind deshalb immer an eine Zielsprache (genauer: deren abstrakte Syntax) gebunden.

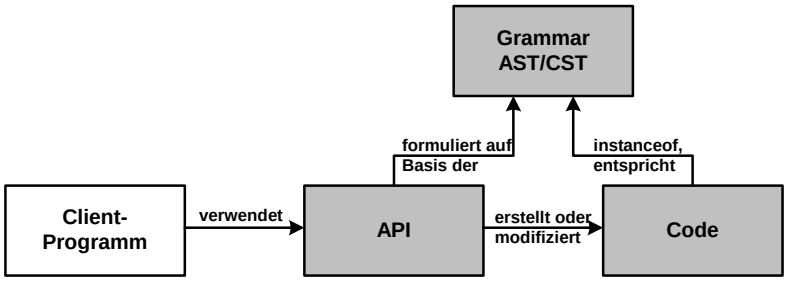


Abb. 97 *Funktionsprinzip API-basierter Generatoren*

Im Folgenden zur Abwechslung ein Beispiel aus der Welt von .NET; der folgende Code soll erzeugt werden:

```
public class Vehicle : object {
}
```

Das folgende Stückchen C#-Code erzeugt diesen:

```
CodeNamespace n = ...
CodeTypeDeclaration c = new CodeTypeDeclaration
(„Vehicle");
c.IsClass = true;
c.BaseTypes.Add (typeof (System.Object) );
c.TypeAttributes  = TypeAttributes.Public;
n.Types.Add(c);
```

Obiger Code baut eine interne Repräsentation des Codes auf, typischerweise als abstrakten Syntaxbaum (AST). Ein Aufruf einer entsprechenden Hilfsfunktion gibt dann den wirklichen Quellcode aus.

Diese Art von Generatoren ist recht intuitiv und einfach zu verwenden, und es ist dabei auch recht einfach zu erzwingen, dass nur syntaktisch korrekter Code erzeugt werden kann: der Compiler des Generator-Codes in Kombination mit der API kann dies sicherstellen. Allerdings existiert hier das Problem, dass man große Mengen immer gleichen Codes „programmieren“ muss, statt einfach Templates zu definieren, in denen ein paar Textstellen ersetzt werden.

Effizient wird die Verwendung solcher Generatoren dann, wenn man sich domänenspezifische Generator-Klassen baut, und zwar durch Anwendung der üblichen OO-Konzepte auf Ebene des Generators. Im Folgenden wird mittels des Werkzeugs Jenerator eine Java-Klasse definiert sowie eine (leere) Main-Methode:

```
public class HelloJenerator {
  public static void main( String[] args ) {
    CClass createdClass =
           new CClass( “demo", "HelloWorld" );
    CMethod mainMethod =
            new CMethod( CVisibility.PUBLIC,
```

```
                          CType.VOID, "main" );
    mainMethod.addParameter(
      new CParameter( CType.user( "String[]" ), "args" )
    );
    mainMethod.setOwnership( Cownership.STATIC );
    createdClass.addMethod( mainMethod );
  }
}
```

Dieses Programm ist sehr lang. Ein Refactoring würde darin bestehen, dass man eine Generator-Klasse *MainMethod* erstellt; Javas *main*-Methoden haben per Definition immer die gleiche Signatur:

```
public class MainMethod extends CMethod {
  public MainMethod() {
    super(CVisibility.PUBLIC, CType.VOID, "main" );
    setOwnership( Cownership.STATIC );
    addParameter(
      new CParameter( CType.user( "String[]" ),
          "args" ) );
  }
}
```

Das *HelloJenerator* von oben vereinfacht sich damit deutlich:

```
public class HelloJenerator {
  public static void main( String[] args ) {
    CClass createdClass =
       new CClass( “demo", "HelloWorld" );
    createdClass.addMethod( new MainMethod() );
  }
}
```

Man kann sich vorstellen, dass mittels solcher Generator-Klassen effiziente, domänenspezifische Generatoren erzeugt werden können. Flexibilität erreicht man durch entsprechende Parametrisierung der Generator-Klassen.

Wie oben bereits erwähnt, sind derartige Generatoren an die abstrakte Syntax der Zielsprache gebunden, nicht an deren konkrete Syntax. Wenn man also verschiedene Sprachen mit derselben abstrakten Syntax hätte, könnte man durch Austausch des Codegenerator-Backends verschiedene Zielsprachen generieren. Dies ist z.B. im Rahmen des .NET-Frameworks möglich. Hier definiert man mittels CodeDOM einen abstrakten Syntaxbaum basierend auf der abstrakten Syntax, die für .NET-Sprachen im Rahmen der CLS vorgegeben ist. Man kann dann durch Auswahl eines passenden *ICodeGenerators* die konkrete Syntax für eine beliebige .NET-Sprache (C#, VB, C++) erzeugen.

Bytecode-Modifier, wie sie vor allem in Java gerne verwendet werden, sind üblicherweise auch API-basierte Generatoren, wobei sie meist auf der Abstraktionsebene des Bytecodes operieren. Auch mittels .NET’s CodeDOM kann man direkt .NET IL-Code erstellen.

10.3.5.Inline-Generierung

Inline-Generierung beschreibt den Fall, bei dem im regulären Quellcode Konstrukte enthalten sind, die während der Kompilierung weiteren Quell- oder Byte/Maschinencode erzeugen. Beispiele sind C++-Präprozessor-Anweisungen oder C++-Templates.

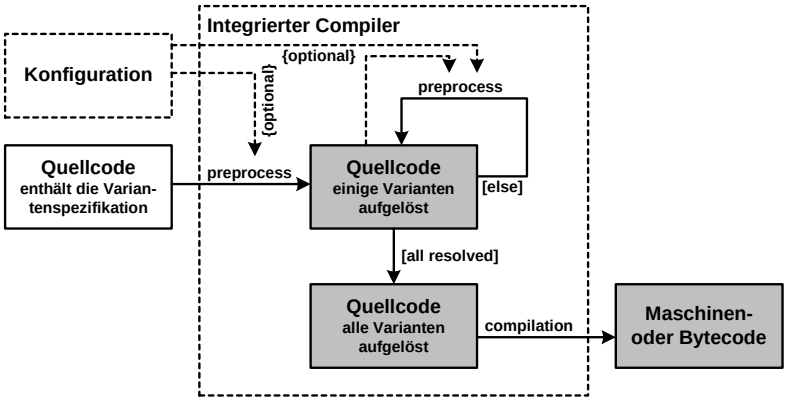


Abb. 98 *Funktionsprinzip Inline-Generierung*

Ein triviales Beispiel mittels des C++-Präprozessors könnte folgendermaßen aussehen:

```
#if defined (ACE_HAS_TLI)
static ssize_t t_snd_n (ACE_HANDLE handle,
  const void *buf, size_t len, int flags,
  const ACE_Time_Value *timeout = 0,
  size_t *bytes_transferred = 0);
#endif /* ACE_HAS_TLI */
```

Hier wird der Code zwischen dem *#if* und dem *#endif* nur kompiliert, wenn die Konfiguration, also das Flag *ACE_HAS_TLI*, definiert (also praktisch *true*) ist. Auch komplexere Ausdrücke mit Parameterübergabe sind möglich:

```
#define MAX(x,y) (x<y ? y : x)
#define square(x) x*x
```

Wird im Code an einer Stelle *MAX(v1, v2)* geschrieben, so wird dies textuell entsprechend obiger Vorschrift ersetzt; im Programm steht also nach dem Präprozessor-Lauf *(v1<v2 ? v1 : v2)*. Das Ganze basiert auf reiner Textersetzung, und es werden keine Typprüfungen oder Präferenzregeln befolgt. Insofern ist dieses Vorgehen nur in Ausnahmefällen sinnvoll.

Template-Metaprogrammierung erlaubt da schon ein etwas strukturierteres Vorgehen. Templates (bzw. ihre Verarbeitung durch den

Compiler) stellen nämlich eine Turing-vollständige, funktionale Programmiersprache, die auf Typen und Literalen arbeitet. Man kann also komplette Programme schreiben, die zur Compile-Zeit ausgeführt werden. Ein Beispiel ist das folgende, welches zur Compile-Zeit die Fakultät eines *Integers* berechnet.

```
#include <iostream>
using namespace std;

#include "../meta/meta.h"
using namespace meta;

struct Stop
{ enum { RET = 1 };
};

template<int n>
struct Factorial
{ typedef IF<n==0, Stop, Factorial<n-1> >::RET
  PreviousFactorial;
  enum { RET = (n==0) ? PreviousFactorial::RET :
  PreviousFactorial::RET * n };
};

void main()
{ cout << Factorial<3>::RET << endl;
}
```

Wie das Ganze funktioniert, sei dem Leser zur Übung überlassen (alternativ kann man es auch in [EC00] nachschlagen). Aufgrund der gewöhnungsbedürftigen Syntax und den teils sehr seltsamen Fehlermeldungen ist dieses Vorgehen aber nur in Ausnahmefällen sinnvoll und für größere Generierungsvorhaben ungeeignet. Dies liegt unter anderem auch daran, dass die Compiler (die die Metaprogramme ja ausführen) nicht für diese Zwecke erstellt und optimiert wurden.

10.3.6. Code-Attribute

Ein weiterer Mechanismus sei erwähnt, der im Java-Umfeld recht verbreitet ist: Code-Attribute. Angefangen hat dies im Java-Umfeld mit *JavaDoc*, wo spezielle Kommentare verwendet wurden, um die automatische Generierung von HTML-Dokumentation zu erlauben. Aufgrund der erweiterbaren Architektur von *JavaDoc* kann man nun eigene Tags und Code-Generatoren einhängen. Das wohl bekannteste Beispiel dafür ist *XDoclet* [XDOC]. Hiermit werden unter anderem EJB *Remote/LocalInterfaces* sowie Deployment-Deskriptoren generiert. Der Entwickler schreibt dabei nur noch die Implementierungsklasse von Hand, und fügt dieser Klasse eben entsprechende Kommentare hinzu, die

vom Code-Generator ausgelesen werden. Der Generator hat außerdem Zugriff auf den Syntaxbaum des Quellcodes, an den die Kommentare angefügt sind. Damit kann der Generator Informationen sowohl aus den Kommentaren als auch aus dem Code selbst entnehmen. Das Folgende ist ein Beispiel einer Java-Klasse, das mit XDoclet-Kommentaren ergänzt wurde:

```
/**
 * @ejb:bean type="Stateless"
 *           name="vvm/VVMQuery"
 *           local-jndi-name="/ejb/vvm/VVMQueryLocal"
 *           jndi-name="/ejb/vvm/VVMQueryRemote"
 *           view-type="both"
 */
public abstract class VVMQueryBean
    /**
     * @ejb:interface-method view-type="both"
     */
    public List getPartsForVehicle( VIN theVehicle ) {
        return super.getPartsForVehicle( theVehicle );
    }
}
```

Das Charmante an dieser Methode der Codegenerierung ist, dass eben viele der Informationen, die der Generator benötigt, bereits im Code vorhanden sind. Der Entwickler muss nur noch wenige, spezielle Kommentare angeben. Dem Generator stehen sowohl der AST des Codes als auch die eingefügten Kommentare zur Verfügung.

Eine oft erwähnte Kritik in diesem Zusammenhang ist, dass man derartige Werkzeuge ja nur braucht, weil J2EE (und da vor allem EJB) eine derart große Redundanz im Code erfordert, dass dies manuell kaum mehr zu handhaben ist. Dies ist sicherlich richtig – man muss die Generierung mittels Attributen aber nicht auf die Generierung von EJB-Infrastruktur beschränken. Man kann mittels XDoclet beispielsweise auch sehr schön Persistenz-Mappings für *Hibernate* [HIBE] oder ähnliche Frameworks erzeugen.

.NET bietet den Attributmechanismus als integrales Konzept der .NET-Platform. Verschiedene Elemente des Quelltextes – Methoden, Attribute, Klassen – können mit Attributen versehen werden, wie das folgende Beispiel zeigt:

```
[QoSServicePriority(Prio.HIGH)]
class SomeService : ServiceBase {
    [QoSTimeLimit(100, Quantity.MS)]
    public void processRequest( Request r ) {
      ….
    }
}
```

Hier wird – zunächst rein deklarativ – angegeben, dass Instanzen dieser Klasse eine Service-Priorität *HIGH* haben und die Ausführung der Operation *processRequest()* maximal 100ms dauern darf. Die Idee dabei ist, dass diese Services in einem Framework ausgeführt werden, welches diese Parameter (eben z.B. die Ausführungszeit) misst. Es kann dann – bei Überschreiten der per Attribut angegebenen Limits – z.B. einen Log-Eintrag machen oder keine weiteren Requests mehr annehmen und eine Exception zum Client werfen.

Der Zugriff des Frameworks auf das Attribut (und damit das definierte Zeitlimit) findet per Reflection statt. Technisch wird das so realisiert, dass Attribute serialisierbare .NET-Klassen sind, die der Compiler während des Übersetzungsvorganges instanziiert und in die *Assembly* serialisiert. Per Reflection können diese Objekte dann ausgelesen werden.

Codegeneratoren können diese Informationen aus dem kompilierten .NET Assembly auslesen und wie bei XDoclet die erhaltenen Informationen als Basis für die Generierung verwenden. Man beachte, dass derartige Features ab Version 1.5 auch in Java zur Verfügung stehen.

10.3.7. Code-Weaving

Code-Weaving beschreibt das Zusammenfügen verschiedener, jeweils vollständiger und daher unabhängiger Codebestandteile. Dazu muss definiert werden, wie diese verschiedenen Bestandteile zusammengewoben werden können – man nennt diese Stellen auch *Join Points* oder *Hooks*. Ein Beispiel für diese Art von Generator ist *AspectJ* [ASPJ]. Hier werden regulärer OO-Programmcode und Aspekte auf Quellcode- oder Bytecode-Ebene verwoben. Aspekte beschreiben dabei so genannte querschnittliche Belange (engl. Cross-cutting concerns), also solche Funktionalität, die mit regulären OO-Mitteln nicht vernünftig lokalisiert (in einer Klasse oder einer Funktion) beschrieben werden können.

Im Folgenden als Beispiel ein Aspekt, der bei allen Klassen, die Methoden auf der Klasse *Account* aufrufen, Log-Ausgaben einfügt. Vor jedem Methodenaufruf wird dann auf die Konsole ausgegeben, *aus*

welcher Methode heraus die Methode auf einem *Account* aufgerufen wird[46]:

```
aspect Logger {
 public void log( String className, String methodName ) {
   System.out.println( className+“.“+methodName );
 }
 pointcut accountCall():  call(* Account.*(*));
 before() calling: accountCall() {
   log( thisClass.getName(), thisMethod.getName() );
 }
}
```

Nachdem nun dieser Aspekt auf ein System angewandt wurde (also durch den Weaver mit dem regulären Code zusammengewoben wurde), wird ungefähr folgender Code erzeugt:

```
public class SomeClass {
  private Account someAccount = ...;
  public someMethod( Account account2, int d ) {
    // aspect Logger
    System.out.println(„SomeClass.someMethod“);
    someAccount.add( d );
    // aspect Logger
    System.out.println(„SomeClass.someMethod“);
    account2.subtract( d );
  }
  public void anotherMethod() {
    //aspect Logger
    System.out.println(„SomeClass.anotherMethod“);
    int bal = someAccount.getBalance();
  }
}
```

Ein weiteres Beispiel für diese so genannte invasive Komposition von Quellcode ist die *Compost-Bibliothek* [COMP]. Diese stellt letztendlich eine API zur Verfügung, um auf Basis des AST die Struktur von Programmen zu verändern. Compost ist in Java implementiert und operiert auch auf Java-Quellcode. Eine weiterführende Bibliothek (das *Boxology Framework*) erlaubt die systematische Modifikation von Quellcode an so genannten Hooks. Dabei wird zwischen expliziten (vom Ersteller des zu modifizierenden Quellcodes deklarierten) und impliziten Hooks unterschieden. Implizite Hooks sind bestimmte, wohldefinierte Stellen im AST eine Programmes, beispielsweise der „implements“-Hook. Durch Erweiterung dieses Hooks können weitere Interfaces

[46] Die Syntax von AspectJ entwickelt sich regelmäßig weiter, es kann also gut sein, dass die hier gezeigte Syntax mit der aktuellen Version nicht mehr funktioniert.

implementiert werden. Dieses Framework kann damit als Basis für verschiedenste Quellcodemodifikationen dienen.

10.3.8. Kombination verschiedener Techniken

Kombinationen verschiedener Techniken sind natürlich auch möglich. Das Open-Source-Werkzeug *AndroMDA* [ANDR] erzeugt beispielsweise mittels Templates Quellcode, der wiederum Codeattribute enthält. Die Erstellung des Quellcodes mittels Templates geschieht mit *Velocity* [VELO]. Die weitere Verarbeitung dann mit XDoclet. Das Folgende ist ein Beispiel für ein Velocity-Template mit XDoclet-Kommentaren.

```
// --------------- attributes --------------------
#foreach ( $att in $class.attributes )
#set ($atttypename =
transform.findFullyQualifiedName($att.type))
    private $atttypename ${att.name};

   /**
#generateDocumentation ($att "    ")
    *
#set ($attcolname = $str.toDatabaseAttriName(${att.name},
"_"))
#set ($attsqltype = $transform.findAttributeSQLType($att))
#if ($transform.getStereotype($att.id) == "PrimaryKey")
    * @hibernate.id
    *     generator-class="uuid.string"
#else
    * @hibernate.property
#end
    *     column="$attcolname"
    *
    * @hibernate.column
    *     sql-type="$attsqltype"
```

Eine andere beliebte Kombination sind API-basierte Generatoren, die optional Templates einlesen können, um die „Bedienung“ der API zu vereinfachen.

10.3.9. Gemeinsamkeiten und Unterschiede der verschiedenen Ansätze

Zunächst möchten wir hier eine Klassifizierung der verschiedenen Ansätze bezüglich der in Abschnitt 10.2 erwähnten Kriterien vornehmen:

	Zeitpunkt bzgl. Kompilierung	**Programm/ Metaprogramm**	**Generierter/ manueller Code**

Templates + Filtering	vor	getrennt	getrennt
Template + Metamodell	vor	getrennt	getrennt
Frame-Prozessoren	vor	getrennt	getrennt
API-basierte Generatoren	vor/während/nach	getrennt	getrennt
Inline-Generierung	vor/während	vermischt	integriert
Code-Attribute	vor/während	getrennt/vermischt	getrennt
Code-Weaving	vor/während/nach	getrennt	integriert

Diese Tabelle bedarf einiger Erläuterung, insbesondere an den Stellen, wo mehrere Möglichkeiten angegeben sind:

- API-basierte Generatoren können sowohl vor der Kompilierung (wie in den Beispielen erläutert) als auch während der Kompilierung (Compile-Time Metaobjektprotokolle) sowie zur Laufzeit (Runtime-Metaobjektprotokolle) stattfinden.
- Inline-Generierung kann sowohl vor der eigentlichen Kompilierung (mittels eines mehr oder weniger gut integrierten) Präprozessors stattfinden als auch während der Kompilierung (Lisp's Quoting-Mechanismen, C++-Templates)
- Code-Attribute werden entweder während der Kompilierung ausgewertet (wie z.B. in .NET) oder davor mittels eines Präprozessors (XDoclet). Je nach Generator kann das Generat direkt in den Quellcode eingebettet werden oder in separate Artefakte münden.
- Auch Code-Weaving kann in einem separaten Lauf vor der Kompilierung, während der Kompilierung (bei voll aspektorientierten Sprachen) als auch zur Laufzeit (dynamisches Aspekt-Weaving) stattfinden. In Java ist auch das Verweben zur Ladezeit (durch spezielle Classloader) verbreitet.

Wir möchten noch auf einige weitere Charakteristika, Unterschiede und Gemeinsamkeiten hinweisen. Frame-Prozessoren und API-basierte Generatoren bauen beide prinzipiell einen AST des zu erstellenden Systems auf. In beiden Verfahren kann man die „Nähe zur Problemdomäne" (und damit die Abstraktionsebene und die Effizienz) dadurch erhöhen, dass man domänenspezifische Generatorkonstrukte

einfügt (z.B. einen Frame, der ein JavaBeans-Property erzeugt). Jedoch beginnt man eigentlich immer mit dem AST der Zielsprache/Plattform.

Beim Ansatz „Templates + Metamodell“, baut dagegen der Generator zur Laufzeit einen AST des Modells auf, also (je nach Metamodell) eine Repräsentation des Problemraumes. Damit startet man bereits von einem höheren Abstraktionsniveau. Die Templates leisten dann die Übersetzung hin zur Zielplattform. Diese Art von Generatoren eignet sich also vor allem für Anwendungsfälle, wo das Metamodell des Problemraums bereits recht umfangreich ist.

Eine Frage, die sich oft stellt, ist die, ob API- oder Template-basierte Generatoren besser geeignet sind. Unseres Erachtens sind Template-basierte Systeme vor allem dann besser, wenn große Mengen gleichartigen Codes erzeugt werden; API-basierte Generatoren sind effizienter, wenn sehr feingranularer Code erzeugt werden soll (beispielsweise State Machines oder Algorithmen). Im Rahmen (architekturzentrierter) modellgetriebener Entwicklung eignen sich Template-basierte Generatoren meist besser.

Code-Attribute können als eine Form der Inline-Generierung angesehen werden. Das Generat kann direkt an die Stelle generiert werden, wo die Spezifikation im Basisprogramm-Quellcode zu finden ist. In den allermeisten Fällen geschieht dies aber nicht: Der aus dem Basisprogramm und den Attributen generierte Code ist typischerweise extern und meistens vollständig und muss also nicht mehr mit manuellem Code integriert werden, da er üblicherweise technische Aspekte (Persistenz, EJB-Gluecode) abdeckt. Dieses Verfahren bietet sich dann an, wenn man eben *nicht* modellgetrieben arbeitet, sondern bereits vorhandenen (Interface-)Code zusammen mit zusätzlichen Informationen in den Attributen als Spezifikation verwenden kann.

Was Inline-Generierung und Code-Weaving im Kern unterscheidet, ist, dass man mit Letzterem nicht-invasiv querschnittliche Belange lokalisieren kann. Der bestehende Code wird also quasi „von außen“ modifiziert. Auch diese beiden Verfahren eignen sich vor allem dann gut, wenn man ausschließlich mit Code, und nicht mit Modellen, arbeitet.

10.3.10. Weitere Systeme

Es gibt im Bereich der Codegenerierung noch eine ganze Reihe weiterer Systeme, deren Bedeutung für modellgetriebene Softwareentwicklung jedoch zu vernachlässigen ist. Ein Abriss solcher Systeme ist unter anderem in [Voe03] zu finden. Hier seien nur einige Beispiele kurz erwähnt:

- *Kompositionssysteme* wie z.B. Kompost erlauben die beliebige Manipulation von ASTs von Quellcode. Damit sind beliebige invasive Systeme (und Code-Weaver) erstellbar.
- *Metaobjektprotokolle:* Metaobjektprotokolle erlauben den Zugriff auf die Strukturen des Compilers (Compile-Time MOPs wie *OpenC++* [OC++]) oder des Laufzeitsystems (Runtime-MOPs, z.B. *CLOS* [KRB91]), um das Verhalten des Programmes zu verändern oder das kompilierte bzw. laufende Programm zu verändern.
- *Bytecode-Modifier:* Vor allem im Rahmen von Java gibt es eine Reihe von Werkzeugen, die es erlauben, generierten Bytecode zu modifizieren. Beispielsweise können transparente Zugriffe auf eine OO-Datenbank an die entsprechenden Stellen im Bytecode hineingeneriert werden. Beispiele dafür sind *BCEL* [BCEL] oder *Javassist* [JASS].

11. MDSD-Werkzeuge: Rollen, Architektur und Auswahlkriterien

Im letzten Kapitel dieses Teils wollen wir konkreter auf Konstruktionsmerkmale generischer MDSD- bzw. MDA-Werkzeuge eingehen[47]. Diese können gleichzeitig als Auswahlkriterien verwendet werden. Aus nahe liegenden Gründen können und wollen wir hier jedoch keine Hinweise auf kommerzielle Werkzeuge geben. Die OMG bietet eine Aufstellung von MDA-Werkzeugen [OMGT]. Allerdings erfüllen nicht alle dort aufgeführten Werkzeuge die hier diskutierten Anforderungen, zumal die Einträge durch Registrierung seitens der Hersteller und nicht durch eine Zertifizierung entstehen.

Das Kapitel ist in zwei Abschnitte unterteilt. Im ersten Teil beschreiben wir die Kategorien von Werkzeugen, die bei der MDSD-Entwicklung eine Rolle spielen.

Im zweiten Teil beschreiben wir einige Grundlagen von MDSD-Werkzeugen. Die einzelnen Punkte sollten bei der Implementierung und Auswahl von MDSD-Werkzeugen bzw. Werkzeugketten berücksichtigt werden.

---

[47] „Echte“ MDA-Werkzeuge sind auch MDSD-Werkzeuge, aber nicht jedes MDSD-Werkzeug ist ein MDA-Werkzeug. Dies ist in der Fokussierung der MDA auf UML bzw. MOF begründet. Die Unterscheidung ist jedoch für unsere Betrachtungen nicht essentiell.

11.1. Die Rolle von Werkzeugen im Entwicklungsprozess

Modellgetriebene Entwicklung macht ohne Werkzeugunterstützung wenig Sinn. Dieser Abschnitt soll einen kurzen Überblick über typische Werkzeugkategorien geben, die im Rahmen eines MDSD-Projektes bzw. einer Domänenarchitektur Verwendung finden können.

11.1.1. Modellierung

Das zentrale Werkzeug ist das Modellierungswerkzeug. Je nach DSL können in dieser Kategorie verschiedenste Ausprägungen eingesetzt werden - man sollte allerdings schon versuchen, für eine bestimmte DSL auch einen dazu passenden Editor zur Verfügung zu stellen, der die DSL „kennt“ und bei der Modellierung effektiv unterstützt. Man kann beispielsweise Texteditoren verwenden, die die DSL durch Syntax-Highlighting und Code-Completion unterstützen; auch formularbasierte Editoren sind oft sinnvoll. Für grafische DSLs kommen grafische Editoren zum Einsatz – der Aufwand für deren Erstellung kann aber erheblich sein, sofern nicht die UML als Basis verwendet wird.

Werden die Modelle mittels UML-Profilen erstellt, ist ein geeignetes UML-Werkzeug vonnöten. Prinzipiell kann man mit jedem UML-Werkzeug arbeiten – es gibt jedoch einige Hinweise, die man beachten sollte. Die folgende Liste ist nach Wichtigkeit absteigend sortiert.

- XMI Export: Zunächst sollte das Werkzeug in der Lage sein, das Modell im XMI-Format zu exportieren. Die meisten MDA/MDSD-Generatoren „verstehen“ u.a. XMI als Eingabeformat. XMI enthält zwar immer noch viele Freiheitsgrade und damit Interpretationsspielraum, aber ein XMI-Export ist trotzdem eine gute Voraussetzung zur Weiterverarbeitung von Modellen.
- Stereotypen und Tagged Values: Die Annotation von Stereotypen an Modellelementen ist mit praktisch jedem Modellierungswerkzeug möglich und insofern ein triviales Kriterium. Bei Tagged Values wird es schon etwas weniger selbstverständlich. Gute UML-Werkzeuge verwalten Stereotypen als Modellelemente und nicht nur als „Strings an einer Klasse“. Man sollte definieren können, an welche Modellierungselemente (also Elemente des UML-Metamodells) ein bestimmter Stereotyp angebracht werden

kann. Außerdem sollte man festlegen können, dass ein bestimmter Stereotyp auch bestimmte Tagged Values erfordert oder zuläßt. Das Werkzeug sollte den Entwickler dann zu einer konsistenten Eingabe bei der konkreten Modellierung zwingen.

- Metamodellierung/Profile: Idealerweise unterstützt das Werkzeug echte Metamodellierung bzw. die Erstellung von UML-Profilen. Dies umfasst die Definition eigener Metamodelltypen sowie Constraints auf dem Metamodell. Idealerweise lässt sich dann auch die grafische Darstellung der neuen Modellelemente anpassen, und auch die Oberfläche des Werkzeuges (es sollte also beispielsweise eine Schaltfläche zum Einfügen eines selbstdefinierten Modellelementtyps erscheinen). Derartige Werkzeuge gehen schon in Richtung von Metamodellierungs-IDEs und sind derzeit leider noch rar.
- OCL: Ein nützliches Feature besteht darin, dass das Werkzeug OCL-Ausdrücke auf der MOF-Ebene M1 (vgl. <Ref MM>) nicht nur als Kommentare zulässt, sondern die Constraints zumindest syntaktisch bzgl. des Modells prüft. Ein Constraint wird also nur zugelassen, wenn es einen gültigen Ausdruck bzgl. des aktuellen Modells darstellt.

11.1.2. Modellvalidierung & Codegenerierung

Derzeit sind die allermeisten UML-Werkzeuge nicht in der Lage, ein Modell auf Korrektheit bzgl. eines domänenspezifischen Metamodells zu überprüfen. Codegenerierung kann aber nur funktionieren, wenn das Modell korrekt bzgl. des Metamodells ist. Auch in dem Fall, dass ein Modell nicht mit UML (+ Profilen) modelliert wird, muss eine Prüfung des Modells stattfinden, bevor Code generiert wird.

In den meisten Fällen wird diese Überprüfung des Modells durch ein separates Werkzeug durchgeführt – den *Generator*. Wichtig ist dabei, dass die Validierung in diesem Tool separat von der Codegenerierung geschieht. Die Transformationen sollten sich nicht um die Validierung des Modells kümmern, sondern von einem bzgl. des Metamodells korrekten Modell ausgehen. Ansonsten werden die Transformationen unnötig kompliziert. Außerdem ist die Validierung eines Modells völlig unabhängig von der Frage, was daraus generiert wird (also von den Transformationen). Sie ist ausschließlich eine Frage der Domäne, für die das Modell erstellt wurde. Wenn man verschiedene Transformationen auf

demselben Modell ausführt, müsste man die Korrektheits-Constraints ja in jeder Transformation unterbringen – sicherlich nicht besonders erstrebenswert.

Im Folgenden einige Punkte, die für die Validierungs- und Transformationsphase des Generators essenziell sind, wieder in Reihenfolge ihrer Wichtigkeit.

- Abstraktion von der konkreten Syntax: Die Validierung der Modelle sollte unbedingt unabhängig von der konkreten Syntax geschehen, in der das Modell vorliegt. Ein derartiges Vorgehen erlaubt die Änderung der konkreten Syntax, ohne dass die Validierungsregeln angepasst werden müssen; es muss lediglich ein neuer Parser zur Verfügung stehen, der die geänderte konkrete Syntax lesen kann. Wir erläutern diesen Punkt später noch detaillierter. Idealerweise erfolgt die Abstraktion von der konkreten Syntax der DSL durch eine explizite Repräsentation des Metamodelles.

- Explizite Repräsentation des Metamodells: Essenziell – sowohl für die Validierung als auch für die Transformation von Modellen - ist die Eigenschaft, dass auch das Metamodell, welches dem gerade verarbeiteten Modell zugrunde liegt, im Generator eine explizite Repräsentation besitzt. Diese Forderung impliziert, dass der Entwickler das vom Generator verwendete Metamodell anpassen kann. Eine bewährte Möglichkeit zur Lösung dieses Problems besteht z.B. darin, die Metamodellelemente als Klassen einer Programmiersprache zu repräsentieren. Das Modell besteht dann aus Objekten, also Instanzen der betreffenden Metamodellelemente (vgl. <Ref Code::TplMM>). Auch auf diesen Punkt gehen wir später nochmals ein.

- Deklarative Constraints: Idealerweise sollten die Constraints, die die Korrektheit eines Modells angeben, deklarativ definierbar sein. OCL ist ein Beispiel für eine solche deklarative Sprache, die allerdings noch nicht in der Breite von Werkzeugen unterstützt wird. Doch auch jenseits dieses Optimums sollte man versuchen, Constraints möglichst deklarativ zu formulieren. Im Rahmen der zweiten ausführlichen Fallstudie (vgl. <Ref CSII>) in diesem Buch zeigen wir entsprechende Ansätze im Detail.

Anhand des Open-Source-Generators *openArchitectureWare* [OAW] können alle diese Eigenschaften exemplarisch studiert werden.

11.1.3. Build-Werkzeug

In vielen Fällen wird eine große Menge verschiedener Artefakte durch den Generator erstellt. Diese müssen im nächsten Schritt geeignet kompiliert, gepackt und weiterverarbeitet werden. Dafür ist ein geeignetes Build-Werkzeug vonnöten. Prinzipiell eignet sich dafür jede Skriptsprache. In der Java-Welt hat sich Ant durchgesetzt.

In vielen Fällen wird man auch das Build-Skript im Rahmen des Generierungsprozesses erstellen und dann direkt ausführen.

11.1.4. IDE-Baukasten

Ein Werkzeug, welches zwar nicht unbedingt notwendig ist, aber – wenn vorhanden – sehr nützlich sein kann, ist ein „IDE-Baukasten“. Die Idee ist, dass die verschiedenen Arbeitsschritte und die generierten Artefakte in einer IDE für den Anwendungsentwickler bearbeitbar sind. Es sollten angepasste Editoren für Konfigurations-Files, Zugriff auf das Modell sowie auf die generierten Artefakte vorhanden sein. Insbesondere sollte die mit dem Baukasten erstellte domänenspezifische IDE nur die Dinge zeigen, die für den Entwickler relevant sind. Irrelevante Zwischenergebnisse sollten nur bei Bedarf angezeigt werden.

Ein solcher IDE-Baukasten ist zum Beispiel die Eclipse-Plattform. Damit lassen sich mit vertretbarem Aufwand projekt- oder domänenspezifische IDEs herstellen.

11.2. Werkzeugarchitektur und Auswahlkriterien

11.2.1. Implementierung des Metamodells

Ein formal definiertes Metamodell für eine Domäne ist ein guter Anfang, aber es muss auch entsprechend bei der Anwendungsentwicklung in der Domäne zum Einsatz kommen. Solange das Metamodell nur auf Papier oder in einem Modellierungswerkzeug dokumentiert ist, ohne dass weitere Werkzeuge darauf aufbauen, hat es keinen produktiven Nutzen.

Manuelle Überprüfung von Modellen in Bezug auf Konsistenz mit dem zugrunde liegenden Metamodell ist zeitraubend und fehleranfällig. Standard-Modellierungswerkzeuge (aktuelle UML-Werkzeuge) helfen da auch nicht weiter, da sie die Regeln des domänenspezifischen Metamodells nicht „verstehen“ und diese somit nicht zur Prüfung von

Modellen verwenden können. Die einzigen Regeln, die ein klassisches UML-Werkzeug anwendet, sind die Regeln in Bezug auf wohldefinierte UML-Modelle im Allgemeinen. In diesem Punkt ist zu hoffen, dass UML-Werkzeuge in näherer Zukunft Unterstützung für Profile bieten.

Um modellgetriebene Anwendungsentwicklung betreiben zu können, muss der Generator sinnvollen und lauffähigen Code erzeugen, was nur gewährleistet ist, wenn die zur Generierung verwendeten Modelle ausschließlich Konstrukte und deren Kombinationen enthalten, die nach den Regeln des zugrunde liegenden Metamodells zulässig sind.

Implementieren Sie also das Metamodell in einem Werkzeug, welches Modelle lesen und auf Korrektheit gegenüber diesem Metamodell überprüfen kann. Die Korrektheitsprüfungen müssen alle Regeln und Bedingungen abdecken, welche das Metamodell vorschreibt. Nur so sind sinnvolle Modelltransformationen (Codegenerierung) möglich.

Die Prüfung der Generatoreingabedaten ist eine praktische Verwendung des Metamodells. Diese Wertschöpfung ist ein essenzieller Bestandteil des MDSD-Prozesses und der Domänenarchitektur.

Natürlich kann die Implementierung des Metamodells selbst wiederum durch modellgetriebene Techniken (d.h. eine entsprechende Meta-Domänenarchitektur) erfolgen.

11.2.2. Ignorieren der konkreten Syntax

Jedes Modell muss zwangsläufig in einer konkreten Syntax dargestellt werden (z.B. UML oder XMI für MOF-basierte Modelle, XML für textuelle Modelle). Allerdings sind Transformationen, welche auf der Basis der konkreten Syntax der zu transformierenden Modelle definiert sind, ziemlich unhandlich, weil sie auf die konkrete Syntax Rücksicht nehmen müssen, wo doch eigentlich die Transformation von Instanzen des Metamodells im Vordergrund stehen sollte. Dies macht die Transformationen unnötig kompliziert. Außerdem können die Transformationen nicht weiterverwendet werden, wenn die konkrete Syntax der Modelle geändert wird – was im Laufe von Projekten nicht selten vorkommt. Wie kann man erreichen, dass die Transformationen (und die Modellvalidierung) nicht auf der konkreten Syntax der Modelle aufsetzen, damit also nicht von der konkreten Syntax abhängig sind?

Die Definition von Transformationen auf der Basis von konkreter Syntax ist typischerweise sehr fehleranfällig und ineffizient. Nehmen wir zum Beispiel XMI. XMI hat eine sehr komplizierte Syntax, und Transformationen auf der Basis von XMI und eventuell XSLT zu definieren ist in der Praxis höchstens für Trivialfälle tauglich.

In vielen Fällen ist es außerdem nützlich, mehrere konkrete Syntaxformen für dasselbe Metamodell zu verwenden, zum Beispiel wenn verschiedene DSLs zur Beschreibung verschiedener technischer Subdomänen verwendet werden, oder wenn die konkrete Syntax im Projektverlauf geändert wird. Definitionen von Transformationen (sowie Modellvalidierungen) basierend auf konkreter Syntax binden die Transformationen unnötig an eine spezifische konkrete Syntax.

Die Transformationsdefinitionen sollten also auf dem Quell- und ggf. dem Zielmetamodell basieren. Implementieren Sie dazu einen dreistufigen Ansatz im Transformationswerkzeug:

- Zuerst wird das Ausgangsmodell geparst und eine Abbildung des Modells im Hauptspeicher erzeugt (typischerweise in Form einer Objektstruktur, z.B. durch Instanziierung der Metamodellklassen).
- Dann wird das Ausgangsmodell in das Zielmodell transformiert (und dabei wieder als Objektstruktur abgebildet).
- Und zuletzt wird das Zielmodell wieder in die konkrete Syntax der Ziel-DSL umgewandelt.

Dieser Ansatz erlaubt eine wesentlich effizientere und produktivere Art, Transformationen zu spezifizieren. Es macht auch den Transformator wesentlich flexibler, da er jetzt mit beliebiger konkreter Syntax arbeiten kann. Dies trifft insbesondere auf XMI-basierte konkrete Syntax zu, da unterschiedliche UML-Werkzeuge unterschiedliche XMI-Dialekte erzeugen. Sie sollten vermeiden, Ihre Transformationen an ein bestimmtes Werkzeug oder sogar an eine bestimmte Werkzeugversion zu binden.

Dieses Vorgehen ist übrigens seit langer Zeit im Compilerbau üblich. Compiler arbeiten immer in mehreren Phasen; die Implementierungen dieser Phasen sind austauschbar. Somit können Compiler relativ leicht auf verschiedene Zielplattformen angepasst werden oder auch verschiedene Quellsprachen „verstehen“ (z.B. *C* und *Ada*).

Abb. 99 zeigt das Prinzip für Codegeneratoren.

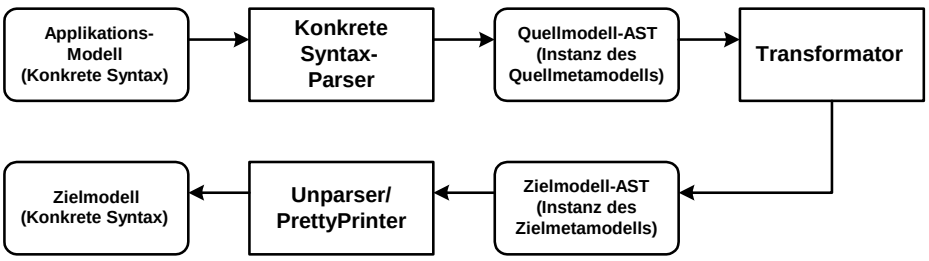


Abb. 99 *Funktionsprinzip eines typischen AST-basierten Generators*

MDSD-Codegeneratoren benutzen oft nicht den vollständigen dreistufigen Ansatz, sondern generieren direkt textuellen Output auf der Basis der Ausgangsmodellinstanz. Einen abstrakten Syntaxbaum des Zielmodells zu erzeugen wäre übermäßig kompliziert. Statt dessen werden Template-Sprachen benutzt, um über das Ausgangsmetamodell zu navigieren und damit das Ausgangsmodell zu lesen. Das Ursprungsmodell sollte aber unbedingt trotzdem als Objekt-Graph repräsentiert werden.

Dieses Vorgehen passt sehr gut dazu, das Metamodell zu implementieren (siehe vorheriger Abschnitt). Idealerweise wird dieselbe Implementierung für beide Zwecke verwendet. Die Templates können dann direkt mit den Metaobjekten arbeiten, und die Eigenschaften der Metaobjekte können benutzt werden, um Informationen zur Template-Evaluierung bereitzustellen, wie in der Abb. 100 gezeigt (vgl. auch <Ref CSI> und <Ref CSII>).

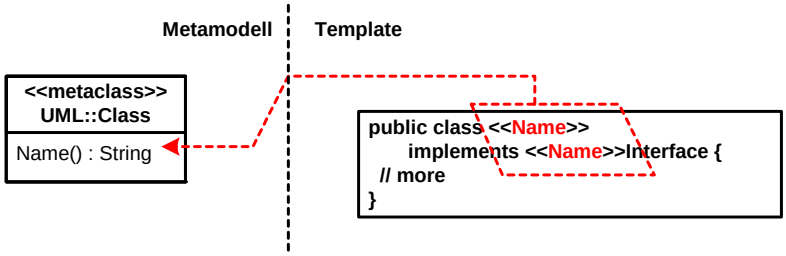


Abb. 100*Zugriff auf den Modell-AST aus den Templates gemäß [OAW]*

11.2.3.Mehrstufige Generierung

Es ist oft sehr kompliziert, alle Produktkonfigurations- und Generierungsschritte in eine einzige Transformation einzubauen. Features können Abhängigkeiten untereinander aufweisen. Verschiedene Systemteile sind oft auf unterschiedliche Weise modelliert. Wie kann dennoch ein effizienter Transformationsprozess aufgebaut werden?

Betrachten wir zum Beispiel die Auswahl der Zielplattform. In Abhängigkeit von der Zielplattform müssen unterschiedliche Transformationen ausgeführt werden. Die Wahl der Zielplattform beeinflusst also, welche Transformationen zum Einsatz kommen. Es ist recht komplex, alle Alternativen in einer Menge von Transformationen so abzubilden, dass alle Möglichkeiten abgedeckt werden. Ein ähnliches Problem ist die Auswahl bestimmter Bibliotheken oder die Behandlung typischer Cross-Cutting-Aspekte.

Um die Komplexität zu reduzieren, spaltet man die Generierung in mehrere Stufen auf:

- Die erste Stufe liest Konfigurationsinformationen und bereitet den Generator auf die eigentliche Transformation vor.
- Die zweite Stufe beinhaltet die Ausführung der Transformation unter Benutzung der Informationen, welche in der ersten Stufe aufbereitet worden sind.

Beispielsweise kann die erste Phase dem Generator plattformabhängige Template-Dateien „einspielen“, die zweite Phase führt den Generator unter Verwendung dieser Templates aus.

Einige Werkzeuge verfügen über eine dritte Stufe, in welcher in der zweiten Stufe generierte Tokens durch Informationen aus der ersten Stufe ersetzt werden können. Dies kann zum Beispiel dazu verwendet werden, manuell geschriebenen Quellcode – insbesondere Code für spezifische Varianten – mit generiertem Quellcode automatisiert zu mischen.

In vielen Fällen wird die erste Stufe in einem anderen Werkzeug (wie z.B. einer Batch- oder Ant-Datei) realisiert als die zweite Stufe. Während die erste Stufe eher einen Werkzeugkonfigurationsschritt darstellt, generiert die zweite Stufe die gewünschte Applikationsfunktionalität oder -Struktur.

Werkzeuge (oder Werkzeugketten), die mehrstufige Generierung nicht unterstützen, führen leider oftmals dazu, dass die Struktur der Transformationen stärker durch die Werkzeugarchitektur geprägt ist als durch die Struktur der Domäne.

11.2.4. Modulare Transformationen

Die Bedeutung modularer Transformationen aus Sicht der Domänenarchitektur haben wir in Abschnitt <Ref Trans> bereits beschrieben. Aus Sicht der MDSD-Werkzeuge stellt die Unterstützung dieses Paradigmas ggf. ein Konstruktionsmerkmal oder Auswahlkriterium dar.

11.2.5. Modelltransformationen sind “First Class Citizens”

Wie wir gesehen haben, sind Modelltransformationen keine „Nebensächlichkeiten“ im Rahmen der Domänenarchitektur, sondern sie sind vollwertige, essenzielle Artefakte, genauso wie Modelle und manuell erstellter Quellcode. Dies hat Auswirkungen sowohl auf den

Umgang der Entwickler mit den Transformationen als auch auf Werkzeuge:

- Entwickler sollten Transformationen vernünftig strukturieren, modularisieren und regelmäßig refaktorisieren.
- Werkzeuge müssen die Transformationen in einer Art und Weise ablegen, dass sie mit den anderen im Projekt verwendeten Werkzeugen versioniert und verwaltet werden können, ggf. in verteilten Teams.

Einige der heutigen Werkzeuge behandeln Transformationen immer noch als „zweitrangig“ verglichen mit Modellen und traditionellem Quellcode. Glücklicherweise speichern Template-basierte Generatoren einzelne Code-Templates meist als separate (Text-)Dateien, die sich beispielweise gut mittels CVS versionieren (und ggf. mergen) lassen. Nichtsdestotrotz sollte festgehalten werden, dass es noch kein bewährtes und generell akzeptiertes Paradigma für die Repräsentation und Verwaltung von Transformationen gibt.

.

12. Teil 3: Prozesse und Engineering

Nachdem wir uns im letzten Teil den konstruktiven und technischen Gesichtspunkten von Domänenarchitekturen genähert haben, wollen wir nun aufzeigen, wie dieses zentrale MDSD-Artefakt im Projektverlauf entsteht und wie die Entstehung mit der eigentlichen Anwendungsentwicklung synchronisiert werden kann. Das richtige Vorgehen ist für MDSD-Projekte essenziell. Dieser Teil des Buches betrachtet MDSD daher aus Sicht der konkreten Projektabwicklung und stellt relevante Prozessbausteine und Engineering-Methoden vor.

Wir beginnen mit einer Reihe von Best Practices, die zu einem praktischen und pragmatischen Entwicklungsprozess kombiniert werden können. Anschließend gehen wir auf die Themen Test und Versionierung im Kontext von MDSD ein. Es folgt ein Abriss über Product Line Engineering, der durch ein reales Projektbeispiel begleitet und durch eine ebenso reale Fallstudie im Anschluss konkretisiert wird. Zum Schluss geht es um den Übergang von objektorientierter Geschäftsprozessanalyse zu modellgetriebener Softwareentwicklung.

13. MDSD-Prozessbausteine und Best Practices

Mit Jorn Bettin

13.1. Einleitung

Dieses Kapitel zeigt wichtige, praxiserprobte Vorgehensweisen, die den Einsatz modellgetriebener Softwareentwicklung in Projekten ermöglichen und unterstützen. Wir abstrahieren hier vom architekturzentrierten Fall, wie wir ihn in <Ref Basis> und <Ref CSI> bereits skizziert haben. Die dort zum Teil intuitiv verwendeten Techniken werden hier noch einmal explizit aufgearbeitet, verallgemeinert und vertieft. Die meisten Vorgehensweisen und Praktiken lassen sich relativ leicht auf MDSD im Allgemeinen übertragen. Techniken, die nur im architekturzentrierten Fall Sinn machen, oder dort eine spezielle Ausgestaltung besitzen, werden explizit gekennzeichnet. Wir setzen auf der in <Ref Begriffsb.> definierten MDSD-Terminologie auf, so dass es mindestens zweckmäßig ist, dieses Kapitel vorher gelesen zu haben.

Wir haben nicht die Absicht, hier einen in sich geschlossenen und vollständigen Entwicklungsprozess vorzustellen – dafür gibt es je nach Geschmacksrichtung von agil bis schwergewichtig bereits hinreichend Literatur. Stattdessen konzentrieren wir uns auf diejenigen prozessualen Aspekte, die im Rahmen der modellgetriebenen Softwareentwicklung im Sinne einer *Anpassung* (Tailoring) relevant sind. Umgekehrt bedeutet dies auch, dass z.B. ein Freiheitsgrad darin besteht, wie formal diese Best Practices Eingang in den Projektalltag finden.

Konkret empfehlen wir als Einbettung iterativ inkrementelle und insbesondere agile Entwicklungsmethoden. Wie bereits erwähnt, steht MDSD nicht im Widerspruch zu Letzteren, sondern ist vielmehr geeignet, ihre Vorteile zu skalieren. MDSD lässt sich theoretisch auch

mit wasserfallartigen Vorgehensmodellen kombinieren – allerdings bleiben die typischen Integrationsrisiken dieser Modelle bestehen, weswegen wir ihnen ganz unabhängig von MDSD kritisch gegenüberstehen.

13.2. Trennung von Anwendungs- und Domänenarchitektur-Entwicklung

13.2.1. Grundprinzip

Wir haben bereits in der Fallstudie (<Ref CS>) gesehen, welche Vorteile sich aus der Trennung von fachlicher Anwendungsentwicklung und technischer Infrastrukturentwicklung ergeben. Wir sind in der Lage, die Softwarearchitektur zu formalisieren und generativ zu unterstützen – ganz unabhängig von der konkreten Anwendung. Im Beispiel haben wir dabei offenkundig den architekturzentrierten Fall behandelt: Die Domäne lautete „Softwareinfrastruktur für E-Business-Anwendungen“, und es entstand eine Domänenarchitektur mit einer entsprechenden architekturzentrierten DSL (UML Profil), einer ebensolchen Plattform (J2EE + Struts) und passenden Generator-Templates. Verallgemeinern wir nun das Prinzip, erhalten wir das in Abb. 101 Gezeigte.

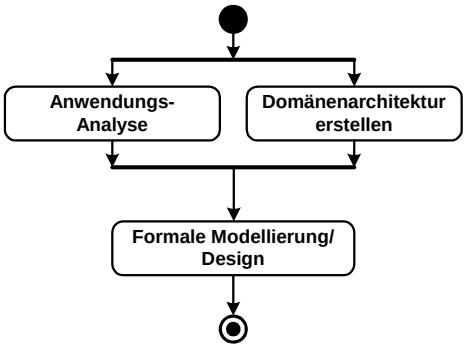


Abb. 101 *Fachliche Analyse und Domänenarchitektur als Basis für formale Modellierung*

Eine der wichtigsten Grundideen von MDSD ist die Erkenntnis, dass der Schritt der formalen Modellierung zwei Voraussetzungen impliziert, die zwar nicht ohne Wechselwirkung, aber doch weit gehend nebenläufig hergestellt werden können:

- Die funktionalen Anforderungen für eine Iteration bzw. ein Inkrement der konkreten Anwendung müssen bekannt sein.

- Die formale Sprache, die zur Modellierung verwendet werden soll (DSL), muss definiert sein. Für eine automatisierte Weiterverarbeitung muss zusätzlich eine Bindung dieser Sprache an eine konkrete MDSD-Plattform in Form von Transformationsvorschriften existieren. Dies subsummiert der Begriff Domänenarchitektur (<Ref Begr.>).

Die Domänenarchitektur formalisiert und unterstützt, wie der Name schon sagt, eine Domäne (z.B. „Softwareinfrastruktur für E-Business-Anwendungen“, „Versicherungen“ oder „Embedded Systems“) und ist im Prinzip unabhängig von einer einzelnen Anwendung (Unikat) – anders ausgedrückt: Sie spannt eine Software-Systemfamilie auf.

Das obige Aktivitätsdiagramm soll nicht als Wasserfallmodell missverstanden werden. Es zeigt vielmehr nur das Grundprinzip, das jeder Iteration zugrunde liegt – unabhängig von der Gewichtung.

Durch die formale Modellierung wird die Fachlichkeit der konkreten Anwendung mit den Konzepten, die die Domänenarchitektur bietet, verbunden – genauer gesagt: Die Fachlichkeit wird in der Sprache (DSL), welche die Domänenarchitektur zur Verfügung stellt, ausgedrückt. Das formale Modell wird anschließend generatorgestützt transformiert und auf die Plattform abgebildet. Wie dies im architekturzentrierten Fall aussehen kann, haben wir bereits gesehen. Ein Beispiel für die Domäne „Embedded Systems“ zeigt Kapitel <Ref CSII>. Im Fall der Domäne „Versicherungen“ könnte die DSL versicherungsfachliche Konstrukte zur kompakten Modellierung von Versicherungsprodukten enthalten. Die Plattform bestünde dann aus vorgefertigten fachlichen Komponenten wie *Tarifrechner* oder *VertragsdatenErfassung*, und die Transformationen erzeugen aus dem Modell eines Versicherungsproduktes z.B. Konfigurationen für die fachlichen Komponenten, die zur Laufzeit ausgewertet werden. So entsteht aus dem Modell dann eine Versicherungsanwendung, die zu 100% automatisch aus dem Modell erzeugt wird und keine manuelle Kodierung wie im architekturzentrierten Fall erfordert.

Das oben dargestellte Grundprinzip legt nun nahe, die Trennung, die auf der Ebene der Software-Artefakte besteht, auf der prozessualen und organisatorischen Ebene fortzusetzen, um die positiven Effekte zu maximieren. Entsprechend schlagen wir eine Trennung von (Domänen-)Architektur-Entwicklungsstrang und Anwendungs-Entwicklungsstrang vor.

13.2.2. Architektur-Entwicklungsstrang

Wir beschäftigen uns in diesem Abschnitt mit Artefakten und Tätigkeiten, die zur Erstellung einer Domänenarchitektur notwendig oder hilfreich sind. Der MDSD-Architektur-Entwicklungsstrang zielt auf Wiederverwendbarkeit, Qualitäts- und Effizienzsteigerung. Er ist das MDSD-Kernstück aus prozessualer Sicht. Abb. 102 zoomt also zunächst einmal in die im letzten Abschnitt gezeigte Aktivität *Domänenarchitektur erstellen* hinein.

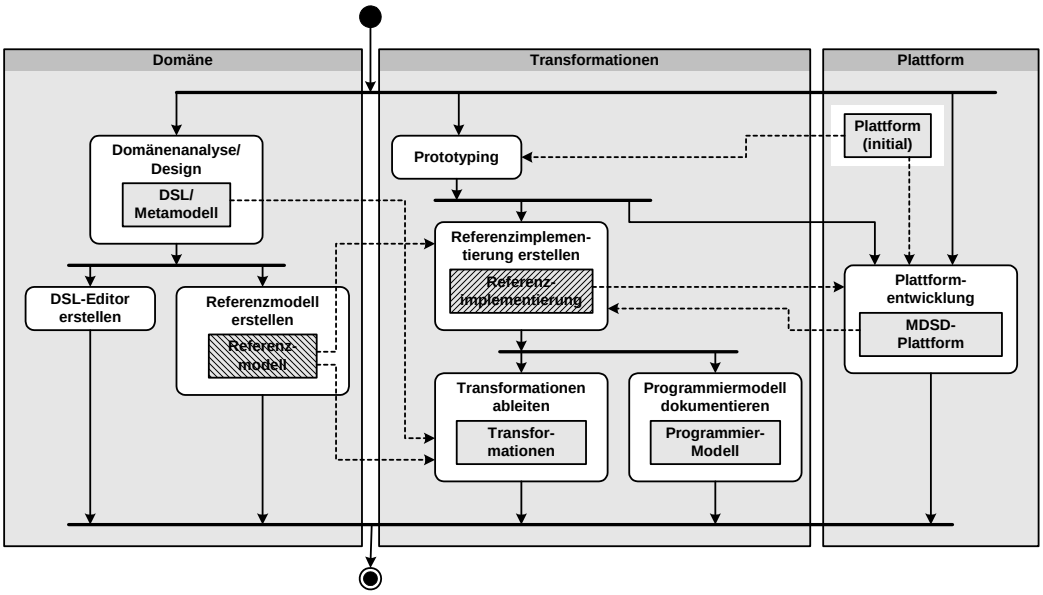


Abb. 102 *Erstellung der Domänenarchitektur*

Die im Diagramm dargestellten Partitionen unterteilen die Aktivitäten in die Kategorien *Domäne*, *Transformation* und *Plattform*. In jeder Kategorie werden Artefakte der Domänenarchitektur (grau, nicht schraffiert) produziert. Vorher entstehen ggf. wichtige Zwischenergebnisse (grau, schraffiert). Es ist lediglich der Normalfall mit den wichtigsten Abhängigkeiten dargestellt, da alles darüber hinaus erfahrungsgemäß eher zur Verschleierung des Wesentlichen führt. Insbesondere sind keine Iterationszyklen dargestellt. Am Anfang eines Projektes kann ein Durchlauf mehrere Wochen in Anspruch nehmen, im späteren Verlauf dann je nach Umfang der Erweiterungen oder Änderungen nur noch wenige Stunden oder sogar Minuten. Auch können die Bestandteile der Domänenarchitektur bei Projektbeginn schon vorhanden sein (Re-Use) oder es entsteht ein Derivat einer bestehenden Domänenarchitektur.

Zu Beginn eines Projektes sollte es zweckmäßigerweise eine *Elaborations*-Phase geben, in der der Architektur-Entwicklungsstrang quasi als Bootstrapping initial einmal durchlaufen wird. Andererseits kann dieser Punkt vollkommen entfallen, wenn sich das Projekt auf „bekanntem Terrain“ bewegt (d.h. ein Element einer bereits vorhandenen Software-Systemfamilie ist).

Die wichtigsten Schritte (Tätigkeiten/Aktivitäten) und Ergebnistypen im Einzelnen:

Prototyping

Oftmals existiert zu Beginn eines Projektes bereits eine „Plattform“, wie z.B. J2EE oder spezifische Frameworks, die Verwendung finden sollen. Ein Ziel des MDSD-Architektur-Entwicklungsstranges ist es, diese Artefakte in einer semantisch reichen und domänenspezifischen MDSD-Plattform (<Ref Archi::Plat>) aufgehen zu lassen.

Dazu ist es wie immer zweckmäßig, durch einen Prototypen im Sinne eines technischen Durchstichs erste Erfahrungen zu sammeln und die Dinge miteinander zu verknüpfen. Der Prototyp ist damit u.a. auch als Vorstufe zur MDSD-Plattform zu sehen.

Plattform entwickeln

Den Begriff der MDSD-Plattform haben wir bereits in Kapitel <Ref Begr.> definiert und in Kapitel <Ref Archi> die konstruktiven Merkmale behandelt. Als Synonym verwenden wir auch den Begriff des *Laufzeitsystems* der Domänenarchitektur. Laufzeitsystem-Komponenten sind ideale Kandidaten für Re-Use – sogar über die Grenzen von Software-Systemfamilien hinweg.

Die Plattform bildet, wie wir bereits gesehen haben, ein wichtiges Mittel, um generierten und nicht generierten Code abzustützen und die Transformationen einfach zu halten.

Der generative Anteil der Domänenarchitektur besitzt eine Abhängigkeit zu den benutzten Laufzeitsystem-Komponenten (Plattform), aber nicht umgekehrt.

Auch die Entwicklung der Plattform kann/sollte iterativ voranschreiten. Refactoring-Techniken (siehe [Fow99]) lassen sich gerade hier äußerst gewinnbringend einsetzen.

Zu beachten ist auch, dass sich die Grenze zwischen Plattform und Generat im Verlauf der Evolution der Domänenarchitektur verschieben kann – sowohl in die eine als auch in die andere Richtung. Entsprechende Kriterien haben wir bereits in <Ref Archi> diskutiert.

Referenzimplementierung erstellen

Die *Referenzimplementierung* ist zwar „nur“ ein Zwischenergebnis des MDSD-Architektur-Entwicklungsstranges, jedoch ein sehr wichtiges bei der Entstehung einer Domänenarchitektur.

Die Referenzimplementierung darf nicht als einfaches, isoliertes Beispiel missverstanden werden, aus dem man bei Bedarf Anregungen für die Implementierung bezieht. Sie kann zwar aus einem Prototyp entstehen, hat aber einen viel höheren Anspruch: Zusammen mit dem Referenzmodell/Design zeigt sie die Anwendung und Umsetzung der zur Domäne gehörenden DSL. Diese *zweiteilige* Referenz verdeutlicht den Übergang vom Modell zur Implementierung auf der gegebenen Plattform. Bei einer neuen Software-Systemfamilie wird die Referenzimplementierung zunächst „von Hand“ erstellt. Später werden aus ihr die Transformationen abgeleitet. Die generative Umsetzung des Referenzmodelles (ggf. plus manuell programmierte Fachlogik) muss dann wiederum eine lauffähige Referenzimplementierung ergeben.

Die volle Wertschöpfung einer Referenzimplementierung im Sinne von MDSD ergibt sich also erst im Zusammenspiel mit einem Referenzmodell (und damit einer DSL – s.u.).

Der konkrete fachliche Gehalt einer Referenzimplementierung ist im Grunde irrelevant – es geht ausschließlich um die Domäne. Dennoch wird in der Regel ein fachlich halbwegs sinnvoller UseCase – allerdings mit minimalistischer Ausprägung – implementiert. Viel wichtiger ist eine hinreichende Überdeckung von Konstrukten der DSL und ihren Kombinationen.

Zugleich zeigt die Referenzimplementierung auch die Verwendung der MDSD-Plattform und damit deren Schnittstelle.

Bei einer neuen Software-Systemfamilie wird die Referenzimplementierung zunächst natürlich komplett von Hand erstellt. Sobald dann aber die automatischen Transformationen vorhanden sind, wird die Referenzimplementierung auf die manuell programmierte Fachlogik reduziert (sofern vorhanden). Der restliche Rahmen ist dann aus dem Referenzmodell (s.u.) heraus generierbar.

Wenn Sie bereits eine Reihe von Anwendungen innerhalb einer Domäne besitzen und jetzt auf einen modellgetriebenen Produktionsprozess umstellen möchten, kann die Domänenarchitektur auch aus den bestehenden Anwendungen extrahiert werden, sofern die Implementierungen entsprechend wohlstrukturiert sind. Bei typischen Legacy-Anwendungen ist dies aber in der Regel nicht der Fall.

Es ist im Prinzip praktikabel, die Referenzimplementierung nur zum Bootstrapping der Domänenarchitektur zu verwenden und die weitere Evolution dann an ausgewählten UseCases echter Anwendungen zu

vollführen. Es entfällt dann zwar der Pflegeaufwand für die separate Referenzimplementierung, andererseits besteht aber die Gefahr, den Überblick zu verlieren, weil ein echter UseCase eben nicht mehr fachlich minimalistisch ist. Außerdem fehlt evtl. die Überdeckung wichtiger DSL-Konstrukte, sofern diese zufällig in den realen UseCases nicht verwendet werden. Wir empfehlen aus unserer praktischen Erfahrung daher, im Zweifelsfall die Referenzimplementierung beizubehalten und als ein Artefakt der Domänenarchitektur zu verstehen.

Noch ein wichtiger Hinweis: Die Evolution der Domänenarchitektur und insbesondere die frühe Phase des Bootstrappings (z.B. DSL-Findung und Stabilisierung) werden in aller Regel Rückwirkungen auf die Referenzimplementierung und ggf. auch auf die Plattform haben. Dies ist im Sinne der iterativ inkrementellen Softwareentwicklung vollkommen normal. Ein striktes Wasserfallmodell wäre an dieser Stelle sicherlich kontraproduktiv.

Domänen-Analyse/Design

Diese Tätigkeit dient im Wesentlichen der Findung des Metamodelles der Domäne und einer dazu passenden konkreten DSL. Wir haben diesen Themen auf Grund ihrer Bedeutung jeweils vertiefende Kapitel gewidmet (<Ref PLE>, <Ref MM>). An dieser Stelle seien daher nur allgemeine Best Practices zur Konstruktion einer DSL genannt.

Eine *architekturzentrierte* DSL nennen wir auch *Designsprache*. Typisch, wenn auch nicht zwingend für eine solche Designsprache wird die Verwendung von UML als Basis sein, wie in der ersten Fallstudie ausführlich gezeigt (vgl. <Ref CSI>). Für einige Aspekte, wie z.B. die Modellierung von GUI-Layout, ist UML allerdings vollkommen ungeeignet, so dass man ggf. auf eine andere Form ausweichen wird. Letztlich spielt aber die konkrete Syntax gegenüber der abstrakten nur eine untergeordnete Rolle (vgl. <Ref Begr> und <Ref Tool::IgnSyntx>).

Beim Entwurf einer DSL sollten allgemein folgende Regeln berücksichtigt werden:

- Die DSL sollte möglichst abstrakt und frei von technischen Termini sein, die ein Implementierungsdetail der MDSD-Plattform (s.o.) darstellen (z.B. Verwendung von *EntityObject* statt *EntityBean* als Stereotyp). Diese Maßgabe führt auf Modelle, denen man ihre technische Umsetzung erst im Kontext einer Plattformbindung ansieht. Dies erleichtert natürlich z.B. spätere Migrationen oder architektonische Änderungen. Außerdem sind solche DSLs für verschiedene Software-Systemfamilien wiederverwendbar. Wir haben diese Modelle gelegentlich intuitiv

als plattformunabhängig (PIM) bezeichnet, doch ist dieser Begriff relativ zu sehen (vgl. <Ref Begr>): Sie sind typischerweise *unabhängig* bezüglich einer Industriestandard-Plattform wie J2EE, aber *abhängig* von den Konzepten der MDSD-Plattform (s.u.) der Domänenarchitektur, denn die DSL hat ja gerade den Zweck, diese Konzepte auf Modellebene verwenden zu können.

- Die DSL sollte möglichst alle relevanten Konzepte der Domäne durch Sprachelemente semantisch überdecken. Idealerweise sind alle schematisch implementierbaren Codefragmente der Referenzimplementierung durch Konstrukte der DSL abgedeckt. Unsere Fallstudie (vgl. <Ref CSI>) zeigt dies beispielhaft: Die Stereotypen *<<Entity>>*, *<<ProcessObject>>*, *<<Presentation>>*, *<<SystemUseCase>>*, *<<ActivityController>>*, *<<ControllerState>>* usw. benennen und überdecken gerade die Architekturkonzepte der Beispielanwendung.

- Die DSL sollte so kompakt wie möglich sein und keine Redundanzen aufweisen. Sie kann z.B. auch dynamische Konstrukte z.B. zur Abbildung von Geschäftsprozessen und Controller-Logik in Form von Aktivitäts- bzw. Zustandsdiagrammen besitzen.

- Die DSL muss die Wohlgeformtheit von Modellen überprüfbar machen. Insbesondere wenn die DSL eine Spezialisierung einer allgemeineren Sprache ist (z.B. ein UML-Profil als Spezialisierung der UML), muss sichergestellt sein, dass alle im Sinne der DSL unzulässigen Modellierungsalternativen, welche die Basissprache standardmäßig bietet, ausgeschlossen sind.

Mit der Konzeption der DSL legt die Architektin unweigerlich auch die Trennlinie zwischen Generat und Fachlogik und damit die Freiheitsgrade der Entwickler fest. Einen Extrempunkt stellt der Versuch dar, die DSL soweit auszuweiten, dass überhaupt keine manuelle Programmierung mehr erfolgen muss. Wir haben bereits erläutert, unter welchen Voraussetzungen dies sinnvoll sein kann (<Ref Begr und Archi>).

Für typische Business-Anwendungen im Kontext *architekturzentrierter* Entwicklung ist dies in der Breite sicherlich heute weder praktikabel noch sinnvoll. Es geht also um die Frage, welche Implementierungsaspekte eine *architekturzentrierte* DSL abdecken sollte und welche nicht. Als Hilfestellung können folgende Leitfragen dienen:

- Sind in der Referenzimplementierung oder einer bereits generativ unterstützten Anwendung Codefragmente mit Copy-Paste-Charakter vorhanden, die noch nicht generiert wurden ?
- Wäre eine Modellierung dieser Aspekte einfach und kompakt, oder womöglich aufwändiger und umfangreicher als die manuelle Kodierung?

Bei dieser wichtigen Abwägung bedarf es einiges an Erfahrung und Fingerspitzengefühl.

Die DSL muss dokumentiert sein, damit sie anwendbar ist. Dazu gehören folgende Teile:

- Die konkrete Syntax (z.B. ein UML-Profil oder ein XML-Schema).
- Die abstrakte Syntax (z.B. als MOF-Diagramm – vgl. Kapitel <Ref MM>).
- Die statische Semantik, d.h. Constraints bzw. Modellierungsregeln: Welche Konstellationen sind nicht erlaubt oder zwingend vorgeschrieben. Im Falle einer UML-basierten Modellierungssprache ist dies Teil des Profils (vgl. <Ref CS1>).
- Die Semantik (Bedeutung der Sprachkonstrukte in textueller Form). Formal wird die Semantik im Grunde erst durch die Referenzimplementierung bzw. Transformationsvorschriften definiert, dadurch ist die Definition allerdings bereits auf eine konkrete Plattform ausgerichtet. Um dies zu vermeiden und den Wiederverwendungsgrad zu erhöhen, sollte die Bedeutung in möglichst allgemeiner Form, unter Verwendung der Architekturkonzepte und nicht der Plattform selbst beschrieben werden.
- Ggf. ein User-Guide mit Verweisen auf das Referenz-Design (s.u.)

Sicherlich ist die Definition einer adäquaten Modellierungssprache eine der größten Herausforderungen im Kontext modellgetriebener Softwareentwicklung. Kapitel <Ref PLE> gibt dazu konkrete Hinweise. Es ist aber in jedem Falle einiges an Erfahrung oder eine vorhandene Basis, wie z.B. das UML-Profil in der ersten Fallstudie (vgl. <Ref CSI>), erforderlich. Im Übrigen kann und darf auch die Modellierungssprache einer gewissen Evolution unterliegen. Es ist z.B. durchaus typisch, dass man erst nach einiger Zeit feststellt, dass gewisse Modellierungskonstrukte im ursprünglichen Entwurf nicht abstrakt

genug waren und einer Generalisierung bedürfen. Refactoring ist daher auch auf dieser „Meta-Ebene“ ein starker Verbündeter. Zugegebenermaßen können *grundlegende* Änderungen der abstrakten Syntax (also der eigentlichen Sprachstruktur) und der Semantik weitreichende Änderungen bereits vorhandener Transformationsvorschriften nach sich ziehen. Sprach*erweiterungen* hingegen sind eher unkritisch. In der Elaborations-Phase (s.o.) sollte daher entsprechendes Gewicht auf der DSL liegen, so dass die Transformationsvorschriften erst abgeleitet werden, wenn Referenzimplementierung und Referenz-Design ein stimmiges Bild abgeben.

In der Regel wird man eine Modellierungssprache nicht im „luftleeren Raum“ entwerfen. Um ein Gefühl dafür zu bekommen, ob die gewählten Konstrukte adäquat und ergonomisch sind, muss man ein angewandtes Vorkommen, also ein Beispiel, betrachten, an dem man den Sprachentwurf kontinuierlich messen kann. Dazu dient unter anderem das Referenz-Design.

Referenzmodell/Design erstellen

Das *Referenzmodell* ist eine Instanz der DSL, d.h. es drückt ein fachliches Beispiel mit den Mitteln der DSL aus.

Entscheidend dabei ist, wie bereits erwähnt, das Zusammenspiel mit der Referenzimplementierung: Referenzmodell und Referenzimplementierung zusammen zeigen exemplarisch die Syntax und Semantik der DSL bis auf Implementierungsebene und konkretisieren damit das Konzept der Domänenarchitektur bis ins Detail.

Programmiermodell dokumentieren

Die Definition eines Programmiermodelles ist nur relevant, wenn die Domänenarchitektur „semantische Lücken“ enthält – soll heißen: Durch die Modell-Transformation entsteht ein Code-*Rahmen*, der durch den Anwendungsentwickler programmiersprachlich ergänzt werden muss, damit eine lauffähige Anwendung entsteht.

Der Begriff des Programmiermodelles wurde bereits etabliert (vgl. <Ref Archi>. So definiert CORBA (…) beispielsweise eine abstrakte Schnittstellenbeschreibungssprache (IDL = Interface Definition Language) mit Mappings auf verschiedene Programmiersprachen, wie C++ und Java. Das Programmiermodell des jeweiligen Sprach-Mappings definiert dann (Namens-)Konventionen bezüglich der Abbildung einzelner IDL-Konstrukte auf die Sprachkonstrukte wie Klassen,

Attribute und Methoden. Diese Konventionen werden vom IDL-Compiler (Generator) eingehalten, und er erzeugt entsprechende Signaturen und Skeletons. Umgekehrt entsteht so durch Generierung aus der IDL-Beschreibung eine definierte API, mit deren Hilfe der Anwendungsentwickler die Fachlogik programmieren kann. Des Weiteren definiert das Programmiermodell bestimmte Idiome und Muster, wie man mit der betreffenden Architektur richtig umgeht – nicht alle „Regeln“ lassen sich nämlich vom Compiler oder dem Laufzeitsystem automatisiert überprüfen.

Überträgt man dies auf MDSD, so beschreibt das Programmiermodell also die Sicht des Anwendungsentwicklers auf die Domänenarchitektur (genauer: Transformationen bzw. Generat und Plattform) und Regeln, wie er richtig mit ihr umgeht.

Ziel ist es also unter anderem, dem Entwickler transparent zu machen, welche programmiersprachlichen Artefakte aus den Konstrukten der DSL werden, die *für ihn* relevant sind. Eine im Modell notierte Assoziation beispielsweise kann auf Implementierungsebene (z.B. in Java) eine Menge von Zugriffsoperationen wie *getElementAt()*, *addElement()*, *removeElement()* usw. mit einer spezifischen Implementierung bedeuten. Die Implementierung ist aus Sicht des Entwicklers irrelevant, er benötigt nur die Signaturen. Diese bilden dann das Programmiermodell des Konstruktes „Assoziation“ der DSL.

Das Programmiermodell kann bei Bedarf relativ einfach in Form einer Tabelle dokumentiert werden, die neben dem betreffenden Konstrukt der DSL einen Verweis auf einen adäquaten Ausschnitt des Referenzmodelles und der Referenzimplementierung enthält. Über die Notwendigkeit der expliziten *Dokumentation* des Programmiermodelles kann man geteilter Meinung sein, die bewusste *Definition* in Form von MDSD-Transformation und Plattform-API ist eine Notwendigkeit. Die Referenzimplementierung stellt eine *implizite* Dokumentation dar. Enthält das Programmiermodell „Do’s and Don’ts“, die nicht toolgestützt erzwungen oder kontrolliert werden können, ist eine gute explizite Dokumentation allerdings erforderlich.

Das initiale Programmiermodell und seine Dokumentation entsteht üblicherweise im Rahmen der Elaborations-Phase und ist selbstverständlich auch Gegenstand iterativer Verbesserung.

Transformationen ableiten

Diese Aktivität formalisiert eine Plattformzuordnung zur DSL und das Programmiermodell so weit, dass eine automatische Transformation ein gegebenes Anwendungsmodell automatisch in eine Implementierung oder einen Rahmen überführen kann.

In unserer Fallstudie (vgl. Kapitel <Ref CSI>) wurde in diesem Schritt ein Satz von *Generator-Templates* erstellt. Die Templates wurden aus der Referenzimplementierung mit Hilfe des Referenzmodelles abgeleitet. Das im Fallbeispiel verwendete Generator-Framework verlagert einen Teil der Meta-Programmierung auf die Erstellung eines in Java implementierten Metamodelles der DSL. Diese Trennung ist aus Prozess-Sicht jedoch lediglich ein Implementierungsdetail und daher irrelevant.

Falls nicht die gesamte Fachlogik einer Anwendung vollständig durch die DSL ausdrückbar ist, sind Techniken zur Integration von generiertem und nicht-generiertem Code erforderlich (vgl. <Ref Archi>).

DSL-Editor erstellen

Nicht alle DSLs sind UML Profile, so dass ein Standard-Werkzeug zum Einsatz kommen kann. Im Falle hochspezialisierter Domänen ist es durchaus üblich und angebracht, ein spezielles Werkzeug zur Eingabe von DSL-konformen Modellen zu erstellen, um z.B. die Ergonomie und damit die Effizienz des MDSD-Ansatzes weiter zu steigern. Letztlich handelt es sich also um eine Frage von Kosten und Nutzen, die nur individuell beantwortet werden kann. Weiterführende Betrachtungen finden sich in Kapitel <Ref PLE>.

13.2.3. Anwendungs-Entwicklungsstrang

In diesem Abschnitt widmen wir uns der Sicht des Anwendungsentwicklers (wir sehen dies wohlgemerkt als Rolle), der mit einer gegebenen Domänenarchitektur (siehe Abb. 103) arbeitet.

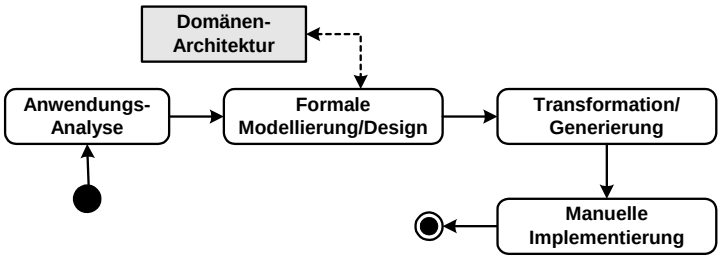


Abb. 103*Aktivitäten im Anwendungs-Entwicklungsstrang*

Auch hier ist der einfache Normalfall ohne Iterationszyklen dargestellt. Ein Durchlauf kann je nach Intensität der einzelnen Schritte einige Tage oder nur Minuten in Anspruch nehmen.

Formale Modellierung/Design

In diesem Schritt treffen Analyse- und Architektur-Strang aufeinander: Die fachlichen Anforderungen werden nun in der Sprache der Domänenarchitektur ausgedrückt – der DSL. Das Referenzmodell dient dabei als Orientierungshilfe. Der Schritt stellt einen echten Informationsgewinn dar und ist daher nicht automatisierbar.

Genau an dieser Stelle sollte eine funktionierende Feedback-Schleife zum Architektur-Entwicklungsstrang etabliert sein, denn gerade im frühen Entwicklungsstadium einer Domänenarchitektur wird es häufiger dazu kommen, dass Unzulänglichkeiten oder Fehler in der DSL entdeckt werden. Durchaus typisch ist auch der Effekt, dass während der Anwendungsentwicklung weiteres Automationspotenzial entdeckt wird, was dann zu einer Erweiterung oder sogar Generalisierung der DSL führt.

Generierung

Dies ist ein rein mechanisch durchführbarer Schritt. Es findet kein Informationsgewinn gegenüber dem formalen Modell statt. Es wird mittels der Transformationen der Domänenarchitektur vollautomatisch in eine zur MDSD-Plattform passende Form überführt. Dabei können an definierten Stellen im Generat Integrationspunkte für manuell zu programmierende Fachlogik entstehen. Dies können z.B. geschützte Bereiche (Protected Regions) sein, deren Inhalt bei iterativer Generierung erhalten bleibt, oder Implementierungsklassen, auf die der Implementierungsrahmen delegiert (vgl. <Ref Archi>).

Manuelle Implementierung

Anwendungslogik, welche in der DSL *nicht* ausdrückbar ist, muss nach erfolgter Generierung manuell ergänzt werden. Bei unserer Fallstudie handelt es sich genau um die Inhalte der geschützten Bereiche im Implementierungsrahmen.

Auch während der Implementierung können sich selbstverständlich Rückwirkungen auf die Domänenarchitektur ergeben. Die Projektorganisation muss entsprechende Feedback-Schleifen mindestens zulassen.

13.2.4.Organisatorische Aspekte

Die Trennung von Anwendungs- und Domänenarchitektur-Entwicklung sollte idealerweise nicht nur durch prozessuale, sondern auch durch organisatorische Maßnahmen gestützt werden. Diese werden in den Kapiteln <Ref Orga> und <Ref Vers> behandelt.

13.3. Zweigleisig iterative Entwicklung

Wir haben nun ausführlich die Rollen- und Artefakt-Trennung von Anwendungs- und Domänenarchitektur-Entwicklung diskutiert. Dieser Abschnitt beschäftigt sich mit der Synchronisation der beiden Stränge. Es gibt offensichtlich eine gerichtete Abhängigkeit von der Anwendungs- auf die Domänenarchitektur-Entwicklung – analog zum Einsatz eines Frameworks in einem Projekt. Aus Sicht des Anforderungsmanagements bedeutet dies, dass die Anwendungsentwicklung die Kundenrolle gegenüber der Domänenarchitektur-Entwicklung einnimmt.

Sie sollten die Domänenarchitektur sowie mindestens eine darauf basierende Anwendung parallel in mindestens virtuell getrennten Projekten entwickeln. Das bedeutet in der Praxis, dass die Anwendungsentwickler in einer Iteration immer die in der letzten Iteration entwickelte Domänenarchitektur verwenden, also die Domänenarchitektur-Entwickler immer eine Iteration voraus sind. Stellen Sie sicher, dass die Anwendungsentwickler den Domänenarchitektur-Entwicklern sofortiges Feedback liefern.

Neue Versionen der Domänenarchitektur werden ggf. immer zu Beginn einer Iteration in die Anwendungsentwicklung integriert. Um ein ausreichendes Maß an Agilität im Entwicklungsprozess zu erreichen, sollte die Dauer von Iterationen auf maximal vier bis sechs Wochen beschränkt sein. Idealerweise und der Einfachheit halber empfehlen wir eine konstante Dauer für alle Iterationen vorzuschreiben (Timeboxing). Dies führt zu einem regelmäßigen Entwicklungsrhythmus, auf den sich die Teams einspielen.

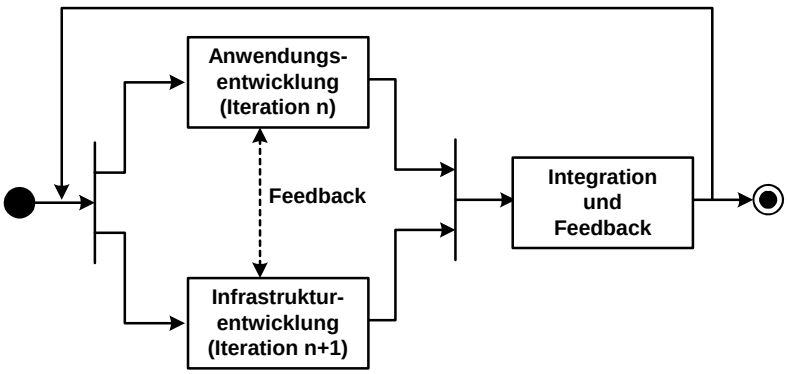


Abb. 104*Iterative, zweigleisige Entwicklung*

Beachten Sie, dass der inkrementelle, iterative Prozess auf der Basis von synchronisierten Timeboxen eine Domänenanalyse vor Einstieg in den iterativen Zyklus *nicht* ausschließt. Im Gegenteil, ein gutes Verständnis der Grundzusammenhänge der Domäne ist sogar notwendig. Sobald die Anwendungsentwicklung in vollem Gange ist, erfolgt weitere Domänenanalyse iterativ – als Teil des Architektur-Entwicklungsstranges, der sich nun in einem eigenen Projekt befindet.

Infrastrukturteams (die Domänenarchitektur-Entwickler) haben manchmal die Tendenz, sich auf interessante Technologien zu stürzen, um – mit besten Absichten – den Rest der Welt mit einer neuen Silver Bullet zu beglücken. Dieses Risiko bekommt man am besten durch Einrichtung einer Architekturgruppe (vgl. <Ref Orga>) in den Griff, welche aus Repräsentanten der Anwendungsentwicklungsteams besteht, und welche die Befugnis hat, von Iteration zu Iteration den Funktionsrahmen für die Infrastrukturentwicklung mittels Scope-Trading zu bestimmen, sowie durch Validierung von (Domänenarchitektur-)Iterationen über die Abnahme von Infrastrukturfunktionalität zu entscheiden. Somit wird sichergestellt, dass die entwickelte Domänenarchitektur einen wirklichen Mehrwert für die Anwendungsentwicklung darstellt und Ihnen bei der täglichen Arbeit tatsächlich hilft..

Timeboxing mit festem Budget, Scope-Trading und Validierung von Iterationen sind agile Techniken, die iteratives Anforderungsmanagement unterstützen und gerade in Kombination mit zweigleisig iterativer Entwicklung für MDSD hilfreich sein können. Wir wollen hier nur die Grundideen in wenigen Sätzen zusammenfassen, da diese Themen im Prinzip unabhängig von MDSD sind:

Für jede Timebox steht ein festes Budget zur Verfügung. Zu Beginn jeder Iteration werden in einem Scope-Trading-Workshop der Funktionsumfang und die Prioritäten für diese Iteration mit den

Interessenvertretern (z.B. Kunden und Endanwender) ausgehandelt – dabei müssen im Sinne der Risikominimierung auch Architekturaspekte berücksichtigt werden. Das Budget der Timebox darf dabei nicht überschritten werden. Innerhalb einer Timebox bleibt der Funktionsumfang konstant, damit die Entwickler ihr Ziel, am Ende der Timebox validierbare Software zu liefern, verfolgen können. Die Validierung am Ende einer Iteration durch die Interessenvertreter entscheidet, welche Funktionen den Anforderungen genügen und welche überarbeitet werden müssen. Zwischenzeitlich erkannte Neuanforderungen werden dann im Scope-Trading-Workshop der nächsten Timebox gleichberechtigt mit offenen Anforderungen oder bei der Validierung abgelehnter Funktionen neu priorisiert.

Weiterführende Quellen zu diesen Themen finden sich u.a. im Web unter www.mdsd.info.

Im Kontext von MDSD können diese Praktiken wie gesagt insbesondere hilfreich sein, um eine funktionierende Feedback-Schleife zwischen Anwendungs- und Domänenarchitektur-Entwicklung zu etablieren – die Anwendungsentwicklung stellt dabei in Form eines Querschnitt-Teams die Interessenvertretung für das Scope-Trading der Domänenarchitektur-Entwicklung.

14. Testen

Software kann in den seltensten Fällen auf Basis von Spezifikationen vollständig verifiziert werden. Aufgrund dieser Tatsache hat das Testen im Rahmen der Softwareentwicklung einen sehr hohen Stellenwert.

Bisher haben wir die Rolle von Tests im Entwicklungsprozess nicht explizit behandelt. Das liegt nicht etwa daran, dass wir Tests für unwichtig oder untergeordnet halten, sondern daran, dass sie bei modellgetriebener Entwicklung im Wesentlichen dieselbe Rolle spielen (sollten) wie bei anderen Methoden auch. Gerade die agilen Prozesse ([Coc01], [Bec02]) nehmen hier eine Vorbildfunktion ein: Tests stellen eine Art Sicherheitsnetz für die Änderbarkeit und Erweiterbarkeit einer Software dar. Deswegen sollten Tests nicht erst am Ende eines Projektes durchgeführt werden, sondern diese kontinuierlich begleiten.

Zudem ist die Testautomation ([HT04], [Cla04]) ein wichtiges Schlüsselelement, um eine möglichst kontinuierliche und reproduzierbare Validierung der Software während der Konstruktion zu ermöglichen. Wir wollen in diesem Abschnitt nicht bei Adam und Eva in Sachen Tests beginnen, sondern uns auf die Besonderheiten im Zusammenhang mit modellgetriebener Entwicklung im Allgemeinen und architekturzentrierter im Besonderen konzentrieren.

Wir haben beschrieben, wie wir ein Projekt in einen Architektur- und einen Anwendungs-Entwicklungsstrang aufteilen. Die beiden Stränge sind zwar z.B. durch Meilensteine miteinander synchronisiert, aber sie produzieren unterschiedliche Artefakte: Auf dem Anwendungs-Entwicklungsstrang entsteht eine Anwendung, auf dem Architektur-Entwicklungsstrang eine Domänenarchitektur. Beide Artefakte müssen getestet bzw. validiert werden. Wir beginnen mit Ersterem. Zuvor noch eine kurze, allgemeine Einführung über Testarten.

14.1. Testarten

Die Diskussion, welche Aspekte eines Softwaresystems wann und zu welchem Zweck zu testen sind, führt auf die so genannten Testarten:

- *Komponententests* (auch *Unit-Tests* genannt) werden vom Entwickler im Rahmen seiner täglichen Arbeit erstellt und dienen der Absicherung der korrekten Funktionsweise von Systemteilen (Klassen, Module, Subsysteme etc.).
- *Integrationstests* stellen das korrekte Zusammenspiel aller (bereits vorhandenen) Systemteile sicher.
- *Akzeptanz- und Abnahmetests* zeigen, ob das System aus Sicht des Kunden sinnvoll ist und seinen Anforderungen genügt.
- *Oberflächentests*[48] können entweder die Rolle von Akzeptanztests (Ergonomie) oder von Integrationstests spielen.
- *Nicht-funktionale Tests* validieren Anforderungen an die Zielarchitektur, wie Security, korrektes Transaktionsverhalten in einer Multi-User-Umgebung oder Robustheit gegenüber Hardware-Ausfällen. Eine Untermenge sind die Lasttests.
- *Lasttests* dienen der Messung von Performance (Antwortzeiten, Datendurchsatz) und Skalierbarkeit. Lasttests sind oft recht aufwändig und werden daher nur zu bestimmten Zeitpunkten im Rahmen des Projektes durchgeführt.
- *Regressionstests* werden eingesetzt, um nach einer Veränderung des Quellcodes (Hinzufügen oder Korrektur) festzustellen, ob sich unerwünschte Seiteneffekte ergeben haben. Regressionstests werden meist durch andere Testarten realisiert.

Ein äußerst wichtiges Mittel zur Steigerung der Effizienz und Effektivität von Tests ist die Testautomation. Erst reproduzierbare, automatisierte Tests bilden das Sicherheitsnetz, durch das Software änderbar bleibt. Testautomation hilft Regressionstests zu implementieren und ist praktisch für alle Testarten außer Ergonomietests einsetzbar. Die Spitze der Testautomation stellt die so genannte *kontinuierliche Integration* (Continuous Integration, [Cla04], [CRUI]) dar. Dabei werden annähernd kontinuierlich und vor allem parallel zur Entwicklung automatisierte

[48] Tests grafischer Benutzeroberflächen (GUIs)

Tests auf speziellen Testmaschinen durchgeführt, die den Entwickler asynchron (z.B. per Mail) über aufgetretene Fehler informieren.

Bei diesem Überblick fällt auf, dass grundlegende Dinge wie Programmiersprachen bzw. ihre Compiler oder Interpreter nicht explizit getestet werden. Kaum jemand dürfte heutzutage noch generierten Assembler- oder Bytecode daraufhin prüfen, ob der Compiler korrekt gearbeitet hat, da die Korrektheit dieser Tools in aller Regel vorausgesetzt werden kann.

Im Rahmen der modellgetriebenen Softwareentwicklung allerdings spielt die Domänenarchitektur durch ihren generativen Anteil eine ähnliche Rolle wie ein Compiler – ihre Korrektheit kann jedoch nicht immer vorausgesetzt werden. Doch dazu später mehr.

Auf der anderen Seite ergeben sich durch MDSD gute Chancen zur Vereinfachung der oben angeführten Testarten für den Anwendungs-Entwicklungsstrang. Darauf wollen wir im Folgenden eingehen.

14.2. Tests in der modellgetriebenen Anwendungsentwicklung

Innerhalb des Anwendungs-Entwicklungsstrangs spielen die gleichen Testarten eine Rolle, wie in nicht-modellgetriebenen Prozessen auch. Die Modelle sind Teil des Codes, da sie formal sind und durch Transformationen automatisch z.B. in 3GL-Code überführt werden. Aus dieser Perspektive handelt es sich bei der DSL also um eine (domänennahe) Programmiersprache, die aber semantische Lücken haben kann. Nach der Generierung liegt jedoch genau der Code vor, den man auch hätte, wenn man nicht modellgetrieben vorginge, und er ist daher genauso testbar. Selbst ein Test-First-Ansatz stellt hier per se keinen Widerspruch dar: Niemand hindert den Entwickler daran, *vor* der Modellierung Testcode zu schreiben, wenn dies als sinnvoll erachtet wird.

Allerdings birgt der modellgetriebene Ansatz darüber hinaus Potenziale, die auch das Erstellen von Testcode vereinfachen können:

Ein (Blackbox-)Test ist stets der Vergleich eines Ist- mit einem Soll-Zustand (wobei wir nicht zwischen Rückgabewerten von Methoden und Seiteneffekten wie Datenbankänderungen unterscheiden). Ein Test sichert also die Semantik hinter einer syntaktisch definierten Schnittstelle[49] ab. Anders formuliert *kennt* der Test die Signatur und die

[49] Die kann je nach Testart z.B. ein Java-Interface oder ein GUI sein.

Semantik der Schnittstelle eines Testlings – und genau dies kann mit Fug und Recht als *Design* bezeichnet werden. Ein Entwickler, der Testcode schreibt, spezifiziert also dadurch Designinformation. Umgekehrt kann kein Entwickler Testcode schreiben, ohne sich vorher über das Design des Testlings zumindest annähernd im Klaren zu sein.

Im Kontext von MDSD besitzen wir nun aber ein höherwertiges Mittel als eine 3GL-Sprache, um Designs auszudrücken: die DSL. Je nach „semantischer Tiefe" der DSL kann es also sehr viel effektiver sein, Designinformation zunächst im Modell und nicht gleich in Form von 3GL-Testcode zu spezifizieren, um dann nämlich genau diesen Code oder Teile davon zu generieren. Der Vorteil liegt in der Abstraktion und der damit verbundenen leichteren Änderbarkeit des Testcodes: Ein Teil des Testcodes wird in die Domänenarchitektur verlagert und kann damit zentral gepflegt und geändert werden. Die Domänenarchitektur wird damit auf die Erzeugung von *Testware* ausgeweitet. Ob sich dieses Vorgehen lohnt und in welcher Ausbaustufe, ist natürlich wesentlich vom zu erwartenden Wiederverwendungsgrad abhängig.

14.2.1.Unit-Tests

Im Rahmen der „konventionellen" Softwareentwicklung gibt es für Unit-Tests kleine aber feine Hilfs-Frameworks wie z.B. JUnit (oder C#Unit, CppUnit etc.). Bei der modellgetriebenen Entwicklung wollen wir diese Tools weiterhin einsetzen und darüber hinaus die Testerstellung möglichst sinnvoll durch MDSD-Techniken unterstützen.

Trennung von Testfällen und Testdaten

Unit-Tests sind leider meistens so programmiert, dass Testfälle und Testdaten innerhalb des Codes vermischt sind. Dies erscheint zwar zunächst besonders einfach, birgt aber unabhängig von MDSD gravierende Nachteile für den Projektverlauf:

- Testfälle sind nicht wiederverwendbar für unterschiedliche Testdaten.
- Testfälle und Testdaten verlieren an Wartbarkeit.
- Testdaten können nur von den Entwicklern modifiziert oder bereitgestellt werden.
- Die eigentliche Struktur der Testfälle wird verschleiert, was die Lesbarkeit einschränkt.

Es ist also eine gute Idee, Testfälle und Testdaten bei der Implementierung von Komponententests zu trennen. Dies kann z.B. dadurch geschehen, dass die Testdaten in Form von XML-Dokumenten oder Excel™-Dateien ausgelagert und vom eigentlichen Testfall (z.B. einer JUnit-Testmethode) zur Laufzeit typsicher eingelesen werden. Das gilt nicht nur für die Testeingaben (Stimuli), sondern auch für die Sollwerte. Dazu lässt sich entsprechende Basisfunktionalität bereitstellen.

Im Kontext von MDSD ergibt sich durch die Trennung zudem die Möglichkeit, auch die Generierung getrennt zu behandeln (Komplexitätsreduktion), bzw. nur die Testfälle generativ zu unterstützen und die Einbindung der Testdaten generisch zu lösen.

Das externe Format zur Formulierung der Testdaten wird durch die Trennung Teil einer „Test"-Partition der DSL – oder anders formuliert: Testdaten können MDSD-Modelle sein – egal ob sie noch transformiert oder direkt von der (Test-)Plattform interpretiert werden.

Generierung von Testinfrastruktur

Die einfachste und naheliegendste Unterstützung, die MDSD für Komponententests anbietet, ist die Generierung von Testinfrastruktur aus dem fachlichen Modell. Dies kann z.B. (leere) Testmethoden für alle Business-Operationen der Anwendung umfassen sowie deren Komposition zu Testsuiten. Man kann in die Testmethoden auch eine Default-Implementierung hineingenerieren, die den Test fehlschlagen lässt, um eine „echte" Implementierung durch den Entwickler zu erzwingen.

Constraints Checking

Eine sehr mächtige Technik ist das Generieren von Testfällen auf Basis von Constraints im fachlichen Modell[50]. Diese spezifizieren deklarativ Eigenschaften des Systems, sagen aber nichts darüber aus, *wie* diese Eigenschaften realisiert sind.

Typischerweise unterscheidet man bei solchen Constraints zwischen Vorbedingungen (Preconditions) zur Beschreibung der Voraussetzungen für den Aufruf einer Operation, Nachbedingungen (Postconditions) zur

[50] Nicht zu verwechseln mit Modellierungs-Constraints auf der Meta-Ebene (vgl. <Ref MM>), welche „gültige Modelle" im Sinne der DSL definieren.

Beschreibung des Zustandes nach dem Aufruf (Test) und Invarianten (Invariants) zur Beschreibung von Bedingungen, die stets gelten müssen.

Man kann bei entsprechender Modellierung also einen Unit-Test generieren, der einen benötigten Grundzustand herstellt (Setup), gültige Testdaten bereitstellt (Precondition), die zu testende Operation aufruft und hinterher die Nachbedingung überprüft. Dabei gibt es folgende Problemstellungen:

- Wie kann das Setup hergestellt werden?
- Wie sollten die Testdaten beschaffen sein?
- Welche „Umgebung“ muss vorhanden sein, damit die Testausführung funktioniert?

Da dies nicht immer einfach zu definieren und zu implementieren ist, macht es oft Sinn, die Constraints nicht durch spezielle Unit-Tests zu überprüfen, sondern einfach den Constraint-Überprüfungscode in das System mit hineinzugenerieren und im Rahmen von Integrationstests die Constraints mit zu überprüfen.

In Abb. 105 ein Beispiel:

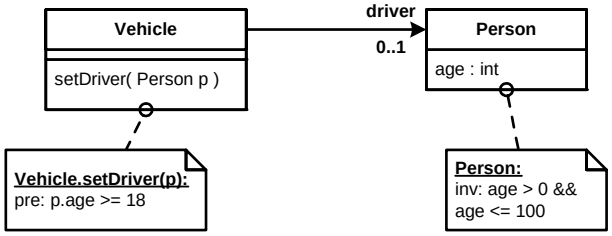


***Abb. 105** Beispiel eines Klassendiagramms mit Constraints*

Aus diesen Constraints kann durchaus Code generiert werden:

```
class Vehicle {
  ...
  public void setDriver( Person p ) {
    if (driver.getAge() < 18 )
           throw new ConstraintViolated();
    // … implementation code..
  }
}
```

Dabei gilt es zwei Dinge zu erläutern. Zum einen stellt sich die Frage, wie man aus OCL-Constraints Code generiert (es gibt entsprechende Tools, z.B. [EMPO], [DRES]). Es stellt sich auch die Frage, wo und wie die Constraints im Modell eingebaut werden können, und ob sie dann (zum Beispiel via XMI) verfügbar sind. Ein „Trick“, der mit allen UML-Tools funktioniert, ist das „Missbrauchen“ der Dokumentation von

Modellelementen. In diese lässt sich beispielsweise für die *setDriver()*-Operation folgender Text eintragen:

```
Diese Operation dient dazu, einem Auto einen Fahrer
zuzuordnen. Fahrer sind immer Personen. Ohne gültigen
Fahrer kann das Auto nicht losfahren.
<Constraints>
<pre id=”driverAge”>
  <expr>driver.getAge() >= 18</expr>
  <error>Alter des Fahrers muss >= 18 sein.<error>
</pre>
</Constraints>
```

Nun kann in der MDSD-Transformation einfach das Textfeld nach dem <*Constraints*>-Tag durchsucht werden. Das XML kann geparst werden, und damit stehen die Constraints zur weiteren Verarbeitung bereit. Hier gehen wir der Einfachheit halber auch davon aus, dass der Text innerhalb <*expr*>…</*expr*> ein gültiger Bool'scher Ausdruck *in der Zielsprache* ist. Dies schränkt zwar die Portabilität ein, aber als pragmatischer Ansatz ist es ein gangbarer Weg.

Interessant ist noch die Frage, wie man dieses Vorgehen realisiert, wenn die Implementierung der eigentlichen Methode manuell erfolgt. Wie kann sichergestellt werden, dass der Constraint trotzdem *immer* überprüft wird? Hier bietet sich die Verwendung das Template Method Patterns an [GHJ+94]. Generiert wird eine abstrakte Klasse *VehicleBase*, die die Operation *setDriver()* folgendermaßen implementiert:

```
abstract class VehicleBase {
  ...
  public void setDriver( Person p ) {
    // generated from precondition constraint
    if (driver.getAge() < 18 )
        throw new ConstraintViolated();
    setDriverImpl( Person p );
    // generated from postcondition constraint
    if (driver == null) throw new ConstraintViolated();
  }

  protected abstract void setDriverImpl( Person p );

}
```

Diese Operation ruft eine abstrakte Operation *setDriverImpl()* auf. Der Entwickler schreibt nun die Klasse *Person* selbst und implementiert hier nur noch die abstrakten Operationen, die die Oberklasse definiert hat:

```
class Vehicle extends VehicleBase{

  protected abstract void setDriverImpl( Person p ) {
    driver = p;
  }
```

```
}
```

Man kann diese Constraints beliebig komplex definieren, beispielsweise auch mittels Protokollzustandsautomaten. Einem Interface kann ein solcher zugeordnet werden. Es kann daraus Code generiert werden, der diese damit ausgedrückten Constraints überprüft.

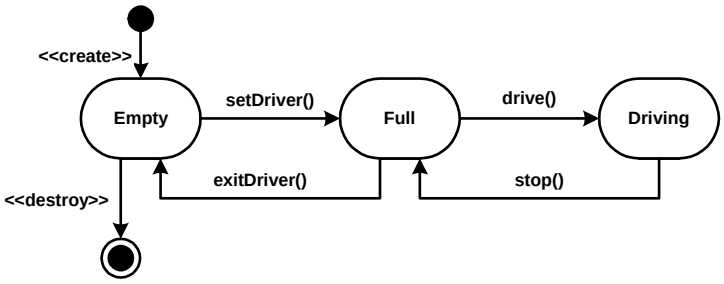


***Abb. 106** Beispiel für einen Protokollzustandsautomat*

Die Protokollzustandsmaschine in Abb. 106 bringt zum Beispiel zum Ausdruck, dass man erst losfahren kann (*drive()*), wenn ein Fahrer im Auto sitzt. Der generierte Code kann diese Zustände automatisch verfolgen. Wird die drive()-Operation aufgerufen, wenn das Fahrzeug leer ist, so kann eine ConstraintViolation-Exception geworfen werden.

Generierung von Extremwert-Tests

Für den Fall, dass bei einer Operation keine Constraints angegeben sind, können natürlich keine fachlich sinnvollen Tests automatisch erzeugt werden, sie müssen von Hand implementiert werden. Allerdings darf man davon ausgehen, dass beliebige Eingabeparameter verwendet werden dürfen, ohne dass es zu Laufzeitfehlern kommt. Man kann also entsprechende Testfälle und auch Testdaten generieren. Vor allem bei Operationen mit primitiven Typen ist dies ein praktikabler Ansatz. Beliebte Werte für *int*-Parameter sind z.B. 0, -1, 87684335, für Objektparameter bietet sich *null* an. Dies testet auf Extremwerte bzw. auf das Auftreten von *Nullpointer*-Fehlern.

Mock-Objekte

Mock[51]-Objekte sind ein nützliches Hilfsmittel beim Testen von Software. Sie dienen dabei der Entkopplung von Testcode und Applikations-Infrastruktur, die aus Testsicht irrelevant ist. Dazu werden

[51] Mock = Fälschung/Simulation

entsprechende Interfaces durch Mock-Objekte implementiert, die sich auch ohne komplexe Infrastruktur so verhalten, wie es für ein Testszenario nötig ist. Angenommen wir wollen überprüfen, ob ein Client mit Exceptions, die von einem entfernten Objekt geworfen werden, richtig umgeht:

```
public Object doSomething( RemoteObject o ) throws SomeEx
{
  try {
    return o.anOperation();
  } catch ( RemoteException ex ) {
    log( ex );
    throw new SomeEx();
  }
}
```

Wie können wir nun sicherstellen, dass das *RemoteObject* zu Testzwecken auch wirklich eine Exception wirft, wenn der Client die Operation *anOperation()* aufruft? Wir können das Interface *RemoteObject* natürlich manuell implementieren und eine entsprechende Exception werfen – für andere Testszenarien implementieren wir dasselbe immer wieder.

Neben anderen Ansätzen wie bspw. *EasyMock* [EASY] kann hier auch Codegenerierung helfen. Zum einen kann man sehr einfach Implementierungsklassen generieren, die ein bestimmtes Verhalten aufweisen. Als Spezifikation könnte folgendes XML dienen:

```
<MockConfig type="examplepackage.RemoteObject"
            name="RemoteObjectTestImpl">
  <operation names="anotherOperation, thirdOperation">
    <return value="null" occurrence="1"/>
  </operation>
  <operation names="anOperation">
    <throw type="RemoteException" occurrence="all"/>
  </operation>
</MockConfig>
```

Hier wird definiert, dass eine Klasse *RemoteObjectTestImpl* generiert werden soll, wobei das Interface *examplepackage.RemoteObject* implementiert werden soll. Die Operationen *anotherOperation* und *thirdOperation* werden so implementiert, dass einfach *null* zurückgegeben wird; Für *anOperation* wird definiert, dass eine *RemoteException* geworfen werden soll. Diese Klasse kann nun in einem Test verwendet werden.

Man kann solche Mocks auch implizit definieren, was den Aufwand nochmals weiter verringert (siehe Abb. 107):

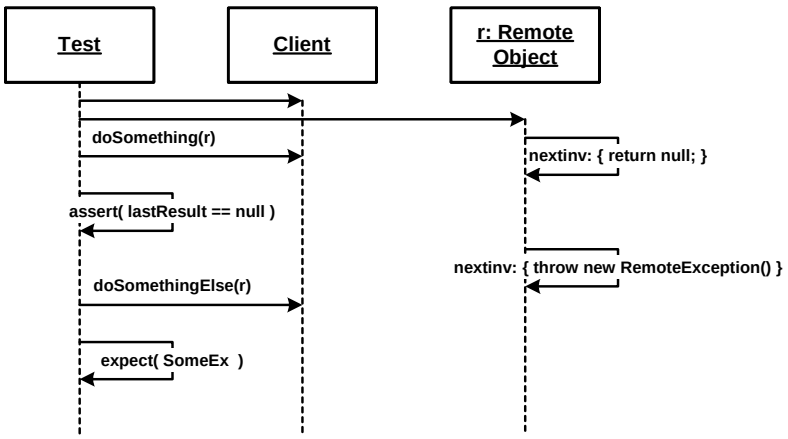


***Abb. 107** Beispiel für die Verwendung von Mock-Objekten*

Hier wird ein *RemoteObject* erzeugt und definiert, dass beim nächsten Methodenaufruf *null* zurückgegeben werden soll (der Implementierungscode für die Methode steht einfach in den geschweiften Klammern). Wir überprüfen, ob die *doSomething()*-Methode dies auch korrekt zurückgibt. Dann wird auf dem *RemoteObject* definiert, dass es beim nächsten Aufruf eine *RemoteException* werfen soll. Wir rufen dann beim Client *doSomethingElse()* auf und erwarten, dass eine *SomeEx*-Exception geworfen wird.

Um dieses Szenario zu ermöglichen, kann der Codegenerator einfach eine passende Implementierung des *RemoteObject*-Interfaces generieren, die sich entsprechend verhält.

14.2.2. Akzeptanztests

Diese Tests *müssen* unabhängig vom Anwendungsmodell definiert werden. Es wäre sinnlos, diese Tests aus demselben Modell zu generieren, aus denen die Anwendung generiert wird, da sie dann *niemals* fehlschlagen würden (sofern die Domänenarchitektur korrekt ist). Dies führt auf das Prinzip der Zweikanaligkeit: Man braucht einen zweiten, unabhängigen „Kanal“, um Tests zu erhalten, die unabhängig von der ursprünglichen Spezifikation sind. Auch der zweite Kanal *kann* allerdings durch MDSD unterstützt werden (siehe Abb. 108).

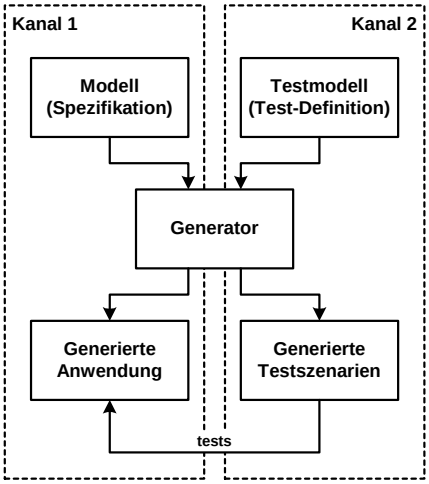


Abb. 108 *Beispiel für das Prinzip Zweikanaligkeit*

Man kann also Testszenarien ebenso modellieren wie die Anwendung selbst, allerdings in einem separaten Akzeptanztest-Modell. Constraints auf die Operationen der Testskripte sorgen für die „asserts“, die in Tests benötigt werden.

Bei Verwendung einer geeigneten DSL kann ein Fachexperte die Testfälle *deutlich* besser verstehen und produktiver bei der Definition mitarbeiten, als wenn man die Integrationstests „programmiert“. Abb. 109 ist ein Beispiel unter Verwendung eines Sequenzdiagramms:

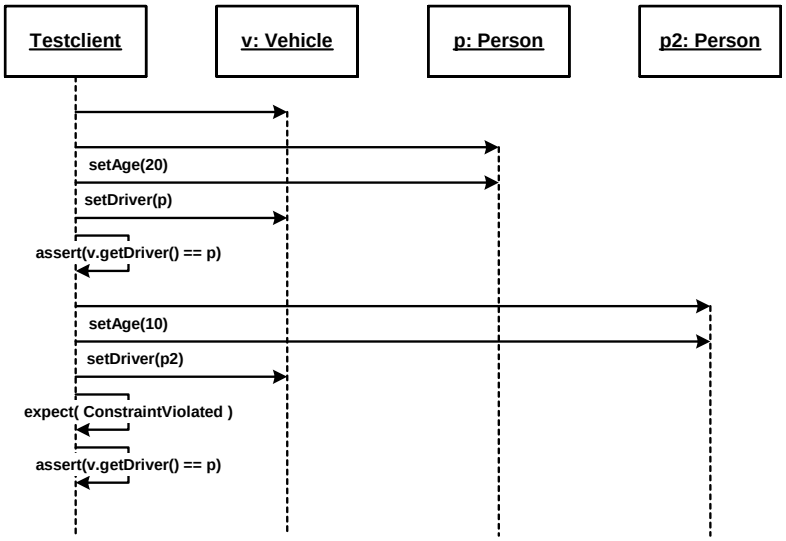


Abb. 109 *Beispiel für die Definition eines Testszenarios mittels eines UML-Sequenzdiagramms*

Der Test-Client erzeugt sich zunächst ein neues *Vehicle* und eine neue *Person*. Er setzt dann das Alter der Person auf 20 Jahre und definiert die Person als Fahrer des Autos. Es folgt dann eine *Assertion*, die sicherstellt, dass der Fahrer des Autos auch wirklich der gerade gesetzte ist. Dann wird eine neue Person erzeugt; diese hat das Alter 10. Wenn diese nun als Fahrer gesetzt wird, muss eine *ConstraintViolated*-Exception geworfen werden, die mit *expect()* ausgewertet wird. Damit wird sichergestellt, dass der Implementierungscode den betreffenden Constraint tatsächlich überprüft. Zu guter Letzt stellen wir sicher, dass der Fahrer immer noch der ursprüngliche ist.

Natürlich würde man – wie weiter oben erwähnt – die Testdaten vom Testcode trennen. Aus Gründen der Einfachheit haben wir das hier nicht getan.

14.2.3. Lasttests

Lasttests arbeiten prinzipiell nach dem folgenden Schema: Eine Menge von Clients werden simuliert, die üblicherweise auf einer Reihe von Rechnern laufen. Die Clients führen Testskripte aus. Das Zeitverhalten der Clients wird in Abhängigkeit der Last (also der Anzahl der Clients und deren „Aufrufintensität“) gemessen. Außerdem wird der Ressourcenverbrauch des Servers protokolliert.

Das Setup der Umgebung (bspw. Server, Netzwerk, Datenbanken) muss manuell erfolgen. Zu definieren sind weiterhin

- das oder die Testskript(e),
- die Anzahl von Clients sowie
- deren interne Parallelität, also die Anzahl von Prozessen und Threads.

Die letzteren zwei Aspekte werden meist von speziellen Werkzeugen erledigt, die sich auch um die Messung des Zeitverhaltens und das Monitoring der Anwendung(en) kümmern.

Die Testskripte eignen sich zur Generierung. Ein Sequenzdiagramm oder ein Statechart eignen sich sehr gut als Basis, wie Abb. 110 zeigt.

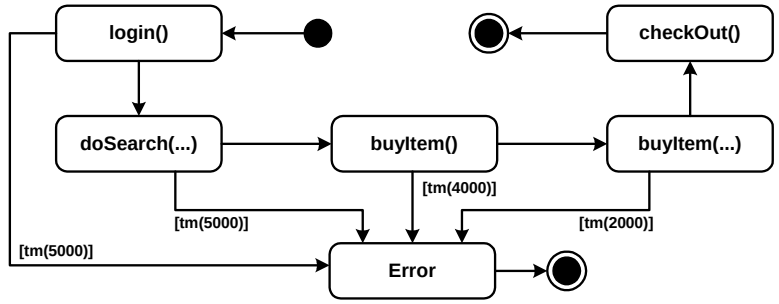


Abb. 110 *Zustandsdiagramm mit Timeouts zur Lasttest-Beschreibung*

14.2.4. Nicht-funktionale Tests

Das Testen nicht-funktionaler Anforderungen wie Zuverlässigkeit, transaktionale Konsistenz oder Deadlock-Freiheit lässt sich nicht in einer einfachen Umgebung (in JUnit) durchführen. Realistische Szenarien sind nötig. Ausfälle von Netzwerken oder Datenbanken müssen simuliert (oder tatsächlich herbeigeführt) werden. Auch Security-Tests können nur von Hand durchgeführt werden. Modellgetriebene Entwicklung bietet hier keine spezifische Hilfestellung.

14.2.5. Modellvalidierung

Die *Modellvalidierung* ist eine Testart, die erst durch den MDSD-Ansatz entsteht und völlig neue Möglichkeiten des Testens bietet. Es sind drei verschiedene Unterarten zu unterscheiden: *Akzeptanztests auf Modellebene* zur Validierung der Modellsemantik, *Wohlgeformtheits-Tests* zur Überprüfung der Einhaltung formaler Modellierungsregeln (DSL-Constraints) und *Simulation von Modellen*.

Akzeptanztests auf Modellebene

Insbesondere bei einer fachlichen Ausrichtung der MDSD-Domäne und ihrer DSL bieten MDSD-Modelle das Potenzial, in direkter Kommunikation mit Kunden oder Fachexperten validiert zu werden (sofern diese die Modelle nicht bereits selbst erstellen). Dieses Vorgehen kann zusätzliche Sicherheit gewährleisten, zumal aus dem Modell Code oder Konfiguration erzeugt wird, deren fachliche Intention damit bereits im Vorfeld auf einer abstrakteren Ebene geprüft wurde.

Wohlgeformtheits-Tests

Die Bedeutung von Modellierungsregeln im Kontext von MDSD haben wir bereits am Beispiel der ersten Fallstudie (vgl. <Ref CS1>) kennen gelernt. Auch haben wir gesehen, dass sie die Rolle der statischen Semantik der DSL spielen (vgl. <Ref Begr>) und gezeigt, wie sie im Rahmen der Metamodellierung (vgl. <Ref MM>) definiert werden.

Aus der Test-Perspektive sind solche Modellierungsregeln Invarianten, die für alle möglichen Modelle einer DSL gelten. Sie *müssen* überprüft werden, bevor die eigentlichen MDSD-Transformationen ansetzen, andernfalls ist das Transformationsergebnis undefiniert[52]. Der Test auf Wohlgeformtheit kann durch das Modellierungswerkzeug erfolgen (sofern es dazu in der Lage ist) oder durch ein nachgelagertes MDSD-Werkzeug (vgl. <Ref CSI::Archi> und <Ref Tool::Valid>). Aus technischer Sicht sind solche Tests nichts anderes, als die Prüfung der statischen Semantik einer klassischen Programmiersprache durch den jeweiligen Compiler. Aus Sicht des Anwendungsentwicklers allerdings sind sie *weit* effektiver: Der Compiler einer klassischen Programmiersprache „weiß“ nichts von der MDSD-Domäne – daher kann er nur Fehlermeldungen bezüglich des Lösungsraumes (d.h. der Programmiersprache) erzeugen. Modellierungsregeln hingegen sind spezifisch für die MDSD-Domäne und können daher Fehlermeldungen mit den Begriffen des Problemraumes formulieren.

Eine gewisse Klasse von Fehlern, die bei konventioneller Entwicklung typischerweise noch nicht einmal durch den Compiler gefunden werden, sondern erst zur Laufzeit des Systems auftreten, lässt sich also durch den MDSD-Ansatz schon sehr frühzeitig (während der Modellierung oder vor der Codegenerierung) detektieren.

Dazu ein einfaches Beispiel: Das Metamodell in Abb. 111 definiert, dass ein Konfigurationsparameter (eine spezielle Art von Attribut) immer den Typ *String* haben muss.

---

[52] Was leider nicht zwingend bedeutet, dass sich das Transformationsergebnis nicht kompilieren ließe.

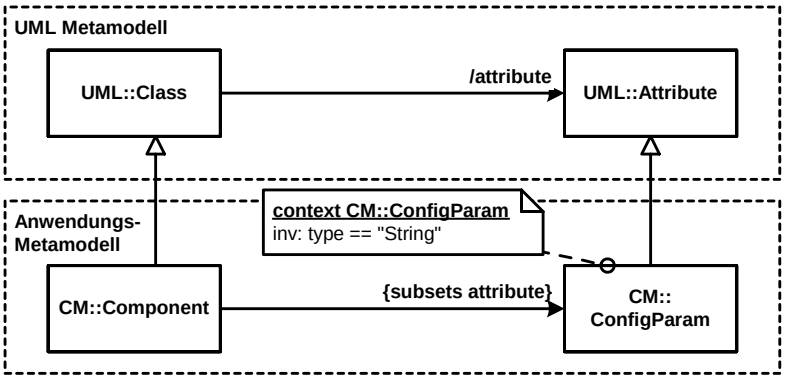


***Abb. 111** Beispiel für Constraints im Metamodell*

Bei allen Modellen, die mit dieser DSL formuliert werden, muss also das angegebene Constraint „wahr“ sein. Wenn ein Konfigurationsparameter kein String ist, kann eine entsprechende, sinnvolle Fehlermeldung ausgegeben werden („ConfigParam must be of type String“), anstatt dass erst zur Laufzeit eine Exception auftritt, weil versucht wurde, einem *String* einen *Integer* zuzuweisen. Weitere, komplexere Beispiele für Modellierungs-Constraints finden sich in den oben referenzierten Kapiteln.

Simulation des Modells

Falls in einem Modell die dynamischen Aspekte eines Systems vollständig beschrieben werden, so können diese durch Simulation des Modells (d.h. Ausführung des Modells in einem Testbett) validiert werden. Bei eingebetteten Systemen ist dieser Ansatz ziemlich verbreitet, allerdings nur für recht spezielle Arten der Verhaltensdefinition, den Zustandsautomaten. Auch die UML 2.0 und die Action Semantics stellen einen Schritt in diese Richtung dar (siehe <Ref MDA>). In der allgemeinen Praxis ist der Ansatz der Modellsimulation allerdings meist zu aufwändig oder nicht möglich, weil nicht das komplette Systemverhalten durch das Modell spezifiziert ist.

14.3. Testen der Domänenarchitektur

Auch die Domänenarchitektur ist Software und muss daher in geeigneter Weise getestet werden. Dieses Problem lässt sich glücklicherweise recht elegant in einzelne Aspekte zerlegen und auf bekannte Problemstellungen zurückführen.

14.3.1. Test der Referenzimplementierung und der MDSD-Plattform

Die Referenzimplementierung spielt – zumindest während des Bootstrappings der Domänenarchitektur – die zentrale Rolle. Da man hier typischerweise ein kleines Expertenteam einsetzt, sind agile, testgetriebene Methoden geradezu prädestiniert, um valide Ergebnisse zu erzielen. Das Gleiche gilt für die MDSD-Plattform.

Hier können alle bereits erläuterten Testarten eingesetzt werden – zunächst natürlich ohne modellgetriebene, generative Unterstützung. Allerdings kann die Referenzimplementierung sehr wohl bereits eine Partition für Testware enthalten, um eben hier Generierungspotenzial zu entdecken.

14.3.2. Akzeptanztest der DSL

Sehr wichtig ist auch die Validierung der Modellierungssprache (DSL). Dies geschieht durch Anwendung in Form des Referenzmodelles (siehe <Ref Proz::ArchiStrg>). Letzteres ist also insbesondere ein (Akzeptanz-)Test der DSL im Hinblick auf die Handhabbarkeit und Ergonomie der DSL. Da die Referenzimplementierung und das Referenzmodell gerade den Anspruch haben, die Konstrukte der DSL möglichst gut und gleichzeitig minimalistisch zu überdecken, sollte das Referenzmodell schon ein recht aussagekräftiger Test sein. Nach dem Bootstrapping ist dann das Anwendungsmodell aus dem Entwicklungsstrang der nächste DSL-Test.

14.3.3. Test der MDSD-Transformationen

Im Architektur-Entwicklungsstrang erfolgt die Formalisierung der Aspekte der Referenzimplementierung und das „Gießen“ in eine computerverarbeitbare Form – meist Generator-Templates oder ähnliche Transformationsvorschriften, die am Metamodell der DSL festgemacht sind. In aller Regel stützen sich die Transformationsvorschriften auf ein generisches MDSD-Werkzeug, dessen Korrektheit wir voraussetzen[53].

---

[53] Etwas anders sieht es aus, wenn der Generator z.B. für sicherheitskritische Systeme zertifiziert werden muss. Dies wollen wir hier jedoch nicht betrachten.

Bleibt also der Test der Domänen- und Plattform-spezifischen Transformationen selbst – wie kann dies sinnvoll geschehen?

Ein besonders nahe liegender Test fällt beim Bootstrapping der Domänenarchitektur quasi ab: Da die Transformationsvorschriften oder -Regeln aus Referenzimplementierung und Referenzmodell abgeleitet wurden, sollten sie genau denjenigen Teil der Referenzimplementierung reproduzieren können, der durch die DSL abgedeckt wird. Mit anderen Worten: Wendet man die frisch gewonnenen Transformationen auf das Referenzmodell an, erhält man je nach Tragweite der Domänenarchitektur entweder die vollständige Referenzimplementierung oder „nur“ ihren Implementierungsrahmen. Ergänzt man diesen Rahmen ggf. durch die noch fehlenden Codefragmente aus der ursprünglichen Referenzimplementierung, muss sich wiederum eine komplette und lauffähige Anwendung ergeben, welche allen Tests genügt, die während der Konstruktion der Referenzimplementierung erstellt wurden. Ganz nebenbei hat man jetzt eine generative Referenzimplementierung, und das Bootstrapping der Domänenarchitektur ist abgeschlossen.

Im weiteren Projektverlauf wird dieser initiale Domänenarchitektur-Test dann auf die eigentliche Anwendung ausgedehnt. Die Architektur wird also implizit durch die verwendenden Anwendungen (d.h. deren Tests) validiert. Dies ist sehr effektiv und in der Regel praktisch vollkommen ausreichend.

Wie sähe ein expliziter Test der Transformationsvorschriften aus? Nun, ein Generator ist ein Metaprogramm, also ein Programm, das aus Modellen Programme erzeugt. Eingaben sind also Modelle, Ausgaben Programme bzw. Programmteile (Sourcecode-Fragmente). Ein expliziter Test z.B. einer einzelnen Transformationsregel würde also ein relevantes Modellstück als Test-Setup verwenden, die Regel darauf anwenden und das Ergebnis schließlich mit einer Spezifikation, also einem entsprechenden Soll-Generat (Sourcecode-Fragment), vergleichen. Abb. 112 zeigt dieses Prinzip.

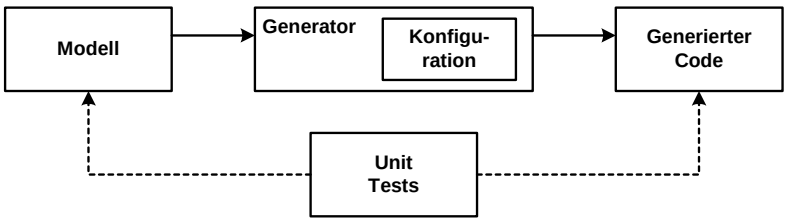


***Abb. 112** Expliziter Transformationstest*

Bei entsprechend modular aufgebauten Generatoren ist dieses Vorgehen und der Aufbau entsprechender Testsuiten durchaus möglich (vgl. <Ref Trans::ModTrans>). Doch betrachten wir die Konsequenzen:

Ein (Blackbox-)Test versorgt den Testling mit spezifischen Stimuli (Setup und Parameter) und vergleicht den Output oder die Seiteneffekte der Funktionsausführung des Testlings mit einer Spezifikation. Dabei wird offensichtlich von der Implementierung des Testlings abstrahiert (darin besteht ja gerade der Sinn). Wie verhält es sich mit der Abstraktion nun im Falle der oben beschriebenen *expliziten* Transformationstests? Ist die Festlegung auf eine ganz spezifische und textuell festgelegte Form des Generats hilfreich, oder ist nicht vielmehr die *Semantik* des erzeugten Codes relevant? Stellen Sie sich eine Testsuite mit expliziten Transformationstests für die Domänenarchitektur einmal in der Praxis vor: Was passiert, wenn die Architekten merken, dass eine andere Implementierung für einen generativen Softwarearchitekturaspekt besser geeignet ist als die vorhandene? Die Testsuite würde die Änderung nicht zulassen – obwohl sie möglicherweise aus Sicht der nutzenden Anwendungen vollkommen transparent wäre! Umgekehrt bedeutet dies, dass explizite Transformationstests eine kontraproduktive Überspezifikation darstellen. Transformationstests sollten also *nicht* auf der Meta-Ebene (Test des Generators) stattfinden, sondern auf der konkreten Ebene (Test des Generats).

Wie steht es nun mit der Modularisierung der Transformationstests? Referenzimplementierung (inkl. Ihrer konkreten Testsuite) und Referenzmodell stellen in diesem Sinne sozusagen *einen* recht mächtigen Testfall für die generative Softwarearchitektur dar, jede nutzende Anwendung einen weiteren. Je nach Kombinatorik der Modellkonstrukte kann es aber durchaus sinnvoll sein, wiederum eine Testsuite aus mehreren kleinen Testfällen für die Domänenarchitektur zu erstellen. Dies kann folgendermaßen aussehen: Ein Transformations-Testcase konzentriert sich auf ein spezifisches Konstrukt der DSL (z.B. Assoziationen zwischen persistenten Entitäten), d.h. das Setup besteht aus einem diesbezüglich minimalen Modell und ggf. benötigter vorimplementierter Fachlogik. Der Testlauf beinhaltet dann die Anwendung der Transformationen, wobei aus dem Modell ein Generat entsteht, welches ggf. die vorgegebene Fachlogik einbettet. Das lauffähige Generat wird dann durch Testcases auf der konkreten Ebene validiert. Eine Testsuite aus solchen Transformationstests, welche die DSL überdeckt, validiert den generativen Anteil der Domänenarchitektur.

15. Versionierung

Richtiges Versions- und Konfigurationsmanagement ist bei Softwareentwicklungsprojekten immer eine wichtige Sache. Wir möchten in diesem Kapitel allerdings nur auf die MDSD-spezifischen Besonderheiten zu diesem Thema eingehen.

15.1. Was wird versioniert?

Innerhalb eines MDSD-Projektes sind folgende Dinge zu verwalten bzw. zu versionieren:

- Das (generische) Generierungswerkzeug. Derzeit sind diese Tools oft selbst noch im Fluss – insofern macht es Sinn, diese auch zu versionieren.
- Die Generatorkonfiguration (also der generative Anteil der Domänenarchitektur). Dazu gehören z.B. DSL/Profildefinition, Metamodell, Templates, Transformationen etc.
- Der nicht-generative Teil der Domänenarchitektur, also die MDSD-Plattform.
- Die Anwendung selbst, also Modelle, Spezifikation und (meist) manuell entwickelter Code.

Zu beachten ist zunächst, dass der generierte Code idealerweise nicht versioniert wird. Dieser ist ja jederzeit aus dem Modell reproduzierbar und somit keine echte Programm-*Quelle*. Dieses Postulat ist natürlich nur dann einfach umzusetzen, wenn der manuell entwickelte und der generierte Code strukturell und auf Dateiebene vollständig voneinander getrennt sind. Unter anderem deshalb legen wir so viel Wert auf diese Trennung (vgl. <Ref Trans::BP-Vorg::SepCode>). In der Praxis ist dies allerdings nicht immer zu 100% möglich oder sinnvoll, so dass wir für

diese Fälle ein etwas komplexeres Verfahren benötigen. Wir gehen darauf später in diesem Kapitel noch ausführlicher ein.

Des Weiteren ist zu beachten, dass das Ziel modellgetriebener Entwicklung ja auch darin besteht, mehrere Anwendungen basierend auf derselben Domänenarchitektur zu entwickeln. Es ist also essenziell, die Plattform sowie die Generatorkonfigurationen komplett von der Anwendung zu trennen.

15.2. Projekte und Abhängigkeiten

Abb. 113 zeigt einen Überblick über eine bewährte Projektstruktur. Dabei zeigen die gestrichelten Pfeile die Abhängigkeiten. Die zentrale Leitlinie ist, dass die Domänenarchitektur frei gehalten werden muss von anwendungsspezifischen Artefakten.

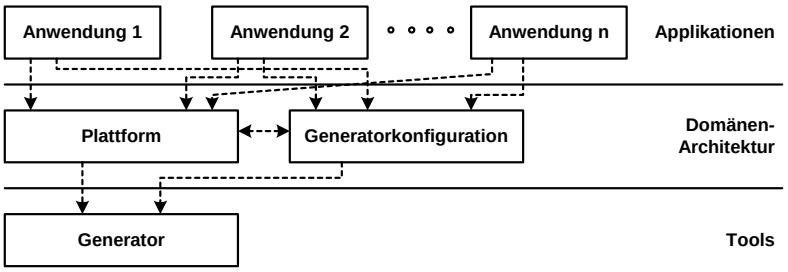


Abb. 113*Projekte und Ihre Abhängigkeiten*

Im Versionsmanagementsystem werden nun alle die obigen Projekte verwaltet. Die Tools und die Domänenarchitektur werden komplett eingecheckt. Die Anwendungen nicht: Der generierte Code, der als Ergebnis der Ausführung des Generators entsteht, sollte idealerweise nicht eingecheckt werden.

Wichtig ist es nun, die Abhängigkeiten der verschiedenen Projekte inklusive der Versionen zu verwalten. Es muss beim abhängigen Projekt verwaltet werden, *von welcher Version* der Domänenarchitektur das Projekt abhängt. Gibt es also z.B. eine neue Version der Plattform, die Änderungen im Anwendungsprojekt erfordert, und wird die Anwendung nicht angepasst, so muss eben im Rahmen der Versionierung festgehalten werden, dass die Anwendung sich noch auf die letzte Version bezieht. Zur Verdeutlichung ist hier die Framework-Metapher sehr hilfreich: Stellen Sie sich die Domänenarchitektur bezüglich der Versionierung und des Release-Managements als Framework vor, das von mehreren Anwendungen genutzt wird. Es sind die gleichen Abhängigkeiten und

damit auch die gleichen Methoden, die bei der Evolution einer Domänenarchitektur zum Tragen kommen.

Diese Sichtweise kann insbesondere auch kaskadieren. Zum Beispiel macht es Sinn, die MDSD-Plattform und die Generatorkonfiguration einer Domänenarchitektur getrennt zu versionieren, insbesondere wenn die Plattform in anderen Domänenarchitekturen wiederverwendet wird. Auf der nächsten Stufe wird man wiederverwendbare Transformationsmodule (Cartridges) getrennt versionieren wollen, und eine mächtige MDSD-Plattform zerfällt möglicherweise in entkoppelte Frameworks. In einem noch weiter fortgeschrittenen Szenario ist eine fachliche MDSD-Plattform möglicherweise mit Hilfe einer architekturzentrierten Domänenarchitektur erstellt worden. Diese Abhängigkeiten müssen erkannt und mittels der Framework-Metapher auf die Versionierung und das Abhängigkeitsmanagement übertragen werden.

15.3. Struktur von Anwendungsprojekten

Abb. 114 zeigt, wie ein Anwendungsprojekt (auf oberster Ebene) strukturiert werden kann und wie die Werkzeuge darauf arbeiten.

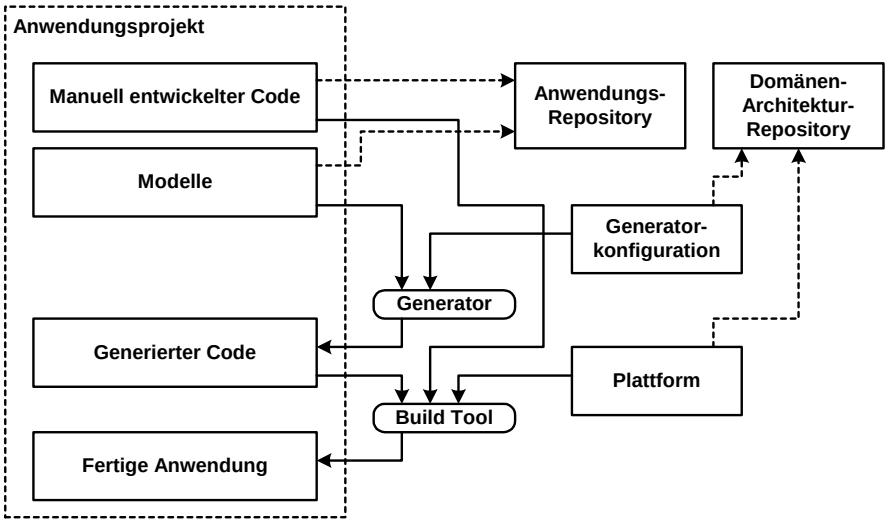


Abb. 114*Projekte, Artefakte und Repositories*

Im Anwendungs-Repository befinden sich also die Modelle der betreffenden Anwendung sowie der manuell entwickelte Code. Der Generator erzeugt den generierten Code (inkl. Konfigurations-Files etc.). unter Zuhilfenahme der Generatorkonfiguration. Dieser befindet sich im

Domänenarchitektur-Repository. Im nächsten Schritt wird nun durch das (meist auch generierte) Build-Skript die Anwendung generiert. Dieser Schritt verwendet den manuell entwickelten Code der Anwendung sowie die Plattform (auch diese aus dem Domänenarchitektur-Repository).

15.4. Versionsmanagement und Build-Prozess bei gemischten Dateien

Am Anfang des Kapitels haben wir bereits angesprochen, dass eine vollständige Trennung von generiertem und nicht-generiertem Code nicht immer möglich oder sinnvoll ist. Beispiele hierfür sind:

- Custom Code in J2EE-Deployment-Deskriptoren.
- Custom Code in JSP-Dateien.
- Custom Code in Property- oder andere Konfigurations-Dateien.

Ganz allgemein: Stellen, an denen die Zielsprache keine ausreichenden Delegationsmechanismen zur Verfügung stellt oder durch die Delegation ein zu stark „zerfasertes“ Design entsteht.

In diesen Fällen wird man in der Regel mit Mechanismen arbeiten, wie wir sie bereits in der Fallstudie (vgl. <Ref CSI>) kennen gelernt haben: Geschützte Bereiche (Protected Regions), die man in den Generator-Templates definieren kann. Im Generat entstehen dann Markierungen, die der Generator zwecks iterativer Generierung nutzt, um den darin enthaltenen manuell programmierten Code aufzufinden und zu erhalten. Vor dem Compiler oder Interpreter der Zielsprache werden diese Metakommentare jedoch verborgen, indem man sie syntaktisch als Kommentare der Zielsprache kennzeichnet.

Die Verwendung solcher Protected Regions führt also offensichtlich zu Dateien, in denen generierter und nicht-generierter Code vermischt ist. Das Problem dabei ist nun, dass diese Dateien üblicherweise nur als Ganzes versioniert werden können. Es wird also redundanter Code eingecheckt, denn das Generat (ohne die Inhalte der Protected Regions) ist wie gesagt ja keine Quelle – die Quelle wäre das (Teil-)Modell, aus dem dieser Code generiert wurde.

Durch diese Redundanz können bei Entwicklung im Team wiederum Inkonsistenzen entstehen, die mit zunehmender Teamgröße unangenehmer werden: Nehmen wir an, Entwickler A ändert etwas im Modell oder in der Architektur, während Entwickler B gerade Fachlogik programmiert, die sich in einer Datei befindet, deren generierter Anteil von der Änderung durch A betroffen ist. Durch diese Situation kann es

dann zu einem Aktualitätsproblem bzw. einem Konflikt beim Einchecken von A bzw. B kommen. Der Grund dafür ist wie gesagt die Redundanz im Repository. Ziel muss es also sein, diese Redundanz zu vermeiden und z.B. die Inhalte der Protected Regions isoliert zu verwalten. Der verwendete Generator muss dies allerdings zulassen. Auf der anderen Seite ist für den Anwendungsentwickler aber eine isolierte Protected Region in der Regel wenig hilfreich, weil er den Kontext des Generats benötigt, um sich zu orientieren und den Edit-Compile-Run-Zyklus durchzuführen. Es wird also ein Verfahren benötigt, das Redundanz im Repository vermeidet *und* dem Entwickler seine gewohnte, dateiorientierte Sicht bietet. Dies kann z.B. durch zwei automatisch synchronisierte Repositories abgebildet werden: Im *Master*-Repository sind ausschließlich echte Quellen eingecheckt – d.h. insbesondere isolierte Protected Regions. Die Isolation dieser manuell erstellten Inhalte vom umgebenden Generat erfolgt beim Check-in durch entsprechende Skripts. Aus dem Master-Repository wird nun automatisch (z.B. via CruiseControl [CRUI]) ein Slave-Repository „betankt“, welches Generat plus manuell implementierten Code enthält. Der „Betankungs“-Schritt muss also einen Build-Prozess inklusive Codegenerierung durchführen. Zusätzlich können natürlich auch automatisierte Tests angestoßen werden. Die Entwickler „lesen“ dann aus dem Slave-Repository und „schreiben“ in das Master-Repository. Konflikte werden ausschließlich über das Master-Repository detektiert.

Es sei noch einmal darauf hingewiesen, dass die Trennung von generiertem und nicht-generiertem Code unserer Meinung nach essenziell ist und in jedem Falle verfolgt werden sollte, auch wenn ein solches Versionierungsverfahren Konsistenzprobleme reduzieren kann.

15.5. Modellierung im Team und Versionierung von Teilmodellen

Große Systeme müssen partitioniert werden. Die einzelnen Teil- oder Subsysteme werden mehr oder weniger unabhängig voneinander entwickelt. Interfaces definieren, wie die beiden Systeme miteinander arbeiten werden. Regelmäßige Integrationsschritte führen die Teile zusammen. Dieses Vorgehen ist besonders dann vonnöten, wenn die Teile aus organisatorischen Gründen an verschiedenen Standorten oder von verschiedenen Teams entwickelt werden. Dies hat natürlich vor allem Auswirkungen auf den Entwicklungsprozess und die Kommunikation im Team, ggf. auch auf die Systemarchitektur. Dieser

Abschnitt soll einige Aspekte im Zusammenhang mit Versionierung beleuchten.

15.5.1. Partitionierung vs. Subdomänen

Zunächst ist es wichtig, nochmals auf den Unterschied zwischen Partitionierung und Subdomänenbildung hinzuweisen (siehe Abb. 115):

- (Technische) Subdomänen isolieren verschiedene Aspekte des Gesamtsystems. Jede Subdomäne hat ein eigenes Metamodell und eine DSL. Über Gateway-Metaklassen werden die verschiedenen Metamodelle konzeptionell zusammengeführt. Im Rahmen eines Enterprise-Systems könnte es beispielsweise die Subdomänen Businessprozesse, Persistenz und GUIs geben.
- Partitionierung hingegen beschreibt Teilsystembildung in dem Sinne, dass eine große Menge technisch ähnlicher Anforderungen aus organisatorischen oder komplexitätsbedingten Gründen in verschiedene, durch Interfaces integrierbare Teile zerlegt werden.

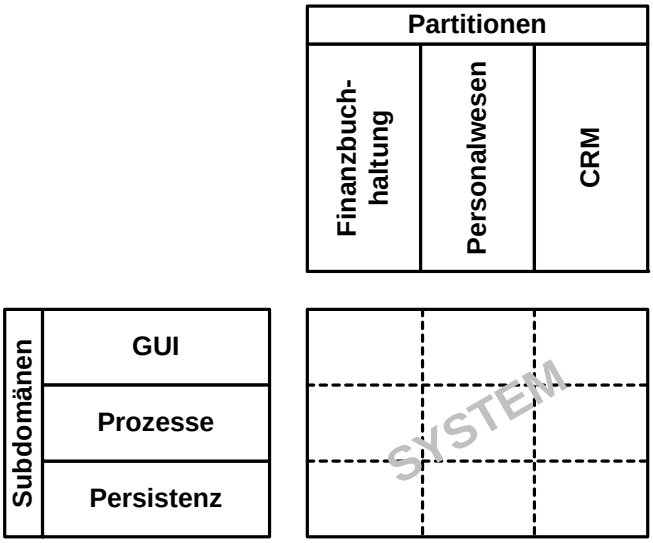


Abb. 115 *Technische Subdomänen vs. Partitionierung*

15.5.2. Verschiedene generative Softwarearchitekturen

Werden in verschiedenen Partitionen eines Projektes verschiedene (Versionen von) generativen Architekturen verwendet, so stellt sich die Frage, ob die generierten Artefakte zusammen funktionieren. Generell

bedeutet dies, dass eine Integration der Anwendung auf Ebene des Generates stattfinden sollte. Ein Beispiel: Angenommen wir arbeiten mit verschiedenen Versionen von generativen Infrastrukturen, die alle Teile einer großen J2EE-Anwendung erstellen. Nachdem ja in einem solchen Szenario alle generierten Artefakte J2EE-konform sein müssen, kann die Integration auf Ebene der „fertigen“ Anwendung erfolgen. Es ist also nicht zwangsweise nötig, dass alle Teile mit derselben Domänenarchitektur arbeiten..

15.5.3.Weiterentwicklung der DSL

Typischerweise entwickelt sich die DSL im Rahmen eines (oder mehrerer) Projekte weiter. Das Wissen und das Verständnis über die Domäne wird umfangreicher und ausgeprägter, die DSL kann also entsprechend erweitert werden. Es muss sichergestellt werden, dass Modelle, welche auf der alten DSL beruhen, weiterverarbeitet werden können.

Die oben erwähnten Mittel der Versionierung sind sicherlich eine Möglichkeit, dies zu erreichen. Eine andere besteht darin, die Generator-Konfiguration so zu modifizieren, dass sie mit verschiedenen Versionen von DSL und Metamodell zurechtkommt. Dies ist vor allem dann nötig, wenn Sie regelmäßig mit verschiedenen Versionen der DSL parallel arbeiten müssen. Dies kommt z.B. vor, wenn am Ende einer Iteration das Anwendungsteam auf die neue Version der Domänenarchitektur umstellt. Hier stellen Sie eine neue – meist mächtigere – Version der Domänenarchitektur zur Verfügung, während die von den Entwicklern verwendeten Modelle noch die alte Version der DSL verwenden. Sie müssen in diesem Fall einen Migrationspfad zur Verfügung stellen, der aus Sicht der fachlichen Entwickler möglichst wenig Änderungen am Modell erzwingt. Die neuen Features der DSL sollten den Entwicklern angeboten, nicht aufgezwungen werden. Alte Features können gegebenenfalls *deprecated* werden. Der Generator kann Warnungen ausgeben, wenn solche alten Features verwendet werden.

Die Realisierung der Fähigkeit, dass die Generator-Konfiguration mit verschiedenen Versionen der DSL zurechtkommt, ist in der Praxis nicht so kompliziert, wie es sich vielleicht zunächst anhört. Die Generator-Konfiguration wird ja von den Entwicklern der Domänenarchitektur implementiert. Sie definieren das Metamodell, die konkrete Syntax sowie den zu generierenden Code. Sie können zum Beispiel in den Modellen

eine Versionsnummer unterbringen[54], die steuert, wie der Generator das Modell interpretiert, oder welchen Code er erzeugt. Sie können auch implizite Regeln implementieren: Wenn ein bestimmtes Attribut im Modell nicht vorhanden ist, soll ein bestimmter Defaultwert verwendet werden. Ein weiteres Beispiel ist die Validierung von Attributen von Entitäten. Angenommen Sie haben ein Metamodell, welches das Konzept einer „Entität“ enthält. Eine Entität hat eine Reihe von Attributen. Das folgende Listing zeigt ein Beispielmodell, notiert in XML:

```
<Entity name=”Person”>
  <Attribute name=”name” type=”String” label=”Name”/>
  <Attribute name=”firstname” type=”String”
             label=”Vorname”/>
</Entity>
```

Typischerweise werden Sie die Attribute auf Korrektheit überprüfen wollen. Dazu kann man verschiedene Constraints angeben. Zunächst implementieren Sie das vielleicht so, dass Sie einfach ein benanntes Constraint angeben:

```
<Attribute name=”name” type=”String” label=”Name”
           constraint=”notNull”/>
```

Dies soll zum Ausdruck bringen, dass das Attribut nicht leer gelassen werden darf. Im Laufe des Projektes stellen Sie nun fest, dass Sie mehr als ein Constraint benötigen, was Sie mit XML-Attributen schlecht zum Ausdruck bringen können. Statt nun alles umzubauen, können Sie die zusätzlichen Constraints als *zusätzliche* XML-Elemente erlauben; das „alte“ Attribut können Sie weiterhin unterstützen:

```
<Attribute name=”name” type=”String” label=”Name”
           constraint=”notNull”>
  <Constraint name=“startsWithLetter“/>
</Attribute>
```

Nun stellen Sie vielleicht fest, dass Sie einen beliebigen Boole’schen Ausdruck benötigen. Auch dieses können Sie leicht unterstützen, indem sie einen Boole’schen Ausdruck (formuliert in der Zielsprache) erlauben:

```
<Attribute name=”age” type=”int” label=”Alter”
```

[54] Eine portable, toolunabhängige Möglichkeit, Versionsinformationen im UML-Modell unterzubringen, besteht darin, Modellelemente mit entsprechenden Tagged Values anzureichern.

```
          constraint="notNull">
  <Constraint>
    1 <= age <= 110
  </Constraint>
</Attribute>
```

Wenn diese Flexibilität nicht genügt, können Sie noch einen Klassennamen der Zielsprache angeben, wobei die betreffende Klasse ein von der Plattform definiertes „Validierer"-Interface implementiert:

```
<Attribute name="age" type="int" label="Alter"
         constraintChecker="person.AgeChecker"/>
```

Wenn Sie Ihre DSL also so weiterentwickeln, dass Sie „alte" Features beibehalten und neue hinzufügen, ist eine Evolution der DSL im Projektalltag leicht möglich. Mit solchen Mechanismen lassen sich erstaunlich viele Fälle abdecken. Natürlich steigert dieses Vorgehen die Komplexität der Generator-Konfiguration. Sie sollten also dafür sorgen, dass im Laufe der Zeit veraltete Features entfernt werden. Durch kontrollierte Verwendung von *deprecated* können Sie alte Features im Laufe des Projektes „aussortieren". Der Generator kann im Übrigen leicht eine Datei erstellen, in der protokolliert wird, welche Features noch verwendet werden. Damit können Sie ein Feature aus der Generator-Konfiguration entfernen, wenn es keiner mehr verwendet.

15.5.4. Partitionierung und Integration

Angenommen verschiedene Teams benötigen die gleichen Schnittstellen, weil beispielsweise ein Team eine Komponente implementiert, die der Code eines anderen Teams verwendet; Abb. 116 zeigt dies.

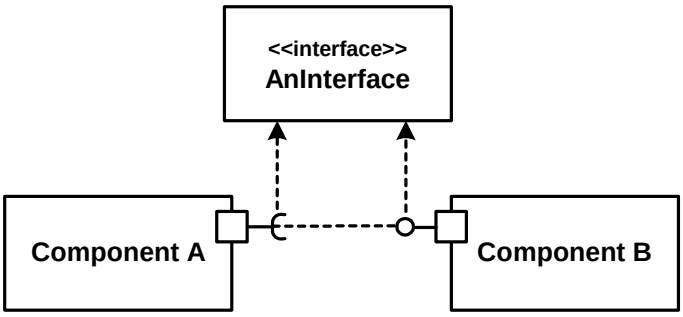


***Abb. 116** Zugriff auf gemeinsame Modellelemente*

Wenn modellgetrieben gearbeitet wird, dann ist es notwendig, dass zumindest das Modell des Interfaces in beiden Modellen vorhanden ist, wie in Abb. 117 gezeigt. Dieser Ansatz ist allerdings nicht ideal, da Sie

Informationen nun in zwei Modellen duplizieren. Es ist schwer, diese beiden Modelle konsistent zu halten. Je nach Werkzeug gibt es allerdings noch andere Möglichkeiten.

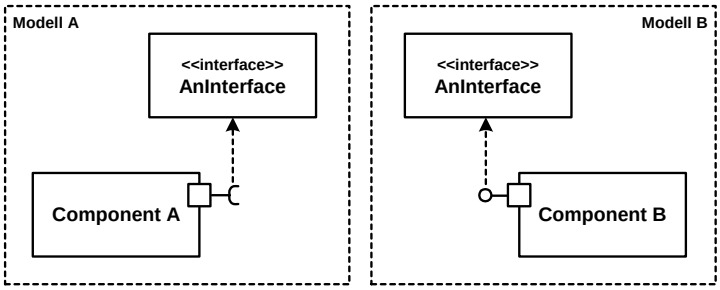


Abb. 117*Duplizierung der „gemeinsam genutzten" Modellelemente*

Integration im Modell

Wenn es das Modellierungswerkzeug unterstützt, lässt sich dies auch so realisieren, dass das Interface tatsächlich nur einmal vorhanden ist und das Interface aus den beiden Komponentenmodellen referenziert wird. Aus Sicht des Generators handelt es sich also um *ein* konsistentes Modell (siehe Abb. 118).

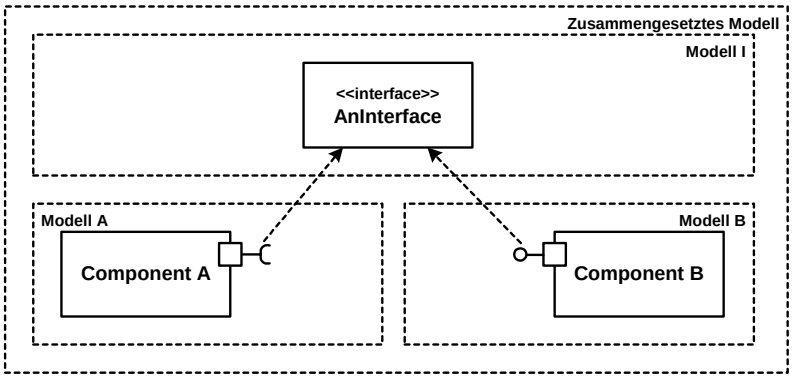


Abb. 118*Gemeinsame Nutzung durch tatsächliche Verwendung des gleichen Modells*

Ob dieser Ansatz möglich ist, hängt wie gesagt vom Modellierungswerkzeug ab. Bei den UML-Tools sind Repository-basierte Werkzeuge, die eine verteilte Modellierung unterstützen, ideal.

Integration im Generator mittels Modellabgleich

Bietet das Modellierungswerkzeug keine vernünftigen Integrationsmöglichkeiten, kann auch auf Generatorebene integriert

werden. Der Generator liest dabei mehrere Modelle ein, die dann jeweils bestimmte Modellelemente enthalten, wie in Abb. 119 ersichtlich.

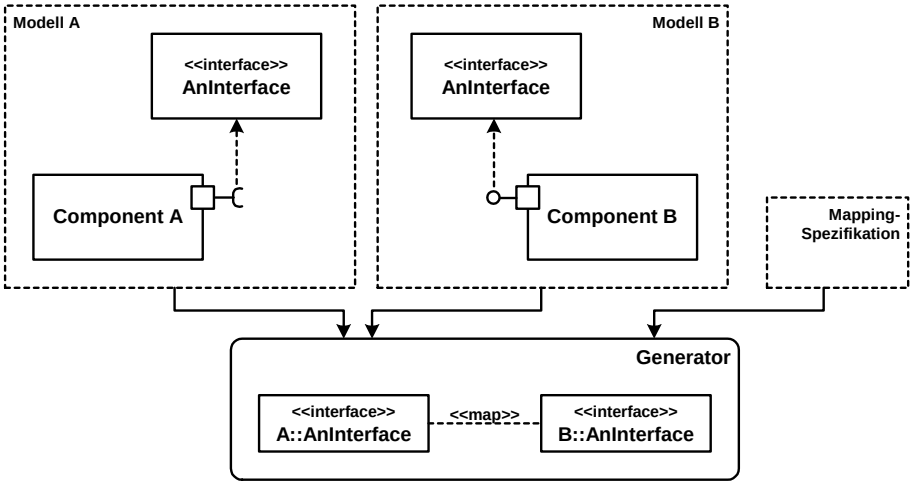


Abb. 119*Duplizierung der gemeinsamen Modellelemente und Auflösung durch den Generator*

In diesem Fall ist es Aufgabe des Generators, das mögliche Konsistenzproblem zu lösen.

- Das Mapping kann entweder explizit über eine Mapping-Spezifikation durchgeführt werden oder einfach aufgrund von Namensgleichheit erfolgen.
- Sind die beiden (per Mapping als *gleich* definierten) Modellelemente inhaltlich unterschiedlich, kann entweder eine Fehlermeldung ausgegeben werden oder aber eine Adaption findet statt. Auch diese kann (in gewissen Grenzen) wieder automatisch geschehen.

Integration im Generator mittels Referenzen (Proxy-Elemente)

Eine weitere Möglichkeit der Integration besteht in der Verwendung von Referenzen, wie es in Kapitel <Ref Trans::BP-Tech::Proxy> erläutert wurde. Abb. 120 zeigt nochmals das Prinzip: Das Interface *AnInterface* ist nur in einem Modell vorhanden. Die anderen Modelle enthalten lediglich Interface-*Referenzen* auf das Interface. Die Zuordnung kann beispielsweise über den Namen geschehen und wird automatisch vom Generator erledigt.

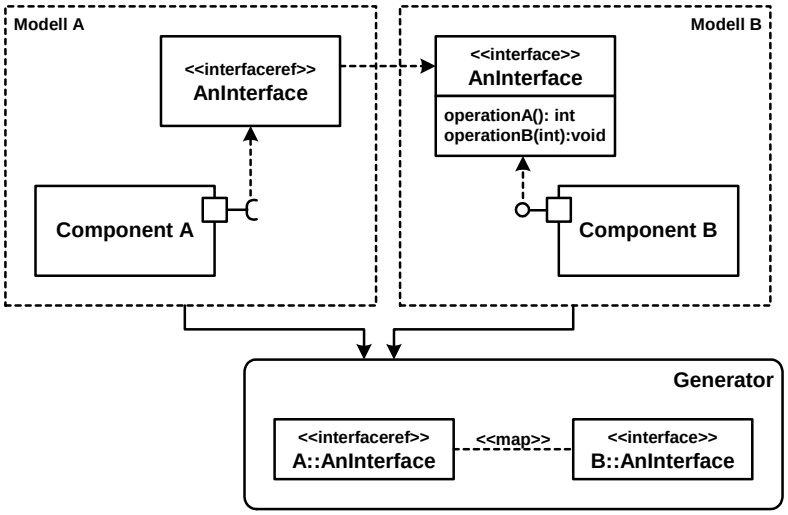


Abb. 120 *Verwendung von Referenz-Modellelementen, um gemeinsam genutzte Modellelemente zu realisieren*

16. Grundlagen des Product Line Engineering

Mit Jorn Bettin

Product Line Engineering befasst sich mit der systematischen Analyse von Domänen bis hin zum Entwurf von Software-Produktionsstraßen. Ziel ist es, das Automations- und Wiederverwendungspotenzial bei der Herstellung von Softwaresystemen voll auszuschöpfen. Product Line Engineering fügt sich damit nahtlos als Analysemethode in den MDSD-Kontext ein.

Eine umfassende Darstellung des Themas würde den Rahmen dieses Buches bei weitem sprengen. Wir begnügen uns daher mit der Darstellung der Grundlagen und der Verknüpfung mit den schon vermittelten MDSD-Kenntnissen. Am Ende des Kapitels geben wir ausführliche Literaturhinweise. Mit den ökonomischen und organisatorischen Aspekten des Product Line Engineerings im Zusammenhang mit MDSD befassen wir uns in Teil 4 des Buches.

16.1. Software-Systemfamilien und Produktlinien

Zwei zentrale Schlüsselbegriffe, nämlich *Software-Systemfamilie* und *Produktlinie*, haben wir bereits in Kapitel <Ref Begr> kurz definiert.

Die ursprüngliche Definition einer Software-Systemfamilie ist folgende:

> *We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members. [Par76]*

Eine Software-Systemfamilie liegt also immer dann vor, wenn man eine Reihe von Systemen (in diesem Zusammenhang oft als *Produkte* bezeichnet) entwickelt, die „relevante“ gemeinsame Eigenschaften besitzen. Im Kontext von MDSD sind diese Gemeinsamkeiten in der Domänenarchitektur zusammengefasst. Das bedeutet umgekehrt, dass die Gemeinsamkeiten je nach Ausrichtung der Domäne infrastruktureller, architektonischer oder auch fachlicher Natur sein können.

Eine *Produktlinie* besteht aus einer Menge von fachlich aufeinander abgestimmten Einzelprodukten mit einem gemeinsamen Zielmarkt – d.h. die Zusammenstellung ist kundengruppenspezifisch. Idealerweise wird eine Produktlinie mit Hilfe einer Software-Systemfamilie realisiert.

16.2. Einordnung in den MDSD-Prozess

Betrachten wir den in Kapitel <Ref Proz> beschriebenen Entwicklungsprozess. An verschiedenen Stellen treten analytische Tätigkeiten auf, die wir zwar konstruktiv, aber noch nicht methodisch systematisiert haben. Dazu zählen

- die Definition und Begrenzung der Domäne, für die Software entwickelt werden soll,
- die Definition der wichtigsten Kernkonzepte der Domäne,
- die Analyse der Gemeinsamkeiten und Unterschiede von Softwaresystemen der Domäne – und damit
- die Trennung von Anwendung und Domänenarchitektur,
- die Definition der wichtigsten Basiskomponenten der MDSD-Plattform (Lösungsraum),
- die Definition des Produktionsprozesses für die Software-Systemfamilie.

Product Line Engineering liefert die methodische Basis für systematische Wiederverwendung im Rahmen von Software-Systemfamilien.

MDSD kann in diesem Zusammenhang also entweder als ein Realisierungsmechanismus für Product Line Engineering angesehen werden, oder umgekehrt Product Line Engineering als eine Analysemethode für MDSD. Product Line Engineering kann und sollte durchaus iterativ und inkrementell im Sinne von MDSD angewendet werden, es ist nicht als abgetrennte Vorstufe zu MDSD zu verstehen,

sondern vielmehr als begleitende Methode, wenngleich die Intensität während der Elaborationsphase sicherlich besonders groß ist.

16.3. Methodik

Der Prozess des Product Line Engineering besteht aus drei Phasen, die wir nun im Einzelnen kurz erläutern werden (siehe Abb. 121).

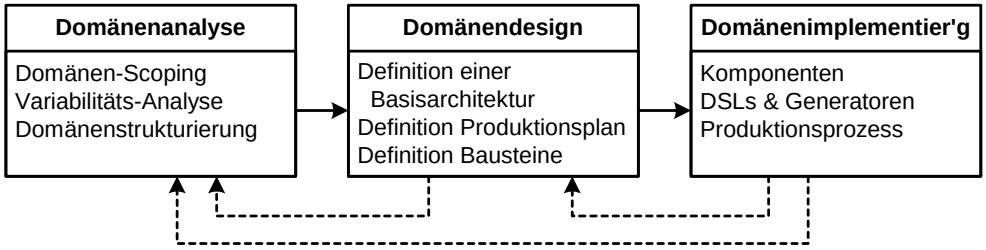


Abb. 121 *Schritte im Rahmen des Product Line Engineering*

16.3.1. Domänenanalyse

Der erste Schritt der Domänenanalyse ist das Domain Scoping. Hier werden zunächst die Grenzen der Domäne festgelegt. Wenn wir als Beispiel Motorsteuergeräte (für Automobile) annehmen, so ist es beispielsweise entscheidend, festzulegen, ob diese nur für Benzin- oder auch für Dieselmotoren verwendbar sein sollen, oder ob sowohl PKW- als auch LKW-Motoren dazugehören. Dies ist aus zwei Gründen wichtig:

- Dinge, die so unterschiedlich sind, dass sie im Rahmen einer Familie nicht sinnvoll abgebildet werden können, müssen ausgegrenzt werden, damit eine konsistente Realisierung überhaupt möglich ist. Nicht zuletzt ist dies eine Risikominimierungs-Strategie.
- Unklare Anforderungen führen zu fortwährenden Diskussionen und damit zu Reibungsverlust. Dies bedeutet nicht, dass ein iteratives Anforderungsmanagement ausgeschlossen wird, sondern nur, dass die Anforderungen (in diesem Fall der Scope) zu jedem Zeitpunkt klar definiert sind.

Dies erklärt auch, warum es vor allem im Rahmen einer „reifen“ Domäne sinnvoll ist, so vorzugehen: Wenn Sie die Unterschiede der

Produkte der Domäne noch nicht kennen, dann können Sie die nötigen Entscheidungen auch nicht treffen[55]. Diese Aussage bezieht sich nicht auf die Implementierung der Domänen-Konzepte, sondern vielmehr auf ihre „äußere Semantik".

Kehren wir noch einmal zu unserer ersten Case-Study (<Ref CSI>) zurück: Die Domäne hieß dort: „Anwendungsarchitektur für E-Business Software". Die Car-Sharing-Applikation ist ein Produkt dieser Domäne. Startet man nun mit dem Architektur-Bootstrapping via Referenzimplementierung, so wird man zweifelsohne nur Erfolg haben, wenn die Architekten bereits Kenntnisse über Schichtungen, typische J2EE-Patterns, MVC-Strukturen, deklarative Ablaufsteuerungen etc. haben. Da dies mehr oder weniger Allgemeingut ist, kann man hier von einer „reifen" Domäne sprechen. Es ist jedoch nicht notwendig, die konkrete Implementierung der Architekturmuster bei Projektstart bereits zu kennen. Auch die Designsprache (UML-Profil) kann sich während des Projektes noch in Maßen weiterentwickeln (siehe <Ref Vers>) – vor allem kann sie leicht erweitert werden.

Um die Unterschiede der einzelnen Produkte der Familie herauszufinden und systematisch zu dokumentieren, gibt es verschiedene Möglichkeiten. Eine recht mächtige entstammt der FODA [FODA] und ist unter dem Namen Feature-Modellierung [EC00] bekannt.

Ein Feature-Modell (grafisch durch ein Feature-Diagramm ausgedrückt) zeigt, welche Features die einzelnen Produkte einer Software-Systemfamilie haben können, müssen oder dürfen. Ein Feature-Modell ist eine hierarchische Struktur, wobei jedes Feature Unterfeatures enthalten kann. Diese können für ein spezifisches Produkt verpflichtend, optional, alternativ oder n-aus-m sein.

Abb. 122 soll dieses Konzept verdeutlichen. Es zeigt ein (stark vereinfachtes) Feature-Diagramm für die Systemfamilie *Stack*.

---

[55] Dies mag zunächst nach *Big Design Up Front* klingen, ist aber nicht so gemeint: Wir sagen nicht, dass man eine unbekannte Domäne vorher analysiert, sondern dass man die Domäne und ihre Eigenheiten aus *Erfahrung* kennt. Dies ist vergleichbar mit der industriellen Massenfertigung: Man wird keine vollautomatische Produktionsstraße aufbauen für Produkte, die man noch nicht richtig kennt.

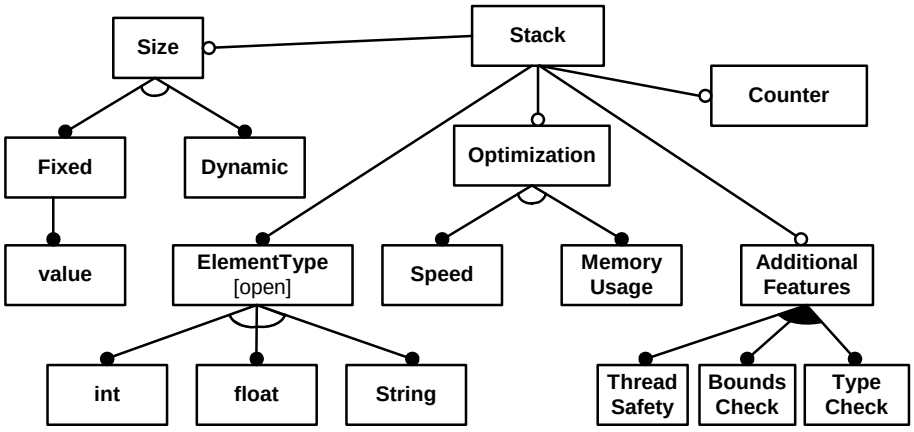


***Abb. 122** Beispielhaftes Feature-Diagramm*

Es beschreibt, dass jeder *Stack* einen *ElementType* haben muss („muss“: dargestellt durch den ausgefüllten Kringel). Dieser kann vom Typ *int*, *float* oder *String* sein (*1-aus-n*: erkennbar durch den nicht-ausgefüllten Bogen zwischen den Assoziationen *ElementType-int*, *ElementType-float* und *ElementType-String*). Die Größe des Stacks kann entweder fix (wobei man dann einen Wert für die Größe angeben muss) oder dynamisch anpassbar sein. Optional (*optional:* leerer Kringel) kann der Stack einen statischen *Counter* mitführen – tut er das nicht, wird beim Aufruf von *size()* die Größe jedes Mal neu berechnet. Weitere Features sind Thread-Sicherheit, Bounds-Checking sowie Typsicherheit. Eines oder mehrere dieser Features können in einem Produkt vorhanden sein, (*n-aus-m*: erkennbar durch den ausgefüllten Bogen). Auch kann die Implementierung entweder auf Geschwindigkeit oder Speicherverbrauch hin optimiert werden.

Das Diagramm in Abb. 122 beschreibt damit den „Konfigurationsraum“ für Mitglieder der Systemfamilie *Stack*. Einzelne Mitglieder müssen gültige Kombinationen aufweisen. Beispiele:

- Dynamische Größe, ElementType: int, Zähler, Threadsafe
- Statische Größe mit dem Wert 20, ElementType: String
- Dynamische Größe, geschwindigkeitsoptimiert, Bounds-Check

Abgesehen von den in der grafischen Notation direkt darstellbaren Angaben kann ein Feature-Modell noch weitere Informationen enthalten, beispielsweise Namen für bestimmte Kombinationen von Features (Makros), die Multiplizität von Subfeatures, die Priorität eines Features im Rahmen der Implementierung, vom Feature betroffene Stakeholder etc. Auch zusätzliche Constraints, die nicht in die grafisch dargestellte

Struktur passen, lassen sich definieren. Die Praxis zeigt, dass man mit den folgenden vier zusätzlichen Constraints schon sehr weit kommt:

- *requires:* Ein bestimmtes Feature erfordert ein anderes zwangsläufig.
- *excludes:* Das Vorhandensein eines Features verbietet das gleichzeitige Vorhandensein eines anderen.
- *recommends:* Weichere Variante von *requires*; ein bestimmtes Feature legt die Verwendung eines anderen nahe.
- *discourages:* Weichere Variante von *excludes*; man sollte zwei Features nicht gleichzeitig verwenden.

Ein Beispiel hierfür sind die beiden Optimierungs-Features:

- Optimization for Speed: *requires* Counter, *requires* Fixed Size, *discourages* Thread Safety, *discourages* Bounds Check.
- Optimization for Memory Usage: *requires* Dynamic Size, *discourages* Counter.

Das Interessante an dieser Methode und Notation ist, dass sie absolut nichts über die spätere Implementierung der Features aussagt. Würde man eine Systemfamilie in dieser Phase bereits mittels UML modellieren, dann müsste man bereits zu diesem frühen Zeitpunkt Entscheidungen treffen bzgl. Vererbung, Typgenerizität, Assoziationen etc. Dies ist zu dieser Zeit der Domänenanalyse aber weder notwendig noch hilfreich. Es geht hier lediglich um eine Analyse der konzeptionellen Gemeinsamkeiten und Unterschiede zwischen den Produkten einer Software-Systemfamilie. Wie diese Variabilitäten später im Rahmen der Software implementiert werden, wird erst während der Designphase festgelegt.

Des Weiteren können Merkmale in obigen Diagrammen zu verschiedenen Kategorien gehören, die das Feature-Modell nicht unterscheidet:

- Klassische Komponentenmerkmale geben das Vorhandensein modularer Features an – d.h. ob ein Produkt ein bestimmtes Subsystem/Modul besitzt oder nicht.
- Aspektuelle Merkmale sind im Gegensatz dazu solche, die nicht als Komponente realisiert werden können (z.B. das Merkmal „optimiert auf Performance“ oder „optimiert auf Codegröße“). Solche Merkmale schlagen sich später möglicherweise an vielen

Stellen im System nieder, indem verschiedene Komponenten anders implementiert werden.

Feature-Diagramme können beide Arten von Merkmalen nebeneinander aufweisen, wohingegen vor allem aspektuelle Merkmale mittels UML nur sehr schwer darstellbar sind.

Es ist in dieser Phase zudem wichtig, Abhängigkeiten zwischen den Features festzulegen. Ein Produkt kann möglicherweise bestimmte Features nur besitzen, wenn es bestimmte andere Features auch besitzt. Auch das Gegenteil ist natürlich möglich, nämlich, dass sich bestimmte Features gegenseitig ausschließen. Damit wird auch eine Entscheidungsreihenfolge festgelegt. Dies ist essenziell, um im nächsten Schritt den „Produktionsprozess“ zu definieren. Die Definition sinnvoller Defaults für den Fall, dass bestimmte Features nicht explizit angegeben sind, ist ebenfalls wichtig.

In der zweiten Fallstudie (<Ref CSII>) finden sich weitere Beispiele für Feature-Modelle.

16.3.2.Domänendesign & -implementierung

Im Rahmen des Domänendesigns geht es nun darum, die Softwarestruktur der Domäne festzulegen. Beginnen wird man damit, die gemeinsamen Merkmale der Produkte einer Domäne in Form einer Plattform zu implementieren. Dadurch, dass diese gemeinsamen Merkmale eben bei allen Produkten identisch sind, ist es nicht nötig, sie in irgendeiner Weise generativ zu implementieren. Sie stellen die Basis für die gemeinsame Zielarchitektur dar.

Für die variablen Merkmale (in denen sich die verschiedenen Produkte unterscheiden) muss nun entschieden werden, wann ein Merkmal „gebunden“ werden soll – d.h. zu welchem Zeitpunkt man sich bei einem Produkt für oder gegen ein bestimmtes Merkmal entscheidet. Dabei gibt es verschiedene Alternativen:

- Auf Quellcodeebene: Hier wird die Entscheidung für oder gegen ein Feature bereits im Rahmen der Programmierung festgelegt, also im Quellcode „fest verdrahtet“.
- Zur Übersetzungszeit: Dem Compiler können bestimmte Entscheidungen überlassen werden (Beispiele: überladene Funktionen, Präprozessoren, Code/Aspekt Weaver)
- Zur Link-Zeit: Auch der Linker kann durch Hinzubinden der einen oder anderen Bibliothek dazu dienen, ein Produkt zu

konfigurieren (Beispiel: Ein Makefile, welches bestimmte Bibliotheken statisch zum Code bindet – oder auch nicht).

- Zur Installationszeit: Bei Produkten, die einen expliziten Deployment-Schritt enthalten, kann man typischerweise auch hier noch bestimmte Konfigurationen vornehmen (Beispiel: J2EE bietet mittels der Deployment-Deskriptoren die Möglichkeit, bestimmte Einstellungen bei der Installation anzupassen).
- Zur Ladezeit: Auch beim Laden der Anwendung kann man noch eingreifen, DLLs sind ein Beispiel (Beispiel: Das optionale Laden verschiedener DLLs, die die gleichen Funktionen unterschiedlich implementieren).
- Zur Laufzeit: Zu guter Letzt kann man Entscheidungen natürlich auch zur Laufzeit treffen (Beispiele: Polymorphismus, Dynamic Classloading in Java, interpretierte Konfigurationsparameter).

All diese verschiedenen Möglichkeiten haben ihre Vor- und Nachteile, einige davon seien im Folgenden kurz erläutert:

- Performance: Typischerweise ist die Performance von fest kodierten Features deutlich besser, als wenn Entscheidungen dynamisch zur Laufzeit stattfinden. Entscheidungen zur Compile-Zeit sind üblicherweise genauso performant. Solche, die vom Linker oder Loader getroffen werden, sind meist auch nicht erheblich langsamer.
- Codegröße: Wenn Verhalten zur Lade- oder Laufzeit entschieden werden soll, dann muss im Programm-Image natürlich der Code für alle Alternativen vorhanden sein, die Größe des Images steigt. In diesem Zusammenhang muss allerdings noch unterschieden werden zwischen Codegröße des Images und der des laufenden Programms im Hauptspeicher. Entscheidungen, die der Compiler trifft, sind üblicherweise sehr nah am Optimum, jedoch muss man aufpassen, dass der Compiler nicht Overhead durch „Expansion“ erzeugt (siehe C++-Templates).
- Flexibilität: Merkmale, die fest kodiert sind, lassen sich – außer durch Neuprogrammierung – nicht mehr ändern. Ansonsten steigt die Flexibilität typischerweise, je später die Entscheidung für oder gegen das Feature getroffen wird.
- Komplexität: Bestimmte Features lassen sich leicht „herausschneiden“ und damit auch sehr spät zum System wieder hinzufügen (Beispiel: Dynamisches Anziehen einer Komponente).

> Aspektartige Features sind problematischer, weil sie sich quer durch das System ziehen. Sie lassen sich umso einfacher realisieren, je früher man sie festlegt.

Der Produktionsprozess beschreibt nun, wie man aus einer Produktspezifikation (dem Modell) ein lauffähiges Produkt erstellt. Dies ist also der Übergang zu den Inhalten, die wir im Rahmen des Buches bereits diskutiert haben. MDSD stellt eine Möglichkeit dar, die genannten Alternativen rationell und konsistent umzusetzen.

Ein Generator ist dafür das ideale Werkzeug. Für die Quellcodeebene ist dies offensichtlich: Die Generierung von Quellcode ist die originäre Aufgabe. Aspekt-Features lassen sich ebenfalls implementieren (vgl. <Ref Code> und [Voe04]). Merkmale, über die zur Linkzeit entschieden werden soll, können realisiert werden, indem der Generator die betreffende Bibliotheken erstellt bzw. ein Makefile generiert. Das Deployment kann vom Generator zusammen mit dem Deployment-Skript übernommen werden. Auch der Ladevorgang lässt sich generativ beeinflussen. Last but not least kann ein Generator (Default-)Konfigurationsdateien erzeugen, die zur Laufzeit interpretiert werden. Insgesamt kann der Generator also zu einem Integrationswerkzeug für die Software-Systemfamilie werden.

Der Generator wird durch die Domänenarchitektur (inklusive DSL) konfiguriert. Diese kann in Subdomänen unterteilt oder partitioniert sein, wie in <Ref Trans::BP-Tech::SubDom> und <Ref Vers::TeamMdl> geschildert. Der Generator muss also ggf. in der Lage sein, die mit verschiedenen DSLs modellierten Teilmodelle eines Systems zu lesen und daraus ein homogenes, konsistentes System zu erzeugen. Ein Beispiel dafür bietet die Fallstudie im nächsten Kapitel.

16.4. Domänenmodellierung

Eine Domäne besteht aus domänenspezifischen Abstraktionen, Konzepten und Regeln. Wie stellen wir sicher, dass die definierten DSLs, und somit die modellierten Anwendungen, eine korrekte Abbildung der Domäne darstellen?

Den Schlüssel dazu stellt die in Kapitel <Ref MM> beschriebene Metamodellierung dar. Das Metamodell der DSL muss entsprechend restriktiv und so „nah“ an der Domäne wie möglich sein. Die oben gezeigten Feature-Modelle können in dieser Hinsicht sehr hilfreich sein. Das Metamodell muss ggf. nicht erwünschte oder benötigte

Eigenschaften des Basismetamodells (z.B. das der UML) „ausblenden“. Constraints stellen in diesem Zusammenhang ein mächtiges Mittel dar.

Ein Glossar oder eine Ontologie der Domäne kann der erste hilfreiche Schritt in die Richtung eines geeigneten Metamodells sein. Im Rahmen der zweigleisig iterativen Entwicklung (vgl. <Ref Proz::IterX2> wird es verfeinert.

16.4.1.Beispiel: Simple Components

Am besten erkennt man die Wichtigkeit der kontinuierlichen Validierung und Weiterentwicklung des formalen Metamodells einer Domäne an einem kleinen Beispiel (siehe Abb. 123).

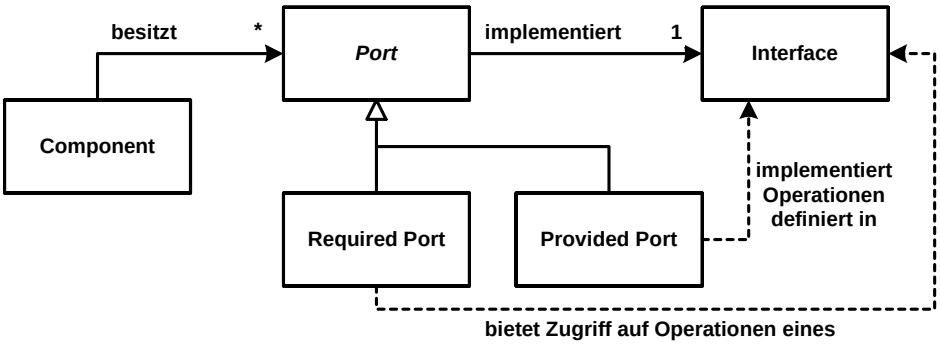


Abb. 123 *Simple Components Meta Model*

Das Metamodell muss mit Hilfe von Domänenexperten überprüft werden. Es muss für die Entwicklungsteams ohne Missverständnisse reibungslos benutzbar sein. Zu diesem Zweck ist es nützlich, in allen Diskussionen mit Stakeholdern und Domänenexperten konsequent die Terminologie des Metamodells in Gesprächen zu verwenden. Fehler und Ungereimtheiten, die sich ins Metamodell eingeschlichen haben, werden auf diese Art und Weise meist schnell entdeckt. Im Gespräch mit Stakeholdern können beispielsweise Sätze benutzt werden wie

- Eine Komponente besitzt eine Reihe von Ports.
- Jeder Port implementiert genau ein Interface.
- Es gibt zwei Arten von Ports: Required Ports und Provided Ports.
- Ein Provided Port implementiert die Operationen seines Interfaces.
- Ein Required Port bietet Zugriff auf die Operationen, welche in seinem Interface definiert sind,

Sobald sich herausstellt, dass etwas sich nicht reibungslos mit dem Vokabular des Metamodells ausdrücken lässt, bedeutet dies entweder, dass die Formulierung nicht dem Metamodell entspricht, oder dass das Metamodell fehlerhaft bzw. zu ungenau ist.

Diese Technik ist in der Domänenmodellierung weit verbreitet. In seinem Buch über Domain-Driven Design, bezeichnet Eric Evans diese Technik als Ubiquitous Language [Eva00]. Dieses Buch beschreibt übrigens viele nützliche Techniken zum Entwurf von domänenspezifischen Frameworks.

16.4.2.Beispiel: Enterprise Applications

Für den Fall, dass zum Beispiel aus Mangel an tiefem fachlichem Domänenwissen kein ausgeklügeltes domänenspezifisches Metamodell entwickelt werden kann, können zumindest wichtige Konzepte einer technischen Architekturplattform im Metamodell verankert werden. Dies führt zu architekturzentrierter modellgetriebener Softwareentwicklung und relativ einfachen Transformationen. Über mehrere Iterationen bzw. Releases können dann mehr und mehr fachliche Konzepte in das Metamodell eingebaut werden, welche die Ausdrucksstärke der Modelle erhöhen und die Abhängigkeit von den Implementierungstechnologien weiter verringern. Dies resultiert in komplexeren Transformationen und liefert daher Anreize zur Erstellung einer fachlichen MDSD-Plattform,

Abb. 124 bis Abb. 126 zeigen ein Metamodell, welches für die Entwicklung von komplexen verteilten Geschäftsanwendungen konzipiert wurde und welches eine Reihe von Randbedingungen sicherstellt:

5. Nur *BusinessComponents* können *Attribute* haben.
6. Nur *BusinessComponents* haben *VisualRecognition.*
7. Die erlaubten Typen in Operationen von *BusinessComponents* beschränken sich auf *BusinessTypes.*
8. Die erlaubten Typen in Operationen von *EnterpriseComponents* beschränken sich auf *EnterpriseParameterTypes*.
9. *BusinessComponents* können nur als Teil von *EnterpriseComponents* existieren.
10. *BusinessComponents* können keine *EnterpriseComponents* enthalten.

11. *BusinessComponents* können über Assoziationen, Aggregation und Komposition miteinander verknüpft werden.

12. *EnterpriseComponents* können nur durch *requiredComponents* mit anderen *EnterpriseComponents* verknüpft werden.

13. Unterscheidung von *Responsibilities* und *Operations*. Erstere werden von Analytikern und Fachexperten benutzt, um informell die wesentlichen Eigenschaften von Komponenten zu definieren, während Letztere Teil der formalen Komponentenspezifikation sind, welche zur Generierung eingesetzt wird.

Das primäre Ziel dieser Randbedingungen ist Skalierbarkeit der Architektur und Kontrolle über die Stärke von Abhängigkeiten zwischen Enterprise-Komponenten – kurz Komplexitätsmanagement.

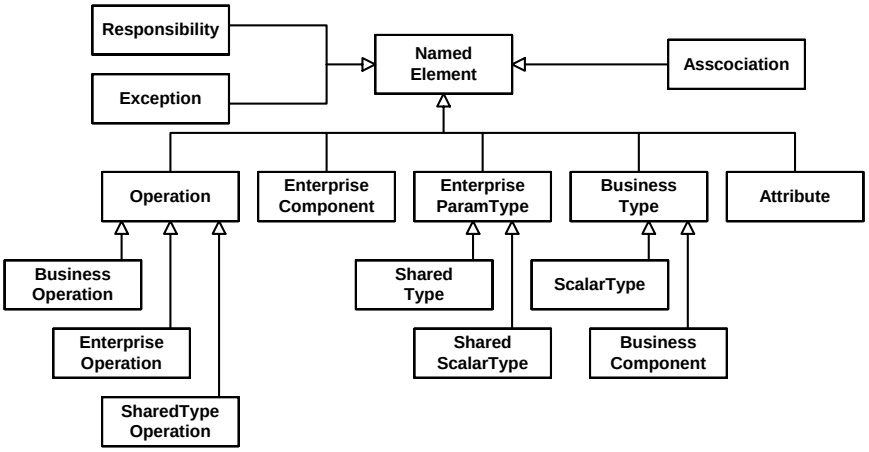


Abb. 124*Enterprise Application Meta Model – Vererbungsbeziehungen*

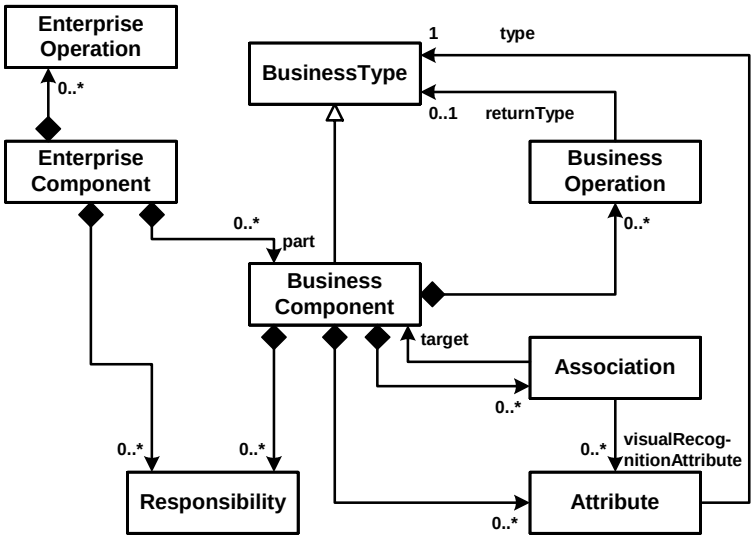


Abb. 125*Enterprise Application Meta Model – Kern*

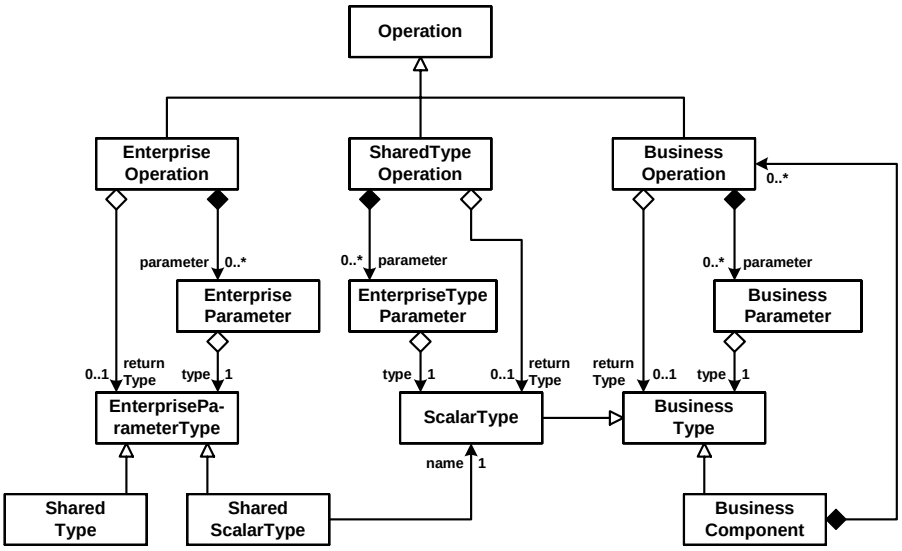


Abb. 126*Enterprise Application Meta Model – Verhalten*

16.4.3. Beispiel: Modellierung von Hochspannungsnetzen

Das Beispiel in diesem Abschnitt kommt direkt aus der Anwendungspraxis eines Hochspannungsnetzbetreibers, welcher Netzmodelle in einer DSL abbildet und die Informationen dann zur Berechnung von Stromtransportkosten zwischen Generatoren und Stromvertriebsgesellschaften benutzt. In diesem Fall wird die DSL von Ingenieuren, d.h. von Endanwendern, verwendet, um Hochspannungsnetze zu modellieren. Zur Stromtransportkostenberechnung wird das Modell dynamisch von einer entsprechenden Anwendung ausgewertet.

In Abb. 127 sehen wir das Kern-Metamodell, welches den Hochspannungsnetzmodellen zugrunde liegt.

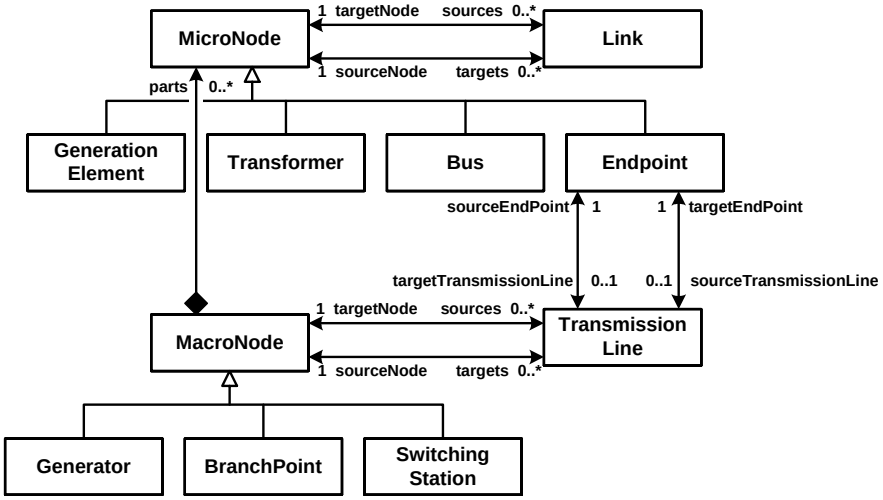


***Abb. 127** Hochspannungsnetz-Metamodell*

Die daraus resultierende DSL bietet sich ideal zur Abbildung auf UML-Composite-Structure-Diagramme an:

1. Alle Unterklassen von *MacroNode* werden auf Instanzen abgebildet.
2. *EndPoints* werden auf Ports abgebildet.
3. Alle anderen Unterklassen von *MicroNode* werden als Instanzen-Teile (Parts) abgebildet.
4. *Links* und *TransmissionLines* werden auf Konnektoren abgebildet.

Mit dieser Notation können nun Standard-UML-Werkzeuge benutzt werden (Voraussetzung ist Unterstützung des UML 2-Standards), um Hochspannungsnetze grafisch zu modellieren. Abb. 128 und Abb. 129 zeigen Beispiele.

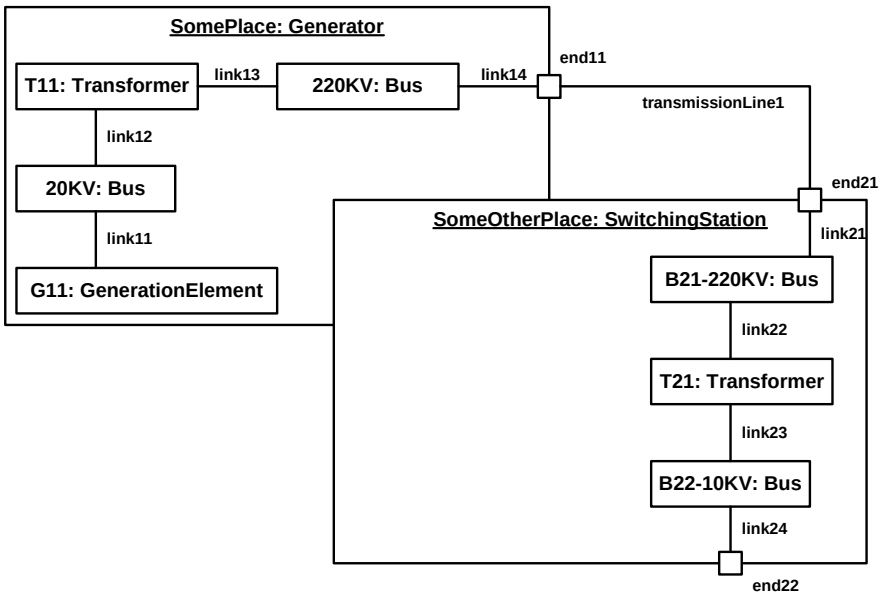


Abb. 128 *Ausschnitt eines Hochspannungsnetzes – Mikro-Maßstab*

In der Praxis kann die Restriktivität des obigen Metamodells nur durch MOF-basierte Werkzeuge oder solche, die zumindest einen Constraint-Mechanismus (z.B. OCL) auf Ebene des UML-Metamodells anbieten, direkt abgebildet werden. In allen anderen Fällen muss die Wohlgeformtheit der Modelle in einem nachgelagerten Schritt validiert werden (vgl. <Ref MM::Valid>).

Um zu verstehen, warum Ports notwendig sind, lohnt es sich, unser kleines Modell aus „größerer Entfernung“ zu betrachten – auf einer Abstraktionsebene, wo der Inhalt von *MacroNodes* irrelevant ist.

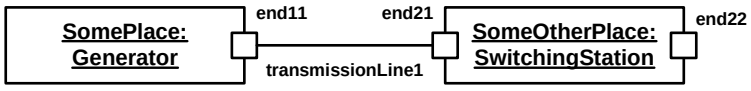


Abb. 129 *Hochspannungsnetz – Makromaßstab*

Ohne Ports wäre das Diagramm in Abb. 129 unpräzise und keine saubere Abstraktion des Diagramms in Abb. 128. Ports bieten die Möglichkeit, komplexe (Komponenten-)Systeme im „top-down“-Verfahren zu modellieren und nach und nach zu verfeinern. Wie unser Beispiel zeigt,

ist dies nicht nur bei der Modellierung von Software nützlich, sondern auch in anderen Domänen.

Im konkreten Anwendungsbeispiel wird die hier vorgestellte DSL in Kombination mit einer weiteren DSL zur dynamischen Definition von Preismodellen verwendet. Beide DSLs zusammen erlauben dann Endbenutzern sowohl Änderungen an der Struktur des Netzes als auch Änderungen an der Definition von Preismodellen. Somit wurde eine Anwendung, die ursprünglich immer wieder durch traditionelle Programmierung an neue gesetzliche Richtlinien und Preismodelle angepasst werden musste, durch Endbenutzer-„Programmierung“ mittels DSLs flexibilisiert.

16.5. Weiterführende Literatur

Aufgrund der Tatsache, dass wir in diesem Kapitel nicht in ausreichender Tiefe auf das Thema Product Line Engineering eingehen können, möchten wir etwas ausführlicher auf weiterführende Literatur verweisen.

- *Clements, Northrop, „Software Product Lines – Practices and Patterns" [CN01]*: Dieses Buch ist ein gutes Nachschlagewerk zur Product-Line-Engineering-Terminologie vom Software Engineering Institute der Carnegie Mellon Universität. Darüber hinaus bietet das Buch einen Überblick von klassischen, nicht unbedingt modellgetriebenen Product-Line-Engineering-Verfahren.
- *Weiss, Lai, „Software Product Line Engineering – A Family-Based Software Development Process" [WL99]*: Dieses Buch bietet einen systematischen Überblick von FAST (Family-oriented Abstraction, Specification and Translation), einer erprobten und seit vielen Jahren im Einsatz befindlichen Product-Line-Engineering-Methode.
- *Cleaveland, „Program Generators with XML and Java" [CG01]*: Dieses Buch bietet einen guten Einstieg in die Domänenanalyse, mit sehr gut zugänglichen Beispielen. Das Buch schließt ab mit wichtigen Überlegungen zur Entwicklung von DSLs zur Definition und Verknüpfung von Komponenten. Der Mittelteil des Buchs befasst sich mit Implementierungstechniken von Codegeneratoren.
- *Bosch, „Design & Use of Software Architectures – Adopting a product-line approach" [Bos00]*: Dieses Buch bietet interessante

Fallstudien und beschreibt einen Ansatz zur Architekturentwicklung von Produktlinien. Die Stärke des Buchs liegt im Vergleich und in der Auswertung einer Reihe von Organisationsformen, welche für Product Line Engineering in Betracht kommen.

- *Atkinson et al., „Component-based Product Line Engineering with UML" [ABB+01]*: Das Buch beschreibt den KobrA-Ansatz zum Product Line Engineering vom Fraunhofer Institut für Experimentelles Software Engineering. Der Schwerpunkt liegt hierbei auf praktischen Standards zur Entwicklung von Komponentenarchitekturen. Zu diesem Aspekt bietet dieses Buch nach unserer Einschätzung das beste Material.
- *Eisenecker, Czarnecki, „Generative Programming" [EC00]*: Dieses Buch beschäftigt sich mit der generativen Programmierung. Unter anderem enthält das Buch aber auch eine gute Einführung zum Thema Product Line Engineering und betrachtet insbesondere auch die Feature-Modellierung recht ausführlich.

17. Fallstudie: Eingebettete Komponenteninfrastrukturen

17.1. Überblick

Dieses Kapitel enthält eine zweite ausführliche Fallstudie. Dabei handelt es sich um ein Beispiel aus der Welt der eingebetteten Systeme (*Embedded Systems*). Es geht um die modellgetriebene Entwicklung einer Komponenteninfrastruktur. Unter anderem wird dieses Kapitel die folgenden Themen illustrieren:

- Modellierung mittels textueller DSLs
- Verwendung und Anpassen des openArchitectureWare-Generators
- Verwendung verschiedener Modelle für verschiedene Subdomänen
- Erstellung von Infrastruktur, nicht von Anwendungen
- Produktlinien und Feature-Modelle
- „Pseudodeklarative“ Programmierung von Metamodell-Constraints
- Interceptoren im Generator, um verschiedene Aspekte im Metamodell zu trennen
- Generierung von Build-Files
- Kaskadierte Domänenarchitekturen

Die Fallstudie stützt sich dabei auf Erfahrungen aus verschiedenen realen Projekten.

Beachten Sie bitte, dass die Erläuterungen in diesem Kapitel voraussetzen, dass Sie prinzipiell verstanden haben, wie der

openArchitectureWare-Generator [OAW] funktioniert[56]. Die Fallstudie im ersten Teil des Buches (<Ref CSI>) vermittelt diese notwendigen Grundlagen und sollte daher vorher gelesen werden.

17.1.1. Einführung und Motivation

Eingebettete Systeme werden immer komplexer:

- Handys können fotografieren, im Internet surfen sowie Videostreams anzeigen (und tatsächlich auch immer noch telefonieren).
- Autos enthalten bis zu 70 mehr oder weniger kleine Rechner (ECUs – Electronic Control Units), die die verschiedensten Aspekte des Fahrzeuges steuern. Die Rechner sind mit verschiedenen Bussystemen vernetzt und enthalten große Mengen an Software.
- Immer mehr „Appliances“, also elektronische Geräte, enthalten immer größere Mengen Software.

Man hat es sich in der Welt der eingebetteten Systeme lange leisten können, die Software von Hand ideal auf die zur Verfügung stehende Hardware anzupassen, mit den offensichtlichen Folgen für Time-to-Market und Aufwand. Dies wird aber zunehmend schwieriger – alle paar Monate kommen neue Handy-Modelle auf den Markt, die Modellzyklen bei modernen Automobilen werden auch immer kürzer.

Des Weiteren muss aufgrund der immer größeren Verbreitung von Elektronik und Software in unserem Alltag die Qualität und Zuverlässigkeit der Software gehalten, wenn nicht gar verbessert werden – in der jüngeren Vergangenheit wurden da ja durchaus massive Qualitätsprobleme bei den Automobil- und den Handyherstellern bekannt.

Die manuelle „Einzelstückproduktion“ von eingebetteten Systemen muss also durch Produktlinien und Software-Systemfamilien abgelöst werden. Die Bedeutung von Softwarearchitektur als zentraler Baustein der Softwareentwicklung wird in diesem Zusammenhang immer wichtiger. Modellgetriebene Softwareentwicklung als Umsetzungsparadigma spielt dabei eine zentrale Rolle.

---

[56] Mittlerweile hat sich die konkrete API des Generators weiterentwickelt. Details und ein Tutorial finden sich unter www.openarchitectureware.org

Codegenerierung an sich ist im Rahmen der Entwicklung von eingebetteten Systemen bereits weit verbreitet. Allerdings wird in erster Linie Anwendungslogik generiert, basierend auf Modellen, die üblicherweise Zustandsmaschinen (für diskrete Systeme) oder Signalflussdiagramme (für kontinuierliche Systeme) sind. Aber auch die modellgetriebene Erstellung der technischen Infrastruktur, in der diese Funktionalitäten ablaufen, werden in steigendem Maße modellgetrieben erstellt. In verschiedenen Branchen sind firmenübergreifende Standardisierungsbemühungen unterwegs (beispielsweise [ASAR]).

Ein viel versprechender Ansatz ist dabei insbesondere die Kombination von Komponenteninfrastrukturen [Voe02] und modellgetriebener Softwareentwicklung.

17.1.2.Komponenteninfrastrukturen

Komponenteninfrastrukturen, wie in [VSW02] beschrieben, basieren auf zwei zentralen Punkten:

- Komponenten kapseln eine bestimmte, für sich allein sinnvolle und damit abgeschlossene Portion Anwendungsfunktionalität. Sie stellen diese über wohldefinierte Schnittstellen zur Verfügung. Die Komponente beschreibt weiterhin, welche Ressourcen (also z.B. andere Komponenten-Interfaces) sie zur Ausführung ihrer Funktionalität benötigt.
- Ein Container stellt den in ihm laufenden Komponenteninstanzen technische Basisdienste zur Verfügung. Wie diese genau aussehen, hängt von der Domäne ab und wird weiter unten für eingebettete Systeme beschrieben.

Eine Anwendung besteht damit also aus einer Reihe von Komponenteninstanzen, die in Containern laufen. Die Vorteile sind offensichtlich: Der Entwickler kann sich vor allem auf die Implementierung der Anwendungslogik konzentrieren, der Container liefert die technische Basisfunktionalität. Bei entsprechender Standardisierung kann dieser dann wiederverwendet oder zugekauft werden[57].

Die konkreten Anforderungen an Container sind dabei stark vom System abhängig, wie wir noch sehen werden.. Typische Beispiele, die

[57] Diese Effekte werden im Enterprise-Bereich am Beispiel von J2EE, COM+ oder CCM deutlich.

natürlich in Zusammenarbeit mit dem (Echtzeit-)Betriebssystem umgesetzt werden, umfassen Remote-Kommunikation mittels verschiedener Bussysteme, Abhängigkeitsmanagement, Startup-Koordination und Lifecycle-Management, Tasking und Scheduling usw.

17.1.3. Anforderungen an Komponenteninfrastrukturen bei eingebetteten Systemen

Nun gibt es in der Welt der eingebetteten Systeme eine besondere Herausforderung: Die Knappheit von Ressourcen wie Speicher oder elektrischer Energie, Prozessorleistung und – ganz wichtig – der Stückpreis des Gesamtsystems. Insofern ist es absolut zwingend, dass solche Komponenteninfrastrukturen bei all ihren Vorteilen nicht mit dem Preis erhöhten Ressourcenverbrauches und schlechterer Performance erkauft werden. Daraus ergeben sich interessante Herausforderungen, die sich mit Mitteln der MDSD gut angehen lassen.

17.1.4. Grundsätzlicher Ansatz

Der grundsätzliche Ansatz beruht nun darauf, den Container für die Komponenten zu generieren. Damit lassen sich die Container spezifisch für das Zielsystem anpassen und optimieren.

Durch verschiedene Modelle, die unterschiedliche technische Subdomänen beschreiben, werden die einzelnen Merkmale des Systems modelliert. Daraus erstellt der Generator einen Container, der genau zur Plattform, zu den Anforderungen der Komponenten und zur Hardware passt.

17.2. Product Line Engineering

Bevor wir nun tatsächlich zeigen, wie die einzelnen Artefakte erstellt werden, möchten wir anhand dieses Beispiels die im letzten Kapitel vermittelten Grundlagen des Product Line Engineering applizieren.

Die Container stellen eine Software-Systemfamilie dar. Die auf verschiedenen Systemen zum Einsatz kommenden Container besitzen Gemeinsamkeiten, aber auch Unterschiede. Beispielsweise werden sie nur dann Code für die Kommunikation über einen bestimmten Bus enthalten, wenn die betreffende Hardware auch an einem solchen Bus angeschlossen ist. Gleiches gilt für die Kommunikationsparadigmen. Wenn nur mittels asynchroner Events kommuniziert wird, braucht der

Container keinen Code enthalten, der synchrone Operationsaufrufe zwischen verschiedenen Komponenteninstanzen über den Bus transportiert.

Insofern macht es Sinn, dieses Projekt aus Sicht des Product Line Engineerings zu betrachten. Die Argumentation ist einfach:

- Hätten wir nur genau einen einzigen Container, der in allen Szenarien und Systemen Verwendung finden könnte, so wäre es ein Framework. Wie oben erläutert, ist dieser Ansatz aber aus Performance- und Codegrößengesichtspunkten unpraktikabel.
- Wenn jeder Container ein komplettes Unikat wäre, dann wäre das Konzept der Container ad absurdum geführt. Gewisse Dinge müssen bzw. sollen aus ökonomischen Gründen ja gerade ähnlich sein zwischen den Containern für verschiedene Systeme.

17.2.1. Domain Scoping

Zunächst müssen wir definieren, für welche Szenarien die Container geeignet sein sollen. Um diese Frage systematisch anzugehen, könnte man bereits hier Feature-Modellierung für die gesamte Domäne der eingebetteten Systeme verwenden, um damit festzusetzen, welchen Teil der Gesamtdomäne wir adressieren möchten.

In dieser Phase ist dieses Vorgehen aber oft übermäßig aufwendig – wir verwenden daher hier einen pragmatischeren Ansatz. Die folgenden Aspekte können zu Abgrenzung der Domäne herangezogen werden:

- **Betriebssystem:** In der Welt der eingebetteten Systeme kommt es durchaus vor, dass man direkt gegen die Hardware programmiert und kein Betriebssystem einsetzt. Auch die Features, die ein Betriebssystem für eingebettete Systeme bieten (sowie deren performante Umsetzung), variieren erheblich zwischen verschiedenen Betriebssystemen. Für unseren Container gehen wir davon aus, dass ein Betriebssystem zur Verfügung steht, welches sich zumindest um das Scheduling kümmert. Speicherschutz sowie andere weiterführende Features sehen wir als optional an.

- **Echtzeitfähigkeit:** Die Frage, ob ein System echtzeitfähig[58] ist, hängt von vielen Faktoren ab; im Falle von verteilten Systemen vor allem auch vom verwendeten Bus (FlexRay ist beispielsweise echtzeitfähig), vom Scheduler des Betriebssystems und von der Anwendungsfunktionalität. Wir beschränken uns hier auf Systeme, die keine harten Echtzeitanforderungen erfüllen müssen.
- **Sicherheitskritisch:** Für viele Anwendungsgebiete im sicherheitskritischen Umfeld sind verschiedenste Zertifizierungen vonnöten. Zertifiziert werden dabei neben der Software meist auch Werkzeuge zur Erstellung der Software sowie der Entwicklungsprozess. Auch solche Systeme schließen wir aus.
- **Dynamische vs. Statische Konfiguration:** Komponenteninstanzen, ihre Kommunikationsbeziehungen sowie verschiedene andere Aspekte des Gesamtsystems können entweder dynamisch zur Laufzeit oder statisch zur Konfigurationszeit definiert werden. Aus Performance-Gründen legen wir uns hier auf den statischen Fall fest, dynamisch werden nur die Lebenszyklus-Stati verwaltet.

Wie oben erkennbar wird, schließen wir für unsere Systeme gezielt bestimmte Szenarien aus. Dies hilft uns, den Fokus bei der Entwicklung der Systemfamilie zu behalten. Wir fokussieren die Software-Systemfamilie also auf Komponentencontainer mit folgenden Eigenschaften

- Nicht (hart) echtzeitfähig.
- Nicht sicherheitskritisch.
- Statisch konfiguriert.
- Auf einem Betriebssystem operierend, das mindestens das Scheduling übernimmt.

[58] In diesem Zusammenhang verstehen wir unter „echtzeitfähig“ die Tatsache, dass eine nicht eingehaltene Deadline einen Systemfehler darstellt (dies wird auch als harte Echtzeit bezeichnet).

17.2.2. Variabilitätsanalyse und Domänenstrukturierung

Nachdem wir die Domäne nun kennen, müssen wir die Merkmale dieser Systeme analysieren, um festzustellen, welche Features alle Mitglieder der Systemfamilie besitzen und welche variabel sind.

Zunächst ist es sinnvoll, die Domäne in Subdomänen zu unterteilen, wie in Kapitel <Ref PLE> angesprochen:

- Kommunikation über verschiedene Bussysteme und deren performante Abbildung auf die vom Komponentenmodell definierten Abstraktionen und Mechanismen.
- Container Services – zusätzliche Dienste, die der Container zur Laufzeit für die Komponenten erbringen kann.
- Deployment und Konsistenzchecks – optimale und konsistente Verteilung der Komponenten über die Knoten zu erreichen.
- Integration des Scheduling mit dem Betriebssystem.

Für jede dieser Subdomänen sollte man nun eine Variabilitätsanalyse durchführen. Beispielhaft werden wir uns auf zwei Subdomänen konzentrieren: Kommunikationsmechanismen sowie querschnittliche Dienste des Containers.

Kommunikationsparadigmen

Konnektoren dienen dazu, die Kommunikationsbeziehung zwischen zwei Ports von Komponenten zu beschreiben, wie in Abb. 130 gezeigt.

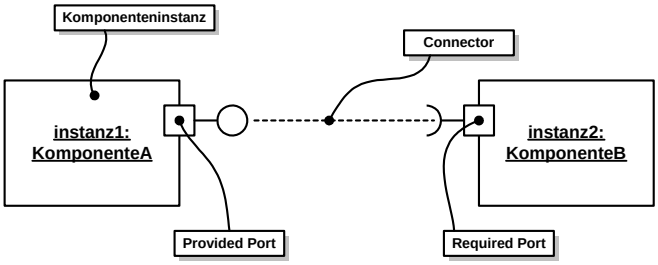


Abb. 130 *Komponenteninstanzen, Ports und Konnektoren*

Die Art und Weise, wie kommuniziert werden soll (Operationsaufruf, nachrichtenorientiert), wird also durch das Interface der beiden Ports sowie insbesondere den Konnektor bestimmt. Das Feature-Modell in Abb. 131 zeigt, welche Möglichkeiten es bei der Kommunikation gibt:

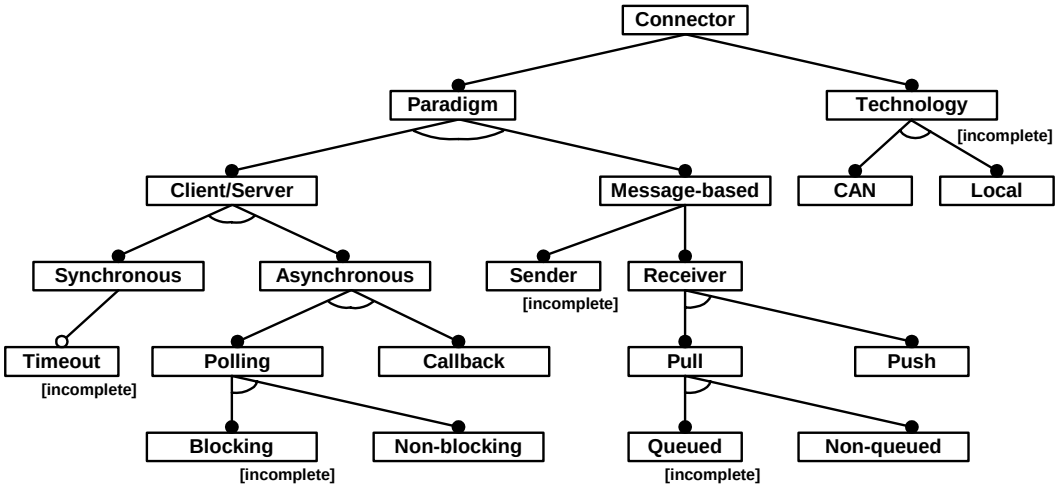


Abb. 131 *Feature-Modell der Kommunikationsattribute*

Erkennbar ist, dass man zunächst zwei Merkmale bestimmen muss: Die Technologie sowie das Paradigma.

Die Technologie hängt natürlich von den Randbedingungen ab: Man kann nur über die Bussysteme kommunizieren, die am betreffenden Knoten angeschlossen sind – auch kann man nur lokal kommunizieren, wenn die beiden Instanzen im gleichen Container liegen (ein Container definiert einen Adressraum).

Das Paradigma ist schon etwas diffiziler. Man kann prinzipiell entweder per Client/Server oder nachrichtenbasiert kommunizieren. Im Falle von Client/Server muss man festlegen, ob synchron oder asynchron kommuniziert werden soll. Im asynchronen Fall gilt es zu entscheiden, wie man über das Ergebnis des Aufrufs informiert werden möchte. Auch im Falle von nachrichtenbasierter Kommunikation muss man diese und weitere Entscheidungen treffen.

Wichtig ist es nun, zu bestimmen, wann der Entwickler welches Feature festlegen muss. Dies kann man beispielsweise durch eine entsprechende Annotation des Feature-Diagramms durchführen, wie in Abb. 132 gezeigt.

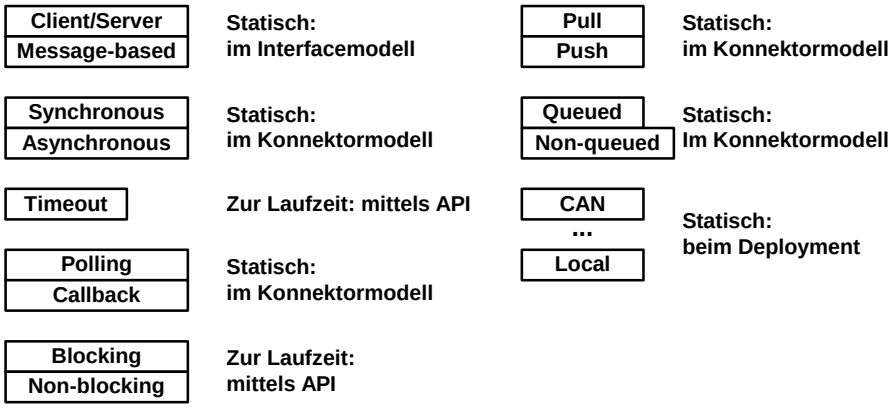


Abb. 132 *Binding Times für die verschiedenen Features*

Der Grund, warum hier sehr viele Merkmale statisch festgelegt werden müssen, ist wieder im Zusammenhang mit Codegröße, Speicherverbrauch, Determinismus und Performance zu sehen.

Eine letzte Anmerkung: Die Frage, ob blockierend oder nicht-blockierend gepollt werden soll, muss auf einigen Betriebssystemen (z.B. Osek) statisch entschieden werden, damit das Betriebssystem so konfiguriert werden kann, dass Blockierung überhaupt möglich ist[59].

[59] BCC vs. ECC – für die Osek-Insider

Container-Services

Ein weiterer wichtiger Aspekt der Software-Systemfamilie „Komponenten-Container“ sind die Dienste, die der Container anbietet (vgl. Abb. 133).

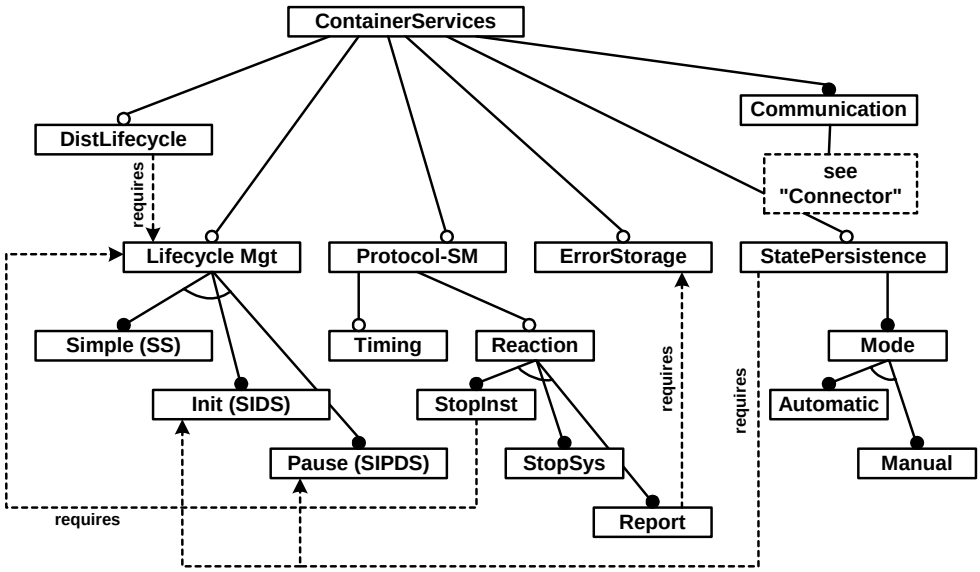


Abb. 133*Feature-Modell für die Dienste des Containers*

Zunächst bieten Container immer Kommunikationsdienste an, wie sie im letzten Abschnitt diskutiert wurden. Zusätzlich kann sich der Container um das Lifecycle-Management der Komponenteninstanzen kümmern, indem er sie kontrolliert aktiviert und deaktiviert. Dabei gibt es drei Modi:

- *Simple* kennt nur die Ereignisse *Start* und *Stop*.
- *Init* kennt zusätzlich eine Initialisierungsphase, also *Start, Init, Deinit, Stop.*
- *Pause* kennt des Weiteren noch einen *Pause*-Modus (z.B. zum Energiesparen).

Weiterhin kann der Container an Interfaces Protokollzustandsautomaten überwachen. Dabei gibt es zwei Dinge festzulegen: Erstens, ob auch Zeit-Constraints mit überwacht werden sollen, und zweitens, wie auf einen Fehler reagiert werden soll:

- *StopInst* stoppt die betreffende Instanz (dazu ist Lifecycle-Management nötig),

- *StopSys* stoppt den ganzen Container.
- *Report* legt den Fehler einfach in einem zentralen Fehlerspeicher ab.

Für Letzteres muss der Container allerdings das Feature Fehlerspeicher aktiviert haben.

Der Container kann sich außerdem um das Management des Zustands der Komponenteninstanzen kümmern. Wenn dieses Feature aktiviert ist, kann der Container entweder automatisch oder nach API-Aufruf des Entwicklers den Zustand der Komponente sichern. Auch hierfür muss Lifecycle-Management aktiviert sein.

Schlussendlich kann der Container noch bei verteiltem, also systemweiten Lifecycle-Management involviert sein. Dazu ist allerdings das Lifecycle-Management innerhalb des Containers nötig.

Die Frage, wie die entsprechenden Features gebunden werden, ist hier leicht zu beantworten: Grundsätzlich statisch. Allerdings gibt es Unterschiede im Detail:

- Beispielsweise muss der Lifecycle-Modus bereits bei der Implementierung der Komponenten bekannt sein, zumal im Implementierungscode auf die Lifecycle-Ereignisse reagiert werden muss.
- Die Frage, ob Protokollzustandsautomaten zur Laufzeit überwacht werden sollen, kann zur Generierungszeit des Containers entschieden werden.

Wir werden darauf bei der Definition des Produktionsplans nochmals eingehen.

17.2.3. Domänendesign

Im Rahmen des Domänendesigns wird die Zielarchitektur festgelegt – also die gemeinsame Basisarchitektur der verschiedenen Mitglieder der Software-Systemfamilie. Außerdem wird der Produktionsprozess definiert, der besagt, wie der Weg vom Modell zum fertigen Produkt aussieht.

Zielarchitektur

Wir können hier nicht auf jedes Detail eingehen, das würde den Rahmen der Fallstudie sprengen. Abb. 134 gibt einen Überblick:

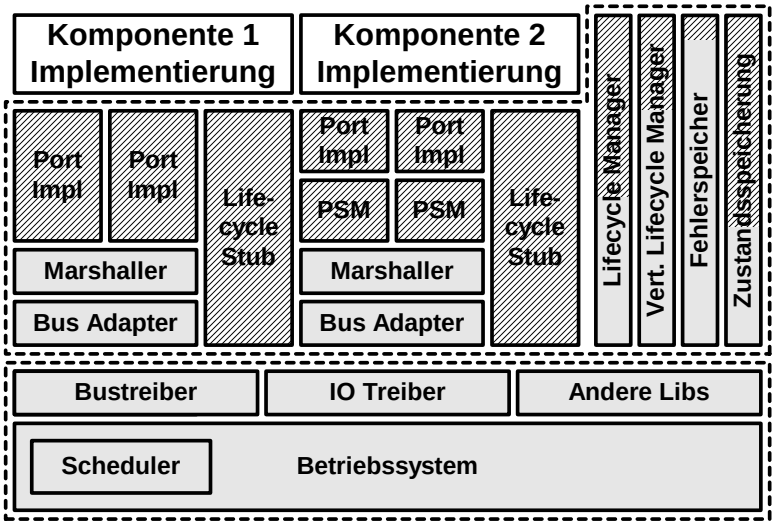


Abb. 134 *Familienarchitektur der zu erstellenden Middleware*

Die grau hinterlegten Teile bilden die MDSD-Plattform, die schraffierten Teile sind generiert, und die weißen Teile stellen die manuellen Implementierungen der Komponenten dar.

Natürlich wollen wir so wenig wie möglich neu erfinden, weshalb von den Möglichkeiten des Betriebssystems intensiv Gebrauch gemacht wird. Der untere, gestrichelt umrandete Kasten umfasst das Betriebssystem sowie gewisse Basisbibliotheken und Treiber. Der obere, gestrichelte Kasten ist in Summe der Container.

Man beachte, dass die hier dargestellten Dinge nicht alle Artefakte umfassen, die benötigt bzw. generiert werden. Beispielsweise brauchen wir generierte Makefiles bzw. generierte Konfigurationsdateien. Diese sind jedoch für die Zielarchitektur des fertigen Systems nicht mehr relevant.

Produktionsprozess

Der Produktionsprozess beschreibt, wie man von den Modellen zum ausführbaren System kommt. Abb. 135 zeigt (vereinfacht) diesen Prozess für unsere Fallstudie. Die einzelnen Schritte werden im Folgenden erläutert. Wir gehen dabei nicht näher auf die Rollen in diesem Prozess ein. In der Grafik bedeuten die nicht-gestrichelten Pfeile zeitlichen Ablauf und die gestrichelten eine Abhängigkeits- bzw. „verwendet“-Beziehung.

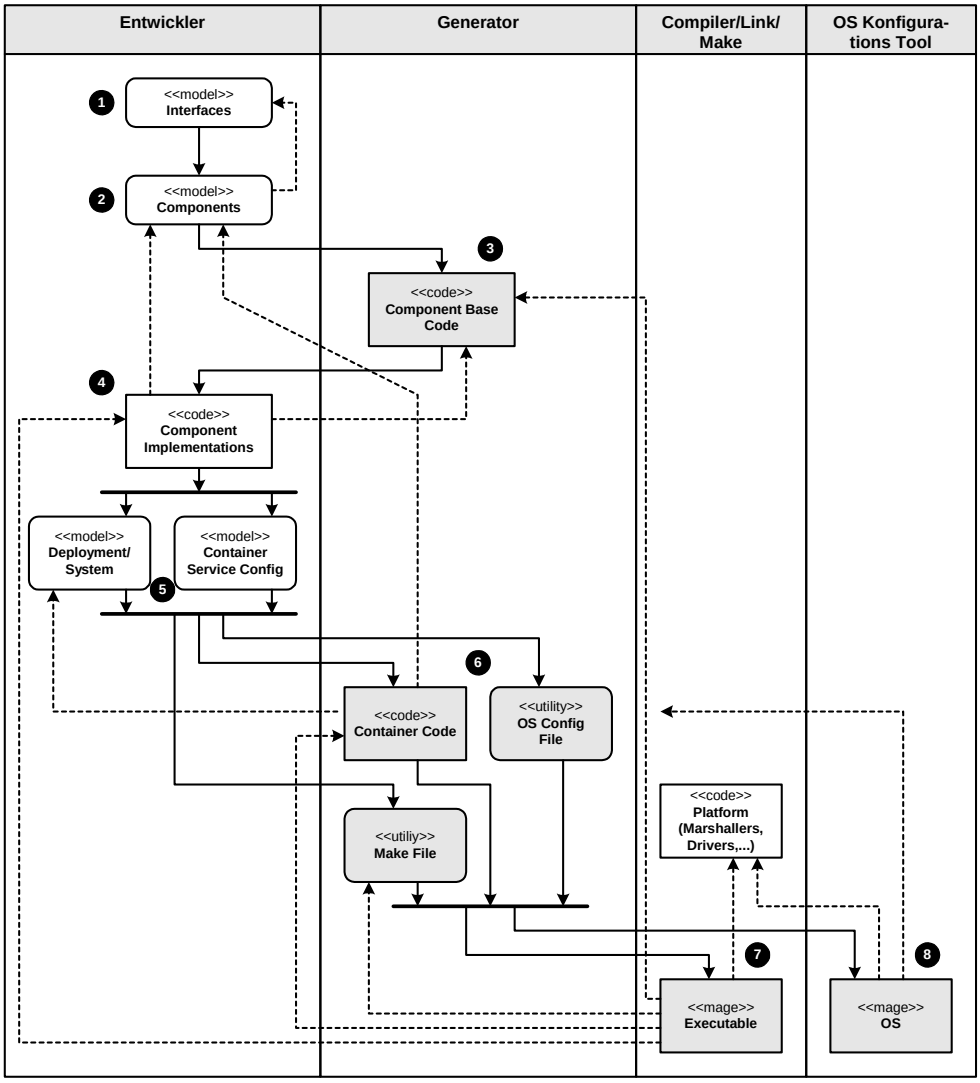


Abb. 135 *Produktionsprozess der Middleware-Systemfamilie*

Hier einige Erläuterungen zu diesem Prozess – Details werden im Verlauf der Fallstudie noch geklärt.

1. Im ersten Schritt werden die Interfaces modelliert. Interfaces stellen ja später dann die Basis für die Kommunikation zwischen Komponenten(instanzen) dar.

2. Im zweiten Schritt werden dann unter Verwendung dieser Interfaces die Komponenten definiert, insbesondere deren Ports.

3. Aus diesen beiden Modellen kann der Generator nun Komponenten-Basiscode erstellen, also beispielsweise C-Headerfiles.

4. Auf Basis des generierten Basiscodes kann der Entwickler nun die eigentliche Komponentenimplementierung erstellen.

5. In diesem Schritt kann der Entwickler das Gesamtsystem definieren, also Komponenteninstanzen, die Konnektoren zwischen diesen sowie die Verteilung der Instanzen auf die Hardware (die hier auch beschrieben werden muss). Außerdem kann nun das Container-Service-Konfigurationsmodell erstellt werden. Es definiert, welche der weiter oben beschriebenen zusätzlichen Dienste der Container für das betreffende System tatsächlich erbringen soll.

6. Dies ist der eigentliche Generierungsschritt. Der Generator erstellt den Code, der zur Implementierung der geforderten Containerfunktionalität nötig ist. Dazu benötigt er sämtliche bisher definierten Modelle. Der Generator erstellt auch eine Konfigurationsdatei für das betreffende Betriebssystem sowie ein Makefile.

7. In diesem Schritt können Compiler, Linker und Make das Image des „Programms“ erstellen, bestehend aus dem Container, dem Komponentenbasiscode, den Komponentenimplementierungen sowie den nötigen Bibliotheken aus der Plattform.

8. Im letzten Schritt wird das Betriebssystem aus der betreffenden Konfigurationsdatei erstellt. Dies erledigt ein zum Betriebssystem gehöriges Werkzeug.

Wir können nun Betriebssystem und Anwendung (d.h. Container und Komponenten) auf die Zielhardware übertragen und ausführen.

17.2.4. Domänenimplementierung

Die letzte Phase des Product Line Engineerings befasst sich mit der Implementierung der bisher erläuterten Konzepte. Dies stellt den Hauptteil unserer Fallstudie dar und wird im Folgenden erläutert.

17.3. Modellierung

Wir wollen zeigen, wie in den verschiedenen Subdomänen des Systems modelliert wird. Dabei wird für jede der drei Subdomänen eine eigene DSL verwendet. Wie alle DSLs bestehen auch diese aus

- dem Metamodell, welches die abstrakte Syntax und die statische Semantik der Modelle definiert,
- der konkreten Syntax, welche die Notation festlegt, mit der die Modelle niedergeschrieben werden, und
- der (dynamischen) Semantik, welche die Bedeutung der Metamodellelemente definiert.

Die Metamodelle werden in Java implementiert, wie es der *openArchitectureWare*-Generator erfordert. Sie werden mittels Gateway- und Proxymetaklassen aneinander gekoppelt.

Die konkrete Syntax der DSL ist in allen drei Fällen unterschiedlich. Es kommt sowohl ein UML-Profil zum Einsatz als auch eine (an IDL angelehnte) textuelle Notation sowie eine spezielle XML-DTD.

In allen drei Fällen wird die Semantik der DSL über die Transformationen implementiert, die der Generator verwendet, um Implementierungscode zu generieren.

17.3.1. Definition von Interfaces

Zunächst möchten wir auf die Definition von Komponenten eingehen. Eine Komponente besteht aus einer Reihe von Ports, die entweder die Dienste eines Interfaces anbieten oder diese verwenden. Abb. 136 zeigt das Metamodell für Interfaces:

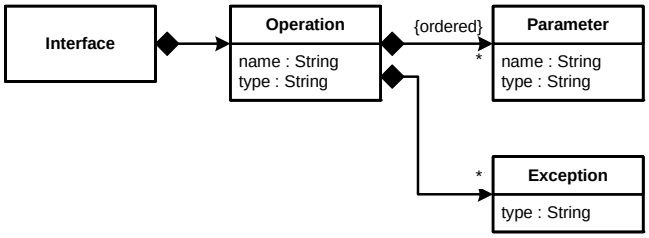


***Abb. 136** Metamodell für Interfaces*

Die konkrete Syntax wird textuell sein, zumal grafische Notationen für diesen Zweck nicht produktiver sind.

Beginnen wir mit der Definition eines Interfaces:

```
interface Sensor {
  operation start():void;
  operation stop():void;
  operation measure():float;
}
interface Controller {
  operation reportProblem(Sensor s,
            String errorDesc ):void;
}
```

Zusätzlich zu dieser syntaktischen Definition von Interfaces können beispielsweise noch Protokollzustandsautomaten modelliert werden – z.B. mittels UML. Diese definieren, in welcher Reihenfolge die Operationen eines Interfaces aufgerufen werden dürfen. Aus diesem Automat kann leicht Code generiert werden, welcher zur Laufzeit überwacht, ob die damit definierten Constraints eingehalten werden. Wenn nicht, kann z.B. ein Fehler in einen Fehlerspeicher abgelegt werden, welcher später zur Diagnose herangezogen werden kann. Abb. 137 zeigt ein Beispiel eines Protokollzustandsautomaten.

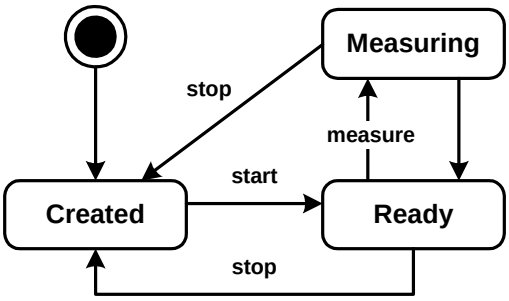


Abb. 137*Beispiel für eine Protokollzustandsmaschine*

Das Metamodell hierfür entspricht dem wohlbekannten Metamodell für Zustandsmaschinen, bestehend aus Zuständen und Transitionen inkl. Guards und Events. Hier sind die Events die Operationen, die auf dem Interface aufgerufen werden können. Auf die Frage, wie das Interface zu seinem Protokollzustandsautomat „findet“ und wie man den Laufzeitüberwachungscode generiert, gehen wir später ein.

17.3.2.Definition von Komponenten und Ports

Im nächsten Schritt modellieren wir Komponenten und ihre Ports. Das dafür verwendete Metamodell ist in Abb. 138 dargestellt.

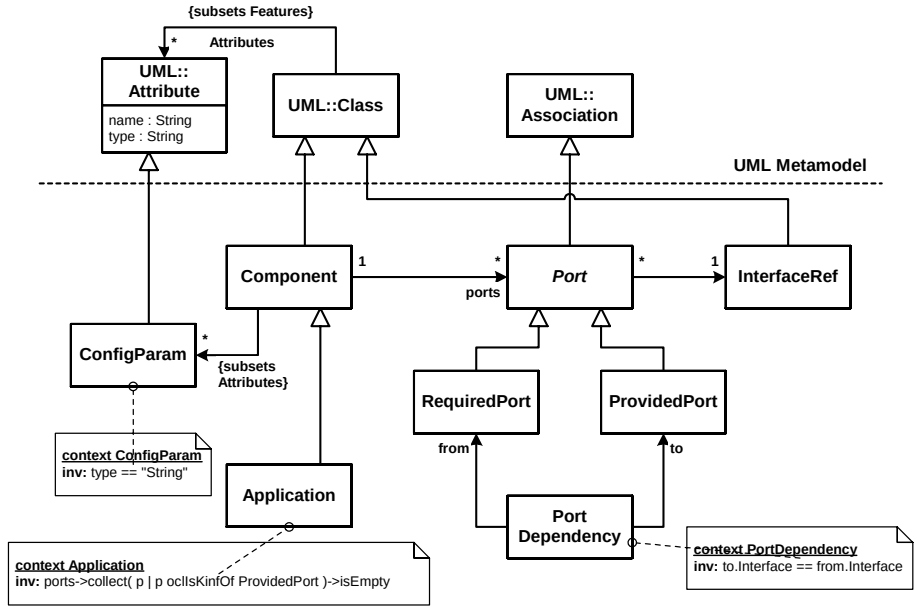


Abb. 138 *Metamodell für Komponenten, Ports und Interfaces, angebunden an das Metamodell der UML*

Da die konkrete Syntax ein UML-Profil sein wird, verwenden wir das UML-Metamodell als Basis. Interessant ist vor allem das Metamodellelement *InterfaceRef* – es dient als Referenz auf die in anderen Modellen definierten Interfaces. Die Zuordnung der Referenz zum definierenden Vorkommen basiert auf dem Namen. Abb. 139 ist die auf UML basierende, konkrete Syntax eines Beispiels:

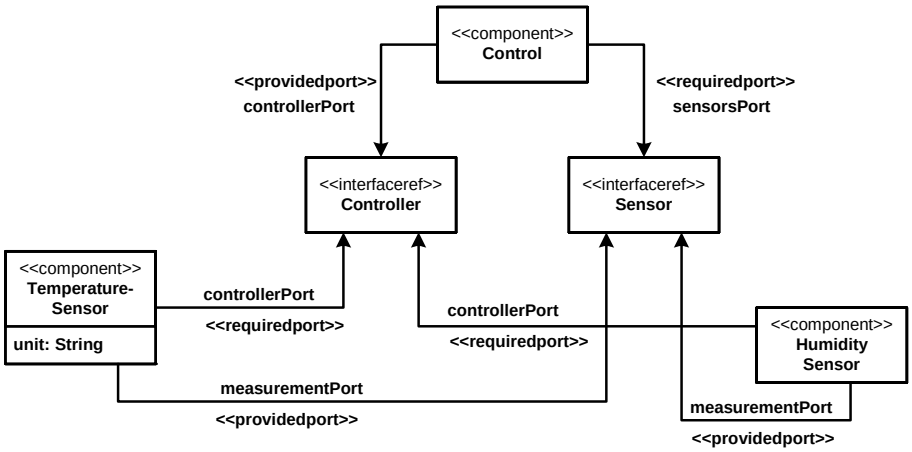


Abb. 139 *Beispielhaftes Komponentenmodell*

17.3.3. Definition eines Systems

Ein System besteht aus einer Menge von Knoten. Auf jedem Knoten sind ein oder mehrere Container aktiv. Ein Container beinhaltet eine oder mehrere Komponenteninstanzen. Die Ports von Komponenteninstanzen werden mit so genannten Konnektoren verbunden. Wie aus dem Metamodell ersichtlich, gibt es verschiedene Arten von Konnektoren. Abb. 140 zeigt dieses Metamodell.

Auch hier arbeiten wir wieder mit Referenzen, um Elemente, die in anderen Modellen definiert sind, zu referenzieren.

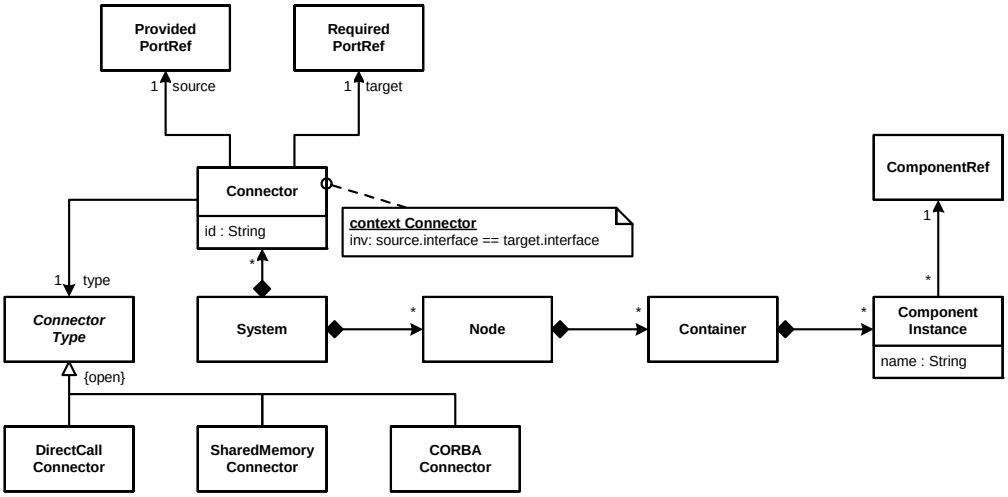


Abb. 140 *Metamodell für Systeme, Container und Konnektoren*

Damit ist das Metamodell (natürlich nicht mit allen Constraints) definiert. Die konkrete Syntax ist in diesem Falle ein zu diesem Zweck entworfenes XML Schema. Im ersten Schritt definieren wir zunächst die Komponenteninstanzen, Systemknoten und Container.

```
<system name=”weatherStation”>
  <node name=”main”>
    <container name=”main”>
      <instance name=”controller”
                type=”Control”/>
    </container>
  </node>
  <node name=”inside”>
    <container name=”sensorInside”>
      <instance name=”tempInside”
                type=”TemperatureSensor”>´
        <param name=”unit” value=”centigrade”/>
      </instance>
    </container>
  </node>
```

```
  <node name=”outside”>
    <container name=”sensorsOutside”>
      <instance name=”tempOutside”
                type=”TemperatureSensor”>´
        <param name=”unit” value=”centigrade”/>
      </instance>
      <instance name=”humOutside”
                type=”HumiditySensor”/>
    </container>
  </node>
```

Anschließend beschreiben wir die Verbindungen zwischen den Komponenteninstanzen in Form von Konnektoren.

```
  <!-- temperature sensor outside -->
  <connector name="toSensorTempOutside">
    <providedPort instance="tempOutside"
                  port="measurementPort">
    <requiredPort instance="controller"
                  port="sensorsPort">
  </connector>
  <connector name="fromSensorTempOutside">
    <providedPort instance="controller"
                  port="controllerPort">
    <requiredPort instance="tempOutside"
                  port="controllerPort>
  </connector>

  <!-- humidity sensor outside -->
  <connector name="toSensorHumOutside">
    <providedPort instance="humOutside"
                  port="measurementPort">
    <requiredPort instance="controller"
                  port="sensorsPort">
  </connector>
  <connector name="fromSensorHumOutside">
    <providedPort instance="controller"
                  port="controllerPort">
    <requiredPort instance="humOutside"
                  port="controllerPort>
  </connector>

  <!-- temperature sensor inside -->
  <connector name="toSensorTempInside">
    <providedPort instance="tempInside"
                  port="measurementPort">
    <requiredPort instance="controller"
                  port="sensorsPort">
  </connector>
  <connector name="fromSensorTempInside">
    <providedPort instance="controller"
                  port="controllerPort">
    <requiredPort instance="tempInside"
                  port="controllerPort>
  </connector>
</system>
```

Dies legt die logische Struktur des Systems fest. Es fehlt noch die genaue Charakterisierung der Eigenschaften der Konnektoren. Hier kommt das

eingangs definierte Feature-Modell ins Spiel. Die folgenden beiden Konfigurationen wären bezüglich des Feature-Modells gültig:

```
connector.technology=CAN
connector.paradigm=async-cs,polling,blocking

connector.technology=local
connector.paradigm=sync
```

Solche Konfigurationseinstellungen müssen bei der obigen Konnektor-Definition zusätzlich angegeben werden.

Dabei wird ein Constraint erkennbar: Ein lokaler Konnektor kann nur zwischen zwei Instanzen definiert werden, wenn diese im gleichen Container aktiv sind. Der Generator muss dies prüfen und ggf. entsprechende Fehlermeldungen ausgeben.

17.3.4.Gesamtmodell

Das Gesamt-Metamodell besteht also aus der Obermenge der Metamodelle der verschiedenen Subdomänen. Abb. 141 zeigt die Referenzklassen (Proxy-Elemente), die die Beziehung zwischen den verschiedenen Subdomänen herstellen.

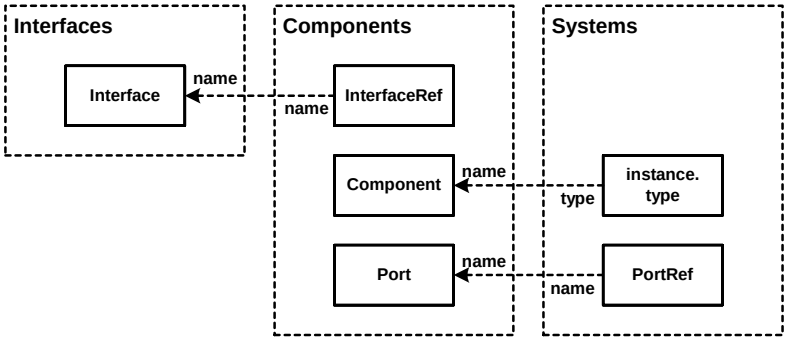


Abb. 141*Beziehungen der Metamodelle untereinander*

17.3.5.Verarbeitung

Abb. 142 zeigt nun, wie die verschiedenen Artefakte vom Generator verarbeitet werden.

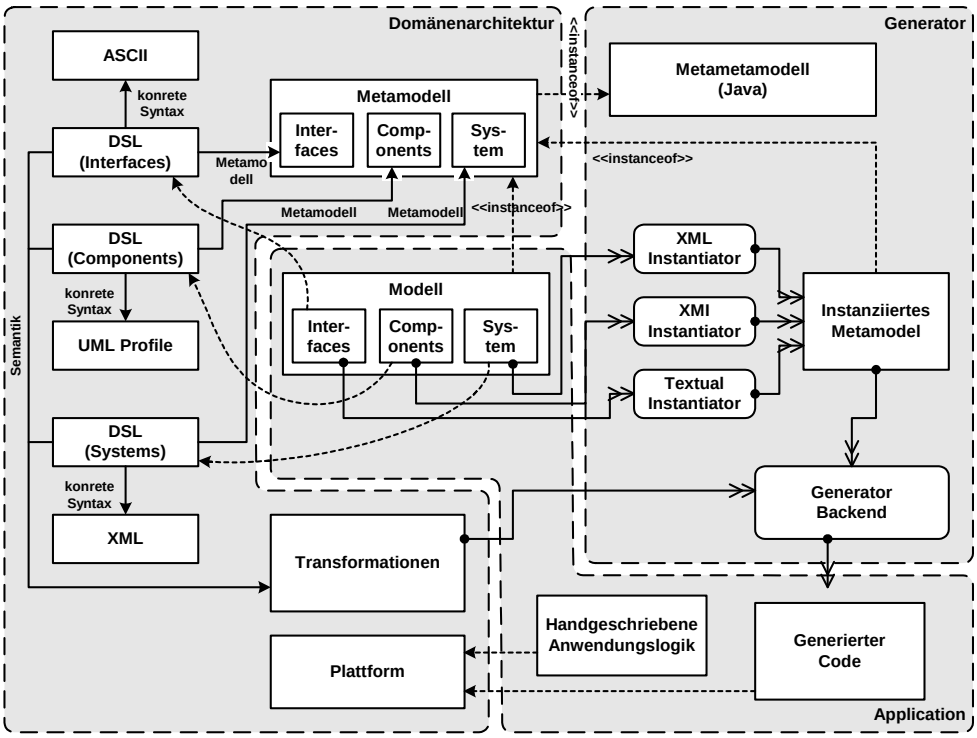


Abb. 142 *Überblick über den Generierungsprozess*

17.4. Implementierung von Komponenten

Dieser Abschnitt zeigt, wie das Konzept der „Komponente“ auf Programmiersprachen abgebildet wird, und wie der Container mit den Komponenten interagiert. Wir gehen dabei der Einfachheit halber ausschließlich auf Java ein. Wir sind uns natürlich im Klaren, dass für „echte“ eingebettete Systeme das C-Mapping relevanter ist. Allerdings ist die Abbildung auf Java einfacher zu verstehen und findet zum Beispiel im Bereich von Mobiltelefonen auch in ähnlicher Weise Anwendung.

17.4.1. Abbildung auf Java

Aufgrund der Tatsache, dass Java eine objektorientierte Sprache ist, ist die Abbildung ziemlich direkt möglich und sieht grob folgendermaßen aus (Abb. 143 zeigt ein Beispiel):

- Ein (Client/Server-)Komponenten-Interface wird auf ein Java Interface abgebildet.
- Pro Komponente wird eine abstrakte Basisklasse generiert. Von diesen leitet der Entwickler eigene Klassen ab, welche die von der Komponente angebotenen Operationen implementieren.

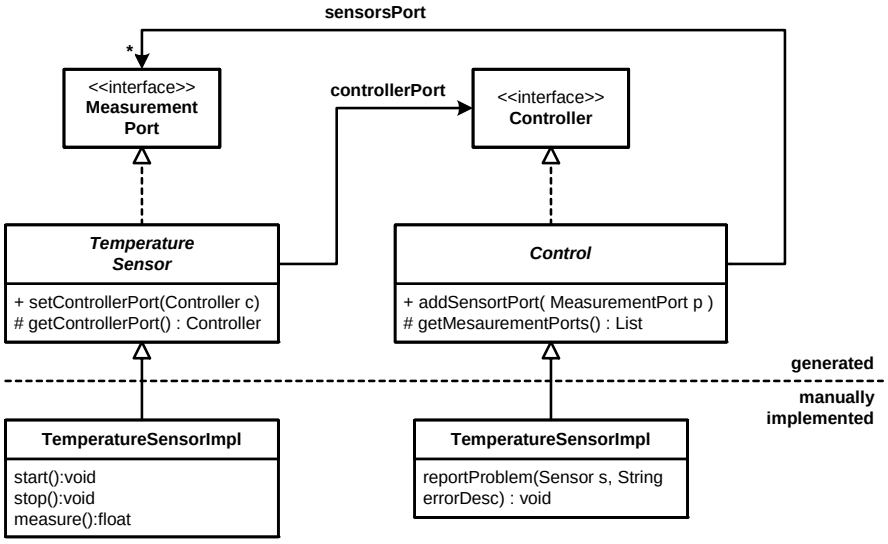


Abb. 143*Klassenmodell der Implementierung in Java (lokal)*

Der Container ist dafür verantwortlich, die Referenzen der Ports beim Hochfahren des Systems richtig zu setzen. Der Entwickler kann dann im Rahmen der Implementierung auf diese Ports zugreifen:

```
public class TemperatureSensorImpl
             extends TemperatureSensor {
  public void start() { … }
  public void stop() { … }
  public float measure() {
    float value = // use driver to measure
    if ( /* there is a problem */ ) {
      getControllerPort().reportProblem(this,
        “cannot measure….” );
    }
  }
}
```

Damit ist generierter und nicht-generierter Code komplett getrennt. Obiges Beispiel funktioniert bereits in dieser Art und Weise, wenn die beiden Komponenten im gleichen Container (also auch im gleichen Prozess der gleichen VM) aktiviert sind. Als *RequiredPort*-Objekte werden direkt die jeweils „gegenüberliegenden“ Komponentenobjekte

gesetzt. Im Falle von verteilten Komponenten müssen automatisch Proxys generiert werden, wie Abb. 144 zeigt.

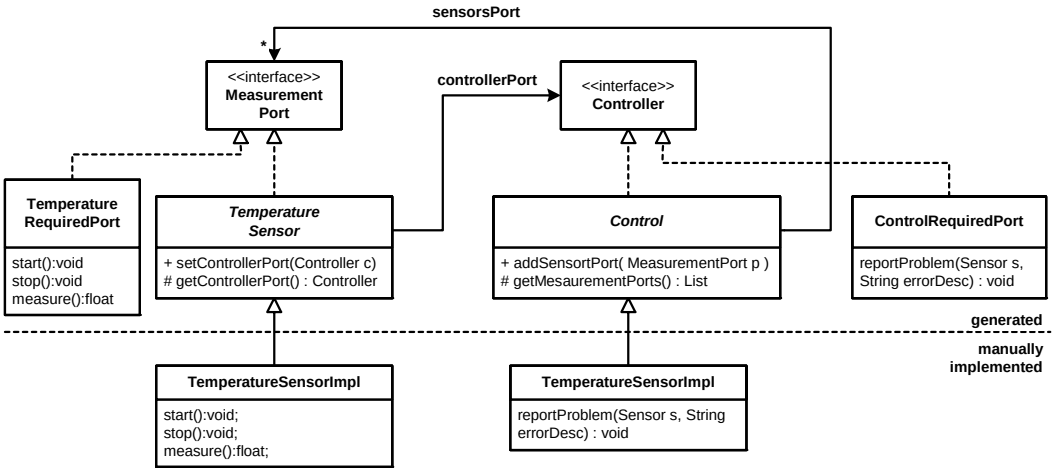


Abb. 144*Klassenmodell der Implementierung in Java (remote)*

Diese Proxys werden nur generiert, wenn tatsächlich verteilte Kommunikation stattfindet. Die Proxys enthalten dann auch den betreffenden Code, um das ausgewählte Kommunikationsparadigma zu realisieren, also Marshalling, Verbindungsmanagement etc. Natürlich wird möglichst viel davon in Form von Bibliotheken innerhalb der Plattform implementiert.

Da die Proxys das gleiche Interface implementieren wie die dienstanbietende Komponente, ändert sich die Komponentenimplementierung im Falle verteilter Kommunikation nicht – der Container setzt eben den Proxy statt direkt die entsprechende Komponente.

17.5. Generatoranpassung

Der Generator muss zur Bewältigung der geschilderten Aufgaben angepasst werden. Unter anderem fällt an:

- Das Parsen der textuellen Syntax
- Das Parsen der XML-Konfiguration
- Die Zusammenführung der verschiedenen Teilmodelle und das Auflösen der Referenzen

- Die Validierung des Gesamtmodells
- Die Generierung der verschiedenen Implementierungen

Wir werden auf jede dieser Anforderungen kurz eingehen.

17.5.1. Parsen der textuellen Syntax

Wir verwenden den Parsergenerator JavaCC [JCC], gehen aber nicht auf die Details ein, sondern betrachten im Wesentlichen nur die Integration in den Generator. Statt JavaCC könnte auch ein anderer Parsergenerator verwendet werden.

Wir beginnen also mit dem Parsen von Interface-Definitionen der folgenden Art:

```
interface Sensor {
  operation start():void;
  operation stop():void;
  operation measure(): float;
}
```

Die Aufgabe von JavaCC ist dabei das Parsen sowie das Erstellen des Abstrakten Syntaxbaumes (AST). Dieser ist letztendlich einfach ein Objektbaum, wobei für jeden Syntaxknoten ein entsprechendes Objekt verwendet wird. Um die verschiedenen Syntaxelemente zu repräsentieren, werden verschiedene AST-Klassen verwendet. Diese entsprechen den Metaklassen im Kontext des *openArchitectureWare*-Generators. Da dieser erwartet, dass alle Metaklassen von der Klasse *ModelElement* erben, müssen wir dafür sorgen, dass die von JavaCC verwendete Basisklasse der AST-Knoten ihrerseits von *ModelElement* erbt. Damit sind dann alle AST-Klassen Modellelemente.

Wir modifizieren also die Klasse *SimpleNode* (die standardmäßig bei JavaCC mitgeliefert wird und die Basisklasse für alle AST-Knoten darstellt) folgendermaßen:

```
public class SimpleNode
   extends ModelElement implements Node {

  public SimpleNode(int i) {
    id = i;
    JCCHelper.getMetaEnvironment().
              addElement( this );
    setName("");
  }

  // rest as before…

}
```

Zum einen erbt *SimpleNode* von der vom Generator stammenden Klasse *ModelElement*. Zum anderen fügt der Konstruktor das aktuelle Element zum *MetaEnvironment* des Generators hinzu. Ohne diese Zeile würde das Ganze nicht funktionieren, weil der Generator das Objekt ansonsten nicht „kennt".

Angenommen, wir haben den Parser (*InterfaceParser.java*) aus einer entsprechenden Syntaxdefinition generiert, so müssen wir nun als Nächstes ein entsprechendes Frontend für den Generator implementieren. Generator-Frontends dienen dazu, die konkrete Syntax von Modellen zu lesen und eine Menge von *ModelElements* zu liefern, die dann im Generator zu Laufzeit vorliegen und als Basis für die Generierung dienen. Frontends müssen das *InstantiatorInterface* implementieren. Die Implementierung des Frontends delegiert das Parsen an den von JavaCC generierten Parser. Das Frontend sieht damit für unseren Fall folgendermaßen aus:

```
public class JCCInstantiator
   implements InstantiatorInterface {

  public ElementSet loadDesign(
              InstantiatorEnvironment env)
              throws ConfigurationException,
              InstantiatorException {
     JCCHelper.setMetaEnvironment(
               env.getMetaEnvironment() );
     ElementSet result = new ElementSet();
     String spec = // read specification from somewhere…
     StringBufferInputStream s =
               new StringBufferInputStream( spec );
     InterfaceParser p = new InterfaceParser(s);
     try {
        ASTStart start = p.Start();
        ASTClass cls = (ASTClass)start.Child().get(0);
        result.add( cls );
     } catch ( Exception ex ) {
        ex.printStackTrace();
     }
     return result;
   }
}
```

Hier sind zwei Dinge zu beachten: Zum einen wird in der ersten Zeile der Operation das aktuelle *MetaEnvironment* in den *JCCHelper* gespeichert – wie oben gezeigt, holen die AST-Klassen es dort in ihrem Konstruktor wieder heraus. Im *try*-Block wird nun die *ASTClass*-Instanz herausgeholt und als einziges Element in das *result* dieser Operation geschrieben. Der Instantiator liefert also als Toplevel-Element des Modells die Interface-Deklaration zurück.

Damit ist die Integration in den Generator fast fertig. Wir müssen nun lediglich noch dem Generator mitteilen, dass er diesen Instantiator

verwenden soll, statt des Standard-XMI-Frontends. Dazu muss im Generator folgendes Property gesetzt werden[60]:

```
System.setProperty("de.bmiag.genfw.instantiator.class",
                   "util.JCCInstantiator");
```

17.5.2. Parsen des Systemdefinitions-XML

Um die XML-Konfigurationen zu parsen, verwenden wir einen generischen XML-Parser, welcher aus beliebigen XML-Dokumenten Modelle im Sinne des *openArchitectureWare*-Generators erstellt. Dabei ist die Idee, mittels des XML-Dokumentes quasi direkt den AST, also die Modellstruktur abzubilden. Der Parser muss dann nur noch Java-Objekte der gleichen Struktur erstellen. Das Integrationsprinzip zwischen Parser und Generator ist dabei das gleiche wie das soeben gezeigte.

Der Parser erwartet dabei, dass es in einem bestimmten (vom Entwickler frei wählbaren) Package Metaklassen gibt, die genauso heißen wie die betreffenden XML-Elemente. Betrachten wir dazu das folgende Beispiel:

```
<system name=”weatherStation”>
  <node name=”main”>
    <container name=”main”>
      <instance name=”controller”
                type=”Control”/>
    </container>
  </node>
</system>
```

Der Generator wird bei Lesen dieses XML-Files zunächst versuchen, eine Klasse *System* zu instanziieren. Auf dem betreffenden Objekt ruft er dann eine Method *setName()* auf und gibt dieser das Argument „*weatherStation*“ mit. Danach instanziiert der Parser eine Klasse *Node*, setzt deren Name und versucht dann, auf dem vorher erzeugten System-Objekt die Methode *addNode()* aufzurufen; das gerade eben erstellte *Node*-Objekt dient als Argument.

Mittels dieses Mechanismus kann der Parser beliebige XML-Strukturen instanziieren, es müssen lediglich die zur XML-Struktur passenden Metaklassen vorhanden sein.

---

[60] Inzwischen bietet openArchitectureWare komfortablere Möglichkeiten, Frontends zu konfigurieren. Siehe Dokumentation unter www.openarchitectureware.org

Der *openArchitectureWare*-Generator bietet bereits ein entsprechendes Frontend, welches die hier geschilderten Aufgaben erledigt.

17.5.3. Parsen und Zusammenführen des Gesamtmodells

Das Gesamtmodell macht nur Sinn, wenn die Modelle aller drei Technischen Subdomänen gemeinsam gesehen werden. Der Generator muss also alle drei Modelle parsen und entsprechend zusammenführen. Dabei müssen die Referenzobjekte (z.B. die *InterfaceRefs*) mit den referenzierten Objekten (also das Interface mit demselben Namen) verbunden werden. Abb. 141 zeigt die Zusammenhänge.

Parsen der verschiedenen Modelle

Um die verschiedenen Modelle zu parsen, erstellen wir ein Frontend, welches wiederum andere Frontends verwendet, um die Modelle zu parsen. Ein solches Frontend ist bereits im openArchitectureWare-Generator vorhanden und heißt *CompositeInstantiator* (weil es das Composite Pattern aus [GHJ+94] implementiert). Wir leiten davon ab, um unsere verschiedenen Frontends zu konfigurieren.

```
package util;

public class EmbeddedComponentsInstantiator
             extends CompositeInstantiator {

   private String systemConfFile =
       System.getProperty("EC.SYSTEM");
   private String interfaceFile =
       System.getProperty("EC.INTERFACE");
   private String componentsFile =
       System.getProperty("EC.COMPONENTS");

   public EmbeddedComponentsInstantiator () {

     // a frontend that reads the UML model
     add( new XMIInstantiator( componentsFile ) );

     // a frontend that reads the XML system spec
     // use ecMetamodel as package prefix when
     // attempting to load metamodel classes
     add( new XMLInstantiator( systemFile,
          "ecMetamodel" ) );

     // a frontend that reads the textual spec
     // for the interfaces
     add( new JCCInstantiator( interfaceFile ) );
   }
}
```

Damit der Generator diesen Instantiator auch verwendet, muss er wiederum über das betreffende System Property konfiguriert werden.

```
System.setProperty("de.bmiag.genfw.instantiator.class",
                   "util.EmbeddedComponentsInstantiator");
```

Zusammenführen der Modelle

Der *CompositeInstantiator*, der oben zum Einsatz kommt, sorgt automatisch dafür, dass die Modellelemente aus den verschiedenen geparsten Modellen in einem Modell im Speicher des Generators (als Instanzen der betreffenden Java-Metaklassen) vorliegen. Nach Parsen der Modelle liegen also verschiedene Instanzen der Klasse *Interface* und der Klasse *InterfaceRef* vor. Im Folgenden ein Ausschnitt der Definition von *InterfaceRef*, der dafür sorgt, dass *InterfaceRefs* das betreffende *Interface* finden:

```
package cmMetamodel;

public class InterfaceRef extends Class {

  private Interface referencedInterface = null;

  public Interface Interface() {
    if ( referencedInterface == null ) {
      String myName = Name().toString();
      referencedInterface =
                  (Interface)MMUtil.findByName( this,
                  Interface.class, myName,
                  “Cannot find interface named “+myName );
    }
    return referencedInterface;
  }

}
```

Dieser Code bedarf der Erläuterung: Die Operation *Interface()* liefert das Interface zurück, auf welches die betreffende *InterfaceRef* verweist. Wie bereits erläutert, wird der Zusammenhang über den Namen hergestellt. Die Operation *MMUtil.findByName()* erledigt die ganze Arbeit für uns: Sie sucht im Modell, zu dem auch das aktuelle Objekt (*this*) gehört, nach allen Instanzen der Klasse *Interface*. In diesen wird dann das Interface mit dem übergebenen Namen gesucht. Wird genau eines gefunden, so wird dies zurückgegeben. Wird keines gefunden, wird ein *DesignError* mit der übergebenen Fehlermeldung erzeugt.

Man beachte dabei, dass die *InterfaceRef* ein Modellelement ist, welches aus dem UML-basierten Komponentenmodell entstanden ist, wohingegen das Interface an sich aus dem textuellen Interfacemodell stammt. Dieser Unterschied ist hier im Generator bereits nicht mehr zu

erkennen, da er ausschließlich auf dem instanziierten Metamodell (der abstrakten Syntax) operiert.

Mit diesem Ansatz werden alle Referenzen aufgelöst – auch die Verbindung zwischen Interface und Protokollzustandsautomat.

Partitionierung von Modellen

Wir haben nun die Frage beantwortet, wie man verschiedene Subdomänen (Interfaces, Komponenten, Deployment) des Systems mittels getrennter Modelle und unterschiedlichen konkreten Syntaxformen darstellen und später im Generator zusammenführen kann. Ein verwandtes Thema ist die bereits in Kapitel <Ref Vers> angesprochene Partitionierung von DSLs und Modellen. Hierfür können die gleichen Kopplungs- und Integrationsmechanismen auf Generator-Ebene verwendet werden, falls das Modellierungswerkzeug keine ausreichende Unterstützung für verteilte Modellierung bietet.

17.5.4. Pseudodeklarative Metamodellimplementierung

Constraint-Überprüfung

Letztendlich stellt das Auflösen der Referenzen einen von vielen Validierungsschritten dar. Ist das referenzierte Modellelement nicht vorhanden, so muss eine Fehlermeldung ausgegeben werden. Im Rahmen ernsthafter MDSD-Systeme gibt es eine Vielzahl solcher Constraints, die vom Generator überprüft werden müssen.

openArchitectureWare bietet dafür einen separaten Schritt im Ablauf des Generators. Nachdem das komplette Modell geparst wurde, und bevor die Codegenerierung beginnt, wird bei allen Modellelementen die Operation *CheckConstraints()* aufgerufen. In dieser Operation kann der Entwickler die metaklassenspezifischen Constraints (Modellinvarianten) unterbringen.

Dabei ist es aus unserer Sicht essenziell, dass man diese Constraints möglichst ausdrucksstark und knapp formulieren kann. Eine an deklarative Sprachen (wie z.B. OCL) angelehnte Syntax ist dabei sehr hilfreich. Der openArchitectureWare-Generator bringt mit den Klassen *Checks*, *Filters* und *MMUtil* eine Reihe solcher Hilfsfunktionen mit. Wir zeigen im Folgenden einige Beispiele.

Ein Modell-Constraint lautet, dass Komponenten keine eigenen Operationen definieren dürfen. Auch Superklassen oder implementierte Interfaces sind nicht erlaubt:

```
public class Component extends Class {
```

```
  public String CheckConstraints() {
     Checks.assertEmpty( this, Operation(),
        "must not have attributes." );
    Checks.assertEmpty( this, Generalization(),
        "must not have superclasses or subclasses." );
    Checks.assertEmpty( this, Realization(),
        "must not implement any interface." );
```

Eine weitere Forderung ist, dass die Ports einer Komponente eindeutige Namen haben müssen.

```
    Checks.assertUniqueNames( this, Port(),
        "a component's ports must have unique names." );
  }
  // more …
}
```

Die Operation *Port()* liefert die Menge aller definierten Ports zurück.

Jetzt noch ein kurzer Blick auf die Constraint-Validierung bei Interfaces: Angenommen, wir würden nur die primitiven Typen *int* und *long* unterstützen, dann müsste eine Prüfung stattfinden, dass der Typ des Parameters tatsächlich nur einer der beiden Werte ist:

```
public class ASTParameter
       extends SimpleNode { // SimpleNode extends
                            // ModelElement
    private String type;
    private String name;

    private static String[] types = new String[]
                                    {“int”, “long”};

    public void CheckConstraints() {
      Checks.assertOneOf( this, type, types,
             “Type must be one of int, long” );
    }

    public ASTParameter(int id) {
        super(id);
    }

    public void setType(String type) {
        this.type = type;
    }

    // set name implemented in ModelElement
}
```

Wie aus diesem Beispiel ersichtlich ist, sind die Unterschiede in der konkreten Syntax (UML, textuell, XML) vollständig neutralisiert.

Filterung

Auf Metamodellebene ist ein Port eine UML-Assoziation, die auf ein *InterfaceRef* zeigt. Die Menge aller Ports kann also durch entsprechende Filterung aller Assoziationen der Komponente[61] erstellt werden. Hier die Implementierung der Operation *Port()*:

```
public class Component extends Class {

  // as before

  public ElementSet Port() {
    return Filters.filter( this, AssociationEnd(),
      new AssocEndOppositeClassTypeFilter(
          InterfaceRef.class ),
      new AssocEndAssocMapper() );
  }
}
```

Um zu verstehen, was genau passiert, ist es hilfreich, sich die Struktur des relevanten Metamodellausschnitts anzusehen. Da wir das Metamodell als Erweiterung des UML-Metamodells realisieren, finden sich hier auch Konstrukte des UML-Metamodells wieder. Insbesondere wird eine Assoziation im Modell als Instanz eines *AssociationEnd*, einer *Association* und eines weiteren *AssociationEnds* abgebildet. Die Ports realisieren wir als Unterklassen von *Association*. Abb. 145 zeigt den relevanten Ausschnitt des Metamodells.

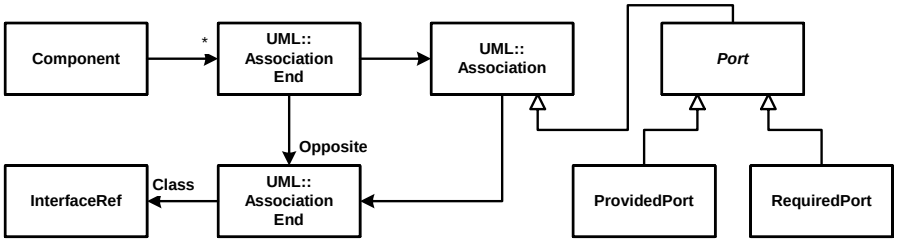


Abb. 145 *Metamodell für Ports als Erweiterung von Assoziationen*

Basierend auf diesem Metamodell wird nun der Code verständlich: Es wird eine Operation *filter()* aufgerufen, die vier Argumente erwartet:

- Zunächst, wie bei allen Hilfsfunktionen, wird das aktuelle Objekt übergeben. Dies ist wichtig, um bei Fehlermeldungen den richtigen Kontext mit ausgeben zu können.

---

[61] Eine Komponente kann auch noch andere Arten von Assoziationen haben. Dies wird hier allerdings nicht näher beleuchtet.

- Dann wird die Menge (genauer: das *ElementSet*) der Modellelemente angegeben, welche gefiltert werden sollen. Hier sind das die betreffenden *AssociationEnds*, also die Assoziations-Enden, die am aktuellen Element (hier: die Komponente) enden.
- Dann wird ein Filter übergeben. Hier ist es ein *AssocEndAssocTypeFilter*. Er geht davon aus, dass er ein *AssociationEnd* als Eingabe geliefert bekommt, holt sich dann die Assoziation und filtert diese auf den übergebenen Typ.
- Zu guter Letzt wird noch ein *Mapper* übergeben, welcher für jedes Modellelement (hier: *AssociationEnd*) das betreffende Assoziations-Objekt liefert. Wir wollen ja als Ergebnis der Operation die Ports (als Unterklasse von *Association*) haben, nicht die Enden.

Mit den vorgestellten Konstrukten kann man mit wenig Aufwand mächtige Filter „pseudodeklarativ“ programmieren. Die Alternative wären verschachtelte *for*-Schleifen.

17.6. Codegenerierung

Die Codegenerierung verwendet die schon in früheren Kapiteln beschriebene Template-Sprache *Xpand* des openArchitectureWare-Generators. Wir möchten in diesem Abschnitt allerdings weniger die Syntax der Template-Sprache vertiefen, sondern vielmehr wichtige Eigenschaften aufzeigen und elegante Techniken für die Metamodellimplementierung vorstellen, welche die Template-Programmierung stark vereinfachen.

17.6.1.Referenzen

Das Zusammenführen der verschiedenen Modelle geschieht mittels Referenzen. Nach Zusammenführen des Modells sieht also beispielsweise eine Komponente, ein Port und das dazugehörige Interface so aus, wie in Abb. 146 gezeigt.

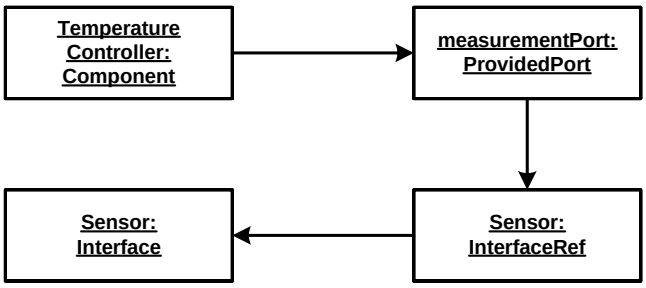


Abb. 146*Zusammengeführtes Modell (InterfaceRef referenziert Interface)*

Allerdings ist es aus Sicht der Templates unerheblich, dass das Interface über eine Referenz angebunden wird – das „konzeptionelle“ Metamodell sagt, dass ein Port mit einem Interface assoziiert ist. Um diese Abstraktion aus Sicht der Templates zu ermöglichen, führen wir in der Klasse *Port* einfach eine Operation *Interface()* ein, welche die Referenz implizit „dereferenziert“:

```
public abstract class Port extends Association {

  public IntefaceRef InterfaceRef() {
    // return the model element on the “target” side
    // of this association; remember ports are modelled
    // as associations that point from component to
    // interface reference.
  }

  public Interface Interface() {
    return InterfaceRef().Interface();
  }
}
```

Damit kann ein Xpand-Template folgendermaßen geschrieben werden:

```
«DEFINE Something FOR Component»
  public class «Name» implements
  «FOREACH ProvidedPort AS pp EXPAND»
    «Interface.Name»
  «ENDFOREACH»
  {
    // more stuff to come
  }
«ENDDEFINE»
```

Bei verteilter Anwendungsentwicklung kommt es oft vor, dass bestimmte Modellkonstrukte entweder ein Element bezeichnen oder eine Referenz darauf. In obigem Beispiel könnte man sich Folgendes vorstellen: Angenommen, Interfaces würden auch im UML-Modell definierbar sein. Dann könnte es vorkommen, dass ein definiertes Interface entweder im gleichen Modell definiert ist oder dass es in einem anderen Modell

vorliegt. Im ersten Falle kann man völlig ohne Referenzen arbeiten und den *Port* direkt auf das *Interface* zeigen lassen. Im zweiten Fall müsste man – wie oben gezeigt – mit einer gleichnamigen Referenz arbeiten. Als Folge kann es also vorkommen, dass ein Port entweder direkt mit einem Interface verbunden ist oder eben mit einer Referenz, die ihrerseits auf ein Interface „zeigt“. Abb. 147 zeigt dies:

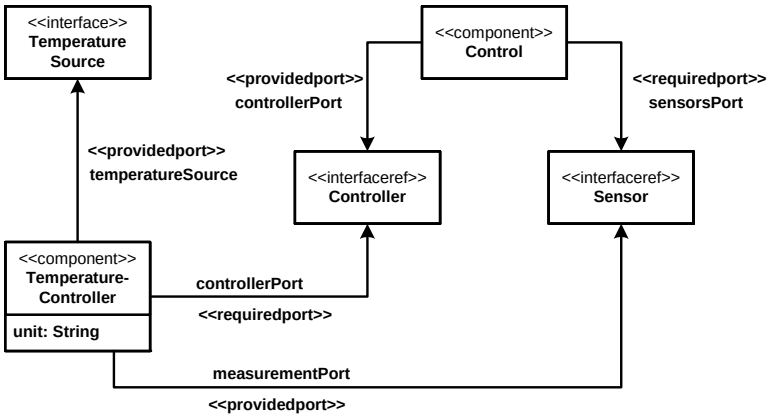


Abb. 147 *Beispielhaftes Komponentenmodell mit InterfaceRefs*

Ein *Port* ist also entweder mit einem *Interface* assoziiert oder mit einem *InterfaceRef*. Um dieses Problem elegant zu lösen, können wir auf die bewährten Mittel der Objektorientierung bauen und auf Metaklassen-Ebene das Proxy Pattern implementieren, wie in Abb. 148 gezeigt.

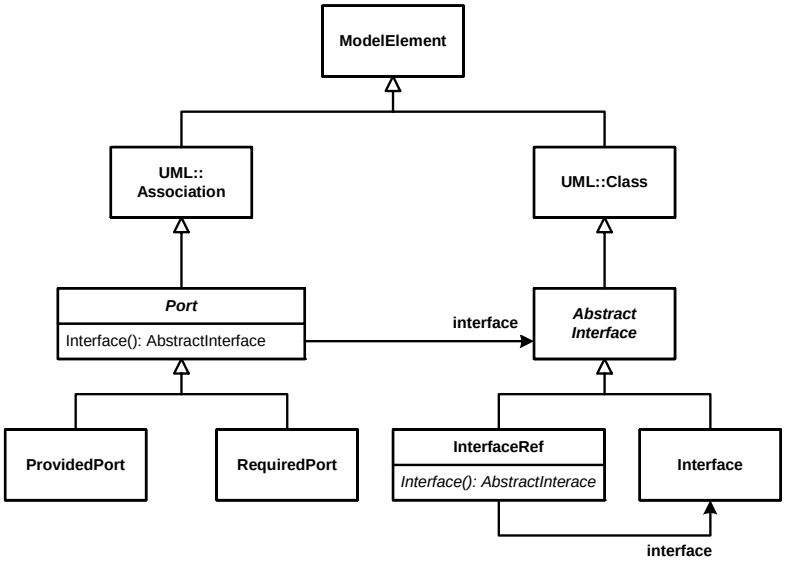


Abb. 148*Das Proxy Pattern als Basis für die Implementierung der InterfaceRefs*

Interessant ist dann vor allem die Operation *Interface()* des Ports. Dort wird unterschieden, ob es sich um eine Referenz oder tatsächlich um ein Interface handelt:

```
public abstract class Port extends Association {

  protected AbstractInterface AbstractInterface() {
    // return the “other” end’s class of the
    // association
  }

  public Inteface Interface () {
    AbstractInterface f = AbstractInterface();
    if ( f instanceof InterfaceRef ) {
      return ((InterfaceRef)f).Interface();
    }
    return (Interface)f;
  }

}
```

Das Xpand-Template von oben kann unverändert verwendet werden. Die Operation *Interface()* des Ports liefert immer das Interface zurück und dereferenziert – falls nötig – die Referenz intern.

Eine weitere Möglichkeit, Referenz-Proxys zu implementieren, besteht darin, dass das Referenzobjekt (der Proxy) alle Operationen so implementiert, dass sie an das referenzierte Objekt weiterdelegiert

werden. Ist dies der Fall, kann ein Proxy exakt so verwendet werden wie das vom Proxy referenzierte Objekt. Eine explizite Unterscheidung (wie oben in der Operation *Interface()* der Klasse *Port*) kann dann entfallen. Allerdings muss bei diesem Vorgehen auch wirklich sichergestellt sein, dass *alle* (Metamodell-relevanten) Operationen weitergeleitet werden. Wird die referenzierte Metaklasse weiterentwickelt, kann das „Nachziehen“ des Proxys leicht vergessen werden. Um dieses Problem zu adressieren, kann man die Referenz-Proxys (und die Metamodell-Implementierung) modellgetrieben generieren. Das openArchitectureWare-Framework bringt eine solche Meta-Cartridge bereits mit.

17.6.2. Polymorphismus

Der Generator unterstützt auf Template-Ebene Polymorphismus: Wenn in einer Template-Datei mehrere Template-Definitionen gleichen Namens vorhanden sind, die aber für verschiedene Metaklassen gelten, so expandiert der Generator für ein Modellelement immer dasjenige Template, dessen Typdeklaration dem dynamischen Typ des Modellelements am besten entspricht. Wir wollen dies an einem einfachen Beispiel verdeutlichen.

Zusätzlich zu Konfigurationsparametern können Komponenten auch gewöhnliche Attribute besitzen, die den Zustand der Komponente beschreiben. Abb. 149 zeigt ein Beispiel und dessen Realisierung im Metamodell.

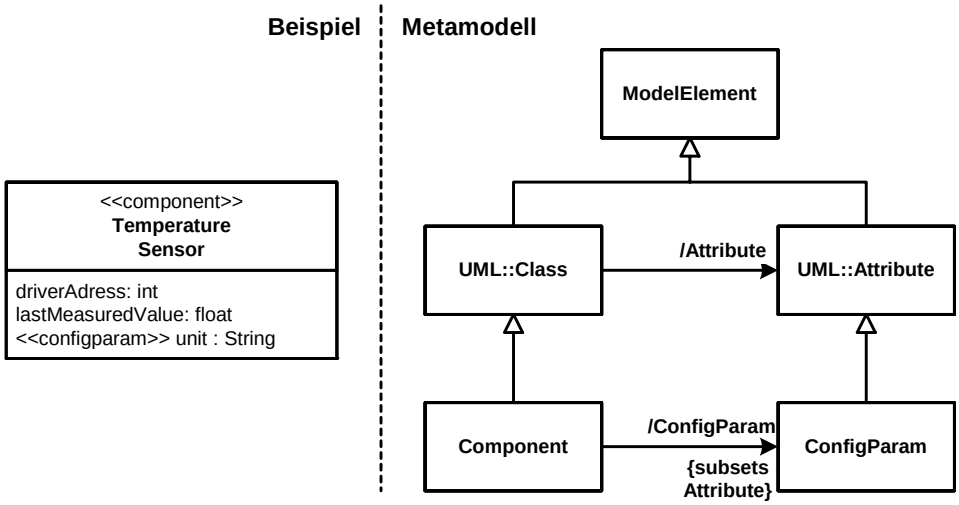


Abb. 149 *Konfigurationsparameter: Modell und Metamodell*

Die Menge aller Attribute enthält also demnach sowohl die „normalen“ Attribute als auch die *ConfigParams*. Das folgende Xpand-Template wird dadurch möglich:

```
«DEFINE Something FOR Component»
  public class «Name» {
    «EXPAND AttrDef FOREACH Attribute»
  }
«ENDDEFINE»

«DEFINE AttrDef FOR Attribute»
  private «Type» «Name»;
«ENDDEFINE»

«DEFINE AttrDef FOR ConfigParam»
  private String «Name»; // config param
  public void configure«UpperCaseName»( String value ) {
    this.«Name» = value;
  }
«ENDDEFINE»
```

Dieses Template iteriert in der dritten Zeile über alle Attribute und expandiert das Template *AttrDef* für jedes Attribut. Falls nun aber das aktuelle Attribut tatsächlich ein *ConfigParam* ist, wird das spezifische Template ausgeführt. Man erspart sich dadurch die typischen *if - isInstanceOf* -Kaskaden. Die Templates werden leichter erweiterbar.

17.6.3. Trennung von Verantwortlichkeiten im Metamodell

Utilities

Das Metamodell repräsentiert die Konzepte des Problemraums im Generator – so weit, so gut. Allerdings wandern auch oft Aspekte in das Metamodell, die mit dem Problemraum an sich nichts zu tun haben. Oben haben wir beispielsweise aus einem Template das Property *UpperCaseName* verwendet. Die bedeutet ja, dass in der Metaklasse, in deren Kontext wir uns gerade befinden (im Beispiel oben: *ConfigParam*), die folgende Operation vorhanden sein muss:

```
public String UpperCaseName() {
  String n = // format Name().toString() appropriately
  return n;
}
```

Nun wollen wir dieses *UpperCaseName* nicht nur für *ConfigParams* verwenden, sondern für alle Modellelemente mit Namen. Ebenso gibt es noch weitere nützliche Utilities, z.B. das aktuelle Datum, den Wert eines inkrementierbaren Zählers usw. Es ist nicht besonders elegant, all dieses

fest im Metamodell (damit es „überall“ verfügbar ist, also in *ModelElement*) zu implementieren. Schöner wäre es, wenn man diese Dinge „dazuschalten“ könnte, wenn sie nötig sind.

Genau dies ist mit den so genannten *Invokern* möglich. Wenn aus einem Template ein Property aufgerufen wird, welches nicht existiert, dann wird (entsprechend dem Interceptor-Pattern in [POSA2]) ein vorher registrierter *Invoker* aufgerufen. Dieser hat nun die Chance, einen Wert für das betreffende Property zurückzuliefern. Der folgende Invoker stellt den *UpperCaseName* zur Verfügung:

```
package genfwutil.mmutil.invoker;

public class UtilInvoker
       implements PropertyInvoker {

  public Object handleInvocation( Element element,
         Syntax syntax, Element name, String propertyName)
         throws EvaluationException {
    if ( propertyName.equals("UpperCaseName")) {
      String n = element.getNameProperty();
      return n.substring(0,1).toUpperCase()+
             n.substring(1);
    }
    throw new DesignError( "Property "+propertyName+
                           " not found on "+element );
  }
}
```

Der Generator ruft bei dem Invoker die Operation *handleInvocation()* auf. Diese hat verschiedene Parameter, insbesondere das aktuelle Modellelement (*element*), auf dem das Property (*propertyName*) aufgerufen werden soll. Wird ein Property verlangt, welches der Invoker nicht liefern kann, so wirft er einen *DesignError*.

Um diesen Invoker nun tatsächlich zum Einsatz zu bringen, muss – wie immer – ein Systemproperty gesetzt werden (hier mittels Ant):

```
<property name="de.bmiag.genfw.propertyNotFoundHandler"
          value="genfwutil.mmutil.invoker.UtilInvoker"/>
```

Wie auch schon bei den Frontends gezeigt, lässt sich das Composite-Pattern anwenden, um Invoker zu bauen, die mehr als eine Aufgabe erledigen:

```
public class MyInvoker extends CompositeHandler {
  public void init() {
    add( new UtilInvoker() );
    add( new CounterInvoker() );
    // add more, if necessary
  }
}
```

Der *CompositeInvoker* fragt eben alle mittels *addHandler()* registrierten Invoker, ob sie ein bestimmtes Property auflösen können. Im Beispiel wird noch der *CounterInvoker* verwendet, welcher mittels der Properties *CounterReset*, *CounterInc* und *CounterValue* einen Zähler in den Templates zur Verfügung stellt.

Verschiedene Zielsprachen

Ein ähnlich gelagertes Problem entsteht, wenn man wie hier im Beispiel mit verschiedenen Zielsprachen arbeitet. Für jede der Zielsprachen benötigt man Utility-Funktionen im Metamodell. Im Falle von Java beispielsweise *PackageDirectory*, welches das relative Verzeichnis für das *Package* des aktuellen Modellelements zurückliefert:

```
«DEFINE JavaImpl FOR Component»
  «FILE PackageDirectory"/"Name".java"»
  package «JavaPackage»
  public class «Name» {
    …
  }
«ENDDEFINE»
```

Man kann sich leicht vorstellen, dass man die Metamodellimplementierung sehr schnell mit allen möglichen Utilities für die verschiedenen Zielsprachen „verunreinigt“. Eine besser strukturierte Lösung muss her. Diese ist verfügbar in Form eines speziellen Invokers, dem *SubpackageInvoker*. Dieser versucht, eine andere Klasse zu instanziieren und auf dieser das betreffende Property per Reflection aufzurufen. Angenommen wir haben folgende Metaklasse:

```
package ecMetamodel;
public class Component extends Class {
  // some Operations
}
```

Wenn wir nun Operationen implementieren möchten, die für die Java-Plattform spezifisch sind, können wir diese in einer anderen Klasse implementieren:

```
package ecMetamodel.java;
public class Component_Java {
  public String JavaPackage( ModelElement el ) {
    …
  }
  public String PackageDirectory( ModelElement el ) {
    …
  }
}
```

Diese Operationen sind ja nun nicht im Kontext der Metaklasse implementiert, also gibt es auch kein *this*. Dafür wird das aktuelle Modellelement als Parameter übergeben. Der Entwickler kann nun hier Funktionalität genauso implementieren wie in der Metaklasse selber, allerdings muss er (explizit) mittels *el* statt (implizit) mittels *this* auf das Modellelement zugreifen.

Um diese Klassen auch tatsächlich zu verwenden, wird ein Invoker registriert, eben der *SubpackageInvoker*:

```
public class MyInvoker extends CompositeHandler {
  public void init() {
    // add UtilInvoker and CounterInvoker as before
    add( new SubpackageInvoker( “java”, “Java” ) );
  }
}
```

Ein derartig konfigurierter *SubpackageInvoker* versucht, für eine Metaklasse *p.q.C* die Hilfsklasse *p.q.java.C_Java* zu instanziieren und die Operation darauf aufzurufen.

Es wird damit möglich, verschiedene Aspekte im Metamodell in separate Klassen auszulagern und per Invoker explizit zum Modell hinzuzufügen.

17.6.4.Generierung der Build-Files

Im Rahmen der Codegenerierung wird nicht nur Java-, C- oder C++-Quelltext erstellt. Vielmehr werden auch verschiedene Konfigurationsdateien erstellt, die die verwendete Betriebssystemplattform konfigurieren. Echtzeitbetriebssysteme weisen meist eine große Vielfalt an Konfigurationseinstellungen auf, um das System für den betreffenden Anwendungsfall möglichst klein und performant einzustellen. Des Weiteren werden Makefiles und Ant-Files erstellt, die den generierten Code mit dem manuell erstellten zusammen kompilieren, linken, packen usw. – je nach den Anforderungen der betreffenden Plattform.

Im folgenden Beispiel zeigen wir kurz die Erstellung von Jar-Files. Für jeden im System definierten Container wird ein eigenes Jar-File erstellt, falls es sich um einen Java-Container handelt:

```
«DEFINE BuildFile FOR System»
    ...
  «FOREACH Container AS c EXPAND»
    «IF c.LanguageID == "java"»
      <jar jarfile="«c.Name».jar">
        <fileset dir="${APP.BUILD}">
          «FOREACH c.UsedComponent AS comp EXPAND»
```

```
          <include
name="generated.comp.«comp.Name»/**/*.class"/>
          <include
name="manual.comp.«comp.Name»/**/*.class"/>
           «FOREACH comp.UsedInterface AS i EXPAND»
            <include
name="generated.interfaces.«i.Name»/**/*.class"/>
           «ENDFOREACH»
         «ENDFOREACH»
       </fileset>
     </jar>
   «ENDIF»
  «ENDFOREACH»
   ...
«ENDDEFINE»
```

Wie aus dem Template-Ausschnitt erkennbar, werden auch alle Interfaces inkludiert, die von den betreffenden Komponenten verwendet werden.

17.6.5. Verwendung von AspectJ

In diesem Abschnitt möchten wir kurz eine mögliche Integration von AspectJ in den MDSD-Produktionsprozess zeigen und damit auf das Verhältnis von MDSD und AOP eingehen.

Als Beispiel verwenden wir hier Tracing. Dies ist zwar das Standard-Beispiel für AOP, allerdings ist diese Anwendung in der Welt der eingebetteten Systeme durchaus realistisch, weil Debugging auf der Zielplattform oft nicht möglich ist. Außerdem ist es wichtig, dass man *sämtlichen* durch Tracing erzeugten Overhead bei Bedarf auch wieder „ausschalten“ kann, um im realen Betrieb Ressourcen zu sparen.

Die Idee ist nun, dass man bei der Konfiguration des Systems definieren kann, für welche Container Tracingcode erzeugt werden soll:

```
<node name=”outside”>
  <container name=”sensorsOutside” tracing=”app”>
     …
  </container>
</node>
```

Das XML-Attribut tracing kann die folgenden Werte haben:

no Tracing ist ausgeschaltet.

app Nur die Anwendungslogik-Operationen der Komponenten werden getraced.

all Alle Operationen im Container werden getraced.

Auch hier gehen wir von der Maxime aus, dass nur Code generiert wird, der nicht sinnvoller manuell geschrieben werden kann. Wir definieren also als Teil der Plattform einen abstrakten Aspekt, welcher die Trace-

Funktionalität enthält. Dieser enthält einen abstrakten Pointcut, den wir später per Generierung konkretisieren.

```
package aspects;

public abstract aspect TracingAspect {

   abstract pointcut relevantOperationExecution();

   before(): relevantOperationExecution() {
         // use some more sophisticated logging,
         // in practice
        System.out.println( thisJointPoint.toString() );
   }

}
```

Dieser Aspekt erzeugt Trace-Informationen an allen Stellen, die durch den Pointcut *relevantOperationExecution* identifiziert werden. Dieser ist noch als abstrakt definiert. Im Rahmen der Generierung werden wir davon erben und diesen Pointcut definieren. Das folgende Template zeigt, wie dies gehen könnte:

```
«DEFINE LoggingAspect FOR System»
    ...
  «FOREACH Container AS c EXPAND»
    «IF c.Tracing == "app"»
      «FILE "aspects/”c.Name”Tracing”»
      package aspects;
      public aspect «c.Name»Trace extends TracingAspect {
        pointcut relevantOperationExecution() :
          «FOREACH c.UsedComponent AS comp
                   EXPAND USING SEPARATOR “||”»
           execution( * manual.comp.«comp.Name»..*.*(..) )
          «ENDFOREACH»
        ;
      }
      «ENDFILE»
    «ENDIF»
  «ENDFOREACH»
    ...
«ENDDEFINE»
```

Für den Außencontainer würde also folgender Code erstellt:

```
package aspects;
public aspect sensorsOutsideTrace extends TracingAspect {
  pointcut relevantOperationExecution() :
    execution( * manual.comp.temperatureSensor..*.*(..) )
    ||
    execution( * manual.comp.humiditySensor..*.*(..) )
    ;
}
```

Wenn jetzt statt mit *javac* mit dem AspectJ-Compiler übersetzt wird, wird dieser Aspekt automatisch dazugewoben. Er erzeugt Trace-

Ausgaben direkt vor der Ausführung aller Operationen des Codes im Implementierungs-Package. Die Mittel von AspectJ lassen auch aussagekräftigeres Tracing zu, zum Beispiel die Aufzeichnung der aufrufenden Stelle.

Durch die Verbindung von MDSD-Generatoren und AOP lassen sich – wie dieses Beispiel andeutet – mächtige Lösungen schaffen. Details zum Zusammenspiel MDSD/AOSD finden sich auch in [Voe04].

17.7. Kaskadierte Domänenarchitekturen

Zum Schluss wollen wir zeigen, wie man aufbauend auf der bereits definierten Infrastruktur weitere domänenspezifische Modellierungssprachen und Infrastrukturen kaskadierend aufbauen kann.

Nehmen wir an, es gilt ein modulares Bildverarbeitungssystem für eingebettete Systeme zu konstruieren. Sei weiterhin angenommen, dass man die zuvor beschriebene Container-Infrastruktur nutzen möchte, um beispielsweise von der Unterstützung für die verschiedenen Plattformen oder von der Verteilbarkeit auf verschiedene Knoten zu profitieren.

Eine Möglichkeit, dies anzugehen, besteht natürlich darin, mittels der definierten DSLs entsprechende Modelle zu erstellen, Code zu generieren und letztlich die Algorithmen manuell zu implementieren.

Eine effektivere Möglichkeit besteht darin, eine höhere Domänenarchitektur auf die bereits bestehende aufzusetzen.

17.7.1. Modellierung

Wir setzen unsere höhere Bildverarbeitungs-DSL auf dem Blackboard-Pattern auf. Dies organisiert Systeme derart, dass es ein Daten-Repository gibt, auf dem so genannte „Arbeiter“ (*Worker*) operieren. Jeder dieser Arbeiter entnimmt aus dem Repository Daten eines bestimmten Formates und bearbeitet sie. Nach dem Bearbeitungsschritt legt der Arbeiter die verarbeiteten Daten – nun in einem neuen Format – wieder zurück in das Repository, wo andere Arbeiter-Typen diese Daten ggf. entnehmen und weiterverarbeiten.

Letztendlich muss man eigentlich nur noch angeben, welche Datenformate existieren, und welche Arbeiter welche Daten produzieren und/oder konsumieren. Abb. 150 illustriert dies:

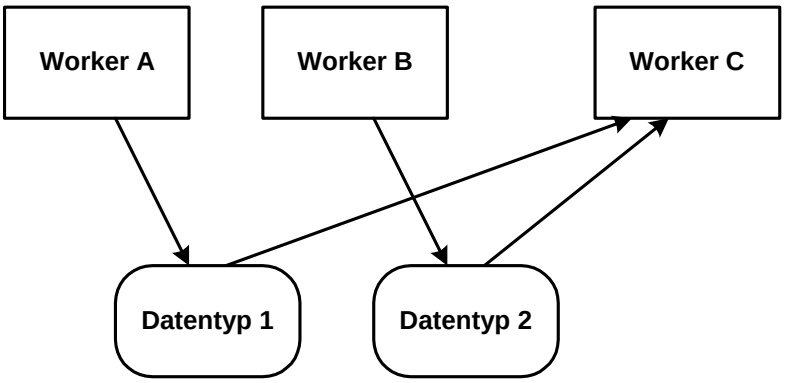


Abb. 150 *Worker und Datenfluss*

Ein solches Modell lässt sich z.B. sehr effizient mittels XML notieren. Man beachte, dass man auf dieser Abstraktionsebene nichts von den in diesem Kapitel beschriebenen Komponenten oder Containern sieht.

```
<ImageProcessingSpec>
  <DataType name=”RawData”>
    <ArrayMember name=”data” type=”byte” size=”1000”/>
    <ArrayMember name=”colorMap” type=”int” size=”256”/>
  </DataType>
  <DataType name=”ColorMapping”>
    <ArrayMember name=”fromMap” type=”int” size=”256”/>
    <ArrayMember name=”toMap” type=”int” size=”256”/>
  </DataType>
  <DataType name=”ProcessedData”>
    <ArrayMember name=”data” type=”byte” size=”1000”/>
  </DataType>
  <Worker name=”W1” processingClass=”img.RawDataProducer”>
    <Producer dataType=”RawData”/>
  </Worker>
  <Worker name=”W2” processingClass=”img.MappingProducer”>
    <Producer dataType=”ColorMapping”/>
  </Worker>
  <Worker name=”W3” processingClass=”img.ColorMapper”>
    <Consumer dataType=”ColorMapping”/>
    <Consumer dataType=”RawData”/>
    <Producer dataType=”ProcessedData”/>
  </Worker>
</ImageProcessingSpec>
```

17.7.2.Generierung

Aus diesem „höheren“ Modell kann man nun verschiedene Artefakte generieren:

- Wir können die Modelle, die als Eingabe für die Domänenarchitektur unserer Komponenteninfrastruktur dienen, erzeugen: Für jeden Worker wird eine Komponente generiert und

für jeden Datentyp ein Datentyp im Sinne des Komponentenmodells. Außerdem wird eine Repository-Komponente generiert. Auch die Ports für die Komponenten können aufgrund der Consumer/Producer-Angaben generiert werden. Ein einfaches System-Deployment-Modell kann auch erzeugt werden, indem von jeder Komponente eine Instanz erstellt wird und alle Instanzen einem Knoten zugewiesen werden. Dies kann manuell angepasst werden, falls ein verteiltes System gewünscht ist.

- Es können die Implementierungen für die Komponenten erstellt werden, indem die im Modell angegebenen Algorithmenklassen geeignet als Komponentenimplementierung gekapselt werden. Die Implementierungen der Algorithmen sind dann Teil der neuen, höheren MDSD-Plattform für Bildverarbeitung.

Anwender der Bildverarbeitungs-Domänenarchitektur können nun ihre Systeme kurz und prägnant in „ihrer“ DSL beschreiben. Die Generator-Konfiguration dieser Ebene erzeugt daraus die Modelle für die nächsttiefere Ebene. Auch kann natürlich die MDSD-Plattform der „höheren“ Domänenarchitektur mit Hilfe der (einer) „niedrigeren“ modellgetrieben erstellt werden. Diese Art der Kaskadierung von Domänenarchitekturen ist ein sehr mächtiger Ansatz und lässt sich natürlich auch im Bereich der Business-Software sehr sinnvoll einsetzen. Typisch ist dabei sicherlich die zweistufige Kaskadierung einer „fachlichen“ Domänenarchitektur auf einer architekturzentrierten. Abb. 151 illustriert die Beziehungen.

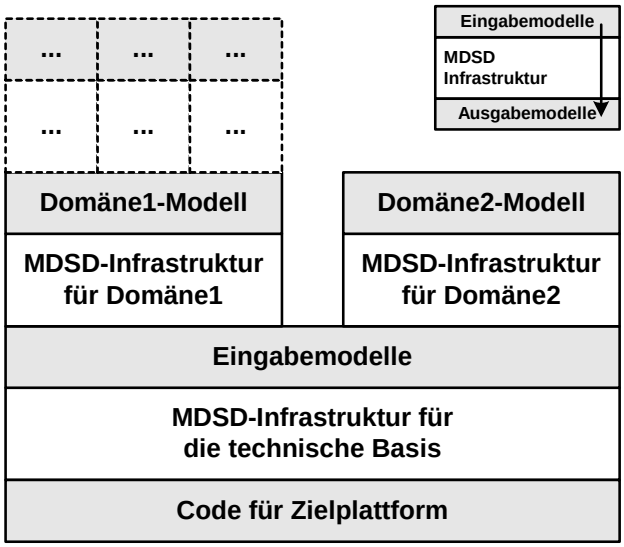


Abb. 151 *Kaskadierte modellgetriebene Entwicklung*

18. Objektorientierte Geschäftsprozessmodellierung und MDSD

Von Wolfgang Lorenz und Tim Weilkiens

Geschäftsprozessmodellierung, Softwareanalyse, Softwaredesign. Disziplinen, die eng aufeinander aufbauen, zwischen denen aber oft tiefe Gräben liegen. Das führt zu Inkonsistenzen zwischen modellierten Geschäftsprozessen und Softwareanforderungen und schließlich zu unpassenden Anwendungen. Die objektorientierte Geschäftsprozessmodellierung (OOGPM) setzt auf die UML als gemeinsame Beschreibungssprache, um den Graben zwischen traditioneller Geschäftsprozessmodellierung und objektorientierter Systemanalyse zu überwinden. Wir greifen in diesem Kapitel die Grundlagen dieser Methodik auf und zeigen, wie der Ansatz durch die Verknüpfung mit MDSD-Techniken Teil eines durchgängigen Vorgehens wird.

Durchgängigkeit bedeutet, dass in allen beteiligten Disziplinen – von der Geschäftsprozessmodellierung über Softwareanalyse zum Softwaredesign – einheitliche Konzepte, Methodiken und Notationen eingesetzt werden. Das führt zu einer besseren Kommunikation, bringt mehr Sicherheit, dass spezifizierte Anforderungen auch korrekt umgesetzt werden und schafft gute Voraussetzungen, um Verfolgbarkeit von Anforderungen durch alle Modelle zu ermöglichen (Requirements Traceability).

18.1. Geschäftsprozessmodellierung

Die Geschäftsprozessmodellierung (GPM) ist aus Sicht der Informatik ein Randbereich. Meist wird im Rahmen der Anforderungsanalyse für

ein Softwareentwicklungsprojekt eine begrenzte GPM betrieben, um den Kontext des zu entwickelnden Systems zu analysieren. Im umgekehrten Fall werden Softwareentwicklungsprojekte aus einer GPM abgeleitet, um Teilprozesse durch Automatisierung zu optimieren. Wir gehen von der in Abb. 152 skizzierten Situation aus. Ein Kfz-Vermietungsunternehmen hat seine Geschäftsprozesse untersucht und daraufhin drei Softwaresysteme identifiziert, die die Geschäftsprozesse unterstützen sollen.

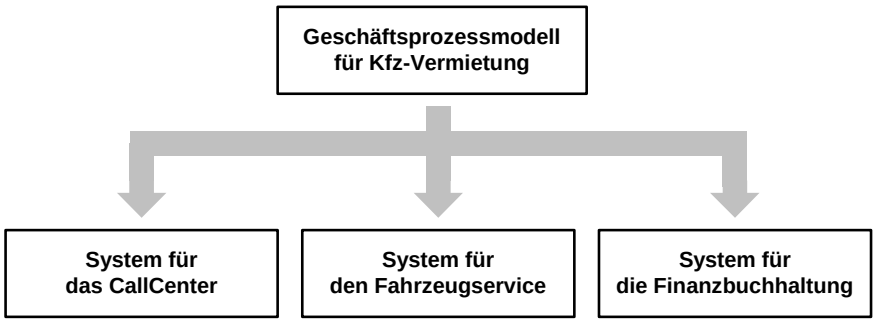


Abb. 152*Geschäftsprozessmodell und unterstützende Softwaresysteme*

18.2. Einheitlichkeit

Die Vorgehensweisen der GPM beinhalten viele Aktivitäten, die ähnlich oder identisch mit Aktivitäten der Softwareanforderungsanalyse sind. In beiden Disziplinen geht es vorwiegend um die Analyse und Beschreibung von Anforderungen und Abläufen. Die Informatik hat hierzu mehr bewährte Ansätze und standardisierte Notationen hervorgebracht als die GPM. Der Schluss liegt nahe, diese Techniken auch für die GPM einzusetzen. In der objektorientierten Softwareentwicklung hat sich die Unified Modeling Language (UML) als Modellierungsstandard und der Rational Unified Process (RUP) als Methodikgrundlage durchgesetzt. Beide enthalten keine ausgeprägte Unterstützung der GPM, können aber leicht adaptiert werden.

Das führt zu dem großen Vorteil gegenüber spezifischen GPM-Methodiken (z.B. ARIS, [ARIS]), dass eine einheitliche Beschreibungssprache vom Geschäftsprozessmodell bis zum Systemdesignmodell verwendet wird und die Gemeinsamkeiten der Methodik Synergieeffekte hervorbringen.

Ein weiterer Vorteil ist die formale Fundierung der UML. Das ist eine wichtige Grundvoraussetzung für MDSD (oder auch MDA) und ermöglicht bereits auf GPM-Ebene potenzielle Ansatzpunkte.

18.3. OOGPM-Methodik

Abb. 153 zeigt die Methodik der objektorientierten GPM (OOGPM) im Hinblick auf eine geplante Softwareentwicklung. Jeden Schritt der Methodik zu erläutern, würde den Rahmen des Kapitels deutlich sprengen und am eigentlichen Fokus vorbeigehen. Interessierte Leser seien auf das Buch [OWS+03] verwiesen, welches das Vorgehen detailliert beschreibt.

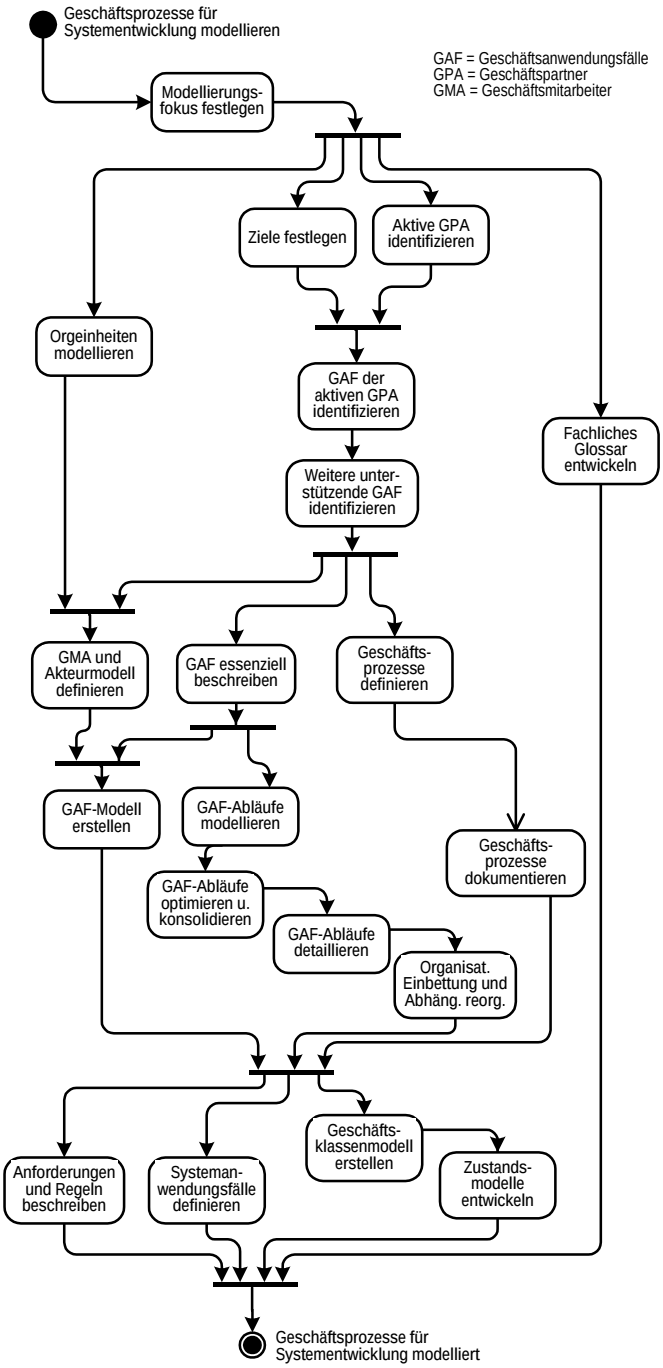


Abb. 153 *OOGPM-Methodik*

Im Kern ist die OOGPM-Methodik ähnlich dem Vorgehen in objektorientierten Softwareentwicklungsprojekten. Funktionale Anforderungen werden mit Anwendungsfällen, die jeweiligen detaillierten Abläufe mit Aktivitätsdiagrammen beschrieben. Die Strukturen der geschäftlichen Objekte sind in Klassendiagrammen modelliert. Die Ergebnisse der GPM beinhalten also dieselben Artefakte wie die der objektorientierten Softwareanalyse: Anwendungsfälle, Aktivitätsdiagramme und Klassen (siehe Abb. 154)

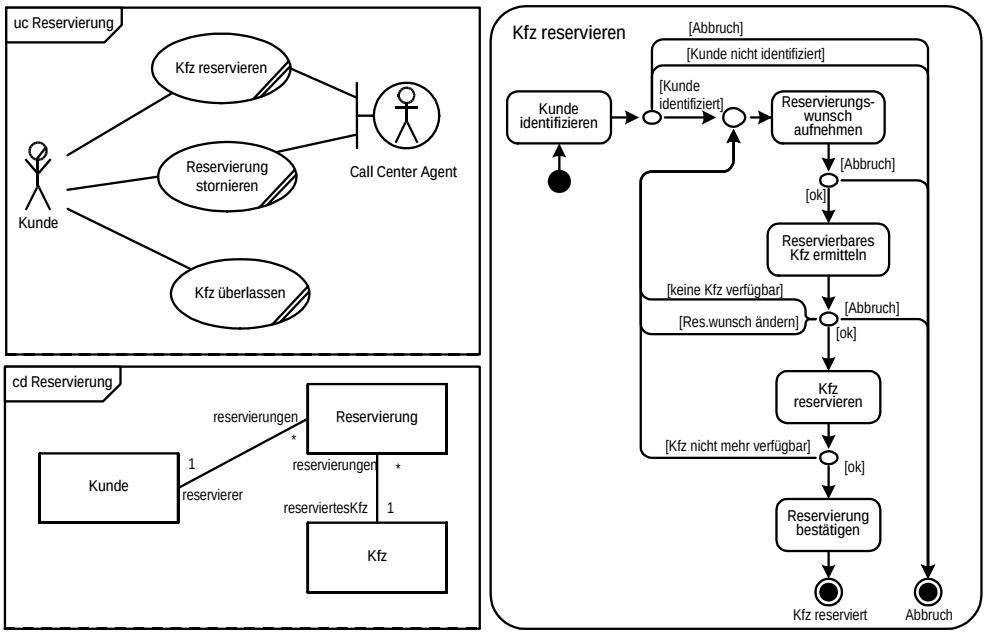


Abb. 154*Einige Ergebnisse der GPM: Anwendungsfälle, Aktivitätsdiagramm, Geschäftsklassen*

Weitere Aktivitäten der GPM-Methodik führen beispielsweise zu Anforderungen wie Geschäftsregeln, einem Glossar und einem Modell der Organisationsstruktur.

Auch ein wichtiges Ergebnis der GPM sind natürlich die Geschäftsprozesse selbst. In der OOGPM-Methodik ist ein Geschäftsprozess eine Gruppe fachlich zusammengehöriger Geschäftsanwendungsfälle, wie in Abb. 155 gezeigt.

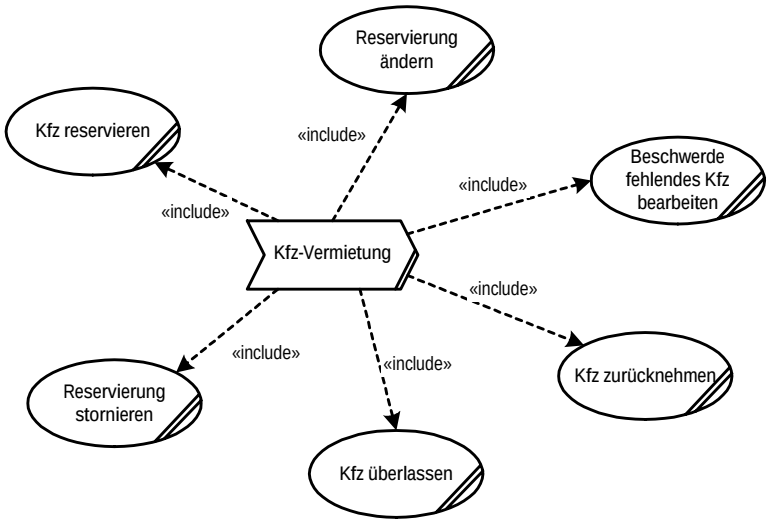


Abb. 155*Geschäftsprozess Kfz-Vermietung*

Die sachlogische Reihenfolge, in der die Geschäftsanwendungsfälle auftreten dürfen, kann in einem Aktivitätsdiagramm ausgedrückt werden. Im Geschäftsprozess Kfz-Vermietung findet „Kfz reservieren“ vor „Kfz überlassen“ und das wiederum vor „Kfz zurücknehmen“ statt. Diese Abläufe sind z.B. für Workflow-Systeme wichtig.

Im Geschäftsprozessmodell werden, beispielsweise im Rahmen einer Optimierungsstrategie, Bereiche identifiziert, die von einem Softwaresystem umgesetzt werden sollen. Das können ganze Anwendungsfälle oder nur Teilabläufe sein. Die ausgewählten Modellelemente werden mit dem Stereotyp *<<system support>>* gekennzeichnet.

18.4. Transformation in das Software-Analysemodell

Das GPM-Modell enthält bereits einen Großteil der Anforderungen an die zu entwickelnde Software: Anwendungsfälle, Abläufe, Strukturen. Diese Informationen wollen wir vollständig und unverfälscht in das Software-Analysemodell überführen.

Was passiert in der Praxis? Der Analytiker wird die markierten Anwendungsfälle und Abläufe aus dem Geschäftsprozessmodell in das Analysemodell kopieren und dann weiter ergänzen und verfeinern. Dieser Schritt ist teilweise automatisierbar. Es ist eine einfache

Transformation, da die Elemente lediglich kopiert, aber nur geringfügig mit weiteren Informationen angereichert werden. Die Vorteile sind, dass

- alle identifizierten Bereiche auch tatsächlich in das Analysemodell übernommen werden,
- keine Übertragungsfehler entstehen, wie sie beim manuellen Übertragen („abschreiben“) leicht auftreten können,
- automatisch ein Bezug zwischen den kopierten Elementen im Software-Analysemodell und den Elementen im GPM-Modell hergestellt werden kann (UML: <<*trace*>>-Beziehung).

Die Transformation ist ein Forward Engineering Ansatz. Änderungen müssen an der Quelle vorgenommen werden, d.h. wenn sich beispielsweise der Ablauf eines Geschäftsanwendungsfalles ändert, muss diese Änderung auch im GPM-Modell durchgeführt werden und nicht im Software-Analysemodell oder gar im Sourcecode selbst. Das bedeutet zwar zunächst in der Regel einen erhöhten Aufwand, der sich aber schnell bezahlt macht. Wenn man Geschäftsprozessmodellierung im Sourcecode betreibt, schleichen sich schnell ungewollte Nebeneffekte ein, da der nötige Weitblick auf dieser Abstraktionsebene fehlt. Man ist gerne geneigt, Wirkungen zu bekämpfen. Aber nur die Ursachenbekämpfung führt auf Dauer zum Erfolg.

Modellelemente, die ihren Ursprung im Geschäftsprozessmodell haben, werden im Analysemodell mit dem Stereotyp <<*gpm*>> gekennzeichnet. Sie sind schreibgeschützt, damit es bei wiederholten Transformationen nicht zu Konflikten kommt.

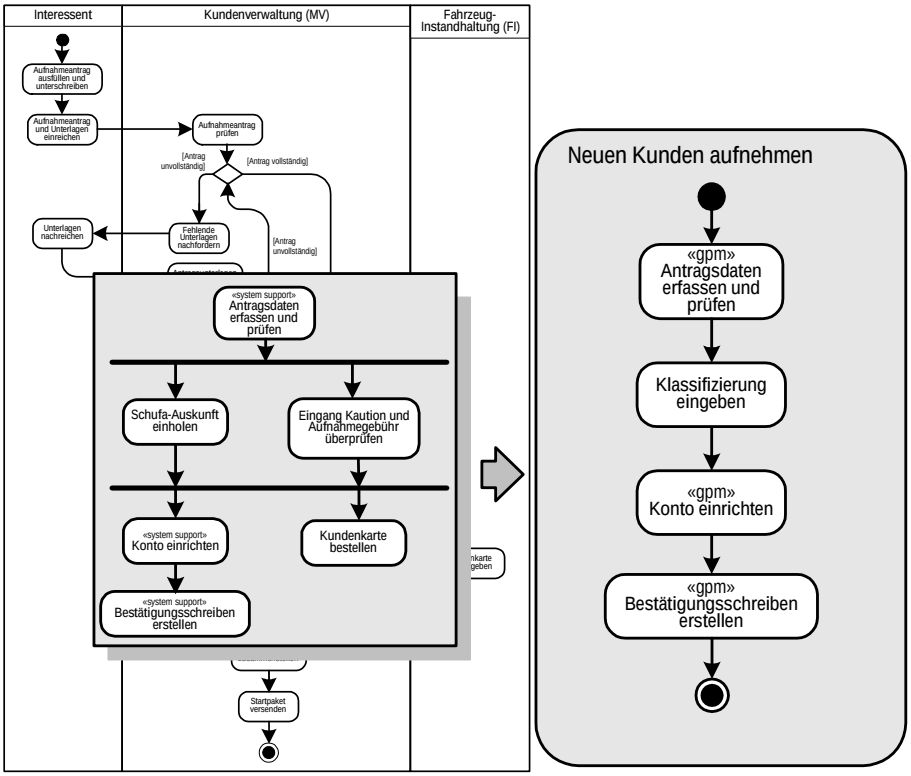


Abb. 156 *Transformation eines geschäftlichen Ablaufes*

Die Transformation kann erweitert werden, um neben Anwendungsfällen und Abläufen auch weitere Informationen automatisch aus dem GPM-Modell in das Software-Analysemodell zu übernehmen. Wurde beispielsweise in Aktivitätsdiagrammen auch der Objektfluss modelliert, können automatisch Fachklassen im Analysemodell angelegt werden (Abb. 157).

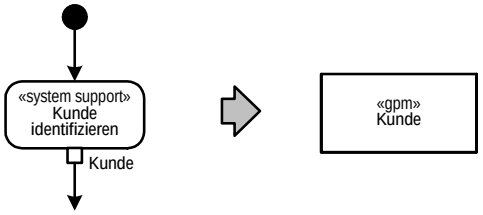


Abb. 157 *Ableitung von Fachklassen aus dem Aktivitätsdiagramm*

18.5. Software-Analysemodell

Das Software-Analysemodell beschreibt und modelliert die Anforderungen an ein spezifisches Softwaresystem. Es ist sehr ähnlich zum Geschäftsprozessmodell, nur dass nun der Fokus nicht auf dem gesamten Unternehmen, sondern eingeschränkt auf einem Softwaresystem liegt. Entsprechend werden nur bestimmte Abläufe und Strukturen betrachtet, dafür aber wesentlich detaillierter, als es im GPM-Modell der Fall ist.

Der Analytiker verfeinert die transformierten Geschäftsprozesselemente. Das führt oft zu neuen Erkenntnissen, die bei der GPM nicht erkannt worden sind und rückwirkend im GPM-Modell einfließen können. Eine enge Zusammenarbeit des Geschäftsprozessmodellierers und des Softwareanalytikers ist also wünschenswert, wenn es nicht sogar dieselbe Person ist, die beide Rollen einnimmt.

Natürlich werden in der Analyse auch Anwendungsfälle, Abläufe und Strukturen identifiziert und beschrieben, die nicht in die Geschäftsprozessmodellierung gehören. Beispielsweise könnte das Softwaresystem eine Benutzerverwaltung haben, die zu Anwendungsfällen wie „Benutzer anmelden“ führt. In der GPM würde man diesen Detaillierungsgrad nicht betrachten, sondern eher nur eine Anforderung formulieren, dass eine Benutzerverwaltung erforderlich ist.

18.6. Systemdesign und Implementierung

In OOGPM und Systemanalyse wurden die erhobenen fachlichen Anforderungen an unser System in formale UML-Modelle umgesetzt. Das funktionale „Was“ ist – im iterativ inkrementellen Sinne – damit bereits beschrieben. Es liegt zum jetzigen Zeitpunkt also zumindest ein sinnvoller Ausschnitt der Fachlichkeit vor, mit dem weitergearbeitet werden kann.

Es bleibt noch die Frage nach dem „Wie“: Wie setzt das System die modellierte Fachlichkeit um? Die Beantwortung dieser Frage ist Hauptaufgabe des Systemdesigns. Auch hier ist eine modellbasierte, formale Lösungsbeschreibung gefragt. In den Kapiteln <Ref CSI> sowie <Ref Basis::AC> haben wir uns bereits ausführlich mit architekturzentrierter, modellgetriebener Softwareentwicklung (AC-MDSD = Architecture Centric MDSD, Architekturzentriertes MDSD) befasst. Durch diesen Ansatz sind die Übergänge von Systemanalyse zu Systemdesign und von Systemdesign zu Implementierung zumindest für

Fachklassenmodelle bereits beschrieben: Ersterer besteht in der Profilierung des Analysemodells, d.h. der semantischen Anreicherung von Modellelementen durch architekturzentrierte Sprachkonstrukte, Letzterer in der anschließenden Generierung eines Infrastrukturcode-Rahmens sowie dessen Ausprogrammierung.

An der Implementierung eines Systemanwendungsfalles (UseCases) sind in der Regel jedoch drei Arten von Objekten beteiligt. Die UML kennt sie als *Entity*, *Boundary* und *Control*.

- <<*entity*>> – passive Objekte, die innerhalb von Anwendungsfällen benötigt werden. In diese Kategorie fallen die bereits behandelten Fachklassen.
- <<*boundary*>> – Dialoge und andere Interaktionsschnittstellen, die gebraucht werden, um Aktionen eines Akteurs in Ereignisse eines Systems zu übersetzen.
- <<*control*>> – Controller steuern die Interaktionen zwischen verschiedenen Objekten und kapseln damit das Verhalten eines UseCases.

Offen ist mit der Controller-Logik also noch die komplette Dynamik des Systems. Ansatzpunkt für deren Beschreibung ist hier das weniger bekannte UseCaseController-Pattern, eine Spezialisierung des MVC-Patterns bezüglich der Verantwortlichkeit des Controllers. Abb. 158 zeigt ein entsprechendes Metamodell.

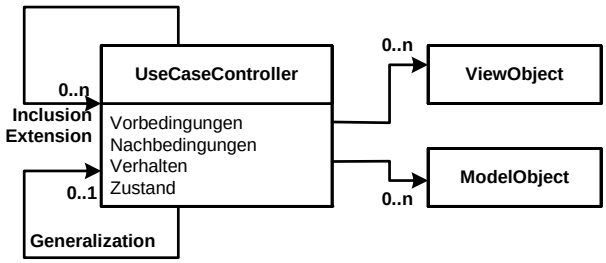


Abb. 158 *UseCaseController-Pattern*

Der UseCaseController repräsentiert aus technischer Sicht genau einen UseCase, kennt Vorbedingungen, Nachbedingungen, besitzt einen Zustand und ein Verhalten. Anstatt Letzteres nun programmatisch zu beschreiben, wird dem Controller im Design die aus der Systemanalyse stammende, formale Ablaufbeschreibung zugeordnet. Konzeptionell gesehen wird der Controller dadurch zum Interpreter seiner Ablaufbeschreibung.

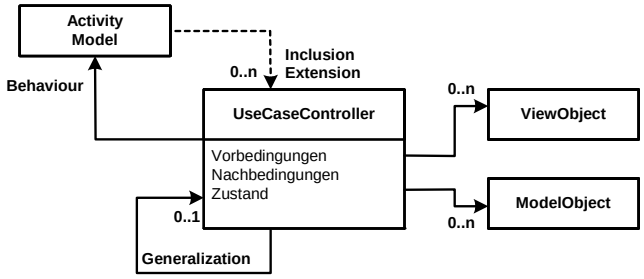


Abb. 159 *Spezialisiertes UseCaseController-Pattern*

Die Zuordnung des Ablaufmodells (Behavior) zur UseCaseController-Klasse (BehavioredClassifier) kann in der UML 2.0 mit Hilfe einer Assoziation vorgenommen werden, wie in Abb. 160 gezeigt.

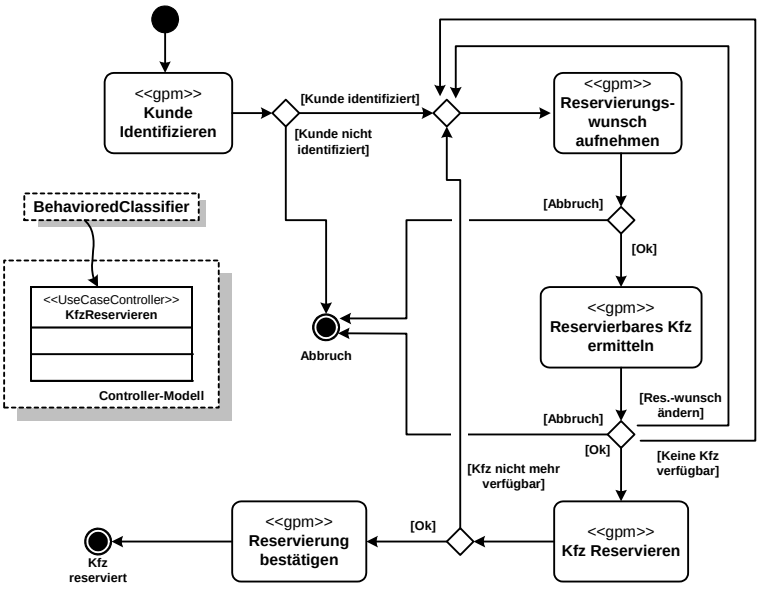


Abb. 160 *Zuordnung eines Aktivitätsdiagramms zu einem BehavioredClassifier*

Da es sich bei den Ablaufbeschreibungen um Artefakte der Systemanalyse handelt, ist ihr Abstraktionsgrad relativ hoch und die Koordination von Model- und View-Objekten noch nicht spezifiziert. Es ist also Aufgabe des Systemdesigners, entsprechende Modelle zu entwickeln, die die Beschreibung der Systemdynamik komplettieren, ohne dass dabei Redundanzen gegenüber den Ablaufmodellen entstehen.

Umsetzung und Integration erfolgen hier mit Hilfe des ActionController-Patterns, einer weiteren Spezialisierung des UseCaseController-Patterns. Es sieht eine Spezifizierung des Verhaltens

primitiver Actions mit Hilfe von Zustandsmodellen (Event-Action-Modellen) vor. Abb. 161 zeigt ein entsprechendes Metamodell.

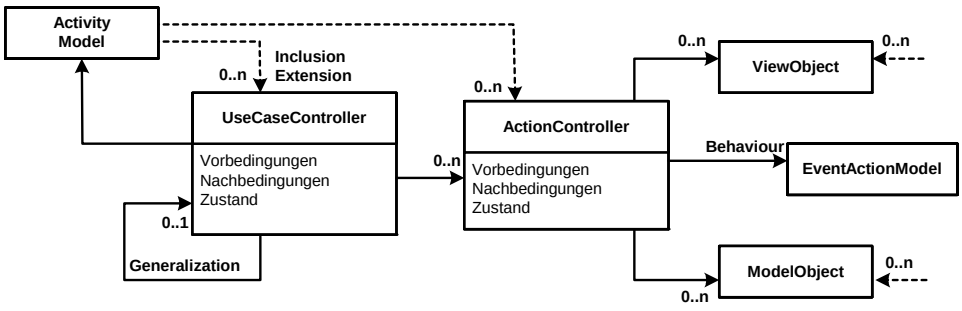


Abb. 161 *UseCaseController-/ActionController-Pattern*

Ein ActionController repräsentiert jeweils eine primitive Action im Ablaufmodell des UseCaseControllers. ActionController sind dabei in unterschiedlichen UseCases wiederverwendbar.

Der UseCaseController koordiniert seine ActionController und delegiert einen Großteil seiner Verantwortung auf sie. Wie beim UseCaseController wird das Verhalten des ActionControllers modelliert und das Modell der Controller-Klasse zugeordnet. Abb. 162 zeigt ein Beispiel.

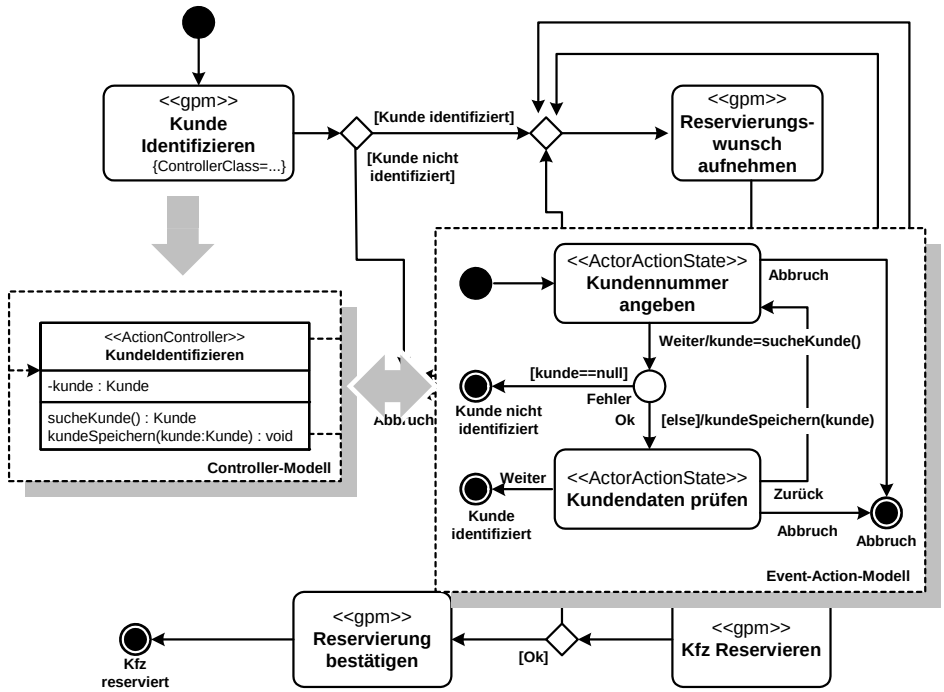


Abb. 162 *Zusammenspiel von Ablauf- und Zustandsmodell (Event-Action-Modell)*

Die Information, auf welchen *ActionController* der *UseCaseController* jeweils delegiert, wird mit Hilfe von *ControllerClass*-Tags auf das Ablaufmodell dekoriert. Der *ActionController* definiert die Menge der in seinem Zustandsmodell verfügbaren Aktionen (Methoden der Controller-Klasse) und zugreifbaren <<*entity*>>-Objekte (Attribute der Controller-Klasse). Das Zusammenspiel von Ablauf- und Zustandsmodellen gestaltet sich wie folgt:

Verfügt eine Action in einem Ablaufmodell über eine ausgehende Kante zu einer Verzweigung, muss an jeder ausgehenden Kante dieser Verzweigung eine Bedingung (Guard) notiert werden. Jeder Guard muss dabei jeweils einem der Ausgänge des Zustandsmodells des ActionControllers der der Verzweigung vorgelagerten Action namentlich entsprechen (s. Beispiel „Kunde identifizieren“ in Abb. 162).

Zum Verständnis ist vielleicht auch folgendes Bild hilfreich: Man stelle sich das Ablaufmodell als eine Platine vor, die einzelnen Actions als ICs. Die Pins der ICs (Ein- und Ausgänge der Zustandsmodelle) sind auf der Platine miteinander verbunden (über die Kanten).

Ähnliches gilt übrigens auch für UseCase-Aufrufe in Ablaufmodellen – den Abbildungen der <<*include*>>- bzw. <<*extend*>>-Beziehungen im entsprechenden UseCase-Diagramm. Verfügt hier eine

<<UseCaseCall>>-Action über eine ausgehende Kante zu einer Verzweigung, muss an jeder ausgehenden Kante dieser Verzweigung eine Bedingung (ein *Guard*) notiert werden. Jeder Guard muss dabei jeweils einem der Ausgänge des Ablaufmodells des gerufenen UseCases namentlich entsprechen. Die Information, auf welchen *UseCaseController* beim Aufruf delegiert wird, wird mit Hilfe des *ControllerClass*-Tags auf die *UseCaseCall*-Action dekoriert.

Alle direkten und indirekten Beziehungen zwischen Controllern (*<<Delegate>>*/*<<Call>>*) werden in einem zentralen Controller-Modell mit Hilfe von Dependencies modelliert. Es integriert die Modelle der Systemanalyse mit denen des Systemdesigns und macht eine komplexere Model-to-Model-Transformation damit unnötig.

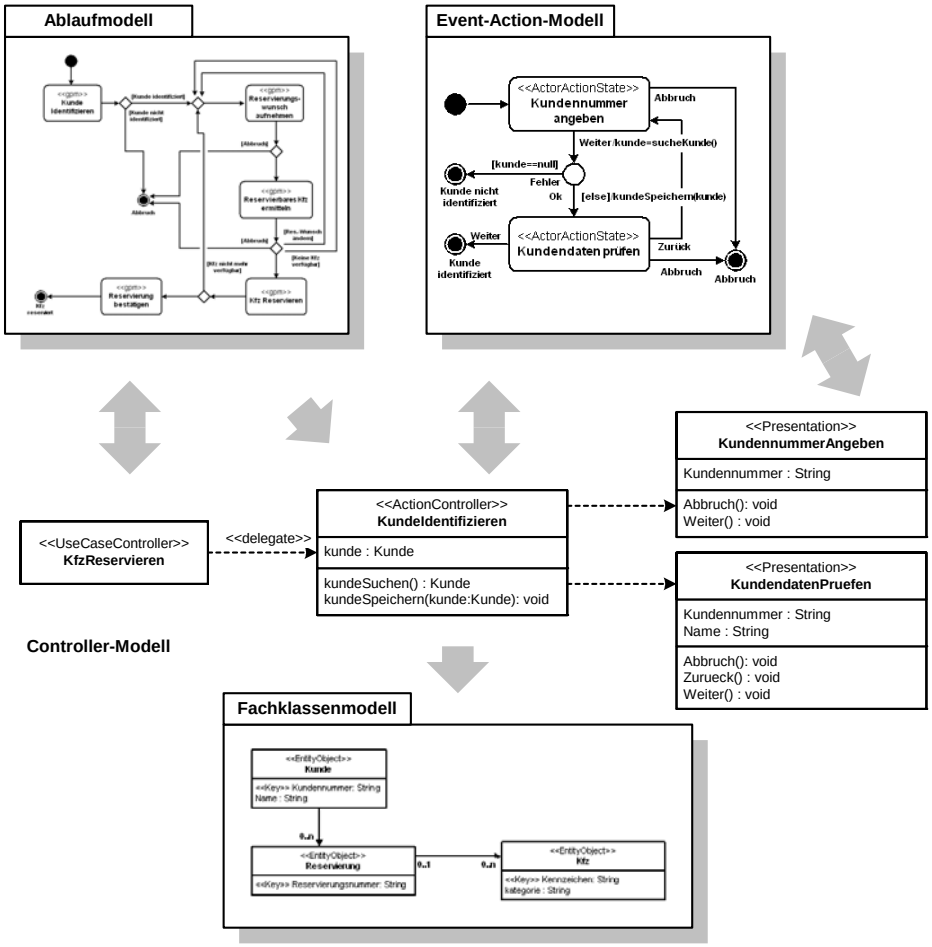


Abb. 163 *Zusammenspiel aller Modellarten*

Die noch offene Herstellung des Bezugs zu den <<*boundary*>>-Objekten (*Presentations*) geschieht im Controller-Modell ebenfalls mit Hilfe von Dependencies, im Zustandsmodell – ähnlich wie bei den Actions – mit Hilfe von *PresentationClass*-Tags an den *ActorActionStates*. Ein *ActorActionState* ist dabei ein entsprechend gestereotypter Zustand, der direkt an einer Benutzerinteraktion beteiligt ist. Der Stereotyp <<*ActionState*>> würde konsequenterweise einen Zustand spezifizieren, der internes Systemverhalten beschreibt und daher ohne Präsentationsklasse auskommt.

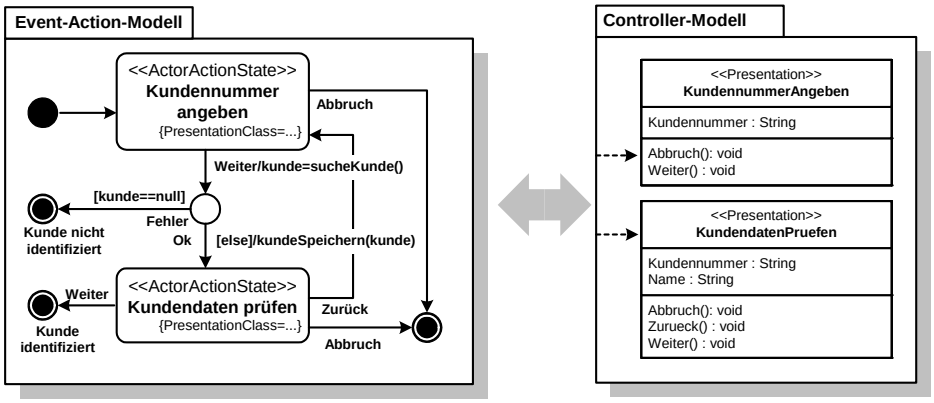


Abb. 164*Zustandsmodell (Event-Action-Modell)*

Die Präsentationsklasse selbst spezifiziert mit ihren Methoden die Events, die über sie ausgelöst werden können und einer Entsprechung im jeweiligen Zustandsmodell (Event-Action-Modell) bedürfen. Attribute werden für die Definition der Datenfelder der Präsentation verwendet. Präsentationsklassen sind nicht an einen ActorActionState gebunden und damit wiederverwendbar.

Auch für die Dynamik unseres Systems ist der Graben zwischen Systemanalyse und Systemdesign damit überbrückt. Die verwendeten Muster sind dabei abstrakt genug, um in verschiedenen Software-Systemfamilien eingesetzt werden zu können. Eine mögliche Plattformbindung haben wir bereits in Kapitel <Ref CSI> dargestellt. Der Übergang vom Systemdesign zur Implementierung kann dank der profilierten Modelle wie schon beim Fachklassenmodell automatisiert werden.

18.7. Fazit

Der hier vorgestellte Weg von der Geschäftsprozessmodellierung bis hin zur Implementierung ist dank der Verbindung von OOGPM und MDSD kurz, performant und redundanzarm. Vor allem die Möglichkeit, direkt mit den Artefakten der im Entwicklungsprozess vorgelagerten Stufe weiterzuarbeiten, macht das Vorgehen geradlinig und funktioniert selbstverständlich iterativ inkrementell. Obwohl hier die UML 2 verwendet wurde, funktioniert dieses Vorgehen natürlich auch mit heutigen, noch nicht UML 2-konformen Modellierungstools.

19. Teil 4: Management

Dieser Teil des Buches untersucht modellgetriebene Softwareentwicklung (MDSD) aus der Management-Perspektive. Unter Management verstehen wir neben den kommunikativen Aspekten ganz allgemein die Definition und das Verfolgen von Strategien, Zielen und Maßnahmen für spezifische Bereiche. Konkret wollen wir mit diesem Teil des Buches insbesondere IT-, Architektur- und Projekt-Manager ansprechen.

Die Ausführungen umfassen ökonomische Betrachtungen im Kontext von MDSD sowie organisatorische Randbedingungen. Unter anderem gehen wir auch auf das Thema Outsourcing und Offshoring ein. Abgeschlossen wird dieser Teil des Buches durch die Behandlung von Adaptionsstrategien für MDSD, welche die Einführung dieses Paradigmas bei Projekten oder Unternehmen stützen.

20. Entscheidungshilfe

Mit Jorn Bettin

Dieses Kapitel erläutert die ökonomischen Vorteile und die Investitionen, die mit modellgetriebener Softwareentwicklung (MDSD) im Allgemeinen und architekturzentrierter im Besonderen verbunden sind. Außerdem werden Antworten auf typische, kritische Fragen gegeben.

Ein Überblick über die Motivation und die Grundprinzipien von MDSD findet sich in Kapitel <Ref Basis>, Abschnitte <Ref Basis::Herausforderung> bis <Ref Basis::MDSD Ansatz>. Wir empfehlen diese Abschnitte zum besseren Verständnis vor der Lektüre dieses Kapitels zu lesen.

20.1. Betriebswirtschaftliches Potenzial

In der modellgetriebenen Softwareentwicklung (MDSD = Model-Driven Software Development) kombinieren wir die skalierbaren Aspekte agiler Ansätze mit weniger bekannten Techniken zur Verhinderung von Qualitäts- und Wartbarkeitsproblemen in großen Systemen sowie mit Techniken zur Automatisierung sich wiederholender Abläufe in der Softwareentwicklung.

Die Potenziale modellgetriebener Entwicklung ergeben sich aus wenigen Grundideen:

- Formalisierung und Komprimierung von Softwareentwürfen durch Schaffung von Modellierungssprachen, die eher am Problemraum als am Lösungsraum orientiert sind.
- Abstraktion vom Ausdrucksniveau heutiger Programmiersprachen und Plattformen.
- Einsatz von Generatoren zur Automatisierung repetitiver Abläufe.

- Trennung von Verantwortlichkeiten (Separation of Concerns) ermöglicht weit gehend getrennte Bearbeitung und Evolution von fachlichem und technischem Code.
- Wiederverwendung – auch über Projektgrenzen hinweg – durch Bildung von Software-Systemfamilien und Produktlinien (vgl. <Ref Begr::SSF>).

Ausgehend von diesen Grundideen lassen sich eine ganze Reihe von Eigenschaften der so erzeugten Softwaresysteme ableiten und auf den resultierenden betriebswirtschaftlichen Nutzen abbilden.

	Betriebswirtschaftlicher Nutzen			
MDSD Eigen-schaften	**Schnellere Implementierung neuer Geschäftsanforderungen**	**Geringere Einführungskosten von neuen Technologien**	**Reduzierte Kosten über den gesamten Produktlebenszyklus**	**Strategischer Geschäftsvorteil**
Verwertung von Expertenwissen	Zur Umsetzung neuer Geschäftsanforderungen wird primär Domänenwissen und weniger technisches Wissen benötigt.	Technologie-Experten „gießen“ ihr Wissen in MDSD-Plattformen und Transformationen, so dass es für die Anwendungsentwicklung zur Verfügung steht.	In der Anwendungsentwicklung werden weniger Technologieexperten benötigt.	Geschäftswissen ist weit gehend in konsistenten Modellen abgebildet und somit maschinell verwertbar. Technologiewissen ist in Plattformen und Transformationen abgebildet.
Automation in der Anwendungsentwicklung	Software-Produktionsstraßen verkürzen die Entwicklungszeit.	Implementierungs-technologien können durch die Automation mit geringeren Kosten adaptiert werden.	Insgesamt geringerer Zeit- und Personalbedarf.	Kürzere Time-to-Market.
Sicherstellung von Anwen-dungs-qualität		Klare und formale Trennung von Infrastruktur und fachlichem Code erleichtert einen Technologiewandel	Deutlich verbesserte Wartbarkeit von Architektur- und Technologieaspekten – über die Grenzen einzelner Softwaresysteme hinaus. Automation reduziert potenzielle Fehlerquellen.	Reduktion von Wartungskosten, Verbesserung der Kundenzufriedenheit, vereinfachte Revisionssicherheit.
Weit gehende Entkopplung von Technologien	Technologieänderungen beschränken sich weit gehend auf die Plattform und Transformationen, und beeinflussen nicht die Anwendungsmodelle.	Jede neue Technologieanbindung wird nur einmal zentral implementiert.	Insgesamt geringerer Zeit- und Personalbedarf.	Neue, für das Geschäftsfeld relevante Technologien können frühzeitig genutzt werden.
Nutzung formaler Anwendungsmodelle	Änderungen und Erweiterungen werden modellgetrieben realisiert. Die bestehende Produktionsstraße wird unverändert genutzt.	Anwendungsmodelle sind in der Regel technologieunabhängig und daher robust gegen Technologieänderungen.	Isolierung von Technologiedrift. Der Anwendungslebenszyklus ist besser vom Technologielebenszyklus entkoppelt.	Konsistente und weit gehend technologieunabhängige Softwarespezifikation verringert Technologie- und Herstellerabhängigkeiten.

Tab. 1 *Betriebswirtschaftlicher Nutzen von MDSD*

Wir wollen nun die positiven Effekte, welche die MDSD-Prinzipien auf Softwareprojekte haben können, im Einzelnen betrachten.

20.2. Automation und Wiederverwendung

In einem klassischen OO-Entwicklungsprozess [JBR99] entsteht aus dem Analyse-Modell inkrementell und iterativ (d.h. durch schrittweise Verfeinerung eines Modellausschnittes, der durch Projektion auf wenige UseCases entsteht) letztlich ein Designmodell, welches so weit verfeinert ist, dass es mehr oder weniger direkt in eine Implementierung überführt werden kann.

Umgekehrt kann ein solches Modell auch vollautomatisch durch Reverse Engineering aus einer Implementierung gewonnen werden. Diese Abstraktionsstufe bezeichnen wir daher als *Implementierungsmodell*, zumal es eben sämtliche Signatur-Details (und in der Regel auch nicht mehr) der Implementierung enthält.

Die Kurve in Abb. 165 zeigt idealisiert den Verlauf der Erstellung eines Software-Inkrements[62] von der Analyse bis zur Implementierung. Mit der Zeit wächst sowohl der gewonnene Informationsgehalt als auch der „Detaillierungsgrad“ der Ausarbeitung des Inkrementes. Zu Anfang werden mehr Informationen gewonnen, gegen Ende mehr Details ausgearbeitet. Ein GUI-Entwurf in Form einer Skizze oder Folie beispielsweise enthält schon fast dieselbe Information wie die ausgearbeitete Implementierung in Form einer JSP-Seite o.Ä. Das heißt, die Ausarbeitung ist im Wesentlichen Fleißarbeit und erhöht zwar den Detaillierungsgrad erheblich, aber den Informationsgehalt nur wenig – dennoch ist diese Arbeit notwendig, um die eigentliche Information in eine computerverarbeitbare Form zu bringen.

Ob dies iterativ geschieht oder nicht, spielt bei dieser Betrachtung nur eine untergeordnete Rolle. Der Aufwand besteht im Wesentlichen gerade aus dem Fortschritt in den beiden genannten Dimensionen, entspricht also qualitativ der Fläche unter der Kurve.

---

[62] Im Spezialfall eines Wasserfallmodells gibt es nur ein globales „Inkrement“.

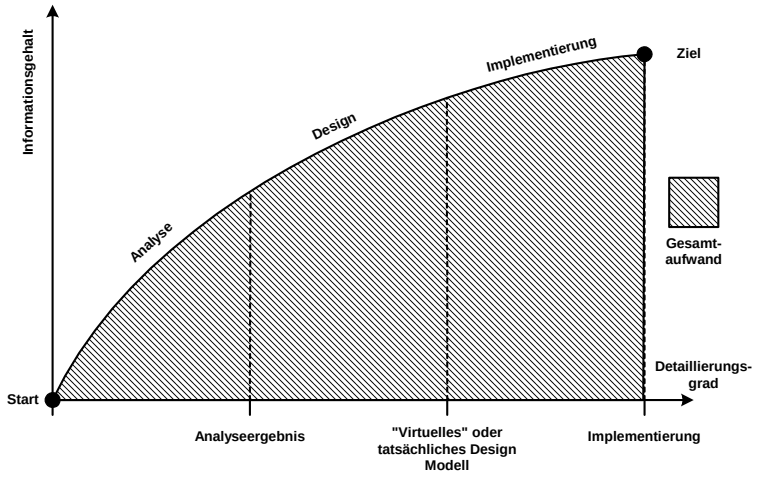


Abb. 165*Aufwand bei traditioneller Softwareentwicklung*

Die Nachteile des Implementierungsmodelles haben wir bereits erläutert. Zusammengefasst:

- Das Implementierungsmodell bietet einen schlechten Überblick, weil die eigentlich wichtigen Informationen in der Fülle der Details untergehen.
- Zur Einarbeitung neuer Teammitglieder ist ein solches Modell relativ ungeeignet
- Der Weg vom Analyseergebnis zum Implementierungsmodell ist sehr lang und es gibt keine definierten „Zwischenstopps“. Die Zwischenergebnisse unterschiedlicher Entwickler auf dem Weg tendieren zur Uneinheitlichkeit, und sie bilden in ihrer Gesamtheit nur selten ein nutzbares Gesamtmodell.
- Designänderungen werden aufgrund des Aufwandes lieber im Sourcecode durchgeführt und anschließend das statische Implementierungsmodell per Reverse Engineering konsistent gemacht. Abstraktere Zwischenergebnisse werden kaum gepflegt, zumal auch oftmals nicht klar ist, welche Auswirkung eine Änderung im Implementierungsmodell auf ein Zwischenergebnis hat oder haben sollte. Letztlich führt dies dazu, dass lediglich (wenn überhaupt) das statische Implementierungsmodell auf einem aktuellen Stand ist. Die dynamischen Aspekte werden vernachlässigt oder sogar aus dem Modell entfernt, weil sie eben nicht mehr passen. Am Ende des Projektes gibt es dann meist keine konsistente Dokumentation, die abstrakter ist als der Sourcecode selbst (vgl. erster Punkt).

Abb. 165 ist bei einer hinreichend abstrakten Sichtweise ebenso gut auch auf agile Prozesse übertragbar. Hier ist das Implementierungsmodell typischerweise nur in „virtueller“ Form vorhanden – nämlich in Gestalt des Sourcecodes. Dadurch entfallen natürlich gewisse Nachteile bzw. es wird bewusst auf gewisse Artefakte verzichtet, die enthaltene Information jedoch muss genau wie bei einem schwergewichtigen Prozess auch im Verlauf der Entwicklung gewonnen werden. Das Gleiche gilt für die fachliche Information (Analyseergebnis), die letztlich in einer Software (oder einem Inkrement) enthalten ist. Mit anderen Worten: Die „Meilensteine“ auf der Kurve heißen je nach Prozess evtl. anders oder sind vielleicht auch gar nicht vorhanden, der *Verlauf* der Kurve ist jedoch *qualitativ* gleich.

Wir reduzieren in dieser Betrachtung Softwareentwicklung bzw. ihren Aufwand also ganz allgemein auf die Faktoren Informationsgewinn und Detaillierung und lassen z.B. etwaige Rückschläge im Informationsgehalt, die durch geänderte Anforderungen oder neue Erkenntnisse entstehen, bewusst außer Acht. Nicht, weil diese Effekte keine Rolle spielen, sondern weil man sie separat und unabhängig von den hier dargestellten Sachverhalten betrachten kann.

Wir wollen nun die essenziellen Potenziale modellgetriebener Softwareentwicklung *Automatisierung* und *Wiederverwendung* auf diese Sichtweise projizieren:

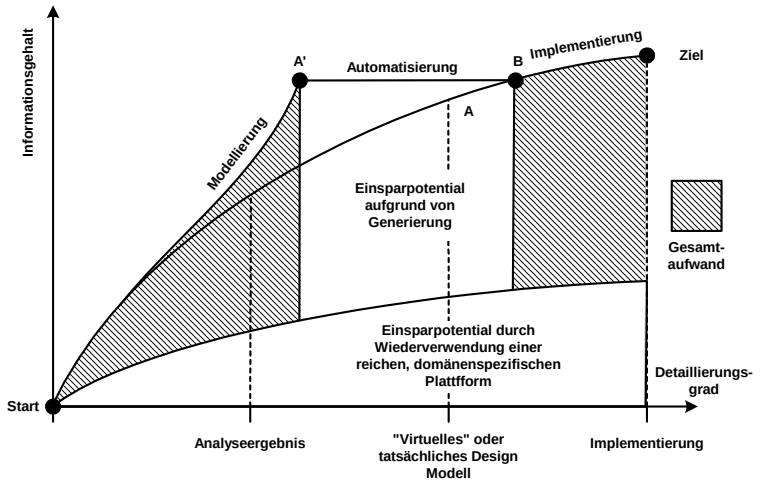


Abb. 166*Aufwand bei modellgetriebener Softwareentwicklung mit teilweise manueller Kodierung*

Hier sehen wir plakativ die Auswirkungen der Abstraktion, d.h. die Verlagerung der Modellierungssprache in Richtung des Problemraumes: Es entstehen sehr kompakte Modelle mit einheitlichem und formal

definiertem Abstraktionsgrad, der erheblich über dem eines Implementierungsmodells liegt (Punkt A' liegt deutlich links vom Implementierungsmodell A). Dennoch besitzen sie durch die formalisierte Semantik sogar einen deutlich größeren Informationsgehalt als die entsprechenden Implementierungsmodelle, zumal eben nicht nur Signaturen, sondern auch (Teil-)Implementierungen definiert sind (Punkt A' liegt oberhalb vom Implementierungsmodell A). Genau hier liegt das Automationspotenzial: Ein Generator kann zwar keine Information gewinnen, aber erhebliche Detaillierungsarbeit leisten. Deswegen verläuft die „Automatisierung" im Diagramm in der Waagerechten und führt ohne Aufwand[63] zu Punkt B. Je nach Domäne und konkretem Ansatz (vgl. <Ref BegrB>) liegt Punkt B mehr oder weniger dicht am Ziel, wodurch der Aufwand der verbleibenden (manuellen) Kodierung bestimmt ist. Im Extremfall spezifiziert das Modell die komplette Semantik der (Teil-)Anwendung, so dass die manuelle Kodierung ganz entfällt (vgl. Abb. 167). Unter welchen Voraussetzungen dies möglich oder sinnvoll ist, erläutern wir später in Abschnitt 20.6.2.

Wie bereits erwähnt kann ein Generator keine Information gewinnen, also muss die Information mittels der domänenspezifischen Modellierungssprache (DSL = Domain Specific Language – siehe <Ref Basis::MDSD-Ansatz> und <Ref Begr::Mod>) bereits im Modell vorhanden sein. Die Kurve steigt also in der Nähe von Punkt A' steiler an als bei Punkt A.

Auch hier gibt es einen Extremfall: Wenn die Modellierungssprache sehr nahe am Problemraum liegt, kann das Analyseergebnis mehr oder weniger direkt in ein formales Modell gegossen werden, d.h. Punkt A' rückt bei gleichem Informationsgehalt nach links an das Analyseergebnis heran oder fällt mit diesem zusammen, wodurch das Automatisierungspotenzial entsprechend wächst. Das bedeutet natürlich andererseits auch, dass bereits die Analyse formaler ausfällt (vgl. Abb. 167).

Führen wir als Beispiel die Automobilbau-Metapher an: Beim Automobilhändler füllen Sie einen einfachen Bestellzettel mit Kfz-Typ und gewünschter Sonderausstattung aus. Die Merkmale auf dem Zettel bilden die domänenspezifische Sprache. Sie müssen sich um Implementierungsdetails wie den Motoraufbau bei der Bestellung nicht kümmern. Die Fabrik erstellt anhand des *fachlichen* Modells (Ihrem Bestellzettel) das gewünschte Produkt (Ihr Auto) unter Verwendung

[63] Den Aufwand der Erstellung eines Generators betrachten wir an dieser Stelle noch nicht.

vorgefertigter Komponenten. Offensichtlich steckt hier die Ingenieursleistung und der Aufwand in der Erstellung der Produktionsstraße und der Abbildung Ihrer „fachlichen“ Bestellung auf diese Produktionsstraße. Die Produktion eines Exemplars selbst ist automatisiert.

Der zweite wichtige Aspekt für eine Effizienzsteigerung ist die Wiederverwendung in Form einer domänenspezifischen Plattform bestehend aus Komponenten, Frameworks etc. (siehe <Ref Basis::MDSD-Ansatz>). Hier wird Aufwand schlicht von der Anwendungsentwicklung auf die Erstellung der Plattform verlagert (vgl. Abb. 166 und Abb. 167), welche allerdings wiederverwendbar ist. Selbstverständlich können solche wiederverwendbaren Artefakte auch in fast allen anderen Entwicklungsprozessen sinnvoll eingesetzt werden und spielen ihr Potenzial dabei auch aus. Ihre Erstellung ist aber in der Regel nicht *Teil der Methode*: Wie bereits erwähnt ist im Kontext modellgetriebener Softwareentwicklung die Plattform gleichsam das Komplement zum Generator und der DSL.

Wir sehen nun deutlich das Einsparungspotenzial, welches durch die Automatisierung im Zusammenspiel mit einer domänenspezifischen Plattform entsteht. Abbildung 4 zeigt den Extremfall. Aus Sicht der Anwendungsentwicklung bietet er offensichtlich maximale Effizienz, im Gegenzug muss aber eine mächtige, „domänenspezifische Software-Produktionsstraße“ aufgebaut werden, um in der Automobilbau-Metapher zu bleiben. Um die Vorteile modellgetriebener Softwareentwicklung ausschöpfen zu können, bedarf es offensichtlich auch einiger Investitionen, die dem Nutzen gegenüberzustellen sind. Doch dazu später mehr.

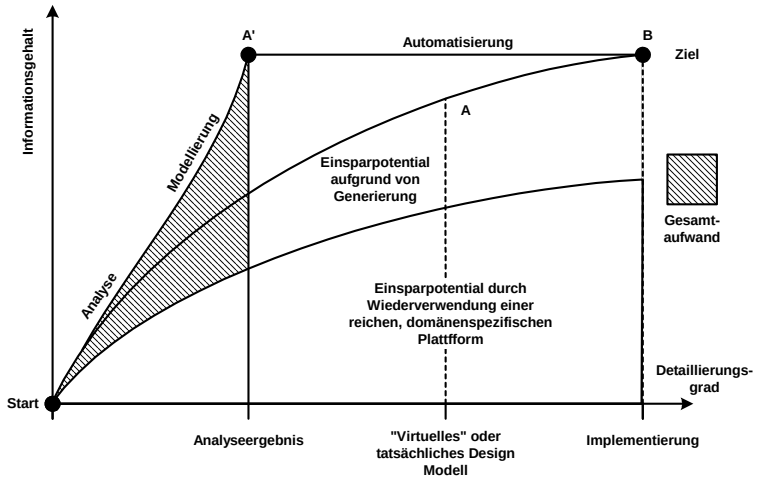


Abb. 167*Aufwand bei modellgetriebener Softwareentwicklung ohne manuelle Kodierung*

Ganz allgemein ergeben sich folgende Vorteile gegenüber einem Standard-Entwicklungsprozess:

- Die Modellierungs-(bzw. „Design“-)Phase endet wesentlich früher. Der Aufwand beträgt nur einen Bruchteil der Erstellung eines Implementierungsmodelles. Der Schlüssel hierfür ist die domänenspezifische Modellierungssprache.
- Der Implementierungsaufwand lässt sich durch den Einsatz eines Generators, der die domänenspezifische Modellierungssprache „versteht“, erheblich reduzieren. Im Falle eines architekturzentrierten Ansatzes (vgl. <Ref Basis::AC) bleibt zwar ein deutlicher Anteil an klassischer Programmierung erhalten, aber die Entwickler können sich auf die Kodierung der eigentlichen Fachlogik konzentrieren. Der architektonische Infrastrukturcode wird generiert (vgl. <Ref CSI>) und dient außerdem als „Führungsschiene“ für die Implementierung. Es entsteht eine formalisierte, generative Softwarearchitektur inklusive Programmiermodell.

20.3. Qualität

In unserer bisherigen Betrachtung haben wir uns ganz auf die Effizienzsteigerung in der Softwareentwicklung konzentriert, doch die Qualität spielt insbesondere durch ihren langfristigen Effekt eine mindestens ebenso wichtige Rolle. Im Folgenden werden wir daher einige, die Qualität betreffende Aspekte betrachten.

20.3.1. Wohldefinierte Architektur

Codegenerierung lässt sich nur sinnvoll einsetzen, wenn die Umsetzung bestimmter Konzepte im Modell auf den Implementierungscode systematisch – also genau definierten Regeln folgend – vor sich geht. Es ist also unerlässlich, diese Regeln zunächst zu definieren. Um den Aufwand der Codegenerierung zu reduzieren, ist es ratsam, mit einer möglichst kleinen und wohldefinierten Menge solcher Regeln auszukommen. Eine kleine Menge wohldefinierter Konzepte und Implementierungs-Idiome zeichnen aber gerade eine gute Architektur aus. Andersherum folgt, dass man sich im Rahmen von MDSD sehr genaue Gedanken über die Architektur der Plattform und der Umsetzung von Anwendungsfunktionalität auf dieser Plattform machen muss. MDSD erzwingt damit eine klare, wohldefinierte Architektur, und es ergibt sich eine dauerhafte Konsistenz zwischen Modell und Implementierung, trotz des hohen Abstraktionsgrades. Entsprechende Generierungstechniken gewährleisten dies auch bei stark iterativer Entwicklung (vgl. <Ref-CSI>).

20.3.2. Konserviertes Expertenwissen

Modellierungssprache, Generator und Plattform bilden eine wiederverwendbare Software-Produktionsstraße für eine bestimmte Domäne. Heutige technische Plattformen wie J2EE, .NET oder CORBA bieten eine große Menge an Basisdiensten für ihre entsprechenden Anwendungsgebiete, J2EE beispielsweise für große Enterprise-Anwendungen. Allerdings ist der *Umgang* mit diesen Diensten – und damit der Plattform an sich – nicht einfach. Man muss sich an bestimmte Patterns oder Idiome halten, um das volle Potenzial einer solchen Plattform voll auszuschöpfen, man muss sie „richtig“ verwenden. Nun sind alle diese Patterns dokumentiert und prinzipiell bekannt. Es ist aber in größeren Projekten ein massives Problem, dafür zu sorgen, dass die typischerweise unterschiedlich gut qualifizierten Mitarbeiter auch alle konsequent die richtigen Patterns einsetzen. MDSD kann dabei helfen, weil die Umsetzung von definierten Domänenkonzepten auf die Plattform *immer gleich* und *automatisiert* stattfindet. Die Transformationen sind damit quasi in maschinenverarbeitbare Form gebrachtes Design-Wissen. Sie stellen damit einen Wert an sich dar, sie konservieren das Wissen über den Umgang mit der Plattform im Rahmen der betreffenden Domäne.

20.3.3. Stringentes Programmiermodell

In der heutigen Praxis ist es oft nicht sinnvoll, 100% des Anwendungscodes zu generieren; in der Regel wird ein Skelett, ein Anwendungsrahmen generiert, in den der Entwickler manuell entwickelten Anwendungscode einbringt. Die Stellen, an denen manuell entwickelter Code eingebaut wird, und die Art und Weise, wie dieser Code strukturiert ist, werden dabei von der Plattformarchitektur und dem generierten Anwendungsrahmen vorgegeben. Damit ist für den Entwickler eine „Führungsschiene“ vorgegeben. „Big-Ball-of-Mud“-Systeme werden damit unwahrscheinlich.

20.3.4. Aktuelle und nutzbare Dokumentation

Im Rahmen von „normaler“ Softwareentwicklung wird die Anwendungslogik vermischt mit technischem Code typischerweise in einer 3GL-Sprache gegen eine bestimmte Plattform programmiert. Dies gilt, wenn auch in weniger starkem Maße, auch für Technologien wie EJB, wo der Applikationsserver bestimmte technische Dienste (wie Transaktionen oder Sicherheit) übernimmt. Weil auf dieser Abstraktionsebene, wie gesagt, viele Konzepte und Zusammenhänge nicht mehr erkennbar sind, muss die Software mit verschiedensten Mitteln dokumentiert werden. Diese Dokumentation in zeitkritischen Projekten manuell synchron zu halten, gelingt meistens nicht und stellt daher in der Regel das erste Opfer dar.

Im Rahmen von MDSD sieht diese Situation grundlegend anders aus. Die einzelnen Artefakte stellen formale Dokumentationen einzelner Systemaspekte dar:

- Das domänenspezifische Modell bietet einen ausgezeichneten Überblick, da immer wiederkehrende Realisierungs-Schemata und Details aus dem Modell herausfaktorisiert sind. Es ist um ein Vielfaches kompakter als ein Implementierungsmodell.
- Die DSL und die Transformationen dokumentieren den Umgang mit der Plattform und die Semantik der Anwendungsdefinitionen (Modelle).
- Die Modelle dokumentieren die Anwendungsstruktur und -logik in einer für Domänenexperten verständlichen Form. Neben dem Modell selbst können aus den Anwendungsmodellelementen und deren Beschreibungen Benutzerhandbücher und andere Dokumente generiert werden. Die generative Erstellung

garantiert, dass die Dokumentation auch über das erste Release hinaus mit der Codebasis konsistent bleibt.

Diese Artefakte sind zwangsläufig auch immer mit der tatsächlichen Anwendung synchron, weil sie als Quelle für die Generierung dienen.

Natürlich bleibt auch im Rahmen von MDSD ein Anteil nicht-formaler Dokumentation. Insbesondere muss die Semantik der DSL mit den zur Verfügung stehenden Konzepten dokumentiert werden, damit sie in der Breite anwendbar ist. Dies ist jedoch nur einmal pro DSL und nicht pro Anwendung nötig (vgl. Abschnitt 20.4).

20.3.5. Qualität von generiertem Code

Generierter Code genießt allgemein einen eher zweifelhaften Ruf: schlecht lesbar, nicht dokumentiert, nicht performant. Dies ist im Kontext von MDSD aber ein unbegründetes Vorurteil, weil die Qualität des generierten Codes direkt von den Transformationen (z.B. Templates) abhängig ist. Letztere wiederum werden idealerweise aus einer Referenzimplementierung abgeleitet (vgl. <Ref Proz::ArchiStrg>). Das heißt, die oben genannten Eigenschaften wie Lesbarkeit oder Performance pflanzen sich aus der Referenzimplementierung bis in die Anwendung fort. Dies ist auch der Grund, weshalb der Referenzimplementierung entsprechende Aufmerksamkeit geschenkt werden sollte. Werden dann die Transformationen mit ausreichender Sorgfalt erstellt, so ist generierter Code qualitativ nicht schlechter als manuell erstellter Code. Im Gegenteil: Generierter Code ist üblicherweise systematischer und konsistenter, weil sich wiederholende Aspekte ja vom gleichen Generator erstellt werden und damit immer gleich aussehen und funktionieren.

Es ist mit den gängigen Generatoren problemlos möglich, den Code beispielsweise vernünftig einzurücken. Auch Kommentare können mit generiert werden. Man sollte auf diese Dinge Wert legen, der Entwickler dankt es spätestens bei der Fehlersuche, die natürlich (auch) im generierten Code stattfindet.

20.3.6. Testaufwand und mögliche Fehlerquellen

Eine Reihe von Aspekten beim Einsatz von MDSD wirkt sich positiv auf die Fehlerrate in der produzierten Software aus. Dies schlägt sich bereits im ersten MDSD-Projekt nieder, kommt aber insbesondere bei Folgeprojekten und der Wartung zur Geltung.

- Modellgetriebene Entwicklung bietet Hilfe bei der Automatisierung von Tests durch Benutzung von generativen Methoden zur Testscript- und Testdatenerzeugung (vgl. <Ref Test>).
- Die Verallgemeinerung von schematischem Code in Form von Transformationen (bzw. Templates) ermöglicht eine neue Herangehensweise zur Validierung der Anwendungsarchitektur im Hinblick auf Skalierbarkeit, Performance und die Abdeckung von ungewöhnlichen Anwendungskonstrukten: Durch die Erstellung eines Anwendungsmodells speziell für Testzwecke lässt sich mit geringem Aufwand eine Testanwendung generieren, welche wesentlich breiter als die manuell entwickelte Referenzimplementierung ist, und welche die Grenzfälle der Anwendbarkeit der Architektur abdeckt. Somit können bereits sehr früh Schwächen in der Architektur erkannt und durch Verfeinerung der Referenzimplementierung und anschließende Verfeinerung der Templates behoben werden. Dies wirkt sich positiv auf das Risikoprofil von Projekten aus, und zeigt, dass MDSD gerade bei riskanten Projekten sinnvoll einsetzbar ist.
- Extensive Tests mit der generierten Testanwendung eliminieren eine ganze Klasse von Tests, welche sonst mit jeder einzelnen generierten Anwendung durchgeführt werden müssten.
- Da schematischer Code typischerweise mehr als 50% der Codemasse ausmacht, werden durch konsequente Automatisierung die möglichen Fehlerquellen bei der Anwendungsentwicklung erheblich gesenkt. Auf längere Sicht ist der Qualitätsvorteil von MDSD mindestens so hoch einzuschätzen wie der Gewinn an Produktivität.
- Die Generierung von Benutzerhandbüchern und anderen Dokumenten auf der Basis von Beschreibungen und Charakteristiken von Anwendungsmodellelementen ist möglich. Die generative Erstellung garantiert, dass die Dokumentation auch über das erste Release mit der Codebasis konsistent bleibt.

20.4. Wiederverwendung

Neben dem Automatisierungspotenzial und der Qualitätssteigerung besteht zusätzlich auch ein sehr großes Wiederverwendungspotenzial für eine Domänenarchitektur. Zur Erinnerung: Dies ist eine Software-

Produktionsstraße bestehend aus einer domänenspezifischen Modellierungssprache[64] (DSL), einer domänenspezifischen Plattform und dazu passenden Transformationen, um vom Modell zur lauffähigen Software zu kommen – und zwar ganz oder teilweise automatisiert (siehe <Ref Begr::SSF>).

Eine solche Produktionsstraße ist dann für ähnliche Softwaresysteme (Anwendungen) mit gemeinsamen Eigenschaften wiederverwendbar. Die Definition von „Ähnlichkeit“ und das damit verbundene Wiederverwendungspotenzial ergibt sich gerade aus den Entscheidungen:

- Festlegung der Domäne
- Festlegung der Implementierungs-Plattform für die Domäne

Damit ist MDSD vor allem im Bereich von Software-Systemfamilien interessant. Eine Software-Systemfamilie ist dabei eine Menge von Anwendungen, die auf ein- und derselben Domänenarchitektur beruhen. Dies bedeutet, dass

- sie auf einer gemeinsamen Plattform ablaufen,
- sie eine gemeinsame DSL zur Spezifikation der Anwendungen (also des „Familienmitglieds“) besitzen,
- es eine gemeinsame Menge von Transformationen gibt, mittels derer die Modelle in ausführbaren Code auf Basis der Plattform überführt werden.

Ein Beispiel: Unsere erste Fallstudie (siehe Kapitel <Ref CSI>) fällt in die Domäne „Architektur für Business-Software“ mit der MDSD-Plattform J2EE/Struts. Die beschriebene Modellierungssprache ist also für alle Softwaresysteme wiederverwendbar, welche die von der Sprache zur Verfügung gestellten Merkmale aufweisen (Schichtung etc.). Die generative Softwarearchitektur des Fallbeispiels bildet die Modellierungssprache auf die Plattform ab. Diese Transformationen sind also für alle Softwaresysteme wiederverwendbar, welche dieselben Architekturmerkmale aufweisen und auf derselben Plattform implementiert werden sollen.

Typischerweise ist die Wiederverwendbarkeit von DSLs auf Grund ihrer Abstraktheit noch größer als die der Transformationen und Plattformen.

---

[64] Vgl. z.B. das architekturzentrierte UML-Profil in der Fallstudie in Kapitel <Ref CSI>.

Im Kontext eines Unternehmens kann es also z.B. sinnvoll sein, die Unternehmensarchitekturen durch entsprechend hochwertige DSLs abzubilden und durch Transformationen (Generatoren) auf die relevanten Plattformen abzubilden.

Im Falle einer klaren fachlichen Ausrichtung, wie z.B. „Software für Versicherungsgesellschaften“, kann eine entsprechend fachlich motivierte DSL sogar noch deutlich effektiver sein (vgl. Abbildung 4), um z.B. die softwaretechnische Umsetzung von Versicherungsprodukten oder -Tarifen modellgetrieben zu unterstützen.

Besonders effektiv ist eine Kombination, genauer gesagt eine Kaskadierung von fachlichen und technischen MDSD-Domänen: Die Plattform einer *fachlichen* Domänenarchitektur kann meist sehr gut mit Hilfe einer *architekturzentrierten* Domänenarchitektur implementiert werden (vgl. <Ref Archi::Plat>. Auf diese Weise kommen die MDSD-Vorteile sowohl in der fachlichen als auch in der technischen Dimension zum Tragen.

In jedem Falle gilt: Je mehr Anwendungen oder Anwendungsteile auf diesen Software-Produktionsstraßen entstehen, desto schneller rentiert sich deren Erstellung, und desto größer ist der Gewinn.

20.5. Portabilität, Änderbarkeit

Ein wichtiger Benefit modellgetriebener Softwareentwicklung ist der so genannte Fan-out. Dieser beschreibt die Tatsache, dass aus *einem* Modell eine ganze Reihe weniger abstrakte Artefakte generiert werden können. Dabei gibt es zwei Dimensionen dieses Fan-outs:

Zum einen lassen sich im Rahmen eines Projektes aus einem Modell verschiedene Artefakte generieren. Beispielsweise können aus einem UML-basierten Datenmodell relationale Datenbankschemata, XML-Schemata sowie Serialisierer für verschiedene Programmiersprachen generiert werden. In Projekten, wo all diese verschiedenen Artefakte gegenseitig konsistent gehalten werden müssen, ist dieser Vorteil erheblich.

Zum anderen lassen sich aus einem Anwendungsmodell im Laufe der Zeit verschiedene Implementierungen generieren. Damit wird die Migration auf neue Versionen der technischen Plattform bzw. auf eine ganz neue Plattform erleichtert.

Generell erlaubt der MDSD-Ansatz eine schnelle Änderung der erstellten Anwendung. Dadurch, dass viele Artefakte automatisch aus *einer* Spezifikation erstellt werden, wirken sich Änderungen an dieser

einen Spezifikation natürlich auf das gesamte System aus. Damit steigt die Agilität der Projektabwicklung.

20.6. Investitionen und erzielbare Gewinne

Bisher haben wir die Potenziale und den Nutzen modellgetriebener Softwareentwicklung betrachtet. Doch leider gibt es auch in der IT kaum etwas geschenkt: Um in den Genuss der Vorteile zu kommen, muss man zunächst Investitionen in Training und Infrastruktur tätigen – sowohl technischer als auch (je nach Ausbaustufe) organisatorischer Art. In diesem Abschnitt wollen wir einige Erfahrungswerte und anschauliche Verdichtungen aus realen Projekten vorstellen, damit Sie sich ein genaueres Bild über Kosten, Nutzen und die Gewinnschwelle modellgetriebener Softwareentwicklung machen können.

20.6.1. Architekturzentrierte MDSD

Wir empfehlen den Einstieg in MDSD über architekturzentrierte MDSD, da dies die geringste Investition erfordert und sich der erforderliche Einführungsaufwand schon innerhalb eines sechsmonatigen Projekts durchaus rentieren kann. Architekturzentrierte MDSD setzt keine fachlich motivierte domänenspezifische Plattform voraus, und beschränkt sich im Wesentlichen auf die automatische Generierung sich wiederholenden Codes, welcher bei der Benutzung von kommerziellen und Open-Source-Frameworks bzw. -Infrastrukturen typischerweise anfällt.

Die Investition besteht in erster Linie aus dem Lernaufwand zur Bedienung eines MDSD- Generators und der dazugehörenden Template-Sprache sowie der Definition einer geeigneten DSL zur Modellierung.

Auswirkungen auf die zu pflegende Codemasse

Um eine quantitative Analyse vornehmen zu können, haben wir Messungen an einem realen MDSD-Projekt zum Zeitpunkt der Abnahme vorgenommen. Es handelt sich bei dem Projekt um eine strategische, web-basierte Backoffice-Anwendung für einen großen Finanzdienstleiter. Tab. 2 zeigt (gerundet), welche Auswirkung architekturzentrierte MDSD auf die Quellcode-Masse hat.

Menge des Quellcodes [kBytes]	Traditionelle Entwicklung	MDSD
Quellcode Referenzimplementierung	1,000	1,000
von Hand geschriebener Code	18,800	2,200
Modelle		3,400
Transformationen		200
Gesamt	19,800	6,800

Tab. 2 *Code-Massenverteilung bei architekturzentrierter MDSD*

Als Modell-Masse wurde hier einfach die Gesamtgröße der UML-Tool-spezifischen Dateien zur Speicherung in KB eingetragen. Des Weiteren wird angenommen, dass in jedem Falle eine manuell erstellte Referenzimplementierung entwickelt wird, um die Anwendungsarchitektur zu validieren.

Insgesamt ergibt sich in unserem konkreten Beispiel bei Einsatz von architekturzentrierter MDSD eine Reduzierung der manuell zu pflegenden Quellcode-Masse auf 34%. Diese Zahl mag sehr hoch erscheinen, ist aber nach unseren Erfahrungen durchaus repräsentativ. Nicht selten veröffentlichen Werkzeughersteller Zahlen über den Prozentsatz an generiertem Code. Solche Zahlen sind aber nicht aussagekräftig, wenn nicht klar ist, ob die erforderlichen Modell- und Transformationsquellen wie in Tab. 2 mit in die Rechnung einbezogen wurden. In unserem Beispiel machen Modell- und Transformationsquellen über 50% des zu pflegenden Quellcodes aus. Abb. 168 zeigt die Zahlen aus unserem Beispiel in grafischer Form.

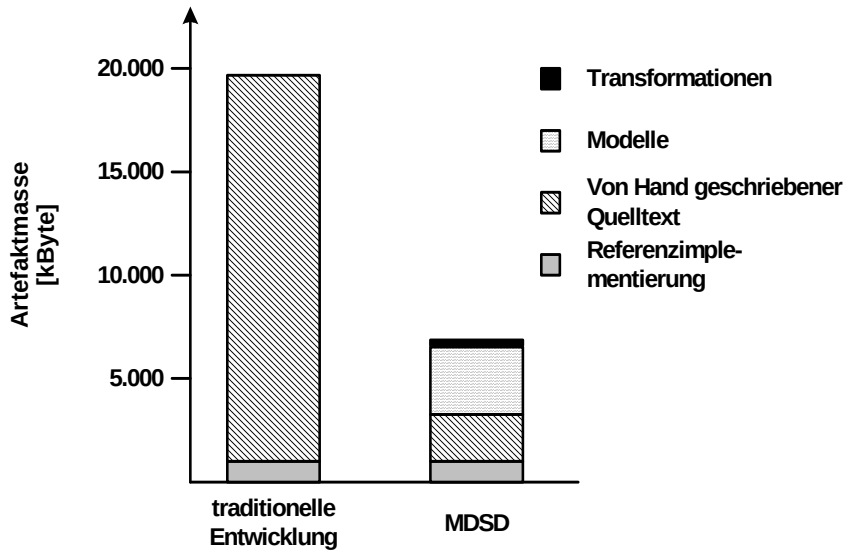


Abb. 168 *Code-Massenverteilung*

Natürlich bleibt beim Einsatz von MDSD die Menge an zu kompilierenden Quellen gegenüber manueller Implementierung unverändert, siehe Abb. 169.

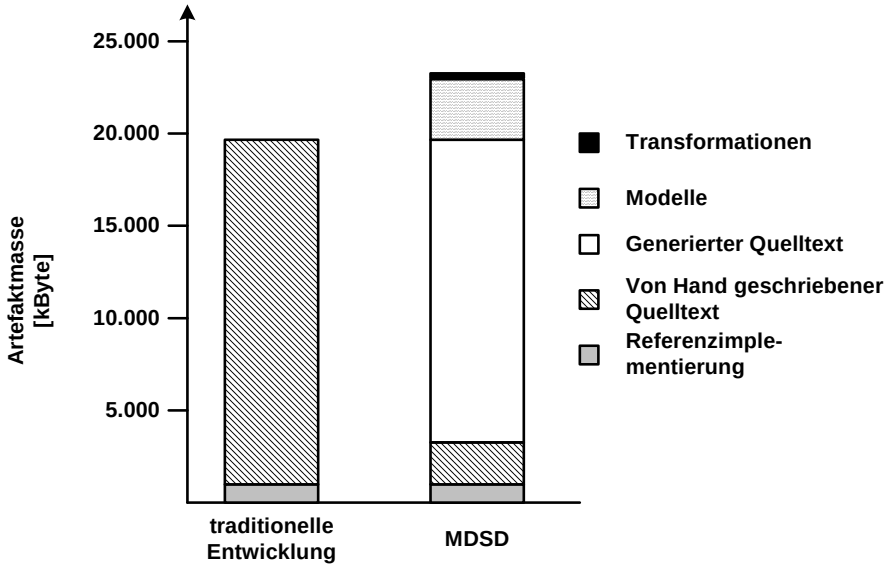


Abb. 169 *Verhältnis von generierten „Quellen" zu manuell erstellten Quellen*

Der Unterschied ist vielmehr, dass generierter Code keine Quelle darstellt, sondern ein aufwandsneutrales Zwischenergebnis

Wenn wir die Modell- und Transformationsquellen sowie die Referenzimplementierung unter den Tisch fallen lassen, dann ergibt sich das Verhältnis von generiertem Code zu manuell erstelltem Code als 88% zu 12%. Wir halten solche Zahlen allerdings für Augenwischerei. Die Modelle nehmen in MDSD den gleichen Stellenwert an wie „normaler“ Quellcode, und die Referenzimplementierung sollte bei MDSD immer mitgepflegt werden, da sie als Basis für alle Verfeinerungen und Erweiterungen der Architektur dient. Bei dieser Betrachtungsweise ergibt sich das Verhältnis von generiertem Code zu manuell erstelltem Code als 72% zu 28%. Diese Zahlen sind immer noch beeindruckend genug, um erkennen zu lassen, dass sich architekturzentrierte MDSD bereits bei der Entwicklung einer einzelnen Anwendung rechnen kann, insbesondere vor dem Hintergrund des mittelfristigen Wartungsaufwandes. Um dies zu belegen, müssen wir allerdings den gesamten Projektaufwand berücksichtigen und uns nicht nur auf die Programmierung beschränken.

Auswirkungen auf den Projektaufwand

Der Projektaufwand beinhaltet insgesamt neben der Implementierung zumindest im Kern

- Analyse und Dokumentation der Anforderungen,
- Architektur- und Design-Arbeit, ggf. Definition der Modellierungssprache (z.B. UML-Profil),
- Testerstellung und Durchführung,
- Projektmanagement.

Darüber hinaus können noch dem Projekt vorgelagerte Geschäftsprozessanalyse, zu erstellende Produktdokumentation (Benutzerhandbücher, Hilfetexte) sowie Produktionsaufwände zu Buche schlagen. MDSD hat auf einige dieser Aktivitäten einen positiven Einfluss, welcher sich aber nur schwer in allgemeingültigen Zahlen ausdrücken lässt. Deshalb wollen wir uns darauf beschränken, zu untersuchen, inwieweit sich MDSD auf den Implementierungsaufwand auswirkt und wie sich dies auf den Gesamtaufwand für die Kernaktivitäten niederschlägt.

Die Zahlen aus dem vorherigen Abschnitt sind für eine Betrachtung des Arbeitsaufwandes nur begrenzt geeignet, weil sie in keiner Weise berücksichtigen, wie viel Zeit – und somit Geld – für die Erstellung von Modellen und Transformationen sowie für die manuelle Programmierung von Referenzimplementierungen und anwendungsspezifischem Quellcode anfällt. Auch ist die „Denkarbeit“ zur Erstellung der DSL wie gesagt nicht in der Betrachtung enthalten.

In einem kleinen praktischen Experiment [Bet02] haben wir untersucht, wie viel Arbeit die Erstellung und Pflege von Modellen in Form von Dateneingabe und Mausklicks darstellt, und diese Information über eine grobe Formel in Quellcodezeilen-Äquivalente umgerechnet. Wenn man dann den insgesamt anfallenden, äquivalenten Programmieraufwand als die Summe von verbleibenden manuell erstellten Quellcodezeilen und der sich aus den Modellen ergebenden Quellcodezeilen-Äquivalente definiert, ergibt sich ein interessantes Bild:

- In dem Experiment sinkt durch MDSD der Programmieraufwand auf 48% gegenüber vollständig manueller Programmierung.
- Wenn MDSD nicht verwendet wird und stattdessen herkömmliche UML-Werkzeuge zum Einsatz kommen, um aus Modellen Skelettquellcode zu generieren, steigt der Programmieraufwand auf 105% bis 149% gegenüber vollständig manueller Programmierung ohne UML-Einsatz, je nachdem in welchem

Ausmaß die Modelle neben Klassendiagrammen auch Interaktionsdiagramme enthalten, welche nicht zur Skelettgenerierung beitragen.

Diese Zahlen zeigen deutlich, dass MDSD nicht dasselbe ist wie UML Roundtrip Engineering, und sie zeigen auch, warum viele Softwareentwickler dem Einsatz von UML-Werkzeugen skeptisch gegenüberstehen.

Bei architekturzentrierter MDSD besteht die Plattform fast ausschließlich aus externen kommerziellen und Open Source Frameworks, und die zu erstellende Domänenarchitektur besteht im Wesentlichen aus einer Referenzimplementierung und daraus abgeleiteten Transformationen (Generierungsregeln).

Die Zahlen aus dem Beispiel des letzten Abschnitts (Referenzimplementierung ca. 1.000 kB Quellcode, Transformations-Quellcode ca. 200 kB) unterstützen unsere Behauptung, dass die Ableitung von Transformationen aus einer Referenzimplementierung nur zwischen 20% und 25% des Aufwands zur Erstellung der Referenzimplementierung darstellt – insbesondere wenn man berücksichtigt, dass ein Großteil der Denkarbeit bereits in der Referenzimplementierung steckt.

Der Aufwand für die Erstellung einer Referenzimplementierung wird durch MDSD nicht beeinflusst, und für die weiteren Überlegungen nehmen wir an, dass bei manueller, herkömmlicher Programmierung der Programmieraufwand für die Referenzimplementierung 15% des Programmieraufwandes eines Projekts darstellt. Im Beispiel des letzten Abschnitt war die Größe der Referenzimplementierung in kB gemessen nur 5% der Größe der gesamten Anwendung, so dass selbst unter Berücksichtigung des höheren Schwierigkeitsgrades bei der Referenzimplementierung 15% eine konservative Schätzung darstellt.

Entwicklungsaufwand [%]	**Tranditionelle Entwicklung**	**MDSD**
Transformationen	-	4 (0.15 * 25)
Referenzimplementierung	15	15
Applikationsmodelle und -Code	85	41 (0,48 * 85)
Gesamt	**100**	**60**

Tab. 3 *Vergleich von Modellierungs- und Programmieraufwand*

Aus Tab. 3 ergibt sich, dass architekturzentrierte MDSD unter den oben erläuterten Annahmen den Programmieraufwand um 40% senken kann. Diese Zahl deckt sich gut mit unseren praktischen Erfahrungen und bietet

einen Anhaltspunkt bei Kostenüberlegung zur Einführung von MDSD. Sie sollten am besten Metriken aus ihrem eigenen Team verwenden, um den Anteil des Programmieraufwandes am Gesamtprojektaufwand zu ermitteln und daraus das Potenzial von MDSD abzuleiten.

Angenommen die Programmierung stellt 40% der Projektaktivitäten dar, dann ergibt sich eine potenzielle Kosteneinsparung von bis zu 16%.

Bei Folgeprojekten entfällt der Aufwand für Referenzimplementierung und die Transformationsentwicklung, so dass sich der Programmieraufwand um 59% reduziert und sich die potenzielle Kosteneinsparung auf 24% erhöht.

In der Realität muss natürlich der Lernaufwand des Teams sowie der Einsatz eines MDSD-Experten als Trainer im ersten Projekt mit einkalkuliert werden.

Eine empirisch begründete Einschätzung: Wenn die Dauer eines MDSD-Pilotprojektes sechs Monate oder mehr beträgt, und wenn das Team aus mehr als fünf Personen besteht, kann sich die MDSD-Einführung schon innerhalb dieses ersten Projektes amortisieren, selbst wenn die „Produktionsstraße“ komplett neu aufgebaut werden muss. Dazu sollten die in Kapitel <Ref Adopt> behandelten Adaptionsstrategien und Voraussetzungen berücksichtigt werden.

20.6.2. Fachlich ausgerichtete MDSD-Domänen

Wie Sie vermutlich bereits aus dem vorherigen Abschnitt entnommen haben, bietet architekturzentrierte MDSD einen einfachen und risikoarmen Einstieg in modellgetriebene Softwareentwicklung. Wie die Zahlen aus unseren Beispielen zeigen, bestehen die zu pflegenden Quellen ungefähr zur Hälfte aus Modellen und Transformationen, und zur anderen Hälfte aus herkömmlichem Quellcode für die Referenzimplementierung und anwendungsspezifische Funktionalität.

Wenn eine ausgereifte Implementierung architekturzentrierter MDSD vorliegt, lässt sich der verbleibende traditionelle Quellcode nicht weiter reduzieren. Eine weitere Effizienzsteigerung ist nur durch die Verlagerung des MDSD-Paradigmas von der *architektonisch/technischen* Ebene auf die *fachliche* Ebene möglich. Dann besteht die Möglichkeit durch Domänenanalyse bzw. Product Line Engineering <Ref PLE> *fachliche* Gemeinsamkeiten und Variabilitäten zwischen Anwendungen aufzudecken und die Abstraktionsebene der Anwendungsmodelle weiter zu erhöhen: Die Modelle beschreiben dann fachliche Problemstellungen

und Konfigurationen statt architektonische Lösungen. Dies bewirkt eine weitere Reduktion des Modellierungs- und insbesondere des Programmieraufwandes. Zur Veranschaulichung ist es hilfreich, die Code-Massenverteilung bei der Entwicklung einer Reihe von fachlich ähnlichen Anwendungen zu betrachten. Abb. 170 skizziert die Reduzierung der Codemasse durch die Einführung einer fachlichen Domänenarchitektur bei der Entwicklung von drei Anwendungen unter der Annahme, dass sich die Anwendungsmodelle und der anwendungsspezifische, manuell kodierte Quellcode auf 50% reduzieren lassen. Wir möchten betonen, dass Abb. 170 rein illustrativen Charakter hat, und dass die tatsächlichen Verhältnisse stark von der Domäne und von ihrer Breite abhängen. Je enger sich der Rahmen der Domäne abstecken lässt, desto mehr Funktionalität lässt sich in wiederverwendbarer Form in einer fachlichen MDSD-Plattform verankern.

In der Praxis ist die Entwicklung einer fachlichen Domänenarchitektur ein inkrementeller Prozess, welcher auf der Basis architekturzentrierter MDSD aufsetzt.

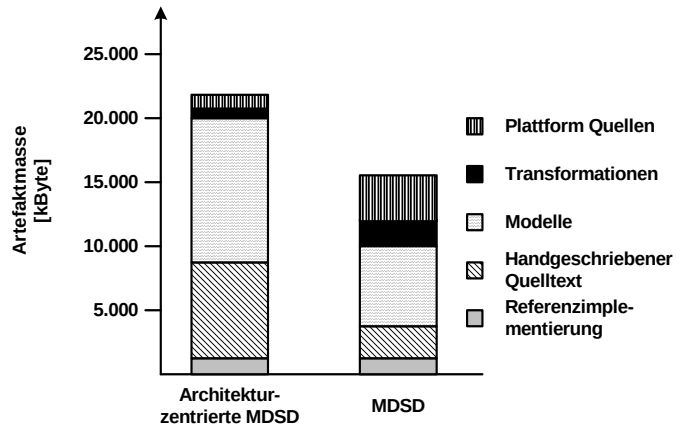


Abb. 170 *Code-Massenverteilung bei Einführung einer fachlich ausgerichteten Domänenarchitektur*

Der Aufwand der Entwicklung fachlich motivierter, domänenspezifischen Sprachen (DSLs) und entsprechender Frameworks ist nicht zu unterschätzen. Die erfolgreiche Entwicklung solcher Sprachen setzt erhebliche Erfahrungen in der Domäne voraus und sollte nicht im ersten MDSD-Projekt versucht werden.

Es gibt natürlich auch den Fall, dass bereits funktionsstarke fachliche Frameworks vorliegen, welche ohne Zuhilfenahme von modellgetriebenen Generatoren benutzt werden. In diesem Fall kann MDSD wie im architekturzentrierten Fall aufsetzen, d.h. die bestehenden

Frameworks werden als MDSD-Plattform angesehen, welche die Zielarchitektur (siehe <Ref Archi>) definieren. Aus den Konfigurationsmöglichkeiten wird dann eine DSL abgeleitet. Ähnlich wie im architekturzentrierten Fall lassen sich dann mit bereits geringen Investitionen erhebliche Produktivitätssteigerungen erzielen.

20.7. Kritische Fragen

In den vorangegangenen Abschnitten und auch in Kapitel <Ref EuA> haben wir bereits einige Vorurteile gegenüber modellgetriebener Softwareentwicklung angesprochen. Einige dieser Vorurteile haben ihren Ursprung in den 80er und 90er Jahren und gehen auf negative Erfahrungen mit CASE-Werkzeugen zurück, welche jetzt aus verständlicher Skepsis auf modellgetriebene Verfahren im Allgemeinen projiziert werden. Auf der anderen Seite gibt es auch ernstzunehmende kritische Fragen, deren Beantwortung zumindest nicht trivial oder offensichtlich ist. Im Folgenden wollen wir auf beide Kategorien (ohne sie explizit zu differenzieren) eingehen:

Frage: Was ist neu an MDSD?

Einzelne Ideen, wie z.B. die Generierung von Quelltext aus höherwertigen Beschreibungen/Modellen, sind natürlich nicht neu. Jedoch sind die einfachen Codegeneratoren, die jeder Entwickler irgendwann selbst einmal schreibt, nicht zu vergleichen mit den hochflexiblen MDA/MDSD-Werkzeugen neuester Generation, die es erlauben, Modellierungssprachen und modulare Transformationen auf einfache Weise und dennoch frei zu definieren. MDSD ist zudem nicht bloß eine Technik – das Engineering-Prinzip steht im Vordergrund.

Frage: Wenn der Ansatz so gut ist, warum ist er dann nicht schon längst in der Breite etabliert?

Die notwendigen Voraussetzungen wie entsprechend flexible Werkzeuge und vor allem ein breiteres Know-how sind noch nicht sehr lange verfügbar. Im Bereich der Werkzeuge spielt die steigende Popularität der MDA sicherlich eine treibende Rolle. Auf der anderen Seite hat sich die ganzheitlich architekturzentrierte MDSD-Sichtweise und insbesondere ihre generative Umsetzung bei den so genannten „Early Adopters“ bereits durchgesetzt. Gleiches gilt für die zentralen prozessualen MDSD-Konzepte.

Frage: Wirkt sich MDSD nachteilig auf die Performance oder die Quelltext-Lesbarkeit der erstellten Anwendungen aus?

Im modellgetriebenem Ansatz werden die notwendige Infrastruktur und entsprechende Entwurfsmuster in Form einer kleinen, manuell erstellten Referenzimplementierung erstellt. Die Referenzimplementierung ist schmal, aber tief, d.h. sie deckt alle Schichten der Architektur ab und wird zu Projektbeginn von den erfahreneren Teammitgliedern erstellt. Im Laufe des Projekts wird die Referenzimplementierung nach Bedarf verfeinert und optimiert, so dass mittels generativer Techniken diese Eigenschaften und Verbesserungen auf den gesamten Quellcode übertragen werden können. Die Performance ist also beim modellgetriebenen Ansatz im Schnitt mindestens so hoch wie im traditionellen Ansatz. Das Gleiche gilt für die Lesbarkeit des Quelltextes (vgl. <Ref Trans::Pretty>).

Frage: Sind UML-basierte IDEs mit Roundtrip-Unterstützung mittlerweile nicht wesentlich ausgereifter als MDA/MDSD-Werkzeuge?

Das mag zum Teil vielleicht sogar zutreffen, ist allerdings irrelevant, da diese Tools eine andere Klasse von Werkzeugen darstellen. Mit ihnen lassen sich die o.g. MDSD-Vorteile wegen des mangelnden Abstraktionsgrades schlichtweg nicht erreichen (vgl. Abschnitt 20.2 und <Ref EuA>).

Frage: Werden bei MDSD keine unvertretbar starken Bindungen an ein bestimmtes Werkzeug oder an eine bestimmte Technologiekombination erzeugt?

Modulare, automatisierte Transformationen und die Betrachtung von Modelltransformationen als erstrangige Artefakte stellen sicher, dass Werkzeugabhängigkeiten lokalisiert bleiben (siehe <Ref Tool::Archi>). Da die MDA noch keinen Standard für Transformationssprachen zur Verfügung stellt, legt man sich mit der Wahl des MDA/MDSD-Werkzeuges auf eine spezifische Transformationssprache fest. Allerdings können MDSD-Werkzeuge zunehmend als allgemeine Gebrauchsgüter betrachtet werden, insbesondere wenn sie in Form von Open-Source-Software vorliegen. Da moderne Template-basierte Generatoren keine Beschränkungen im Bezug auf die Zielsprachen beinhalten, ist keinerlei Einschränkung in Bezug auf bestimmte Technologiekombinationen für die zu erstellenden Anwendungen vorhanden.

Frage: Impliziert der modellgetriebene Ansatz nicht ein Wasserfall-Modell, und steht er nicht insbesondere im Widerspruch zu Agilität? Was ist der Unterschied zu CASE?

Der iterative, zweigleisige Ansatz von MDSD (vgl. <Ref-Proz::IterX2>), in welchem die Infrastrukturentwicklung parallel zur Anwendungsentwicklung erfolgt, unterscheidet sich erheblich von

traditionellen Wasserfallmethoden, denen eine „big design up-front"-Philosophie zugrunde liegt. MDSD basiert auf domänenspezifischen Plattformen und steht im krassen Gegensatz zum CASE-Ansatz, wo versucht wird, mittels einer universellen Sprache quasi allen Situationen gerecht zu werden. Agile Methoden eignen sich insbesondere für die Erstellung von MDSD-Referenzimplementierungen. Darüber hinaus hilft MDSD agile Methoden zu skalieren, da gewonnene Erfahrung und Expertenwissen in Softwareform für alle Entwickler verfügbar gemacht werden können. MDSD fördert testgetriebene Verfahren (vgl. <Ref Test>).

Frage: Wie wird sichergestellt, dass Modell und Code nicht irgendwann auseinander laufen?

In MDSD wird konsequent vom Modell aus generiert, und generierter Code wird konsequent von nicht-generiertem Code getrennt. Handgeschriebener Code fällt nicht völlig weg. Im Gegenteil, MDSD betrachtet Entwickler und Generatoren als sich ergänzende Partner, die effizient zusammenarbeiten müssen. Die Regeln für die Zusammenarbeit zwischen Entwicklern und Generatoren sind nicht starr, sondern werden entsprechend den Erfordernissen festgelegt, genauso wie die Regeln für das Zusammenspiel zwischen mehreren Entwicklern im Team. Allerdings sorgt der MDSD-Generator dafür, dass die Entwickler die von der Domänenarchitektur gesetzte Grenze nicht überschreiten können, zumal bei iterativer Generierung Manipulationen am Generat hinfällig werden.

Frage: Wie hoch schlägt der Aufwand der Generator- bzw. Transformations-Entwicklung zu Buche? Lohnt sich der Aufwand?

Es stehen (auch oder gerade im Open-Source-Umfeld) schlanke, aber ausgereifte, generische Werkzeuge zur Verfügung, die einfache und dennoch mächtige Template-Sprachen mitbringen. Auf diese Generatoren kann und sollte sich die Entwicklung der Transformationen stützen.

Moderne Template-Sprachen ermöglichen eine sehr intuitive Generalisierung von komplexem manuell geschriebenem Code. Der Aufwand für die Generalisierung oder „Templatisierung" von Quellcode beträgt nur 20% bis 25% des Aufwandes, welcher benötigt wird, um eine Referenzimplementierung manuell zu kodieren. Ab einer gewissen Projektgröße und insbesondere bei gleichartig gelagerten Folgeprojekten ist der MDSD-Ansatz auf Grund der Automation daher deutlich effizienter als die traditionelle Softwareentwicklung (siehe Abschnitt 20.6).

Frage: Dauert nicht z.B. das Ändern eines Attributtyps im Modell mit anschließendem Generieren länger, als die Deklaration im Code an einer Stelle von Hand zu ändern?'

Bei reicher Semantik wie z.B. Persistenz des Attributs genügt eben nicht eine Stelle im Code, sondern alle Stellen in Interfaces (Getter/Setter) bis zur erzeugten DDL müssen konsistent angepasst werden.

Frage: Ist es nicht schneller, nachträgliche Änderungen durch manuelle Anpassung im generiertem Code nachzuziehen, als die MDSD-Transformationen (Generierungsvorschriften) anzupassen?

Wenn die Richtlinien für MDSD befolgt werden und generierter Code strikt von nicht-generiertem Code getrennt wird sowie iterativ die Referenzimplementierung verfeinert wird, dann sind Änderungen an Code-Templates immer schneller durchführbar als manuelle Änderungen von Entwurfsmustern in zahlreichen Quelldateien.

Frage: In zeitkritischen Phasen der Anwendungsentwicklung kann möglicherweise nicht auf das nächste Release der Domänenarchitektur gewartet werden. Dann werden manuelle Eingriffe in das Generat gemacht und der MDSD-Ansatz scheitert, weil sich der Generator dann nicht mehr einsetzen lässt. Kann das verhindert werden?

Bei kleineren Patches kann es sinnvoll sein, auf dem Anwendungsentwicklungsstrang einen spezifischen, temporären Branch der Domänenarchitektur zu erstellen, der beim nächsten Release der Domänenarchitektur stillgelegt wird. Zeigen sich gravierendere Unzulänglichkeiten der Domänenarchitektur, so muss (temporär oder permanent) auf manuelle Programmierung ausgewichen werden. Dies geschieht dann aber nicht durch Manipulation des Generats, sondern durch Öffnung der Domänenarchitektur an der betreffenden Stelle: Die Transformationen werden so geändert, dass sie an der betreffenden Stelle nur eine (veränderbare) Default-Implementierung erzeugen. Die Öffnung der Domänenarchitektur kann ggf. auch wieder als temporärer, projektspezifischer Branch zur Verfügung gestellt werden.

Frage: Ist es nicht besser, ein objektorientiertes Framework zu implementieren als einen Generator? Gibt es objektorientierte Generatoren und hochqualitatives objektorientiertes Generat?

Objektorientierte Frameworks und Generatoren passen ideal zusammen (vgl. <Ref CSI>). Die Kombination beider Ansätze stellt einen echten Fortschritt dar.

In MDSD ergänzen sich Frameworks und Generatoren ähnlich wie die dynamischen und strukturellen Aspekte eines objektorientierten

Modells. Moderne Generatoren sind typischerweise in objektorientierten Sprachen realisiert und sind in keiner Weise vergleichbar mit den einfachen Hilfswerkzeugen, die jeder Programmierer irgendwann einmal geschrieben hat.

Frage: Modellgetriebene Entwicklung in größeren Teams bedeutet zwangsweise verteilte Modellierung. Stellen diesbezüglich die aktuell verfügbaren (UML-)Modellierungswerkzeuge nicht eher ein Problem als eine Hilfestellung dar ?

Natürlich ist die Auswahl des Modellierungswerkzeuges essenziell für den Erfolg von MDSD in großen Projekten. Es gibt aber durchaus ausgereifte (UML-)Werkzeuge, die eine verteilte Modellierung auch in großen Teams unterstützen. Unabhängig davon kann die schlichte Partitionierung des Anwendungsmodelles in mehrere, lose gekoppelte Teilmodelle hilfreich sein. Die Partitionierung kann horizontal (z.B. entlang der Architekturschichten) oder vertikal (z.B. entlang lose gekoppelter UseCase-Implementierungen) erfolgen. Die Partitionierung muss durch die DSL und das Generatorwerkzeug zugelassen bzw. unterstützt werden (siehe <Ref Vers>).

Frage: Ist das explizite Ausarbeiten einer Architektur nicht viel zu aufwändig?

Jede Anwendung besitzt eine mehr oder weniger konsequent umgesetzte Architektur – aber nur durch explizite Ausarbeitung (in welcher Form auch immer) wird es eine „gute“ Architektur. Qualitätsattribute wie Wartbarkeit, Skalierbarkeit usw. können iterativ entstehen, aber nicht zufällig.

Frage: Ist handgeschriebener Code nicht zuverlässiger als generierter Code? Kann man einem Generator in allen Situationen trauen, und wird es nicht immer Situationen geben, wo manuelle Eingriffe erforderlich sind?

Es ist falsch, davon auszugehen, dass ein paar „unvorhersehbare“ Ausnahmen zu einem Entwurfsmuster den Einsatz von Generatoren unpraktisch oder unökonomisch machen. Es ist nicht das Ziel von MDSD, die manuelle Erstellung von Quellcode vollständig zu eliminieren. Vielmehr bietet MDSD pragmatische Techniken, um generierten Code mit nicht-generiertem Code an entsprechenden Stellen zu ergänzen, und zwar ohne die Möglichkeit von iterativer Neugenerierung zu gefährden.

Als Beweis für die höhere Effizienz und geringere Fehleranfälligkeit von handgeschriebenem Code werden gelegentlich obskure Compiler- oder Assemblerfehler angeführt. Dabei wird übersehen, dass diese angebliche Achillesferse der Generatortechnologie schon längst die

Entwicklung und Benutzung der heutigen Programmiersprachen verhindert hätte.

Im Gegenteil zeigt die Praxis, dass bei der Ableitung von Templates aus der Referenzimplementierung durch die intensive Beschäftigung mit den implementierten Entwurfsmustern üblicherweise eine ganze Reihe von Fehlern aufgedeckt wird, die ansonsten unerkannt mittels traditioneller Copy & Paste-Technik über den gesamten Quellcode weiterverbreitet würden.

Wahr ist allerdings, dass die Trennlinie zwischen Generat und manueller Implementierung durch die Domänenarchitekten sorgfältig austariert werden muss. Es ist ein Balanceakt zwischen Automation und notwendigen Freiheitsgraden. Oftmals ist aber gerade eine Beschränkung von Lösungsalternativen innerhalb einer Domänenarchitektur zielführend (siehe <Ref Archi>).

20.8. Zusammenfassung

Mit architekturzentrierter MDSD sind bereits innerhalb eines Projektes Produktivitäts- und Qualitätsgewinne möglich, insbesondere sinkt der Modellierungs- und Programmieraufwand auf ungefähr die Hälfte gegenüber traditioneller manueller Programmierung. Unter Berücksichtigung des notwendigen Trainingsaufwands zur Einführung von MDSD sind echte Kosteneinsparungen erst ab dem zweiten Projekt zu erwarten, wenn das Team mit dem MDSD-Paradigma und einem konkreten Werkzeugsatz und der Methodik vertraut ist. Der Einsatz von MDSD mit fachlichen MDSD-Plattformen sollte der zweite Schritt sein, welcher sich insbesondere bei der Entwicklung von Produktlinien anbietet (vgl. <Ref PLE>).

20.9. Weiterführende Literatur

Die Menge an öffentlich zugänglichem Datenmaterial zur Effizienz von modellgetriebener Softwareentwicklung und zur Produktlinienentwicklung ist leider begrenzt. Dies hat mehrere Gründe: Zum einen wird niemand auf die Idee kommen, ein reales Projekt in zweifacher Ausführung parallel durchzuführen, nur um Metriken zu sammeln – und selbst das wäre wegen differierender Randbedingungen nur bedingt aussagekräftig. Des Weiteren ist oft der Zeit- und Qualitätsgewinn für alle Projektbeteiligten offensichtlich, und die interessanten Metriken beziehen sich mehr auf die inkrementelle

Verfeinerung der Verfahren als auf den Vergleich mit vollständig manuellen Verfahren. Zu guter Letzt weisen nicht alle Firmen auf die Verwendung von generativen Techniken und die damit verbundenen Einsparungen hin, wenn der Kunde sich nur für den gelieferten, traditionellen Quellcode oder das Compilat (d.h. die fertige Anwendung) interessiert.

Die Quellen [Bet04c], [PLP] und [Bet02] enthalten weiteres Zahlenmaterial und ökonomisch relevante Aussagen zum MDSD-Entwicklungsprozess.

21. Organisatorische Aspekte

Mit Jorn Bettin

Beim Einsatz modellgetriebener Softwareentwicklung ist es erforderlich, zwischen der Entwicklung von konkreten Anwendungen und der Entwicklung der Domänenarchitektur zu unterscheiden. Die Gründe hierfür sind praktischer Natur: Erstens müssen konkrete Projekte zu festen Terminen fertiggestellt sein, und können nicht auf eventuelle technische Schwierigkeiten bei der Domänenarchitektur-Entwicklung Rücksicht nehmen. Zweitens ist die Qualität der Domänenarchitektur ganz entscheidend für den Erfolg von MDSD, und es ist davon abzuraten, durch Projektdruck falsche Kompromisse im Design der Domänenarchitektur zu erzwingen. Drittens unterscheiden sich die erforderlichen Kenntnisse zur Domänenarchitektur-Entwicklung von den erforderlichen Kenntnissen zur Anwendungsentwicklung. Daraus ergibt sich eine natürliche Abgrenzung. Ziel der organisatorischen Best Practices für MDSD ist es, eine Umgebung zu schaffen, welche eine sinnvolle Aufgabenverteilung unterstüzt.

MDSD zeichnet sich durch Skalierbarkeit aus und ist auch gut für den Einsatz in größeren, verteilten Projektumgebungen geeignet. Verteilte Softwareentwicklung ist nicht nur ein Thema in sehr großen Projekten, sondern zunehmend auch im Kontext von Offshoring.

21.1. Rollenverteilung

In diesem Abschnitt gehen wir auf MDSD-spezifische Rollenverteilungen ein. Wir orientieren uns dabei an dem in Kapitel <Ref Proz> skizzierten Entwicklungsprozess.

21.1.1. Domänenarchitektur-Entwicklung

Für erfolgreiche Domänenarchitektur-Entwicklung ist eine recht große Wissens- und Erfahrungsbandbreite erforderlich, welche sich üblicherweise auf eine Gruppe von Personen verteilt. Einige der benötigten Rollen unterscheiden sich nicht von denen in traditionellen Softwareentwicklungsprojekten, während andere Rollen im Rahmen von MDSD eine besondere Bedeutung annehmen. Auf diese Rollen werden wir in erster Linie eingehen. Dabei handelt es sich nicht etwa um eine Art „Stellenbeschreibung", sondern vielmehr um die Repräsentation der unterschiedlichen Tätigkeiten, die in einem konkreten Projekt abzudecken sind. Eine Person kann natürlich mehrere Rollen im Projekt übernehmen und umgekehrt kann eine Rolle auch von mehreren Personen ausgefüllt werden.

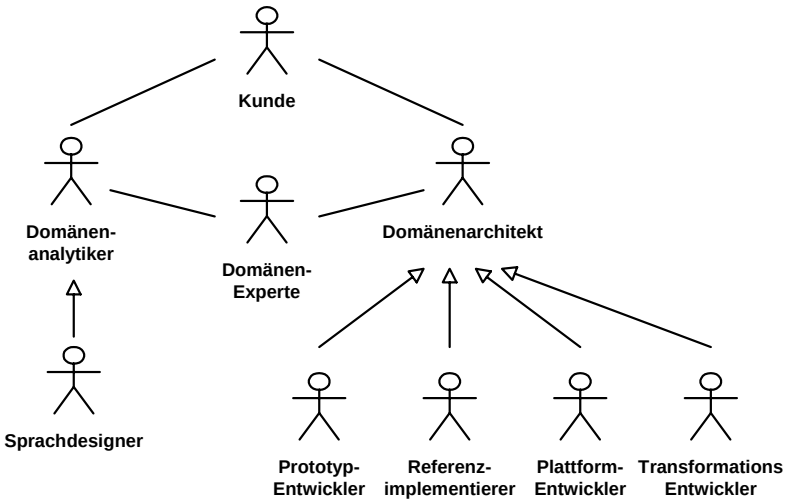


Abb. 171 *MDSD-spezifische Rollen bei der Entwicklung von Domänenarchitekturen*

Domänenexperten

Um eine gute Domänenarchitektur zu entwickeln, wird Expertenwissen über die MDSD-Domäne (vgl. Abschnitt <Ref Begr::Mod>) benötigt, d.h. denjenigen Bereich, der modellgetrieben unterstützt werden soll. Als Domänenexperten gelten Personen, welche in der Vergangenheit bereits mindestens zwei Anwendungen in der Domäne realisiert haben, beziehungsweise potenzielle Anwender, die Experten in Teilen der Domäne sind. Zum Beispiel können für die Entwicklung von Versicherungsanwendungen Juristen mit Spezialisierung auf Schadensabwicklung, Versicherungsmathematiker,

Direktmarketingspezialisten usw. als Domänenexperten herangezogen werden. Achtung: Bei architekturzentrierter MDSD sind die Domänenexperten Softwarearchitekten mit Erfahrung in den Technologien der Zielarchitektur (siehe Kapitel <Ref Archi>).

Expertenwissen ist zwar auch für die traditionelle Anwendungsentwicklung relevant, aber für die Entwicklung einer Domänenarchitektur essenziell. Dies ist ein Grund, warum die Domänenarchitektur-Entwicklung zumindest während des Bootstrappings für Offshoring eher ungeeignet ist.

Domänenanalytiker

Im Kapitel über Product Line Engineering (<Ref PLE>) sind wir bereits näher auf die Facetten der Domänenanalyse eingegangen. Die Ermittlung von Gemeinsamkeiten und Variationen innerhalb einer Domäne lässt sich am besten durch eine Reihe von Workshops mit Domänenexperten, ggf. Produktmanagern, Kunden, Anforderungsanalytikern und Domänenarchitekten iterativ erarbeiten.

Sprachdesigner

Ein spezieller Domänenanalytiker ist der Sprachdesigner. Er ist verantwortlich für die Entwicklung von Metamodellen auf der einen Seite sowie UML-Profilen oder anderen konkreten Syntaxformen für die Modellierung auf der anderen Seite (siehe Abschnitt <Ref Begr::Mod>). Mit anderen Worten: Er definiert domänenspezifische Sprachen (DSLs). Zu diesem Zweck arbeitet der Sprachdesigner in der Domänenanalyse eng mit Domänenexperten und Domänenanalytikern zusammen. Auf der Implementierungsseite ist der Sprachdesigner das entscheidende Bindeglied zu den Domänenarchitekten, zumal er auch die zu den Referenzimplementierungen passenden Referenzmodelle erstellt (vgl. <Ref Proz::ArchiStrg>).

Domänenarchitekt

Domänenarchitekten sind für die Realisierung von Domänenarchitekturen (siehe Teil 2 des Buches und Kapitel <Ref Proz>) verantwortlich. Dazu gehört insbesondere die Definition der Zielarchitektur (siehe Kapitel <Ref Archi>) und die Erarbeitung der Struktur von Subsystemen innerhalb der Domänenarchitektur. Letzteres geschieht in Kooperation mit dem Sprachdesigner. Entsprechend der Bestandteile einer Domänenarchitektur lassen sich spezifische Ausprägungen des Domänenarchitekten benennen:

Prototypentwickler

Prototypentwickler erstellen Technologieprototypen und sind daher meist Technologiespezialisten oder arbeiten eng mit solchen zusammen. Im Gegensatz zur traditionellen Entwicklung ermöglicht MDSD einen sehr ökonomischen Einsatz von Technologie-Expertenwissen – es kann mittels des generativen Ansatzes in die Anwendungsentwicklung transportiert werden, wodurch sich Coaching-Aufwände dort deutlich reduzieren lassen.

Referenzimplementierer

Referenzimplementierer bilden auf Basis der Prototypen die Konstrukte der DSL auf die Zielarchitektur ab. Dazu verwenden sie die Referenzmodelle aus der Domänenanalyse (s.o.) als Spezifikation.

Prototypen haben einen eher experimentellen Charakter, Referenzimplementierungen dagegen einen sehr formalen, da aus ihnen die Transformationen abgeleitet werden müssen. Oftmals finden sich innerhalb eines Teams Personen mit entsprechenden unterschiedlichen Neigungen, die man gemäß dieser beiden Rollen optimal einsetzen sollte.

Plattformentwickler

Mit der Bedeutung domänenspezifischer MDSD-Plattformen haben wir uns unter anderem bereits ausführlich in <Ref Begr::Plat> und <Ref Archi::Plat> auseinander gesetzt. Für die Rolle des Plattformentwicklers sind insbesondere Personen geeignet, die Erfahrung in der Framework-Entwicklung besitzen. Plattformentwickler müssen eng mit den Referenzimplementierern zusammenarbeiten, da die MDSD-Plattform ja letztlich eine abgeschlossene Partition der Referenzimplementierung bildet. Anders ausgedrückt: Die Referenzimplementierung stützt sich auf die domänenspezifische MDSD-Plattform und zeigt exemplarisch deren Verwendung.

Transformationsentwickler

Transformationsentwickler müssen die Generierungs- respektive Transformationswerkzeuge beherrschen. Sie leiten aus der Referenzimplementierung z.B. Code-Templates zur Erzeugung von Quellcode ab oder definieren Modell-zu-Modell-Transformationen (vgl. Kapitel <Ref Trans>). Template-Entwicklung erfolgt natürlich ebenfalls Hand in Hand mit der Entwicklung der domänenspezifischen Plattform. Ziel ist es, Transformations- und Plattformentwicklung so aufeinander

abzustimmen, dass weder die interne Komplexität der Plattform noch die Komplexität der Transformationen außer Kontrolle gerät.

Koordinatoren

Die Teams der Domänenanalytiker und Domänenarchitekten benötigen jeweils einen Ergebnis-verantwortlichen Koordinator. Je nach Fähigkeiten und Teamgröße können natürlich auch diese beiden Rollen in Personalunion ausgefüllt werden. Die Rolle des Domänenanalyse-Koordinators ist zudem gut mit der des Sprachdesigners vereinbar.

Weitere Rollen

Neben den bereits erwähnten Rollen sind die folgenden Rollen für die Domänenarchitektur-Entwicklung relevant:

- Kunden: Sie sollten so weit wie möglich in die Entwicklung von Benutzeroberflächen und Anwendungsprototypen mit einbezogen werden. Damit praktische Benutzbarkeitskriterien nicht unter den Tisch fallen, sollten auch potenzielle Endanwender zur Verfügung stehen. Kunden können auch Domänenexperten im obigen Sinne sein.
- Produktmanager: Bei der Entwicklung von Produkten und Produktlinien ist es wichtig, nicht nur die Wünsche von einzelnen Kunden zu berücksichtigen, sondern auch die strategische Richtung des Produkts oder der Produktlinie. Der Produktmanager sollte sowohl einen guten Überblick über den Produkteinsatz bei bestehenden Kunden besitzen als auch die Anforderungen von weiteren potenziellen Kunden im Zielmarkt kennen.
- Projektmanager: Wie für traditionelle Softwareentwicklung wird ein Projektmanager benötigt. Im MDSD-Kontext sind Kenntnisse über iterative Softwareentwicklung besonders wichtig.
- Anforderungsanalytiker: Wie in der traditionellen Softwareentwicklung werden Analytiker zur Analyse und Beschreibung von Kundenanforderungen in Form von UseCases und ähnlichen Artefakten eingesetzt.
- Testingenieur: Testingenieure sind verantwortlich für die Entwicklung von Teststrategien unter Berücksichtigung des besonderen Potenzials von MDSD in diesem Kontext (vgl. <Ref Test>).

21.1.2. Anwendungsentwicklung

Die Rollen in der Anwendungsentwicklung sind identisch mit denen in der traditionellen Softwareentwicklung, wenn man die folgenden Aspekte ausklammert, da sie beim Einsatz von MDSD in den Domänenarchitektur-Entwicklungsstrang ausgelagert sind:

- Die Entwicklung von Frameworks (d.h. die Entwicklung der MDSD-Plattform).
- Die Entwicklung von Anwendungsprototypen und Referenzimplementierungen sowie Experimente mit neuen Implementierungstechnologien.
- Die Analyse von Anforderungen, welche für den gesamten Zielmarkt relevant sind.

Wie leicht erkenntlich ist, sind gerade die kritischen Aktivitäten, welche oft zu Terminverzögerungen oder Qualitätseinbußen führen, in die Domänenarchitektur-Entwicklung ausgelagert und von konkretem Kundentermindruck besser entkoppelt (vgl. Abschnitt <Ref Proz::IterX2>).

Zusätzlich zu den Kenntnissen über traditionelle Softwareentwicklung benötigen die Anwendungsentwickler ein Grundverständnis über MDSD im Allgemeinen und die DSL der zu verwendenden Domänenarchitektur im Besonderen.

21.2. Teamstruktur

Wir wollen keine allgemeingültige Teamstruktur vorschieben, schließlich entwickeln unterschiedliche Teams ihre eigene Arbeitsweise, welche auf die Kenntnisse der Individuen zugeschnitten ist. Es gibt oftmals „böses Blut“, wenn man versucht, Personen in fest vorgegebene (neue) Rollen zu zwängen. Die Aufgabenverteilung auf Personen sollte sich möglichst an den vorhandenen Fähigkeiten und den individuellen Interessen orientieren.

Wir legen vielmehr Wert auf eine passende Makrostruktur der Teams (zum Beispiel die bereits häufiger erwähnte Trennung zwischen Domänenarchitektur-Entwicklung und Anwendungsentwicklung).

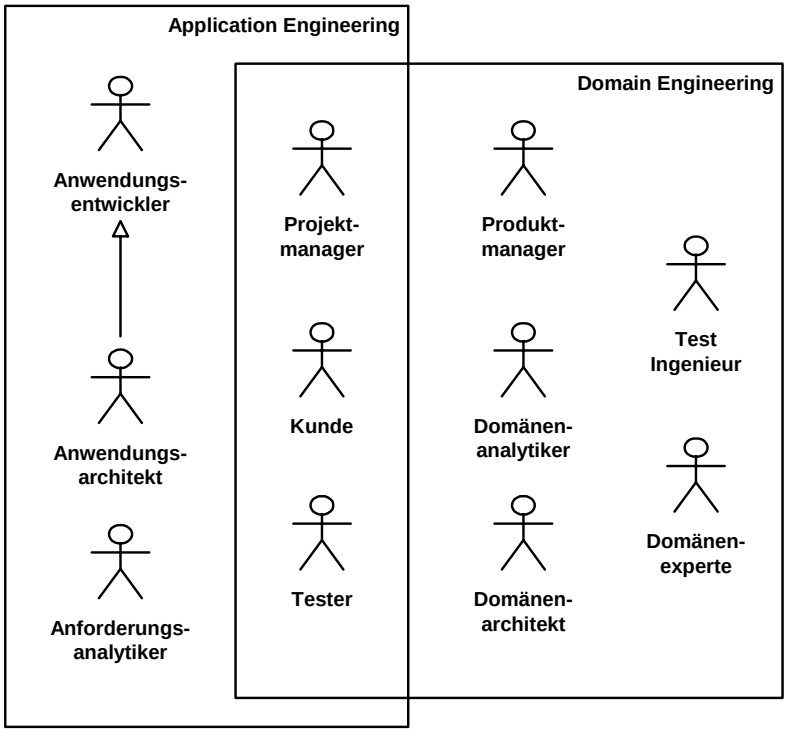


Abb. 172 *Makroskopische Teamstruktur*

Abb. 172 fasst zusammen, welche Rollen für die Anwendungsentwicklung und welche Rollen für die Domänenarchitektur-Entwicklung relevant sind.

Wie zu erkennen ist, gibt es Rollen, welche in beiden Entwicklungssträngen vorhanden sind. Bei räumlicher Trennung von Domänenarchitektur-Entwicklung und Anwendungsentwicklung müssen diese Rollen in beiden Standorten vertreten sein. Die in der Domänenarchitektur-Entwicklung tätigen Spezialisten müssen ihr Wissen in Workshops und Trainings an die Anwendungsentwickler weitergeben.

Für die Skalierbarkeit der Domänenarchitektur-Entwicklung ist weiterhin die Orientierung der Teamstruktur an der Komponentenstruktur der Domänenarchitektur hilfreich. Eine entsprechende Aufteilung der Verantwortlichkeiten ermöglicht weitgehend ein Top-Down-Vorgehen beim Grobentwurf der Domänenarchitektur. Zunächst werden die Schnittstellen definiert, so dass die Arbeit an verschiedenen Subsystemen und Frameworks anschließend besser parallelisierbar ist.

21.2.1. Ausgestaltung der Rollen und Personalbedarf

Die konkrete personelle Besetzung der Rollen hängt im Wesentlichen von den Parametern *Projektgröße*, *Verteilung von Fähigkeiten im Team* und *Ausrichtung der MDSD-Domäne* ab. Bei architekturzentrierter MDSD können z.B. die Rollen *Domänenanalytiker* und *Domänenarchitekt* sehr gut von den gleichen Personen ausgefüllt werden, weil in diesem Fall die MDSD-Domäne ohnehin dem Fachwissen der Domänenarchitekten entspricht. In einem sehr kleinen, architekturzentrierten MDSD-Projekt können alle Rollen aus Abb. 171 ohne weiteres in Personalunion ausgefüllt werden – ggf. sogar zusammen mit der eigentlichen Anwendungsentwicklung (Ein-Personen-Projekt). Im Rahmen einer größeren, fachlichen Produktfamilien-Entwicklung sollten insbesondere Domänenanalytiker und Domänenarchitekten durch unterschiedliche Personen besetzt sein.

Besonders wichtig ist die Skalierung der Rollenbesetzung für die Domänenarchitektur-Entwicklung im Verhältnis zur Anwendungsentwicklung: Ein MDSD-Projekt mit beispielsweise über hundert Anwendungsentwicklern, aber nur einem Transformationsentwickler, ist zum Scheitern verurteilt, sofern die Domänenarchitektur nicht bereits ausgereift ist. Berücksichtigen Sie auch, dass die Domänenarchitekten die Anwendungsentwickler gerade zu Beginn coachen müssen und Feedback zur Domänenarchitektur entsteht.

Eine einfache Faustformel können wir naturgemäß leider nicht angeben, zumal die Bootstrapping-Phase einer Domänenarchitektur typischerweise deutlich personalintensiver ist als die iterative Erweiterung. Der Zuwachs bei den Anwendungsentwicklern sollte gerade umgekehrt sein – und erst nach der Bootstrapping-Phase „in die Breite“ gehen. Abb. 173 zeigt qualitativ und idealisiert unsere Erfahrungswerte hinsichtlich der rollenbezogenen Skalierung über die Zeit:

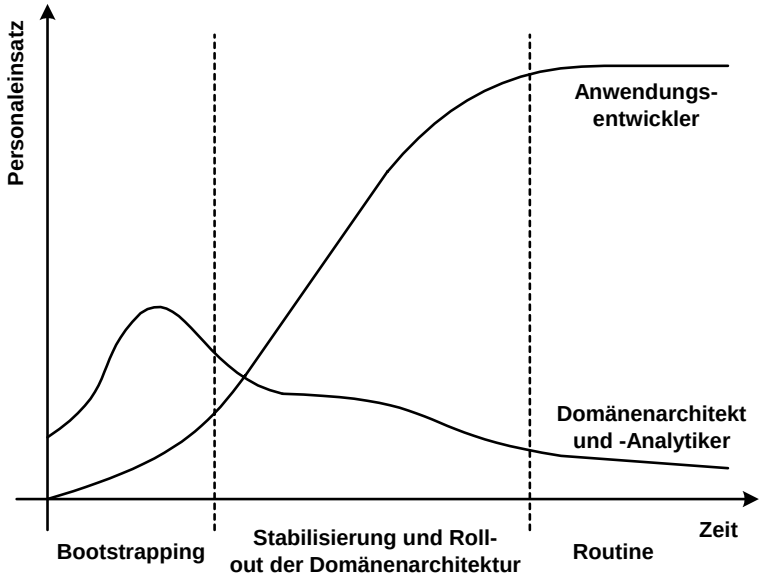


Abb. 173*Zeitliche Entwicklung der MDSD-Rollen (idealisiert)*

21.2.2. Querschnitts-Teams

Beim Thema „effektive Kommunikation zwischen Teams“ gelten die gleichen Regeln wie bei traditioneller Softwareentwicklung im Großen: Sobald mehrere Entwicklerteams parallel arbeiten, sollte durch Einrichtung „horizontaler“ Gruppen entlang der im Entwicklungsprozess definierten Rollen die Einhaltung von Standards und die kontinuierliche Verbesserung unterstützt werden. Als Ansatzpunkt schlagen wir folgende horizontale Gruppen vor (dabei ist nur die Architekturgruppe obligatorisch, die anderen Gruppen werden erst für Teams ab ungefähr 15 Personen relevant):

- Die *Architekturgruppe* ist verantwortlich für die Vorgaben und Prioritäten der Domänenarchitektur-Entwicklung. Diese Gruppe besteht typischerweise aus Softwarearchitekten der Anwendungs- oder Produktentwicklungsteams und mindestens den Koordinatoren der Domänenarchitektur-Entwicklung.
- Die *Qualitätssicherungsgruppe* ist verantwortlich für die Definition und Einhaltung von Qualitätsstandards.
- Die *Produktmanagementgruppe* ist verantwortlich für die Qualität von UseCases und anderen Anforderungsdefinitionen sowie für

die Prioritäten der Produktentwicklung. Diese Gruppe sollte (pro Produktfamilie) Produktmanager, Domänenexperten und die Anforderungsanalytiker einschließen.

- Die *Projektmanagementgruppe* ist verantwortlich für die Einhaltung von Terminen und Budgets, Controlling und die Definition von erforderlichen Projektmanagementstandards.

21.2.3. Aufgaben der Architekturgruppe

Ein querschnittlich agierendes Architekturteam ist insbesondere wichtig, wenn auf Basis einer Domänenarchitektur *mehrere* Anwendungen entwickelt werden. Es liegt uns sehr am Herzen zu betonen, dass sich die Architekturgruppe als Dienstleister und „Hüter der Software-Produktionsstraße“ begreifen muss, um erfolgreich zu sein. Sie darf keine „Elfenbeinturm“-Architektur betreiben. Die Aufgabe ist es primär, die Evolution der Domänenarchitektur(en) gemäß der Anforderungen aus Projekten oder Produktentwicklungen zu steuern. Deswegen werden die Prioritäten nicht von den Domänenarchitekten selbst gesetzt, sondern von der Architekturgruppe. Die Domänenarchitekten müssen zudem sehen, wie sich ihre Architektur in der Praxis bewährt. Dazu kann es sehr hilfreich sein, wenn sie zeitweise auch als Anwendungsentwickler oder Projektarchitekten arbeiten. Die sekundäre Aufgabe der Architekturgruppe ist es, die Qualitätsattribute der Domänenarchitektur(en) zu bewahren (also auf Performance, Skalierbarkeit, Wiederverwendbarkeit und Wartbarkeit zu achten). Die Architekturgruppe muss diese beiden Sichtweisen zu einer geeigneten Synthese führen.

Wir empfehlen *nicht*, der Architekturgruppe die operative (Weiter-)Entwicklung von Domänenarchitekturen aufzubürden. Dazu wird sie in der Regel nicht genügend Freiraum besitzen. Zu diesem Zweck ist die Organisationsform eines (Teil-)Projektes mit eigener Planung, Iterationen und Releasezyklen – ähnlich wie bei einer Framework-Entwicklung – wesentlich besser geeignet. Die Architekturgruppe sollte lediglich die Evolution der Domänenarchitekturen im Sinne einer querschnittlichen Interessenvertretung steuern.

Diese Evolution umfasst typischerweise zwei Dimensionen – Variantenbildung und Verbesserung:

Varianten entstehen unter anderem beim Austausch einer Plattform-Komponente wie z.B. dem Wechsel der Datenzugriffsschicht von Technologie X auf Y. Wenn die Domänenarchitektur gut modularisiert ist, bilden diejenigen Transformationen, welche für Generierung der

Datenzugriffsschicht verantwortlich sind, ein lose gekoppeltes Modul – oft auch *Cartridge* genannt – innerhalb der Domänenarchitektur. Sie können die X-Cartridge nun also gegen eine (ggf. zu erstellende) Y-Cartridge austauschen, ohne dass sich die Schnittstellen zur Businesslogik oder die Modellierungssprache (DSL) ändern. Es ensteht dabei also ein Derivat der Domänenarchitektur. Verfolgt man diesen Gedanken weiter, so ergibt sich als Zielvorstellung ein Baukasten von Cartridges, die maximal kombinierbar und minimal redundant sind. Dieses Ziel wird man allerdings sicher nicht „vom Reißbrett“ aus erreichen, sondern nur durch Evolution.

Verbesserungen entstehen aus den Feedbackschleifen zu den Projekten. Dabei handelt es sich z.B. um die Bereinigung von Defiziten wie mangelndem Abstraktionsgrad der DSL oder weitergehende generative Unterstützung für Anwendungsbereiche, die bisher ausgespart wurden, bzw. deren Automationspotential bislang nicht erkannt wurde. Durch Verbesserung entstehen meist neue Versionen bestehender Cartridges. Ein entsprechend klares Versions- bzw. Releasemanagement sollte dann den Projekten ermöglichen, selbst zu entscheiden, ob sie auf eine neue Version umstellen oder bei der alten bleiben wollen (vgl. <Ref Vers> und <Proz::IterX2>).

Beide Dimensionen der Evolution (Variantenbildung und Verbesserung) können sich zwar im Prinzip negativ auf die konkrete Wiederverwendung von Domänenarchitekturen auswirken (eine Variante muss erst erstellt werden, eine Verbesserung ist vielleicht nicht abwärtskompatibel), es wäre jedoch kontraproduktiv, die Evolution aus diesem Grund verhindern zu wollen, denn sie stellt die eigentliche Innovationskraft und einen Wertzuwachs dar. Wichtig ist allerdings wie gesagt die rechtzeitige Modularisierung auf dieser Ebene, damit ein Baukasten für Domänenarchitekturen entstehen kann. Eine modular aufgebaute Produktionsstraße für Softwaresysteme ist eine Investition, die geschützt werden sollte. Diese projektübergreifenden Werte sollte die Architekturgruppe vertreten und die Evolution von Domänenarchitekturen in ihrem Sinne steuern.

Die Betrachtungen sind natürlich allesamt perspektivisch und sicher nicht für das erste MDSD-Pilotprojekt relevant. Sie betreffen je nach Unternehmensgröße einen Zeithorizont von vielleicht zwei bis fünf Jahren. Sie sind aber keineswegs hypothetisch oder rein theoretisch, sondern basieren allesamt auf Erfahrungen, die wir bereits machen durften.

21.3. Software-Produktentwicklungsmodelle

In den folgenden Abschnitten untersuchen wir verschiedene organisatorische Softwareentwicklungsmodelle aus der Perspektive eines Softwareherstellers, das heißt, aus der Perspektive einer Firma, welche Softwareprodukte herstellt und vertreibt, und wir analysieren den Anwendungsbereich der verschiedenen Modelle mit speziellem Augenmerk auf MDSD. Dieselben Modelle können bei der Entwicklung von Anwendungen zum Einsatz kommen, der Einfachheit halber verwenden wir aber die produktorientierte Terminologie.

21.3.1. Terminologie

An der Entwicklung eines Softwareprodukts können mehrere Organisationen beteiligt sein, mit entsprechenden „Kunde–Lieferant"-Beziehungen zwischen den beteiligten Parteien – so genannten *Zulieferketten* (Supply Chains). Der Begriff „Kunde" wird üblicherweise auch benutzt, um den Zielmarkt des Produkts zu beschreiben. Um potenzielle Missverständnisse auszuschließen, verwenden wir die folgende Terminologie:

- *Ihr Unternehmen* ist der Softwarehersteller, welcher Produkte an einen bestimmten Markt verkauft.
- *Softwaredienstleister* sind Firmen, welche Ihr Unternehmen im Rahmen der Produktentwicklung einsetzt. Wenn erforderlich, verwenden wir den Begriff Softwaredienstleister mit entsprechendem Präfix wie *On-shore-*, *Off-shore-*, *externer* etc.
- *Kunden* sind Firmen im Zielmarkt Ihrer Produkte.

Im Bezug auf „Vermögenswerte" reicht es nicht mehr, nur von Softwarekomponenten zu sprechen.

- *Softwareassets* sind alle wertschöpfungsrelevanten Artefakte, die bei der Softwareentwicklung eine Rolle spielen: Modelle, Komponenten, Generatoren, Sprachen, Techniken und Prozessbausteine.

Ihre Organisation kann sich auf die Entwicklung eines Produkts beschränken oder kann sich auf die Entwicklung einer Produktfamilie oder Produktlinie erstrecken. Ein *Einzelprodukt* besteht aus einer Codebasis, welche Kunden in einer Standardkonfiguration zur

Verfügung gestellt wird, die gegebenenfalls zum Installationszeitpunkt oder später vor Ort verändert werden kann.

- Eine *Produktfamilie* besteht aus einer Menge von Produkten, die auf der gleichen Grundlage erzeugt wurden – d.h. denen mindestens ein Softwareasset gemeinsam ist (in aller Regel sind es mehrere). Im Falle von MDSD ist diese Gemeinsamkeit typischerweise die Domänenarchitektur, so dass aus der Produktfamilie eine Software-Systemfamilie wird (vgl. <Ref Begr::SSF>). Die Elemente einer Produktfamilie bezeichnen wir als *Familienmitglieder*.
- Eine *Produktlinie* besteht aus eine Menge von Produkten mit einem gemeinsamen Zielmarkt – d.h. die Zusammenstellung ist kundengruppenspezifisch. Eine ökonomisch wartbare Produktlinie besteht idealerweise aus einer Produktfamilie. Allerdings sind Produktstrukturen in der Praxis teilweise weit von diesem Ideal entfernt, da Softwareunternehmen oft durch Akquisition von anderen Herstellern wachsen. Die Elemente einer Produktlinie bezeichen wir als *Produktlinienmitglieder*.
- *Produktrationalisierung* ist der Prozess von Konsolidierung und Refactoring einer Produktlinie in eine Produktfamilie.
- Wir reden von *Software-Massenfertigung*, wenn Produktfamilienmitglieder mittels eines weitgehend automatisierten Prozesses aus Kundenspezifikationen abgeleitet werden können und wenn dabei die Effizienz an Massenproduktion grenzt – dies ist das Ziel von MDSD im Kontext fachlicher, domänenspezifischer Plattformen.

21.3.2. In-House-Entwicklung

Vorteilhaft bei diesem Modell ist die Konzentration von Domänenwissen und technischem Know-how, worauf sich eine gute Marktposition aufbauen lässt. Auch wird es kritischen Time-to-Market-Anforderungen gerecht.

Ohne Zweifel ist In-House-Entwicklung ideal geeignet, um agile Softwareentwicklungsmethoden einzusetzen und um Kunden vor Ort in den Produktentwicklungsprozess mit einzubeziehen.

In den letzten Jahren hat allerdings der wirtschaftliche Druck und die Konkurrenz großer Hersteller dazu geführt, dass dieses Modell immer weniger attraktiv geworden ist.

Einsatz von MDSD

Bei diesem Entwicklungsmodell kann Ihre Organisation selbst MDSD einsetzen und die damit verbundenen Kostenvorteile voll ausschöpfen.

Risiken

In erster Linie sind hier die hohen Personalkosten zu nennen.

Anwendbarkeit

Dieser Ansatz ist in erster Linie in Situationen anwendbar, wo der Zielmarkt relativ zu Ihrem Unternehmen lokal ist, und wo die Nähe zum Markt als signifikanter Vorteil in Form von Time-to-Market ausgenutzt werden kann.

Der Ansatz ist nicht gut geeignet im Falle eines globalen Zielmarktes, wo die Kundennähe an keinem einzelnen Standort gegeben ist.

21.3.3.Klassisches Outsourcing

Dieses Modell ist seit langem etabliert und ist unter anderem attraktiv für Unternehmen wie zum Beispiel Telekommunikationsdienstleister, welche komplexe Software benötigen, um ihre Dienste zu unterstützen. In solchen Fällen kann Softwareentwicklung an einen externen Dienstleister ausgelagert werden.

Es ist interessant zu betrachten, wie dieses, bei der Erstellung von Individualsoftware weit verbreitete Entwicklungsmodell bei der Produktentwicklung aussieht. Für den Softwaredienstleister besteht hier ein Anreiz, ein Produkt (beziehungsweise eine Produktfamilie) von genereller Anwendbarkeit zu entwickeln, welche durch mehrfachen Verkauf eine weitere Einnahmequelle darstellt. Der externe Softwaredienstleister kann entweder bereits bei der Erstellung eines Prototypen einbezogen werden oder kann zu einem späteren Zeitpunkt zwecks Weiterentwicklung hinzugezogen werden.

Typische Dienstleistungsanbieter in diesem Umfeld sind die großen multinationalen Firmen wie IBM, EDS, CSC und andere. Die Produkt-(familien-)entwicklung wird von diesen Dienstleistern zum Beispiel unter dem Stichwort „Strategisches Outsourcing“ vermarktet.

Kosten und Vorteile

Ihr Unternehmen profitiert in diesem Fall von vertraglich festgelegten Kostenlimits für die Entwicklung von bestimmten Softwareprodukten, und es reduziert die Personalkosten interner Softwareentwicklung in

Ihrem Unternehmen. Da die Softwarediensteister in diesem Umfeld meist große, finanzstarke Firmen sind, benutzen Unternehmen dieses Modell gern als Versicherungspolice.

Durch die örtliche Nähe zwischen Ihrem Unternehmen und dem Softwaredienstleister (teilweise im gleichen Gebäude), können bewährte Risikominimierungsstrategien wie *on-site customer* problemlos angewandt werden.

Einsatz von MDSD

Bei diesem Entwicklungsmodell kommt MDSD ggf. dem Outsourcing-Dienstleister zugute.

Risiken

Oft übernimmt der Softwaredienstleister Personal (typischerweise ganze Abteilungen) von Ihrem Unternehmen, um sich entsprechendes Fachwissen anzueignen. Dies bedeutet, dass ein großer Anteil an Fachwissen Ihr Unternehmen permanent verlässt. Um diesem Risiko entgegenzuwirken, bezieht sich der Outsourcing-Vertrag üblicherweise auf Zeiträume von fünf Jahren oder länger.

Anwendbarkeit

Wenn Sie selber die entwickelte Software einsetzen und gleichzeitig die Software als Produkt verkaufen wollen, kommt dieses Modell nur in Betracht, wenn die Software nicht den Kern Ihres Unternehmens ausmacht. Schließlich müssen Sie damit rechnen, dass das entwickelte Produkt von Ihrer Konkurrenz gekauft und angewandt wird.

21.3.4.Offshoring

In einer Welt, in der hohe Softwareentwicklungskosten zum wachstumshemmenden Faktor werden, sind Offshoring-Ansätze zur Auslagerung von Softwareentwicklungsaktivitäten in kostengünstige Standorte zunehmend attraktiv, und verteilte Softwareentwicklung wird zur Norm. Wer in Softwareprojekten, die sich nicht nur über Zeitzonen, sondern auch über kulturelle und sprachliche Grenzen erstrecken, gearbeitet hat, weiß, dass Offshoring nicht ohne Risiko ist. Obwohl Einsparungen bei Softwareentwicklungskosten erreichbar sind, können Erfüllung von Time-to-Market-Zielen und die Entwicklung eines den Anforderungen des Marktes entsprechenden Produkts durchaus nicht als selbstverständlich vorausgesetzt werden. Um die größten Risiken zu

minimieren, bieten einige Offshore-Softwaredienstleister sowohl Offshore-Teams als auch unterstützende Ressourcen vor Ort (Onshore) an. Dies kann bei der Überwindung von grundlegenden Kommunikationsproblemen helfen, aber die Effizienz des gesamten Produktentwicklungsprozesses wird nach wie vor durch die Qualität der benutzten Softwareentwicklungsmethodik bestimmt.

Die Softwareindustrie hat auf schmerzhaftem Wege gelernt, dass *big design up-front* nicht funktioniert, insbesondere nicht, wenn hohe Qualität gefordert ist, und wenn feste Meilensteine vorgegeben sind. Ein räumlich zusammenhängendes Team, welches Endbenutzer einschließt, kann die Vorteile der Kundenpräsenz vor Ort voll ausnutzen. Welche Risikoreduktionsstrategien werden also benötigt, um bei verteilter Softwareentwicklung die mangelnde Kundennähe zu kompensieren? Ihr Offshoring-Partner hat womöglich eine Reihe von Onshore-Ressourcen vor Ort zur Verfügung gestellt, als Bindeglied zwischen Ihrem Produktmanagement, Domänenexperten und dem Offshore-Entwicklungszentrum. Um Qualität zu gewährleisten, müssen zumindest einige Onshore-Teammitglieder zwischen den Onshore- und Offshore-Standorten pendeln. Welche Auswirkung hat dies auf die Kosten und insbesondere auf Time-to-Market?

Wir werden diese Fragen vor dem MDSD-Hintergrund betrachten und aufzeigen, dass MDSD eine wichtige Rolle bei der Risikominimierung verteilter Softwareentwicklung spielt.

21.3.5.Radikales Offshoring

In diesem Ansatz wird die Softwareentwicklung so weit wie möglich in kostengünstige Standorte verlagert. Idealerweise besitzt Ihr Unternehmen in diesem Szenario erhebliche Anteile des Offshore-Softwaredienstleisters.

Bei der Implementierung dieses Modells handelt es sich entweder um den Aufbau eines Offshore-Softwareunternehmens oder um die Suche nach einer geeigneten Firma, die sich evtl. sogar zum Kauf anbietet. Auch hier gilt „start small“: Zunächst sollte man ein kleines, unkritisches Projekt mit einem Offshore-Dienstleister durchführen, anschließend das Projekt bewerten und letztlich ggf. eine Struktur für zukünftige Projekte finden.

Um dieses Modell erfolgreich einzusetzen, muss eine große Menge an Fachwissen an das Offshore-Softwareunternehmen weitervermittelt werden.

Kosten und Vorteile

Dieser Ansatz erfordert extensiven Personalaustausch und entsprechend hohes Reiseaufkommen. Ihre Domänenexperten und Produktmanager müssen das Offshore-Team zusammenstellen und entsprechendes Wissen vermitteln. Dieses Modell versucht, die Vorteile von kostengünstigen Standorten zu maximieren. Allerdings müssen die Vorteile gegen hohe Reisekosten und verzögertes Time-to-Market abgewogen werden.

Einsatz von MDSD

Bei diesem Entwicklungsmodell kommt MDSD allenfalls Ihrem Offshore-Partner zugute.

Risiken

Das Hauptrisiko von radikalem Offshoring besteht in der Notwendigkeit des signifikanten Wissenstransfers. Dazu können erschwerend Sprachbarrieren (insbesondere im Bereich branchenspezifischer Terminologie) hinzukommen. Im Gegensatz zum normalen Outsourcing besteht nicht die Option, ganze Teams mit entsprechendem Fachwissen permanent in das Offshore-Softwareunternehmen zu verlagern. Das Offshore-Team muss von Grund auf aufgebaut werden, oder es muss aus einem unbekannten Pool von Ressourcen des Offshore-Softwareunternehmens zusammengestellt werden.

Sobald das notwendige Fachwissen vermittelt worden ist, ist Ihr Unternehmen sehr stark von dem Offshore-Softwareunternehmen abhängig. Deshalb kann es sinnvoll sein, sich entsprechende Eigentumsrechte an dem Offshore-Softwareunternehmen zu sichern.

Je nachdem, wo sich der Zielmarkt befindet, kann Time-to-Market zum Problem werden. Des Weiteren kann die Implementierung von Risikominimierungsstrategien wie *on-site customer* [Bec00] problematisch sein.

Die Idee des radikalen Offshoring ist der Versuch, die einfache Struktur von klassischem Outsourcing zu kopieren. Dabei können die Schwierigkeiten, welche durch die räumliche Trennung zwischen Produktmanagement und Softwareentwicklung verursacht werden, schnell zum Fallstrick werden.

Das Modell ist insgesamt sehr riskant.

Anwendbarkeit

Dieses Modell ist anwendbar, wenn Time-to-Market unkritisch ist, und insbesondere, wenn Ihr Unternehmen bereits ein erfahrenes Offshore-Softwareunternehmen besitzt. Allerdings bedeutet der Mangel an technischem Fachwissen in der Mutterorganisation, welcher sich unter Umständen im Laufe der Zeit verschärft, dass das Reiseaufkommen zwischen Standorten über den gesamten Produktlebenszyklus permanent hoch bleibt.

21.3.6.Kontrolliertes Offshoring

Bei diesem Ansatz wird nur ein Teil des Entwicklungsprozesses in kostengünstige Standorte ausgelagert. Wir sprechen von „kontrolliertem“ Offshoring nicht etwa, weil andere Varianten von Offshoring unkontrollierbar sind, sondern weil beim kontrollierten Offshoring erhebliches Domänenwissen, technisches Fachwissen und die Kontrolle über geistiges Eigentum weitgehend in Ihrem Unternehmen bleibt. Dieses Modell öffnet die Möglichkeit, Offshore-Softwaredienstleister einzuspannen, ohne das Risiko einer monopolartigen Abhängigkeit einzugehen.

Kosten und Vorteile

Bei diesem Ansatz arbeitet Ihr Unternehmen eng mit Kunden zusammen, um Produkte zu spezifizieren, Produktprototypen und Referenzimplementierungen zu entwickeln und Softwarearchitekturen zu entwerfen. Das Modell erfordert technisch versierte Onshore-Ressourcen, welche ohnehin vorhanden sein sollten, wenn Sie bis jetzt In-House-Softwareentwicklung betrieben haben. Auf den ersten Blick mag es so aussehen, als ob sich dies in hohen Kosten niederschlägt, allerdings zeigt die Erfahrung, dass die theoretischen Kostenvorteile von Offshore-Prototypentwicklung von Kommunikationsineffizienz zunichte gemacht werden. Die räumliche Nähe von Produkmanagement, Kunden, und wichtigen Mitgliedern des Softwareentwicklungsteams bleibt erhalten, welches zur Reduzierung des Gesamtrisikos im Vergleich zum radikalen Offshoring führt. Mit entsprechenden Methoden und Werkzeugen vermindert dieses Modell eine kritische Abhängigkeit von einem bestimmten Offshore-Softwaredienstleister.

Einsatz von MDSD

Bei dieser Offshore-Variante kann MDSD seine Vorteile auch aus Sicht Ihres Unternehmens optimal ausspielen: Indem die Domänenarchitektur-Entwicklung – inklusive Domänenanalyse und Sprachdesign – in Ihrer Organisation verbleibt, bietet MDSD nicht nur formal definierte (und damit nachvollziehbare) Softwareanforderungen in Form von Modellen, sondern dem Offshore-Dienstleister auch ein vollständiges Architekturskelett für das Produkt. Hierdurch wird erstens die Einhaltung von vorgegebenen Architekturstandards maschinell garantiert, und zweitens ermöglichen die formalen Spezifikationen einen testgetriebenen Ansatz (vgl. Kapitel <Ref Proz> und <Ref Test>). Das hohe Maß an Kontrolle über die technische Produktqualität, welche mit MDSD erreicht werden kann, verringert die Offshoring-Risiken, und auch das Reiseaufkommen lässt sich reduzieren.

MDSD ist hinreichend und vermutlich sogar notwendig, um kontrolliertes Offshoring zu realisieren.

Risiken

Dieses Verfahren führt zu höheren Onshore-Kosten im Vergleich zu radikalem Offshoring – zumindest das Verhältnis von Onshore- zu Offshore-Kosten ist höher. Die Gesamtkosten können durch den Einsatz von MDSD aber sogar niedriger gehalten werden.

Sobald ein Unternehmen sich für eine Offshoring-Strategie entschieden hat, besteht das Risiko, die Vorteile von Offshoring zu überschätzen. Insbesondere wenn mittels MDSD ein hoher Automatisierungsgrad erreicht werden kann, sollte der verbleibende manuelle Entwicklungsaufwand neu abgeschätzt werden, um sicherzustellen, dass Offshoring auch tatsächlich die beste Option ist.

Anwendbarkeit

Dieses Verfahren eignet sich besonders, wenn sich Ihr Unternehmen in der gleichen geographischen Lage wie der Zielmarkt befindet. Die Personaleinsparungen beim Einsatz von MDSD wirken sich insbesondere positiv aus, wenn dadurch die Größe des Offshore-Teams verringert werden kann und sich dadurch der Projektmanagement-Overhead reduzieren lässt. Der MDSD-Ansatz ist sehr gut skalierbar und eignet sich zur Entwicklung von ganzen Produktfamilien und Produktlinien. Die Domänenarchitektur-Entwicklung sollte nicht ausgelagert werden – dies wäre jedenfalls der Übergang zu radikalem Offshoring mit entsprechend höherem Risiko.

21.3.7.Komponentenweise Entscheidung

Produkte oder Produktfamilien mit gutem Design bestehen aus relativ lose gekoppelten Komponenten. Die Erstellung eines neuen Produktes oder die Rationalisierung mehrerer bestehender Produkte (s. Abschnitt 21.3) bietet eine ausgezeichnete Gelegenheit, um eine gute Komponentenarchitektur [Bet04b] zu realisieren.

Wieso ist eine klare Komponentenarchitektur so wichtig? Erstens ermöglicht eine solche Architektur eine *divide-and-conquer*-Technik, um die Arbeit an großen Systemen in greifbare und parallelisierbare Teile aufzuteilen. Zweitens ermöglicht eine solche Architektur strategische „Bauen/Kaufen/Open-Source"-Entscheidungen auf Komponentenebene. Auf diese Art und Weise ergeben sich auch natürliche Migrationsstrategien hin zu einer Zielarchitektur im Rahmen der Produktrationalisierung.

Beim Entwurf einer Produktfamilie ist es wichtig, die Entscheidung über das Softwareentwicklungsmodell (In-House, klassisches Outsourcing, radikales Offshoring, kontrolliertes Offshoring) nicht etwa global, sondern komponentenweise zu treffen. Entsprechendes gilt dadurch für die jeweiligen Zulieferketten.

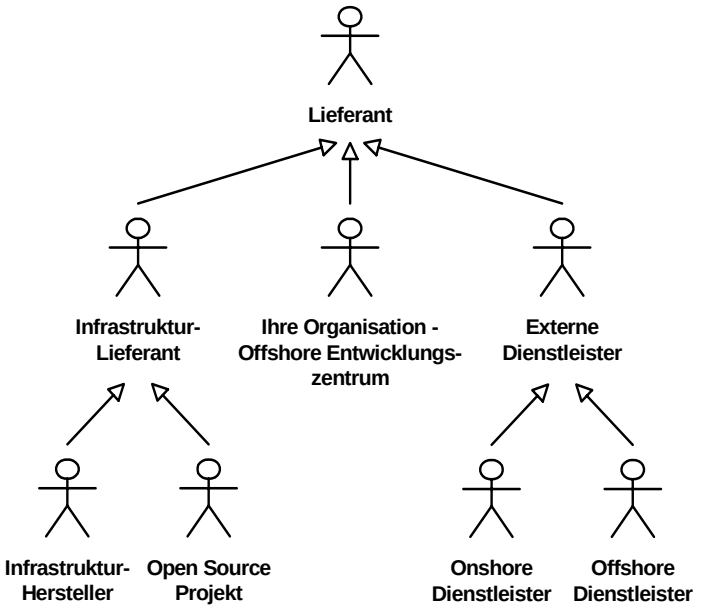


Abb. 174*Potenzielle Softwarelieferanten*

In der heutigen Realität sind die meisten Softwareprodukte strukturell nicht besonders „sauber" entworfen und bestehen aus eng gekoppelten Komponenten, die nur schwer zu entflechten sind. Die Folge ist, dass die

meisten Produkte eng an eine bestimmte Kombination von Implementierungstechnologien gebunden sind. Zudem können Änderungen in einer dieser Technologien die Ursache für erhebliche Wartungsaufwände darstellen. Daher ist auch für die Mehrzahl der heutigen Produkte die Idee von ersetzbaren Komponenten Fiktion. Der Anreiz, eine klare Komponentenarchitektur zu implementieren, ist erst gegeben, wenn ein Unternehmen das volle Potenzial des Konzepts einer Produktfamilie erkannt hat.

Die Quellen [Bos00], [Coc01] und [Bet04b] enthalten weiterführendes Material zum Thema Organisation.

22. Adaptionsstrategien für MDSD

Mit Jorn Bettin

Wir hoffen, nachvollziehbar gemacht zu haben, welches Potenzial in modellgetriebener Softwareentwicklung steckt und welche Investitionen bzw. organisatorischen Maßnahmen prinzipiell sinnvoll oder nötig sind. Sie werden sich an dieser Stelle hoffentlich fragen, wie und unter welchen Voraussetzungen Sie in Ihrem Projekt oder Unternehmen Schritte in diese Richtung unternehmen können, und wie der weitere „Fahrplan“ zur Adaption von MDSD im Unternehmen aussehen kann. Das vorliegende Kapitel soll zu diesen Fragen eine Hilfestellung geben.

22.1. Voraussetzungen

Modellgetriebene Softwareentwicklung im Sinne dieses Buches ist unabhängig von der konkreten Ausprägung sicherlich als sehr fortschrittlich in Sachen Software Engineering zu bezeichnen. Möglicherweise gibt es aber viel grundlegendere Veränderungen, die erst einmal bewerkstelligt sein wollen, bevor eine Optimierung in Form von MDSD Sinn macht:

- Ist ein iterativ inkrementelles Vorgehen für die Projektabwicklung etabliert – insbesondere im Bereich des Anforderungsmanagements? Besitzt das Projektteam diesbezügliche Erfahrungen? Die bei der Einführung von MDSD bevorstehende Trennung von Domänenarchitektur-Entwicklung und Anwendungsentwicklung erfordert die funktionierende Koordination von zwei Teilprojekten.
- Ist Architektur-Know-how auf operativer Ebene im Team vorhanden?

Sollten Sie eine dieser Fragen für sich mit nein beantworten, so empfehlen wir diese Dinge vor oder zumindest mit der Definition eines

MDSD-Pilotprojektes anzugehen – im Zweifelsfall mit externer Unterstützung. Dadurch reduzieren sich vermeidbare Reibungsverluste, Risiken und Kosten in der Projektabwicklung.

22.2. Getting Started – MDSD-Pilotierung

Ein allererster Schritt in Richtung MDSD besteht oftmals in einem vorgelagerten „Proof of Concept“ für die technischen Facetten (z.B. Auswahl von Tools – vgl. <Ref Tools>) von MDSD im jeweiligen Kontext. Ist dies erfolgreich, schließt sich ein Pilotprojekt an.

Bei der Frage nach einem *geeigneten* Pilotprojekt werden wir oftmals mit Aussagen konfrontiert wie „Projekt X wäre eigentlich gut geeignet, aber der Zeitplan ist so eng, dass wir das Risiko, etwas Neues auszuprobieren, lieber nicht eingehen wollen“. Dazu ist zu sagen, dass erstens fast alle Projekte einen engen Zeitplan haben, so dass man vermutlich nur schwer eine Alternative findet und zweitens MDSD gerade dazu gedacht ist, genau diese Situation letztlich zu entschärfen. Anders ausgedrückt: Genau solche Projekte sind im Grunde ideale Kandidaten – zumindest wenn folgende Randbedingungen erfüllt sind oder erfüllt werden können:

- Idealerweise besitzt das Projekt genügend „Anziehungskraft“, um als Kristallisationskeim in Sachen MDSD zu dienen.
- Das Projekt besitzt eine ausreichende Breite, Laufzeit und genügend schematische Aspekte, so dass sich eine Automation lohnt.
- In jedem Fall sollte das „start small“-Prinzip angewandt werden. Unter den richtigen Voraussetzungen kann durchaus ein technisch schwieriges, risikobehaftetes Projekt angegangen werden, solange an der Entwicklung nicht mehr als zehn Personen beteiligt sind. Natürlich sollte das Projekt auch kein Ein-Mann-Projekt sein. Ein überschaubares Team bietet die beste Grundlage für erfolgreichen Wissenstransfer.
- MDSD wird vom Team mindestens als Experiment akzeptiert. Es ist durchaus normal, dass zunächst Überzeugungsarbeit geleistet werden muss. Insbesondere müssen die Entwickler erkennen, dass sie durch den Ansatz nichts verlieren, sondern gewinnen (es wird ihnen nämlich lästige Copy-Paste-Programmierung abgenommen).

- Es ist im Team (ggf. durch externes Coaching) ausreichend MDSD-Skill vorhanden.
- Die Softwareentwicklungsumgebung sollte für MDSD nicht kontraproduktiv sein. Dazu zählen geschlossene CASE-Tools, (UML-)Tools, die ein Roundtrip Engineering erzwingen (vgl. <Ref EuA>), geschlossene Entwicklungsumgebungen mit eigenem Sourcecode-Repository ohne Schnittstelle und generative IDEs mit proprietären „Modellen“, die sich nur interaktiv erstellen lassen und daher nicht durch eine MDSD-Transformation erzeugt werden können.
- Wählen Sie idealerweise schon im Vorfeld geeignete MDSD-Werkzeuge aus (s.o.).

22.2.1. Risikoanalyse

Das zusätzliche Risiko, welches MDSD in ein Pilotprojekt trägt, ist gering, denn es ist zu jedem Zeitpunkt möglich, die formale Modellierung und Generierung aufzugeben, wenn man feststellen sollte, dass dies aus irgendeinem Grunde für das Projekt nicht trägt. Die Arbeit, die bis dahin in Architektur und Systematik investiert wurde, ist jedoch sicher nicht vergebens, sondern per se schon ein Gewinn für das Projekt.

Da auch MDA noch keine standardisierte Transformationssprache hervorgebracht hat, muss sich ein MDSD-Projekt auf einen proprietären Ansatz bzw. ein entsprechendes Tool einlassen, um Transformationen/Generatoren zu erstellen. Die Code-Masse dieser „Metaprogramme“ ist jedoch im Verhältnis zum generierten Code (vgl. Auswertung im Kapitel <Ref Öko>) verschwindend gering. Ob ein Tool grundsätzlich ungeeignet oder suboptimal für MDSD ist, wird man während des Pilotprojektes relativ früh feststellen, so dass eine eventuelle Migration der Transformationen auf ein anderes Tool keinen signifikanten Overhead darstellen sollte.

Investitionsschutz und Unabhängigkeit von einem Hersteller erreichen Sie insbesondere durch die Verwendung von Open-Source-Werkzeugen, die mittlerweile einen hohen Reifegrad aufweisen (vgl. <Ref-Tools>).

22.2.2. Projektinitialisierung

Die Initialisierung des Pilotprojektes sollte Folgendes umfassen:

- Das Team muss mit den Grundlagen von MDSD vertraut gemacht werden. Dazu gehören insbesondere auch die in Kapitel <Ref Proz> behandelten prozessualen Aspekte, wie Trennung von Anwendungsentwicklung und Domänenarchitektur-Entwicklung und die zweigleisig iterative Synchronisation. Stellen Sie ggf. einen MDSD-Coach bei.
- Starten Sie mit architekturzentrierter MDSD und lassen Sie das Team zunächst mit dieser MDSD-Variante Erfahrung sammeln.
- Das Team sollte die in Kapitel <Ref Orga> definierten Rollen in angemessener Weise (d.h. in Abhängigkeit zur Projektgröße) einnehmen. Berücksichtigen Sie insbesondere die dynamische Balancierung gemäß Abschnitt <Ref Orga::Personalbedarf> – d.h. planen Sie die Bootstrapping-Phase der Domänenarchitektur und die damit verbundene Rollenauslastung ein. Die Technologie-Experten im Team sollten in der Lage sein, eine auf die konkreten Gegebenheiten zugeschnittene Abschätzung zu geben.

Falls ein Vorlauf für das Bootstrapping der Domänenarchitektur auf Grund von Randbedingungen nicht möglich sein sollte, kann die Einführung über ein separates Teilprojekt erfolgen. In diesem Zusammenhang können wir für Enterprise-Softwareentwicklung folgende Empfehlung geben:

- Bilden Sie ein kleines, schlagkräftiges Teilteam zur Erstellung einer Referenzimplementierung.
- Starten Sie parallel mit der fachlichen Anwendungsentwicklung – und zwar GUI-zentriert – mit dem Ziel, dem Auftraggeber möglichst schnell durch einen fachlich klar definierten Ausschnitt (1. fachliches Inkrement) einen ersten Eindruck der realen Anwendung zu vermitteln. Implementieren Sie zunächst nur die dazu minimal benötigte Businesslogik und kapseln Sie sie so gut wie möglich in separaten Klassen/Modulen. Ersetzen Sie die noch nicht zur Verfügung stehende architektonische Infrastruktur (z.B. Datenbank-Zugriffsschicht) durch einfache Mock-Objekte (vgl. <Ref Test>).
- Die Referenzimplementierung sollte auch architektonische Aspekte wie die Integration mit existierenden Legacy-Anwendungen abdecken (vgl. Abschnitt <Ref Trans::Integr>).
- Zwischen beiden Teams muss ein kontinuierlicher Informationsaustausch in beide Richtungen gewährleistet sein: Das Referenzimplementierungsteam muss die Anforderungen und

Ideen der fachlichen Entwicklung kennen und darauf reagieren, die Anwendungsentwickler müssen frühzeitig wissen, in welche Richtung sich die Referenzimplementierung bewegt.

- Sobald die architektonische Referenzimplementierung in einer ersten Fassung fertig gestellt ist, synchronisieren Sie die fachliche Anwendungsentwicklung. Ab diesem Zeitpunkt werden die Konstruktions- und Programmier-Paradigmen der Referenzimplementierung in der Anwendungsentwicklung übernommen. Je nach Zeitplan und zu Verfügung stehender Kapazität kann ein kleines Teilteam auf bereits bestehende Anwendungsteile parallel Refactorings anwenden und zudem die Mocks durch die Konzepte der Referenzimplementierung ersetzen. Die Feedbackschleife zwischen dem Anwendungsentwicklungsteam und dem kleineren Referenzimplementierungsteam ist nach wie vor von zentraler Bedeutung.
- Die Domänenarchitekten können dann mit der Extrahierung der formalen Aspekte und dem Aufbau der Domänenarchitektur beginnen.
- Sobald diese in einem ersten Release zur Verfügung steht, wird die Anwendungsentwicklung auf MDSD umgestellt und ggf. bereits bestehende Anwendungsteile „nachmodelliert“, so dass auch dort die Umstellung erfolgen kann.

22.3. MDSD-Adaptierung bestehender Systeme

Oftmals besteht der Wunsch, nicht nur neue Systeme modellgetrieben zu entwickeln, sondern die Vorteile von MDSD auch bereits bestehenden Systemen zugute kommen zu lassen und sie damit evtl. sogar zu modernisieren, d.h. auf eine neue technologische Basis umzustellen.

Die in diesem Buch vermittelten Vorgehensweisen und Best Practices gelten genauso auch für diesen Projekttyp – allerdings mit einer anderen Gewichtung und Ausrichtung von Teilschritten beim Bootstrapping der Domänenarchitektur:

- Die Rolle des Prototyps und der Referenzimplementierung übernimmt zum Großteil die bestehende Anwendung. Im Falle einer Modernisierung zeigt die Referenzimplementierung die neue Zielarchitektur.

- Möglicherweise ist die (MDSD-)Plattform noch nicht klar erkennbar oder von der Anwendung separiert. Ggf. muss ein Refactoring stattfinden.
- Möglicherweise entsteht zusätzlicher (Infrastruktur-)Code in der Referenzimplementierung, um die Altanwendung später generativ mit Standard-Schnittstellen oder Adaptern versehen zu können. Evtl. sollen auch *nur* solche Adapter generativ ergänzt werden und die Altanwendung im Kern nicht auf MDSD umgestellt werden. Der neue Infrastruktur-Code muss exemplarisch innerhalb der Referenzimplementierung ausgeführt werden.
- Altanwendungen besitzen meist nicht die klaren Strukturen, die für eine Referenzimplementierung notwendig sind, um z.B. Transformationsvorschriften in Form von Generierungsschablonen abzuleiten. Es muss analysiert werden, welche Bauprinzipien valide sind und welche nicht. Gegebenenfalls müssen exemplarisch einige Anwendungsteile durch Refactoring in den Idealzustand gebracht werden.
- Die Domänenanalyse und die DSL-Konstruktion müssen auf die validen Strukturen der Altanwendung bzw. des neu hinzukommenden Infrastruktur-Codes abgestimmt sein.
- Zur Herstellung eines Referenzdesigns/Modells wird die Altanwendung oder ein Teil mittels der DSL nachmodelliert und daran die Generierung erprobt.
- Auf dem Anwendungs-Entwicklungsstrang werden die in der DSL definierten Teilaspekte der Altanwendung nachmodelliert, daraus der neue Implementierungsrahmen generiert und ggf. verbleibender Code der Altanwendung in diesen Rahmen eingepasst. Im Falle der Modernisierung wird er migriert. Auch hier ist also mit entsprechenden Aufwänden zu rechnen.

An dieser Aufzählung ist deutlich zu sehen, dass je nachdem, wie „tief“ der MDSD-Ansatz in die Altanwendung eingreift, dort erheblicher Umstrukturierungsaufwand entstehen *kann*. Die tatsächliche Situation und insbesondere das Kosten-Nutzen-Verhältnis lässt sich nur im Einzelfall durch eine genaue Analyse ermitteln. Auch bei der Adaptierung von Altanwendungen ist die richtige Balance zwischen DSL, Plattform, Generat und nicht-generiertem Code essenziell.

An dieser Stelle sei noch einmal ausdrücklich vor der Vorstellung gewarnt, dass ein einfaches Reverse Engineering der Altanwendung – das zweifelsohne auch ein Modell erzeugt – eine Lösungsalternative sein

könne: Sie erhalten in Teilen eine Code-Visualisierung Ihrer Altanwendung, mehr nicht. Die Vorteile modellgetriebener Entwicklung lassen sich damit nicht erreichen (siehe <Ref Öko::MDSD>).

22.4. Klassifikation des Software-Inventars

Sind die ersten MDSD-Projekte erfolgreich absolviert, kann der MDSD-Wirkungsbereich vom einzelnen Projekt hin zur einer Gruppe von Projekten/Produktentwicklungen geöffnet werden. Anders ausgedrückt: MDSD kann Eingang in die IT-Strategie des Unternehmens finden. Dies führt dann auf die bereits erläuterten Konzepte *Product Line Engineering* (vgl. <Ref PLE>), *Software-Systemfamilie* (vgl. <Ref Begr::SSF>), *Produktfamilie* und *Produktlinie* (vgl. <Ref Orga::Terminologie>).

Um diese Konzepte zu adaptieren, ist zunächst die Klassifikation des Softwareinventars in Bezug auf die Nützlichkeit und die Wartungskosten der einzelnen Teile sinnvoll. Dieser Schritt kann vor, während oder nach der MDSD-Pilotierung erfolgen, sollte aber als Voraussetzung für den unternehmensweiten MDSD-Einsatz betrachtet werden.

Wenn die Prioritäten bei Softwarewartung und Weiterentwicklung nur durch kurzfristige Ziele beeinflusst werden, geht im Laufe der Zeit die Intention des ursprünglichen Entwurfs verloren. Neben akutem Projekttermindruck fördert die rein buchhalterische Betrachtung von Software als abschreibbaren Wertgegenstand die Sichtweise, dass nur sehr begrenzt in Wartung investiert werden sollte. Diese Sichtweise ist schwer mit der Idee vereinbar, inkrementell eine Domänenarchitektur aufzubauen und zu pflegen sowie Softwareassets innerhalb einer Produktfamilie gezielt wiederzuverwenden.

Durch den Einsatz von domänenspezifischem Fachwissen und einem disziplinierten Vorgehen bei der Entwicklung von strategischen Softwareassets (Modelle, Komponenten, Frameworks, Generatoren, Sprachen und Methodenbausteinen) aus Teilen bestehender Software *steigt der Wert der Software*, ganz im Gegensatz zur herkömmlichen Softwarewartung. Der Wert von strategischen Softwareassets wird am besten ausgeschöpft, wenn eine langfristig angelegte Investmentstrategie verfolgt wird. Sie ist erforderlich, um falsche Kompromisse bei der Qualität der Domänenarchitektur zu verhindern. Auch aus dieser Perspektive ist also die bewusste Trennung von Domänenarchitektur-Entwicklung und Anwendungsentwicklung förderlich: In der Anwendungsentwicklung können kurzfristige taktische Entscheidungen helfen, Time-to-Market-Ziele zu erreichen. Im Rahmen der Domänenarchitekturentwicklung kann parallel an generellen Lösungen

gearbeitet werden, welche dann ab einem gewissen Release zur Verfügung stehen. Hier bietet der MDSD-Ansatz erhebliche Vorteile, da nachträgliche Änderungen in der Architektur deutlich leichter in die gesamte Codebasis eingearbeitet werden können.

Natürlich ist es nicht sinnvoll, alle Softwareteile, welche in einer Anwendung verwendet werden, als strategische Softwareassets zu bewerten. Zur Investitionsplanung in Software eignet sich das folgende Klassifizierungsschema:

- **Stategische Softwareassets** bilden das Herz Ihres Unternehmens. Es sind Werte, die zu einer menschen- und maschinenlesbaren „Wissensbasis“ über Ihre Produkte und Geschäftsprozesse weiterentwickelt werden.
- **Nicht-strategische Softwareassets** sind notwendige Infrastrukturen, welche den Änderungen von Implementierungstechnologien ausgesetzt sind und über zwei bis drei Jahre abgeschrieben werden sollten.
- **Altlasten** sind Software-Komponenten oder -Systeme, die nicht-vertretbare Wartungskosten verschlingen.

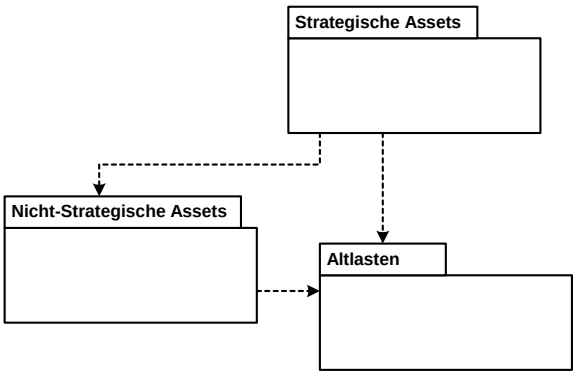


Abb. 175*Klassifikation des Softwareinventars*

Die Identifizierung von strategischen Softwareassets ist nur möglich, wenn eine klare Geschäftsstrategie vorliegt, wenn die gewinnbringenden Geschäftsprozesse im Unternehmen bekannt sind, und wenn das Unternehmen in der Lage ist, die mit diesen Prozessen verbundenen Softwareanforderungen zu artikulieren. Die strategischen Softwareassets eines Unternehmens definieren den Wettbewerbsvorteil gegenüber der Konkurrenz. Die Funktionalität von nicht-strategischen Softwareassets ist nicht einmalig und kann von einer Reihe von Herstellern erworben werden. Als konkretes Beispiel seien hier nur Betriebssysteme, relationale Datenbanken und Applikationsserver genannt. Strategische

Softwareassets bauen typischerweise auf nicht-strategischen (Infrastruktur-)Assets auf. Letztere müssen identifiziert werden und als solche behandelt werden.

Modellgetriebene Softwareentwicklung stellt Werkzeuge zum Management von strategischen Softwareassets zur Verfügung.

Modellgetriebene Integration erlaubt die Benutzung von kommerziellen Fremdprodukten zur ökonomischen Bereitstellung von nicht-strategischen Softwareassets.

Die Benutzung von Open-Source-Infrastruktur als öffentliches Asset führt zu reduzierten Kosten bei der Erstellung und bei der Wartung von strategischen Softwareassets.

22.4.1. Bauen, Kaufen oder Open Source

Softwarehersteller sind berüchtigt für das *not-invented-here*-Syndrom. Die Wahl zwischen *Bauen, Kaufen oder Open Source* sollte auf der Basis längerfristiger ökonomischer Überlegungen erfolgen:

- Vergleichen Sie nicht nur die kurzfristigen Kosten, sondern vergleichen Sie Total Cost of Ownership einschließlich Wartungs- und Kapitalkosten.
- Lohnt sich in Anbetracht der Wartungskosten die Entwicklung von Infrastruktur? Ergibt sich ein Wettbewerbsvorteil – und wenn ja, für wie lange?
- Sobald selbstentwickelte Infrastruktur durch (Industrie-)Standards „überholt“ wird, sollten die Konsequenzen der weiteren Wartung mit denen eines Austauschs verglichen werden.

Allerdings sollten Sie nicht leichtfertig hart erarbeitete, domänenspezifische Plattformen oder MDSD-Transformationen (strategische Softwareassets) durch weniger ausgefeilte Standardwerkzeuge ersetzen.

Kommerzielle Standardsoftware von Fremdherstellern ist zum größten Teil als nicht-strategisches Softwareasset zu bewerten. Typischerweise basiert kommerzielle Standardsoftware nur zu einem Teil auf offenen Standards, und Sie können nicht davon ausgehen, dass das Softwareprodukt über beliebig lange Zeiträume unterstützt wird. Investitionen in Produkte von Fremdherstellern sollten als abschreibbare Kapitalinvestitionen betrachtet werden.

Seien Sie ehrlich und deklarieren Sie nicht all Ihre Legacy-Komponenten als strategische Softwareassets. Identifizieren Sie Altlasten und planen Sie deren Ersetzung durch eine Kombination von strategischen und nicht-strategischen Softwareassets.

Wenn der Qualität von strategischen Softwareassets nicht genügend Aufmerksamkeit geschenkt wird, entwickeln sich diese schnell zu Altlasten. Daher gilt in Bezug auf Ihre Produktarchitektur die Extreme-Programming-Regel: *Refactore mercilessly*.

Eine gesunde Mischung aus strategischen und nicht-strategischen Softwareassets kann durchaus nur einen kleinen Kern strategischer Assets beinhalten. Je kleiner dieser Kern, desto leichter ist es, ihn mit der notwendigen Sorgfalt weiterzuentwickeln. Nicht-strategische Softwareassets spielen eine wichtige Rolle und können in gewisser Hinsicht als notwendige Verbrauchsgüter betrachtet werden.

Bewährte Standardsoftware-Komponenten und robuste Open-Source-Infrastruktur-Komponenten, welche von tausenden von Organisationen benutzt werden, sollten bei der Softwareentwicklung nicht von der Hand gewiesen werden. Solide Open-Source-Software kann sogar als strategisches (öffentliches) Softwareasset betrachtet werden. Verwenden Sie gegebenenfalls Open-Source-Infrastruktur, um die mit Herstellerabhängigkeiten verknüpften Risiken zu vermeiden. Bei der Evaluierung von Open-Source-Software sollten Sie sich das zugehörige Lizenzmodell genau ansehen. Einige Lizenzen erlauben die problemlose Nutzung im Rahmen kommerzieller Produktentwicklung, während andere Lizenzen die Nutzung nur erlauben, wenn das Endprodukt wiederum an die gleiche Open-Source-Lizenz gebunden ist.

22.5. Entwurf einer Zulieferkette

Wenn Sie Ihre Software klassifiziert haben und klar ist, welche Softwareassets strategisch bzw. nicht-strategisch sind, können Sie beginnen, eine Zulieferkette für zukünftige Produktfamilien zu entwerfen.

Dazu wenden wir obige Klassifikation auf die wichtigsten MDSD-Artefakte an (siehe <Ref Begr::SSF>):

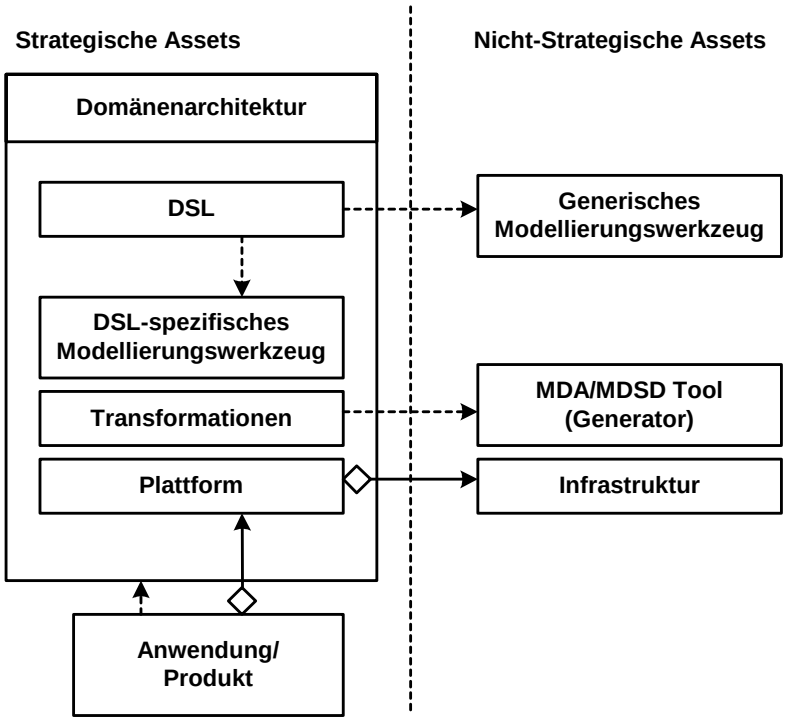


Abb. 176*Klassifikation der MDSD-Artefakte für Zulieferketten*

Die Anwendung erhält ihre Struktur durch die Domänenarchitektur. Die Anwendung beinhaltet die MDSD-Plattform und diese wiederum die Infrastruktur. Unter dem Begriff Infrastruktur sind dabei alle bereits durch die Klassifikation des Inventars (s.o.) ermittelten nicht-strategischen Assets subsumiert. So ergibt sich eine sehr pragmatische Definition für die Grenze zwischen Plattform und Infrastruktur.

Generische Modellierungswerkzeuge (z.B. UML-Tools) sind nicht strategisch, während spezifische Editoren, die für eine konkrete DSL erstellt und optimiert wurden, durchaus als strategisch gelten. Sie tragen relevantes Domänenwissen in sich.

Für die nicht-strategischen Softwareassets gelten die oben angestellten Ausführungen – d.h. hier können z.B. Open-Source-Werkzeuge bzw. Frameworks oder kommerzielle Produkte zum Einsatz kommen.

Anwendungen bzw. Produkte sind natürlich in aller Regel strategisch. Hier bietet sich daher die In-House-Entwicklung an oder ggf. kontrolliertes Offshoring, wie in Kapitel <Ref Orga> erläutert. In diesem Fall ist es wichtig, den Softwareentwicklungsprozess so zu gestalten, dass die Kosten eines potenziellen Wechsels zwischen Offshore-Softwaredienstleistern nicht exorbitant sind. In dieser Hinsicht bietet

MDSD durch die Entkopplung von Anwendung/Produkt und Domänenarchitektur natürlich erhebliche Vorteile. Daher sollte die Entwicklung der Domänenarchitektur und damit der Plattform im Sinne einer Risikominimierung auch besser nicht ausgelagert werden (vgl. <Ref Orga::CtrlOffshore>).

Es sei an dieser Stelle kurz wiederholt, dass der Einsatz von Offshoring zudem erst *nach* erfolgreicher Einführung architekturzentrierter MDSD erfolgen sollte. So ist sichergestellt, dass zum Zeitpunkt des Offshorings ein wohldefinierter Entwicklungsprozess und eine definierte Softwarearchitektur vorliegen, welche durch entsprechende Werkzeuge unterstützt werden.

22.6. Inkrementelle Evolution von Domänenarchitekturen

Sobald mit architekturzentrierter MDSD Erfahrungen in einer ersten Anwendung gesammelt wurden, kann inkrementell eine fachliche MDSD-Plattform aufgebaut werden <Ref Öko::Money>. Auf dieser Basis können dann hochautomatisierte Produktfamilien bzw. Produktlinien entstehen.

Die Domänenarchitektur-Entwicklung erstreckt sich über die gesamte Lebensdauer einer Produktfamilie. Die Kostenverteilung zwischen Domänenarchitektur-Entwicklung und Anwendungsentwicklung hängt von der Reife der Plattform und von der Breite der Produktfamilie ab (vgl. <Ref Öko::Money>).

22.7. Risikomanagement

Die Adaption einer neuen Methodik oder Technik sollte auch im Rahmen des Risikomanagements der involvierten Projekte Berücksichtigung finden. Wir wollen hier typische Risiken behandeln, die eine erfolgreiche Durchführung modellgetriebener Projekte respektive Produktentwicklungen gefährden können. Zu jedem Risiko nennen wir den entstehenden Schaden und Gegenmaßnahmen:

22.8. Risiko: Toolzentriertheit

Beschreibung:
Das Management oder das Projektteam stellt ein Tool (z.B. EAI-Tool,

IDE oder MDA-Tool) ins Zentrum des Projektes, in der Hoffnung, dass es die schwierigsten Probleme löst. Die Softwarearchitektur und der Entwicklungsprozess geraten in den Hintergrund.

Schaden:
Damit geraten auch die eigentlichen Anforderungen in den Hintergrund. Beispiele: Ein Portal wird „gesetzt" – die eigentliche Anforderung lautet aber „Prozessintegration"; oder ein CASE-Tool wird angeschafft, dessen Codegenerierung an den Anforderungen an die Softwarearchitektur vorbeigeht. Die Folge ist meist, dass entweder das Werkzeug „verbogen" wird oder eine nicht-tragfähige Softwarearchitektur bzw. Entwicklungsprozess entsteht. In beiden Fällen resultieren oftmals sehr hohe (Folge-)Kosten, oder Projekte scheitern sogar.

Maßnahmen:
Definieren Sie zunächst grob die Softwarearchitektur und den Entwicklungsprozess und führen dann eine Werkzeug-Projektion durch, d.h. wählen Sie geeignete Werkzeuge zur Überdeckung der einzelnen Aspekte aus. Im Rahmen von MDSD kann die Referenzimplementierung zur Validierung herangezogen werden.

22.8.1. Risiko: Für MDSD kontraproduktive Entwicklungs-Toolkette

Beschreibung:
Dieser Punkt wurde zum Teil schon in Abschnitt 22.2 beleuchtet: Geschlossene, integrierte Entwicklungsumgebungen können den Einsatz außerhalb liegender Codegeneratoren erheblich erschweren. Den anderen „Stolperstein" bilden Modellierungs-Tools, die nicht Multiuser-fähig sind und damit das modellgetriebene Arbeiten im Team erschweren bzw. verhindern.

Schaden:
Entweder entstehen Mehraufwände, um einen funktionierenden Entwicklungs- und Build-Prozess zu etablieren, oder der Mehraufwand wird in die Anwendungsentwicklung verlagert und damit vervielfacht.

Maßnahmen:
Wählen Sie die Werkzeuge in der MDSD-Toolkette anforderungsgetrieben aus (vgl. <Ref Tools>).

22.8.2. Risiko: Überlastetes Domänenarchitekturteam

Beschreibung:
Der rollenbezogene Personalbedarf oder der benötigte Vorlauf beim Bootstrapping der Domänenarchitektur wird nicht ausreichend berücksichtigt, oder das Verhältnis von Domänenarchitekten zu Anwendungsentwicklern ist kritisch.

Schaden:
Die Anwendungsentwicklung wird durch fehlende Features oder späte Lieferungen ausgebremst. Im Extremfall kann das zum Scheitern des Ansatzes führen.

Maßnahmen:
Berücksichtigen Sie die in Abschnitt <Ref Orga::Personal> skizzierte Rollen-Dynamik beim Personalbedarf.

22.8.3. Risiko: Wasserfall-Vorgehensmodell, datenbankzentrierte Entwicklung

Beschreibung:
Zu Beginn der Realisierung liegen abgenommene, „fertige“ DV-Konzepte, GUI- oder Datenbank-Designs vor, die als unverrückbar gelten, in Wahrheit jedoch unvollständig und ggf. sogar inkonsistent sind. Die Datenbank-Entwicklung findet außerhalb des MDSD-Projektes statt.

Schaden:
Ganz allgemein erhöhen sich dadurch die Projektrisiken im Vergleich zu inkrementell iterativen Vorgehensweisen. Bei einem MDSD-Projekt wird die zweigleisig-iterative Entwicklung (vgl. <Ref Proz::IterX2>) behindert. Im Falle der Abkopplung der Datenbank-Entwicklung wird das Potenzial des ganzheitlich generativen MDSD-Ansatzes evtl. nicht optimal genutzt.

Maßnahmen:
Etablieren Sie einen inkrementell iterativen Entwicklungsprozess.

22.8.4. Risiko: Elfenbeinturm

Beschreibung:
Die Domänenarchitekten arbeiten ohne ausreichende Projektkopplung und Feedbackschleifen.

Schaden:
Es entstehen Frameworks, die an dem eigentlichen Projektbedarf vorbei gehen oder nicht „belastbar“ sind. Stattdessen enthalten sie oft unnütze „Technik auf Vorrat“. Die Anwendungsentwickler müssen dies dann kompensieren, wodurch sie wiederum in den „Zuständigkeitsbereich“ der Architekten gelangen, so dass Zwistigkeiten und gegenseitige Schuldzuweisungen vorprogrammiert sind. Insgesamt kann großer wirtschaftlicher Schaden entstehen.

Maßnahmen:
Etablieren Sie regelmäßige Feedbackschleifen und sorgen Sie für möglichst kurze Kommunikationswege. Fördern Sie Teamverständnis. Wechseln Sie ggf. Rollenbesetzungen zwischen Anwendungsentwicklern und Domänenarchitekten.

22.8.5. Risiko: Keine Trennung von Anwendung und Domänenarchitektur

Beschreibung:
Das MDSD-Rollenmodell (siehe Abschnitt <Ref Orga::Roles>) impliziert nicht notwendigerweise eine *personelle* Trennung zwischen Anwendungsentwicklung und Domänenarchitektur-Entwicklung. Die Trennung der Tätigkeiten und *Artefakte* jedoch ist essenziell (siehe Kapitel <Proz>). Ist dies nicht gewährleistet, vermischen sich Anwendung und Domänenarchitektur.

Schaden:
Die Domänenarchitektur ist nicht oder nur schlecht in anderen/weiteren Projekten wiederverwendbar. Die Evolution findet dann ggf. zwar noch innnerhalb der Projekte statt – aber unterschiedlich und mit entsprechend vielen Derivaten. Dadurch entsteht entweder ein erheblicher Refactoring-Bedarf oder aber der Gedanke der Software-Systemfamilie (vgl. <Ref Begr::SSF>) wird fallen gelassen. Im ersten Fall entsteht Mehraufwand, im zweiten Fall wird das MDSD-Potenzial nicht voll ausgeschöpft.

Maßnahmen:
Achten Sie darauf, dass die Domänenarchitektur insbesondere auch auf Package- und Repository-Ebene von den Anwendungen entkoppelt ist. Es darf keine Abhängigkeiten von Artefakten der Domänenarchitektur auf eine nutzende Anwendung geben. Sorgen Sie ggf. für Refactoring. Unabhängig von der Rollenbesetzung muss das Team die Domänenarchitektur-Entwicklung als eigenständiges (Sub-)Projekt begreifen (vgl. <Ref Proz::IterX2>).

23. MDA-Standard

Dieses Kapitel beschreibt wichtige Bestandteile der MDA. Wir werden dabei nicht auf alle Details eingehen, da es zu diesem Thema bereits einige Bücher (z.B. [Fra02]) und Ressourcen gibt – nicht zuletzt den Standard selbst.

Das Kapitel ist unterteilt in drei Teile. Wir beschreiben zunächst kurz die Ziele der MDA und anschließend die zentralen Konzepte des Standards. Zum Schluss gehen wir auf zukünftige Herausforderungen für die MDA (und zum Teil auch für MDSD) ein. Für den aktuellen Stand der Standardisierung verweisen wir auf die Webseiten der OMG [OMGM].

23.1. Ziele

Model-Driven Architecture (MDA) ist ein Begriff, der sich zunehmend in der Fachpresse wiederfindet und ganz unterschiedlich verwendet wird. Wir sehen die MDA als die Standardisierungsinitiative der OMG zum Thema MDSD an. Da die MDA jedoch (bisher) nicht das gesamte MDSD-Spektrum abdeckt, kann man die MDA auch als eine MDSD-Ausprägung respektive Interpretation auffassen.

MDA ist ein junger Standard der Object Management Group [OMG]. Die OMG selbst wurde 1989 gegründet und ist ein offenes Konsortium aus derzeit ca. 800 Firmen weltweit. Die OMG erstellt herstellerneutrale Spezifikationen zur Verbesserung der Interoperabilität und Portabilität von Softwaresystemen und ist traditionell eine Plattform für Middleware- und Tool-Hersteller zur Synchronisation und Standardisierung ihrer Betätigungsfelder. Bekannte Ergebnisse sind CORBA (Common Object Request Broker Architecture) und IDL, UML (Unified Modeling Language), MOF (Meta Object Facility) und XMI. Das neue Flaggschiff der OMG heißt MDA.

Gemäß der Ausrichtung der OMG lauten die beiden primären Motivationen für MDA *Interoperabilität* (Herstellerunabhängigkeit

durch Standardisierung) und *Portabilität* (Plattformunabhängigkeit) von Softwaresystemen – dieselbe Motivation also, die bereits zur Definition von CORBA geführt hat. Dazu postuliert die OMG, die Spezifikation von Systemfunktionalität von der Spezifikation der Implementierung dieser Funktionalität auf einer gegebenen Plattform zu trennen. Aus dieser Sicht hat MDA den Zweck, Richtlinien und Standards zur Verfügung zu stellen, die auf eine entsprechende Strukturierung von Spezifikationen in Form von Modellen führen. Dies verspricht nicht zuletzt eine verbesserte *Wartbarkeit* von Softwaresystemen *durch Trennung von Verantwortlichkeiten* sowie die *Handhabbarkeit von Technologiewandel.*

23.2. Kernkonzepte

23.2.1.UML 2.0

UML ist aus Sicht der MDA zentral, weil sehr viele Tools auf UML und Profilen beruhen (werden). Damit dies mit der UML auch wirklich funktioniert, hat die OMG im Rahmen der UML 2.0 einige wichtige Anpassungen vorgenommen, die wir hier kurz schildern möchten[65].

- **Superstructure:** Intern basiert die UML nun nicht mehr nur „so ungefähr“ auf der MOF, sondern formal. Das komplette UML-Standarddokument enthält Definitionen der UML-Sprachkonstrukte (also des Metamodells) mittels MOF-Modellen. Die UML ist nun also formal beschrieben. Dies ist – wie inzwischen klar geworden sein sollte – Voraussetzung für modellgetriebene Softwareentwicklung, insbesondere für Modelltransformationen und Codegenerierung. Auch die OCL verwendet nun die MOF als Metametamodell, die MOF musste dazu entsprechend erweitert werden. UML und OCL basieren nun also auf demselben Metametamodell und sind insofern im Kern konzeptionell kompatibel.
- **Erweiterung, Profiles, Stereotypen:** Die Definition (und zum Teil auch die Notation) von Profilen und Stereotypen, allgemein

[65] Es wurde noch viel mehr geändert bzw. hinzugefügt, als wir hier beschreiben. Wir konzentrieren uns hier nur auf die für MDA direkt relevanten Aspekte.

also der Erweiterungsmöglichkeiten der UML, wurden überarbeitet. Wir gehen hier nicht weiter darauf ein, da diese Eigenschaften im Kapitel <Ref MM> erläutert werden.

Auch wenn formal alle MOF-basierten Modelle im Rahmen der MDA einsetzbar sind, wird sich – schon aufgrund der entsprechenden Werkzeugunterstützung – vor allem die UML und entsprechende Profile zur Modellierung im MDA-Umfeld durchsetzen (was auch an den unten geschilderten Core Models erkennbar ist). David Frankel erörtert in [Fra02] die Vor- und Nachteile der UML in diesem Zusammenhang. Zunächst die Vorteile:

- Trennung von konkreter und abstrakter Syntax
- Erweiterbarkeit (durch Profile)
- Plattformunabhängig
- Standardisiert
- Pre- und Postconditions, Invarianten sind via OCL möglich (Design by Contract)

Auf der anderen Seite stehen allerdings auch eine ganze Reihe von Nachteilen, die sich allerdings noch auf UML 1.x beziehen – UML 2 hat einige dieser Aspekte verbessert:

- UML ist groß und schlecht partitioniert
- Keine konzeptionelle Unterstützung für Viewpoints
- Komponenten und Pattern sind zu schwach unterstützt (durch UML 2 verbessert)
- Vages Relationship-Modell
- Beschränkte Mittel für Profile, bzw. allgemein Anpassung des Metamodells
- UML und MOF sind (noch) nicht richtig aufeinander ausgerichtet (durch UML 2 verbessert)
- Diagramm-Interoperabilität fehlt (durch UML 2 verbessert)
- Keine abstrakte Syntax für OCL (durch UML 2 verbessert)

23.2.2.MOF – die Meta Object Facility

Kern der MDA ist die MOF, die Meta Object Facility (siehe <Ref MM>). Diese beschreibt das Metametamodell, welches MDA-konformen Tools zugrunde liegt (oder liegen sollte). Die Definition eigener DSLs bzw. Metamodelle sollte die Mechanismen der MOF verwenden (oder die UML mittels Profilen erweitern, was letztendlich auch auf eine Metamodellerweiterung mittels MOF hinausläuft). Auch die UML ist mittels der MOF definiert. MOF selbst verwendet den konkreten Syntax der UML, was zu Verwirrungen führen kann[66].

Wir haben die Mechanismen der MOF an vielen Stellen im Buch implizit verwendet. Immer dann, wenn wir das UML-Metamodell erweitert haben, haben wir automatisch die MOF verwendet.

Die MOF ist nicht nur als formale Basis für Metamodelle wichtig, sondern hat auch ganz konkrete Bedeutung für die Konstruktion von MDA-Tools wie Repositories, Modellierungswerkzeuge, Codegeneratoren etc. Generische Tools brauchen eine solide Basis; diese kann nur das Metametamodell sein. Auch wenn Portabilität von Tool-Daten gewährleistet werden soll, muss man sich auf ein Metametamodell einigen. Es ist also für die Standardisierungsbemühungen der OMG und einen vernünftigen Tool-Markt unerlässlich, das Metametamodell vollständig, formal und in sich korrekt zu definieren.

Abb. 177 zeigt einen Ausschnitt aus der MOF.

---

[66] Genau genommen ist es seit UML 2.0 so, dass der Kern der UML (also das *classifier*-Package) auf allen Metaebenen eingesetzt werden kann und damit also auch den Kern der MOF darstellt.

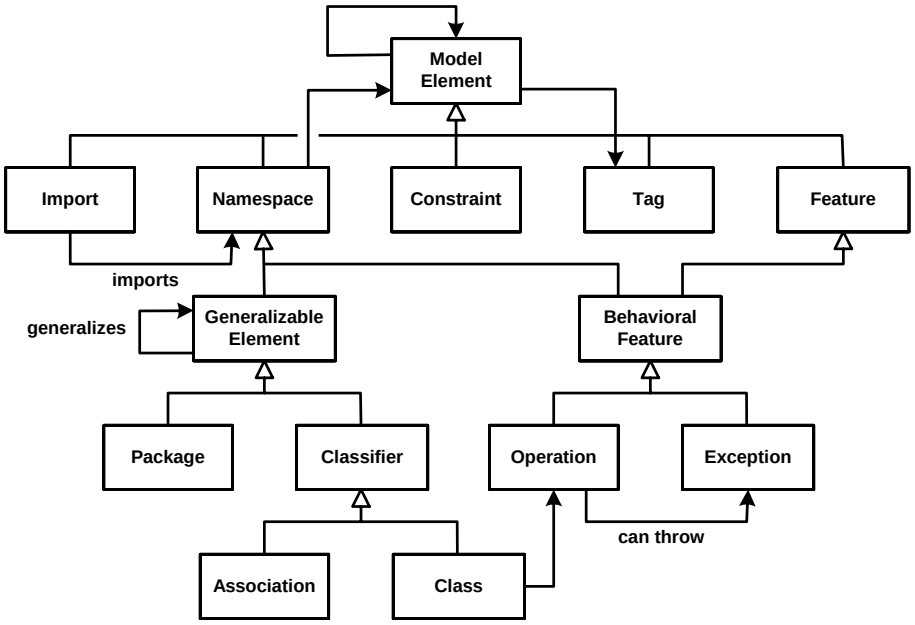


Abb. 177*Ein Ausschnitt aus der MOF*

Die MOF hat allerdings auch einige Nachteile (wobei man sich darum streiten kann, ob die folgenden Aspekte in die MOF gehören sollten). Beispielsweise bietet die MOF keine Hilfe bei der Definition von konkreter Syntax, Versionierungsproblematiken, und die Komposition von Metamodellen aus Teilmetamodellen wird nicht adressiert.

Neben der MOF als Metametamodell sowie ihrem Serialisierungsformat XMI (siehe nächster Abschnitt) gibt es außerdem noch vordefinierte APIs, um MOF-Modelle in verschiedenen Sprachen zu bearbeiten bzw. zu speichern – in Java ist der JMI-Standard ein Beispiel [JMI]. Basierend auf einem in MOF definierten Metamodell kann man sich ein Repository bzw. eine API generieren lassen, mit der man Instanzen des betreffenden Metamodells bearbeiten kann. Im Umfeld von Eclipse erledigt EMF eine ähnliche Aufgabe [EMF].

23.2.3.XMI

XMI steht für XML Metadata Interchange und ist ein XML-Mapping für MOF (und nicht etwa nur eine DTD/Schema für UML, wie fälschlicherweise oft behauptet wird). Derzeit ist XMI die Basis für die Interoperabilität zwischen verschiedenen MDA-Tools, da MOF-Repositories noch nicht besonders weit verbreitet sind. Ab Version 2.0

erlaubt XMI auch die Serialisierung von Diagramm-Layoutinformationen, was für einen funktionierenden Modellaustausch zwischen Modellierungswerkzeugen (und nicht nur zum Zwecke der Codegenerierung) unerlässlich ist, wenn man sich nicht auf die Autolayout-Mechanismen der gängigen Tools verlassen möchte.

Aktuell gibt es immer noch eine ganze Reihe von Inkompatibilitäten zwischen den XMI-Formaten verschiedener Tools, was den Diagrammaustausch zwischen Modellierungswerkzeugen erschwert. Die Verwendung von XMI als Basis für Codegeneratoren ist hingegen in der Praxis kein großes Problem, weil alle verbreiteten Generatoren Parser für die verschiedenen XMI-Dialekte unterstützen.

Es gibt zwei Spielarten von XMI. Die eine ist die komplett generische, die mittels einer allgemeinen DTD alle MOF-basierten Modelle abspeichern kann. Die Dokumentenstruktur ist also auf MOF-Level definiert, was die darauf basierenden XMI-Dokumente recht umfangreich werden lässt. Dafür ist es aber auch generisch für alle MOF-basierten Modelle verwendbar. Für menschliches Lesen oder XSLT-basierte Transformationen eignet es sich aber nur sehr bedingt.

Auf der anderen Seite umfasst der XMI-Standard auch die Möglichkeit, für ein bestimmtes, auf Basis der MOF definiertes Metamodell eine DTD bzw. ein Schema zu generieren. Dokumente können dann zwar nur noch Instanzen dieses Metamodells speichern, dafür ist die resultierende Datei aber sehr viel kompakter und konkreter, da die Strukturen auf dem Niveau des Metamodells und nicht des Metametamodells abgebildet werden.

Die meisten Tools verwenden zum Austausch naheliegenderweise den ersten Ansatz.

23.2.4.PIM/PSM/PDM

Die OMG hat eine recht konkrete Vorstellung vom „Sinn“ der MDA. Es geht ihr nicht in erster Linie um Software-Systemfamilien, Einbindung von Fachexperten oder gesteigerte Agilität in der Softwareentwicklung, sondern vor allem um Plattformunabhängigkeit der Anwendungslogik. Dadurch, dass sich sowohl technologische Lösungen als auch die Businesslogik schnell, aber voneinander unabhängig weiterentwickeln, ist es im Sinne der Langlebigkeit essenziell, die Anwendungslogik unabhängig von einer Implementierungsplattform spezifizieren zu können – sozusagen in Reinform. Die OMG sieht dazu eben MOF- bzw. UML-basierte Modellierung als besten Lösungsweg an. Daraus können mittels entsprechenden Transformatoren die Implementierungen für verschiedene Plattformen (mehr oder weniger) vollautomatisch generiert

werden. Zentralen Stellenwert hat daher das PIM, das Platform Independent Model. Es beschreibt die Businesslogik in ihrer Reinform.

Durch Modell-Transformation wird aus dem PIM ein PSM erstellt. Das PSM – Platform Specific Model – ist nun, wie der Name schon sagt, plattformspezifisch, zum Beispiel spezifisch für J2EE oder .NET. Weitere Transformationen können immer spezifischere Modelle erstellen, bis letztendlich der Quellcode für eine Plattform generiert wird, der dann durch Übersetzung und Packaging zum lauffähigen Artefakt wird.

Anhand dieser Diskussion ist – wie auch schon früher erläutert – ersichtlich, dass es sich bei PIM und PSM um relative Konzepte handelt. Ein PSM ist möglicherweise spezifisch für J2EE, aber noch unabhängig von der Frage, für welchen Applikationsserver es optimiert ist; es stellt also ein PIM bezüglich der konkreten Applikationsserverplattform dar.

Abb. 178 zeigt eine Taxonomie der Modelle, die in der MDA eine zentrale Rolle spielen (siehe auch [Fra02]).

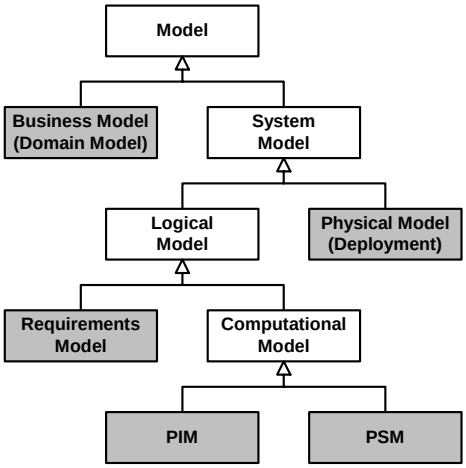


Abb. 178*Klassifizierung von Modellen*

Es gibt noch eine weitere wichtige Art von Modellen in der MDA, das Platform Description Model (PDM). Dies ist letztendlich das Metamodell der Zielplattform. Dadurch, dass Modell-Transformationen immer als Transformationsregeln zwischen Metamodellen definiert werden, ist es essenziell, dass auch die Zielplattform mittels eines Metamodells definiert ist. Die Quellmodelle verwenden ja sowieso ein domänenspezifisches Metamodell (ggf. ein standardisiertes Core Model, siehe unten). Im Rahmen architekturzentrierter MDSD (vgl. <Ref Begr>) setzt die Modellierung direkt auf dem PDM (dem Architekturmetamodell der Zielplattform) auf.

23.2.5. Mehrstufige Transformationen

In den Beispielen und Best Practices dieses Buches haben wir meist aus (PIM-)Modellen Quellcode für eine bestimmte Plattform generiert. Wir tun dies vor allem aus Gründen des Pragmatismus: Modell-Transformationswerkzeuge, die mehrstufige Transformationen für große Systeme alltagstauglich unterstützen, existieren bisher noch nicht.

Die OMG verfolgt mit der MDA das Ziel, mittels aufeinander folgender Modell-zu-Modell-Transformationen letztlich zum Quellcode zu kommen. Dies hat – entsprechende Werkzeuge vorausgesetzt – durchaus Vorteile. Das Beispiel in <Ref Tool::Archi::ModTrans> belegt dies. Anhand des Beispiels wird klar, warum man im Rahmen der MDA versucht, möglichst viel auf Modellebene zu tun und auch die Informationen möglichst lange in Modellform zu belassen. Die Transformations-Engines werden eben auf Basis der MOF definiert. Solange wir uns in MOF-basierten Metamodellen bewegen, sind diese Tools anwendbar. Sobald wir uns „in die Niederungen“ von (natürlich nicht MOF-basierten) Programmiersprachen herablassen, können diese Tools nicht mehr viel tun.

In einigen Fällen ist es nötig, die „Zwischenprodukte“ weiter manuell zu konfigurieren, um die weiteren Transformationsschritte zu steuern. Die OMG nennt diese Konfigurationen Model Markings. Diese Model Markings können nicht direkt im PIM annotiert werden, weil damit die Plattformunabhängigkeit aufs Spiel gesetzt würde. Aus Konsistenzgründen ist es allerdings auch kritisch, die Zwischenmodelle zu modifizieren. Wir schlagen deshalb die Verwendung von externen Modellmarkierungen vor, wie es in <Ref Trans::BP-Tech::MdlMrk>> erläutert ist.

23.2.6. Action-Sprachen

Wahrscheinlich ist beim Lesen des Buches irgendwann der gerechtfertigte Einwand aufgekommen, dass man heutzutage keine *kompletten* Softwaresysteme mittels UML (bzw. anderen MOF-basierten Sprachen) modellieren kann. Dies ist zum Teil wahr. Allerdings haben wir ja in diesem Buch bereits gesehen, dass man tatsächlich Anwendungen sehr weit gehend modellieren kann, wenn man die Domäne weit genug einschränkt, Konzepte stark standardisiert und dann dafür eine passende DSL definiert. Natürlich bleiben auch dann noch viele Dinge offen oder unpraktikabel. Insbesondere fehlte bisher eine Möglichkeit, algorithmisches Verhalten zu spezifizieren.

Um dieses Problem anzugehen, definiert die OMG mit UML 2.0 die Action Semantics (welche sich nicht nur mit UML einsetzen lässt, sondern auch mit anderen MOF-basierten Sprachen). Action Semantics erlaubt, prozedural Verhalten zu modellieren. Zu beachten ist, dass die OMG keine konkrete Syntax definiert, sondern nur die abstrakte Syntax. Die Semantik wird verbal beschrieben. Damit steht es den Werkzeugherstellern offen, eigene textuelle oder grafische Notationen für die standardisierte Semantik zu definieren. Insbesondere wird es damit auch möglich, ein- und dasselbe Verhalten in verschiedenen Formen darzustellen, z.B. textuell oder grafisch.

Die Action Semantics umfasst unter anderem folgenden Elemente:

- Variablen (*Instance Handlers*), Zuweisungen, Lesen, auch für Mengen von Variablen (*Sets, Bags, Sequences*)
- Die üblichen arithmetischen und logischen Operationen
- Typische Features sequenzieller Programmiersprachen wie *Switch*, *If*, *For*, Statements, Blockkonzept
- Instanzerzeugung, Zerstörung von Instanzen
- Klassen-Extents, die mit SQL-artigen Querys abgefragt werden können
- Navigation über Assoziationen
- Erzeugung von Links (also Instanziierung von Assoziationen) sowie die Löschung von Links
- Erzeugung von Signalen inkl. Parametern
- Definitionsmöglichkeit für Funktionen mit Ein- und Ausgabeparametern sowie die Möglichkeit, diese aufzurufen
- Timer

Was die Action Semantics natürlich nicht enthält, sind statische Konstrukte wie Klassen, Attribute und Beziehungen. Diese werden ja bereits im strukturellen Teil des Modells definiert. Action Semantics definieren lediglich „Verhaltensbausteine“, die nur zusammen mit anderen (Teil-)Modellen Sinn machen. Insofern sind Action-Semantics-Segmente immer mit Elementen des regulären UML-Modells assoziiert, also beispielsweise mit Operationen von Klassen oder *onEntry*-Actions in Zustandsmaschinen.

Im Folgenden zeigen wir ein kleines Beispiel für die Verwendung von Action-Sprachen. Abb. 179 zeigt das Klassendiagramm, welches als Basis für das Beispiel dient. Als Syntax für die Action Semantics

verwenden wir hier die Syntax, wie sie in dem Tool iUML von Kennedy Carter [IUML] verwendet wird.

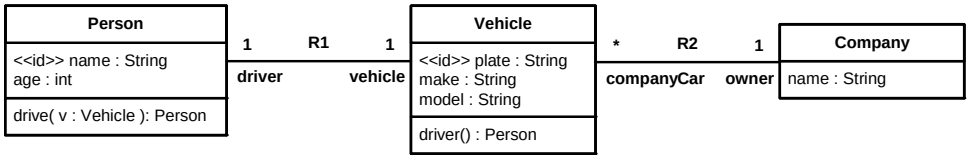


***Abb. 179** Beispielmodell zur Erläuterung von Action Semantics*

Wir implementieren zunächst ein „Hauptprogramm“, welches mit Instanzen der Klassen aus Abb. 179 arbeitet. Zunächst sollten wir eine Instanz der Klasse *Vehicle* erstellen. Danach weisen wir dem Typ einen Wert zu.

```
myVWBus = create Vehicle with plate = “HDH-GS 142”
myVWBus.make = “Volkswagen”
myVWBus.model = “Transporter T4 TDI”
```

Die *with* …-Klausel weist den identifizierenden Attributen (<<*id*>>) bereits bei der Objekterzeugung Werte zu (vgl. Konstruktor). Wir können dann weiterhin eine Instanz von *Person* definieren, die später der Fahrer sein wird.

```
john = create Person with name = “John Doe”
```

Wir können nun die Operation *drive()* aufrufen, um den Fahrer das Fahrzeug fahren zu lassen.

```
[actualDriver] = drive[aVehicle] on john
```

Was natürlich noch fehlt, ist die Implementierung der Operation *drive()*. Das mindeste, was sie tun muss, ist die Assoziation *R1* zu instanziieren (also einen *Link* zwischen den beiden betroffenen Objekten zu erstellen).

```
link this R1 aVehicle
```

Dies erstellt die bidirektionale Assoziation zwischen dem *Vehicle* und einem Fahrer. Man kann nun beispielsweise das Auto befragen, wer es gerade fährt; dies entspricht der Implementierung der *drive()*-Operation der Klasse *Vehicle*.

```
theCurrentDriver = this.R1.“driver“
```

Im Folgenden noch eine kurze Vorstellung der Query-Operationen. Angenommen wir möchten alle Personen im System finden:

```
{allPersons} = find-all Person
```

Die geschweiften Klammern geben an, dass es sich bei *allPersons* um eine Menge von Objekten handelt, und nicht nur um ein einzelnes. Man kann eine solche Suche auch einschränken, beispielsweise könnten alle Fahrzeuge gesucht werden, bei denen es sich um einen Audi handelt.

```
{audis} = find Vehicle where make = „Audi“
```

Man könnte nun kritisieren, dass es sich bei Action Semantics nur um eine weitere Programmiersprache handelt. Dies ist zwar im Prinzip richtig, trifft aber nicht den Kern:

- Man muss sich nicht um Plattformspezifika kümmern (Speicherverwaltung oder z.B. die Definition eines Links zwischen zwei *EntityBean*-Instanzen oder die Verwendung von relationalen Schlüsseln).
- Es handelt sich hier nur um „Semantikbausteine“, die konkrete Syntax kann verschieden aussehen. Dies könnte mit traditionellen Sprachen zwar prinzipiell auch gemacht werden, ist aber unüblich.
- Action Semantics sind vollkommen in das Modell integriert. Das folgende abschließende Beispiel soll dies verdeutlichen.

Angenommen wir haben das triviale Modell in Abb. 180. Rechts daneben sehen wir einige Zeilen Action Language.

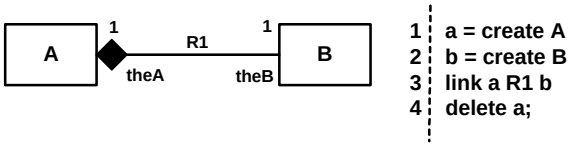


Abb. 180 *Ein weiteres Beispielmodell*

Beim Mapping auf eine Programmiersprache (also bei der Implementierungscodegenerierung) wird daraus grob der folgende Code generiert:

Zeile 1 Erstellen einer neuen Instanz von *A*, Anlegen der Variable *a*.

Zeile 2 Erstellen einer neuen Instanz von *B*, Anlegen der Variable *b*.

Zeile 3 Ab hier wird es interessant: Es wird dem Attribut *theB* von *a* der Zeiger auf *b* zugewiesen. Weil die Assoziation aber bidirektional ist, wird auch *b.theA* automatisch auf *a* zeigen. Dies muss der Programmierer in Action Statements nicht angeben – der Generator kann dies durch die Information im Klassendiagramm automatisch tun.

Zeile 4 Aufgrund der Tatsache, daß *B* kompositorisch mit *A* assoziiert ist, kann der Generator Code erzeugen, der auch die Instanz *b* löscht, wenn *a* gelöscht wird.

Die Informationen für die Generierung werden auch aus dem Strukturdiagramm entnommen. Der Entwickler schreibt in ASL nur das *link* …-Statement. Der potenziell deutlich komplexere Implementierungscode wird automatisch erzeugt.

Um noch mal auf den Abstraktionseffekt der Action Semantics einzugehen: Natürlich sagt auch diese textuelle Repräsentation nichts über die Realisierung aus. Wenn das betreffende System auf EJB realisiert wird, muss natürlich völlig anderer Code erzeugt werden (Factories, Homes, Lookup) als bei einfachem Java, oder womöglich im Zusammenhang mit eingebetteten Systemen.

23.2.7.Core Models

Um möglichst stark von der MDA-Initiative profitieren zu können, muss möglichst viel standardisiert werden. Dazu zählen Plattformen (dies ist durch J2EE, .NET, CORBA oder Webservices zumindest auf technischer Ebene schon geschehen) und Transformationssprachen (dies findet gerade im Rahmen der QVT-Abstimmung statt – s.u.). Um es aber Anwendern zu ermöglichen, die Anwendungslogik in UML unabhängig von der Plattform zu modellieren, ist es nötig, auch das Metamodell für bestimmte Domänen zu standardisieren. Damit wird es möglich, nicht nur die Transformationssprachen zu standardisieren, sondern auch Transformationsregeln in Form von Bibliotheken zu verkaufen. Der Entwickler modelliert seine Anwendung mittels des standardisierten UML-Profils oder Metamodells für die entsprechende Domäne – und die kommerziellen Transformationsmodule erstellen daraus den plattformspezifischen Code.

Diese standardisierten Metamodelle nennt die OMG im Zusammenhang mit der MDA Core Models. Es sind derzeit verschiedene Core Models in der Entstehung[67], die derzeit alle als UML-Profile definiert werden. Darunter sind:

- UML Profile for CORBA (definiert das Mapping vom PIMs auf CORBA).

[67] Wem das bekannt vorkommt: Es gab auch vor MDA schon Bemühungen, Profile für bestimmte Domänen zu standardisieren.

- UML Profile for CCM (definiert das Mapping auf CCM, das CORBA Component Model).
- UML Profile for EDOC (definiert das Metamodell für die Definition von PIMs für verteilte Enterprise-Systeme).
- UML Profile for EAI (definiert das Metamodell für PIMs für EAI-Anwendungen und andere, lose gekoppelte Systeme).
- UML Profile for Quality of Service (QoS) and Fault Tolerance (für Realtime- und sicherheitskritische Systeme).
- UML Profile for Schedulability, Performance and Time (definiert ein Metamodell für PIMs, deren Echtzeiteigenschaften sich quantitativ analysieren lassen).
- UML Testing Profile (unterstützt das automatische Testen in MDA-basierten Systemen).

Der aktuelle Status der verschiedenen Profiles kann unter [OMGP] eingesehen werden.

23.2.8.Steuerung der PIM/PSM-Transformation

In einigen Fällen kann der Transformator ein Modell nicht übersetzen, weil die Informationen im Quellmodell nicht spezifisch genug sind. Beispielsweise kann es im Zielmetamodell mehrere Möglichkeiten geben, ein Konstrukt des Quellmetamodells zu realisieren. David Frankel beschreibt in [Fra02] vier Alternativen, wie man in einem solchen Falle vorgehen kann:

- Es kann in der Transformation fest kodiert werden, dass immer eine bestimmte Alternative verwendet wird.
- Der Entwickler definiert manuell, welche der Alternativen verwendet werden sollen.
- Im PIM kann direkt angegeben werden, welche Alternative im PSM verwendet werden soll (Model Markings – siehe unten).
- Die Entscheidungskriterien, aufgrund derer der Transformator entscheidet, welche Alternative verwendet werden soll, werden in das PIM abstrahiert (auch dazu später mehr).

Model Markings

Die MDA sieht so genannte Model Markings vor. Dies sind zusätzliche Informationen im Quellmodell einer Transformation, die die Transformation „steuern“. Üblicherweise sind diese Annotationen vom Zielmetamodell abhängig. Abb. 181 zeigt das Prinzip.

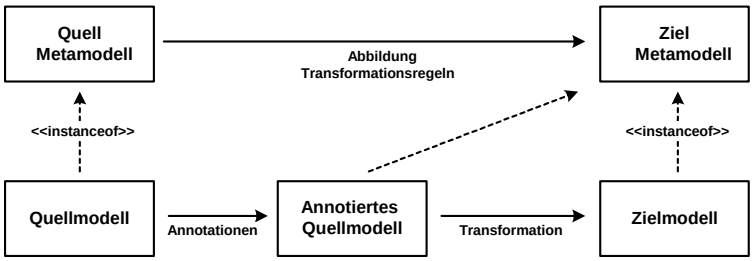


***Abb. 181** Beispiel für Modellmarkierungen*

Als Beispiel soll wieder J2EE dienen. Im PIM definieren wir eine *BusinessEntity* namens *Account*. *BusinessEntities* können in J2EE nun auf zwei verschiedene Arten und Weisen abgebildet werden: Zum einen als *EntityBeans* und zum anderen als *Stateless Session Beans*, die *Data-Transfer-Objekte* verarbeiten. Man könnte also das Quellmodell so markieren, wie es in Abb. 182 gezeigt wird.

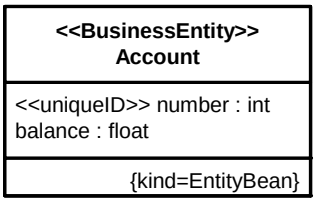


***Abb. 182** Beispiel für eine Entität mit EJB-spezifischen Markierungen*

Zu beachten ist dabei, dass nicht das Quellmodell selbst verändert wird, es wird lediglich eine „Referenzkopie“ definiert, die die zusätzlichen Angaben enthält. Abb. 183 illustriert diese Idee.

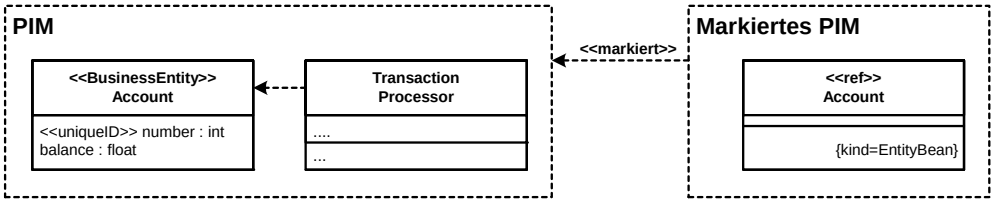


***Abb. 183** Beziehung zwischen PIM und markiertem PIM*

Das markierte PIM enthält also nur die Modellelemente des ursprünglichen PIMs die „markiert“ werden sollen; es enthält auch nur die zusätzlichen Markierungen. Dies bedeutet, dass bei Änderungen am PIM das markierte PIM nicht manuell angepasst werden muss[68].

---

[68] Dies verdeutlicht, wie wichtig es für MDA ist, dass Modellierungswerkzeuge entsprechend mächtige Repositories und

Entscheidungskriterien im PIM

Es sei noch auf eine Möglichkeit hingewiesen, die es erlaubt, weitestgehend ohne Markierungen auszukommen, wenn man andere Konsequenzen in Kauf nimmt. Dieser Mechanismus besteht darin, das Quellmetamodell so zu erweitern, dass genügend Informationen im Modell vorhanden sind, so dass der Generator die Entscheidung zwischen den verschiedenen Alternativen treffen kann. Abb. 184 zeigt, wie das im Beispiel aussehen könnte.

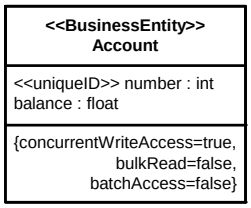


***Abb. 184** Entität mit zielplattformunabhängigen Markierungen*

Das Wissen, wann welche Implementierungsalternative in J2EE verwendet wird, kann nun in die J2EE-Transformationen verlagert werden. Natürlich muss man nun das Quellmetamodell erweitern (was nicht immer möglich ist). Man erweitert es aber nicht direkt mit Konzepten des Zielmetamodells, sondern mit allgemeinen Informationen, die der Transformator verwenden kann. Dies stellt also keine „Verschmutzung", sondern eine Anreicherung des Quellmetamodells dar.

23.2.9. Modelltransformationen

Modelltransformationen überführen ein Modell in ein anderes. Im Rahmen der MDA ist dies neben der Codegenerierung die wohl wichtigste Technologie. In der derzeitigen Praxis spielen Modelltransformationen aber in Ermangelung von Sprachen und Tools eine eher untergeordnete Rolle. Im Bereich der Forschung wird aber intensiv daran gearbeitet. Die OMG hat einen RFP[69] (QVT, Query/Views/Transformations) dafür herausgegeben. Eine Standardisierung ist noch nicht erfolgt.

Funktionalitäten bereitstellen. Die allermeisten Modellierungstools sind davon heute noch weit entfernt.

[69] RFP = Request for Proposal

Wir möchten in diesem Zusammenhang nun etwas näher auf die prinzipielle Mechanik von Modelltransformationen eingehen.

Als Beispiel sollen ein weiteres Mal Enterprise JavaBeans dienen. In Abb. 185 sind zwei (zugegebenermaßen unrealistisch einfache) UML-Diagramme dargestellt. Das linke enthält ein Analysemodell einer Person, das rechte deren Repräsentation als EJB.

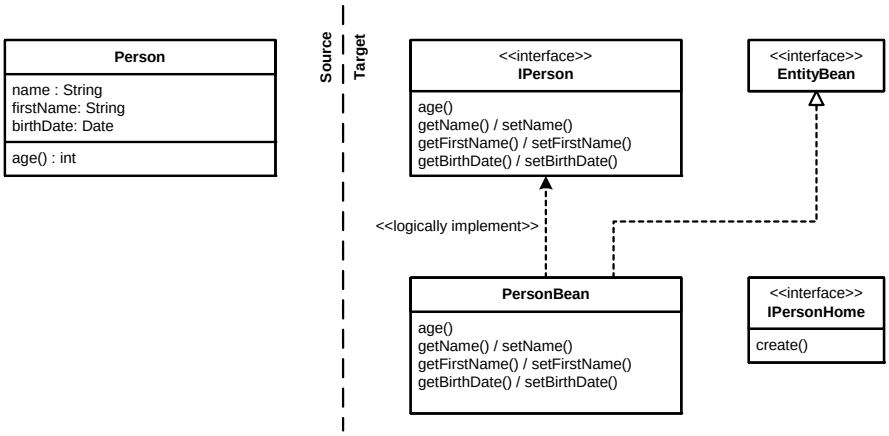


Abb. 185 *Person als Entität und als EJB*

Zunächst ist hier zu beachten, dass das rechte Diagramm direkt auf dem UML-Metamodell aufsetzt und keine EJB-spezifische Erweiterung oder Profil verwendet. Würde man für EJBs ein eigenes Profil definieren, könnte man das wie in Abb. 186 vereinfacht darstellen:

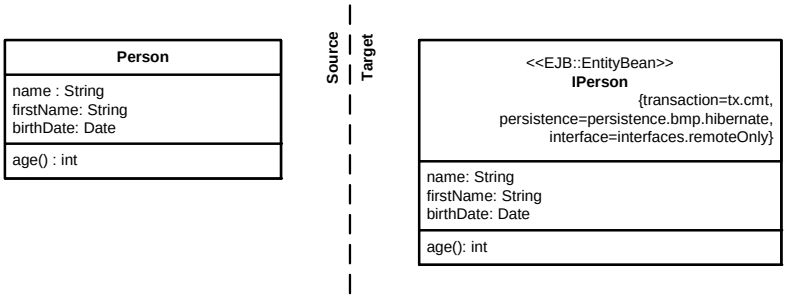


Abb. 186 *Person als Entität und als kompaktere EJB-Darstellung*

Zur weiteren Bearbeitung müsste dann aber später sowieso das native Modell von Abb. 185 generiert werden, weil nur dieses in Programmiersprachen direkt umgesetzt werden kann – wobei natürlich die Alternative besteht, direkt Code zu generieren und das Zwischenergebnis-Modell wegzulassen.

Um die Transformation zu beschreiben, benötigen wir das Metamodell des Ziels und der Quelle. Für die Quelle verwenden wir das

UML-Metamodell. Für das Ziel sollten wir das Metamodell von Enterprise JavaBeans zur Verfügung haben. Ohne Garantie auf Vollständigkeit könnte dies wie in Abb. 187 gezeigt aussehen:

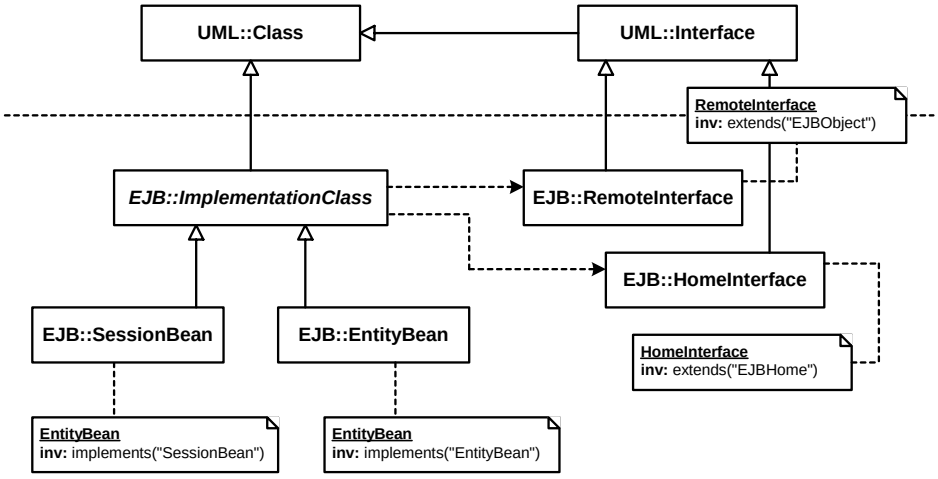


***Abb. 187** Vereinfachtes EJB-Metamodell*

Zu diesem Metamodell sind einige Dinge zu sagen. Zum einen wird nicht ausgesagt, dass die *ImplementationClass* die oben aufgelisteten *Tagged Values* besitzt (*transactions, persistence, interface*). Wir ignorieren auch lokale Interfaces sowie das Constraint, dass die *ImplementationClass* die gleichen Operationen haben muss wie das *RemoteInterface*, plus einiger zusätzlicher Operationen wie *ejbCreate(), ejbRemove()* etc. Zu beachten ist weiterhin, dass wir hier zwei Abkürzungen in den OCL Constraints verwenden. Beispielsweise steht der Ausdruck *implements(„SomeName“)* für das UML-Metamodell-konforme Constraint *specification->exists(specifier.name= “SomeName“).*

Die Transformation vom einen in das andere Modell kann nun auf zwei grundlegend verschiedene Arten spezifiziert werden: prozedural oder deklarativ:

- *Prozedural:* Transformationen werden schrittweise und imperativ definiert.
- *Deklarativ:* Transformationen beschreibt der Entwickler ausschließlich mittels Regeln, die der Transformator dann auf das zu transformierende Modell anzuwenden versucht. Passende Regeln werden ausgeführt, unpassende nicht.

Unabhängig davon gibt es die Möglichkeit, Transformationen textuell oder grafisch[70] zu notieren. Auch eine Kombination beider Arten ist möglich sowie die Darstellung derselben Transformation in verschiedenen Repräsentationsformen, sofern die abstrakte (Transformations-)Syntax identisch ist. Im Folgenden zwei Beispiele.

Prozedural- textuell

Eine textuelle, prozedurale Repräsentation könnte wie folgt aussehen. Die Funktion *createEJBModel()* erzeugt ein neues Modell als Transformation des *source*-Modells, welches als Parameter übergeben wird.

```
function createEJBModel( Model source ): Model {
  Model target = new Model();
```

Zunächst wird über alle Klassen im Quellmodell iteriert.

```
  foreach c:Class in source.classes {
```

Für jede der dort gefundenen Klassen wird im Zielmodell eine neue Klasse angelegt, die denselben Namen hat wie die Quellklasse, ergänzt um das Suffix *Bean*.

```
    ImplementationClass implClass =
                        new ImplementationClass();
    implClass.setName( c.getName()+"Bean" );
    target.addClass( implClass );
```

Wir merken uns jetzt noch aus Gründen der Nachvollziehbarkeit (und um Änderungen effizienter propagieren zu können) die Klasse, aus der die neue Klasse generiert wurde.

```
    Dependencies.define( implClass, c );
```

Analog definieren wir die beiden nötigen Interfaces:

```
    RemoteInterface ri = new RemoteInterface ();
    // set name and add it to target model
    HomeInterface hi = new HomeInterface ();
    // set name and add it to target model
```

Wir legen nun die Operationen an:

```
    foreach o:Operation in c.operations {
```

---

[70] Die direkte Manipulation von XMI-Dateien berücksichtigen wir hier nicht.

```
      ri.addOperation( new Operation( o.clone() ) );
      implClass.addOperation(
          new Operation( o.clone() ) );
    }
  }
  return target;
}
```

Die obige Transformationsfunktion ist operational spezifiziert, also als prozedurales Code-Fragment, welches auf Modellelementen arbeitet. Mittels entsprechender Repository-Software (wie dem NetBeans JMI Repository oder openArchitectureWare [oAW]) kann diese Art von Transformationen heute bereits mehr oder weniger komfortabel implementiert werden.

Grafisch- deklarativ (1)

Grafische Notationen sind gerade erst im Entstehen, und wie auch für die textuellen gibt es noch keinen Standard. Die uns bekannten Proposals haben aber alle gemeinsam, dass sie so weit wie möglich auf vorhandenen UML-Konzepten und Notationen aufbauen. Zunächst ein Beispiel mittels UMLX (siehe [UMLX]).

Bevor wir beginnen, müssen wir allerdings zunächst einen kleinen Exkurs zum Thema Klassendiagramme und Objekt-(oder Instanz-)Diagramme in UML einschieben. Aufgrund der bisherigen Diskussionen zum Thema Metamodellierung sollte klar sein, dass sich jedes UML-Klassendiagramm als Objektdiagramm auf der Meta-Ebene darstellen lässt. Abb. 188 verdeutlicht dies.

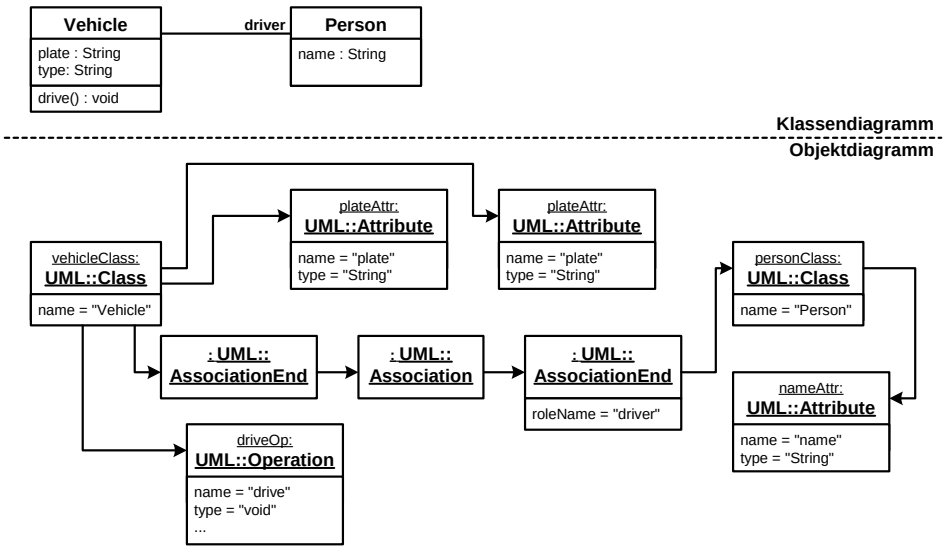


Abb. 188*Klassen- und Objektdiagramm des gleichen Modells*

Die Transformationsdefinitionen in UMLX werden nun mittels einer erweiterten Form von Objektdiagrammen notiert. Objektdiagramme sind deshalb notwendig, weil eine Transformation zunächst eine Selektion der Modell-Objekte im zu transformierenden Klassendiagramm durchführen muss. Abb. 189 zeigt eine beispielhafte Transformation.

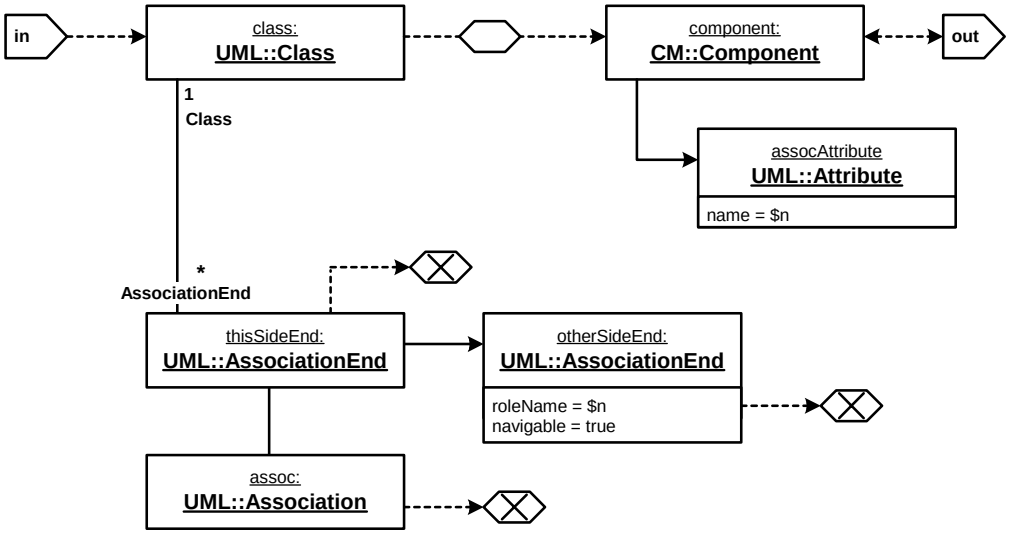


***Abb. 189** Beispiel für eine Transformation mit UMLX*

Als Eingangsdaten werden alle Objekte („Objekte“ sind hier auf der Meta-Ebene die Elemente des Klassendiagramms) herangezogen, die von dem *in*-Symbol aus per Assoziationen erreichbar sind, also *class*, *thisSideEnd*, *otherSideEnd* und *assoc*. Nun wird die Klasse (mit dem Rollennamen *class*) in eine Komponente *component* transformiert, symbolisiert durch das Evolutionssymbol (die gestrichelte Linie mit dem Diamant). Dieses Objekt ist gleichzeitig die Ausgabe der Transformation, symbolisiert durch die Verbindung zum *out*-Symbol. Weiterhin wird für jede Assoziation der Eingangsklasse ein Attribut in der Zielkomponente mit demselben Namen angelegt (wir vernachlässigen hier den Typ). Die anderen Objekte (*thisSideEnd*, *otherSideEnd* und *assoc*) sind im Zielmodell nicht mehr vorhanden, symbolisiert durch das *Removal*-Symbol. Man beachte die Verwendung der freien Variable *$n*, die während der Selektion an den Rollennamen gebunden wird. Man beachte weiterhin, dass dies nur für Assoziationsenden geschieht, die navigierbar sind. Alle in der Transformation nicht explizit erwähnten Modellelemente bleiben dabei „unberührt“, d.h. der Name der Komponente wird gleich sein wie der der Ausgangsklasse, und sie wird auch alle Attribute besitzen, die die Ausgangsklasse hat.

Ein Beispiel für ein Transformationsergebnis ist in Abb. 190 dargestellt.

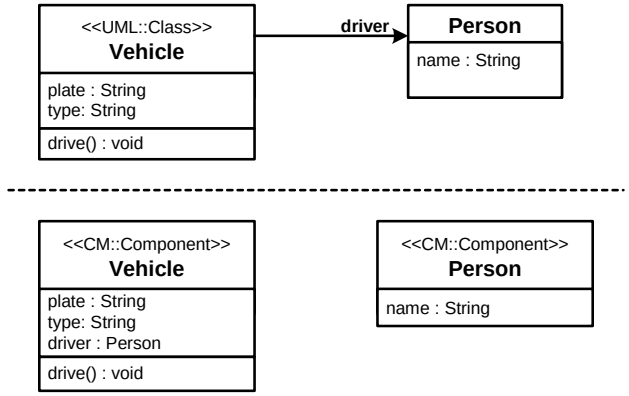


Abb. 190 *Beispiel für ein Transformationsergebnis*

Grafisch- deklarativ (2)

Eine andere Form der grafisch-deklarativen Spezifikation wird von dem Werkzeug Kase [KASE] verwendet. Die grafisch definierten Regeln bestehen aus drei Teilen, wie in Abb. 191 dargestellt:

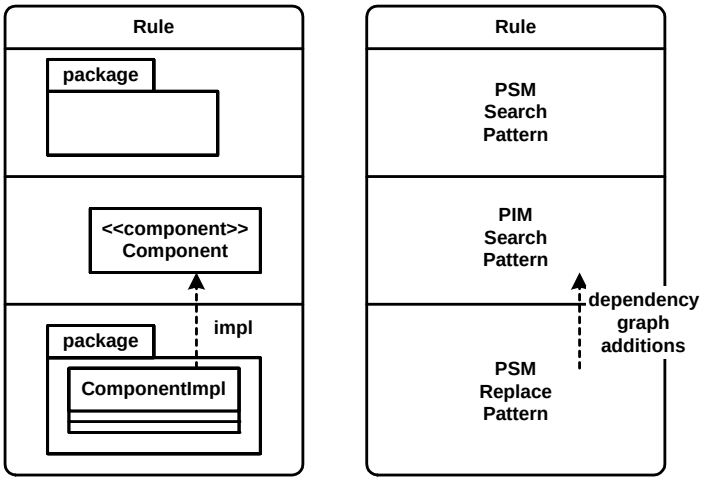


Abb. 191 *Grafisch-deklarative Modelltransformationen in KASE*

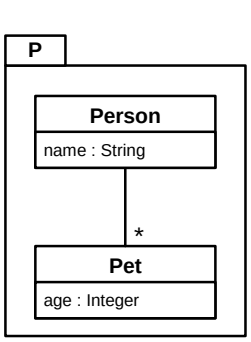
Zunächst sucht die Transformations-Engine nach dem angegebenen Muster im PSM, im Beispiel ein *Package*. Dann wird nach einer *Component* im PIM gesucht. Im dritten Teil der Regel wird definiert, wodurch das gefundene PSM-Element ersetzt werden soll. Hier wird eine Klasse zu dem *Package* hinzugefügt. Interessant sind auch die Abhängigkeits-Pfeile: Die Transformations-Engine erzeugt einen

Abhängigkeitsgraph vom PSM zum PIM. Dieser kann später abgefragt werden. Abhängigkeits-Pfeile in den Regeln definieren neu zu definierende Abhängigkeiten in diesem Graph. Im Beispiel wird also eine Abhängigkeit namens *impl* von der Implementierungsklasse im *Package* zu der *Component* im PIM erstellt.

Im Falle von komplexeren Transformationen muss die Reihenfolge der Transformationen festgelegt werden. Dazu werden Aktivitätsdiagramme verwendet.

Deklarativ- textuell

Im Folgenden wollen wir noch kurz den Vorschlag der QVT-Partner zur QVT-Ausschreibung betrachten [QVTP]. Diese Transformationen können sowohl grafisch als auch textuell dargestellt werden. Wir werden hier die textuelle Notation zeigen, die grafische ist ähnlich zu UMLX. Zunächst sei eine Notation eingeführt, mittels derer Klassen- und Objektdiagramme textuell dargestellt werden können. Das Beispiel in Abb. 192 ist dem Proposal entnommen. Auch diese textuelle Darstellung eines Klassendiagramms ist eigentlich die textuelle Repräsentation des entsprechenden Meta-Objektdiagrammes.




```
(Package) [
  name = 'P',
  contents = {
    (Class, Person) [
      name = 'Person',
      attributes = {
        (Attribute) [
          name = 'name',
          type = String],
        (Attribute) [
          name = 'pet',
          type = (SetType) [
          elementType = Pet]]}],
    (Class, Pet) [
       name = 'Pet',
       attributes = {
       (Attribute) [
         name = 'age',
         type = Integer]}]}]
```

Abb. 192 *Beispiel eines textuell repräsentierten Klassendiagramms*

Der Vorschlag der QVT-Partners setzt auf dieser textuellen Darstellung auf. Im Rahmen des Proposals werden *Relations* und *Mappings* (als Spezialisierung von *Transformations*) definiert. Relationen geben einfach nur die Beziehungen zwischen zwei Modellen an und sind damit bidirektional anwendbar, wohingegen *Mappings* eine „Produktionsregel“ des einen Modelles aus dem anderen darstellt. Da Relationen und Mappings eine fast identische Syntax verwenden, sehen wir uns zunächst

nur die Relationen an. Eine *Relation* wird nach folgendem Schema definiert:

```
relation R {
  domain { pattern-1 when condition-1 }
  domain { pattern-2 when condition-2 }
  when { condition }
}
```

Die beiden Domains geben die beiden Modelle (oder Modellausschnitte) an, die zueinander in Beziehung gesetzt werden sollen. Die *condition*-Klauseln dienen der Definition von Bedingungen, unter denen die Transformation gültig ist. Die folgende Transformation bildet eine Methode aus dem UML-Metamodell auf ein XML-Element ab.

```
relation Method_And_XML {
  domain { (UML.Method)[name = n, body = b] }
  domain {
    (XML.Element)[
      name = "Method",
      attributes =
        {(XML.Attribute)[name = "name", value = n]},
      contents = {b}
    ]
  }
}
```

Auch hier werden freie Variablen eingesetzt (hier: *n* und *b*). Diese müssen in den verschiedenen Domains gleiche Werte haben. Es wird also der Name der Methode aus dem UML-Modell zum Wert des XML-Attributs *name*; der *body* der Methode wird zum Inhalt des XML-Elementes.

23.2.10. Executable UML

Ein Begriff, den man oft im Zusammenhang mit MDA zu hören bekommt, ist *Executable UML*. Dabei handelt es sich nicht um einen formalen Standard, sondern um einen Sammelbegriff für verschiedene Bestrebungen. Sie alle haben zum Ziel, UML als vollwertige Programmiersprache zu etablieren. Dazu muss man sie allerdings von Ballast und Unklarheiten befreien – dadurch werden UML-Diagramme leichter ausführbar: Je kleiner der Umfang des Metamodells der Modellierungssprache (hier UML), desto einfacher ist es, dafür einen Compiler oder Interpreter zu implementieren.

Ein weiterer, notwendiger Bestandteil von Executable UML ist eine Action-Sprache (vgl. Abschnitt 23.2.6). Mit ihrer Hilfe können vollständige Implementierungen von Softwaresystemen definiert werden.

Wichtig ist dabei, zu verstehen, dass es sich im Gegensatz zum MDSD-Ansatz *nicht* um ein *domänenspezifisches* Profil der UML handelt[71]. Die Idee ist vielmehr, eine universelle UML-basierte Programmiersprache zu definieren. Diese kann, wie gesagt, interpretiert oder kompiliert werden.

Weitere Informationen über Executable UML sind in dem Buch von Steve Mellor [Mel02] oder in der Dokumentation von Kennedy Carters iUML [IUML] zu finden.

23.3. Herausforderungen für die MDA

Im Kontext von MDSD und der MDA im Besonderen sind in Zukunft noch einige technische Herausforderungen zu meistern. Viele der im Folgenden angesprochenen Aspekte sind zwar im Forschungsumfeld oder in Nischen bereits Realität. Bis diese Dinge in Form von marktreifen, praxistauglichen Tools auf dem Markt sind, wird aber noch eine ganze Weile vergehen. Die Tatsache, dass wir auf die betreffenden Probleme und Herausforderungen hinweisen, soll allerdings nicht als Kritik an der MDA verstanden werden. Es sind lediglich Hinweise, dass die *vollständige* Anwendung der MDA eben noch nicht ohne weiteres möglich ist. Als Vergleich braucht man sich nur vor Augen zu führen, wie lange es gedauert hat, bis wirklich praxistaugliche UML-Tools auf dem Markt waren[72].

23.3.1. Modell-zu-Modell-Transformationen

Modell-zu-Modell-Transformationen sind praktisch in nicht-trivialen Anwendungsszenarien noch nicht verfügbar. Es gilt dabei verschiedene Probleme zu lösen. Zum einen gibt es noch keine Einigkeit darüber, wie solche Transformationssprachen aussehen sollen. Es zeichnet sich eine Tendenz hin zu grafischen Sprachen ab. Idealerweise werden die Regeln

---

[71] Es sei denn, man betrachtet „Turing-berechenbare Funktionen“ als Domäne.

[72] Einige Leute sind der Ansicht, dass es diese Tools bis heute nicht gibt ☺

so definiert, dass sie sowohl textuell als auch grafisch dargestellt werden können[73].

Zum anderen sind die meisten Ansätze regelbasiert, deklarativ oder funktional. Entsprechende Programmiersprachen haben sich in der Vergangenheit nicht im Massenmarkt durchgesetzt. In dieser Hinsicht darf man also gespannt sein.

Zusätzlich zu der Frage, wie diese Sprachen im Idealfall aussehen sollen, muss im Rahmen der MDA auch noch die Standardisierung erfolgen. Die QVT-Entwicklung bleibt abzuwarten.

23.3.2. Modell-Management

Wenn große Teile einer Anwendung grafisch spezifiziert werden, so muss das Modellierungswerkzeug den Überblick über die potenziell zahlreichen Teil-Modelle gewähren. Aktuelle UML-Werkzeuge leisten dies nur begrenzt. Weitere Probleme, die sich ergeben, sind im Zusammenhang mit der Arbeit im Team zu sehen sowie generell in der Versionierung von (Teil-)Modellen. Zwar haben wir in diesem Buch effektive Techniken aufgezeigt (vgl. z.B. <Ref Vers>), wie man solche Schwächen umgehen kann – besser wäre jedoch eine entsprechend standardisierte Werkzeugunterstützung.

23.3.3. Evolution von DSLs

Metamodellierung und die Definition konkreter Syntaxformen entspricht letztendlich der Konstruktion von (Programmier-)Sprachen. Dies erfordert andere Qualifikationen als die *Verwendung* von Programmiersprachen. Es ist am ehesten noch mit der Erstellung von Frameworks zu vergleichen. Hier wären (standardisierte) Werkzeuge wünschenswert, welche die Evolution von DSLs unterstützen.

23.3.4. Adäquate IDEs

Heutige IDEs bieten neben Editoren noch eine ganze Reihe weiterer Features: Code-Completion, Refactoring und vor allem komfortable Debugger.

---

[73] Ob eine Notation grafisch oder textuell ist, ist letztlich nur eine Ergonomie-Frage. Viel wichtiger ist die Ausdruckskraft der Sprache.

Praktisch nutzbare MDA-Tools sollten das Debuggen auf der Ebene der DSL erlauben. Ggf. sollte ein Drilldown in die nächstniedrigeren Modellebenen (PIM->PSM->…->Code) möglich sein. Auch Refactoring auf DSL-Ebene wäre wünschenswert.

Auch sollte sich das GUI der entsprechenden Tools an das Metamodell anpassen. Für Metamodellkonzepte sollte ein Auslöser vorhanden sein, welcher es erlaubt, eine Instanz des Konzeptes in das Modell einzufügen (einige Metamodellierungs-IDEs können dies bereits, siehe z.B. [MC04])

Natürlich sind diese Anforderungen für Werkzeuge, die mit MOF-basierten Metamodellen und DSLs arbeiten können, relativ schwer zu realisieren. Es wird vermutlich noch eine ganze Weile dauern, bis dies allein für UML-basierte Werkzeuge Wirklichkeit ist.

23.3.5.Interoperabilität

Last but not least gibt es noch den Aspekt der Interoperabilität von MDA-Werkzeugen. Modellgetriebene Entwicklung an sich benötigt diese Eigenschaft nicht zwangsläufig. Für MDA ist sie von zentraler Bedeutung, weil sonst viele der Ziele nicht realisiert werden können. Hier bleibt uns nur die Zukunft abzuwarten oder aktiv an der Standardisierung mitzuwirken.

24. Index

4

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z