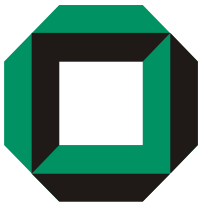
Universität Karlsruhe – Fakultät für Informatik – Bibliothek – Postfach 6980 – 76128 Karlsruhe

Transformationen in der modellgetriebenen Software-Entwicklung

Herausgeber:
Steffen Becker, Thomas Goldschmidt, Henning Groenda, Jens Happe, Henning Jacobs, Christian Janz, Konrad Jünemann, Benjamin Klatt, Christopher Köker, Heiko Koziolek, Klaus Krogmann, Michael Kuperberg, Christoph Rathfelder, Ralf Reussner, Beyhan Veliev

Interner Bericht 2007-9



Universität Karlsruhe
Fakultät für Informatik

ISSN 1432 – 7864

Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Transformationen in der modellgetriebenen Software-Entwicklung

Seminar im Sommersemester 2007

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl für Software-Entwurf und -Qualität
Prof. Dr. Reussner

`http://sdq.ipd.uni-karlsruhe.de`

Steffen Becker, Thomas Goldschmidt, Henning Groenda, Jens Happe, Henning Jacobs, Christian Janz, Konrad Jünemann, Benjamin Klatt, Christopher Köker, Heiko Koziolek, Klaus Krogmann, Michael Kuperberg, Christoph Rathfelder, Ralf Reussner, Beyhan Veliev

Vorwort

Modellgetriebene Software-Entwicklung ist in den letzten Jahren insbesondere unter Schlagworten wie MDA oder MDD zu einem Thema von allgemeinem Interesse für die Software-Branche geworden. Dabei ist ein Trend weg von der codezentrierten Software-Entwicklung hin zum (Architektur-) Modell im Mittelpunkt der Software-Entwicklung festzustellen. Modellgetriebene Software-Entwicklung verspricht eine stetige und automatisierte Synchronisation von Software-Modellen verschiedenster Ebenen. Dies verspricht eine Verkürzung von Entwicklungszyklen und mehr Produktivität. Primär wird nicht mehr reiner Quellcode entwickelt, sondern Modelle und Transformationen übernehmen als eine höhere Abstraktionsebene die Rolle der Entwicklungssprache für Software-Entwickler.

Software-Architekturen lassem sich durch Modell beschreiben. Sie sind weder auf eine Beschreibungssprache noch auf eine bestimmte Domänen beschränkt. Im Zuge der Bemühungen modellgetriebener Entwicklung lassen sich hier Entwicklungen hin zu standardisierten Beschreibungssprachen wie UML aber auch die Einführung von domänen-spezifischen Sprachen (DSL) erkennen. Auf diese formalisierten Modelle lassen sich schließlich Transformationen anwenden. Diese können entweder zu einem weiteren Modell („Model-to-Model“) oder einer textuellen Repräsentation („Model-to-Text“) erfolgen. Transformationen kapseln dabei wiederholt anwendbares Entwurfs-Wissen („Muster“) in parametrisierten Schablonen. Für die Definition der Transformationen können Sprachen wie beispielsweise QVT verwendet werden. Mit AndoMDA und openArchitectureWare existieren Werkzeuge, welche die Entwickler bei der Ausführung von Transformationen unterstützen.

Das Seminar wurde wie eine wissenschaftliche Konferenz organisiert: Die Einreichungen wurden in einem peer-to-peer-Verfahren begutachtet (vor der Begutachtung durch den Betreuer) und in verschiedenen Sessions wurden die Artikel an einem Konferenztag präsentiert. Der beste Beitrag wurden durch einen best paper award ausgezeichnet. Dieser ging an Benjamin Klatt für seine Arbeit „Xpand: A Closer Look at the model2text Transformation Language“.

Gliederung

Die Seminarthemen dieses Seminars spiegeln die Frage- und Problemstellungen wider, die sich bei der Verwendung von Transformationen in der modellgetriebene Software-Entwicklung ergeben. Dieser technische Bericht gliedert sich dabei wie folgt. Zunächst werden grundlegende Themen der modellgetriebenen Software-Entwicklung wie die Transformationssprache QVT und die Verwendung von Entwurfsmustern im Zusammenhang mit Transformationen behandelt. Im Anschluss daran werden mit AndroMDA und openArchitectureWare zwei verbreitete Werkzeuge zur Modelltransformation vorgestellt. Im letzten Beitrag steht dann die Validierung und Verifikation von Transformationen im Mittelpunkt.

Dank

Wir möchten uns an dieser Stelle bei allen Teilnehmern des Seminars für ihre engagierte Mitarbeit sehr herzlich bedanken. Ein mehrstufiger Begutachtungs-Prozess bestehend aus „peer-to-peer-Reviews“ sowie Gutachten durch die Betreuer ermöglichte die Auswahl qualitativ hochwertiger Artikel. Insgesamt wurden sechs Ausarbeitungen für diesen technischen Bericht angenommen.

Juli 2007

Steffen Becker
Thomas Goldschmidt
Henning Groenda
Jens Happe
Heiko Koziolek
Klaus Krogmann
Michael Kupperberg
Christoph Rathfelder
Ralf Reussner

Inhaltsverzeichnis

Transformationen in der modellgetriebenen Software-Entwicklung

Meta Object Facility (MOF) Query/View/Transformation

Henning Jacobs
henning@jacobs1.de

Betreuer: Heiko Koziolek

Zusammenfassung Diese Ausarbeitung gibt einen Überblick über den *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)* Standard der *Object Management Group (OMG)*. Der QVT-Standard spezifiziert erstmals wie Modelltransformationen innerhalb der MDA-Strategie durchzuführen sind. Eine Fallstudie mit dem Werkzeug *SmartQVT* zeigt die konkrete Anwendung des Standards *QVT-Operational*. Anhand dieser Fallstudie und einer Bewertung des Autors versucht die Arbeit den QVT-Standard innerhalb des modellgetriebenen Entwicklungsprozesses einzuordnen.

Key words: Queries Views Transformations, QVT, MOF, MDA

1 Einleitung

Die Komplexität heutiger Software ist um ein Vielfaches größer als noch vor 5–10 Jahren und nimmt weiter zu. Unter der zunehmenden Komplexität werden Softwareprojekte immer schwerer verständlich und es werden verstärkt Abstraktionen vom Quellcode (Modelle) gebraucht, um Softwareprojekte zu bewältigen.

Ein Ansatz ist die konsequente Verwendung von Modellen als Grundlage der Softwareentwicklung. In den folgenden zwei Abschnitten werden kurz die Grundlagen der modellgetriebenen Softwareentwicklung und die dazugehörigen Transformationen vorgestellt. Danach wird die Entstehung und der Aufbau des Transformationsstandards QVT erläutert, um dann den Einsatz anhand einer Fallstudie zu zeigen. Zum Abschluß dieser Arbeit bewertet der Autor den QVT-Standard und erfasst seine Bedeutung in der modellgetriebenen Softwareentwicklung.

1.1 Modellgetriebene Softwareentwicklung

Die modellgetriebene Softwareentwicklung (*Model Driven [Software] Development [MD(S)D]*) zentriert sich auf die Benutzung von Modellen [1]. Modelle sind Systemabstraktionen, die es Entwicklern und anderen Interessenten erlauben, sich effektiv mit den wesentlichen Belangen des Systems zu befassen [2].

Dabei ist ein Ziel der modellgetriebenen Softwareentwicklung die weitgehende Automatisierung im Softwareerstellungsprozess durch die automatische Transformation von Modellen zu Modellen und von Modellen zum ausführbaren Code.

Dadurch werden fehlerträchtige Routineaufgaben in der Implementierungsphase vermieden und der Entwickler kann sich auf den Entwurf der fachlichen Anwendung konzentrieren [3].

Das prominenteste Beispiel für die Umsetzung von MDD ist die *Model Driven Architecture (MDA)* der Object Management Group (OMG) [4] [2].

Die Model Driven Architecture (MDA) MDA ist eine Strategie der Object Management Group (OMG) für die modellgetriebene Softwareentwicklung.

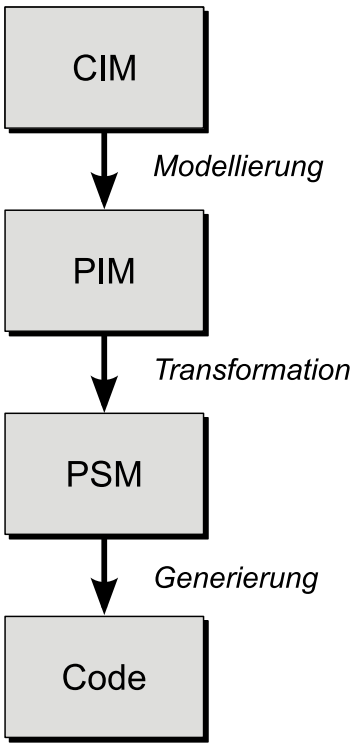


Abbildung 1. Die verschiedenen Modelle im MDA-Prozess und ihre Transformation ineinander

MDA definiert verschiedene Stufen für das Gesamtmodell einer Anwendung (Abbildung 1) [5]:

Computational Independent Model (CIM): Das *Computational Independent Model* ist eine umgangssprachliche Beschreibung der zu modellierenden Geschäftsprozesse in der Realwelt.

Platform Independent Model (PIM): Das *Platform Independent Model* ist eine weitgehende Abstraktion des Anwendungsmodells von einer technischen Plattform. Das PIM wird üblicherweise in einer fachspezifischen Sprache (Domänensprache) modelliert und dient als Spezifikation für die Implementierung von Geschäftsprozessen auf einer bestimmten Plattform.

Platform Specific Model (PSM): Das *Platform Specific Model* ist eine Abstraktion des ausführbaren Codes und wird in Begriffen der technischen Plattform (z.B. J2EE) beschrieben. Das PSM ist eine Implementierung eines PIM auf einer technischen Plattform.

Codemodell: Das PSM wird schließlich durch plattformspezifischen Code (z.B. Java) auf der Zielplattform implementiert.

Üblicherweise wird im Laufe der Anwendungsentwicklung von der abstrakteren Ebene zur konkreteren Ebene (top-down) transformiert. D.h. es wird CIM in PIM, PIM in PSM und schließlich PSM in ausführbaren Code überführt [3].

Neben den verschiedenen, unterschiedlich abstrakten Modellen definiert MDA Standards für die Überführung von einem Modell in ein anderes. Dabei lässt sich zwischen Modell-zu-Modell-Transformationen (PIM nach PSM) und Modell-zu-Code-Transformationen (PSM nach Code) unterscheiden. Die OMG definiert MOF QVT als Standard für Modell-zu-Modell-Transformationen innerhalb der MDA-Strategie.

1.2 Modell-zu-Modell Transformationen

In Abbildung 2 sind die Basiskonzepte einer jeden Modell-zu-Modell-Transformation dargestellt. Eine Modelltransformation überführt immer ein Quellmodell (*source model*) in ein Zielmodell (*target model*). Dabei sind Quell- und Zielmodell konform zu je einem Metamodell, welches auch für Quell- und Zielmodell das selbe sein kann. Die eigentlichen Transformationsregeln (*transformation definition*) werden durch die Transformationssprache (*transformation engine*) ausgeführt.

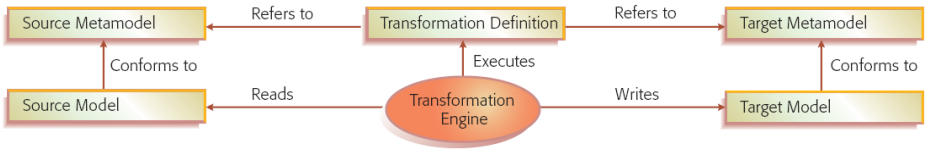


Abbildung 2. Basiskonzepte der Modelltransformation [2]

Die Begriffe Modell und Metamodell sind kontextabhängig, so kann ein Modell ein Metamodell sein und ein Metamodell wiederum eine Instanz eines anderen Modells. Deshalb können Modell-zu-Modell Transformationen auf sehr unterschiedlichen Ebenen erfolgen. So können Metamodelle (z.B. ECORE), Modelle (z.B. UML, relationales Modell) und Modellinstanzen (z.B. UML-Objekte) transformiert werden.

Anhand dieser Möglichkeiten der Modelltransformation wird bereits klar, dass eine Transformationssprache flexibel für unterschiedliche Einsatzzwecke geeignet sein muss.

Nach [1] sollte eine Modell-zu-Modell-Transformationssprache folgende Anforderungen erfüllen, welche zum großen Teil Empfehlungen aus [6] entsprechen:

- Eine Transformationssprache sollte den Transformationsvorgang aufzeichnen (*transformation trace*), damit Transformationsregeln nachvollziehen können, was andere Teile der Transformation bereits berechnet haben.
- Änderungen am Quellmodell sollten an das Zielmodell propagiert werden (*change propagation*). Eine Transformationsprache sollte Änderungen an Teilen des Quellmodells auf entsprechende Teile des Zielmodells abbilden und Elemente nicht einfach sukzessive im Zielmodell hinzufügen.
- Häufig existieren beide Modelle bereits vor der gewünschten Transformation und die Transformationssprache muss initiale Beziehungen zwischen den Modellen aufbauen. Dies unterscheidet sich vom letzten Punkt, denn im Falle von *change propagation* kann die Sprache auf die Ergebnisse und Aufzeichnungen bereits durchgeführter Transformationen zurückgreifen.
- Die Transformationssprache sollte Änderungen inkrementell übernehmen (*incremental update*). Ähnlich eines Buildvorgangs (vgl. „make“) erwartet der Benutzer bei Änderungen eine effiziente Transformation nur der geänderten Teile.
- In der Praxis ist eine Modelltransformation in der Regel nie vollständig isomorph, da Modelle immer nur (unterschiedliche) Ausschnitte eines Systems abbilden. So fehlen dem Platform Independent Model (PIM) in der Regel Informationen für einige Entwurfsentscheidungen im Platform Specific Model (PSM) und man möchte deshalb kontrolliert manuelle Änderungen am Zielmodell (dem PSM) vornehmen. Manuelle Änderungen sollten dabei nicht durch erneute Transformation verloren gehen — die Transformationssprache sollte also manuelle Änderungen erhalten (*retainment policy*).
- Nicht-triviale Transformationen werden nicht auf einmal ausgeführt sondern bestehen aus kleineren Transformationen auf Teilmodellen. Beispielsweise gäbe es für eine Transformation von UML eine Transformation für Packages, eine für Klassen und eine für Attribute. Eine Transformationssprache sollte solche mehrstufige Transformationen unterstützen (*M x N transformations*).
- Bidirektionale Transformationen sollten möglich sein (*bidirectional transformations*). Dies lässt sich nur durch eine deklarative Transformationssprache, welche bidirektionale Beziehungen zwischen Modellelementen definiert, erreichen.

Die obigen Anforderungen liefern einige Gründe, warum das Schreiben von Modelltransformationen in einer universellen Programmiersprache wie Java in realistischen Szenarien nicht ausreichend ist. Bei einer universellen Programmiersprache müsste der Transformationsentwickler zur Erfüllung der genannten Anforderungen viele zusätzliche Daten über die Transformation manuell verwalten, z.B. den *transformation trace*, Änderungsinformationen für *incremental*

update, Beziehungen zwischen Modellelementen für Bidirektionalität, usw. Eine spezielle Transformationssprache, die von diesen Aufgaben abstrahiert, ist im Allgemeinen also wünschenswert.

Um MOF-konforme Modell-zu-Modell-Transformationen durchzuführen, definierte die OMG die Transformationssprache QVT (*Queries, Views and Transformations*). Der QVT-Standard wurde mit dem Ziel entworfen, eine QVT-Implementierung bei der Erfüllung obiger Anforderungen entweder zu unterstützen oder zumindest nicht zu behindern [1]. Daraus ergaben sich drei domänenspezifische QVT-Sprachen.

2 Der OMG QVT Standard

In diesem Abschnitt liegt der Fokus auf Modell-zu-Modell-Transformationen mit dem *Query / View / Transformations* (QVT) Standard der OMG. Da der QVT-Standard sehr umfangreich und komplex geworden ist, wird hier nur ein Überblick über Entstehung, Architektur und Fähigkeiten gegeben.

Der vorliegende QVT-Standard referenziert und verwendet die folgenden zwei OMG-Spezifikationen [7]:

- Meta Object Facility (MOF) Core Specification, version 2.0, Januar 2006
- Object Constraint Language (OCL) Specification, version 2.0, Mai 2006 [8]

2.1 Entstehung

Der MDA Guide der OMG spricht oft über Transformationen zwischen Modellen unterschiedlicher Abstraktionsebenen [1]. Wie bereits in Abschnitt 1.1 ausgeführt, wird in der MDA zwischen dem Platform Independent Model (PIM) und dem Platform Specific Model (PSM) transformiert, bevor aus dem Letzteren Code generiert wird. Der MDA Guide lässt jedoch offen wie diese Modelltransformation geschehen soll.

Im April 2002 veröffentlichte die OMG ein *Request for Proposal* (RFP) für Modell-zu-Modell-Transformationen (genannt Query/Views/Transformations) [9], welche schließlich zur *Final Adopted QVT Specification* [7] im November 2005 führte. Die relative lange Standardisierungszeit (2002 – 2005) kann teilweise durch die immanente Komplexität des Problems erklärt werden [1]. Obwohl viele Leute gute Ideen für die Implementierung von Modell-zu-Modell-Transformationen in Programmiersprachen wie Java hatten, wurde schnell klar, dass realistische Modell-zu-Modell-Szenarien ausgefeiltere Techniken benötigten. Das QVT-RFP fragte explizit nach Vorschlägen für solche Techniken, auch wenn diese Anforderungen noch zum großen Teil nicht verstanden wurden [1].

Ein weiteres Problem war die fehlende Erfahrung mit Modell-zu-Modell-Transformationen, welches keinen guten Ausgangspunkt für eine Standardisierung darstellte. Das Problem verschlimmerte sich durch den Umstand, dass acht verschiedene Gruppen Lösungsvorschläge für das QVT RFP einreichten [1]. Die meisten dieser Vorschläge waren so verschieden, dass es keine klare Basis für eine mögliche Konsolidierung gab. Als Ergebnis spezifiziert QVT heute drei verschiedene Sprachen, welche nur lose miteinander verbunden sind [1].

2.2 Architektur

Die QVT Spezifikation hat eine hybride Natur mit einem deklarativen und einem imperativen Sprachteil. Der deklarative Teil hat wiederum zwei Ebenen: Einen benutzerfreundlichen *Relations*- und einen darunterliegenden *Core*-Teil. (Abbildung 3 aus [7])

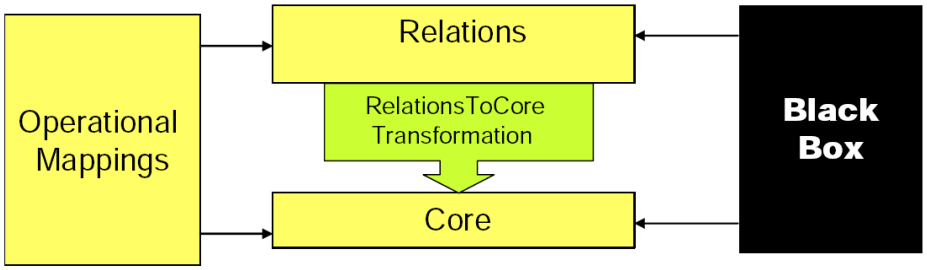


Abbildung 3. QVT-Architektur und Beziehung zwischen den Metamodellen

Relations Language (QVT-Relations) Die Relations Language ist eine deklarative Sprache zur Spezifikation von Relationen zwischen Modellen. Diese Relationen können auch bidirektional sein.

Operational Mapping (QVT-Operational) Das Operational Mapping von QVT ist eine imperative Sprache zur Modellierung von Transformationsabläufen zwischen Modellen. QVT-Operational ist an bestehende imperative Programmiersprachen angelehnt und benutzt eine prozedurale Erweiterung von OCL. Möchte man bidirektionale Transformationen mit QVT-Operational durchführen, muss man jede Transformationsrichtung separat implementieren.

Core Language (QVT-Core) QVT-Core liegt der Relations-Sprache zugrunde und bietet die gleiche Funktionalität auf einer anderen Abstraktionsebene. Theoretisch kann QVT-Relations vollständig auf QVT-Core abgebildet werden (Analogie zu Java: Sprache und Bytecode) [1]. Für den QVT-Anwender ist die Core-Sprache nicht weiter relevant, da diese zwar sehr einfach aufgebaut ist, aber für praktische Transformationen nicht die nötige Unterstützung bietet.

Black Box Implementation QVT ist vollständig erweiterbar durch Laden von externen Funktionen, welche bspw. in Java oder .NET definiert wurden, aber auch durch Einbinden bereits bestehender QVT-Transformationen als „Plug-Ins“. Diese externen Funktionen können dann von QVT-Operational aus als „Black Box“ verwendet werden.

Die Designer der QVT-Spezifikation gehen davon aus, dass die Relations Language vom Entwickler benutzt werden soll, die Operational Mapping Language nur in Ausnahmefällen. [10]

2.3 Einsatzszenarien

QVT-Core (und damit die darauf basierende QVT-Relations Sprache) unterstützen folgende Szenarien:

- Unidirektionale Transformationen
- Bidirektionale Transformationen
- Aufbau von Relationen zwischen bestehenden Modellen
- Inkrementelle Updates (in beliebige Richtung) wenn ein Modell geändert wurde
- Die Möglichkeit Objekte zu erstellen/zu löschen und explizit Modifikation von Objekten zu verbieten

Bei Einsatz von QVT-Operational werden die obigen Möglichkeiten auf unidirektionale Transformationen eingeschränkt.

2.4 QVT-Operational

Aus Gründen der einfacheren Darstellung und der Werkzeugunterstützung wird im weiteren (insbesondere der Fallstudie in Kapitel 3) nur die Sprache QVT-Operational näher betrachtet.

QVT-Operational ist modernen objektorientierten Programmiersprachen im Aufbau sehr ähnlich. Eine sogenannte *Operational-Transformation (OT)* repräsentiert die Definition einer unidirektionalen Transformation in einer imperativen Sprache. Die Operational-Transformation definiert eine Signatur mit dem Ein- und Ausgabemodell als Parameter und einen Einstiegspunkt *main* für ihre Ausführung. Ähnlich einer Klasse ist die Operational-Transformation eine instanzierbare Entität mit Eigenschaften und Methoden.

Folgendes Beispiel definiert das Gerüst einer Transformation von UML-Klassendiagrammen nach RDBMS-Tabellen (Tabellen eines relationalen Datenbanksystems):

```
1 transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS) {
2     // Einstiegspunkt für die Transformation
3     main() {
4         // Aufruf eines Mappings
5         // für alle Package-Instanzen
6         uml.objectsOfType(Package)->map packageToSchema();
7     }
8 ..
9 }
```

Beispielcode von QVT-Operational für das Ausrollen von Klassenhierarchien findet sich in Abschnitt 3.

2.5 Implementierungen

Zum heutigen Zeitpunkt gibt es noch keine Implementierung einer Transformationssprache, die 100% QVT-kompatibel ist ([10]).

Dafür existieren eine Reihe von Werkzeugen, die Teile des QVT-Standards implementieren oder zu QVT sehr ähnliche Konzepte verwenden [11]:

- **M2M**: *M2M*[1] ist das Rahmenwerk der Eclipse Foundation für Modelltransformationen. M2M besitzt drei Komponenten:
  - ATL (siehe unten)
  - Procedural QVT (geplante Implementierung von QVT-Operational)
  - Declarative QVT (geplante Implementierung von QVT-Core und QVT-Relations)

  Bisher existieren für die QVT-Bestandteile jedoch nur *Proposals* und Zusagen einiger Entwickler die Komponenten voranzubringen.
- **Borland Together**: Das Borland Produkt *Borland Together Architect 2006* ist eine kommerzielle Eclipse-Erweiterung und ist teilweise QVT-kompatibel [12]. Leider wird auf der Produktseite[2] und im Datenblatt nicht näher auf die QVT-Implementierung eingegangen.
- **ATL**: Die *ATLAS Transformation Language (ATL)* ist keine eigentliche QVT-Implementierung, sondern verwendet zu QVT-Relations sehr ähnliche Konzepte. ATL ist ein Open Source Produkt des Projektes *Generative Model Transformer (GMT)* der Eclipse Foundation und wird wegen seiner praxisrelevanten Lösungsansätze auch „das QVT von heute“ [10] genannt.
- **ModelMorf**: *ModelMorf*[3] ist eine proprietäre Transformations-Engine von Tata Consultancy und implementiert teilweise QVT-Relational.
- **Tefkat** [13]: Open Source Implementierung von *Tefkat*, einer QVT-ähnlichen deklarativen Transformationssprache, welche von der Syntax stark an SQL angelehnt ist. Tefkat implementiert eine verbesserte Version eines Beitrags der Firmen DSTC und IBM zur QVT RFP. Tefkat ist als Eclipse-Plugin realisiert und basiert auf EMF.
- **MTF**: Das *Model Transformation Framework (MTF)*[4] von IBM AlphaWorks ist ein quelloffenes Rahmenwerk für Modeltransformationen und ist von QVT beeinflusst, aber die aktuelle Version ist laut FAQ der Projektwebsite „kein Versuch eine aktuelle QVT-Spezifikation umzusetzen“.
- **SmartQVT** [14]: Das Open Source Eclipse-Plug-In *SmartQVT* der France Telecom implementiert QVT-Operational (siehe Kapitel 3).
- **medini transformation engine**: Implementierung von QVT-Relations basierend auf EMF von der Berliner Firma ikv++ technologies ag. Laut Website[5] ist *medini* die weltweit erste Implementierung des QVT-Standards der OMG.

[1] `http://www.eclipse.org/m2m/`
[2] `http://www.borland.com/us/products/together/index.html`
[3] `http://www.tcs-trddc.com/ModelMorf/index.htm`
[4] `http://www.alphaworks.ibm.com/tech/mtf/`
[5] `http://www.ikv.de/`

– **OptimalJ**: Laut [11] ist die QVT Core Language im Produkt *OptimalJ*[6] von Compuware implementiert. Leider schweigt sich auch hier die Produktwebsite über QVT-Kompatibilität aus.

Man sollte sich bei der Erstellung einer Liste von QVT-kompatiblen Implementierungen bewusst sein, dass die OMG nicht sehr strikt mit Standardkompatibilität umgeht. Die OMG hat keinen offiziellen Weg die Kompatibilitätsstufe eines Werkzeugs zu verifizieren [1]. Im Falle von QVT wird dies zwangsweise zu einer Vielzahl von Herstellern führen, die QVT-Unterstützung angeben ohne wirklich standardkompatibel zu sein.

Außerdem ist zu beachten was QVT-Kompatibilität bedeutet: Nach der Spezifikation [7] muss ein Werkzeug angeben, welches der drei QVT-Sprachen es unterstützt. Für jede unterstützte Sprache ist anzugeben, ob das Werkzeug den Import bzw. Export von Transformationen entweder in der abstrakten Syntax oder der konkreten Syntax erlaubt. Weiterhin muss separat angegeben werden, ob das Werkzeug *Black Box Implementation* erlaubt. Dies ergibt insgesamt eine große Menge von Möglichkeiten QVT-kompatibel zu sein.

Da kein bestehendes Werkzeug (und insbesondere keine freie Implementierung) QVT-Relations *und* QVT-Operational unterstützt, wird in der Fallstudie im nächsten Abschnitt nur eine der beiden Sprachen, nämlich QVT-Operational, betrachtet.

3 Fallstudie: Ausrollen einer Klassenhierarchie

Im Folgenden wird beispielhaft eine Modelltransformation mit dem Operational Mapping von QVT durchgeführt. Die Fallstudie basiert auf einer Fallstudie der ikv++ technologies ag ([15]). Als Werkzeug wird hierfür SmartQVT für Eclipse [14] eingesetzt. SmartQVT ist eine Implementierung von QVT-Operational durch die France Telecom. Das Werkzeug steht als quelloffenes Eclipse-Plug-In bereit und setzt auf dem Eclipse Modelling Framework (EMF) auf. Für die Installation benötigt man neben Eclipse und den EMF-Plugins auch eine Installation der Skriptsprache Python[7], denn der Parser für die QVT-Operational Sprache ist in Python geschrieben.

3.1 Ausrollen von UML-Klassenhierarchien

Ein gegebenes UML-Modell wird in ein anderes UML-Modell transformiert, welches nur die Blattklassen (Klassen, von denen nicht mehr abgeleitet wird) enthält. Diese Methode könnte z.B. ein Zwischenschritt für die Transformation nach einer Zielplattform ohne Vererbung (z.B. die Sprache C) sein.

Regeln:

– Es werden nur die Blattklassen aus dem Quellmodell transformiert.

[6] `http://www.compuware.com/products/optimalj/`
[7] `http://www.python.org`

– Die geerbten Attribute und Assoziationen werden übernommen.
– Attribute mit dem gleichen Namen überschreiben geerbte Attribute.
– Primitive Typen werden kopiert.

Das Metamodell für Ein- und Ausgabe der Transformation ist *SimpleUML* (Abbildung 4), ein einfaches Metamodell für UML-Klassendiagramme. In Abbildung 5 und 6 wird Eingabe und Ausgabe eines Ausrollens einer Klassenhierarchie gezeigt.

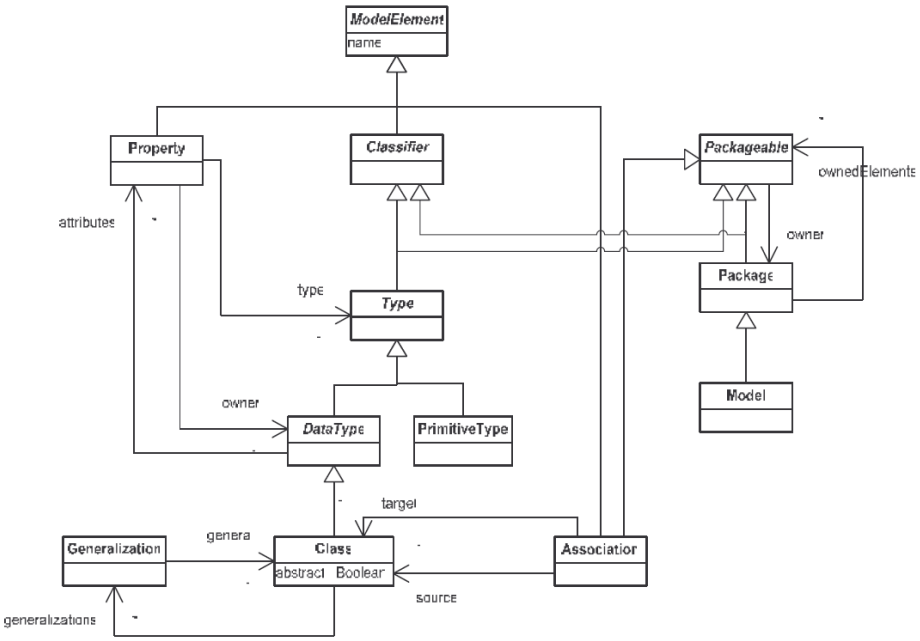


Abbildung 4. Quell- und Zielmetamodell: SimpleUML [15]

3.2 SmartQVT

Die Eingabemodelle müssen als Ecore-Modelle vorliegen (Abbildung 7).
SmartQVT bietet einen einfachen Editor mit Syntaxhervorhebung zum Editieren von QVT-Operational Quellcode (in SmartQVT sind dies *.qvt Dateien). SmartQVT parst die QVT-Datei und erstellt daraus einen abstrakten Syntaxbaum (*Abstract Syntax Tree* [AST]). Aus dem abstrakten Syntaxbaum wird dann ein neues Java-Projekt mit ausführbarem Java-Quellcode generiert, welches nach dem Kompilieren die gewünschte Modelltransformation ausführt. Das Ausführen des generierten Java-Codes transformiert Modell A (ausgedrückt in Metamodell *Ecore-A*) nach Modell B (ausgedrückt in Metamodell *Ecore-B*) (Abbildung 7).

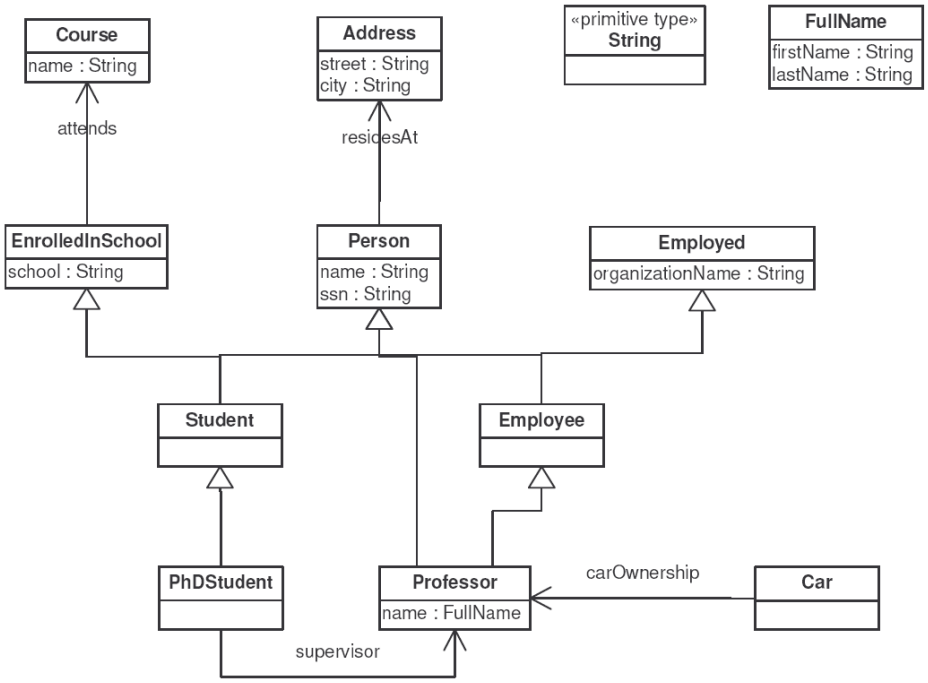


Abbildung 5. Beispiel für ein Quellmodell [15]

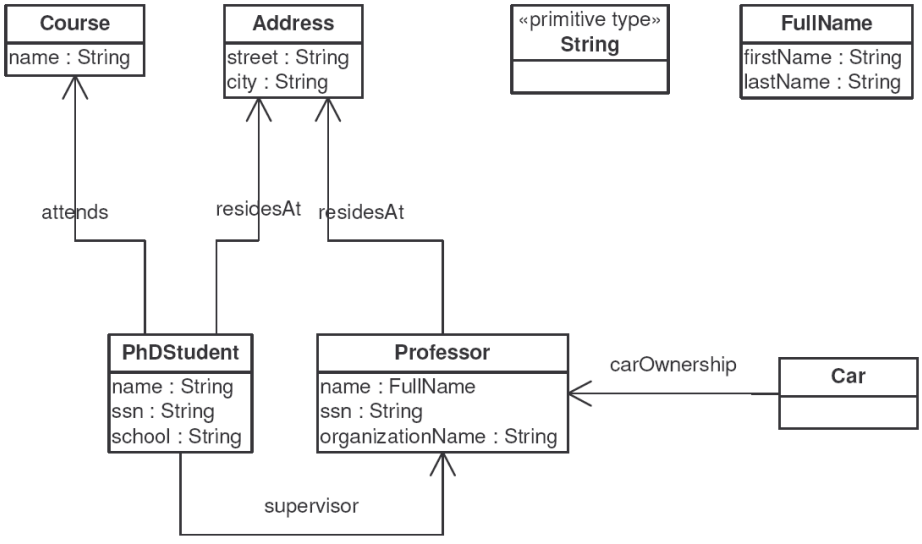


Abbildung 6. Beispiel für ein Zielmodell (Ergebnis von Abb. 5) [15]

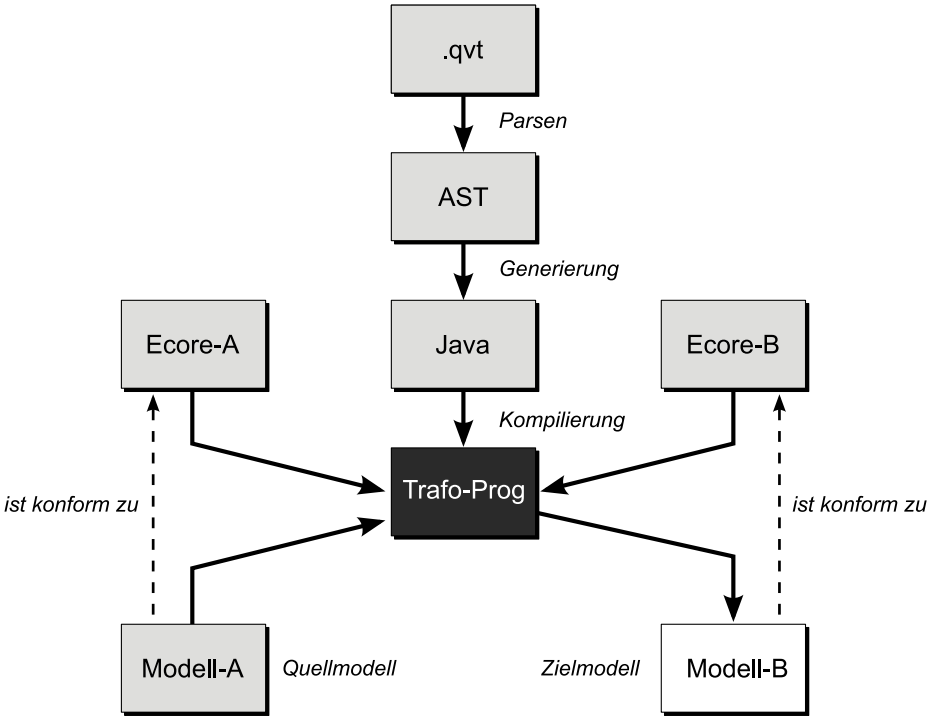


Abbildung 7. Der Transformationsprozess in SmartQVT

3.3 QVT-Operational zum Ausrollen von Klassenhierarchien

Aufbau der QVT-Operational Datei:

```
transformation SimpleUML2FlattenSimpleUML(
    in source : SimpleUML, out target : SimpleUML)
{
    // Einstiegspunkt
    main ()
    {
        // Blattklassen transformieren
        source.objectsOfType(Class)->
            map leafClass2Class(source);
    }
}
..
// Helper
..
// Mappings
..
```

Das Schlüsselwort `transformation` definiert die Signatur der Modelltransformation und benennt die Ein- und Ausgabeparameter. Die Methode *main* ist der Einstiegspunkt für die Ausführung der Transformation. Die Haupttransformationslogik besteht aus seiteneffektfreien *Helper*-Queries (Funktionen) und *Mappings*, welche Objekte aus dem Eingabemodell in Objekte des Ausgabemodells transformieren. In *main* wird für alle Objekte vom Typ *Class* das Mapping *leafClass2Class* ausgeführt. Ein Mapping wird stets über die Operation `map` einer OCL-Collection aufgerufen, welche das Mapping für jedes Element der Collection durchführt.

Im folgenden Listing ist das Mapping für die Regel „Transformiere alle Blattklassen“ dargestellt. Die *when*-Klausel spezifiziert Vorbedingungen für das Mapping. Das Mapping wird nur auf Elementen des Eingabemodells aufgeführt, die die *when*-Klausel erfüllen.

```
mapping Class::leafClass2Class(in model : Model) : Class
when
{
    // Vorbedingung für dieses Mapping:
    // Es gibt keine andere Klasse die
    // von dieser Klasse erbt
    not model.allInstances(Generalization)->
        exists(g | g.general = self)
}
{
    // Klassenname übernehmen
    name := self.name;
    // Abstrakte Klasse bleibt abstrakt
```

```
14      abstract := self.abstract;
15      // Geerbte und klasseneigene Attribute zusammenführen
16      attributes := self.derivedAttributes()->
17          map property2property(self)->asOrderedSet();
18  }
```

Das Mapping operiert auf Instanzen des Typs *Class* (Angabe vor „::“ in der Signatur [Zeile 1]) und hat als Ergebnis wieder Instanzen des Typs *Class* (Letztes Wort Zeile 1). Als zusätzlichen Parameter *model* benötigt dieses Mapping hier das Quellmodell, um in der *when*-Klausel auf alle Modellinstanzen zugreifen zu können (OCL-Ausdruck „model.allInstances“ in Zeile 7) und damit sicherzustellen, dass keine Klasse existiert, die von dieser Klasse erbt.

In Zeilen 11 bis 17 steht das eigentliche Mapping. Hier werden der Klassenname und das *abstract*-Attribut der Klasse übernommen und dann ein Helper-Query zur Ermittlung der neuen Attributmenge aufgerufen.

Das seiteneffektfreie OCL-Helper-Query *derivedAttributes* gibt alle Attribute einer Klasse inklusive der geerbten zurück.

```
1  query Class::derivedAttributes() : OrderedSet(Property)
2  {
3      if self.generalizations->isEmpty() then
4          // Wurzelklasse: Attribute direkt übernehmen
5          self.attributes
6      else
7          // Vereinigung der geerbten und
8          // der eigenen Attribute
9          self.attributes->union(
10             // sammle alle Attribute
11             // der Elternklassen, ..
12             self.generalizations->collect(g |
13                 g.general.derivedAttributes()->
14                     select(attr |
15                         // .. die nicht in unserer
16                         // Klasse existieren (d.h.
17                         // überschrieben wurden)
18                         not self.attributes->
19                         exists(att | att.name = attr.name)
20                     )
21                 )->
22                 flatten())->asOrderedSet()
23     endif
24 }
```

Das Query ist wieder für Instanzen des Typs *Class* definiert und hat keine Parameter, aber einen Rückgabetyp *OrderedSet(Property)*. *OrderedSet* ist ein standard OCL-Typ für sortierte Mengen und *Property* ist ein Typ des Quellmodells. Ein Query besteht immer aus reinem funktionalen OCL-Code ohne imperati-

ve Elemente und ist daher immer seiteneffektfrei. Für nähere Informationen zu OCL sei auf den OCL-Standard [8] verwiesen.

Das Mapping *property2property* kopiert eine Klasseneigenschaft von einer Klasse zu einer anderen und erwartet den neuen Besitzer der Eigenschaft als *ownerClass*-Parameter vom Typ *Class*.

```
mapping Property::property2property(
    in ownerClass : Class) : Property
{
    name := self.name;
    type := self.type;
    owner := ownerClass;
}
```

Schließlich kopiert das Mapping *copyAssociation* Assoziationen vom Quell- in das Zielmodell unter Beibehaltung der Klassenhierarchie.

```
mapping Association::copyAssociation(
    sourceClass : Class) : Association
{
    name := self.name;
    source := sourceClass.resolveByRule(
        'leafClass2Class', Class)->first();
    target := self.target.resolveByRule(
        'leafClass2Class', Class)->first();
}
```

An diesem Transformationbeispiel sieht man, dass QVT-Operational hauptsächlich aus OCL-Code besteht, der in Mappings um einige imperative Konstrukte (z.B. Zuweisungen, Schleifen) erweitert wurde.

Es gibt noch einige Features von QVT-Operational, die in dieser Fallstudie keine Anwendung fanden:

- Explizite Objekterstellung
- Ordnung von Transformationen mit einer Paketstruktur (*packaging*)
- Weitere Kontrollflussstrukturen (while, foreach, if-then-else)
- Unterstützung für die Wiederverwendung: Regelvererbung, Erweiterung von Transformationen
- Nachbedingungen (*where*-Klausel) für Mappings

3.4 Abschließende Bemerkungen zur Fallstudie

Nach Einarbeitung in den OCL-Standard ist die obige Transformation aus Sicht des Autors in QVT-Operational einfach zu implementieren, da die wesentliche Transformationslogik mit einer Erweiterung von OCL ausgedrückt wird. Leider unterstützt SmartQVT (noch) nicht alle Sprachkonstrukte von QVT-Operational, weswegen der obige Quellcode an einigen Stellen für SmartQVT

angepasst werden musste. Beispielsweise kennt SmartQVT nicht die *objectsOfType(Klassse)*-Methode – diese kann aber durch „objects()[Klasse]“ ersetzt werden.

Ein Vorteil des mehrstufigen SmartQVT-Ansatzes (von QVT-Quellcode zum AST, vom AST nach Java) ist die Transparenz für den Entwickler. So kann durch die Generierung von Java-Code der Transformationsprozess nachvollzogen werden. Und das Debuggen geschieht für Java-Entwickler in gewohnter Weise mit Eclipse.

Ein Problem bei der Umsetzung mit SmartQVT war die Bereitstellung der Ein- und Ausgabemodelle im Ecore-Format. Hierbei erfordert SmartQVT Verweise auf bereits unter Eclipse geladene Ecore-Modelle. Daher müssen Ein- und Ausgabemodell jeweils als EMF-Plug-In realisiert werden. Diese Plug-Ins werden dann vor dem Start der SmartQVT-Transformation in die Eclipse-Laufzeitumgebung geladen. Das ist unnötig kompliziert und für Änderungen an den Modellen sehr umständlich, da dies jeweils ein Neukompilieren und -laden der Plug-Ins nötig macht.

4 Bewertung

Die QVT-Spezifikation ist relativ neu und es existieren bisher, auch mangels Implementierungen, noch wenige Erfahrungswerte mit der Transformationssprache. Auch bei der thematische Behandlung in Publikationen scheint der Standard bisher noch keinen rechten Anklang gefunden zu haben, dennoch möchte der Autor auf Basis von Argumenten aus [1] und eigener Einschätzung eine Liste von Vor- und Nachteilen zu QVT zusammenstellen:

4.1 Vorteile

- QVT schließt die Modell-zu-Modell-Lücke in MDA. Im MDA Guide [4] wird viel über Modelltransformationen gesprochen, ohne dabei zu erwähnen wie diese überhaupt durchgeführt werden sollen.
- QVT stellt eine formale Sprache zur Definition von Modelltransformationen bereit, mit deren Hilfe maschinelle Transformationen erst möglich werden.
- QVT ist ein herstellerunabhängiger Standard und ermöglicht dadurch Interoperabilität zwischen verschiedenen Transformationswerkzeugen. Die Nachhaltigkeit von in QVT spezifizierten Transformationsdefinitionen ist dadurch prinzipiell gewährleistet.
- Der Standard ist für viele Einsatzszenarien (siehe 2.3) entwickelt worden und hebt sich durch diese Universalität von anderen (proprietären) Transformationssprachen ab.
- QVT baut of bestehende Standards wie MOF und OCL auf, die bereits sehr breit im Einsatz sind.

4.2 Nachteile

- Umfangreiche Spezifikation: Die QVT-Spezifikation ist sehr umfangreich und vergleichbar mit der UML-Spezifikation, der auch nachgesagt wird, sie sei zu komplex.
- Keine vollständige Implementierung: Es existiert keine vollständige Implementierung und es wird aller Ansicht nach auch nie eine solche geben. QVT bietet eine große Wahlfreiheit (alleine drei verschiedene Sprachen) bei der Lösung eines konkreten Problems, deshalb werden Werzeuge immer nur eine praxis-relevante Untermenge dieser Möglichkeiten implementieren.
- Begrenzte Nützlichkeit: In der Praxis existieren bereits pragmatische Lösungen für das Problem Modelltransformation auf Basis verschiedener Techniken (z.B. XML-Transformation, proprietäre 2GL-Transformationen). QVT besitzt im Vergleich dazu ein sehr begrenztes Verhältnis von Nutzen zu Aufwand und leistet für den Praxisanwender nicht mehr (bspw. hat Bidirektionalität in der Praxis fast keine Relevanz).
- Vorschnelle Standardisierung: Der Standard leidet unter einer vorschnellen Standardisierung [1]. QVT wurde bereits standardisiert als das Problemfeld Modelltransformationen noch nicht hinreichend verstanden wurde.
- Unklare Basis: QVT wurde von vielen unterschiedlichen Parteien definiert. Als Ergebnis musste der Standard viele unterschiedliche Interessen erfüllen und konnte nicht auf eine klare Basis konsolidiert werden [1].
- Deklarative Komplexität: QVT-Relations musste einen hohen Preis für die Möglichkeit von bidirektionalen Transformationen zahlen und ist daher unnötig komplex [1]. Der Nutzen von bidirektionalen Transformationen in der Praxis ist zweifelhaft.
- Unklarer Kompatibilitätsbegriff: Es mangelt an Klarheit hinsichtlich von QVT-Kompatibilität. So gibt es sehr viele Werkzeuge die sich mit QVT-Kompatibilität schmücken, aber sie sind untereinander nicht vergleichbar und teilweise sogar sehr weit vom Standard entfernt (wie z.B. die Transformationssprache Tefkat [13], die auch QVT als Begriff verwendet). Eine Art von Zertifizierung seitens der OMG würde hier sicherlich Klarheit und Sicherheit für den Anwender schaffen.
- Keine Ausnahmebehandlung: Die deklarativen Teile QVTs (Relations und Core) besitzen keinen Mechanismus zum Behandeln von Ausnahmen (Exceptions), welches ein schweres Versäumnis darstellt [1].
- Ungünstige Objektorientierung: In QVT-Operational wurde Objektorientierung zu weit getrieben, so dass sie sehr komplex („fantastically complex“ [1]) und nicht mehr benutzbar ist [1].

4.3 Abschließende Einschätzung des Autors

Nach Ansicht des Autors ist QVT-Relations keine Sprache die man als Entwickler liebgewinnt. Die deklarative Natur von QVT-Relations erfordert für den an moderne imperative Sprachen gewöhnten Entwickler einen hohen Einarbeitungsaufwand und bringt in der Praxis wenige bis gar keine Vorteile. QVT-Operational

kommt dem Entwickler schon näher und könnte sich mit einer ausgereiften Werkzeugunterstützung und einem strikten Kompatibilitätsbegriff als Transformationssprache im Softwareentwicklungsprozess etablieren.

QVT eignet sich nicht für „schnelle“ Lösungen in der modellgetriebenene Softwareentwicklung. QVT lässt sich zur Zeit mangels hinreichend guter Werkzeugunterstützung nicht einfach als „Plug&Play“-Lösung für Modelltransformationen einsetzen. Der Autor erwartet, dass sich Entwickler bzw. Firmen eher strategisch für QVT als herstellerunabhängigen Standard entscheiden dürften, wobei sie dabei bewußt erhöhten Trainingsbedarf und Komplexität in Kauf nehmen müssten. Bei weiterer Verbreitung und Intensivierung modellgetriebener Softwareentwicklung könnte QVT in Zukunft ein Austauschformat für Transformationsdefinitionen sein.

Ob der QVT-Standard die schnelle Evolution der Techniken und Methoden in der modellgetriebenen Softwareentwicklung überlebt, oder von einem de-facto Standard für Modelltransformationen umgangen wird, wird erst die Zeit zeigen müssen.

Literatur

1. Stahl, Völter: Model-Driven Software Development. Wiley & Sons (2006)
2. Czarnecki, Helsen: Feature-based survey of model transformation approaches. IBM Systems Journal, Vol 45 (2006)
3. Petrasch, R., Meimberg, O.: Model Driven Architecture. dpunkt.verlag (2006)
4. Object Management Group (OMG): MDA Guide. Webpublish Version 1.0.1 (Juni 2003) http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.
5. Kleppe, A., Warmer, J., Bast, W.: MDA Explained, The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston (2003)
6. Gardner, Griffin, Koehler, Hauser: A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. OMG (Juli 2003) 21 Seiten http://www.omg.org/docs/ad/03-08-02.pdf.
7. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View-/Transformation Specification. Technical report, Object Management Group (OMG) (2005) http://www.omg.org/docs/ptc/05-11-01.pdf.
8. Object Management Group (OMG): Object Constraint Language (OCL) 2.0 Specification . Technical report, Object Management Group (OMG) (2006) http://www.omg.org/docs/ptc/06-05-01.pdf.
9. Object Management Group (OMG): MOF 2.0 Query/Views/Transformations RFP (April 2002) http://www.omg.org/docs/ad/02-04-10.pdf.
10. Bohlen, M.: QVT und Multi-Metamodell-Transformationen in MDA. OBJEKTspektrum, Nr. 2 2006 (2006) 6 Seiten
11. Wikipedia: QVT. englischsprachiger Wikipedia-Artikel vom 6.5.2007 (2007) http://en.wikipedia.org/wiki/QVT.
12. Hebach, M.: Mit QVT wird MDA erst schön. OBJEKTspektrum Online-Ausgabe Nr. 3 2005 (2005) 3 Seiten http://www.sigs.de/publications/os/2005/MDD/Hebach_MDA_OS_2005.pdf.
13. Lawley, M.: Tefkat - The EMF Transformation Engine (2007) http://tefkat.sourceforge.net/.

14. Belaunde, M., Dupe, G.: SmartQVT - An open source model transformation tool implementing the MOF 2.0 QVT-Operational language (2007) http://smartqvt.elibel.tm.fr/.
15. Kath, O., ikv++ technologies: Overview of QVT with Case Study (Slides). Mailingliste zur MDA-Vorlesung der TU Berlin (November 2006) http://insel.cs.tu-berlin.de/pipermail/mda/2006-November/.

Auf Entwurfsmustern basierende Transformationen

Konrad Jünemann

Betreuer: Steffen Becker

Zusammenfassung Die automatische Generierung von Code aus semiformalen Modellen ist ein wesentliches Merkmal der Modellgetriebenen Architektur (MDA). Um dem Entwickler die Arbeit zu erleichtern wird hierbei versucht, automatisch so viel Quellcode wie möglich direkt aus dem Modell abzuleiten und durch Generatoren zu erzeugen. Dieser Vorgang wirft eine Reihe von Problemen auf, insbesondere durch die Interaktion von generiertem und nicht-generiertem Code.
Diese Arbeit gibt nach einer Einführung in wesentliche Grundlagen eine Übersicht über Lösungsansätze dieser Probleme mit Hilfe von Entwurfsmustern. Dabei werden verschiedene Ansätze zur Trennung von Generat und manuell erstelltem Code vorgestellt und außerdem Möglichkeiten beschrieben, Komponenten untereinander elegant zu entkoppeln.

1 Einleitung

1.1 Problemstellung

Informationssysteme werden heutzutage immer wichtiger, in der Wirtschaft wie auch in der Industrie. Das Konzept der *Modellgetriebenen Softwareentwicklung* (Model-Driven Software Development, MDSD) hilft, die hohen Anforderungen an Qualität und Ausfallsicherheit mit der gestiegenen Größe und Komplexität der Softwareprojekte zu vereinen. Die von der *Object Management Group* (OMG) entworfene MDSD-Variante *Modellgetriebene Architektur* (Model-Driven Architecture, MDA) ist daher immer populärer geworden.

Ein wichtiges Konzept der MDA ist die automatische Erzeugung von Quellcode aus einer zuvor modellierten Beschreibung des Softwareprojekts. Dieser Vorgang wird *Transformation* genannt. In der Praxis ist es aber nicht möglich (und auch nicht erwünscht), nicht-triviale Informationssysteme und ihr Verhalten komplett in formalen Modellen zu beschreiben, da diese Modelle zu komplex und unflexibel werden würden. Daher wird auf einem gewissen Abstraktionsniveau modelliert, was auch andere Vorteile mit sich bringt (siehe Abschnitt 2.1). Als Folge ergibt sich, dass oft nur Teile des Quellcodes mit Hilfe von Modell-zu-Text-Transformationen generiert werden können. Der fehlende Part kann dann von einem Entwickler oder Programmierer manuell eingepflegt werden.

Bei der Generation von Quellcode treten häufig bestimmte Arten von Problemen auf. Insbesondere das Zusammenspiel zwischen generiertem und nicht-generiertem Code führt dabei zu Schwierigkeiten. Entwurfsmuster (das sind Beschreibungen wiederkehrender Probleme mitsamt eines Lösungsansatzes) können helfen, diese zu überwinden. Sie können auch dazu beitragen, besser wartbaren und verständlicheren Code zu generieren. Durch ein solides Wissen über ihre Funktionsweise und ihren Einsatz bzw. ihre Rolle im Kontext von Codegenerierung kann die Arbeit mit automatisch erstelltem Code erleichtert werden.

1.2 Zielsetzung

Diese Arbeit hat zum Ziel, aufzuzeigen, wo und warum Entwurfsmuster im Kontext von Modell-zu-Code-Transformationen eingesetzt werden. Dabei wird sowohl auf die Probleme eingegangen, die durch den Einsatz von Mustern durch Codegeneratoren gelöst werden, als auch Grundlagen zu diesem Thema geliefert, die das Verständnis des Themas erleichtern sollen. Besonderes Augenmerk wird auf zwei Fragestellungen gelegt: Die elegante Kombination von generiertem und nicht-generiertem Code und die Entkopplung der Implementierung einer generierten Software-Komponente von ihren Verbindungen zu anderen Komponenten.

1.3 Aufbau

Abschnitt 2 stellt grundlegende Konzepte und Definitionen vor, die zum Verständnis dieser Arbeit notwendig sind. Dabei wird sowohl auf den Ansatz der

MDSD bzw. der MDA mit besonderem Augenmerk auf Transformationen als auch auf (Entwurfs-)Muster eingegangen.

In Abschnitt 3 wird das Problem der Kombination von generiertem und nichtgeneriertem Code betrachtet. Es werden Möglichkeiten beschrieben, mit der Hilfe von Mustern Code so zu generieren, dass er sich später elegant und effizient manuell erweitern lässt.

Abschnitt 4 befasst sich mit dem Problem der Entkopplung einer Komponente von den von ihr benötigten Komponenten und deren konkreten Implementierungen. Es wird ein Weg aufgezeigt, mit dem erst beim Einsatz eines Systems und nicht schon beim Entwurf der Komponenten entschieden werden muss, welche Komponentenimplementierung verwendet werden soll.

Abschnitt 5 fasst die Arbeit zusammen.

2 Grundlagen

2.1 Modellgetriebene Architektur

In den letzten 20 Jahren ist der Markt für Software immer weiter gewachsen, immer mehr Produkte wurden von der Mikroelektronik und damit auch von der auf ihnen laufenden Software abhängig. Da Software ein immaterielles Produkt ist, sind dessen Kosten hauptsächlich durch die Entwicklung bestimmt. Da technische Waren wie etwa Automobile, Unterhaltungselektronik oder ähnliche immer mehr Digitaltechnik einsetzen, ist auch hier der Preis immer mehr von den Entwicklungskosten der Software abhängig. Hinzu kommt noch, dass Softwareprojekte immer größer und schwieriger beherrschbar werden und Softwarefehler immer höhere Schäden und Kosten verursachen. Lange Zeit wurde aber (und wird in der Regel auch heute noch) Software weniger planmäßig und strukturiert entwickelt als es in anderen technischen Disziplinen wie etwa der Architektur oder anderen Ingenieursberufen der Fall ist. In den 90er Jahren begann auch mit der Entwicklung der *Unified Modeling Language* (UML) [6] ein Umdenken hin zu einem strukturierteren Vorgehen beim Softwareentwurf. Einen Ansatz hierfür bietet die *Modellgetriebene Softwareentwicklung* (*Model-Driven Software Development*, MDSD). Einen guten Überblick über MDSD bietet zum Beispiel [14].

Der MDSD-Ansatz hat im Großen und Ganzen zum Ziel, die während des Entwurfsprozesses anfallende Beschreibung der zu entwickelnden Software in semi-formalen Modellen statt informaler Dokumentation zu binden und diese Modelle zum Mittelpunkt des Entwurfsprozesses zu machen. Beim bisherigen klassischen Softwareentwurf mit UML werden Modelle normalerweise nicht formal definiert und spielen hauptsächlich die Rolle, den Entwicklern und Programmierern eine Orientierungshilfe zu sein. Im MDSD-Kontext ist dies anders: Modelle werden formal und damit für Rechner verständlich definiert. Sie entsprechen nicht der Beschreibung des Ergebnisses einer Phase im Entwicklungsprozess, sondern dem Ergebnis selbst. Die formale Modellierung ermöglicht es, den Entwickler bei der Umwandlung von Modellen durch Tools zu unterstützen. Der MDSD-Ansatz wurde vor allem für folgende Ziele entwickelt (nach [14]):

- MDSD hilft, die Entwicklungszeit zu verkürzen (und damit die Kosten für das Produkt zu senken). Dies wird durch automatische Generierung von Quellcode und / oder Modellen erreicht.
- Der Einsatz von automatisierter Code-Generierung auf formal definierter Basis hilft, die Qualität der erstellten Software zu verbessern, vor allem weil die modellierte Architektur uniform in Quellcode umgesetzt wird. Dabei können häufig auftretende Probleme nach einmaliger Analyse vom Generator bei jedem Entwurf aufs Neue in gleich bleibender Qualität gelöst werden. Die Chance, durch Flüchtigkeit unnötige Fehler bei der Abbildung der Architektur zu machen, wird dabei umgangen.
- Durch MDSD wird eine formale Modellierung auf hoher abstrakter Ebene erzwungen. Dies führt zu einer gründlicheren Planung der Architektur und dadurch zur besseren Beherrschbarkeit der Komplexität und zu besser strukturierter Software.
- Durch Standardisierung der MDSD-Konzepte (wie es im Rahmen der *Model Driven Architecture* (MDA, s.u.) geschehen ist) kann eine bessere Portabilität (Plattformunabhängigkeit) und Interoperabilität (Herstellerunabhängigkeit / Zusammenarbeit von Software-Produkten verschiedener Hersteller) erzielt werden.

Die Object Management Group (OMG), ein 1989 gegründetes Konsortium, das die Entwicklung von Standards zur herstellerunabhängigen und plattformunabhängigen Softwareentwicklung zum Ziel hat, nahm sich 2002 des MDSD-Konzepts an und entwickelte ihre eigene Fassung unter dem Namen *Model Driven Architecture* (MDA) [13]. Kern der MDA sind die verschiedenen Modelle, die ein und dieselbe Software auf verschiedenen Abstraktionsgraden beschreiben. Dabei werden die einzelnen Modelle im Entwicklungsprozess nacheinander entworfen – das eine ist die Grundlage für den Entwurf des nächsten. Die einzelnen Modelltypen werden im Folgenden kurz erläutert (siehe dazu auch Abbildung 1):

Das *Computation Independent Model* (CIM) entspricht der umgangssprachlichen, nicht technischen (computation dependent) Beschreibung der Software [13]. Es entspricht dem Ausgangsprodukt der Anforderungsanalyse und ist als einziges hier beschriebenes Modell nicht formal definiert. In der Definitionsphase wird aus ihm das *Plattform Independent Model* (PIM) entwickelt. Das PIM ist bereits formal definiert, normalerweise in UML. Es beschreibt die Struktur der Software auf einer hohen abstrakten Ebene und ist vollkommen plattformunabhängig, könnte also später beispielsweise sowohl für J2EE als auch .NET entwickelt werden. Im Entwurfsprozess wird auf Grundlage des PIM das sogenannte *Plattform Specific Model* (PSM) erstellt, meist unter Zuhilfenahme von automatischer Modelltransformation (siehe Abschnitt 2.2). Das PSM ist – wie der Name schon sagt – im Gegensatz zum PIM spezifisch zu einer bestimmten Plattform und benutzt deren Konzepte, um das System detailliert zu beschreiben. Da in der Regel bei der Generierung des PSM mehrere Zielplattformen zur Auswahl stehen, kann man sagen, dass die Transformation in das PSM die Wahl der Plattform kapselt. Basiert eine Plattform ihrerseits wieder auf einer anderen Plattform, so wird das zur ersten Plattform spezifische PSM in einem weiteren

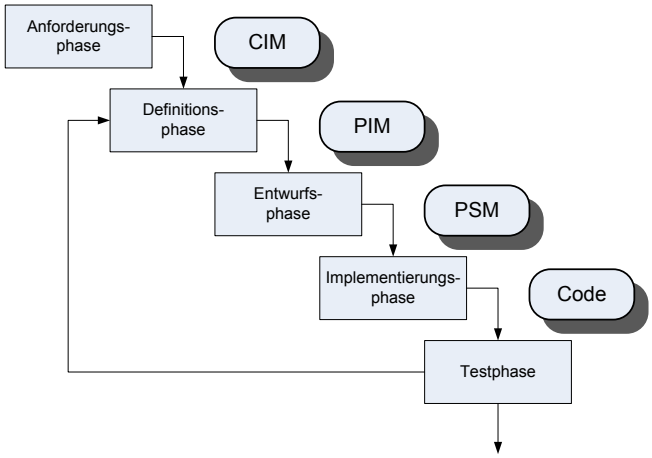


Abbildung 1. Der MDA-Entwurfsprozess und die dabei erstellten Modelle (nach [13])

Schritt zu einem PSM umgewandelt, das zur letzteren spezifisch ist. Ein Beispiel könnte etwa die *Enterprise Java Beans* (EJB) Plattform sein, die auf verschiedenen konkreten Serverplattformen basieren kann, etwa einem *WebLogic Server* (WLS). Hier würde aus dem PIM zuerst ein EJB-PSM und dann aus diesem ein WLS-PSM gebildet. In der Implementierungsphase wird aus dem PSM daraufhin das konkreteste „Modell“ im Softwareentwurf, nämlich der Quellcode, generiert. Dieser Vorgang wird im nächsten Abschnitt näher betrachtet.

2.2 Transformationen

Ein Kernkonzept der MDA ist die formale Beschreibung der zu erstellenden Software durch Modelle. Dabei entspricht jedes Modell dem Ergebnis einer der verschiedenen Phasen des Entwicklungsprozesses. Das resultierende Modell geht dann wiederum in die folgende Entwicklungsphase ein und bildet die Grundlage für die Entwicklung des nächsten Modells. Die automatische Überführung von einem Modell in ein anderes nennt man *Transformation*. Genauer werden Transformationen in [13] definiert:

> „Eine *Transformation* ist das automatische Generieren eines Zielmodells aus einem Quellmodell entsprechend einer Transformationsdefinition, mit Erhalt der Semantik sofern die Sprache des Zielmodells dies zulässt.
>
> Eine *Transformationsdefinition* ist eine Menge von Transformationsregeln, zusammen beschreiben sie, wie ein Modell in der Quellsprache in ein Modell in der Zielsprache transformiert werden kann.

> Eine *Transformationsregel* ist eine Beschreibung, wie eines oder mehr Konstrukte in der Quellsprache in eines oder mehr Konstrukte in der Zielsprache transformiert werden können."

Wie bereits erwähnt, werden Transformationen im Kontext der MDA dafür eingesetzt, verschiedene Modelle ineinander zu überführen. Die Modelle entsprechen insbesondere dem PIM und einem oder möglicherweise mehreren PSM. Der Abstraktionsgrad nimmt dabei von Modell zu Modell ab und infolgedessen werden die Modelle mit immer mehr konkreten Beschreibungen angereichert. Da diese nur schwer auf einer höheren Ebene zu definieren sind, kann die Transformation das Zielmodell in der Regel nicht vollständig automatisch erzeugen. Trotzdem hat der Einsatz von Transformationen große Vorteile gegenüber dem herkömmlichen Ansatz:

- Nach Änderungen in Modellen auf höherem Level müssen die unteren Schichten nicht komplett neu modelliert werden.
- Die automatisierte Modelltransformation spart Zeit gegenüber der manuellen Modellierung.
- Die erzeugten Modelle sind meist von hoher, gleichbleibender Qualität. Einmal modellierte Zusammenhänge werden immer auf die gleiche Art transformiert.

Transformationen von einem abstrakten Modell in ein anderes werden auch als *Modell-zu-Modell-Transformationen* (oder auch *M2M-Transformationen*) bezeichnet. Im Unterschied dazu wird eine Transformationen von einem Modell zu einem Textdokument, wie etwa Programmcode, *Modell-zu-Text-Transformation* (*M2T-Transformation*) genannt. M2T-Transformationen werden im Kontext der MDA für den letzten Transformationsschritt im Entwicklungsprozess eingesetzt. Dies entspricht der Generierung von Quellcode aus einem PSM. In dieser Arbeit werden wir uns auf die Klasse der M2T-Transformationen konzentrieren.

Transformationen werden durch *Transformationssprachen* beschrieben. Die OMG spezifizierte dazu im Rahmen der MOF die Sprache *Query View Transformations* (kurz: *QVT*). Die Spezifizierung ist unter [5] beschrieben. Der Standardisierungsvorgang ist zum gegenwärtigen Zeitpunkt aber noch nicht abgeschlossen. Obwohl Projekte wie zum Beispiel das *Eclipse M2M Projekt* [2] darauf abzielen, die QVT komplett zu unterstützen, existiert momentan noch keine Anwendung, die das QVT gänzlich implementiert. Deshalb setzen existierende Code-Generatoren eine Vielzahl verschiedener Transformationssprachen ein, die jeweils unterschiedliche Vor- und Nachteile mit sich bringen.

M2T-Transformationssprachen teilt man in zwei verschieden Klassen ein, nämlich *schablonen-* und *besucherbasierte* Sprachen (eine Übersicht bietet [10]). Letztere Klasse stützt sich auf das **Besucher**-Entwurfsmuster aus [12]. Dabei wird der Syntaxbaum des Quellmodells vom Generator durchlaufen und Schritt für Schritt umgeformt. Code-Generatoren, die auf diesem Ansatz beruhen, sind relativ einfach zu implementieren, bieten allerdings nicht die Vorteile der schablonenbasierten Generatoren (siehe unten). Besucherbasierte Code-Generatoren sind in der MDA insgesamt kaum verbreitet.

Die Mehrheit der in der MDA eingesetzten Code-Generatoren arbeiten schablonenbasiert. Die dabei verwendeten Schablonen bestehen aus Codefragmenten und Metamarken, die sich auf Eigenschaften des Quellmodells beziehen, sowie Metacode, der zum Beispiel imperative Anweisungen enthalten kann. Man kann sich Schablonen vereinfachend wie komplexe Lückentexte vorstellen, wobei die Lücken zum Beispiel durch den Namen der zu transformierenden Klasse oder ihrer Attribute gefüllt werden. Ein Vorteil der schablonenbasierten Code-Generatoren gegenüber den besucherbasierten ist die bessere Übersichtlichkeit: Während bei letzteren die Transformationsregeln auf mehrere Klassen verstreut implementiert werden, sind Schablonen der Struktur des zu generierenden Codes ähnlicher. Dadurch sind Schablonen in der Regel auch einfacher zu verstehen. Beispiele für schablonenbasierte Code-Generatoren sind das populäre *OpenArchitectureWare* [7] (oAW) mit der Sprache Xpand oder *AndroMDA* [1], das die Sprache VTL einsetzt.

2.3 Muster

Das Konzept der Muster entstammt ursprünglich dem Feld der Architektur und wurde erstmals in [8] beschrieben. Hier schreibt Christopher Alexander: „Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so daß Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.“ [8, Seite x]

Diese Definition enthält bereits – obwohl sie sich nicht auf die Informatik bezieht – die wesentlichen Punkte, die auch Muster in der Softwareentwicklung ausmachen: Die Beschreibung des Kerns eines wiederkehrenden Problems und der dazugehörigen Lösung. Die Probleme können dabei verschiedenen Phasen der Softwareentwicklung entstammen: So werden Muster, die sich auf Probleme während des Entwurfs der grobgranularen Softwarearchitektur beziehen auch als *Architekturmuster* bezeichnet während sich *Entwurfsmuster* mit dem Softwareentwurf auf der Ebene einzelner Klassen oder Module befassen. Als Standardwerk für Entwurfsmuster hat sich [12] etabliert, das 23 grundlegende und viel verwendete Muster beschreibt. Es definiert Entwurfsmuster dabei als „[...] Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.“ [12, Seite 4]. Eine Übersicht über Architekturmuster liefert [9]. [11] konzentriert sich auf Muster in betrieblichen Anwendungen. Es existieren noch weitere Klassen von Mustern, die sich aber nicht mit dem Entwurf der Softwarearchitektur beschäftigen und daher für diese Arbeit nicht relevant sind. *Idiome* sind zum Beispiel feingranulare, in der Regel programmiersprachenspezifische Muster, die bei der Implementierung auftretende Probleme lösen.

Allen Mustern gemein ist, dass sie sowohl das Problem als auch dessen Lösung möglichst weit abstrahieren, um auf möglichst viele Anwendungsfälle anwendbar zu sein. Prägnant beschriebene Muster vereinfachen es dem Softwarearchitekten, den Kern eines Problems rasch zu erkennen und strukturiert zu lösen. Die sich aus dem Einsatz eines Musters ergebenden Konsequenzen sind (sofern sie nicht

auf Spezifika des bearbeiteten Projekts beruhen) im Allgemeinen bereits analysiert und der Beschreibung des Musters beigefügt. Zu beachten ist, dass Muster den Softwarearchitekten nicht ersetzen, sondern vielmehr Werkzeuge für ihn darstellen, deren Einsatz er sorgsam beurteilen und abwägen muss.

Ein Muster ist nicht als statische Vorgabe, sondern als Vorschlag zu sehen, der möglicherweise noch an das konkrete Anwendungsszenario angepasst werden muss. Es ist auch möglich, mehrere Muster miteinander zu „verflechten“. Oft machen bestimmte Muster sogar den Einsatz anderer Muster nötig bzw. lassen ihn sinnvoll erscheinen.

Entwurfsmuster lassen sich nach ihrer groben Aufgabe klassifizieren. In [12] wird beispielsweise zwischen *Erzeugungs- Struktur-* und *Verhaltensmustern* unterschieden. Erzeugungsmuster beschäftigen sich dabei mit Problemstellungen bei der Erzeugung von Objekten, beispielsweise sichert das **Einzelstück**-Entwurfsmuster (engl.: Singleton) zu, dass maximal ein Objekt einer Klasse erstellt werden kann. Strukturmuster befassen sich mit der Komposition von Klassen oder Objekten, während Verhaltensmuster deren Verhaltensweise zum Ziel haben. Das **Strategie**-Entwurfsmuster (engl.: Strategy) macht zum Beispiel bestimmte Methoden einer Klasse zur Laufzeit austauschbar. Es gibt noch eine Vielzahl anderer Kategorien von Mustern. So teilt [9] Architekturmuster etwa in vier weitere Kategorien ein (*From Mud to Structure*, *Distributed Systems*, *Interactive Systems* und *Adaptable Systems*), während in [11] die beschriebenen Muster für betriebliche Anwendungen in zehn abermals neue Kategorien aufgeteilt sind.

3 Kombination von Generat und manuell erstelltem Code

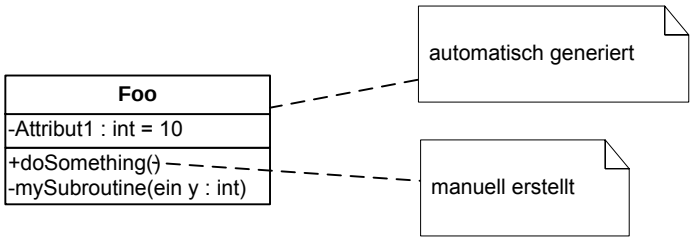


Abbildung 2. UML Diagram der Klasse Foo

3.1 Problem

Auch bei konsequentem Einsatz von MDA kann bei der Entwicklung eines nichttrivialen Systems in der Regel nur ein Teil des Quellcodes durch automatische

M2T-Transformation erzeugt werden. Dadurch kommt es zu einer Mischung aus generiertem und manuell erstelltem Code. Die Integration von manuell erstellten Code in generierte Codefragmente wirft eine Reihe von Problemen auf, die von der Versionsverwaltung der erstellten Software bis zum versehentlichen Überschreiben von Codepassagen reicht. Prinzipiell ist daher eine Trennung oder „geschickte“ Kopplung dieser Code-Typen erwünscht (siehe auch in [15, Seite 30ff]: „Seperate generated and non-generated code“).

In Abschnitt 3.2 werden verschiedene grundsätzliche Möglichkeiten beschrieben, generierten und manuellen Code einer Klasse oder eines Moduls zu koppeln. Diese schließen das manuelle Editieren des Generats mit Hilfe von geschützten Bereichen im Code (engl. „Protected Areas“) ein. Im darauf folgenden Abschnitt 3.3 wird analysiert, wie sich *Vererbung* und *Delegation* als Eigenschaften der objektorientierten Programmierung für die Problemlösung einsetzen. Dabei wird erst auf die generellen Vor- und Nachteile dieser Ansätze eingegangen. In Abschnitt 3.4 werden auf *Vererbung* und *Delegation* basierende Muster beschrieben, die von Generatoren eingesetzt werden können, um den von ihnen erzeugten Code von dem des Entwicklers zu trennen.

Listing 2.1. Die Klasse **Foo** mit geschützten Bereichen

```
1  public class Foo {
2      int x = 10;
3
4      public void doSomething() {
5          // PROTECTED AREA BEGIN #001
6          // insert your code here
7          // PROTECTED AREA END #001
8      }
9
10     private void mySubroutine(int y) {
11         this.x += y;
12     }
13 }
```

3.2 Code bezogenes Vorgehen

Geschützte Bereiche Die einfachste und wohl intuitivste Art, Generat und manuellen Code zu koppeln, ist, die gewünschten Passagen manuell in den generierten Code einzuarbeiten. Dadurch würden allerdings jedes mal, wenn der generierte Code neu erstellt wird, die vorgenommenen Änderungen überschrieben werden und folglich wieder neu integriert werden müssen. Darum ist der Einsatz von sogenannten „geschützen Bereichen“ (oder auch engl. „Protected Regions“) nötig. Das sind vom Generator maschienenlesbar markierte Bereiche im generierten Code, die bei einer erneuten Transformation des Quellmodells nicht überschrieben werden. Der Entwickler darf daraufhin nur noch Änderung innerhalb dieser Bereiche vornehmen. In der Regel werden die Markierungen, die

die geschützten Bereiche umschließen, durch Kommentare in einem vom Transformator gewählten Format umgesetzt. Abbildung 2 zeigt einen Ausschnitt eines UML Modells, das in Listing 2.1 unter Zuhilfenahme von geschützten Bereichen in Java-Code transformiert wurde.

Der Einsatz von geschützten Bereichen hat mehrere Nachteile (siehe auch [14, Seite 160]):

- Der Generator muss die geschützten Bereiche verwalten: Insbesondere muss er sie anlegen, wiedererkennen, einlesen und in neu erstellte Dokumente überführen können. Dadurch wird er komplexer und schwerer handhabbar.
- Besonders das Überführen des manuell erstellten Codes ist nicht immer leicht umzusetzen. Im Falle eines veränderten Zielmodells können sich Position und Zusammensetzung der geschützen Bereiche verändern, neue können hinzu kommen und andere gelöscht werden. In der Praxis zeigt sich, dass manchmal Code verloren gehen kann.
- Die Trennung zwischen Generat und manuell erzeugtem Code wird eingeschränkt, da beide Typen in derselben Klasse und Datei vorkommen. Das führt insbesondere dazu, dass der Entwickler innerhalb von generiertem Code arbeiten muss, wozu er diesen erst einmal verstehen muss. Das ist zum einen oft nicht leicht und zum anderen unerwünscht, weil dadurch ein Vorteil des MDA Ansatzes verloren geht, da dies unnötig Zeit kostet.

Als Vorteil erweist sich dagegen, dass sich geschützte Bereiche – im Unterschied zu allen anderen Ansätzen – in Verbindung mit jedem Dateityp (also nicht nur ausschließlich Quellcode) einsetzen lassen.

Partielle Klassen Manche Programmiersprachen (wie etwa unter der .NET Plattform) unterstützen sogenannte *Partielle Klassen*. Dies sind im Grunde genommen ganz gewöhnliche Klassen, die aber in mehreren unterschiedlichen Dateien definiert werden können. Dafür werden die Klassendefinitionen in jeder Datei mit dem Schlüsselwort `partial` markiert. Beim Kompilieren werden die Klassenfragmente dann zu einer einzigen Klasse zusammengesetzt.

Partielle Klassen ermöglichen eine in manchen Bereichen elegantere Kopplung von Generat und manuell erstellten Code, als es mit geschützten Bereichen möglich wäre. Der Ansatz ist dabei der, für den generierten Code andere Dateien zu benutzen als für den manuell erstellten. Dadurch werden einige der oben beschriebenen Nachteile verhindert:

- Die Verwaltung von partiellen Klassen ist einfacher als die Verwaltung von geschützten Bereichen. Deshalb wird ein weniger komplexer Generator benötigt. Allerdings muss dafür ein Teil der Komplexität auf den Compiler übertragen werden.
- Die Überführung des manuell erstellten Codes ist ebenfalls einfacher, da die manuell erstellten Teile oft nicht modifiziert werden müssen. Code kann so nicht verloren gehen.
- Manuell erstellter und generierter Code bleiben voneinander getrennt. Der Entwickler bekommt den generierten Code im Idealfall nicht zu Gesicht.

Listing 2.2. Foo beschrieben über zwei partielle Klassen (mit geschützten Bereichen)

```
1  // Datei "Foo1" - automatisch generiert:
2  public partial class Foo {
3      int x = 10;
4
5      private void mySubroutine(int y) {
6          this.x += y;
7      }
8  }
9
10 // Datei "Foo2" - automatisch generiert, manuell zu bearbeiten:
11 public partial class Foo {
12     public void doSomething() {
13         // PROTECTED AREA BEGIN #001
14         // insert your code here
15         // PROTECTED AREA END #001
16     }
17 }
```

Leider muss er ihn in der Regel immer noch verstehen, um ihn durch eigenen Code ergänzen zu können.

Trotz dieser Vorteile ist der Ansatz auch mit Nachteilen verbunden. Denn es kommt immer noch zu Problemen sobald der generierte Code manuell erstellte Bereiche benötigt, beziehungsweise der Generator zusichern muss, dass diese erstellt werden. Wenn etwa der Generator die Methode `doSomething()` der Klasse aus dem obigen Beispiel generieren muss, tritt er vor das Problem, in welchem Teil der partiellen Klasse er den leeren Rumpf der Methode erstellen soll. Schreibt er ihn in die Datei, in die der automatisch generierte Code gehört, so kann der Benutzer die Methode nur mit Hilfe von geschützten Bereichen implementieren, was die oben beschriebenen Probleme mit sich bringt (siehe Listing 2.2). Wird der Methodenrumpf in einer Datei des Entwicklers generiert tritt die gleiche Problematik zu Tage. Eine Lösung wäre, die Methode gar nicht zu erstellen sondern dies dem Entwickler zu überlassen. Das führt aber dazu, dass nicht sichergestellt werden kann, dass sie auch wirklich implementiert und nicht vergessen wird. Zudem steht diese Lösung im Widerspruch zum Paradigma der MDA, nach der so viel vom Modell wie möglich transformiert werden sollte.

Trotz der eben beschriebenen Probleme ist die Trennung von Code durch partielle Klassen in Verbindung mit anderen Ansätzen nützlich. Nicht relevanter Code kann effektiv vor dem Entwickler verborgen werden, was die Arbeit für ihn angenehmer macht.

3.3 Methodische Ansätze

Delegation Delegation bedeutet, dass eine Klasse Code einer anderen Klasse aufruft. Normalerweise ist dazu eine dauerhafte Assoziation im Sinne der UML zwischen den beiden Klassen vorhanden, die Referenz zur aufgerufene Klasse kann beispielsweise aber über den Rückgabewert einer Methode erhalten worden sein. Eine korrekte Referenz auf eine Instanz zu übergeben ist die Hauptaufgabe der meisten Erzeugermuster (siehe etwa [12] oder Abschnitt 3.4). Im Unterschied zum „geschützte Bereiche“- und „partielle Klasse“-Ansatz ermöglicht die Delegation, generierten Code in komplett anderen Klassen ablaufen zu lassen. Dies ist eine sehr elegante Art der Kopplung von generiertem und manuell erstelltem Code: Der Code bleibt sauber getrennt und der Entwickler muss sich im günstigsten Fall nur mit den Schnittstellen der generierten Klassen beschäftigen. Leider ist es nicht immer einfach, Software so zu modellieren, dass zu generierender und manuell zu erzeugender Code auf unterschiedliche Klassen verteilt ist. Vererbung (möglicherweise in Verbindung mit bestimmten Entwurfmustern, siehe Abschnitt 3.4) kann in diesen Fällen helfen.

Delegation findet praktisch in jedem MDA-Projekt Verwendung zur Kopplung von generiertem und manuell erstelltem Code. Dies tritt vor allem in Verbindung mit der eingesetzten Plattform und anderen Bibliotheken zu Tage, auf die sich der Generator stützt. Bei diesen Fällen ruft generierter Code nicht-generierten auf. Da sich der Generator darauf verlassen kann, dass sich der angesprochene Code nicht ändert, führt dieser Fall zu wenigen Problemen. Der umgekehrte Fall, nämlich der dass nicht-generierter Code generierten aufruft, ist problematischer, weil es hier zu unerwünschten Abhängigkeiten vom Generat kommt, die nicht formal im Modell beschrieben sind. Dadurch können Änderungen am Modell mit folgender Neugenerierung des Codes zu Inkonsistenzen führen. Durch Vererbung kann dieses Problem gelöst werden.

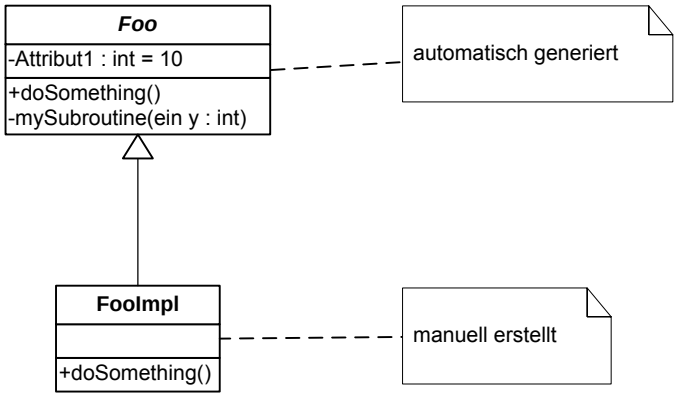


Abbildung 3. Neumodellierung der Klasse `Foo` unter Einsatz von Vererbung

Listing 2.3. Foo transformiert in Ober- und Unterklasse

```
1  // Datei "Foo" - automatisch generiert:
2  public abstract class Foo {
3      int x = 10;
4
5      private void mySubroutine(int y) {
6          this.x += y;
7      }
8
9      public abstract void doSomething();
10 }
11
12 // Datei "FooImpl" - manuell erstellt:
13 public class FooImpl extends Foo {
14     public void doSomething() {
15         // insert your code here
16     }
17 }
```

Vererbung Durch Vererbung „erbt“ eine Unterklasse die Funktionalität einer Oberklasse oder Schnittstelle. Dabei können geerbte Methoden auch überschrieben werden. Wichtig ist die Vererbung bei der Kopplung von generiertem und manuell erzeugtem Code vor allem deshalb, weil sie die Aufteilung des Codes einer Klasse auf mehrere Unterklassen ermöglicht, ohne den Einsatz von geschützten Bereichen zu erfordern. So kann die oben im Beispiel betrachtete Klasse `Foo` durch eine andere Modellierung zu einer abstrakten Oberklasse gemacht werden, die komplett vom Generator erzeugt wird (siehe Abbildung 3). Für die konkrete Implementierung kann dann vom Entwickler eine Unterklasse gebildet und implementiert werden (`FooImpl`, siehe Listing 2.3). Im Unterschied etwa zur Lösung mit geschützten Bereichen wird der generierte Code vor dem Entwickler verborgen und durch die Markierung von `doSomething()` als abstrakt sichergestellt, dass die Methode implementiert wird. Der Zugriff und die Konstruktion des konkreten Objekts wird allerdings erschwert. Hier helfen Entwurfsmuster wie etwa die **Fabrikmethode** (siehe nächsten Abschnitt).

3.4 Muster als Lösungsansatz

Die eben beschriebenen Ansätze zur Kopplung von Generat und manuell erstelltem Code können – wie auch schon oben beschrieben – alleine keine saubere Trennung der beiden Code-Typen gewährleisten. Mit Hilfe von Entwurfsmustern erweisen sie sich aber durchaus als mächtig genug für den genannten Zweck. Im Folgenden werden nun Muster und ihre Funktionsweise im Kontext der Kopplung von Generat und manuell erstelltem Code beschrieben. Die hier aufgezählte Reihe von Mustern ist keineswegs erschöpfend, stellt aber die bei diesem Ziel am häufigsten verwendeten Ansätze dar. Alternativ erfüllen viele Muster aus [12]

einen ähnlichen Zweck wie etwa die Muster **Strategie** [12, Seite 373ff] oder **Brücke** [12, Seite 186ff].

3-Ebenen Hierarchie Im vorangegangenen Abschnitt wurde erläutert, dass ein Vermischung von generiertem mit nicht-generiertem Code innerhalb einer Datei unerwünscht ist. Zudem ist es in manchen Fällen gar nicht möglich, auf diese Weise eine Kopplung zu erreichen, da der Quellcode nicht zugänglich ist, wie etwa bei externen Bibliotheken. Um trotzdem eine grundlegende Trennung von Generat und manuell erstelltem Code zu gewährleisten, wird oft eine **3-Ebenen Hierarchie** [14, Seite 160ff] gewählt. Dieses Muster bedient sich des Vererbungsansatzes um den Code auf mehrere Klassen zu verteilen.

Die 3-Ebenen Hierarchie geht von der Annahme aus, dass die zu generierenden Komponenten in der Regel drei verschiedene Typen von Funktionalität besitzen:

- Funktionalität, die identisch ist für alle Komponenten eines bestimmten Typs. Diese entspricht dem plattformabhängigen Verhalten der Komponenten.
- Funktionalität, die sich zwar bei jeder Komponente unterscheidet, aber dem Modell entnommen werden kann. Diese entspricht im Allgemeinen genau dem im Modell spezifizierten Verhalten.
- Funktionalität, die dem Modell nicht entnommen werden kann und manuell implementiert werden muss.

Die grundlegende Struktur des **3-Ebenen Hierarchie** Musters ist in Abbildung 4 dargestellt. Im Prinzip besteht sie aus drei Ebenen, wobei jeder Ebene einer der drei beschriebenen Typen von Funktionalität zugeordnet ist. Jede der im Modell modellierten Komponenten entspricht nun einer Hierarchie aus drei zu erstellenden Klassen, die voneinander erben.

Die oberste der in Abbildung 4 zu sehenden Ebenen stellt die Plattformebene dar. Auf dieser Ebene wird (manuell) eine abstrakte Oberklasse für alle Komponenten eines bestimmten Typs erstellt. Diese kapselt die Funktionalität des ersten der beschriebenSen Typen. Da dieser das plattformspezifische Verhalten beschreibt, sollte diese erste Klasse „sehr selten“ verändert werden müssen.

Die zweite Ebene entspricht dem Abstraktionsgrad des Modells. Daher werden alle auf dieser Ebene erstellten Klassen vom Generator automatisch erzeugt. Dabei wird für jede Komponente des Modells eine Klasse generiert, die von der entsprechenden Oberklasse erbt. Auch diese generierten Klassen sind abstrakt. Die Modellebene enthält ausschließlich generierten Code, daher braucht der Entwickler nur die Schnittstellen der generierten Klassen zu kennen.

Der dritte Typ von Funktionalität wird auf der tiefsten Ebene definiert. Hierfür erstellt der Entwickler für jede Komponente des Modells eine Unterklasse, die von der entsprechenden abstrakten Klasse auf Modellebene erbt, und implementiert die fehlenden „Lücken“. Dabei kann er sich der weiter unten beschriebenen Muster bedienen, um eventuellen weiteren Probleme bei der Kommunikation mit der Oberklasse zu begegnen.

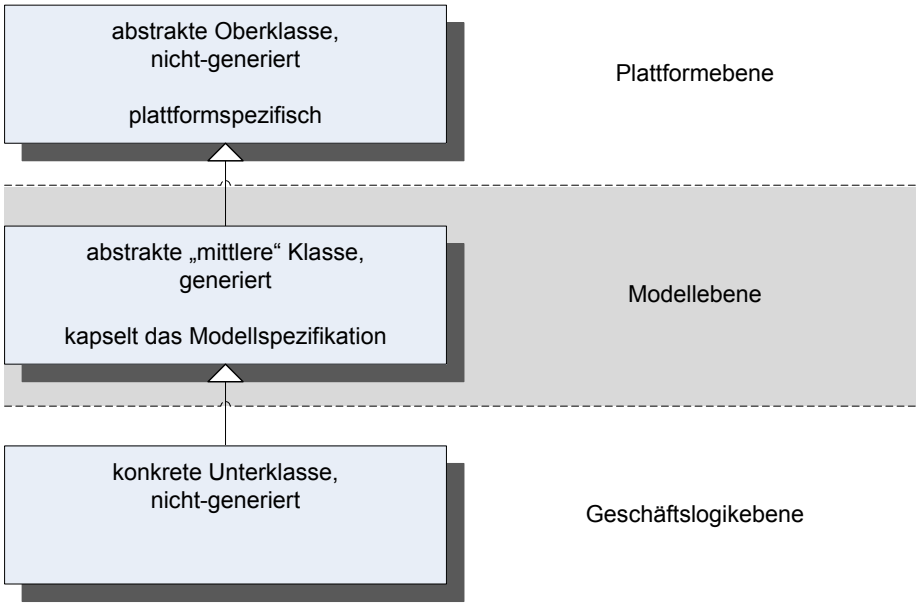


Abbildung 4. Die Struktur der **3-Ebenen Hierarchie**

Der Hauptvorteil dieses Musters ist, dass durch Einsatz der 3-Ebenen Hierarchie die Modell-Ebene klar von den manuell implementierten Teilen entkoppelt ist und sich deshalb sowohl die generierten Teile des Codes als auch die manuell erstellten im Rahmen der Möglichkeiten recht unabhängig voneinander verändern lassen. In keiner Klasse finden beide unterschiedlichen Code-Typen Verwendung. Daher braucht der Generator nicht auf geschützte Bereiche Rücksicht zu nehmen, außerdem muss der Entwickler keinen generierten Code einsehen. Zudem wird durch die Markierung der generierten Klassen als abstrakt zugesichert, dass die bei der Modellierung offen gelassenen Lücken auch vom Entwickler beachtet und gefüllt werden.

Schablonenmethode Wie in Abschnitt 3.2 beschrieben, bringt der Einsatz von geschützten Bereichen viele Nachteile mit sich. Als besonders störend erweist sich die nicht vorhandene Trennung von Generat und manuell erzeugtem Code. Durch Vererbung lässt sich dieses Problem oft vermeiden: Wie in Abbildung 3 dargestellt kann die geerbte Methode implementiert bzw. überschrieben werden. Wenn allerdings eine Methode nur an bestimmten Stellen geändert werden soll ist dies nicht so leicht möglich. Zum Beispiel könnte der Generator wünschen, die Methode automatisch zu Beginn und am Ende eine log-Nachricht ausgeben zu lassen. Beim Überschreiben der Methode durch eine Unterklasse ginge dieses Verhalten verloren.

Die Lösung dieses Problems ist das sogenannte **Schablonenmethode** Entwurfsmuster [12, Seite 366]. Im angesprochenen Beispiel würde der Generator

Listing 2.4. Ersatz von geschützten Bereichen durch **Schablonenmethoden**

```
1  // Datei "Bar" - automatisch generiert:
2  public abstract class Bar {
3      Logger logger = new Logger();
4
5      public final void doSomething() {
6          logger.debug("enter doSomething");
7          doOne();
8          logger.debug("mid of doSomething");
9          doTwo();
10         logger.debug("leave doSomething");
11     }
12
13     protected abstract void doOne();
14     protected abstract void doTwo();
15 }
16
17 // Datei "BarImpl" - manuell erstellt:
18 public class BarImpl extends Bar {
19
20     protected void doOne() {
21         // erster Algorithmenschritt
22     }
23
24     protected void doTwo() {
25         // zweiter Algorithmenschritt
26     }
27 }
```

in der Oberklasse die Methode `doSomething()` als Schablonenmethode generieren, indem er an den Stellen, an denen `doSomething()` erweitert werden soll (also genau den ursprünglichen geschützten Bereichen), abstrakte Methoden aufruft. Diese abstrakten Methoden können dann in einer vom Entwickler erstellten Unterklasse implementiert werden. So bleiben auch bei so enger Zusammenarbeit generierter und nicht-generierter Code getrennt. Listing 2.4 zeigt sowohl die generierte als auch die manuell erstellte Unterklasse des Beispiels (`Bar`, bzw. `BarImpl`).

Das Muster ist auch im umgekehrte Fall hilfreich, wenn also manuell erstellter Code – etwa ein Framework – durch erst nachträglich generierten Code ergänzt werden soll: In der Oberklasse werden genau die Methoden, die vom Generator gefüllt werden sollen, als abstrakt markiert. Sie entsprechen wieder den „geschützten Bereichen“. Da die zu erweiternden „Lücken“ im Code jetzt formal definiert sind, können sie durch Modelle beschrieben und „gefüllt“ werden.

Fabrikmethode Beim Einsatz von Vererbung oder Entwurfsmustern zur Entkopplung von Generat und manuell erstelltem Code tritt oft das Problem auf, Instanzen von Subklassen einer bestimmten Oberklasse zu erzeugen. Im eben zur Verdeutlichung des **Schablonenmethode**-Musters angeführten Beispiel wurde beispielsweise die Unterklasse `BarImpl` gebildet, die den manuell erzeugten Code kapselt. Soll nun aber in generiertem Code eine `BarImpl`-Instanz erzeugt werden kommt es zu dem Problem, dass der Generator die manuell erstellte Klasse nicht kennt (da sie zum Zeitpunkt der Codegenerierung noch nicht existiert) und daher auch keine Instanz erzeugen kann. **Fabrikmethoden** lösen dieses Problem: Sie „[...] ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.“ [12, Seite 131].

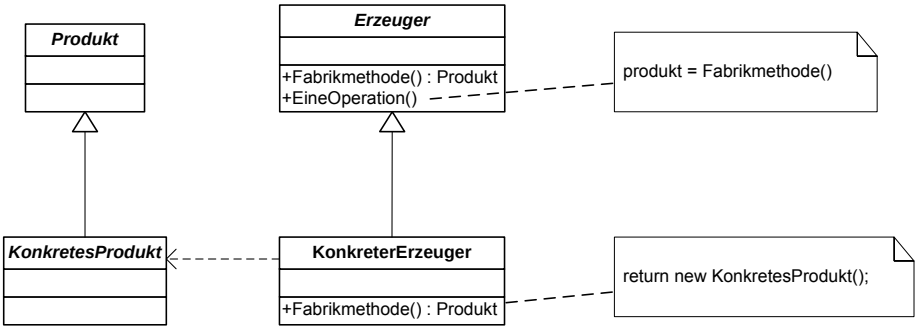


Abbildung 5. Die Struktur des **Fabrikmethode** Entwurfsmusters (aus [12, Seite 133])

Abbildung 5 zeigt die Struktur des Musters. Um dieses Muster einzusetzen, muss der Generator einen abstrakten `Erzeuger` generieren, der eine Fabrikmethode zur Erzeugung einer Instanz der gewünschten Klasse (zum Beispiel `Bar` beziehungsweise `BarImpl`) definiert. Fügt der Entwickler später eine Unterklasse zum abstrakten Produkt (im Beispiel `Bar`) hinzu, so muss er auch einen neuen konkreten Erzeuger definieren, der die Schablonenmethode mit dem nötigen Code zur Erzeugung des konkreten Produkts überschreibt.

Dieses oft verwendete Muster ermöglicht die korrekte Instanzierung von Unterklassen unter Bewahrung der Trennung von Generat und manuell erstelltem Code. Da Vererbung von den meisten Mustern genutzt wird ist auch die Fabrikmethode sehr wichtig für M2T-Generatoren. Ähnliche Vorteile bietet auch das Entwurfsmuster **Abstrakte Fabrik** [12, Seite 107ff].

4 Entkopplung von Komponente und Assembly-Verbindung

4.1 Einleitung

In dieser Arbeit wurden bisher Muster beschrieben, die sich mit Problemen bei der Generierung von Quellcode befassen. In diesem Abschnitt soll nun der Fokus auf Mustern liegen, die vollständig durch generierten Code beschrieben werden. Dadurch, dass sie vollständig vor dem Entwickler verborgen sind, können mit ihrer Hilfe manche Problemstellungen für den Anwender quasi „unbemerkt“ lösen. Als Beispiel wird im Folgenden das Problem der Entkopplung von einzelnen Komponenten und ihrer Assembly-Verbindungen betrachtet.

4.2 Problem

Ein Grundprinzip beim Einsatz von Komponenten ist ihre leichte Austauschbarkeit, da dem Entwickler in der Regel nur ihre Schnittstellenbeschreibung bekannt ist, über die verschiedenen Komponenten kommunizieren. Diese Verbindungen werden durch Assembly Verbindungen dargestellt. Die Kommunikation unter Komponenten wirft ein Problem auf: Die Komponenten benötigen Informationen darüber, mit wem sie verbunden sind. Zur Entwicklungszeit der Komponente ist üblicherweise nur die Schnittstelle der verwendeten Komponenten bekannt, nicht der konkrete Typ. Der Quellcode einer Komponente darf also nicht vom späteren Benutzer geändert werden müssen (oft ist dieser für den Benutzer gar nicht verfügbar). In diesem Anschnitt soll ein Muster beschrieben werden, das dieses Problem löst.

4.3 Lösung mit Dependency Injection

Dependency Injection ist ein Muster, dass durch das „Umkehrung des Kontrollflusses“-Prinzip (engl.: „Inversion of Control“) versucht, die Abhängigkeiten zwischen Objekten oder Komponenten an einer einzelnen Stelle konfigurierbar zu machen [3]. Das Ziel ist, erst beim Deployment bzw. Einsatz des Produkts entscheiden zu müssen, welche Komponentenimplementierungen genau eingesetzt werden sollen.

Martin Fowler beschreibt Dependency Injection in [4] und stellt das Muster ausführlich anderen Entwurfsalternativen gegenüber. Die Struktur ist in Abbildung 6 dargestellt. Sie bezieht sich auf den Fall, dass für den Einsatz von Komponente B Komponente A erforderlich ist. In der Abbildung entsprechen die Klassen A bzw. B diesen Komponenten, genauer gesagt deren Provided- bzw. Required-Types. Für das Entwurfsmuster notwendiger Code wurde der Einfachheit halber an diese Klassen annotiert um sich eine weitere Vererbungsebene zu sparen. Bei Einsatz dieses Musters durch einen Generator würde dieser Code automatisch generiert werden und entsprechend der **3-Ebenen Hierarchie** einer Modelebene zugewiesen. Für das Verständnis des Musters reicht es aus, sich die Klassen A und B als „abstrakte“ Komponenten zu denken, die den Code kapseln, der jeder dieser Komponenten gemein ist.

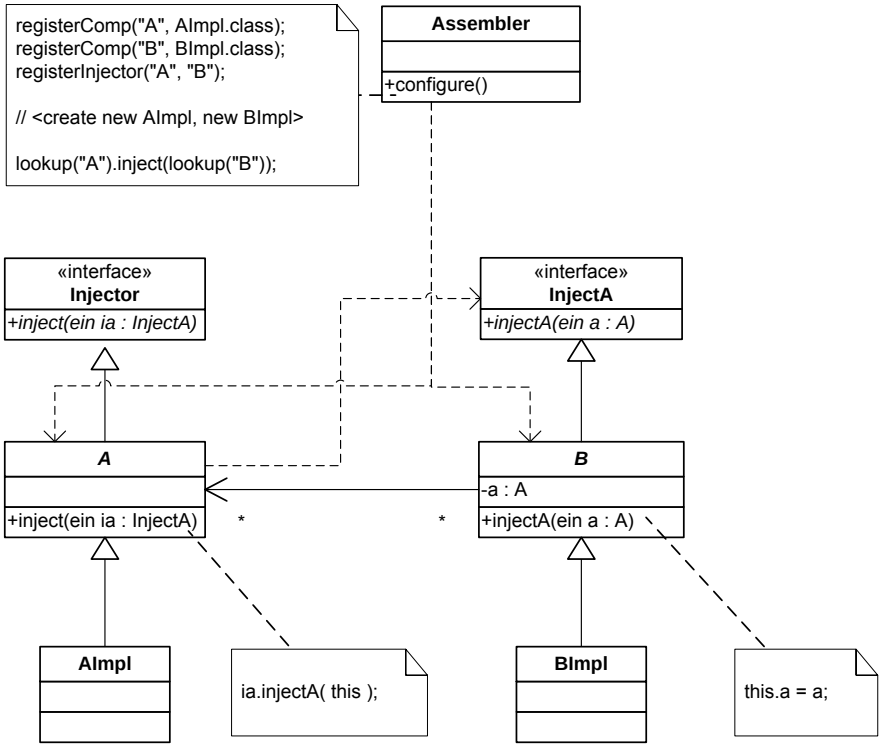


Abbildung 6. Die Struktur des **Dependency Injection** Musters

Im dargestellten Beispiel wird der Komponente B die Abhängigkeit zu einer konkreten Implementierung der Komponente A „injiziert“. Dies geschieht durch einen Aufruf der Methode `inject(InjectA)` der Komponente A mit einer Instanz einer konkreten Implementierung der Klasse B als Argument. Daraufhin wird dieser Instanz das korrekte Objekt der Klasse A übergeben. Welche konkrete Komponente benutzt werden soll, wird von einer dedizierten konfigurierbaren Klasse entschieden. Der Entwickler trägt hier die gewählten Komponenten ein. Zusätzlich registriert er den aufzurufenden `Injector`, in unserem Falle Komponente A. Im Beispiel wurde dies durch Pseudocode verdeutlicht, in Wirklichkeit würden sämtliche Konfigurationseinstellungen natürlich über Konfigurationsdateien durchgeführt werden, bei *Enterprise Java Beans* (EJB) beispielsweise über die Datei `ejb-jar.xml`. Bei Aufruf von `configure()` werden Instanzen der gewählten Komponenten erstellt und die Prozedur zur Injektion der Abhängigkeit gestartet.

Das Muster ist gut für automatische Codegeneration geeignet: Sämtlicher generierter Code kann (mit Hilfe der 3-Ebenen Hierarchie) sauber von manuell erstelltem getrennt werden. Vorraussetzung ist allerdings, dass die eingesetzten Komponenten Dependency Injection unterstützen, also schon mit Hinblick auf dieses Muster entwickelt wurden.

Es existieren zwei weitere Varianten dieses Musters, die auf setter-Methoden bzw. Konstruktor Aufrufen beruhen. Unter [4] werden diese detailliert erläutert. Zu bemerken ist auch, dass das Muster nicht auf den Einsatz mit nur zwei Komponenten beschränkt ist, es kann auch an eine größere Anzahl von zu verwaltenden Komponenten angepasst werden.

5 Fazit

Entwurfsmuster spielen im Kontext der MDA eine wesentliche Rolle, da hier die Architektur der Software eine noch wesentlichere Rolle spielt als in der klassischen Softwareentwicklung. In Abschnitt 2 wurde eine Übersicht über viel verwendete Mustergruppen, das Konzept des MDSD und Transformationen gegeben. Darauf folgend wurde eine Übersicht über grundlegende Kopplungsarten von generiertem und nicht-generiertem Code gegeben.

Die Vermischung von generiertem mit nicht-generiertem Code in einem Dokument sollte im Allgemeinen vermieden werden. Ist dies einmal nicht möglich, bieten sich für den Generator zwei Möglichkeiten an: Der Einsatz von geschützten Bereichen oder von partiellen Klassen (sofern in der Zielsprache vorhanden). Geschützte Bereiche haben den Nachteil, dass der Generator diese verwalten muss und es dabei leicht zu Fehlern kommen kann, die dann eventuell sogar zum Verlust von Code führen können. Zudem wird der Entwickler mit „fremden“ Code konfrontiert, den er erstmal verstehen muss, bevor er seine Implementierung einfügen kann. Partielle Klassen ermöglichen eine elegantere Trennung und sind in der Lage, generierten Code vor dem Entwickler zu verbergen. Leider sind partielle Klassen sprachenspezifisch und beispielsweise unter Java nicht verfügbar. Zudem sind sie in der Regel nicht in der Lage, die Interaktion zwischen gene-

riertem und manuell erstelltem Code ohne Rückgriff auf geschützte Bereiche zu ermöglichen.

Vererbung und Delegation sind Konzepte, die vor allem in Verbindung mit Entwurfsmustern verwendet werden. So kann die Kopplung von Generat und manuell erstelltem Code gut mit Hilfe der 3-Ebenen-Hierarchie gelöst werden. Der Generator erstellt dabei die mittlere Ebene, die wiederum von der Plattformebene abhängig ist. Der Entwickler kann die Anwendung durch Bildung von Unterklassen elegant erweitern ohne viel generierten Code lesen zu müssen. Die dabei noch auftretenden Probleme können durch Muster wie **Schablonenmethode**, **Fabrikmethode** oder anderen (z.B. aus [12]) gelöst werden.

Als ein Beispiel für die Verwendung von Mustern im Generat wurde in Abschnitt 4 das Problem der Entkopplung von Komponenten und ihrer Assembly-Verbindungen beschrieben. Das **Dependency Injection**-Muster löst dieses Problem dadurch, dass es die Entscheidung, welche konkrete Implementierung einer Komponente verwendet wird, an einem Ort kapselt und über Konfigurationsdateien zugänglich macht. Den Komponenten werden dann erst zur Laufzeit die korrekten Referenzen zu den benötigten Komponentenimplementierungen übergeben.

Literatur

1. AndroMDA – Offizielle Homepage. http://www.andromda.org/. Letzter Zugriff: 03.07.2007.
2. Eclipse M2M Projekt. http://www.eclipse.org/m2m/. Letzter Zugriff: 03.07.2007.
3. Inversion of control. http://martinfowler.com/bliki/InversionOfControl.html. Letzter Zugriff: 03.07.2007.
4. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html. Letzter Zugriff: 03.07.2007.
5. MOF QVT Specification. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01. Letzter Zugriff: 03.07.2007.
6. Offizielle Homepage der OMG zu UML. http://www.uml.org/. Letzter Zugriff: 03.07.2007.
7. Open Architecture Ware - Offizielle Homepage. http://www.eclipse.org/gmt/oaw/. Letzter Zugriff: 03.07.2007.
8. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A pattern language*. Oxford University Press, 1977.
9. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture*. Wiley, 1996.
10. K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*. ACM Press, Oktober 2003.
11. M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2003.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster*. Addison-Wesley, 2004.
13. A. Kleppe, J. B. Warmer, and W. Bast. *MDA explained*. Addison-Wesley, 5. print. edition, 2007.
14. T. Stahl and M. Völter. *Model driven software development*. Wiley, 2006.
15. M. Völter and J. Bettin. Patterns for Model-Driven Software-Development. Technical report, Völter – Ingenieurbüro für Softwaretechnologie Heidenheim, Germany, 2004.

AndroMDA

Christian Janz

Betreuer: Christoph Rathfelder

Zusammenfassung In dieser Arbeit wird das Open-Source-Werkzeug AndroMDA [1] vorgestellt und erläutert, wie es im Rahmen der modellgetriebenen Entwicklung eingesetzt werden kann. Hierzu wird nach einer kurzen Beschreibung der Model Driven Architecture (MDA) [2] aufgezeigt, wie sich AndroMDA in diese einordnet. Anschließend wird nach der Erläuterung der Architektur anhand eines Beispiels gezeigt, wie mit AndroMDA aus UML Diagrammen lauffähige Komponenten generiert werden können.

1 Einleitung

Die Object Management Group (OMG) hat mit der Model Driven Architecture einen Rahmenstandard zur modellgetriebenen Entwicklung geschaffen. Hierbei verlagert sich der Fokus des Entwicklers weg von der Entwicklung hin zu Modellen [3]. Die Transformation der Modelle in plattformspezifischen Quellcode erfordert dabei Werkzeuge und Rahmenwerke. Die vorliegende Arbeit stellt das auf Java basierende MDA-Werkzeug AndroMDA vor, welches als Open-Source-Projekt verfügbar ist.

Das nächste Kapitel gibt zuerst eine kurze Übersicht über die MDA und ordnet dann AndroMDA in den Entwicklungsprozess der MDA ein. In Kapitel 3 wird die Architektur von AndroMDA vorgestellt. Es wird hierbei erst eine Übersicht über die an der Codegenerierung beteiligten Komponenten gegeben, bevor diese im Detail vorgestellt werden. Daran anschließend wird in Kapitel 4 auf die Entwicklung einer Anwendung mit AndroMDA eingegangen. In drei Schritten wird erklärt, wie man die UML Modelle erstellt, den Code generiert und welche manuellen Erweiterungen durchgeführt werden müssen, um eine fertige Anwendung zu erhalten. Schließlich folgen das Fazit und ein Ausblick auf zukünftige Entwicklungen des Werkzeugs.

2 Model Driven Architecture

In diesem Kapitel soll die von der OMG standardisierte MDA [2] vorgestellt werden. Im ersten Unterabschnitt wird eine Übersicht gegeben, in der vor allem die verschiedenen Ebenen der Modellierung erläutert werden. Der zweite Abschnitt ordnet dann den Transformationsprozess von AndroMDA in diese Ebenen ein.

2.1 Übersicht über die MDA

Bei der MDA werden Software-Systeme dadurch erstellt, dass man abstrakte Modelle entwickelt und diese dann schrittweise transformiert und verfeinert. Die MDA definiert dabei vier Modellebenen.

Das Computation Independent Model (CIM), das auch als Domain Model oder Business Model bezeichnet wird, beschreibt das System aus einer von Hard- und Software unabhängigen Perspektive, d.h. die Benutzung des Systems bzw. dessen Betriebsumgebung steht im Fokus. Der Hauptnutzen dieser Modellebene besteht darin, ein besseres Verständnis für das zu entwickelnde System und eine gemeinsame Terminologie zu erlangen [4].

Bei der modellgetriebenen Software-Entwicklung ist es wichtig, die Modellierung der Fachdomäne vollständig plattformunabhängig durch ein Platform Independend Model (PIM) zu gestalten. Es sind also ausschließlich die rein fachlichen Aspekte zu betrachten und zu modellieren. Ein PIM genügt - im Gegensatz zum CIM - den formalen Anforderungen an die Transformation im Rahmen von MDA [4].

Die Spezialisierung des PIM erfolgt im Platform Specific Model (PSM). Das PIM wird hierzu durch die Anwendung von Transformationen in ein formales Modell umgewandelt, das das zu entwickelnde Software-System in Bezug auf eine konkrete Plattform spezifiziert.

Die Platform Specific Implementation (PSI) stellt die letzte Ebene der Software-Erstellung dar. Dies ist in aller Regel der Quellcode [4].

2.2 Einordnung von AndroMDA in die MDA

Nun soll AndroMDA in die MDA eingeordnet werden. Ausgangspunkt bei der Software-Erstellung mit AndroMDA bilden UML Modelle der PIM Ebene, d.h. Modelle, die die fachlichen Aspekte des Software-Systems vollständig und formal definieren. Diese UML Modelle werden dann unter Verzicht auf das PSM direkt über eine Model-to-Text Transformation in Quellcode umgewandelt. Es findet also eine Transformation vom PIM zur PSI statt. Diese Transformation erfolgt mit Hilfe von Templatesprachen, die die Erstellung von Schablonen für den zu generierenden Quellcode ermöglichen [3] [5] [6].

3 Die Architektur von AndroMDA

Dieses Kapitel befasst sich mit der Architektur von AndroMDA. Es wird zuerst eine Gesamtübersicht über den Codegenerierungsprozess und die daran beteiligten Komponenten gegeben. Danach wird der Aufbau der einzelnen Komponenten erläutert, insbesondere soll der Aufbau der sogenannten Cartridges erklärt

werden. Die Cartridges stellen eine Art Plugin-Konzept dar, mit Hilfe dessen AndroMDA um Transformationen für neue Zielplattformen erweitert werden kann.

3.1 Übersicht

In Abbildung 1 ist der Codegenerierungsprozess in einer Gesamtübersicht dargestellt.

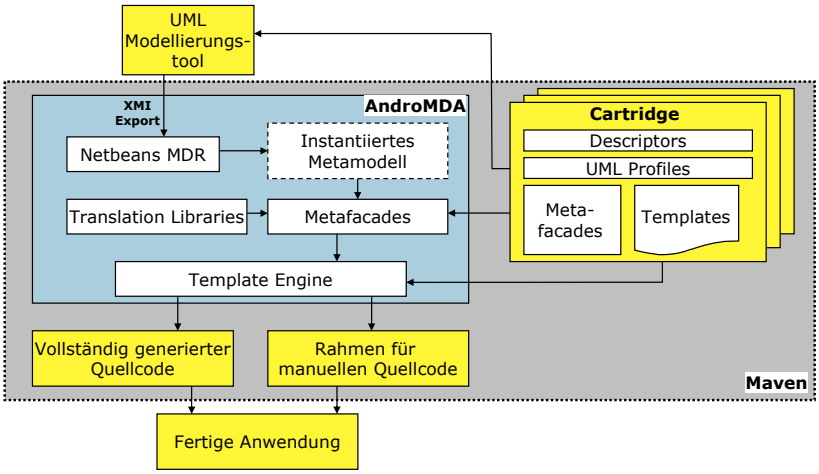


Abbildung 1. Architektur von AndroMDA [3]

AndroMDA besitzt eine Mikrokern-Architektur, das bedeutet, dass es eine kleine Kernkomponente gibt, die nur für das Zusammenspiel, die Integration und das Auffinden der anderen Komponenten verantwortlich ist. Alle anderen Aspekte werden in austauschbaren Spezialkomponenten implementiert.

Der Erstellungsprozess der Software beginnt mit dem Entwickeln des plattformunabhängigen UML Modells. Da AndroMDA lediglich ein Rahmenwerk zur Codegenerierung ist und keine eigenen Modellierungswerkzeuge mitbringt, muss man hier auf ein externes UML-Modellierungswerkzeug zurückgreifen. Die Modellierung in UML basiert auf der Verwendung von Stereotypen und Tagged Values (Eigenschaftswerten), die die Cartridges in Form von UML Profilen bereitstellen. Diese liegen im XML Metadata Interchange (XMI) [7] Format vor und können in das UML-Modellierungswerkzeug importiert werden.
Mit Hilfe der Stereotypen können Modellelemente bestimmten Klassen zugeordnet werden, z.B. dient das Stereotyp *Entity* zur Kennzeichnung einer persistenten Klasse.

Tagged Values werden zur weiteren Feinsteuerung des Generierungsvorgangs verwendet. Mittels eines Tagged Values kann z.B. einer Operation, die zu einer mit *Entity* gekennzeichneten Klasse gehört, eine Datenbankabfrage zugeordnet werden. Ein anderer Eigenschaftswert kann dieser Klasse eine bestimmte Caching-Strategie zuweisen. [3]

Nachdem die Modellierung des PIM abgeschlossen ist, werden die erstellten Modelle per XMIExport AndroMDA zur Verfügung gestellt und der Codegenerierungsprozess kann beginnen.

Die per XMI bereitgestellten UML Modelle werden von der austauschbaren *Repository-Komponente* eingelesen und in Form eines instantiierten Metamodells bereitgestellt. Als Standard-Repository wird das Netbeans Metadata Repository (MDR) [8] verwendet.

Der Zugriff darauf erfolgt jedoch nicht über direktes Verwenden der instantiierten Metamodell-Klassen, sondern über die sogenannten *Metafacades* [9]. Diese verbergen die unterliegende Metamodell-Implementierung und stellen zusätzliche Bequemlichkeitsmethoden zur Verfügung, wodurch die Templates einfacher und weniger komplex werden. Metafacades werden vom AndroMDA Rahmenwerk selbst, sowie von den Cartridges bereitgestellt.

Nachdem durch die Metafacades der Zugriff auf die Modelldaten erreicht wird, werden noch die *Translation Libraries* [10] benötigt. Diese Komponente wird dazu benutzt, Object Constraint Language (OCL) [11] Ausdrücke in andere Sprachen, z.B. SQL, zu übersetzen.

Der eigentliche Codegenerierungsprozess wird dann durch die *Template Engine* realisiert. Diese dient dazu vorher definierte Vorlagen mit Daten zu füllen. Es werden also aus den Templates unter Zugriff auf die Daten des instantiierten Metamodells Quelldateien erzeugt. Der von AndroMDA generierte Quellcode wird in zwei Gruppen unterteilt: Komplett generierter Quellcode und Codeskelette zur manuellen Nachbearbeitung [3]. Als Template Engine steht bisher nur die Velocity Template Engine [12] zur Verfügung.

Die Vorlagen zur Erzeugung des Quellcodes werden durch die *Cartridges* zur Verfügung gestellt. Diese sind die wichtigsten Plugins des AndroMDA Rahmenwerks, da sie unter anderem die Vorlagen für die Quellcode-Erzeugung für bestimmte Zielplattformen bereitstellen und somit dafür verantwortlich sind aus dem bereitgestellten PIM den Quellcode für mehrere Zielplattformen zu erzeugen. Die Auswahl der Cartridges, die beim Generierungsprozess verwendet werden sollen, erfolgt über eine Konfigurationsdatei.

Gesteuert wird der komplette Prozess über das Build-Tool Maven [13]. Das von der Apache Software Foundation entwickelte Open-Source-Projekt unterstützt

den Entwicklungsprozess und kann zum Generieren und Kompilieren von Quellcode eingesetzt werden. AndroMDA stellt für Maven Plugins bereit, die zum Beispiel das Erzeugen von vorkonfigurierten Projekt-Gerüsten per Frage-Anwort-Dialog ermöglichen. Außerdem kann damit der Generierungsprozess automatisiert erfolgen, indem die Generierung in bestehende Prozesse, z.B. nächtliche Builds, integriert wird.

3.2 Repository-Komponente

Die Repository-Komponente dient dazu ein MetaObject Facility (MOF) Modell zu laden und den Metafacades bereitzustellen.
AndroMDA liest standardmäßig UML Modelle aus XMI Dateien. Will man das Verhalten von AndroMDA in der Art ändern, dass es Modelle aus anderen Formaten liest oder andere Repositories als das Netbeans Metadata Repository (MDR) verwendet, muss man eine *RespositoryFacade*-Implementierung erstellen und diese AndroMDA im Klassenpfad zur Verfügung stellen.

Seit Version 3.2 steht nebem dem MDR auch eine Repository-Implementierung zur Anbindung des Eclipse Modeling Frameworks (EMF) [14] zur Verfügung. Somit werden nun auch UML 2.0 und EMF-basierte Werkzeuge unterstützt.

3.3 Metafacade-Komponenten

Die Komponenten, die den Zugriff auf die Modelle, die durch das Repository geladen werden, bereitstellen, werden in AndroMDA als Metafacades bezeichnet. Sie verbergen die verwendete Metamodell-Implementierung. Metamodelle sind dabei MOF Module, wie z.B. UML 1.4 oder UML 2.0.
Neben der Entkopplung von der Metamodell-Implementierung erlauben die Metafacades vor allem den objektoriertierten Zugriff auf die Elemente der Modelle aus den Templates heraus. Dadurch wird die Komplexität der Templates reduziert, da sich die Logik für das Zugreifen, Auffinden und Überprüfen von Modellelementen zentral in den Java-Klassen der Metafacades befindet.

AndroMDA bringt bereits Standardfassaden für den Zugriff auf die UML Modellelemente, z.B. Klassen oder Attribute, mit. Spezialisierungen für Zugriffsklassen für bestimmte Zielplattformen können als Teil der Cartridges entwickelt und benutzt werden. Der Rahmen-Quellcode für die zu erstellenden Metafacades kann dabei über die andromda-meta-cartridge aus einem UML Klassendiagramm generiert werden.
In Abbildung 2 ist eine benutzerdefinierte Erweiterung der Metafacade für Attribute einer UML Klasse dargestellt. Die Fassade für Attribute soll dadurch für Persistenzumgebungen erweitert werden. Im Diagramm werden zwei Stereotypen verwendet: *metafacade* und *metaclass*. Die mit dem Stereotyp *metaclass* gekennzeichneten Klassen sind Teil des Metamodells. In Abbildung 2 sind dies die Elemente *Classifier* und *Attribute*. Die mit dem Stereotyp *metafacade* attributierten Klassen hingegen sind Metamodell-Fassaden-Klassen.

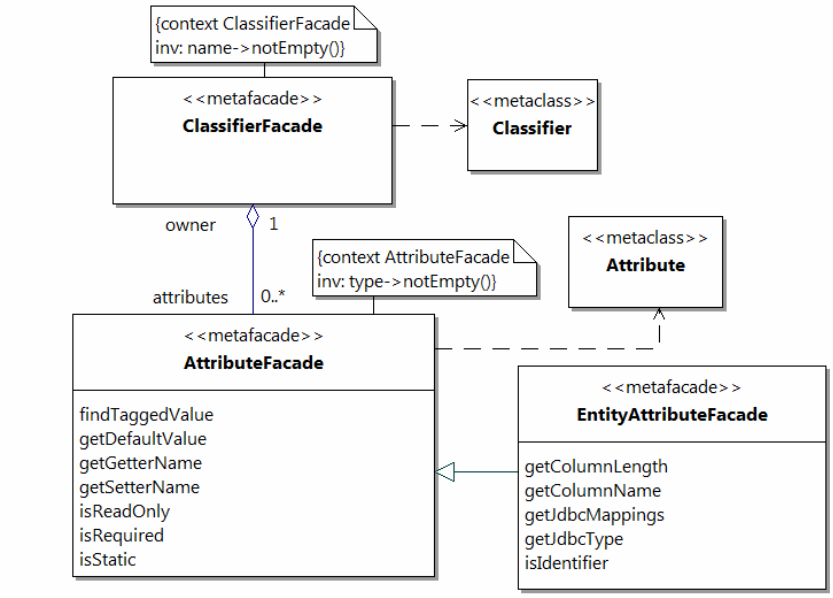


Abbildung 2. Erweiterung von Metafacades [9]

Im Diagramm ist erkennbar, dass die ClassifierFacade von der Metamodell-Klasse Classifier abhängt und die AttributeFacade von der Klasse Attribute. ClassifierFacade und AttributeFacade stehen in einer Kompositionsbeziehung zueinander.
Die neue Klasse EntityAttributeFacade leitet nun von der allgemeinen Klasse AttributeFacade ab, sodass sie alle Methoden zum Zugriff auf die Eigenschaften von Attributen, z.B. getGetterName(), erbt. Sie bietet darüberhinaus zusätzliche Methoden an, die für die Generierung von persistierbaren Klassen notwendig sind, z.B. eine Methode zum Ermitteln der Breite der Tabellenspalte.

Nachdem die Metafacade-Klassen aus dem UML Diagramm generiert wurden und die Logik von Hand hinzugefügt wurde, muss die neue Metafacade per XML Konfigurationsdatei dem AndroMDA Rahmenwerk bekanntgemacht werden. In dieser Datei wird dann die Abbildung zwischen den Metafacade-Klassen und den Metamodell-Klassen definiert, d.h. es wird definiert unter welchen Umständen das Rahmenwerk während des Generierungsprozesses Instanzen der Fassenden-Klassen erzeugen soll.

3.4 Translation Libraries

Ein weiterer Teil des Plugin-Konzepts von AndroMDA sind die sogenannten Translation Libraries. Man versteht darunter Bibliotheken, die OCL Ausdrücke

in andere Sprachen transformieren. Diese Bibliotheken können innerhalb der weiter unten vorgestellten Cartridges verwendet werden, um aus den OCL Ausdrücken von Modellelementen plattformspezifischen Code zu generieren. Man muss die Translation Libraries und OCL nicht verwenden. Stattdessen könnte man Ausdrücke und Abfragen direkt in der Sprache der Zielplattform als tagged value in den Modellen angeben. Das Problem hierbei ist jedoch, dass damit die Plattformunabhängigkeit der Modelle verloren geht, womit diese dann nicht mehr für die Codegenerierung für mehrere Plattformen geeignet sind.
Durch die Verwendung von OCL als standardisierter Teil von UML bleibt das Modell plattformunabhängig und durch die Anwendung der Transformationen der Translation Libraries kann dann während der Codegenerierung daraus plattformspezifischer Code erzeugt werden.

Eine Translation Library besteht aus mehreren Komponenten: Ein XML Dokument beschreibt die Komponenten der Bibliothek und sorgt dafür, dass das AndroMDA Rahmenwerk die Bibliothek laden kann. Außerdem enthält die Bibliothek mehrere XML Dokumente, die für die bestimmten OCL Fragmente plattformspezifische Ausdrücke in Form von Templates angeben. Schließlich muss eine gültige Translation Library noch eine Implementierung der Schnittstelle *Translator* enthalten. Diese steuert den Transformationsprozess und wird aus dem XML Deskriptor referenziert. Eine Anleitung zum Erstellen von eigenen Translation Libraries findet sich unter [10].

3.5 Template Engine

Die Template Engine bildet das Herzsück des Codegenerierungsprozesses. Sie ist dafür verantwortlich, dass die Daten zusammen mit den Vorlagen zu fertigem Quellcode gemischt werden. Es handelt sich hierbei wieder, wie bei den anderen Komponenten auch, um eine austauschbare Komponente. In Version 3.2 von AndroMDA gibt es allerdings nur eine Implementierung der Engine basierend auf der Apache Velocity Template Engine [12].
Die Austauschbarkeit dieser Komponente ist jedoch nur theoretisch möglich, da es viele Abhängigkeiten von der Skriptsprache der Template Engine gibt. So wird die Skriptsprache in den Templates der einzelnen Cartridges und in den Transformationsregeln der Translation Libraries verwendet. Ein Wechsel der Engine ist somit nur mit großem Aufwand möglich, indem alle Templates auf die andere Skriptsprache angepasst werden. Somit sollte auf einen Wechsel der Template Engine verzichtet werden.

3.6 Aufbau einer Cartridge

Nachdem nun die einzelnen Komponenten im Detail erklärt wurden und aufgezeigt wurde, wie dank der Mikrokern-Architektur eigene Komponenten im Rahmenwerk erstellt und eingebunden werden können, soll nun auf die sogenannten Cartridges eingegangen werden. Auf Grund verschiedener Cartridges ist es möglich aus dem selben PIM den Quellcode für mehrere Zielplattformen zu erstellen.

Cartridges sind somit die wichtigste Erweiterungsmethode von AndroMDA, da damit die Codegenerierung für das im Projekt verwendete Rahmenwerk bzw. die verwendete Plattform angepasst werden kann.

Es gibt schon einige Cartridges, die mit AndroMDA ausgeliefert werden. Darunter befinden sich Cartridges für EJB, Hibernate, Java, XMLSchema, JSF, Spring und Web-Services. Außerdem gibt es auch Cartridges, die Code für das Microsoft .NET Rahmenwerk erzeugen. Reichen die verfügbaren Cartridges nicht aus, müssen diese angepasst oder neue Cartridges entwickelt werden.

Eine Cartridge besteht im wesentlichen aus drei Bestandteilen: Metafacade-Klassen (siehe Kapitel 3.3) zum Zugriff auf die Modellelemente, Templates als Vorlagen für die zu erstellenden Codefragmente und drei XML Deskriptoren. Die *namespace.xml* definiert einen Namensraum für die Cartridge, innerhalb dessen Eigenschaften- und Einstellungsbezeichner eindeutig sind, und dient dazu, dass die Cartridge-Komponente vom Rahmenwerk erkannt und geladen werden kann. Die *cartridge.xml* [15] beschreibt im wesentlichen die enthaltenen Templates und legt fest, welche Metafacade-Klassen in den Templates zum Generierungszeitpunkt zur Verfügung stehen sollen. Schließlich beschreibt die *metafacades.xml* die Metafacade-Klassen und ihre Zuordnung zu den Metamodell-Klassen (siehe Kapitel 3.3).

Im folgenden soll der Aufbau einer Cartridge am Beispiel der einfachen XML-Schema Cartridge, die basierend auf einem UML Klassendiagramm ein XML Schema [16] erstellt, erklärt werden.
Das folgende Listing zeigt den Namensraum-Deskriptor der Cartridge.

```
1  <namespace name="xmlschema">
2      <components>
3          <component name="cartridge">
4              <path>META-INF/andromda/cartridge.xml</path>
5          </component>
6          <component name="metafacades">
7              <path>META-INF/andromda/metafacades.xml</path
                   >
8          </component>
9          <component name="profile">
10             <path>META-INF/andromda/profile.xml</path>
11         </component>
12     </components>
13     <properties>
14         <!-- namespace-propertyGroup merge-point -->
15         <propertyGroup name="Outlets">
16             <documentation>
17                 Defines the locations to which output is
                       generated.
```

```
18              </documentation>
19              <property name="schema">
20                  <documentation>
21                      The location to which the XML schema
                            will be written.
22                  </documentation>
23              </property>
24          </propertyGroup>
25          <propertyGroup name="Other">
26              <property name="namespace">
27                  <documentation>
28                      The name that will be given to the
                            target and default namespaces.
29                  </documentation>
30              </property>
31              <property name="xmlEncoding">
32                  <default>UTF-8</default>
33                  <documentation>
34                      The encoding of the schema.
35                  </documentation>
36              </property>
37          </propertyGroup>
38      </properties>
39  </namespace>
```

Es wird ein Namensraum *xmlschema* definiert. In diesem Namensraum werden dann drei Komponenten registriert: Die Cartridge selbst mit der beschreibenden *cartridge.xml*, die Metafacades und die Beschreibung des UML Profils mit den Stereotypen. Außerdem werden Eigenschaften dieser Cartridge deklariert, die die Generierung beeinflussen, z.B. die Eigenschaft *schema* der Gruppe *Outlets*, die den Pfad angibt, wohin das XML Schema Dokument geschrieben werden soll.

Das folgende Listing zeigt den Cartridge-Deskriptor.

```
1  <cartridge>
2      <templateEngine className="org.andromda.
           templateengines.velocity.VelocityTemplateEngine"/>
3      <templateObject name="stringUtils" className="org.
           apache.commons.lang.StringUtils"/>
4
5      <!-- The name of the namespace -->
6      <property reference="namespace"/>
7      <!-- encoding for xml documents -->
8      <property reference="xmlEncoding"/>
9
```

```
10      <template
11          path="templates/xmlschema/XmlSchema.vsl"
12          outputPattern="xmlSchema.xsd"
13          outlet="schema"
14          overwrite="true"
15          outputToSingleFile="true">
16          <modelElements variable="types">
17              <modelElement>
18                  <type name="org.andromda.cartridges.
                        xmlschema.metafacades.XSDComplexType"/
                        >
19              </modelElement>
20              <modelElement>
21                  <type name="org.andromda.cartridges.
                        xmlschema.metafacades.
                        XSDEnumerationType"/>
22              </modelElement>
23          </modelElements>
24      </template>
25  </cartridge>
```

In diesem XML Dokument wird die Cartridge beschrieben. Nachdem als Template-Engine Velocity ausgewählt wurde, wird die Klasse *StringUtils* unter dem Namen *stringUtils* im Kontext der Templates zur Verfügung gestellt. Zusätzlich werden die beiden Eigenschaften, die im Namensraum-Deskriptor deklariert wurden, im Template-Kontext zur Verfügung gestellt. Schließlich wird noch ein Template deklariert: Es wird das Velocity-Template *XmlSchema.vsl* zur Transformation verwendet. Das Template wird auf alle Modellelemente vom Metafassadentyp *XSDComplexType* und *XSDEnumerationType* angewandt und erzeugt die Ausgabe in die Datei *xmlSchema.xsd*. Während der Generierung steht die jeweilige Instanz der Metafacade-Klasse über die Variable *types* zur Verfügung. Beim erneuten Generieren wird diese Datei überschrieben, sie darf also zwischenzeitlich nicht manuell geändert werden.

Im folgenden Listing ist schließlich noch der Metafacade-Deskriptor dargestellt.

```
1   <metafacades>
2       <metafacade class="org.andromda.cartridges.xmlschema.
            metafacades.XSDEnumerationTypeLogicImpl">
3           <mapping>
4               <stereotype>XML_SCHEMA_TYPE</stereotype>
5               <stereotype>ENUMERATION</stereotype>
6           </mapping>
7       </metafacade>
8       <metafacade class="org.andromda.cartridges.xmlschema.
            metafacades.XSDComplexTypeLogicImpl">
```

```
        <mapping>
            <stereotype>XML_SCHEMA_TYPE</stereotype>
        </mapping>
    </metafacade>
    <metafacade class="org.andromda.cartridges.xmlschema.
        metafacades.XSDAttributeLogicImpl">
        <mapping>
            <property name="ownerSchemaType"/>
        </mapping>
    </metafacade>
    <metafacade class="org.andromda.cartridges.xmlschema.
        metafacades.XSDAssociationEndLogicImpl">
        <mapping>
            <property name="ownerSchemaType"/>
        </mapping>
    </metafacade>
</metafacades>
```

Die im Cartridge-Deskriptor verwendeten Metafacade-Klassen werden in diesem XML Dokument definiert, d.h. es wird angegeben, unter welchen Bedingungen sie instanziiert werden. Die Klasse *XSDEnumerationTypeLogicImpl* wird für alle Modellelemente, die mit den Stereotypen *XML_SCHEMA_TYPE* und *ENUMERATION* gekennzeichnet sind, erzeugt. Die Klasse *XSDComplexTypeLogicImpl* wird für alle Modellelemente mit dem Stereotyp *XML_SCHEMA_TYPE* instanziiert. Schließlich werden die Klassen *XSDAttributeLogicImpl* und *XSDAssociationEndLogicImpl* für alle Modellelemente instanziiert, bei denen die Eigenschaft *ownerSchemaType* gültig ist, d.h. für alle Elemente, deren Besitzer ein XML Schema Typ ist (also das Stereotyp XML_SCHEMA_TYPE besitzt).

Das folgende Velocity-Template iteriert über die Typen, die es vom AndroMDA Rahmenwerk in der Variablen *$types* zur Verfügung gestellt bekommt und erzeugt jeweils XML Schema Elemente und Typdefinitionen daraus.

```
<xsd:schema
    targetNamespace="$namespace"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:impl="$namespace"
    elementFormDefault="qualified">
#foreach ($type in $types)
#set ($typeName = "${type.name}Type")
    <xsd:element name="${stringUtils.uncapitalize($type.
        name)}" type="impl:${typeName}"/>
    ...
    <xsd:complexType name="$typeName">
        <xsd:sequence>
        ...
```

```
13          </xsd:sequence>
14 #foreach  ($attribute in $type.attributes)
15 #if ($attribute.xsdAttribute)
16          <xsd:attribute name="${attribute.name}"#if($
                attribute.required) use="required" #end type="
                xsd:${attribute.type.fullyQualifiedName}"/>
17 #end
18 #end
19      </xsd:complexType>
20 #end
21 </xsd:schema>
```

4 Entwicklung einer Anwendung mit AndroMDA

In diesem Kapitel soll anlehnend an [17] das Vorgehen bei der Entwicklung einer Anwendung mit AndroMDA vorgestellt werden. Die entwickelte Anwendung soll zur Erfassung der Arbeitszeit von Mitarbeitern dienen, d.h. es soll erfasst werden, welche Mitarbeiter wann und wie lange an einem Projekt gearbeitet haben. Die Anwendung wird für die J2EE Plattform [18] mit Hilfe der Cartridges BPM4Struts, Spring und EJB realisiert. Die Details der Cartridges und der Plattform bleiben jedoch unberücksichtigt, sodass die vorgestellten Ansätze auch für andere Plattformen genutzt werden können.
Zuerst wird die Modellierung der Anwendung beschrieben, insbesondere sollen die Regeln und Einschränkungen erwähnt werden, die beim Erstellen der Modelle für AndroMDA beachtet werden müssen. Im zweiten Unterkapitel wird dann beschrieben, welche Codefragmente durch AndroMDA aus dem Modell generiert wurden. Schließlich wird im dritten Unterkapitel aufgezeigt, welche manuellen Nacharbeiten noch notwendig sind, bevor die Anwendung produktiv verwendet werden kann.

4.1 Modellierung

Ausgangspunkt der Entwicklung einer Anwendung stellt bei der MDA die Modellierung dar. In diesem Unterkapitel soll deswegen auf das Vorgehen bei der UML Modellerstellung für AndroMDA eingegangen werden. Damit das AndroMDA Rahmenwerk die Quellcodegenerierung korrekt durchführen kann, müssen laut [19] einige Regeln beachtet werden.

- Der Datentyp aller Attribute und Operationsparameter muss spezifiziert werden. Einige UML Modellierungswerkzeuge erlauben es ein Attribut anzulegen ohne seinen Typ zu spezifizieren, die Transformationen der Cartridges benötigen aber auf alle Fälle den Datentyp für die Codeerzeugung.
- Die Sichtbarkeit sollte spezifiziert werden, wo sie von Bedeutung ist. Attribute werden *immer* privat erzeugt, wohingegen die Zugriffs- und Änderungsmethoden jedes Attributs die im Modell definierte Sichtbarkeit erhalten.

- Die Multiplizität beider Enden muss bei allen Assoziationen spezifiziert werden, da die Cartridges diese verwenden und benötigen.
- Die Namen der Assoziationsenden einer Assoziation müssen innerhalb dieser eindeutig sein. Das bedeutet, dass die beiden Assoziationsenden unterschiedlich bezeichnet werden müssen.
- Nicht mehr benötigte Elemente müssen aus dem Modell gelöscht werden. Es reicht nicht, überflüssige Elemente aus den Modelldiagrammen zu entfernen, sie müssen komplett aus dem Modell entfernt werden.
- Falls ein Element nicht ohne ein anderes Element existieren kann, muss die Assoziation als Komposition definiert sein. Zum Beispiel besteht zwischen einem Haus und einem Raum eine Kompositionsbeziehung, da die Zerstörung des Hauses die Zerstörung aller Räume dieses Hauses nach sich zieht.
- Die Reihenfolgentreue sollte dort angegeben werden, wo sie von Bedeutung ist. Bei mehrwertigen Attributen und Assoziationsenden sollte im Modell angegeben sein, ob diese sortiert sind. Die Cartridges verwenden diese Eigenschaft um die entsprechenden Klassen der Zielplattform zu wählen.
- Für Attribute dürfen keine getter und setter Operationen angegeben werden. Die Cartridges erzeugen diese während des Generierungsprozesses automatisch.
- Für Assoziationsenden dürfen ebenfalls keine getter und setter Methoden definiert werden, diese werden auch automatisch erzeugt.
- Es müssen UML Datentypen für einfache Typen wie String, Collection etc. angegeben werden anstatt sprachspezifischer Typen (z.B. java.lang.String). Das Modell muss plattformunabhängig bleiben. Die Übersetzung in plattformspezifische Typen findet durch die Cartridges statt.

Nachdem nun die Regeln aufgezählt wurden, die bei der Modellerstellung beachtet werden müssen, werden nun Modelle der Beispielanwendung gezeigt und erklärt.
AndroMDA verwendet drei Arten von UML Diagrammen: Klassendiagramme werden zur Modellierung der Geschäftsobjekte, der Dienste und der Controller-Klassen verwendet. Für die Erzeugung der Benutzeroberfläche werden Anwendungsfalldiagramme und pro Anwendungsfall ein Aktivitätsdiagramm verwendet.

In Abbildung 3 ist das in der Beispielanwendung verwendete Klassendiagramm dargestellt. Es enthält alle Geschäftsobjekte und ihre Beziehungen zueinander. Die zwei wesentlichen Geschäftsobjekte sind *User* und *TimeCard*, sie repräsentieren die Benutzer des Systems und die zugehörigen Zeiterfassungskarten und befinden sich im oberen Teil des Diagramms. Dadurch, dass sie mit dem Stereotyp *Entity* gekennzeichnet sind, erkennt AndroMDA, dass es sich um Geschäftsobjekte handelt, die persistent gehalten werden müssen. Zusätzlich gibt es zu jedem Geschäftsobjekt auch Klassen, die mit dem Stereotyp *ValueObject* gekennzeichnet sind. Dies sind Kopien der Entity-Klassen, die auf Präsentationsebene verwendet und nicht persistiert werden. Die Entity-Klassen besitzen Abhängigkeiten von den ValueObject-Klassen. Das hat zur Folge, dass die Cartridges Methoden zur Transformation der Entity-Objekte in ValueObject-Objekte

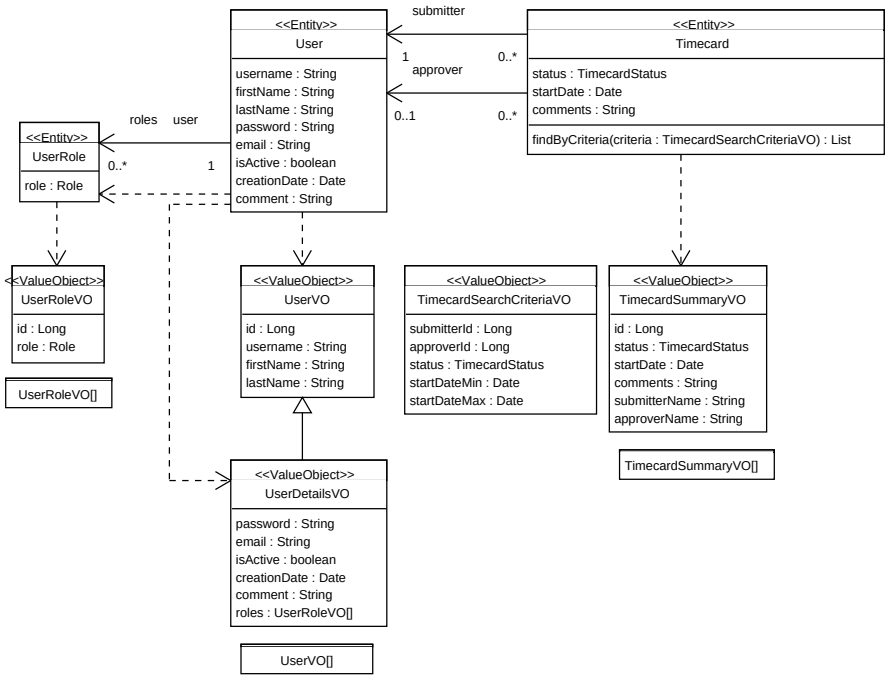
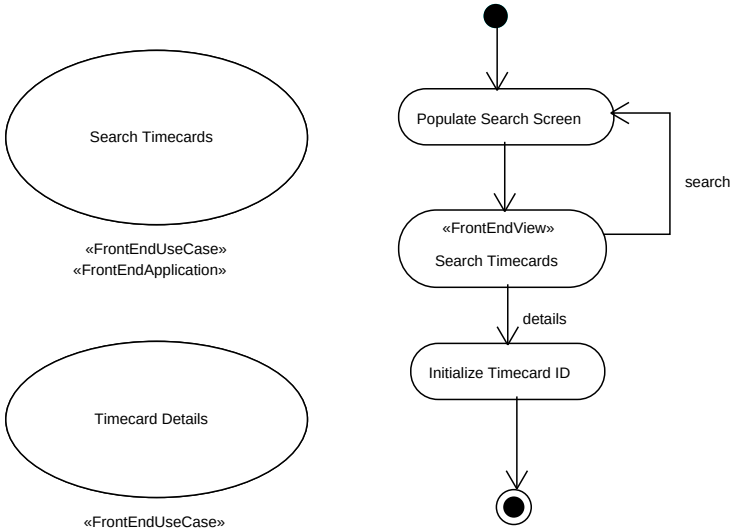


Abbildung 3. Geschäftsobjektmodell [17]

erzeugen.
An diesem Diagramm erkennt man auch gut die Umsetzung der oben genannten Regeln. So sind z.B. bei allen Assoziationen die Multiplizitäten und Rollennamen angegeben. Außerdem sind bei den Klassen keine Methoden für den Zugriff auf die Attribute spezifiziert.

In Abbildung 4 sind das Anwendungsfall- und das Aktivitätsdiagramm für den Anwendungsfall *Search Timecards* in vereinfachter Form dargestellt.
Das Anwendungsfalldiagramm zeigt dabei zwei Anwendungsfälle: Das Suchen von Zeiterfassungskarten und die Anzeige der Details einer Karte. Beide Anwendungsfälle sind mit dem Stereotyp *FrontEndUseCase* gekennzeichnet. Dies signalisiert den Cartridges, dass diese Anwendungsfälle in der Benutzeroberfläche der Anwendung dargestellt und abgehandelt werden müssen. Der Anwendungsfall *Search Timecards* ist zudem mit dem Stereotyp *FrondEndApplication* gekennzeichnet, was bedeutet, dass dies den Einstiegspunkt der kompletten Anwendung darstellt.

Der Anwendungsfall *Search Timecards* wird durch das UML Aktivitätsdiagramm 4(b) verfeinert. Grundsätzlich wird innerhalb einer Aktivität zwischen zwei Arten von Aktionen unterschieden: Aktionen, die eine Repräsentation in der Be-



(a) Anwendungsfälle (b) Aktivität: Search Timecards

Abbildung 4. Modellierung der Benutzeroberfläche

nutzeroberfläche benötigen, z.B. für die Eingabe von Daten, und Aktionen, die keine Repräsentation in der Oberfläche benötigen und ohne Benutzerinteraktion Daten verarbeiten. Aktionen, die in der Benutzeroberfläche angezeigt werden sollen, sind mit dem Stereotyp *FrontEndView* gekennzeichnet. Übergänge aus diesen Aktionen heraus erfolgen über das Anklicken von Buttons durch den Benutzer.
Die Aktivitätsdiagramme müssen noch um ein paar Eigenschaften erweitert werden, damit vollständiger Code generiert werden kann. Es müssen zum Beispiel die Daten spezifiziert werden, die bei einem Zustandsübergang übertragen werden. Zu Gunsten der besseren Übersichtlichkeit wurde jedoch auf diese Eigenschaften verzichtet.

4.2 Codegenerierung

Nachdem im vorherigen Unterkapitel die Modellierung der Geschäftsobjekte und der Benutzeroberfläche im Kontext der Beispielanwendung zur Verwaltung von Zeiterfassungskarten beschrieben wurde, soll in diesem Unterkapitel erläutert werden, welche Codefragmente aus den einzelnen Modellelementen erzeugt werden. Der technische Prozess der Codegenerierung mit Hilfe der Cartridges wurde bereits in Kapitel 3 im Rahmen der Beschreibung der Architektur von AndroMDA beschrieben.

Die Codefragmente der Beispielanwendung werden für die J2EE Plattform generiert, jedoch erfolgt die Betrachtung der Fragmente auf einem hohen Abstraktionsniveau, sodass die vorgestellten Ergebnisse auch auf andere Plattformen übertragbar sind.
Generell wird, wie bereits in Abschnitt 3.1 beschrieben, bei der Codegenerierung zwischen vollständig generiertem Code, der bei jeder Generierung überschrieben wird, und Quellcode mit Einfügepunkten für manuelle Ergänzungen, der nur bei der ersten Generierung erzeugt wird, unterschieden. Der vollständig generierte Quellcode, an dem, auf Grund des Überschreibens bei jeder Generierung, keine manuellen Änderungen durchgeführt werden dürfen, wird im *target*-Zweig des Quellcodebaumens gespeichert, wohingegen die Fragmente, die manuell ergänzt werden müssen, im *src*-Zweig gespeichert werden. Durch diese Auftrennung ist es möglich bei Einsatz eines Versionshaltungssystems nur jene Fragmente unter Versionskontrolle zu stellen, die manuell ergänzt wurden.

Als erstes werden die in Abbildung 3 dargestellten ValueObject-Klassen betrachtet. Da es sich bei diesen Klassen um reine Datencontainer zur Übertragung von Daten an die Präsentationsschicht handelt, an denen keine manuellen Änderungen vorgenommen werden müssen, wird nur eine einfache Klasse im *target*-Zweig generiert, die alle Attribute und Zugriffsmethoden dafür bereitstellt.

Nun werden die ebenfalls in der Abbildung 3 dargestellten Entity-Klassen betrachtet. Diese Klassen stellen die Geschäftsobjekte der Anwendung dar und werden persistiert. Aufgrund dieser Anforderungen müssen pro Entity-Klasse im UML Diagramm mehrere Codefragmente erzeugt werden. Als erstes wird analog der ValueObject-Klassen eine vollständig generierte Klassendatei erstellt, die die Attribute und Zugriffsmethoden dafür bereitstellt. Da das Geschäftsobjekt neben den Attributen auch Methoden bereitstellen kann, die manuell implementiert werden müssen, wird von AndroMDA auf ein gängiges Verfahren der Codegenerierung zurückgegriffen: Neben der automatisch erzeugten Klassendatei im *target*-Zweig wird zusätzlich eine davon abgeleitete Klasse im *src*-Zweig generiert. Diese abgeleitete Klasse wird nur das erste Mal generiert und kann folglich manuell ergänzt und verändert werden. Durch das Überschreiben von Methoden kann somit das Verhalten der Klasse vervollständigt und verfeinert werden.
Neben der Containerklasse selbst ist noch eine Klasse notwendig, die den Zugriff auf Instanzen der Containerklasse steuert, z.B. das Abrufen einer Liste aller verfügbaren Objekte, das Speichern eines Objekts oder das Löschen eines Objekts. Im Kontext von J2EE wird diese Klasse als Data-Access-Object (DAO) bezeichnet. Bei der Erstellung dieser Klassen für die Zielplattform wird das oben erwähnte Verfahren angewandt. Es wird eine automatisch generierte Basisimplementierung im *target*-Zweig erstellt und eine davon abgeleitete Klasse im *src*-Zweig, die durch manuelle Erweiterung das zusätzliche Verhalten bereitstellt.
Schließlich werden noch plattformspezifische Dateien erzeugt, z.B. XML-Deskriptoren, die die Persistierung steuern.

Nachdem nun am Beispiel der Geschäftsobjekte aus Abbildung 3 gezeigt wurde, wie aus Elementen eines UML Klassendiagramms Quellcode erzeugt werden kann, der durch manuelle Erweiterungen verfeinert werden kann, soll nun auf die erzeugten Elemente des UML Aktivitätsdiagrammes aus Abbildung 4(b) eingegangen werden.
Die Erzeugung der Benutzeroberfläche aus einem UML Aktivitätsdiagramm mit AndroMDA besteht grob aus der Generierung von zwei Bestandteilen: Es werden Dateien generiert, die die Sichten auf die Daten beschreiben und Klassen, die für die Steuerung der Oberfläche verantwortlich sind. Die Dateien für die Sichten auf den Datenbestand sind sehr plattform- bzw. technologiespezifisch und sollen deswegen nicht im Detail erklärt werden. Als allgemeine Regel gilt jedoch, dass bei den meisten deklarativen Sprachen zur Definition der Benutzeroberfläche (z.B. XML) keine Vererbung unterstützt wird. Dadurch kann das oben genannte Verfahren, bei dem eine Basisklasse vollständig generiert wird und die Implementierung der erweiterten Logik in einer Unterklasse stattfindet, nicht verwendet werden. In diesem Fall wird die generierte Datei zur Beschreibung der Oberfläche bei der ersten Generierung sowohl in den *target*- als auch in den *src*-Zweig geschrieben. Die Datei aus dem *src*-Zweig kann dann verändert werden und wird dann beim Ausrollen des Projekts durch Maven [13] der Version aus dem *target*-Zweig vorgezogen.
Bei den Klassen zur Steuerung der Benutzeroberfläche werden analog der Generierung der Geschäftsobjekte abstrakte Basisklassen generiert, die dann durch generierte Unterklassen im *src*-Zweig um die Logik ergänzt werden müssen.

4.3 Manuelle Erweiterungen

Nachdem im letzten Unterkapitel beschrieben wurde, welche Codefragmente aus den Modellelementen generiert werden, soll in diesem Unterkapitel zusammengefasst werden, an welchen Stellen manuelle Erweiterungen notwendig sind, damit die Anwendung insgesamt korrekt funktioniert.

Die Geschäftsobjekte werden zu einem großen Teil in voll funktionsfähigen Quellcode umgewandelt. Die einzige Situation, bei der manuelle Implementierungen notwendig werden, ist, falls im Modell für die Geschäftsobjekte statische oder nicht statische Operationen hinzugefügt werden. Statische Operationen beziehen sich auf alle Geschäftsobjekte dieses Typs und müssen folglich in der Datenzugriffsklasse (DAO) implementiert werden, wohingegen Instanzoperationen, also nicht statische Operationen, in der Implementierungsdatei des Geschäftsobjekts ergänzt werden müssen.

Bei Dienstelementen, also Elementen, die über eine Schnittstelle Funktionalitäten bereitsstellen, können lediglich die Schnittstelle und eine leere Implementierung generiert werden. Da die interne Verarbeitungslogik der Dienstklasse nicht im UML Diagramm spezifiziert werden kann, ist es nicht möglich, den Quellcode zu generieren, was zur Folge hat, dass die Implementierung komplett manuell

erfolgen muss.

Das Modell der Benutzeroberfläche kann insgesamt zu einem großen Teil in voll funktionsfähigen Quellcode umgewandelt werden. Die manuellen Ergänzungen des Quellcodes müssen in den Steuerungsklassen durchgeführt werden. Da die Steuerungsklassen eine interne Logik zur Verarbeitung der Benutzeraktionen und zum Aufruf von Diensten besitzen, die nicht im UML Diagramm spezifiziert werden kann, muss diese Logik nachträglich manuell im Quellcode implementiert werden.
Zusätzliche manuelle Änderungen müssen außerdem an den Dateien durchgeführt werden, die das Aussehen der Benutzeroberfläche beeinflussen, um das Erscheinungsbild der Anwendung an die eigenen Bedürfnisse anzupassen.

Insgesamt fällt auf, dass bereits ein Teil des Quellcodes durch Generierung aus den UML Modellen ohne manuelle Ergänzungen komplett lauffähig ist. Eine manuelle Implementierung wird bei der Umsetzung der internen Verarbeitungslogik der Dienstelemente und der Steuerungsklassen der Benutzeroberfläche notwendig, da diese Logik nicht im UML Modell spezifiziert werden kann.

5 Zusammenfassung, Bewertung und Ausblick

Dieses Kapitel soll eine kurze Zusammenfassung der Arbeit geben. Anschließend soll AndroMDA in der vorliegenden Version bewertet werden, bevor schließlich ein Ausblick auf weitere Entwicklungen im Rahmen des AndroMDA-Projekts gegeben wird.

5.1 Zusammenfassung

In der vorliegenden Ausarbeitung wurde auf das MDA-Werkzeug AndroMDA eingegangen. Das auf Java basierende Werkzeug kann dazu verwendet werden, aus UML Modellen lauffähige Komponenten zu erzeugen. AndroMDA besitzt eine Mikrokern-Architektur und verlagert somit die verschiedenen Aufgaben der Codegenerierung in austauschbare Komponenten. Die wichtigsten Komponenten in diesem Plugin-Konzept sind die Cartridges. Sie dienen dazu aus einem PIM den Quellcode für verschiedene Plattformen zu generieren. Hierfür enthalten sie Transformationen und Klassen für den Zugriff auf die Modellelemente. Auf der AndroMDA Webseite [1] stehen Cartridges für mehrere Plattformen, darunter J2EE und Microsoft .NET, zur Verfügung. Selbstentwickelte Cartridges können dem AndroMDA Projekt zur Verfügung gestellt werden und erscheinen dann auch in dieser Auflistung.

5.2 Bewertung

Insgesamt gesehen stellt AndroMDA eine ausgereifte Lösung zum Einsatz der Model Driven Architecture dar [3]. AndroMDA wird dabei bereits auch in Softwareprojekten in der freien Wirtschaft eingesetzt. Zum Beispiel wurde AndroMDA

für die Entwicklung eines Projektplanungssystems im Intranet von Lufthansa Systems erfolgreich verwendet [20]. Einen großen Einfluss auf die Entscheidung über den Einsatz des Werkzeugs hat die angestrebte Zielplattform, für die die Anwendung entwickelt werden soll. Wird die Codegenerierung für diese Plattform bereits von existierenden Cartridges unterstützt, so ist sicherlich ein Effizienzgewinn und eine verbesserte Wartbarkeit zu erwarten. Wird die Generierung für die Plattform von keiner existierenden Cartridge unterstützt, so muss der Aufwand abgeschätzt werden, der bei der Erweiterung existierender Cartridges bzw. bei der Neuentwicklung von Cartridges anfällt. Die komplette Neuentwicklung einer Cartridge erfordert eine Einarbeitung in die Architektur von AndroMDA und lohnt sich erst bei einem größeren Projektumfang. Zusätzlich muss die laufende Pflege der selbstentwickelten Cartridges bei der Projektplanung berücksichtigt werden.

5.3 Entwicklungsumgebung Android

Als Ausblick auf weitere Entwicklungen im Rahmen von AndroMDA ist die integrierte Entwicklungsumgebung Android (= AndroMDA IDE) zu nennen. Diese Entwicklungsumgebung wird als Eclipse-Plugin bereitgestellt und soll vor allem die Bedienung von AndroMDA erleichtern. Hierzu werden grafische Werkzeuge zur Pflege der voneinander abhängigen XML-Deskriptoren und zum Editieren der Velocity-Templates innerhalb der Cartridges bereitgestellt. Schließlich kann die Codegenerierung innerhalb der Entwicklungsumgebung angestoßen werden, wodurch die Aufrufe von Maven aus der Kommandozeile heraus ersetzt werden. Android wird jedoch kein UML-Modellierungswerkzeug beinhalten.
Die Entwicklungsumgebung steht zum Zeitpunkt dieser Ausarbeitung in einer frühen Testversion 0.0.5 zur Verfügung [21].

5.4 AndroMDA 4

Neben der Entwicklungsumgebung Android arbeiten die AndroMDA-Entwickler vor allem an der Version 4 des Rahmenwerks. Die Änderungen und Erweiterungen im Vergleich zu Version 3 sind unter [22] verfügbar.
AndroMDA 4.0 wurde von Grund auf neu entwickelt mit dem Ziel ein leichtgewichtiges Rahmenwerk bereitzustellen, sodass der Benutzer entscheiden kann, welche Komponenten er davon benutzen möchte.
Die Hauptvorteile von AndroMDA 4 sind laut [22] als erstes die Möglichkeit Metamodelle der eigenen domänenspezifischen Sprache (DSL) zu erzeugen und verwenden zu können. Das bedeutet, dass man eigene Metamodelle in einem Plugin bereitstellt und AndroMDA dann beliebige Modelle, die auf diesem Metamodell basieren, laden kann. Dadurch können Domänen, die nicht sehr verständlich und effizient mit UML beschrieben werden können durch ein Modell basierend auf einer eigenen DSL beschrieben werden.
Die zweite große Neuerung besteht darin, dass die Modelltransformationen nun in mehreren kleinen Schritten durchgeführt werden können, was zu einem besseren Verständnis der Schritte führt. Für Model-to-Model Transformationen steht

die Atlas Transformation Language (ATL) bereit, für Model-to-Text Transformationen wird MOFScript benutzt.

Ein großer Nachteil von AndroMDA 4 besteht darin, dass Cartridges, die für AndroMDA 3 entwickelt wurden, nicht kompatibel zur Version 4 sind. Dies liegt daran, dass die Metafacades durch ATL Transformationen ersetzt wurden und MOFScript anstatt der Velocity Template Engine für die Model-to-Text Transformation verwendet wird. Beim Umstieg auf AndroMDA 4 müssen alle selbstentwickelten Cartridges umgeschrieben werden.

Aufgrund der Neuerungen kann sich AndroMDA zu einem Allzweckwerkzeug im Rahmen von MDA entwickeln, wobei beachtet werden muss, dass auf Grund der fehlenden Abwärtskompatibilität der Cartridges ein Umstieg auf die neue Version mit erheblichem Aufwand verbunden sein kann.

Literatur

1. AndroMDA: Projekthomepage. (2007) http://www.andromda.org, zuletzt besucht 10/05/2007.
2. OMG: Model Driven Architecture. (2007) http://www.omg.org/mda/index.htm, zuletzt besucht 10/05/2007.
3. Schulz, D.: MDA-Frameworks: AndroMDA. Seminar Ausgewaehlte Kapitel des Software Engineerings (2005)
4. Petrasch, R., Meimberg, O.: Model driven architecture. 1. aufl. edn. Dpunkt-Verl. (2006)
5. Sturm, T. und Boger, M.: Softwareentwicklung auf Basis der Model Driven Architecture. Dpunkt-Verl. (2003)
6. OMG: Technical Guide to Model Driven Architecture: The MDA Guide. v1.0.1 edn. (2003)
7. OMG: XML Metadata Interchange. (2007) http://www.omg.org/technology/documents/formal/xmi.htm, zuletzt besucht 10/05/2007.
8. Sun: Netbeans Metadata Repository. (2007) http://mdr.netbeans.org, zuletzt besucht 10/05/2007.
9. AndroMDA: Metafacades. (2007) http://galaxy.andromda.org/docs/andromda-metafacades/index.html, zuletzt besucht 10/05/2007.
10. AndroMDA: Translation-Libraries. (2007) http://galaxy.andromda.org/docs/andromda-translation-libraries/index.html, zuletzt besucht 10/05/2007.
11. OMG: UML 2.0 OCL Specification. (2007) http://www.omg.org/docs/ptc/03-10-14.pdf, zuletzt besucht 18/06/2007.
12. Apache: Velocity Projekthomepage. (2007) http://jakarta.apache.org/velocity, zuletzt besucht 10/05/2007.
13. Apache: Maven Projekthomepage. (2007) http://maven.apache.org, zuletzt besucht 10/05/2007.
14. Eclipse: Eclipse Modeling Framework. (2007) http://www.eclipse.org/modeling/emf/, zuletzt besucht 18/06/2007.
15. AndroMDA: 10 steps to write a cartridge. (2007) http://galaxy.andromda.org/index.php?option=com_content&task=blogcategory&id=35&Itemid=77, zuletzt besucht 10/05/2007.

16. W3C: XML Schema. (2007) http://www.w3.org/XML/Schema, zuletzt besucht 10/05/2007.
17. AndroMDA: Getting Started Java. (2007) http://galaxy.andromda.org/index.php?option=com_content&task=category&sectionid=11&id=42&Itemid=89, zuletzt besucht 19/06/2007.
18. Sun: Java EE. (2007) http://java.sun.com/javaee, zuletzt besucht 19/06/2007.
19. AndroMDA: Modeling for AndroMDA. (2007) http://galaxy.andromda.org/docs/modeling.html, zuletzt besucht 10/05/2007.
20. Gebert, T. und Wiese, R.: Model Driven Architecture in der Praxis. Java Spektrum (2005)
21. AndroMDA: Android Update Site. (2007) http://http://www.andromda.org/updatesite/, zuletzt besucht 19/06/2007.
22. AndroMDA: Goals for AndroMDA 4. (2007) http://galaxy.andromda.org/docs-a4/, zuletzt besucht 10/05/2007.

Xpand:
A Closer Look at the model2text Transformation Language

Benjamin Klatt

Chair for Software Design and Quality (SDQ),
Institute for Program Structures and Data Organization (IPD),
University of Karlsruhe, Germany
06. July 2007
benjamin@bar54.de
http://sdq.ipd.uka.de

Zusammenfassung Model-Driven Software Engineering is often constituted as the next level of software development. Since today it is rarely possible to directly execute models they have to be translated into artifacts to be used for further processing. The Xpand language is developed as part of the openArchitectureWare project for such a transformation. This paper provides a overview on the language itself and its context. Furthermore it draws a comparison to other model2text languages. While there is no standardization for model2text transformation yet, the challenge for developers is to find the right language for certain purposes. The main contribution of this paper is to support this selection by delivering a more detailed insight to the Xpand language.

Key words: MDSD, model2text, Xpand,
transformation, openArchitectureWare

1 Introduction

During the advent of software development programmers had to develop code that could be interpreted directly by machines. The next generation were imperative languages like c, which provide some common libraries for system functionality and can be programmed to a large extent independent from the underlying processor architecture. Nowadays state-of-the-art programming languages are object-oriented ones such as Java or C++.

Model-Driven Software Development (MDSD) [14] or Model-Driven Architecture (MDA) as a specialized conduction by the Object Management Group (OMG) are the next level of software development. They focuse more on software design than on its implementation. The main target of this paradigm is to create models in an efficient and domain-specific way. This is accomplished by using domain-specific languages and generating the software from these models. The benefits of this development process are efficiency, quality, maintainability and the focus on the solution.

Currently most software systems still require compiled code that is executed on the software platform. Therefor code or text generation is an essential step in the MDSD process. Textual transformations are nearly as old as the software development discipline itself. As a result there are many solutions available today and also some which are able to process conceptual models. Someone who wants to start with MDSD has to find the right one for their own requirements. It soon becomes clear that efficient and reliable model2text transformation requires more than simple text replacement.

The openArchitectureWare platform [9] is one of the leading MDSD tools. It is developed as an open source project led by a core team who has long time experience in model-driven software development. The platform has been developed to provide a basis that can be used in practice with the current state-of-the-art development technologies. [15]

Xpand is the domain-specific model2text transformation language of the project. Templates are written in this language and used by the generator that is linked to the development workflow of the platform. One of the purposes of this paper is to give a closer look at Xpand, its concepts and its environment to help answering the question of whether Xpand is the right solution for a specific project or not. In the following a short introduction to the language will be given. Furthermore it will be compared to some related languages and plattforms to be able to rate its features.

2 The Xpand Language

2.1 Purpose and Development

Xpand was developed from scratch as a part of the openArchitectureWare platform (oAW). [9]. While there are already many template languages available, the authors of the framework have realized that effective template development is

only possible with an easy-to-learn, domain-specific language (DSL) for code generation, as well as with good tool support. With this focus, the Xpand language itself has a small but sufficient vocabulary [5]. Beside of its own capabilities, it can access functions implemented in the Xtend programming language, which is another domain-specific language that is contained in the oAW framework. Those functions can vary from simple utilities to complex model calculations.
All of the Xpand developers are practitioners and have their focus on the application of the tools [15]. So while there might be a lot of things that are self-evident for language designers, the framework is coined by a lot of experience that comes from working in projects.

2.2 Application

Workflow The openArchitectureWare system is based on a workflow engine which executes different processing steps, like model instantiation, validation, model2model and model2text transformations as well as post-processing steps. Figure 1 shows an example workflow. The components can be configured arbitrarily so that it is possible to parse multiple models and combine them into an internal abstract syntax graph. Also, multiple transformers for model2model transformations with a validation step after each transformation can be configured, as well as multiple generators for different target artifacts.

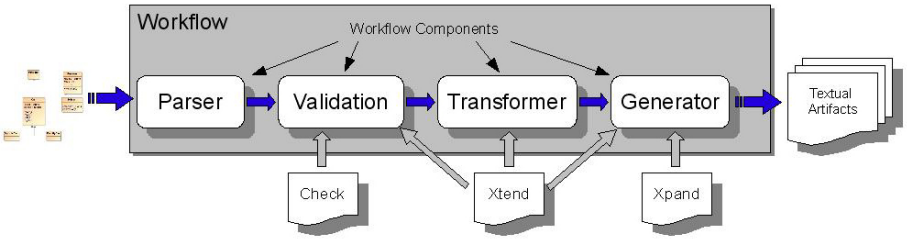


Abbildung 1. oAW example workflow

In openArchitectureWare workflow the code generation (or model2text transformation) step will be linked to the workflow definition. A workflow is defined in an XML descriptor file. A generator can be configured as shown in Example 1. First of all, the generator class is defined in the component tag. Inside this tag the required meta models are referenced so that they can be accessed by the generator. Then, the main template that will be processed is referenced. The outlet tag sets the target directory for the generated artifacts. It includes the JavaBeautifier which reformats the generated code according to predefined format rules.

Example 1.

```
...
<component id="generator"
           class="org.openarchitectureware.xpand2.Generator2">
  <metaModel idRef="emf"/>
  <metaModel idRef="uml2"/>
  <metaModel idRef="profile"/>
  <expand value="Template::define FOR mySlot"/>
  <outlet path="main/src-gen">
    <postprocessor
    class="org.openarchitectureware.xpand2
              .output.JavaBeautifier"/>
  </outlet>
</component>
...
```

Xpand itself is independent from the type of the source model. Different source models are handled by a parser linked to the openArchitectureWare workflow. Parsers can be written for any kind of source model but openArchitectureWare provides out-of-the-box parsers for EMF, Eclipse-UML2, different UML-Tools (MagicDraw, Poseidon, Enterprise-Architect, Rose, XDE) and textual models using the Xtext framework as well as for XML and Visio.

Tool Support One of the biggest benefits of the 4.x generation of the openArchitectureWare is the extended tool support. It is to a large extent based on the Eclipse Rich Client Platform (RCP) [8]. One of the plug-ins developed for this environment is the Xpand editor (figure 2). This editor provides state-of-the-art programming support such as code completion and syntax highlighting. Furthermore it has a built-in metamodel utilization, which means that custom metamodels are supported by the editor if they are accessible (i.e. if the editor has access to your UML2 profile).

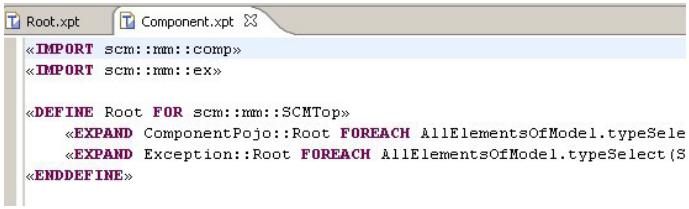


Abbildung 2. Screenshot of the Xpand Editor

2.3 Template Files

Xpand templates are composed of one or more template files storde as textual documents with the file extension *.tpl*. The generator linked to the openArchitectureWare workflow references one of these templates that acts as a point-of-entry and references other templates to execute them. The templates can be organized within packages that are physically stored in directories and subdirectories in almost the same way Java packages are. This provides possibilities for reuse and organization.

2.4 Vocabulary

As mentioned in section 2.1 the target of the language design was to provide an easy-to-learn language. With this in mind, a very small and intuitively understandable vocabulary was developed. The following is a short overview of the vocabulary. Please refer to the Xpand reference for further information [5].

Escaping
Every template language needs some characters to separate template expressions from static text fragments for the output. It is a best practice [15] to use escaping characters which do not occur in the generated artifacts. For the Xpand language the French quotation marks (■guillemots■) have been chosen.

General Template Structure
The template in example 2 imports a model definition, loads an extension and defines a template section for a simple Java class that is applied to elements of type *Entity*. In the following sections a short overview of the different directives is given.

Example 2.

```
«IMPORT meta::model»
«EXTENSION my::ExtensionFile»

«DEFINE javaClass FOR Entity»
  «FILE fileName()»
    package «javaPackage()»;
    public class «name» {
      // implementation
    }
  «ENDFILE»
«ENDDEFINE»
```

Basic Template Elements

DEFINE
This directive marks a template definition. A template has a name and is always

assigned to a specific element type. Collection types are allowed, too. The type definition is also used for template polymorphism. As an example, if a template should be overridden for elements of a subtype of the type *Entity* in the example 2 the DEFINE section will be declared with the subtype instead of *Entity*.

IMPORT
This gives access to all templates in a specific namespace. In example 2 the identifier *Entity* is used instead of the fully-qualified name *meta::model::Entity*.

EXTENSION
This directive loads extensions written in the Xtend language. Extensions can reach from simple utilities to complex model calculations.

Control Flow

EXPAND
This directive is used to execute a DEFINE block for the specified element type. (See Example 3).

Example 3.

```
..
«DEFINE javaClass FOR Entity»
  ..
  «EXPAND methodSignature FOR this.methods»
  ..
«ENDDEFINE»

«DEFINE methodSignature FOR Method»
  ...
«ENDDEFINE»
..
```

FOR and FOREACH (within EXPAND)
These directives are used to apply a template on a collection of elements but in slightly different ways.
FOR simply executes the referenced template for each element resulting from the given expression as shown in example 3. This might also be a single element.
FOREACH instead always requires a collection to work on and therefor provides the capability to define a separator that is added between the output of the called template for each element. (see example 4).

Example 4.

```
  «EXPAND paramTypeAndName FOREACH params SEPARATOR ","»
```

FOREACH (single occurrence)
If foreach is used as a directive, it defines a loop control flow as in many other programming languages. Example 5 shows such a loop with the available options.

Example 5.

```
«FOREACH expression AS variableName [ITERATOR iterName]
                                    [SEPARATOR expression]»
  a sequence of statements using variableName to
  access the current element of the iteration
«ENDFOREACH»
```

IF
IF is used as in many programming languages to mark a block that is only executed if the specified condition is evaluated to true.

File Generation

FILE
The file directive marks a section which specifies the file to which the output of the templates should be written.. In the basic template structure example in example 2 the directive is used to write a distinct file for each generated Java class. In this example an Xtend function is called to compute the name of the file to write to.

Protected Regions

PROTECT
A general problem with generated code is to protect manually added code in the generated artifacts of being overwritten when the generation is executed again. Xpand provides so called protected regions which are marked by the PROTECT directive for this purpose.

Miscellaneous

LET
The LET operator is used to declare variables and their content. LET is designed to define a block in which the result of an expression is bound to a variable. In Example 6 the fully qualified name of a class is stored in a variable named fqn.

Example 6.

```
«LET packageName + "." + className AS fqn»
  the fully qualified name is: «fqn»;
«ENDLET»
```

ERROR
The error directive can be used to stop the template processing. Even if it is possible with this directive it is recommended to implement model checks which are executed earlier in the workflow using the Check language of openArchitectureWare. [4]

COMMENTS
Comments look similar to those used in DOS and Windows batch scripts. (see example 7)

Example 7.

```
«REM»Comment«ENDREM»
```

WHITESPACES
In template languages it is always necessary to have influence on the generated whitespaces. Otherwise it could happened that the generated artifacts are flooded with redundant whitespaces and empty lines. Xpand provides the possibility to mark directives with a minus sign if they should not generate a new line or whitespaces. The code in example 8 produces only one line.

Example 8.

```
«IF hasPackage-»
  package «InterfacePackageName»;
«ENDIF-»
```

AOP
Aspect Oriented Programming (AOP) is a common paradigm to couple crosscutting aspects in the implementation. Xpand also supports this paradigm for the template definition with the AROUND syntax. By this, existing templates can be extended. As shown in Example 9 point cuts define the elements that should be enhanced with the aspect definition.

Example 9.

```
«AROUND [pointcut]»
  do stuff
«ENDAROUND»
```

2.5 Language Features

Czanecki and Helsen have designed a feature model for model transformation approaches. The model is shown in figure 3 [2]. This model is a good guideline to get an overview of the concepts used in a model transformation language. The following sections discuss how Xpand can be categorized according to this model.

Transformation Rules The transformation rules feature consists of multiple sub features. Figure 4 shows the subtree of the model.

In the case of XPand the target or so called out-domain always consists of textual artifacts which could belong to a specific programming language. The source domain handled by Xpand is the abstract syntax graph that was instantiated by the openArchitectureWare workflow and handed over to the generator. This is also described as the in-Domain. The abstract syntax graph is an intermediate structure of oAW. Xpand itself does not use any intermediate structure but it

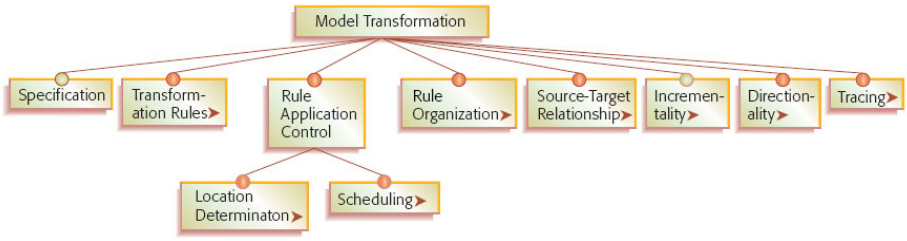


Abbildung 3. Feature Model Overview taken from [2]

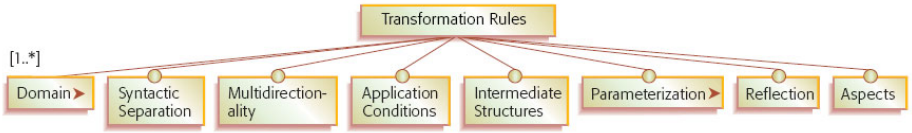


Abbildung 4. Transformation feature model - Transformation rules subtree [2]

becomes decoupled from the type of the source model by linking an appropriate parser to the oAW workflow.

Xpand provides a simple unidirectional transformation. The only syntactical separation in the template definitions is between the template code and the static content.

Application conditions are only provided in form of the IF directive within the templates. Parametrization is possible in a restricted way in the generator configuration by defining the template that is used as point-of-entry and can hold parametrization settings. A real properties file can only be loaded and used in the workflow definition.

In the template definition Xpand provides access to the source metamodel. Working on this meta level is some kind of reflection access to the source model. Aspect orientation is an explicitly supported feature of Xpand. This already has been discussed in the in the Vocabulary section.

Rule Application Strategy Xpand has a strict strategy based on template references and polymorphism rules. This concept ensures that the rules are applied in a deterministic way.

Rule Scheduling Xpand rule scheduling can be defined by the user in an external explicit form by building a template hierarchy and referencing from one template to another. The scheduling can also be defined implicitly by using the polymorphism capabilities of the language. So the templates will be executed in a deterministic order starting from the first template referenced by the generator. This concept protects against any scheduling conflicts. With the foreach

directive a looping construct is provided. Depending on the source meta-model it is possible to call templates recursively as well. Czarnecki and Helsen define an XOR relationship in their feature model for the looping and the recursion features but one does not exclude the other as seen in the case of Xpand. The last feature considered in the subtree is the phasing. Phasing is about the structure of the transformation workflow. Xpand is used within the openArchitectureWare workflow which handles the phasing and the templates are only executed.

Rule Organization Xpand has a modularisation mechanism which enables to organize the templates in packages and to import complete template namespaces. The packages can be defined flexible but the DEFINE directive is coupled to the element type that is processed. So it is more coupled to the source then to the target structure. Additionally to this mechanism template polymorphism is supported and by this Xpand supports all reuse techniques described by Czanecki and Helsen.

Source-Target Relationship This relationship is about the target and source models required by the language. For model to text transformations the target is always a textual model. The source model for the complete oAW workflow depends on the linked parser. Xpand itself always works on an abstract syntax graph provided by the workflow but there is no relationship between this and the generated model.

Incrementality Xpand transfers a model from an abstract syntax graph to a textual representation. The rules defined for abstract syntax graphs can not be applied to text fragments without instantiating a new syntax graph out of them. This could be done with additional steps in the oAW workflow but there is no possibility to transfer the model incremental within one generator run.

Tracing There is no direct support for tracing but it could be realized with the Aspect Oriented Programming support.

Directionality As the most model2text transformation languages Xpand does not provide round trip functionality and performs an unidirectional transformation.

2.6 Perspective

On the roadmap of the openArchitectureWare project an issue for version 5 is a new version of the Xtend language. The new Xtend++ should unify the languages Xtend, Check and Xpand [10].
It is unclear which impact the OMG model2text language specification [7] will have on the language. It defines a language on itself which has to be supported by tools to become certified to the standard. It is not a general specification how model2text languages should be designed.

3 Comparison to other model2text Languages

3.1 Preface

There are many languages available which are able to produce text artifacts out of different model definitions. As with most technologies, the best solution depends on the concrete requirements. This comparison gives attention to some representative languages and procures key factors to keep in mind when a solution is evaluated. Further more it exposes advantages and disadvantages of Xpand over other languages which come up when it is compared to languages which are not covered here.

3.2 Considered Languages

To limit the number of languages to a reasonable set, the languages mentioned in the Eclipse model2text project [6] are handled in this section. The Website lists the following ones:

- JET (Java Emitter Templates)
- MTL (Model to Text Language)
- M2T (Model to Text)

Furthermore the Object Management Group (OMG) currently develops their own language specification for model2text transformation. This specification will be discussed in detail in a later section.
Simple text replacement approaches like XSLT are not covered here. They provide less functionality than JET and in comparison to Xpand they come up with a worse result.

JET (Java Emitter Templates) is an open source component like the openArchitectureWare platform. It provides a framework and facilities for code generation. Java Server Pages (JSP) like template files are used and by this it makes it easy to learn for developers already familiar with this technology. It is easy to extend with custom tags similar to the same concept which already exists in JSPs.

Tool Support
JET is strongly supported by tools integrated in the Eclipse IDE. A transformation specific project can be created in Eclipse and provides all required files. Furthermore a JET-specific editor and a launch configuration to start a transformation run are provided.

Model Access
JET by default requires an XML file as input. This does not have to be any specific XML format and does not have to be related to software modeling. XPath (XML Path Language) [17] is used to access different nodes in the XML source and to navigate between the elements. XPath is a wide-spread language so there are many developers already familiar with it.

When a JET template is developed and saved in the JET editor a Java class is generated out of it. This template class can be called directly from other Java code. So it is possible to program a custom parser that prepares the source model to be transformed by the template class.

Template definition and execution
A JET transformation can be imagined as a blueprint which is applied to a source model. The source model is sometimes just called parameter set because it defines the values which should be used when the blueprint is applied. A JET transformation can be illustrated as: *Parameters + Blueprint = Desired Artifacts* [1].

Example
A simple transformation source that defines different persons can be written as in example 10.

Example 10.

```
<app middle="Hello" >
  <person name="Chris" gender="Male" />
  <person name="Nick" gender="Male" />
  <person name="Lee" gender="Male" />
  <person name="Yasmary" gender="Female" />
</app>
```

Example 11 is a transformation template to generate a class for a person element.

Example 11.

```
class <c:get select="$currPerson/@name" />Person {
  public String getName() {
    return "<c:get select="$currPerson/@name" />";
  }
  public void shout() {
    System.out.println("Hello!!!");
  }
}
```

The transformation processing is defined within a template file as in Example 12. The process iterates over all person elements in the source XML and executes the PersonClass Java template. The results are written into separate Java files.

Example 12.

```
<c:iterate select="/app/person" var="currPerson">
<ws:file template="templates/PersonClass.java.jet"
  path="{$org.eclipse.jet.resource.project.name}/
        {$currPerson/@name}.java"/>
</c:iterate>
```

Discussion
JET is based on existing and established technologies. It is easy to find developers who are already familiar with them. A project is very easy to be set up within a short time. On the other side there is no domain-specific support for the model-driven software development. For example it is not possible to validate a template when it is saved like the Xpand editor does. The state-of-the-art MDSD targets quite complex models and for larger projects it becomes hard to handle the complexity without extensive tool support for this domain.

MTL is an implementation of the OMG model2text language standard. It is still in the development phase and nothing has been released yet. The comparison results for MTL should be nearly the same as in the comparison with the OMG standard presented later in this paper.

M2T can not be seen as a competitor to Xpand. The M2T (model2text) is an eclipse project with focus on model2text transformation. It is not a transformation language on its own but provides basic components which can be used by other projects like JET, Xpand or MTL. There are two main components called M2T core and M2T shared. While the former is an invocation framework for transformations the latter is an infrastructure component for different transformation languages.

3.3 Object Management Group (OMG) Standardization

While the OMG works hard on standardizations in the Model-Driven Architecture (MDA) environment, they investigate in the model2text area, too. In 2004 there was the first Request for Proposal (RFP) of a 'Meta Object Facility Model to Text Transformation Language'. For this RFP a revised submission has been published in August 2006 [7]. Because there are a lot of standardizations successfully established by the OMG and their specification is quite new, it makes sense to take a closer look at this standard in comparison to Xpand.

Basic Language Concept The OMG submission contains a language specification developed by the submitting companies. One of the basic ideas is to reuse already existing standards as much as possible. Especially MOF 2.0 and QVT are used. Tools have to adopt the language to get certified with a specific conformance level. As described in the following section there are some differences between Xpand and the OMG model2text language. Tools which conform to the OMG standard are not conform to Xpand as well.

Syntax The OMG language syntax is nearly as simple as the one of Xpand. The transformations are defined as so called templates which can be organized as modules in different namespaces. Also it contains many basic language elements such as *Template*, *For* and *If*. The elements which are handled in the same way

as in Xpand are not handled in this paper. Please refer to the OMG reference [7] or the Xpand [5] reference for further information about these elements.

Escape Direction
The template definition can be defined in two different ways: text-explicit or code-explicit. In the first style, the static text for the output is written as it is and the template tags are escaped (see Example 13).

Example 13.

```
@text-explicit
[template public classToJava{c: Class}]
  class [c.name/]{
    ...
  }
[/template]
```

In the code-explicit style, the template code and static text are escaped the other way around but with different markers as in example 14.

Example 14.

```
@code-explicit
template public classToJava{c: Class}
  'class 'c.name'{
    ...
  }'
/template
```

This is flexible and in some cases the templates may be shorter but it requires tools and developers to read different languages. This is also true if non-standard escape characters are used for the code-explicit style. Xpand only offers the text-centric style and requires to handle only one syntax.

Template Evaluation
The OMG specified that a module can extend other ones. If this is used a template is able to override other templates. This polymorphism is like the one available in Xpand. Besides of this, the OMG specification has an extended handling for parameter collections. If the template definition requires a parameter of type string and is referenced with a collection of strings it will be executed once for each string of the collection. This is an easy way to make the programming of loops unnecessary. As a drawback this could lead to unexpected results. If the required input parameter is a collection of objects and the template is called with a collection of collections it is undefined whether the template is executed once for all or once per inner collection.

White Space Handling
Xpand offers just one straight way to influence the handling of whitespaces. The

OMG syntax instead does not offer any way to manupulate whitespaces. For each language element it is defined how whitespaces are generated.

Queries
The OMG specification implements a directive to query the processed model. This makes it possible to collect elements which are not at the same place in the model. Example 15 shows an Object Constraint Language (OCL) query to build a collection of attributes to be processed by a template. Xpand only offers the possibility to access a collection of objects in one path. Everything more complex has to be handled by Xtend functions.

Example 15.

```
[query public allOperations(c: Class):Set ( Operation )
  = c.operation->union( c.superClass
    ->select(sc|sc.isAbstract=true)
    ->iterate(ac : Class; os:Set(Operation)
  = Set{}| os->union(allOperations(ac))) /]
```

Protected Areas
The OMG specification provides so called protected areas for text manually added to generated artifacts which should not be overwritten when the transformation is processed again. The areas have to be marked in the generated artifacts. How they have to be marked is not be specified in the template and according to the specification this has to be handled by the tool implementing the standard.

Xpand provides protected regions for this purpose. Xpand protected regions have the possibility to specify the markers that should be used in the generated text. This makes the tool independent from the target syntax of the artifacts. The OMG requires the tool to know how comments are formatted in the target language.

Macros
The OMG specification includes the definition of macros which could be reused in the templates. In example 16 a simple macro which produces a comment is defined and applied.

Example 16.

```
  [macro comment (Body b)]/**[b/]*/[/macro]

usage:

  [comment()]
    ...
  [/comment]
```

File Output
The OMG file definition is nearly the same as the one provided by Xpand. The main difference is an option to append the new generated text to the target file if it already exists. The specification mentions the production of log files as a purpose of this feature.

Discussion The OMG specification shows a lot of overlaps to the Xpand language. These overlaps identify a common set of reasonable features. But the big difference is that Xpand is designed to be as small and straight forward as possible. The small vocabulary is easy-to-learn and challenges are solved as easy as possible. The OMG however shows some conceptual inconsistencies. They provide a flexible template syntax by defining configurable escape characters for the code-explicit language style but on the other hand they force the processing tool to know the comment style of the target languages to mark protected regions.

To sum this up, in the combination with the tightly integrated Xtend language, Xpand is at least as powerful as the OMG specification but much more straight forward to learn and to work with.

3.4 Advantages of Xpand

Domain-Specific Language Unlike many other solutions Xpand is a domain-specific model2text transformation language for the model-driven software development. It is a static-typed language and offers better facilities to access model structures and to write and organize templates. This is an essential requirement for professional MDSD. [15]

Source Model Support Xpand does not care about the source model because it processes the abstract syntax graph provided by the openArchitectureWare workflow. It depends on the linked parser which reads in the model. openArchitectureWare comes with out-of-the-box parsers for common software models. Refer to section 2.2 for a list of the included parsers.

Utilization Since the 4.x generation the openArchitectureWare platform is available as a prepared distribution which includes eclipse and all required plug-ins. In the latest version there is an extensive good tool support for template and workflow development as well as for their execution.

Easy Start-Up The language vocabulary is easy to learn and with its tool support someone can get used to it in a short period of time. This assessment is only about Xpand and no other component related to it like Xtend or oAW as a whole.

Practical Experiences Xpand has been developed by a development team with a lot of practical experiences not only in the model-driven software development but in the general software development area as well. The language has proven itself value in tedium applications. Also it has been chosen by some projects where other languages like JET have come to their limitation, mostly because of structural purposes. [11]

Strong Community Even if it is not big yet the openArchitectureWare has a potential and healthy community. Answers are given fast and professional in the forum and the documentation provides a lot of tutorials, references and screencasts. [9]

3.5 Disadvantages of Xpand

Complexity As a part of openArchitectureWare, the concept and the appliance of the platform has to be understood to set up a transformation project. There are smaller frameworks like JET with a shorter ramp-up time.

Future plans As mentioned in the perspective section there are plans to integrate Xpand into a new version of the Xtend language. Even if the core developers are experienced with real world projects it can not be excluded that there will be an effort required to refactor all templates for the next major release.

4 Conclusion

There are many text generation languages and frameworks available. Some with a more general purpose and others strongly connected to the model-driven software development. Xpand is one of the tight connected ones. The language itself is easy to learn with a small vocabulary but is integrated with more powerful languages to handle quite complex requirements.

As shown in the advantages and disadvantages section Xpand is a good choice for model-driven software development. It surpass many other languages because of its domain-specific features and the practical experience in well-engineered MDSD projects. The openArchitectureWare platform is still young and complex. As with nearly every technology it depends on the concrete case whether Xpand is the right choice or not. For small projects that are not intended to grow and for which it is more important to have a short start-up time Xpand and the openArchitectureWare platform may not be the right choice. Only if the development team is very familiar with it and is able to set it up as easy as a JET project or another simple framework it would make sense to chose Xpand for small and limited projects.
A similar reason is a high fluctuation of the development team if the developers are not in use with the openArchitectureWare but with technologies used by

other frameworks like JET. In this case it may be better to choose a solution the involved developers already have the required knowledge for.

Nevertheless in the end most projects that decide for a model-driven software development approach have the claim to do this on a high and professional level. For those it makes sense to start with openArchitectureWare and Xpand because of the long time investment and to ensure that the development does not run into complexity problems. [11]

5 Appendix

Literatur

1. Chris Aniszczyk and Nathan Marz. Create more better code in eclipse with jet. Internet, 2007. http://www.ibm.com/developerworks/opensource/library/os-ecl-jet/, last access 06/07/2007.
2. Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.
3. AOSD Steering Committee. Aspect-oriented software development community. Internet, 2007. http://www.aosd.net/, last access 06/07/2007.
4. Sven Efftinge. *Check - Validation Language, Reference Documentation*, 2006.
5. Sven Efftinge and Clemens Kadura. *OpenArchitectureWare 4.1 Xpand Language Reference*, 2006.
6. The Eclipse Foundation. Eclipse modeling - m2t - home. Internet, 2007. http://www.eclipse.org/modeling/m2t/, last access 06/07/2007.
7. Object Management Group. *Revised submission for: MOF Model to Text Transformation Language (ad/2004-04-07)*, 2006.
8. Jeff McAffer and Jean-Michel Lemieux. *Eclipse rich client platform*. Addison-Wesley, 2006.
9. openarchitectureware. Official openarchitectureware homepage. Internet, 2007. http://www.openarchitectureware.org, last access 06/07/2007.
10. openarchitectureware. openarchitectureware roadmap. Internet, 2007. http://wiki.eclipse.org/index.php/OAW_Roadmap, last access 06/07/2007.
11. openarchitectureware. openarchitectureware success stories. Internet, 2007. http://openarchitectureware.org/index.php?topic=success, last access 06/07/2007.
12. Remko Popma. Jet tutorial part 1 - introduction to jet. Internet, 2004. http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html, last access 06/07/2007.
13. Remko Popma. Jet tutorial part 2 (write code that writes code). Internet, 2004. http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html, last access 06/07/2007.
14. Thomas Stahl and Markus Völter. *Model driven software development*. Wiley, 2006.
15. Markus Völter and Bernd Kolb. Best practices for model-to-text transformations. In *Eclipse Summit Europe 2006*. EclipseCon, 2006.
16. Markus Völter. Modellgetriebene entwicklung von eingebetteten systemen. In *OOP 2007, Software meets Business*. SigsDatacom, 2007.
17. World Wide Web Consortium W3C. Xml path language - xpath. Internet, 2007. http://www.w3.org/TR/xpath, last access 06/07/2007.

Model-to-Model Transformationen in openArchitectureWare

Beyhan Veliev

Betreuer: Thomas Goldschmidt, Henning Groenda

Zusammenfassung Modellgetriebene Software-Entwicklung (Model Driven Software Development, MDSD) wird in den letzten Jahren immer populärer. Eine sehr wichtige Rolle bei dieser Art von Software-Entwicklung spielt die Transformation eines Modells auf ein anderes Modell oder ein textuelles Artefakt.
Diese Ausarbeitung gibt zunächst eine kurze Einführung in die MDSD und Model-to-Model Transformation. Danach wird openArchitectureWare (oAW), ein Werkzeug für MDSD, vorgestellt. Im nachfolgenden Kapitel wird erklärt, wie eine Model-to-Model Transformation in oAW abläuft und anhand eines Beispiel veranschaulicht. In diesem Kapitel wird auch Xtend, die Model-to-Model Transformationssprache von oAW, näher betrachtet. Abschließend folgt eine Einordnung von oAW bezüglich bereits vorhandener Transformationsansätze, die in Artikel [1] beschrieben sind, gefolgt von einer Zusammenfassung und einem Ausblick in die Zukunft.

1 Einleitung

Dieses Kapitel soll die grundlegenden Begriffe und Konzepte in der Welt der MDSD verdeutlichen. Zuerst wird ein kurzer Blick auf die historische Entwicklung der objektorientierten Software-Entwicklung (OOSE) geworfen. Danach wird der MDSD-Ansatz und am Ende des Kapitels auf Model-to-Model Transformationen eingegangen. Das nächste Kapitel behandelt openArchitectureWare (oAW), ein Werkzeug für MDSD. Im nachfolgenden Kapitel wird erklärt, wie eine Model-to-Model Transformation in oAW abläuft und in einem Beispiel veranschaulicht. In diesem Kapitel wird auch Xtend, die Model-to-Model Transformationssprache von oAW näher betrachtet. Abschließend folgt eine Einordnung von oAW bezüglich bereits vorhandener Transformationsansätze, die in Artikel [1] beschrieben sind, gefolgt von einer Zusammenfassung und einem Ausblick in die Zukunft.

1.1 Objektorientierte Software-Entwicklung

Wie man in Abbildung 1 leicht erkennen kann, wurde die erste objektorientierte Sprache Simula noch in den 60er Jahren erfunden. Simula hat die objektorientierte Sichtweise in die Softwareentwicklung eingeführt [3]. Die große Anzahl an objektorientierten Sprachen, die nach Simula entstanden sind, sind ein Zeichen dafür, dass diese Art der Software-Entwicklung mit den Jahren sehr bekannt und

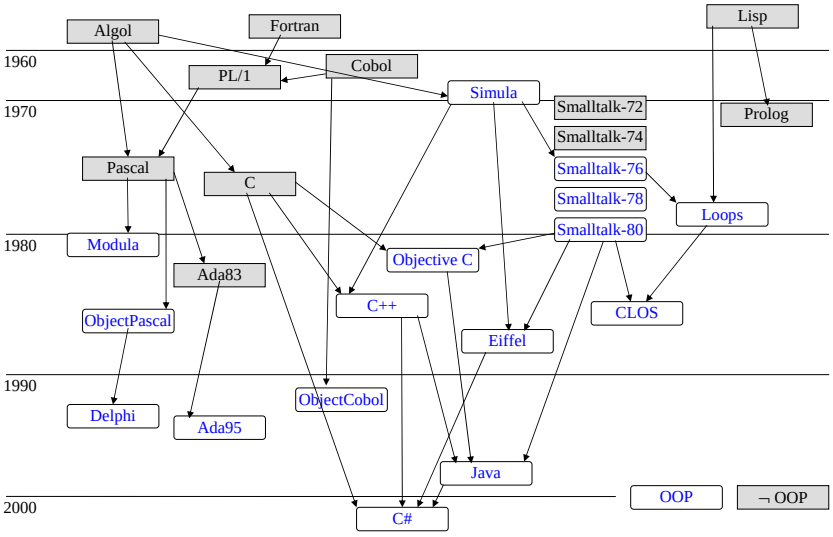


Abbildung 1. Historie der Programmiersprachen. Objektorientierten Sprachen sind durch blaue Schrift markiert (aus [2, S. 99])

beliebt geworden ist, weil die Abbildbarkeit von realen Objekten auf Programmcode „wie menschliches Denken die reale Welt begreift“ [3] möglich geworden ist.

Mitte der 90er Jahre wurde die Unified Modeling Language (UML) von Grady Booch, James Rumbaugh und Ivar Jacobson entworfen. Es sollte mit eingebracht werden, dass es sich bei UML um eine abstraktere Ebene handelt, bzw. dass man mit ihr noch besser die reale Welt in einem Formalismus angeben kann und diese dann leichter auf objektorientierten Programmcode abbilden kann. Dadurch wurde auch eine effizientere Kommunikation zwischen den Beteiligten am Software-Entwicklungsprozess geschaffen. Diese Sprache ist auch eine wichtige Grundlage für die Entstehung von MDSD, die im nächsten Kapitel betrachtet wird. Seit 1997 wird UML unter der Führung der Object Management Group (OMG) als Standard weiterentwickelt. Die historische Entwicklung von UML ist in Abbildung 2 dargestellt.

1.2 Modellgetriebene Software-Entwicklung

In diesem Kapitel werden zuerst grundlegende Begriffe eingeführt und anschließend das Prinzip von MDSD erklärt. Zum Schluss wird auf die Vorteile von MDSD eingegangen.

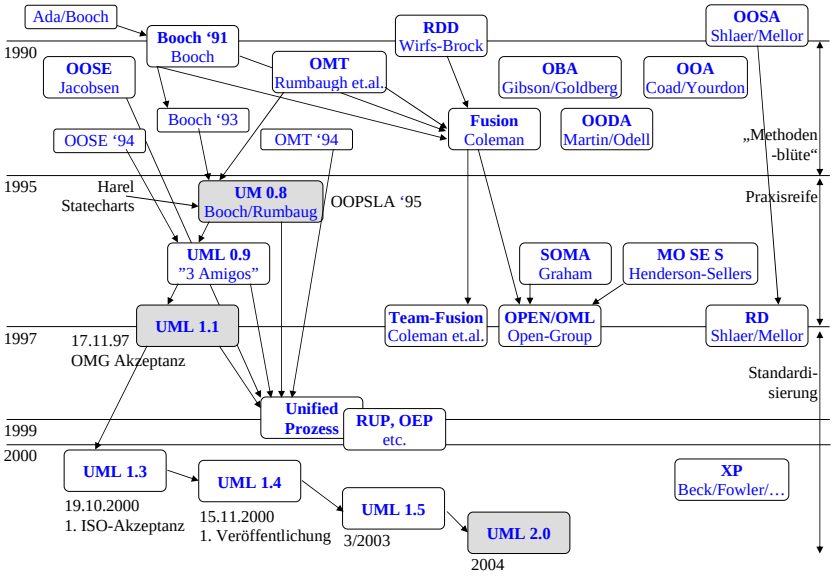


Abbildung 2. Historie der UML (aus [2, S. 100])

Wichtige Begriffe und das Prinzip. Wie man an dem Namen MDSD bereits erkennen kann, spielen Modelle bei dieser Art von Software-Entwicklung eine zentrale Rolle.

Definition 1. *Ein* Modell *ist eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems [4].*

Definition 2. *Ein* Metamodell *dient zur Beschreibung der Elemente eines Modells. Ein Metamodell kann wiederum selbst durch ein Metametamodell definiert sein, oder sich selbst definieren. Ein Beispiel für ein selbstbeschreibendes Metamodell ist der MOF-Standard [5].*

Mit UML 2.0 wurde eine formale Beschreibung von Domänen durch *UML-Profile* eingeführt. Ein *UML-Profile* kann aus Stereotypen, Tagged Values, Constraints und Custom Icons bestehen.
In diesem Zusammenhang wurde das *XML Metadata Interchange (XMI)* um das Format „UML 2.0 Diagram Interchange" erweitert. Das Konzept hinter XMI ist, UML Diagramme in diesem Format von UML-Werkzeugen zu exportieren, so dass sie direkt von anderen Werkzeugen, die dieses Format unterstützen weiterverarbeitet werden können.

Der Begriff *Plattform* wird im Zusammenhang mit MDSD auch oft erwähnt. MDSD konkretisiert den Abstraktionsgrad von *Plattformen* nicht. Eine Plattform könnte die Java-Umgebung sein, oder sogar ein Rechner. Modelle, die für eine bestimmte *Plattform* entwickelt sind, werden als *Plattform abhängiges Modell* (PSM = Platform Specific Model) bezeichnet. Modelle, die sehr abstrakt

sind und keine Plattformspezifikationen enthalten, werden als *Plattform unabhängiges Modell* (PIM = Platform Independet modell) bezeichnet.
Zum Schluss wird noch der Begriff Transformation, analog zu [4], definiert:

Definition 3. Transformationen *bilden Modelle auf die jeweils nächste Ebene ab. Dies können weitere Modelle oder auch Sourcecode sein. Im Sinne der modellgetriebenen Entwicklung müssen Transformationen flexibel und formal auf Basis eines gegebenen Profils definiert werden können.*

In Abbildung 3 werden die erwähnten Begriffe nochmal graphisch zusammengefasst.

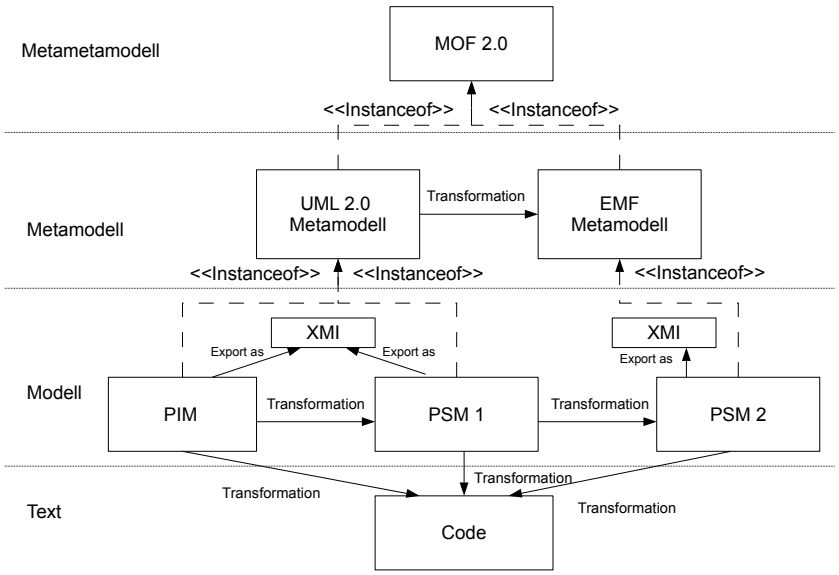
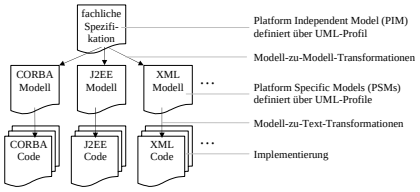
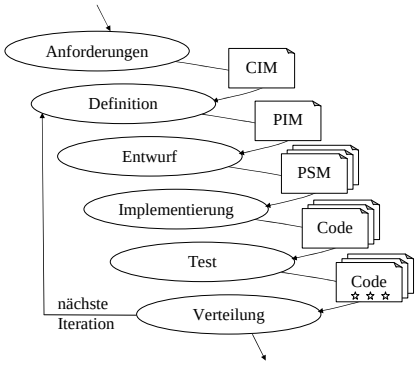


Abbildung 3. Einordnung der verwendeten Begriffe

Im Folgenden wird das Prinzip von MDSD erklärt. Bei dieser Vorgehensweise wird als erster Schritt die domänenspezifische Sprache (Domain Specific Language, DSL) mit Hilfe von eigenen Metamodellen oder UML-Profilen definiert. Dann kann das ganz abstrakte PIM entworfen werden, das eine Instanz der im ersten Schritt definierten DSL ist. Danach kommen die Transformationen ins Spiel. Bei MDSD haben wir zwei Arten von Transformationen. Bei der Transformation von einen PIM oder PSM in ein PSM handelt es sich um eine Model-to-Model Transformation, auf die in den nächsten Kapiteln genauer eingegangen wird. Die Transformation eines PIM oder PSM auf Code ist eine Model-to-Text Transformation. In Abbildung 4 wird dieser Prozess nochmals graphisch gezeigt und in Verbindung mit dem modellgetriebenen Software-Entwicklungsprozess gesetzt.



(a) Grundprinzip von MDSD



(b) Software-Entwicklungsprozess bei MDA

Abbildung 4. Modelle im vorgestellten Software-Entwicklungsprozess (aus [2, S. 102f])

Warum modellgetriebene Software-Entwicklung. Diese Frage ist berechtigt, da es Modelle und Code-Generatoren auch bereits vor MDSD gab. MDSD hat zum Ziel, einen möglichst großen Anteil des Programmcodes bei einer Anwendung aus einem PIM abzuleiten. Diese Ableitung passiert ganz automatisch durch Transformationen. In Abbildung 5 wird diese Erklärung nochmal verdeutlicht. Auch die Art der Fehlerbehebung unterscheidet sich bei MDSD. Wenn ein Fehler in der Implementierungsphase beim generierten Code erkannt wird, wird dieser im PIM korrigiert und eine erneute Transformation bis zum Generieren von Quellcode angestoßen. Es gibt verschiedene Ansätze, die das Überschreiben von manuell geschriebenem Programmcode vermeiden, diese werden in dieser Seminararbeit jedoch nicht näher betrachtet. (Generierter) Programmcode und Modell stimmen so immer überein. Es kann ganz einfach Programmcode für eine neue Plattform generiert werden, indem eine Transformation und eine DSL für diese Plattform definiert wird. Entwickler sollen von anspruchslosen aber fehleranfälligen Tätigkeiten entlastet werden. Auch der Wartungsaufwand wird minimiert, falls es viele PSM gibt, die von einem PIM generiert werden und die Fehler beim PIM liegen, da sie nur an einer Stelle beseitigt werden müssen. Durch Transformationen werden das vorhandene PSM und der Programmcode automatisch aktualisiert.

1.3 Model-to-Model Transformationen

Abbildung 6 zeigt, wie eine Model-to-Model Transformation abläuft. Bevor wir eine Transformation durchführen können, werden die Transformationsregeln (*Transformation Definition* in Abbildung 6), die Metamodelle vom Quellmodell (*Source Metamodel* in Abbildung 6) und Zielmodell (*Target Metamodel* in Abbildung 6) benötigt. Die Metamodelle können gleich oder verschieden sein. In den Trans-

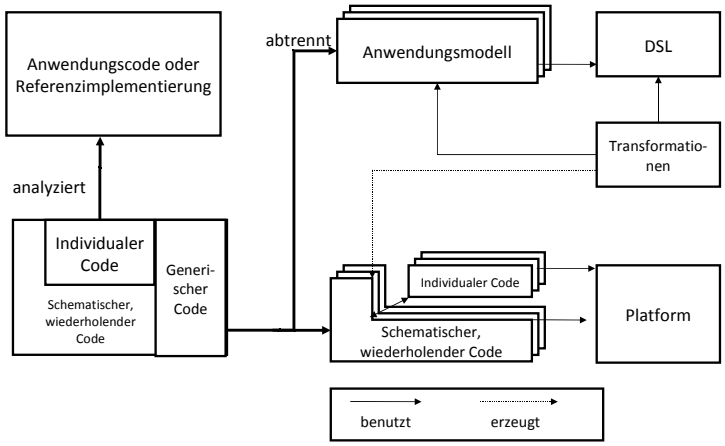


Abbildung 5. Grundidee modellgetriebener Softwareentwicklung aus [4]

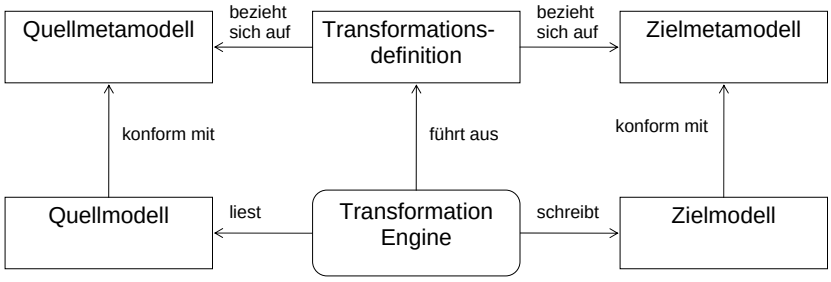


Abbildung 6. Model-to-Model Transformation (aus [1, S. 623])

formationsregeln wird definiert, wie Elemente vom Quellmodell in Elemente des Zielmodells transformiert werden. Als Ergebnis entsteht das Zielmodell. Transformationen von Metamodellen sind ebenfalls möglich. Sie werden nach dem gleichen Prinzip durchgeführt. Statt dem Quell- und Zielmodell gibt es ein Quell- und ein Zielmetamodell. Diese Metamodelle sind wiederum durch Metametamodelle beschrieben, wie wir es in Abbildung 3 schon gesehen haben.

2 openArchitectureWare

In diesem Kapitel wird zuerst die Entstehung von oAW behandelt. Danach wird der Aufbau und die Funktionsweise des Werkzeugs näher betrachtet. Zum Schluß werden die wichtigsten Kerneigenschaften von oAW aufgezählt und beschrieben.

2.1 Entstehung und Architektur von OpenArchitectureWare

oAW ist ein Wergzeug für Model-to-Code und Model-to-Model Transformationen. Zuerst wurde oAW als kommerzielles Projekt von der Firma *b+m* entwickelt. Im Herbst des Jahres 2003 hat b+m den Quellcode von oAW veröffentlicht und die Linzenz auf LGPL geändert. Momentan ist oAW eines der bekanntesten quell-offenen Werkzeuge für Modelltransformationen. Sein großer Vorteil ist die große Anzahl von Sponsoren (*b+m, Informatik AG, Itemis*), eine große Online-Community, sowie die Integration in Eclipse, was die Implementation von Transformationen erheblich erleichtert.

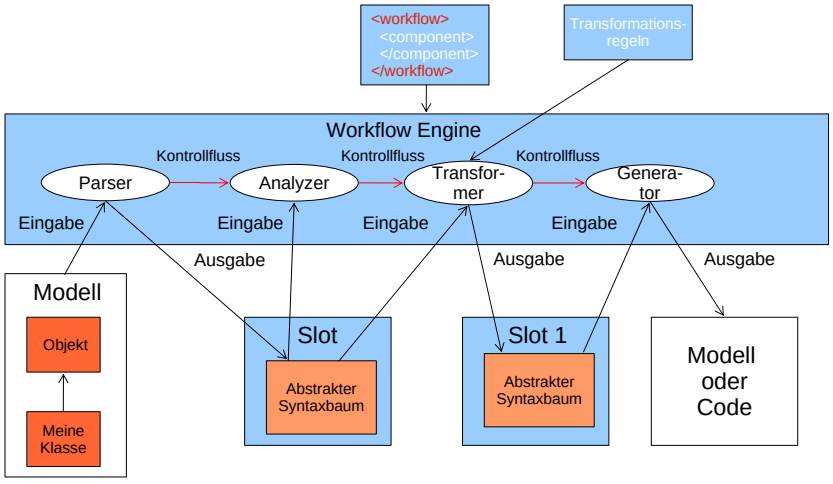


Abbildung 7. Beispiel für eine Transformation in openArchitectureWare (oAW)

Der Aufbau von oAW basiert auf dem Architekturmuster eines Fließbandes und ähnelt sehr stark einem Compiler. Es besteht aus einer Workflow Engine, die

in Java programmiert ist. Die Workflow Engine kann durch eine XML basierte Konfigurationsdatei (*Konfig* in Abbildung 7) sehr flexibel konfiguriert werden. Die Standardkomponenten von oAW sind der Parser, der Analyzer, der Transformer und der Generator. Die Workflow Engine kann diese Komponenten in bestimmte Reihenfolge und mit definierten Parametern aufrufen. Die Funktionalität von oAW kann durch eigene Komponenten ergänzt werden. Um Komponenten in oAW einbinden zu können, wird von dieser gefordert, die Schnittstelle *org.openarchitectureware.workflow.WorkflowComponent* zu implementieren. Die Klasse, die diese Schnittstelle implementiert, muss einen Default-Konstruktor haben, damit sie instanziiert werden kann. In Listening 5.1 ist die Schnittstelle *WorkflowComponent* zu sehen.

Listing 5.1. Die Schnittstelle *WorkflowComponent*

```
1  public interface WorkflowComponent {
2    /**
3    * @param ctx
4    * current workflow context
5    * @param monitor
6    * implementors should provide some feedback
7    * about the progress
8    * using this monitor
9    * @param issues
10   */
11   public void invoke(WorkflowContext ctx,
12                      ProgressMonitor monitor,
13                      Issues issues);
14   /**
15   * Is called by the container after configuration so the
16   * component can validate the configuration
17   * before invocation.
18   *
19   * @param issues -
20   * implementors should report configuration issues
21   * to this.
22   */
23   public void checkConfiguration(Issues issues);
24 }
```

Nach dem Zugriffsprotokoll dieser Schnittstelle wird von oAW zuerst die Methode *checkConfiguration* aufgerufen. Diese Methode soll überprüfen, ob die Konfiguration der Komponente in Ordnung ist, d.h. bevor dieser Aufruf stattfindet, muss die Komponente konfiguriert werden. Danach wird die Methode *invoke* aufgerufen, die die eigentliche Aufgabe der Komponente umsetzt. Wie in Abbildung 7 gezeigt wurde, können die Komponenten durch so genannte *Slots* Informationen (z.B. instanziierte Modelle) tauschen. Diese *Slots* sind durch den Eingabepa-

rameter *ctx* referenzierbar. Die Komponenten können auch Eigenschaften haben. Ausführlichere Information sind in [6] zu finden.

2.2 Kerneigenschaften

Der wichtigsten Kerneigenschaften von oAW sind:

- Die Workflow Engine, die bereits im vorherigen Kapitel erklärt wurde.
- oAW unterstützt das Parsen von den folgenden Modellen: Eclipse Modeling Framework (EMF), Eclipse UML2, eine EMF-basierte Implementierung des UML 2.* Metamodells, sowie Modellinstanzen von verschiedenen UML Werkzeugen (MagicDraw, Poseidon, Enterprise Architect, Rose). oAW kann aber auch für das Parsen von anderen Modellen, die nicht unterstützt werden, angepasst werden, indem man eine eigene Parserkomponente implementiert und einbindet.
- *Check* ist eine der Object Constraint Language (OCL) ähnliche Sprache, die die Definition von Einschränkungen (Constraints) für Modelle unterstützt.
- *Xpand* ist eine leistungsfähige vorlagen-basierte Sprache zur Codegenerierung. Sie unterstützt Template-Polymorphismus, Template-Aspekte und andere nicht-triviale Eigenschaften, die für das Schreiben von komplexen Codegeneratoren benötigt werden.
- *Xtend* ist eine funktionale Sprache, die Model-to-Model Transformationen unterstützt. Auf die Sprache Xtend wird im folgenden Kapitel näher eingegangen.

Eine vollständige Liste mit detaillierten Informationen über die Kerneigenschaften von oAW ist unter [7] zu finden.

3 Model-to-Model Transformationen mit openArchitectureWare

In diesem Kapitel werden zunächst die Eigenschaften der Model-to-Model Transformationssprache *Xtend* von oAW beschrieben. Anschließend wird die Model-to-Model Transformation anhand eines Beispiels veranschaulicht.

3.1 Elemente der Sprache Xtend

Im Folgenden werden die verschiedenen Elemente der Sprache *Xtend* beschrieben. Eine noch ausführlichere Anleitung ist unter [8] zu finden.

Xtend-Dateien haben eine **.ext* Endung. Kommentare werden, wie in Java, durch \\ oder */*... */* gekennzeichnet. In Listing 5.2 ist eine mögliche Strukturierung einer *Xtend*-Datei dargestellt.

Listing 5.2. Beispielstrukturierung einer *Xtend*-Datei

```
1 import my::metamodel;
```

```
2 extension other::ExtensionFile;
3 /**
4 * Documentation
5 */
6 anExpressionExtension(String stringParam) :
7   doingStuff(with(stringParam))
8 ;
9 /**
10 * java extensions are just mappings
11 */
12 String aJavaExtension(String param) :
13   JAVA my.JavaClass.staticMethod(java.lang.String)
14 ;
```

Als nächstes werden die Elemente der Sprache *Xtend* beschrieben. Zu Beginn wird der Name des Elements in Fettschrift dargestellt. Anschließend wird auf die genaue Syntax des Elements eingegangen. Abschließend erfolgt eine Beschreibung der Funktionalität und Semantik des Elements.

import `import my::imported::namespace;`
Durch dieses Element kann ein Namensraum eines Modells oder Metamodells importiert werden. Wildcards oder das Einbinden eines Typs, der innerhalb eines Namensraums existiert, sind in *Xtend* nicht erlaubt.

```
import my::imported::namespace::*; // WRONG!
import my::Type;                   // WRONG!
```

extension `extension fully::qualified::ExtensionFileName;`
Durch dieses Element wird die Funktionalität einer vorhandenen Xtend-Erweiterung eingebunden. Eine Xtend-Erweiterung ist eine ganz normale Xtend-Datei.

reexport `extension fully::qualified::ExtensionFileName reexport;`
Durch dieses Element kann man die Funktionalität der Erweiterungen exportieren. Dieses Element kann benutzt werden, um die Abhängigkeiten zwischen den *Xtend*-Dateien besser zu strukturieren und mehrfaches einbinden zu vermeiden.

Extension
```
ReturnType extensionName(ParamType1 paramName1, ...):
        expression-using-params
      ;
```
Eine *Extension* entspricht einer Methode in einer Programmiersprache. Eine *Extension* kann direkt als eine Methode oder in Form der „member syntax“ aufgerufen werden.

```
extensionName(myNamedElement)  //Aufruf als Methode
myNamedElement.extensionName() //Aufruf in "member syntax"
```

Bei der „member syntax“ Form des Aufrufs wird „myNamedElement“ als erster Eingabeparameter der aufgerufen *Extension* übergeben. Es ist zu beachten, dass die *Extensions* nichts mit objektorientierung zu tun haben. Das

heißt, dass das *Überschreiben* von *Extensions* nicht funktioniert. Die Reihenfolge, in der die *Extensions* aufgelöst werden, ist wie folgt:

1. Zuerst wird in der Datei des Aufrufers nach einer passender *Extension* gesucht.
2. Anschließend wird in den Erweiterungen nach einer passender *Extension* gesucht.

Die Auflösung erfolgt durch den automatischen polymorphen Dispatcher von Xtend, der immer die passendste Signatur für das aktuelle Objekt aufruft. Die *Extensions* sind auch aus Java-Programmen heraus aufrufbar, wie das folgende Beispiel in Listing 5.3 zeigt.

Listing 5.3. Aufruf einer *Extension* aus einem Java-Programm heraus

```
1  //Java
2
3  // setup
4  XtendFacade f = XtendFacade.
5                    create("my::path::MyExtensionFile");
6  // use
7  f.call("sayHello",new Object[]{"World"});
8
9  //Extend
10
11 // Die aufgerufene Extension
12 String sayHello(String s) :
13   "Hello␣" + s
14 ;
```

private `private ReturnType extensionName():`
Die *Extensions*, die als *private* definiert sind, sind außerhalb der Xtend-Datei nicht sichtbar.

public `public ReturnType extensionName():`
Eine als *public* gekennzeichnete *Extension* kann von einer anderen *Extension*, die in einer anderen *Xtend-Datei* liegt, aufgerufen werden. Falls public oder private bei der Definition einer *Extension* weggelassen wird, ist diese *Extension* implizit als public definiert.

JAVA
```
ReturnType extensionName(ParamType1 paramName1, ...):
    JAVA fully.qualified.Type.
           staticMethod(ParamType1 paramName1, ...)
  ;
```
Durch das Element *JAVA* kann eine Java Methode aufgerufen werden. Dabei ist es wichtig zu beachten, dass die importierten Namensräume nicht benutzt werden können. Der vollständige Pfad der Methode musst angegeben sein. Die Java Methode muss ferner „static“ sein, weil in *Xtend* keine Möglichkeit existiert, Instanzen einer Java-Klasse zu erzeugen.

cached
```
cached String getterName(NamedElement ele) :
    'get'+ele.name.firstUpper()
```

```
    ;
```

Dient zum Zwischenspeichern des Ergebnisses. Hier wird „getterName“ für jedes „NamedElement“ nur einmal berechnet. Weil das Ergebnis nur einmal berechnet wird, sollte dieses Element vorsichtig verwendet werden.

```
create create ReturnType extensionName(ParamType1 paramName1):
        expression-using-params
    ;
```

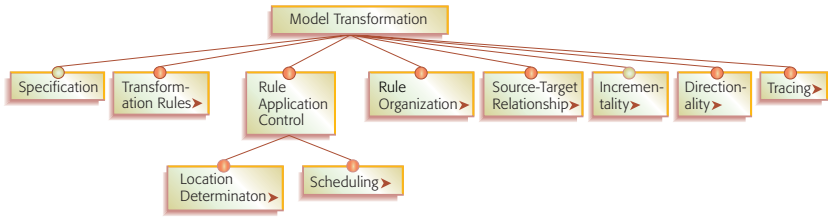
Dieses Element ist eine Erweiterung von Xtend, die in Version 4.1 von oAW eingeführt wurde. Das Neue dabei ist, dass der „ReturnType“ schon beim Aufruf einer *Extension* erzeugt wird und in seinem Rahmen durch das Schlüsselwort *this* referenzierbar ist.

```
-> create String extensionName(ParamType1 paramName1):
     paramName1.getName()->
     paramName1.setName("Name")
    ;
```

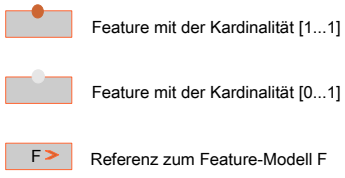
Der Pfeil zeigt das Ende einer Zeile.

3.2 Eigenschaften der Sprache Xtend

In diesem Kapitel wird der Artikel von Czarnecki und Helsen [1] vorgestellt und eine Verbindung mit den Eigenschaften von Xtend hergestellt. Der Artikel beschreibt, welche Eigenschaften ein Transformationsansansatz haben kann. Im Artikel wird zwischen Model-to-Model und Model-to-Code Transformationen nicht unterschieden. Diese Seminararbeit beschränkt sich jedoch auf Model-to-Model Transformationen, sowie ein Bezug zu den Eigenschaften der obersten Ebene des dort entwickelten Modells.



(a) Feature-Diagramm



(b) Legende

Abbildung 8. Feature-Diagramm zu Modelltransformationen (aus [1, S. 626])

In Abbildung 8 sind die Eigenschaften einer Transformation zu sehen. Im Folgenden wird auf die einzelnen Eigenschaften im Detail eingegangen.

- *Specification*: Die Möglichkeit verschiedene Bedingungen vor und nach der Transformation zu prüfen. In oAW kann dies durch in *Check* formulierte *Constraints* geschehen. Normalerweise wird das Quellmodell vor der Transformation und das Zielmodell nach der Transformation auf die Einhaltung angegebener *Constraints* überprüft.
- *Transformation-Rules*: Sind die Transformationbefehle, die die eigentliche Transformation durchführen. Im Falle von *Xtend* ist das eine Extension, die einen Transformationsschritt ausführt. Eine Untergliederung der Eigenschaften von *Transformation-Rules* ist in Abbildung 9 gegeben.
  - *Domains*: Sind Teile eines Befehls, die für den Zugriff auf das Ziel- oder das Quellmodell zuständig sind. Die Befehle könne mehr als ein Ziel- und ein Quellmodell referenzieren. Die Befehle in *Xtend* besitzen nur genau ein Ziel- und ein Quellmodell.
  - *Syntactic separation*: Ist die Gliederung des Befehls in eine linke und rechte Seite. Die Eingabeparameter können als die rechte Seite des Befehls gesehen werden. Im Rahmen einer *Extension* ausgeführte Operationen können als die linke Seite des Befehls gesehen werden.
  - *Multidirectionality*: Transformationsbefehle, die diese Eigenschaft haben, können Transformationen in beide Richtungen durchführen. Leider wird diese Eigenschaft von Xtend nicht unterstützt. Für eine Rücktransformation ist eine neue Transformation zu definieren.
  - *Application condition*: Das sind „bedingt“ ausführbare Teile einer Transformation. Diese sind in Xtend durch den Ternäroperator oder durch *Java Extensions* möglich, in denen kompliziertere Bedingungen definiert werden können.
  - *Intermediate structure*: Sind zusätzliche Hilfstrukturen, die kein Teil des Ziel- oder des Quellmodells sind, aber bei der Ausführung eines Befehls benötigt werden können. *Xtend* benutzt keine *Intermediate structure*.
  - *Parameterization*: Welche Eingabeparameter bei den Transformationsbefehlen möglich sind. *Parameterization* wird nach *Control Parameters*, *Generics* und *Higher-Order Rules* untergliedert. *Control Parameters* sind Eingabeparameter, die den Kontrollfluss der Transformation beeinflussen. In *Xtend* sind sie möglich. *Generics* als Eingabeparameter in Form von Modell-Datentypen sind mit Xtend ebenfalls möglich. *Higher-Order Rules* sind andere Befehle als Eingabeparameter, welche Xtend nicht unterstützt.
  - *Reflection and aspects*: Haben dieselbe Bedeutung wie bei der OOP. Zur Zeit werden diese zwei Eigenschaften von Xtend nicht unterstützt.
- *Rule application control: Location determination*: Ist eine Ablaufplanung der Reihenfolge der Transformationsbefehle. Diese Eigenschaft hat verschiedene Varianten. In Xtend wird eine deterministische Ablaufplanung verwendet. Dies bedeutet, dass das Quellmodell in einer bestimmten Reihenfolge durchlaufen wird und die Transformationsbefehle angewendet werden. In *Xtend*

wird nur ein Einstiegspunkt (eine *Extension*) für die Transformation bereitgestellt und durch den Aufrufablauf der *Extensions* wird das Quellmodell durchlaufen und die Transformationsbefehle angewandt.
- *Rule application control: Rule scheduling* - ist ebenfalls untergliedert. Hier ist das *Explicit scheduling* mit seiner Ausprägung *internal scheduling* interessant, da von Xtend diese Schedulingstrategie implementiert ist. Diese Schedulingstrategie ist dadurch bestimmt, dass man einen Einfluss auf die Ausführungsreihenfolge der Transformationsbefehle nehmen kann, indem man einen Transformationsbefehl (*Extension*) innerhalb eines anderen aufrufen kann.
- *Rule organization*: Charakterisiert, ob die Möglichkeit besteht, die Transformationsbefehle zu gruppieren und zu strukturieren. Bei dieser Eigenschaft sind folgende drei Varianten möglich:
  - *Modularity mechanisms* - ist die Möglichkeit, Transformationsbefehle in große Einheiten (Module) packen zu können. Bei *Xtend* hat man diese Eigenschaft durch die Wiederverwendbarkeit der *Extensions*.
  - *Reuse mechanisms* - die Wiederverwendbarkeit der Transformationsbefehle. Das ist bei Xtend, durch die Möglichkeit externe (Java) Extensions wieder zu verwenden, gut gelöst.
  - *Organizational structure* - wie die Transformationsbefehle bezüglich Quell- oder Zielmodell strukturiert sind. Bei Xtend ist eine Strukturierung bezüglich des Zielmodells umgesetzt.

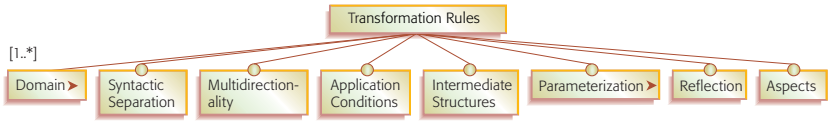


Abbildung 9. TransformationRules

- *Source-target relationship*: Gibt an, ob bei der Transformation ein neues Zielmodell erzeugt oder das Quelmodell verändert wird. Bei Model-to-Model Transformationen mit oAW wird immer ein Zielmodell erzeugt, weil ein Quellmodell nicht verändert werden darf.
- *Incrementality* - wie werden die Veränderungen im Quellmodell bei einer neuen Transformation im Zielmodell propagiert. In oAW ist das Problem durch geschützte Bereiche gelöst. Geschützte Bereiche werden nur bei einer Model-to-Code Transformation eingesetzt, weil generierten Code häufig mit eigenem Code angereichert wird. Bei einer erneuten Transformation soll dieser Code nicht überschrieben werden. Bei Model-to-Model Transformationen wird das existierende Zielmodell mit dem Neuen einfach überschrieben, weil keine manuellen Änderungen in PSMs enthalten sind oder bei der Transformation berücksichtigt werden.
- *Directionality*: Wird benutzt, um die Richtung der Transformationsbefehle zu definieren. Bei Xtend sind die *Extensions* nur in eine Richtung anwendbar.

– *Tracing*: Das Protokollieren, welche Transformationsbefehle ausgeführt wurden, von welchen Quellelement welches Zielelement erzeugt wurde usw. . Xtend bietet keine solche Unterstützung an.

3.3 Model-to-Model Transformation anhand eines Beispiels

In diesem Kapitel wird die Model-to-Model Transformation anhand eines Beispiels gezeigt. Das Beispiel ist aus [9] entnommen und dort ausführlich beschrieben.

In diesem Beispiel wird ein Zustandsautomat, der das UML2-Metamodell als DSL hat, als Quelle benutzt. Der Zustandsautomat wurde mit MagicDraw 11.6 modelliert und ist in Abbildung 10 dargestellt.

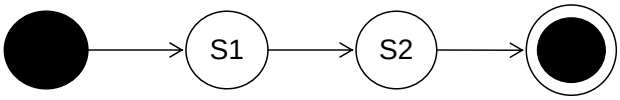


Abbildung 10. Zustandsdiagramm

Dieser Zustandsautomat wird auf einen anderen transformiert, der unser einfacheres Metamodell für Zustandsautomaten als DSL benutzt. Im Vergleich zum UML2-Metamodell besitzt unser Metamodell keine hierarchischen Zustände, keine Regionen, keine Guards und keine Actions. Bei der durchgeführten Transformation handelt es sich um eine PIM-zu-PSM-Transformation. Das vereinfachte Metamodell ist in Abbildung 11 gezeigt.
Nachdem die Architektur von oAW bereits bekannt ist, muss zuerst die Workflow Engine für die Transformation konfiguriert werden. Die Workflow-Konfigurationsdatei ist in Listing 5.4 dargestellt.

Listing 5.4. Workflow-Konfigurationsdatei

```
<workflow>
 <bean class="oaw.uml2.Setup" standardUML2Setup="true"/>
 <component
  class="org.openarchitectureware.emf.XmiReader">
  <modelFile value="./xmi/model.uml2"/>
  <outputSlot value="${uml2model}"/>
 </component>
 <component
  class="org.openarchitectureware.check.CheckComponent"/>
 <component class="oaw.xtend.XtendComponent">
  <metaModel class="oaw.uml2.UML2MetaModel"/>
  <metaModel class="oaw.type.emf.EmfMetaModel">
```

```
13     <metaModelFile value="stama.ecore"/>
14    </metaModel>
15    <invoke value="uml2trafo::main(uml2model)"/>
16    <outputSlot value="stamaModel"/>
17   </component>
18   <component class="oaw.emf.XmiWriter"...
19 </workflow>
```

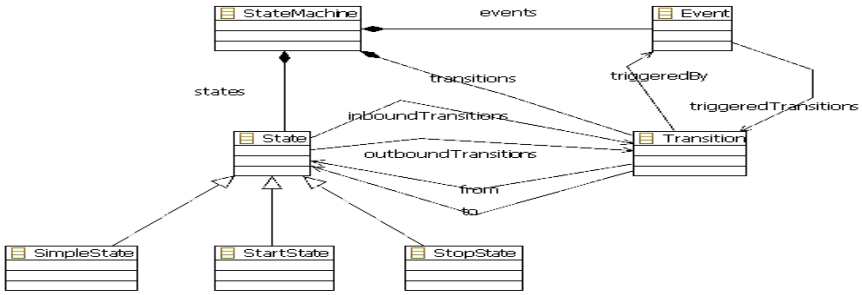


Abbildung 11. Vereinfachtes Metamodell für Zustandsautomaten

Zuerst wird das UML-Modell mit Hilfe des XmiReaders geladen und im *Slot* mit dem Namen *uml2model* abgelegt. Anschließend werden einige Constraints überprüft, da keine falsche Modelle transformiert werden sollen. Danach wird die Xtend-Komponente ausgeführt, die die eigentliche Transformation übernimmt. Während der Transformation wird das UML-Metamodell und unser vereinfachtes Metamodell benötigt, deswegen diese in die Transformation eingebunden werden. Mit dem *outputSlot* Element wird definiert, in welchem *Slot* das resultierende Modell gespeichert werden soll. Durch *invoke...* wird die Xtend-Extension *main* aufgerufen und die Transformation gestartet.
Main ist in Listing 5.5 dargestellt. Die *Extension* hat als Eingabeparameter das geladene UML-Modell. Da im UML-Metamodell und in unserem Metamodell, Klassen mit den gleichen Namen vorkommen, sind die *uml* und *stama* Namensräume explizit anzugeben.

Listing 5.5. Main

```
main( uml::Model model ):
    model.ownedElement.typeSelect(uml::Class).
      ownedBehavior.
      typeSelect(uml::StateMachine).first().newSM()
;
```

Zur Vereinfachung wird angenommen, dass das UML-Modell nur einen Zustandsautomat besitzt und dieser der einzige im Modell vorhandene ist. Es wird über alle *ownedElement* des Modells iteriert und die Klassen heraus gefiltert. Danach wird über die *ownedBehavior* der Klassen iteriert und die *StateMachine* Klasse heraus gefiltert. Durch die Annahme, dass das Modell nur einen Zusandsautomaten besitzt, kann dieser als erstes Element aus der Liste der zu selektierenden Zustandsautomaten angesprochen werden. Mit dem selektierten Automat wird *newSM* in der „member syntax“ Art, die bereits behandelt wurde, aufgerufen. *newSM* ist in Listing 5.6 dargestellt.

Listing 5.6. newSM

```
create stama::StateMachine newSM( uml::StateMachine sm ):
  setName( sm.name ) ->
  setStates( sm.region.first().
    ownedMember.
    typeSelect(uml::Vertex).map() ) ->
  setTransitions( sm.region.first().ownedMember.
    typeSelect(uml::Transition).map() )
;
```

Die *newSM Extension* ist mit dem Schlüsselwort *create* definiert. Dies hat zur Folge, dass der Automat schon beim Aufruf von *newSM* erzeugt wird. An dieser Stelle wird der Automat initialisiert. Zuerst wird der Name gesetzt. Danach werden die Zustände gesetzt, indem alle Zustände des UML Automaten in Zustände aus unserem Metamodell transformiert werden. Dasselbe wird mit den Übergängen gemacht. Hier wird der automatische polymorphe Dispatcher von Xtend, der im vorigen Kapitel schon erwähnt wurde, verwendet. Die Erzeugung der Zuständen und der Übergänge ist in Listing 5.7 dargestellt.

Listing 5.7. Erzeugung von den Zuständen und den Übergängen

```
create stama::SimpleState map( uml::Vertex v):
  setName( v.name )
;

create stama::StartState map( uml::Pseudostate s):
  setName( s.name )
;

create stama::StopState map( uml::FinalState s):
  setName( s.name )
;

create stama::Transition map( uml::Transition t):
  setName( t.name ) ->
```

```
15   setFrom( t.source.map() ) ->
16   setTo( t.target.map() )
17 ;
```

Somit ist das Beispiel fertig. Es musst noch erwähnt werden, dass bei mehrmaligen Aufruf von *map* mit demselben Element, nur beim ersten Aufruf ein neues Element von Typ *stama* erzeugt wird. Beim jedem nächsten Aufruf wird dieser anschließend zurückgegeben. Das passiert durch die implizite *cached*-Funktionalität von *create Extensions*.

4 Zusamenfassung

Bei dieser Ausarbeitung wurde eine neue Art des Software-Entwicklungsprozess vorgestellt: MDSD. Die Unterschiede zwischen MDSD und einem „normalen“ Software-Entwicklungsprozess wurden aufgezeigt. Anschließend wurde näher auf Model-to-Model Transformationen eingegangen, die gerade im Rahmen von MDSD häufige Anwendung finden. Anschließend wurde das Werkzeug openArchitectureWare vorgestellt. Insbesondere wurde die Architektur des Werkzeugs und seine Kerneigenschaften betrachtet. Anschließend wurde die Model-to-Model Transformationssprache *Xtend* von oAW vorgestellt. Die Elemente der Sprache wurden im Detail beschrieben und ihre Semantik verdeutlicht. Danach wurde der Artikel von Czarnecki and Helsen [1] vorgestellt. Die Sprache Xtend wurde in das in dem Artikel vorgestellte Eigenschaftsmodell eingeordnet. Zum Schluss wurde die Anwendung einer Model-to-Model Transformation mit Xtend anhand eines Beispiels veranschaulicht.

5 Ausblick

Die nächste Version von oAW wird 5.0 sein. Die Pläne für diese Version sind, die Sprachen Xpand und Xtend zu integrieren. Auch die Definition von *Constraints* wird innerhalb von Xtend-Dateien möglich sein. Die Entwicklung von Transformationen wird mit oAW 5.0 durch Funktionalitäten wie Profiling und Fehlerbehandlung noch besser unterstützt.

Abbildungsverzeichnis

Literatur

1. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3) (2006) 621–645
2. Becker, S., Dikanski, A., Drechsel, N., Achraf, A., Happe, J., El-Oudghiri, I., Koziolek, H., Kuperberg, M., Rentschler, A., Reussner, R.: Modellgetriebene Software-Entwicklung. Architekturen, Muster und Eclipse-basierte MDA. Interner Bericht 2006-18, Universität Karlsruhe, Fakultät für Informatik (2006)

3. Runge, W.: Werkzeug Objekt. Kybernetik und Objektorientierung. PhD thesis, Universität Flensburg (April 2001)
4. Stahl, T., Völter, M.: Modellgetriebene Softwareentwicklung. dpunkt-Verlag (2005)
5. Object Management Group: OMG's MetaObject Facility. http://www.omg.org/mof/ [Letzter Zugriff am 03.04.2007].
6. Efftinge, S., Voelter, M.: openArchitectureWare 4.1 Workflow Engine Reference. http://www.eclipse.org/gmt/oaw/doc/4.1/r05_workflowReference.pdf [Letzter Zugriff am 22.07.2007].
7. openArchitectureWare.org: openArchitectureWare Core Features. http://www.openarchitectureware.org/staticpages/index.php/about [Letzter Zugriff am 22.07.2007].
8. Efftinge, S.: openArchitectureWare 4.1 Extend Language Reference. http://www.eclipse.org/gmt/oaw/doc/4.1/r25_extendReference.pdf [Letzter Zugriff am 22.07.2007].
9. Völter, M., Groher, I.: Modelltransformationen in der Praxis. JavaSpektrum **01/2007** (Januar 2007)

Validation von Modell-Transformationen

Christopher Köker

Betreuer: Klaus Krogmann

Zusammenfassung Bei der modellgetriebenen Entwicklung von Software spielt die Transformation von Modellen eine entscheidende Rolle. Die syntaktische und semantische Korrektheit dieser Modell-Transformationen muss anhand von geeigneten Kriterien sichergestellt werden. Dazu existieren automatische formale Validierungs-Werkzeuge, die besonders sicherheitskritische Teile von Modellen überprüfen können. Typische Fehler, die bei der Implementierung von Modell-Transformationen auftreten, lassen sich auch mit systematischen Testverfahren erkennen, die Testfälle zur Aufdeckung häufiger Fehler generieren. Besonders zur Validierung der in diesem Artikel beispielhaft vorgestellten Modell-Transformationen von UML-Statecharts in Java-Programmcode eignen sich spezielle Relationen, mit denen semantische Übereinstimmung von Quell- und Ziel-Modell mit Hilfe von Theorem-Beweisern gezeigt werden können.

Key words: Transformationen, Modell-Transformationen, Validierung, Verifikation, formale Validierungs-Werkzeuge, Test von Modell-Transformationen, Model-Checking, Model Driven Architecture, modellgetriebene Software-Entwicklung

1 Einleitung

Qualität ist und bleibt eine, wenn nicht gar DIE Schlüsselfrage der Software-Entwicklung. Von den vielfältigen existierenden Ansätzen zur Verbesserung der Software-Qualität ist die *modellgetriebene Software-Entwicklung*, zum „dominierenden Trend in der Software-Entwicklung“ [8] geworden und kann neben der Qualität auch die Produktivität deutlich verbessern [3].

Der Erfolg der modellgetriebenen Software-Entwicklung, die auf komplexen und hochautomatisierten Modell-Transformationen basiert, hat dabei zu einem großen Bedarf an solchen Modell-Transformationen und an Werkzeugen zu deren Unterstützung geführt [8] [4].

Dabei ist die Qualität der Modell-Transformationen von entscheidender Bedeutung und kann leicht zum „Flaschenhals der modellgetriebenen Software-Entwicklung“ [8] werden.

Nach Varró [8] kann die Qualität in der modellgetriebenen Entwicklung von Software durch die Automatisierung von Transformationen zwar deutlich verbessert werden, da manuelle Fehler in der Implementierung von Modellen vermieden werden können. Dennoch können weiterhin Fehler im Entwurf und der Implementierung von Transformationen als solche auftreten, die zu syntaktisch korrekten, aber semantisch inkorrekten Modellen führen [2].

Gefragt sind also Methoden und Techniken, die eine möglichst automatisierte Validierung von Modell-Transformationen anhand festgelegter Kriterien ermöglichen.

Nach einer Vorstellung der Kriterien, die ein qualitativ hochwertiger Transformationsprozess erfüllen muss, im ersten Kapitel, wird im zweiten Kapitel eine Reihe automatischer formaler Validierungs-Werkzeuge vorgestellt. Dabei wird ein Schwerpunkt auf das Modell-Checking gelegt.

Ein anderer Ansatz neben den automatischen, formalen Validierungswerkzeugen, der nach [4] in der Industrie breite Anwendung findet, ist die Validierung von Modell-Transformationen durch Tests. Verschiedenen Ansätze in diesem Bereich werden im dritten Kapitel beschrieben, ehe im Kapitel vier beispielhaft näher auf die Validerung von Statechart-Transformationen von UML nach Java eingegangen wird.

1.1 Sinn und Zweck von Modell-Transformationen

Die modellgetriebene Software-Entwicklung, die zunehmende Bedeutung in der Software-Technik erfährt [3], basiert auf komplexen und hoch-automatisierten Modell-Transformationen [8].

Wichtig für die modellgetriebene Software-Entwicklung ist dabei nicht nur eine präzise Modellierungssprache. Vielmehr hängt die Qualität der so entstehenden Software stark von möglichst präzisen und möglichst stark automatisierten Modell-Transformationen ab.

Um das Potential der modellgetriebenen Software-Entwicklung voll auszuschöpfen, muss also die Qualität der Modell-Transformationen verbessert werden und deren Verlässlichkeit durch die Validierung gegen festgelegte Kriterien gezeigt werden [3].

1.2 Entwicklung von Modell-Transformationen

Da die Entwicklung von Modell-Transformationen nach Küster [5] keine leichte Aufgabe ist, schlägt er als Konsequenz vor, Erfahrungen aus der Entwicklung von Anwendungs-Software auf die Entwicklung von Modell-Transformationen zu übertragen. Er verweist dabei auf eine Art „vereinfachtes Wasserfall-Modell“, das in Abbildung 1 dargestellt ist.

Wie in der Abbildung zu sehen, werden dabei Modell-Transformationen anhand eines Prozess-Rahmens entwickelt, der zwischen den Phasen Anforderungsspezifikation, Entwurf und Implementierung auch bei der Entwicklung von Modell-Transformationen unterscheidet.

Für die Praxis würde ein solcher Ansatz nach wahrscheinlich zu restriktiv sein und eher ein „inkrementeller und iterativer Ansatz“ [5] gebraucht werden. Der Ansatz zeigt aber die Bedeutung auf, die die Validierung bei der Entwicklung von Modell-Transformationen besitzt, indem er die Validierung als letzten Prozessschritt explizit hervorhebt und damit auf eine Stufe zum Entwurf und zur Implementierung stellt.

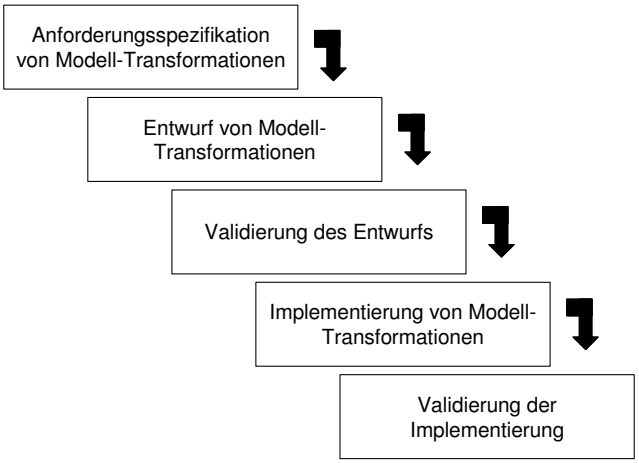


Abbildung 1. Abb. 1: Vereinfachtes Wasserfallmodell nach [5]

In jedem Fall muss man sich im Laufe der Entwicklung über die Anforderungen an Modell-Transformationen klar werden. Hierbei spielen sowohl die funktionalen Anforderungen wie auch die Anforderungen hinsichtlich Einschränkungen im Speicher- und Zeitbedarf eine Rolle.

Anschließend wird die Modell-Transformation entworfen. Schon nach dem Entwurf können analog zur Software-Entwicklung verschiedene Anforderungen validiert werden, bevor der Entwurf mittels eines entsprechenden Transformations-Werkzeugs implementiert wird und anschließend weitere Anforderungen validiert werden.

Bei der Validierung von Modell-Transformationen kommen dabei sowohl das Testen als auch formale Validierungs-Techniken zum Einsatz, wobei Küster den Erfolg von Modell-Transformationen daran knüpft, inwieweit „systematische Validierung“ ermöglicht wird. Dabei versteht er unter der systematischen Validierung die „Entwicklung von spezifischen Test- und Validierungs-Ansätzen, die auf die zugrundeliegenden Ansätze der Modell-Transformationen zugeschnitten sind und von diesen beeinflusst werden“. Auch die kontinuierliche, in den Entwicklungs-Prozess integrierte Validierung ist eine Forderung von Küster [5].

1.3 Kriterien zur Validierung des Transformationsprozesses

Eine „wichtige Dimension der Validierung von Modell-Transformationen“ [5] sind in jedem Fall die Kriterien, anhand derer eine Validierung stattfindet.

Diese unterteilt de Lara [6] zum einen in Kriterien bezüglich des funktionellen Verhaltens, die weiter in Terminierung, also endlichen Zeitbedarf, und Konfluenz, also belibige Reihenfolge der Regelanwendung, unterteilt werden können. Diese Kriterien stellen sicher, dass „die gleiche Ausgabe aus der gleichen Eingabe“ [6] erhalten wird. Weitere Kriterien sind die syntaktische Korrektheit, also

die Sicherstellung, dass die Modelle, die aus den Modell-Transformationen resultieren, gültige Instanzen im Zielraum sind, sowie die Verhaltensäquivalenz, also gleiches Verhalten von Quell- und Ziel-Modell [6].

Terminierung Das Kriterium der Terminierung fordert, dass die Modell-Transformation in jedem Fall nach endlicher Zeit abgeschlossen ist, der Transformationsprozess also in jedem Fall nach endlicher Zeit mit einer Ausgabe endet. Dieses Kriterium der Terminierung muss nach Küster [5], ebenso wie das Kriterium der Konfluenz, vor allem bei regelbasierten Ansätzen zur Modell-Transformation beachtet werden. Denn bei diesen Ansätzen besteht besonders die Gefahr, dass es immer weiter möglich ist, eine oder mehrere Regeln anzuwenden, was in einer natürlich nicht gewollten Endlosschleife enden könnte.

Beispielsweise zeigt de Lara die Terminierung in dem von ihm gewählten Ansatz zur regelbasierten Modell-Transformation [6], indem er zeigt, dass jede Regel in seinem Transformationssystem nur endlich oft angewendet werden kann, der Prozess also nach endlich vielen Anwendungen endlich vieler Regeln auch in endlicher Zeit beendet wird.

Konfluenz Das Kriterium der Konfluenz ist eng mit dem Kriterium der Terminierung verknüpft. Nach de Lara [6] bedeutet Konfluenz bei regelbasierten Ansätzen zur Modell-Transformation, dass die Reihenfolge der Anwendung bestimmter Transformations-Regeln keine Rolle spielt. Das Ergebnis des Transformations-Prozesses muss also immer das gleiche sein, ganz egal, mit welcher Teiltransformation begonnen wird.

Das Kriterium der Konfluenz wird bei regelbasierten Ansätzen zur Modell-Transformation nach dem Beweis des Kriteriums der Terminierung gezeigt. Denn „wenn ein System terminiert und alle möglichen verschiedenen Anwendungen von Regeln“ [6] jeweils unabhängig von ihrer jeweiligen Reihenfolge zum gleichen Ergebnis führen, dann erfüllt das System das Kriterium der Konfluenz.

Syntaktische Korrektheit Unter dem Kriterium der syntaktischen Korrektheit, das er auch abweichend als Kriterium der Konsistenz bezeichnet, versteht de Lara [6], dass die finalen Modelle oder Texte, also die Ausgaben der Transformationen, korrekte Instanzen im jeweiligen Zielraum sind. Es dürfen also alle Anwendungen von Transformationsregeln nur zu solchen Modellen oder Teilen von Modellen führen, die im Zielmodellraum oder in der Zielsprache syntaktisch korrekt sind.

Dieses Kriterium wird auch von Küster [5] hervorgehoben. Er bezeichnet es dann erfüllt, wenn der Prozess der Modell-Transformationen stets „syntaktisch korrekte Modelle“ produziert.

Dabei unterscheidet er zwischen zwei verschiedenen Typen syntaktischer Korrektheit. Wenn eine Sprache vorliegt, in der die Modell-Transformation ausgedrückt wurde, muss die konkrete Modell-Transformation syntaktisch korrekt in Bezug auf diese vorliegende Sprache sein. So eine Sprache muss nicht immer

explizit mathematisch formalisiert vorliegen, sondern kann auch in einem Werkzeug, das die Modell-Transformationen durchführt, verborgen sein.

Die andere Form syntaktischer Korrektheit ist dann gefordert, wenn vor allen Dingen das Resultat der Modell-Transformation betrachtet wird. Dann wird nämlich oft angestrebt, dass dieses Resultat syntaktisch korrekt bezüglich einer konkreten Zielsprache ist [5].

In jedem Fall wird die Konsistenz oder syntaktische Korrektheit von Modell-Transformationen nach [6] beispielsweise über Invarianten oder die Möglichkeit der Anwendung bestimmter Regeln bis zur Terminierung nachgewiesen.

Verhaltensäquivalenz Wenn die Kriterien der Terminierung, der Konfluenz und der Konsistenz nachgewiesen wurden, wurde nur sichergestellt, dass für jede syntaktisch korrekte Eingabe immer auch eine syntaktisch korrekte Ausgabe erzeugt wird, aber noch nicht, dass die Ausgabe auch semantisch, also von ihren Verhaltenseigenschaften her, korrekt ist. Dazu ist es wichtig, die Modell-Transformation auch auf die Erfüllung des Kriteriums der Verhaltensäquivalenz zwischen Eingabe und Ausgabe hin zu überprüfen.

Um sicherzustellen, dass das „Quell- und das Ziel-Modell äquivalent in ihrem Verhalten“ sind, überprüft [6] für jede Regel seiner Transformation, dass deren Anwendung das durch das Quell-Modell beabsichtigte Verhalten des Ziel-Modells gewährleistet.

Auch von [8] wird die „semantische Äquivalenz von Quell- und Ziel-Modell“ als theoretisch bedeutsames Kriterium zur Validierung von Modell-Transformationen hervorgehoben, allerdings auch darauf hingewiesen, dass Modell-Transformationen auch eine „Projektion von der Quell-Sprache auf die Ziel-Sprache“ mit möglichem Informationsverlust darstellen können, die „semantische Äquivalenz von Modellen also nicht immer bewiesen“ werden kann. Als Alternative schlägt [8] die Definition von transformationsspezifischen „Korrektheits-Eigenschaften“ vor, die von der Transformation erhalten werden sollen.

2 Automatische formale Validierungswerkzeuge

In der steigenden Komplexität von Systemen der Informationstechnik und der zum Entwurf eingesetzten Modellierungssprachen sieht Varró [7] einen geradezu zwangsläufigen Grund für das Entstehen von „konzeptionellen, menschlichen Fehlern“ in Modellen, auch beim Einsatz von formalisierten Modellierungs-Paradigmen.

Er schließt daraus, dass die Benutzung von formalen Spezifikationen alleine nicht die Korrektheit und die Konsistenz des entworfenen Systems garantiert und sieht daher die Notwendigkeit von automatischen formalen Validierungs-Werkzeugen, die er in statische Analysemethoden, Theorem-Beweiser und Modell-Checking unterteilt.

Nach Varró ist es natürlich, dass die Korrektheit des Gesamtsystems in der Entwurfsphase aus Kosten- und Zeitgründen nicht gesichert werden kann, so

dass formale Validierungs-Methoden als hochautomatisierte Werkzeuge zur Unterstützung der Fehlerfindung eingesetzt werden [7].

2.1 Statische Analysemethoden

Unter statischen Analysemethoden versteht Varro typischerweise „Semi-Entscheidungs-Techniken“, die als Ausgabe *Ja*, *Nein* oder *Unbekannt* liefern. Diese Analysemethoden seien zwar effizient, könnten die Korrektheit des gesamten Entwurfs aber nicht sicherstellen [7].

2.2 Theorem-Beweiser

Die Korrektheit des Entwurfs kann mit Theorem-Beweisern gezeigt werden, aber durch den hohen benötigten Grad an Interaktion mit den Benutzern zur Konstruktion des Beweises ist diese Form der deduktiven Validierung kosten- und zeitintensiv. Die deduktive Validierung kann nach Varró somit sogar zum „Flaschenhals des Projekts“ werden.

Desweiteren führen die deduktiven Ansätze zur Validierung nach Varró häufig zu wenig hilfreichen „Alles-oder-Nichts“-Aussagen über das System in dem Sinne, dass nicht erfolgreiche Validierungsversuche keine nützlichen Hinweise zur Korrektur liefern.

Varró schließt, dass sich aktuelle Ansätze zu Theorem-Beweisern somit aus Gründen der Kosteneffizienz nur zur Validierung der kritischsten Teile des Systems oder Algorithmus' eignen [7].

2.3 Modell-Checking

Modell-Checker liefern nach Varró nicht nur eine Möglichkeit zum hochautomatisierten Treffen von Validierungs-Entscheidungen, sondern liefern im Falle von Validierungsfehlern auch direkt ein Gegenbeispiel, dass die zu einem Fehler führende Ausführungsreihenfolge zeigt.

Modell-Checker leiden allerdings oft an dem Problem einer „Explosion des Zustandsraums“, das in der Praxis eine Begrenzung darstellt, die die Validierung der Korrektheit des gesamten Modells unmöglich machen kann.

Da die Modell-Checker aber unter den formalen Validierungsmethoden den höchsten Grad an Automatisierung besitzen, also die Korrektheit des Systems mit möglichst wenig Eingriffen des Menschen beurteilt werden kann, sind sie als Werkzeuge für die Fehlerfindung besonders geeignet.

Sie ermöglichen nach Varró die Entdeckung von Fehlern in der Spezifikation in einer relativ frühen Phase der Modell-Entwicklung bereits vor der Implementierung, was sowohl zur Senkung der Entwicklungskosten als auch zur Verbesserung der Software-Qualität beitrage [7].

Das Modell-Checking Problem Ein *Modell-Checking Problem*, dass eingesetzt wird, um Sicherheitseigenschaften und Verklemmungsfreiheit zu zeigen, wird nach [7] wie folgt definiert:

Gegeben sind:

1. ein System-Modell in Form eines Übergangs-Systems TS mit einer Kripke-Struktur KS (Definition der Kripke-Struktur im nächsten Abschnitt)
2. eine Sicherheits-Eigenschaft Φ

Dann kann das Modell-Checking-Problem definiert werden als Entscheidung, ob Φ auf jedem ausführenden Pfad des Systems gilt, also $\forall i : s_i \Rightarrow \Phi$.
Außerdem soll das System frei von Verklemmungen sein, also $\forall i : \exists \tau \in T : s_i \Rightarrow en_\tau$, was bedeutet, dass in jedem möglichen Zustand mindestens eine Regel anwendbar ist, wenn der Zustand kein Endzustand ist.

Dabei ist eine *Sicherheits-Eigenschaft* Φ eine Invariante in Form eines boolschen Bedingung, die in jedem Zustands des Systems WAHR sein muss. Das heißt umgekehrt, dass jedesmal, wenn beim Durchqueren des Zustandsraums ein Zustand erreicht wird, in dem die Invariante verletzt wird, das Modell-Checking unmittelbar mit Ausgabe eines Fehlers anhalten kann.

Die *Verklemmungsfreiheit* ist so definiert, dass sich ein System im Zustand der Verklemmung befindet, wenn im gegebenen Zustand keine Übergänge möglich sind. Verklemmungsfreiheit liegt also immer dann vor, wenn mindestens ein Übergang aus dem aktuellen Zustand heraus möglich ist.

Bei den Übergängen handelt es sich dabei um Übergänge in den im Folgenden definierten Übergangs-Systemen.

Modell-Checking Übergangs-Systeme Ein Übergangs-System ist ein mathematischer Formalismus, der für viele Modell-Checking-Probleme als Eingabe-Spezifikation dient. Den verschiedenen Übergangs-Systemen gemeinsam ist, dass die Zustände dieser Systeme durch die Ausführung von nicht-deterministischen „Wenn-dann-sonst-Regeln“ fortgeschrieben werden. Ein Übergangs-System liegt beispielsweise in Form eines endlichen Automaten vor. Es besteht dabei aus Zuständen und Übergängen. Ein Übergang führt dabei von einem „von-Zustand“ in einen „nach-Zustand“. Zusätzlich werden die Start-Zustände des Übergangs-Systems als solche markiert.

Für die praktische Anwendung ist besonders von Bedeutung, dass der Zustandsraum nicht nur endlich, sondern auch von handhabbarer Größe ist, also von heutigen Rechnern schnell durchsucht werden kann. Denn Modell-Checker durchsuchen typischerweise den gesamten Zustandsraum, um zu entscheiden, ob eine bestimmt Eigenschaft erfüllt ist [7].

Nach Varró wird ein *Übergangs-System* $TS = (V, Dom, T, Init)$ als 4-Tupel definiert [7]. Dabei bezeichnet

1. $Dom = \{D_1, ..., D_m\}$ eine Menge finiter **Domänen**.

2. $V = \{v_1, ..., v_k\}$ eine Menge von **Zustands-Variablen**, die ihren Wert innerhalb der entsprechenden Domäne annehmen.
3. $T = \{T_1, ..., T_n\}$ eine Menge von **Übergängen** der Form $p \rightarrow v_1 := c_1, ..., v_n := c_n$, wobei p eine boolsche Überwachungsbedingung ist und $v_i := c_i$ eine Änderung der Zustandsvariable v_i spezifiziert.
4. $Init$ den **Startzustand**.

Übergangs-Systeme sind also eine Art „abstrakte Syntax einer Spezifikations-Sprache“ die dazu verwendet werden kann, ein System über einen Zustandsraum in Form einer Kripke-Struktur zu beschreiben [7].

Dabei ist eine *Kripke-Struktur* $KS = (\Sigma, N, I, \sigma)$ ein 4-Tupel, wobei

1. Σ eine Menge von Zuständen ist, die durch alle möglichen Werte der Zustandsvariablen induziert wird.
2. $N \subseteq \Sigma \times \Sigma$ die Übergangsrelation ist, die alle möglichen Übergänge zwischen zwei Zuständen aus Σ beeinhaltet.
3. $I \subseteq \Sigma$ eine Menge von Startzuständen ist.
4. $\sigma : \Sigma \rightarrow 2^{AP}$ eine Markierungsfunktion ist, die jedem Zustand eine Teilmenge der möglichen Aussagen aus der Menge der Aussagen AP zuordnet, die in diesem Zustand gültig sind.

Mit Hilfe des Übergangs-Systems und der Kripke-Struktur können dann Eigenschaften eines Modells gezeigt werden, wodurch sich Modell-Checking dann auch zur Validierung von bestimmten Modell-Transformationen eignet, wie im Folgenden näher erläutert wird.

Validierung mit Modell-Checking Modell-Checking lässt sich zur Validierung von Modell-Transformationen anwenden, wenn es sich bei Quell-Modell und Ziel-Modell jeweils um Graphen, beispielsweise um UML-Diagramme, handelt. Dann können die Graphen als Zustände und die Anwendungen von Regeln als Übergänge zwischen den Zuständen in einem Übergangs-Systems interpretiert werden [7].

Varró definiert eine *Graph-Transformations-Regel* als 5-Tupel $R = \{Lhs, Neg, Rhs, Cond, Assgn\}$. Dabei bezeichnet *Lhs* die Eingabe und *Rhs* die Ausgabe. *Neg* enthält die Bedingungen, die einer Anwendung der Regel entgegen stehen, während für alle Elemente aus *Lhs* und *Neg* zusätzliche Bedingungen *Cond* gelten können und für jedes Element aus *Rhs* zusätzliche Attribute in *Assgn* assoziiert werden können.

Die *Anwendung einer Regel* auf ein Modell M überschreibt dieses Modell, indem sie das Muster, das in *Lhs* definiert wurde, durch das in *Rhs* definierte Muster ersetzt. Das Vorgehen dabei lässt sich nach [7] in sechs Schritte unterteilen.

1. Finden einer Übereinstimmung des im Teil *Lhs* der Regel R definierten Teilgraphen mit einem Element des Modells
2. Überprüfung, ob die Bedingungen aus *Neg* die Anwendung von R verbietet

3. Überprüfung, ob die Bedingungen aus *Cond* auch nach Anwendung von R weiter gelten
4. Entfernen der Teile des Modells M, die mit *Lhs*, aber nicht mit *Rhs* übereinstimmen
5. Einfügen der zusätzlichen Elemente aus *Rhs* in M
6. Aktualisierung der Attribute in Übereinstimmung mit *Assgn*

In dieser Form können Modell-Transformationen von Graphen einfach als Übergangs-Systeme gesehen werden. Die direkte Kodierung von Graph-Transformationen in ein Modell-Checking-Problem führt dabei allerdings zu einem Problem. In der Praxis ist die direkte Kodierung nämlich nicht anwendbar, da die Modell-Checker bei dieser Art der Zustands-Repräsentation sehr schnell mit einem nicht handhabbaren, weil exponentiell großen, Zustandsraum zurechtkommen müssten [7].

Eine einfache Kodierung von Graphen sieht demnach wie folgt aus:

- *Abbilden von Graphen in Felder*
  Um die Modelle in Feldern zu repräsentieren, wird (i) jede Klasse in einem eindimensionalen boolschen Zustands-Feld, (ii) jede Assoziation in einem zwei-dimensionalen boolschen Zustands-Feld und (iii) jedes Attribut in einem eindimensionalen Zustands-Feld als Integer-Variable gespeichert. Um diese Felder mit Index erzeugen zu können, muss die Anzahl der Objekte a-priori begrenzt werden. Dazu wird angenommen, dass Objekte erzeugt werden, indem vorhandene, passive Objekte „aktiviert“ werden und gelöscht werden, indem vorhandene, aktive Objekte „deaktiviert“ werden, also eigentlich alle möglichen Objekte ununterbrochen repräsentiert sind. Als wesentliche Restriktion gilt also, dass keine Graphen mit potentiell unendlich vielen Zuständen repräsentiert werden können, oder - in anderen Worten - das Modell-Checking nur einen endlichen Zustandsraum durchsucht.
- *Abbildung von Transformationsschritten in Übergänge*
  Die Transformationsschritte, also die möglichen Anwendungen von Transformations-Regeln, werden in Übergänge im Übergangs-System überführt. Dabei wird eine einzelne Transformations-Regel durch mehrere verschiedene Übergänge dargestellt, die zusammen alle möglichen Anwendungen der Regel abdecken. Die verschiedenen Übergänge zu einer Regel unterscheiden sich dabei in den verschiedenen Vorkommen der Muster in *Lhs* und sorgen alle dafür, dass die entsprechenden Zustandsvariablen aktualisiert werden.

Würde man die Kodierung hier beenden, würde ein korrektes Modell-Checking-Problem vorliegen, allerdings wäre selbst die Überprüfung von vergleichsweise kleinen Anwendungen mit enormem Rechenaufwand verbunden. Varró nennt dafür als Beispiel eine Anwendung mit 20 Zuständen und 20 Übergängen, die für Modell-Checking etwa 500 boolsche Variablen benötigen würde. Ein Zustandsraum von 2^{500} Zuständen liegt aber weit außerhalb einer realistischen Leistungsfähigkeit heutiger Rechner.

Um die praktische Anwendung des Modell-Checkings bei der Validierung von Modell-Transformationen zu ermöglichen, kommen folgende Optimierungen in Frage:

- *Eliminierung statischer Zustandsvariablen*
  Unter der Annahme, dass die Struktur eines Übergangs-Systems während der Lebenszeit eines Modells unverändert bleibt, können Zustandsvariablen nur für *dynamische* Modell-Elemente erzeugt werden, während die *statischen* Modell-Elemente ohne Zustandsvariablen vorausberechnet werden können.
- *Eliminierung „toter" Übergänge*
  Die naive Übertragung der Regeln in Übergänge kann nach Varró leicht zu redundanten Übergängen führen, für die die Bedingungen nie erfüllt werden können. Trotzdem werden sie jedesmal mituntersucht und verzögern so den Ablauf enorm. Durch Vorausberechnungen können solche Übergänge eliminiert werden.
- *„Filterung" der Eigenschaften*
  Da in der optimierten vorausberechneten Version des Übergangs-Systems nur noch dynamische Elemente vorkommen, sollten auch die Sicherheits-Eigenschaften und die Verklemmungs-Kriterien auf ihre dynamischen Teile reduziert werden.

Die Korrektheit und Vollständigkeit dieser Form der Kodierung wird in [7] gezeigt.

Varró schließt in seiner Arbeit, dass die Validierung von Modell-Transformationen durch Modell-Checking mit der Benutzung von Übergangs-Systemen gute Performanz besitzt, wenn auch praktische Limitationen bezüglich der Größe der Modelle bestehen. Bedeutsame Fehler in der Spezifikation werden aber auch bei kleinen Modellen noch oft durch solche Validierungen entdeckt. Außerdem stellt er in seiner Arbeit fest, dass sich wenige komplexe Transformations-Regeln oft einfacher validieren lassen als eine größere Anzahl an simpleren Regeln [7].

3 Validierung von Modell-Transformationen durch Tests

Der Test von Modell-Transformationen erfolgt nach Küster [4] weitgehend analog zu den bekannten Test-Methoden in der Softwaretechnik. Er unterscheidet dabei zwischen „Testen im Großen" und „Testen im Kleinen", was für ihn, angewandt auf den Test von Modell-Transformationen, bedeutet, dass „Testen im Kleinen" den Test der einzelnen Transformations-Regeln beinhaltet, während „Testen im Großen" den Test der gesamten Transformation erfordert.

Die Herausforderungen sieht Küster dabei vor allem in der Erzeugung geeigneter Testfälle und der „Generierung von *Test-Orakeln*, die das erwartete Ergebnis eines Tests festlegen", wohingegen er die dritte wesentliche Aufgabe beim Test von Modell-Transformationen, nämlich die eigentliche Ausführung der Testfälle in einer geeigneten Testumgebung, als relativ einfach zu bewältigen ansieht.

Allerdings muss man sich zur Generierung der Testfälle der typischen Fehler bewusst sein, die bei der Implementierung von Modell-Transformationen auftraten können.

3.1 Typische Fehler bei der Implementierung von Modell-Tranformationen

Die Erfahrungen zeigen laut Küster, dass vor allem sechs Fehlertypen bei der Implementierung von Modell-Transformationen auftreten. Diese werden im Folgenden nach [4] vorgestellt.

1. *Meta-Modell-Überdeckung*
   Hier treten Fehler auf, wenn die Transformations-Regeln unvollständig sind, also nicht alle Elemente des Meta-Modells überdeckt werden. Als Folge können einige Eingabe-Modelle nicht transformiert werden.
2. *Erzeugung syntaktisch inkorrekter Modelle*
   Wenn Teile einer Transformations-Regel nicht korrekt implementiert werden, kann dies dazu führen, dass erzeugte Modelle nicht der Syntax des entsprechenden Ziel-Meta-Modells entsprechen oder spezifizierte Einschränkungen verletzen.
3. *Erzeugung semantisch inkorrekter Modelle*
   Derartige Fehler entstehen, wenn Transformationsregeln auf Quell-Modelle angewendet werden, auf die sie nicht passen. Das Resultat ist dann zwar oft syntaktisch korrekt, aber keine semantisch korrekte Transformation des Quell-Modells
4. *Konfluenz-Fehler*
   Die Transformation produziert hierbei verschiedene Ausgaben aus gleichen Eingaben, weil die Transformation nicht konfluent ist (siehe oben). Ebenfalls hierunter fallen Fehler, bei denen die Transformation zu „Zwischen-Modellen“ führt, die nicht weiter transformiert werden können, weil die Nicht-Konfluenz unerkannt oder unbehandelt geblieben ist.
5. *Korrektheit der Transformations-Semantik*
   Hierunter fallen Fehler, bei denen die Transformation bestimmte an sie gestellt Anforderungen nicht erfüllt, zum Beispiel die oben genannten Validierungs-Kriterien.
6. *Programmierfehler*
   Fehler, die nicht direkt einer der anderen Kategorien zugeordnet werden können, sondern klassische Kodierungs-Fehler darstellen, fallen unter diesen Typ.

Dabei werden die meisten Fehler, die auf fehlerhafte Kodierung zurückzuführen sind, auch schon indirekt durch Tests der ersten vier Fehlertypen gefunden [4].

Neben Techniken der Fehlerfindung wie Inspektionen, Reviews u.ä., die laut Küster auch auf die Validierung von Modell-Transformationen angewandt werden können, sind vor allem die ersten vier Typen von Fehlern mit klassischen Tests zu finden, während die letzten zwei Fehlertypen durch zukünftige Arbeiten noch weiter abgedeckt werden müssen [4].

Die Herausforderung sieht Küster in allen Fällen darin, „die Testfälle systematisch zu generieren“.

Zur Generierung von Testfällen existieren verschiedene Möglichkeiten, die im Folgenden vorgestellt werden.

3.2 Ansätze zur Generierung der Testfälle

Wie bereits erwähnt, stellen die Generierung von Testfällen und Test-Orakeln die beiden wesentlichen Herausforderungen beim Test von Modell-Transformationen dar. Zur Generierung von Testfällen nennt Küster drei verschiedene Techniken [4].

Meta-Modell-Überdeckung Testen anhand von Meta-Modell-Überdeckung verfolgt das Ziel, die Überdeckung aller möglichen Elemente des Meta-Modells sicherzustellen, also insbesondere die Vollständigkeit der Transformationsregeln nachzuweisen, so dass alle möglichen Quell-Modelle auch zu korrekten Ausgaben transformiert werden können.

Im von Küster vorgestellten Ansatz zur Entwicklung von regelbasierten Modell-Transformationen wird jede Regel in eine Meta-Modell-Schablone überführt.

Die Idee dahinter ist, dass aus einer solchen Meta-Modell-Schablone automatisch Instanzen der Schablone erzeugt werden können, die direkt brauchbare Testfälle darstellen.

Bei der Erzeugung einer Meta-Modell-Schablone aus einer Regel muss dabei die zugrunde liegende Modellierungssprache betrachtet werden, um für jeden Parameter der Regel alle möglichen Werte dieses Parameters zu identifizieren. Dabei muss auch besonders beachtet werden, dass alle abstrakten Elemente konkrete Werte zugewiesen bekommen.

Das Testen der Meta-Modell-Überdeckung geschieht dann nach Küster analog zum klassischen White-Box-Testen, ähnelt also klassischen Tests mit Kenntnissen über Implementierungsdetails. Dabei führt das Erzeugen von Meta-Modell-Schablonen für alle Transformations-Regeln zu einem hohen Grad an Meta-Modell-Überdeckung. Kann dabei gezeigt werden, dass Meta-Modell-Überdeckung für jede Regel der Transformation erreicht ist, dann folgt daraus die Meta-Modell-Überdeckung der gesamten Transformation.

Ein Vorteil des Ansatzes zum Testen von Modell-Transformationen anhand der Meta-Modell-Überdeckung besteht darin, dass nach Küster sowohl die systematische Instanziierung der Meta-Modell-Schablonen als auch das eigentliche Testen automatisiert werden kann, denn nachdem die Meta-Modell-Schablonen definiert wurden, führt die automatische Generierung von Instanzen zu den Schablonen direkt zu einem Satz von Testfällen, die die zur Meta-Modell-Schablone gehörende Transformationsregel testet.

Natürlich ist das Testen anhand der Meta-Modell-Überdeckung vor allen Dingen dazu geeignet, Fehler bezüglich der Meta-Modell-Übderdeckung zu finden

(siehe oben). Dieser Ansatz eignet sich nach Küster aber auch dazu, syntaktische und semantische Fehler sowie Fehler, die durch fehlerhafte Kodierung entstanden sind, zu finden, wobei dabei das Auffinden von semantischen Fehlern nicht mehr automatisch erfolgen kann, sondern voraussetzt, dass alle Resultate manuell betrachtet werden müssen.

Als offener Punkt in Bezug auf das Testen von Modell-Transformationen anhand der Meta-Modell-Überdeckung verbleibt nach Küster die Frage, ob es notwendig ist, dass für jede Regel vollständige Meta-Modell-Überdeckung gezeigt wird, oder ob es unter bestimmten Bedingungen ausreichend ist, wenn einige Regeln nur teilweise Überdeckung vorweisen können, dafür aber sichergestellt wird, dass die Testfälle diese Bedingungen erfüllen.

Abschließend kann das Testen der Meta-Modell-Überdeckung als „mächtiger Mechanismus“ betrachtet werden.

Diese Technik unterliegt aber gleichwohl natürlichen Begrenzungen dadurch, dass alle Testfälle direkt aus jeweils genau *einer* Transformationsregel erzeugt werden. Bedingungen bezüglich der Modell-Transformation, die so formuliert wurden, dass sie mehrere Modell-Elemente betreffen, die nicht durch eine einzige Regel abgedeckt werden, können durch die mit dem Ansatz der Meta-Modell-Überdeckung erzeugten Testfälle nicht überprüft werden.

Generierung von Testfällen aus Einschränkungen Die Validierung von Modell-Transformationen mit Hilfe von Testfällen, die aus Einschränkungen generiert werden, adressiert die Probleme, die beim vorgestellten Test anhand der Meta-Modell-Überdeckung auftreten können, indem die Testfälle direkt aus den Bedingungen, deren Einhaltung dabei nicht testbar war, konstruiert werden.

Diese Bedingungen in Form von Einschränkungen sind in der Regel in den Meta-Modellen der Modellierungssprachen enthalten und entweder in natürlicher Sprache oder mit der Object Constraint Language (OCL) formuliert. Werden sie verletzt, können daraus syntaktische Fehler hervorgehen. Besonders zu beachten sind Einschränkungen, die mehrere Transformations-Regeln betreffen, da Fehler darin mit dem bereits vorgestellten Ansatz der Meta-Modell-Überdeckung kaum erkannt werden können.

Um also Fehler zu entdecken, die in der Verletzung von Einschränkungen resultieren, werden die existierenden Einschränkungen bezüglich der Modelle direkt zur Erzeugung entsprechender Testfälle benutzt.

Dadurch, dass viele Transformationen Modell-Elemente verändern, könnten im Ziel-Modell gewisse Bedingungen verletzt sein. Die erzeugten Testfälle testen dann, ob die im Quell-Modell erfüllten Einschränkungen auch im Ziel-Modell, also nach Anwendung der Transformation, weiterhin erfüllt sind. Dabei können die Einschränkungen sowohl „auf Regel-Level als auch auf Transformations-Level“ überprüft werden.

Zum gezielten und effizienten Testen definiert Küsters die *Abhängigkeit der Einschränkungen von einem Modell-Element*. Danach ist eine Einschränkung *unabhängig* von einem Element des Modells, wenn „die Existenz oder der Wert

der Instanzen des Modell-Elements den *booleschen* Wert der Einschränkung nicht beeinflusst“, ansonsten ist die Einschränkung *abhängig*.

Der eigentliche Testvorgang sieht dann so aus, dass die Elemente des Modells, die durch eine Transformation verändert werden, identifiziert werden, und diese Elemente dann wiederum dazu dienen, die von ihnen abhängigen Einschränkungen zu identifizieren. Für die so identifizierten Einschränkungen können dann Testfälle generiert werden, die die „Validität der Einschränkung unter der Transformation“ überprüfen.

Eine wesentliche Herausforderung bei diesem Ansatz zur Erzeugung der Testfälle besteht dabei laut Küster darin, dass es kompliziert ist, die Modell-Elemente zu identifizieren, die von der Transformation verändert werden. Gelöst werden kann dieses Problem teilweise durch Betrachten der einzelnen Transformations-Regeln oder durch Informationen, die direkt von den Programmierern stammen.

Wurden die Einschränkungen identifiziert, deren Einhaltung getestet werden muss, können diese nach Küster in „positive Einschränkungen“ und „negative Einschränkungen“ unterteilt werden. Dabei erfordern positive Einschränkungen die Existenz bestimmter Modell-Elemente, während negative Einschränkungen die Nicht-Existenz solcher Elemente erfordern. In beiden Fällen können Testfälle kreiert werden, die nach Anwendung der Transformation in einer Verletzung der Einschränkungen resultieren würden, falls die Transformation fehlerhaft ist.

Benutzung von Regel-Paaren Eine andere Fehlerquelle liegt nach Küster im Zusammenspiel verschiedener Regeln. Denn auch nachdem gezeigt wurde, dass das Meta-Modell vollständig überdeckt wird und die Einschränkungen auch nach erfolgter Transformation bestehen bleiben, können noch Transformations-Regeln vorhanden sein, die sich gegenseitig unerwünscht beeinflussen, ob direkt oder indirekt.

So kann die Anwendung einer Regel auf ein Modell-Element bei mehrschrittigen Transformationen dazu führen, dass eine weitere Regel auf das selbe Element nicht mehr angewandt werden kann, obwohl dies für eine korrekte Transformation erforderlich wäre. Dabei erfordert das Validierungskriterium der Konfluenz (siehe oben), dass die Anwendung von gleichen Transformations-Regeln auf gleiche Modell-Elemente zu den gleichen Resultaten führt, ganz egal in welcher Reihenfolge die Regeln angewandt werden.

Verletzungen des Konfluenz-Kriteriums können nach Küster zu „sehr subtilen Fehlern führen, die schwer zu entdecken und reproduzieren sind“ und so sowohl zu syntaktischen als auch semantischen Fehlern im Ziel-Modell führen können.

Zur Vermeidung solcher Fehler wird das Konzept der *parallelen Unabhängigkeit* zweier Regeln vorgestellt. Danach sind zwei Regeln parallel unabhängig, wenn alle möglichen Anwendungen der zwei Regeln sich gegenseitig nicht behindern, das heißt es gilt immer, dass wenn eine Regel r_1 anwendbar war, bevor eine Regel r_2 angewendet wurde, dann ist r_1 auch nach der Anwendung von r_2 anwendbar. Das Validierungskriterium der Konfluenz kann demnach dann verletzt werden, wenn zwei Regeln nicht parallel unabhängig sind.

Zur Erkennung von Regel-Paaren, auf die dies zutrifft, schlägt Küster die Konstruktion sogenannter „kritischer Regelpaare“ vor. Seine Idee eines kritischen Paares ist dabei, die Transformations-Schritte, die zu einem Konflikt führen können, in einem „minimalen Kontext“ zu analysieren und ein eventuell gemeinsames Nachfolge-Modell der beiden Regeln zu erzeugen.

Dazu werden systematisch alle möglichen überlappenden Modelle für zwei Regeln konstruiert, also solche Modelle, auf die beide Regeln angewandt werden könnten. Handelt es sich dabei um Regeln, die das Konfluenzkriterium verletzten, kann ein solches kritisches Paar durch systematisches Testen auf einen Konfluenzfehler hin entdeckt werden.

Zur systematischen Konstruktion der überlappenden Modelle schlägt Küster ein Verfahren vor, bei dem ausgehend von den linken Seiten zweier Regeln, also den Quell-Modellen, auf die die Transformations-Regel angewendet wird, „alle möglichen Überlappungen von Modell-Elementen berechnet“ werden. „Basierend auf diesen Überlappungen wird ein Modell konstruiert, das die beiden Modelle am überlappenden Modell-Element vereinigt.“ Sofern dieses überlappende Modell syntaktisch korrekt ist, kann es direkt als Testfall verwendet werden, andernfalls wird es verworfen.

Testverfahren, die auf der Benutzung von Regel-Paaren basieren, können weitgehend automatisiert werden. Die automatische Erkennung von Konfluenzfehlern erfordert allerdings menschliche Eingriffe, wenn kein automatisches Verfahren verfügbar ist, das den Vergleich der Resultaten erlaubt, die die Ausführung der konstruierten Testfälle liefert.

Zusammenfassend schließt Küster, dass die Verwendung der drei vorgestellten Verfahren zum White-Box-Testen von Modell-Transformationen, also zum Testen von Transformationen ohne Kenntnisse über deren Implementierung, „signifikant“ zur Verbesserung der Qualität von Modell-Transformationen beitragen kann. Dabei hat er die Erfahrung gemacht, dass sowohl die Tests anhand der Meta-Modell-Überdeckung als auch die Generierung von Testfällen aus Einschränkungen eine Reihe von Fehlern aufdecken konnten, während die Fehler, die durch die Benutzung von Regel-Paaren gefunden wurden, quantitativ geringer waren [4].

4 Validierung von Statechart-Transformationen

Die Generierung von Code aus Spezifikations-Sprachen wie UML ist ein „wichtiger Aspekt der *Model Driven Architecture*“ [1]. Dabei liegt der Vorteil von UML-Spezifikationen darin, dass Software im Modell beschrieben werden kann, während die Aufgabe der eigentlichen Code-Generierung von automatischen Modell-To-Text-Transformationen, also Transformationen vom Modell (das beispielsweise im UML-Standard vorliegt) zur Programmiersprache (beispielsweise Java), übernommen wird. Zur Sicherstellung von korrekter Software und korrektem System-Verhalten ist es dabei unumgänglich, dass die Semantik des ursprüng-

lichen UML-Modells auch im durch Modell-Transformation erzeugten Code erhalten bleibt.

Im Folgenden wird eine Arbeit von Blech, Glesner, Leiter [1] vorgestellt, die die Korrektheit der Transformation mit formaler Validierung sicherstellt. Es geht dabei beispielhaft um die „Transformation vereinfachter Statecharts in eine Java-ähnliche Sprache“ und um die Validierung des zugrunde liegenden Transformations-Algorithmus' durch Formalisierung der Semantik von Statecharts und der Ziel-Sprache Java.

4.1 Statecharts

Statecharts sind nach Blech eine „Erweiterung von endlichen Automaten, die in vielen Werkzeugen und in der Praxis breite Anwendung findet“. Dabei besteht die Erweiterung darin, dass Statecharts im Gegensatz zu herkömmlichen endlichen Automaten „hierarchische und parallele Kompositionen“ von Zuständen zulassen und auch parallele Übergänge enthalten.

Sie finden als Teil des UML-Standards vor allem dann ihren Einsatz, wenn es um die Modellierung komplexen, dynamischen Verhaltens geht. Im UML Standard können Statecharts dann so modelliert werden, dass ihre Zustände und die Übergänge zwischen diesen mit Aktionen dekoriert werden. Diese Aktionen in Form von Anweisungen in einer imperativen Sprache erhöhen dabei die Ausdrucksstärke der Statecharts [1].

4.2 Code-Generierung aus Statecharts

Die zahlreichen Werkzeuge, die nach [1] bereits zur Code-Generierung aus Statecharts existieren, benutzen meist einen von drei wesentlichen Ansätzen, die sich alle an Methoden zur Code-Generierung aus endlichen Automaten heraus orientieren. Diese drei Ansätze sind:

- *Switch/Case-Schleife*
  Dieser Ansatz ist der einfachste und erzeugt ineinander verschachtelte Switch/ Case-Ausdrücke, die entsprechend der Zuständen und Ereignissen springen. Innerhalb eines Sprungs wird dann der spezifische Code des Übergangs in Form der erwähnten Aktionen ausgeführt. Außerdem wird der Zustand neu gesetzt, so dass er dem Ziel-Zustand des Übergangs entspricht. Hierarchische und nebenläufige Strukturen werden durch Rekurrenzen erreicht.
- *Tabellengetriebener Ansatz*
  Beim tabellengetriebenen Ansatz werden die Aktionen, die von einem Ereignis in einem bestimmten Zustand ausgelöst wurden, in einer verschachtelten Zustand/Eingabe-Tabelle gespeichert. Die Einträge in der Tabelle enthalten dann die zugehörigen Ausgaben und Folgezustände.
- *Virtuelle Methoden*
  In objektorientierten Systemen sind tief verschachtelte Switch/Case-Blöcke nicht wünschenswert, besonders dann, wenn der daraus generierte Code noch manuell modifiziert oder manuell gewartet werden soll. In so einem Fall kann

jeder Zustand als Klasse repräsentiert werden. Die Eingaben und Ereignisse werden dabei als Aufrufe von virtuellen Methoden in den entsprechenden Zustandsklassen implementiert.

Die Validierung von Code-Generierung erfolgt in [1] nur bezogen auf den ersten Ansatz.

4.3 Semantik von Statecharts

Zur Validierung von Statechart-Transformationen ist es notwendig, die Semantik der Statecharts formalisieren zu können. Dazu wird das Verhalten eines Statecharts als Übergangs-System modelliert, wobei die Zustände des Übergangs-Systems die möglichen Konfigurationen des Statecharts repräsentieren. Eine Konfiguration des Statecharts ist dabei „eine maximale Menge an Zuständen, die im Statechart gleichzeitig erreicht werden können“[1].

Das Verhalten des Statecharts wird dann durch die Zustands-Übergangs-Funktion beschrieben, die abhängig von den Eingaben die Übergänge zwischen den Konfigurationen definiert. Das Übergangs-System sollte dabei unmittelbar, also ohne zeitliche Verzögerung, auf Eingaben reagieren. Eingehende Ereignisse und dadurch ausgelöste ausgehende Aktionen sollen also gleichzeitig erfolgen. Diese Eigenschaft der *Synchronität* ist in der Realität kaum zu erreichen. Zur Umgehung und Lösung des Problems sei auf die Ansätze verwiesen, die in [1] vorgestellt werden und einen Einsatz der Formalisierung von Statecharts als Übergangs-Systeme in der Praxis erlauben. Deutlich wird dabei, dass die Semantik von Quell-Meta-Modellen nicht einfach zu fassen ist.

4.4 Validierung von Statechart-Transformationen von UML nach Java

Die Validierung der Modell-To-Text-Transformation von UML nach Java wird von [1] vorgenommen, indem die „semantische Äquivalenz“ der Statecharts und ihrer Implementierung in Java gezeigt wird. Semantische Äquivalenz bedeutet dabei, dass Modell und Programm das „gleiche beobachtbare Verhalten“ zeigen, also zum Beispiel bei einer deterministischen Spezifikation und gleichen Eingaben die gleichen Folgen von Zuständen durchlaufen.

Die Validierung der Modell-Transformation geschieht dann nach dem in Abbildung 2 dargestellten Prinzip. Es werden also die Semantiken der Statechart-Implementierung und des implementierten Java-Quellcodes miteinander verglichen. Die Validierung erfolgt also über einen Vergleich der Semantiken.

Um semantische Äquivalenz feststellen zu können, wird eine Sementik der Statecharts und eine Semantik der Programmiersprache benötigt. Außerdem ist eine Zuordnung von vorliegenden Statecharts und Programmen zu ihren Semantiken erforderlich. Dann kann die semantische Äquivalenz gezeigt werden, also dass die Semantik des ursprünglichen Systems während der Modell-Transformation erhalten bleibt. Hier heißt das, dass die beobachtbaren Zustände des Statecharts und des generierten Programms die gleichen sein müssen.

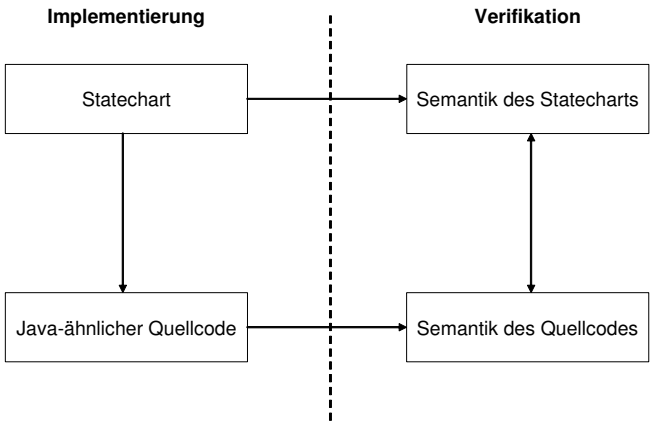


Abbildung 2. Abb. 2: Verifikation der Code-Generierung nach [1]

Formalisiert ausgedrückt, müssen Statechart und generiertes Programm sich *bisimulieren*, also ihre Semantiken in einer *Bisimulations-Relation* stehen.

Bei der Definition einer Bisimulations-Relation nach [1] kommen wieder die schon für das Modell-Checking-Problem definierten Kripke-Strukturen zum Einsatz. Eine Bisimulations-Relation ist damit definiert wie folgt:

Seien $M = (\Sigma, N, I, \sigma)$ und $M' = (\Sigma', N', I', \sigma')$ zwei Kripke-Strukturen. Eine Relation $B \subseteq S \times S'$ ist eine *Bisimulations-Relation* zwischen M und M', genau dann wenn für alle $s \in \Sigma$ und $s' \in \Sigma'$ mit $B(s, s')$ gilt:

1. $\sigma(s) = \sigma'(s')$
2. Für jeden Zustand s_1 mit $N(s, s_1)$ existiert ein s'_1 mit $N'(s', s'_1)$ und $B(s_1, s'_1)$.
3. Für jeden Zustand s'_1 mit $N(s', s'_1)$ existiert ein s_1 mit $N'(s, s_1)$ und $B(s_1, s'_1)$.

Eine Kripke-Struktur M zur Repräsentation der Semantik von Statecharts wird dabei aus den Konfigurations-Übergängen des entsprechenden Übergangs-Systems erhalten. Diese Übergänge entsprechen dann der Übergangsrelation N in der Kripke-Struktur. Die Zustandsmenge Σ der Kripke-Struktur entspricht den möglichen Konfigurationen des Statecharts, wobei I die initiale Konfiguration ist. Die Kripke-Struktur kann auch unendlich viele Zustände besitzen, um nicht-terminierenden Statecharts zu entsprechen [1].

Auch die Semantik eines Programms in einer höheren Programmiersprache kann durch eine Kripke-Struktur M' beschrieben werden. Dazu wird die Semantik so spezifiziert, dass die Ausführung jeder einzelnen Instruktion des Programms atomar ist, und die Semantik sich aus den Zuständen und Zustandsübergängen ergibt. Ein Zustand besteht dann aus der Ausführung der aktuellen atomaren Instruktion, dem aktuellen Speicherinhalt und der aktuellen Belegung der Variablen. Es ist also Σ die Menge der erreichbaren Zustände während der Ausführung

des Programms, während N mögliche Zustandsübergänge und die Bedingungen, unter welchen diese stattfinden, repräsentiert. I ist der Startzustand des Programms und σ ordnet den Zuständen ihren jeweiligen beobachtbaren Teil zu.

Damit bisimulieren sich zwei Programme bzw. ein Statechart und ein Programm, wenn eine „Bisimulations-Relation B zwischen ihnen existiert und die Startzustände der beiden Programme Teil dieser Relation sind“[1].

Zusammenfassend gilt also, dass eine Bisimulations-Relation B Verhaltens-Äquivalenz zwischen einer als Kripke-Struktur M formulierten Semantik eines Statecharts und einem als Kripke-Struktur M' formulierten Programm ausdrückt. Das Äquivalenz-Kriterium kann dabei frei gewählt werden, so dass man sich zum Beispiel auf die Eingaben und Ausgaben der Zustände beschränken kann und keine weiteren Variablen betrachten muss. Die Notation der Bisimulation stellt also ein formales Kriterium da, mit dem beurteilt werden kann, ob Statechart und Programm das gleiche Verhalten besitzen.

Die Notation von Statechart und Programm als Kripke-Struktur erlaubt es dabei sogar, sowohl von terminierenden wie auch von nicht-terminierenden Statecharts und Programmen die Semantik formal zu erfassen und auf Äquivalenz zu überprüfen [1].

Soll die Korrektheit eines Algorithmus' zur Code-Generierung aus Statecharts bewiesen werden, so muss also gezeigt werden, dass Statechart und generierter Programm-Code bei gleichen Eingaben die gleichen Folgen von beobachtbaren Zuständen durchlaufen. Dies ist aber „genau das, was ein Bisimulations-Beweis tut“ [1].

Zur Durchführung des eigentlichen Beweises der Bisimulation werden dann Theorem-Beweiser eingesetzt. Theorem-Beweiser, von denen in [1] *Isabelle/HOL* als Beispiel genannt wird, können im Allgemeinen dazu benutzt werden, Spezifikationen zu erzeugen oder Lemmas zu formulieren. Darauf können dann Theoreme formuliert und deren Korrektheit bewiesen werden.

Im Gegensatz zur automatischen Validierung von Modell-Transformationen mit Modell-Checking erfordert der Einsatz von Theorem-Beweisern aber immer noch ein gewisses Maß an Interaktion mit dem Anwender. Denn die Spezifikationen müssen zum Beweis ihrer Korrektheit sehr sorgfältig formuliert sein, was menschliche Eingriffe erfordert [1].

Die Formulierung von Bisimulation kann in Theorem-Beweisern dabei auf verschiedene Arten erfolgen. Zur genaueren Vorgehensweise bei dem Beweis von Bisimulation mit Theorem-Beweisern sei hier aber auf [1] verwiesen.

Im Unterschied zum bereits vorgestellten Modell-Checking können Bisimulations-Beweise mit einem Theorem-Beweiser in jedem Fall „komplette semantische Äquivalenz validieren und nicht nur bestimmte Aspekte oder Bedingungen“. Der Korrektheits-Beweis einer Spezifikation kann dabei mit unter sehr hohen Aufwand erfordern, aber oft auch Fehler aufdecken, die sonst übersehen worden wären [1].

5 Fazit

Die Qualität von Modell-Transformationen kann durch Validierung anhand von anerkannten Kriterien gesichert werden. Zur Sicherstellung der syntaktischen und semantischen Korrektheit von Modell-To-Modell- und Modell-To-Text-Transformationen gibt es dabei verschiedene Ansätze. Für die automatische Validierung mit formalen Werkzeugen ist Modell-Checking gut geeignet, da es unter Voraussetzung einiger Optimierungen gute Performanz bei kleineren und mittleren Modellen besitzt. Bestimmte Kriterien und (Teil-)Transformationen können so weitgehend automatisch validiert werden, was geringen Aufwand und die Möglichkeit des breiten Einsatzes bedeutet.

Testverfahren können eine Reihe von typischen Fehlern bei der Implementierung von Modell-Transformationen aufdecken und dabei weitgehend automatisiert ablaufen. Sie tragen damit zu deutlichen Qualitätsverbesserungen der Transformationen bei und können insbesondere auch die Einhaltung von Einschränkungen in den Spezifikationen sicherstellen.

Gerade die Generierung von Programmcode aus UML-Modellen ist ein wichtiges Anwendungsgebiet von Modell-Transformationen in der Praxis. Formale Verfahren erlauben dabei die Formulierung der Semantik sowohl des UML-Modells als auch des Programmcodes als Kripke-Struktur, so dass die Äquivalenz von Quell-Modell und Ziel-Modell mit Bisimulations-Relationen automatisch mit Hilfe von Theorem-Beweisern bewiesen werden kann. Auf diese Art und Weise kann dann die semantische Äquivalenz über die gesamte Transformation gezeigt werden, dafür ist der Aufwand oft sehr hoch, was den Einsatz in der Praxis auf sicherheitskritische Teile beschränkt.

Literatur

1. Blech, Jan Olav, Glesner, Sabine, Leitner, Johannes: Formal Verification of Java Code Generation from UML Models. Institut for Program Structures and Data Organization. University of Karlsruhe, 76128 Karlsruhe, Germany, Fujaba Days, 2005
2. Csertan, Gyoergy, Huszerl, Gabor, Majzik, Istvan, Pap, Zsigmond, Pataricza, Andras, Varro, Daniel: VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE 2002, 1527-1366/02
3. Garcia, Miguel, Mueller, Ralf: Certification of Transformation Algorithms in Model-Driven Software Development. in: GI-Edition Lecture Notes in Informatics, Software Engineering 2007, ISBN 978-3-88579-199-7, p.107-118.
4. Kuesters, Jochen M., Abd-El-Razik, Mohamed: Validation of Model Transformations - First Experiences using a White Box Approach. Proceedings of the 9th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, 2006.
5. Kuesters, Jochen M.: Systematic Validation of Model Transformations. Proceedings of the 3rd Workshop in Software Model Engineering at7th International Conference on the UML, 2004.
6. de Lara, Juan, Taenzer, Gabriele: Automated Software Transformation and Its Validation Using AToM and AGG. In: A. Blackwell et al. (Eds.): Diagrams 2004, LNAI 2980, pp. 182-198, 2004. Springer-Verlag Berlin Heidelberg 2004 Escuela Polytecnica Superior,Ingenieria Informatica. Universidad Autonoma de Madrid. Computer Science Department. Technical University of Berlin, Germany
7. Varro, Daniel: Automated Formal Verification of Visual Modeling Languages by Model Checking. Department of Measurment and Information Systems Budapest University of Technology and Economics Journal of Software and Systems Modelling, 2003, Submitted to the Special Issue on Graph Transformations and Visual Modelling Techniques.
8. Varro, Daniel: Automated Formal Verification of Model Transformations. CSDUML, Workshop on Critical Systems Development in UML, September 2003, San Fransisco, Technical Report TUM-I0323 pp. 63-78 - inf.mit.bme.hu