Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals
in den Bereichen Softwareentwicklung,
Internettechnologie und IT-Management aktuell
und kompetent relevantes Fachwissen über
Technologien und Produkte zur Entwicklung
und Anwendung moderner Informationstechnologien.

Siegfried Nolte

QVT – Operational Mappings

Modellierung mit der Query Views Transformation

Springer

Siegfried Nolte
Beethovenstr. 57
22941 Bargteheide
siegfried.nolte@alice-dsl.net

ISBN 978-3-540-92292-6 e-ISBN 978-3-540-92293-3
DOI 10.1007/978-3-540-92293-3
Springer Heidelberg Dordrecht London New York

Xpert.press ISSN 1439-5428

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

© Springer-Verlag Berlin Heidelberg 2010
Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.
Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Einbandentwurf: KünkelLopka, Heidelberg

Gedruckt auf säurefreiem Papier

Springer ist Teil der Fachverlagsgruppe Springer Science+Business Media (www.springer.de)

Vorwort

Warum ein Buch über QVT Operational Mappings?

Beim Paradigmenwechsel von der strukturierten zur objektorientierten Software-Entwicklung Mitte der Neunziger gab es im Grunde genommen zwei Wege, sich den neuen Methoden und Techniken zu nähern: zum einen den datenorientierten, ausgehend von der Lehre der Datenbanken und Datenmodellierung, zum anderen den sprachlichen, ausgehend von den Programmiersprachen, die zur der Zeit den Weg der objektorientierten Entwicklung bereiteten. So ähnlich verhält es sich auch mit dem Thema des modellgetriebenen Vorgehens. Auch hier gibt es grundsätzlich zwei alternative Blickrichtungen, die zum einen auf die modellgetriebene Code-Entwicklung mit generativen Techniken und zum anderen auf den Weg der Entwicklung über Modellierung und Modelltransformation ausgerichtet sind. Letzteres ist der Vorschlag, den die *Object Management Group* (OMG) mit dem *Model Driven Architecture*-Konzept (MDA) propagiert und den ich in diesem Buch konsequent aufnehmen und verfolgen möchte.

Die OMG ist ein herstellerunabhängiges Gremium, welches den Auftrag hat, objektorientierte Techniken und Technologien zu standardisieren. So gibt es zum Beispiel die *Unified Modeling Language* (UML) und das *Meta Object Facility*-Konzept (MOF), die sich als OMG-Standards etabliert haben. Das eine ist eine formale, einheitliche, universelle Modellierungssprache, das andere beschreibt einen Ansatz, auf der Basis von formalen Modellen formale Modellierungssprachen zu entwickeln. Beides dient als Fundament für das MDA-Konzept, welches ebenfalls von der OMG als Standard herausgegeben worden ist.

Bisher handelte es sich um Bausteine zur Modellierung, sicher eine der Säulen eines modellgetriebenen Vorgehens. Die MDA geht allerdings weiter und schlägt neben der Modellierung die Transformation von Modellen vor, um aus einer Modellierungsebene in eine folgende zu gelangen, bis hin zu einer modellgestützten Code-Generierung. Sprachen zur Modellierung sind definiert und eingeführt, mindestens die oben erwähnte UML. Was in diesem MDA-Konzept noch fehlte, sind Sprachen zur Modelltransformation.

Von anderer Seite, zum Beispiel aus der Szene der modellgetriebenen Software-Entwicklung, sind Generierungssprachen entwickelt und eingeführt worden, die durchaus eine gewisse Verbreitung und Anerkennung gefunden haben. Doch auch die OMG hat die Lücke mit der im April 2008 als Standard veröffentlichten Spezifikation MOF/QVT (*Query Views Transformation*) mittlerweile geschlossen.

Die QVT ist ein Konzept, in dem drei alternative Sprachen angeboten werden, um auf verschiedene Weise eine Transformation von Modellen beschreiben und durchführen zu können. Die *Operational Mappings* ist eine von ihnen. Nun ist es meiner Meinung nach sehr schwierig, auf der Basis der OMG-Spezifikationen neue Techniken und Technologien wie die der Transformation zu erlernen – ein Problem, dem ich mit diesem Fachbuch Abhilfe schaffen möchte.

Meine ersten intensiven Kontakte mit Modelltransformationen mit einer operationalen Sprache hatte ich 2006. Borland hatte mit Together 2006 als einer der ersten Hersteller auf der Basis des bewährten Modellierungswerkzeugs ein MDA-Werkzeug auf den Markt gebracht, welches Operational Mappings unterstützte, allerdings in einer frühen Ausprägung, die nur sehr schwer zu erlernen war, da es weder von Borland geeignete Literatur gab, noch die QVT-Spezifikation zum Erlernen des Together-Dialektes taugte. Dies war einmal mehr für mich eine Motivation, selbst ein Fachbuch zu dem Thema anzubieten, ein Buch, in dem ich mich recht konsequent auf die Seite der OMG gestellt habe.

Auch von Borlands Together – mittlerweile in der Version 2008 – ist zu sagen, dass sich der herstellerspezifische QVT/OM-Dialekt etwas mehr der standardisierten Fassung angenähert hat, so dass die hier entwickelten Beispiele auch mit Together anwendbar sein sollten. Aber nicht nur aus dem Hause Borland gibt es QVT-Werkzeuge. Einige weitere werde ich vorstellen.

Für wen ist das Buch gedacht?

Modellierung von Sachverhalten der realen Welt ist ein wesentlicher Bestandteil des ingenieursmäßigen Software-Entwicklungsprozesses. Schwerpunkt dieses Buches ist jedoch nicht die Modellierung mit einer Modellierungssprache, und auch die Generierung von Code auf der Basis von Modellen steht nicht im Fokus der Betrachtung. Modelle, die mit formalen Modellierungssprachen erarbeitet worden sind, sind Mittel zum Zweck, um daraus neue und konsistente Modelle abzuleiten – mittels Transformation. Und vor dem Hintergrund, dass die Entwicklung von komplexen Anwendungen in der Regel über mehrere aufeinanderfolgende Phasen erfolgt, wird die systematische und modellgetriebene Entwicklung auch und gerade bei einem Phasenübergang mit Einsatz von Modelltransformationen von zunehmender Bedeutung sein. Erreichen möchte ich also auf jeden Fall IT-Architekten, Analytiker und Designer, zu deren wesentlichen Aufgaben es gehört, reale Sachverhalte zu beschreiben, zu strukturieren und in abstrahierter Form einer weiteren Entwicklungsarbeit zuzuführen.

Wenn auch das Thema MDA nicht unbedingt im Mittelpunkt steht, so möchte ich doch verantwortlichen Projektleitern für IT-Projekte wie auch IT-Führungskräften einen Einblick in den Dialog zwischen Modellierung und Transformation geben. Zum Thema MDA gibt es weiterführende Literatur, die ich hiermit ergänzen und erweitern möchte, und das Thema UML ist sicher recht erschöpfend behandelt. Ebenso würde ich mir wünschen, dass auch Entwickler, die sonst eher einen Zugang zu Programmiersprachen und sprachlich repräsentierten Darstellungsweisen haben, durch die Lektüre dieses Buches einige Anregungen finden für neue Wege und neue Horizonte.

Dieses Buch diskutiert das Thema Modelltransformation mit *Operational Mappings*, wie ich hoffe, ergiebig und ausführlich an vielen Beispielen, die sämtlich mit den zurzeit verfügbaren QVT-Werkzeugen erarbeitet worden sind, so dass ich denke, ein umfassendes Lehrwerk geschaffen zu haben, das nicht nur zum theoretischen Erlernen, sondern auch zum begleitenden Experimentieren und Üben geeignet ist. Grundkenntnisse der Modellierung mit UML, insbesondere im Zusammenhang mit Modellen und Metamodellen, setze ich dabei voraus, Erfahrungen mit höheren Programmiersprachen sind sicher hilfreich.

Wie sollte man es lesen?

Das erste Kapitel, die Einleitung, soll einen Überblick geben und eine Einordnung ermöglichen. Das nächste Kapitel beschäftigt sich umfassend mit der Entwicklung und Repräsentation von Metamodellen. Ein umfassendes Verständnis von Metamodellen ist wesentlich für die weitere Arbeit mit der Transformationssprache, da QVT als MOF-Konzept grundsätzlich metamodellbasiert ist. Demjenigen, der sich zunächst nur mit den Zusammenhängen und den Einordnungen beschäftigen möchte, empfehle ich ein intensiveres Studium der einleitenden Kapitel; die Beschreibung und das Arbeiten mit der Sprache, kann dann ein wenig oberflächlicher gelesen werden.

Das Kapitel 3 geht nun in die Tiefe und beschreibt die Transformationssprache *Operational Mappings* auf eine recht formale, an der Syntax ausgerichtete Weise. Das ist wahrscheinlich, auch wenn ich mich um spannende Formulierungen bemüht habe, etwas mühsam zu lesen. Allerdings habe ich damit bezweckt, einerseits auf die OCL-Grundlagen der QVT einzugehen und andererseits die Sprachkonzepte vollständig zu präsentieren und zu erläutern. An Beispielen soll es nicht fehlen, diese behandeln im Wesentlichen das berühmte „HelloWorld“.

Kapitel 4 schließlich beschäftigt sich mit zwei komplexen und zusammenhängenden Beispielen, einmal dem Standardbeispiel der OMG-Spezifikation – UmlToRdbm –, auf der Basis eigener simpler Metamodelle, und zum anderen einem Beispiel – UML2EJB – mit pragmatischeren Modellen auf der Grundlage des UML2-Metamodells. Beide Beispiele werden Schritt für Schritt entwickelt, so dass man das Kapitel sequentiell abarbeiten sollte.

Meiner Meinung nach ist dieses Vorgehen recht gut geeignet, um einem eher praktisch veranlagten Leser einen schnelleren Zugang zu dem Thema zu eröffnen, den man sich dann nach und nach mit der Methode „Versuch und Irrtum“ erschließen kann. Ein so veranlagter Leser wird vielleicht mit diesem Kapitel einsteigen und die erforderlichen Grundlagen bedarfsgerecht nachlesen wollen. Zuletzt werde ich einige fortgeschrittene Konzepte erläutern, zum Beispiel die Arbeit mit UML-Profilen, BlackBoxes etc., die nicht einfach, aber sicher spannend sind.

Wem bin ich zu Dank verpflichtet?

Einen großen Anteil an der Entstehung dieses Buches hat der Springer-Verlag, dem ich für die Zusammenarbeit herzlich danken möchte. Einen weiteren erheblichen Anteil daran, dass dieses Buch entstehen konnte, wie es ist, haben die beiden wichtigsten Frauen in meinem Leben, Konstanze und Christine, die sich während der Arbeiten an diesem Buch das ein oder andere Mal durch frühe Fassungen hindurch gearbeitet haben. Als „Nicht-IT-Experten“ gehören sie nicht gerade zur klassischen Zielgruppe; und dafür, dass sie mich und die Entstehung des Buches geduldig ertragen und mit Kritik und Anregungen unterstützt haben, bin ich ihnen sehr dankbar.

Letztendlich ist auch zu erwähnen, dass verschiedene Initiativen und Produkthersteller dazu beigetragen haben, dass es nicht nur bei einer theoretischen Ausarbeitung geblieben ist, sondern dass ich im Laufe der Arbeiten an dem Buch viele Beispiele entwickeln und diskutieren konnte. Im Einzelnen zu nennen sind die Sourceforge-Initiative mit der Herausgabe des Produktes SmartQVT, die Firma Borland für die Freigabe des Produktes Operational QVT (QVTO) und die *Eclipse Modeling Toolkit*-Initiative für die Integration von QVTO in die Eclipse-Plattform. Für die Beispiele im Kapitel „Metamodelle“ ist das freie UML-Werkzeug Topcased eingesetzt worden. Bei fast allen Abbildungen handelt es sich um UML-Diagramme, die mit dem Werkzeug MagicDraw der Firma NoMagic Inc. erstellt worden sind. Allen Herstellern und Initiativen gilt mein Dank.

Und schließlich möchte ich natürlich auch der Gemeinde der Leser danken, insbesondere dann, wenn ich mit anregender Kritik versehen werde.

Inhaltsverzeichnis

1 Einführung ... 1

1.1 Eine kurze Geschichte der modellgetriebenen Software-Entwicklung 1

1.1.1 Strukturierte Software-Entwicklung ... 4

1.1.2 Objektorientierte Software-Entwicklung ... 4

1.2 Die berühmten Akronyme der OMG ... 5

1.2.1 UML – Unified Modeling Language ... 6

1.2.2 MDA – Model Driven Architecture ... 9

Grundbegriffe der Model Driven Architecture ... 11

Transformation ... 14

1.2.3 MOF – Modelle und Metamodelle ... 17

1.2.4 QVT – Query Views Transformation ... 19

Deskriptive Sprachen ... 21

Imperative Sprachen ... 21

1.3 Zusammenfassung und Ausblick ... 22

1.3.1 Hinweise zur Notation ... 24

1.3.2 Werkzeuge ... 25

2 Modelle und Metamodelle ... 27

2.1 Die Metamodelle `SimpleUML` und `SimpleRDBM` ... 28

2.1.1 Das Metamodell SimpleUML ... 29

2.1.2 Das Metamodell SimpleRDBM ... 32

2.2 Serialisierung der Metamodelle ... 34

2.2.1 Deklaration der Metamodelle als QVT-Datenstruktur ... 34

2.2.2 QVT-Datenstrukturen im EMOF/XMI-Format ... 37

2.2.3 Die Verwendung der Metamodelle ... 44

Variante 1: *Inline*-Deklaration von QVT-Datenstrukturen ... 44

Variante 2: Benutzung von extern definierten Metamodellen ... 44

Variante 3: Metamodelle im Eclipse-Kontext ... 45

2.2.4 Werkzeugunterstützung 45
Schritt 1: Modellierung 46
Schritt 2: Export des Modells 47
Schritt 3: Deployment der Plugins 48

3 Operational Mappings – die Sprache 53
3.1 HelloWorld als QVT-Applikation 53
3.2 Die *Operational Mappings*-Plattform SmartQVT 55
3.2.1 Aufbau der SmartQVT-Transformationsumgebung 55
3.2.2 Exemplarischer Aufbau von QVT-Projekten 57
3.2.3 Entwicklung und Durchführung von Transformationen 59
3.3 Allgemeiner Aufbau von *Operational Mappings*-Scripten 63
3.3.1 Zusammenfassung 67
3.4 OCL- und QVT-Grundlagen 68
3.4.1 OCL- und QVT-Datentypen 69
Primitive OCL-Datentypen 69
Komplexe OCL-Datentypen 69
Komplexe QVT-Datentypen 69
Definition eigener Datentypen 70
3.4.2 Deklaration von Variablen 71
3.4.3 Operatoren 72
3.4.4 Imperative QVT-Ausdrücke 74
Logging 74
Blöcke 74
Bedingte Ausdrücke 75
compute-Ausdruck 76
Schleifen 76
for-Iteration 77
assert-Ausdruck 77
Exception-Ausdruck 77
3.4.5 Beispiele von imperativen QVT-Codeabschnitten 78
3.4.6 Operationen auf Sammlungstypen 79
Ein Einblick in die OCL-Standardbibliothek 79
Eine Auswahl von QVT-Standardfunktionen 81
Selektion mit QVT-Standardfunktionen 82
QCL-Selektionstechniken 83
3.5 Operationale Transformationen 86
3.5.1 Definition von Metamodellen mit QVT-Sprachmitteln 86
Definition von Metamodellen 87
Benutzung von Metamodellen in Transformationen 91
3.5.2 Transformationen 93

3.5.3 Mapping-Operationen ... 96
Mapping-Signaturen ... 98
Mapping-Anweisungsteil ... 101
Vorbedingungen und Invarianten ... 103
3.5.4 Erzeugung von Objekten ... 104
Variablen und Objekte ... 104
Objekterzeugung mittels *Inline Mapping* ... 106
Objekterzeugung mittels Konstruktoroperationen ... 108
3.5.5 Helper- und Anfrage-Operationen ... 110
3.5.6 Intermediate Data – Dynamische Metaobjekte ... 112

4 Operational Mappings – Anwendungen ... 117
4.1 UML2RDBM ... 117
4.1.1 Vorbereitung der Transformation ... 119
4.1.2 Entwicklung der Mapping-Operationen ... 122
4.1.3 Behandlung primitiver und strukturierter Datentypen ... 125
Übernahme von primitiven Datentypen ... 126
Übernahme von komplexen Datentypen ... 127
4.1.4 Behandlung von Attributen mit Hilfe von dynamischen Metaelementen ... 129
4.1.5 Behandlung von Vererbungsbeziehungen ... 132
4.1.6 Identifizierung von Tabellen – Primärschlüssel ... 135
4.1.7 Auflösen von Beziehungen – Fremdschlüssel ... 138
4.2 Fortgeschrittene Konzepte der Operational Mappings ... 141
4.2.1 Objektverfolgung ... 141
Allgemeine Resolution ... 141
Spezielle Resolution ... 143
4.2.2 Strukturierung von Transformationen ... 145
4.2.3 Wiederverwendbarkeit von Mapping-Operationen ... 150
4.2.4 BlackBox-Funktionen ... 152
4.3 Transformation von UML-Modellen ... 155
4.3.1 UML2EJB ... 155
4.3.2 Das UML2-Metamodell ... 158
4.3.3 Das Werkzeug – QVT Operational ... 161
4.3.4 Die Transformation UML2EJB ... 164
Schritt 1: Definieren und Einrichten der Transformation ... 164
Schritt 2: Aufbereiten der *Mapping*-Operationen ... 165
Schritt 3: Mapping von Fachklassen zu SessionBeans ... 167
Schritt 4: Erzeugung der *getter*- und *setter*-Methoden ... 169
Schritt 5: Standardmethoden für die Organisation der *Bean*-Klasse ... 171
Schritt 6: Veröffentlichung der Methoden in den *Interfaces* ... 172

4.4 QVT und UML-Profile 175
4.4.1 Definition und Untersuchung eines UML-Profils 175
4.4.2 Transformation von persistenten Klassen 178

A Die Syntax der Operational Mappings 181
A.1 Reservierte Wörter 181
A.2 Ableitungsregeln 182
Metaregeln 182
Operatoren und Symbole 182
Top Level Rules 183
Model Types Compliance and Metamodel Declarations 183
Transformation 184
Library 184
Import of Modules – Transformations and Libraries 184
Syntax for Entries 185
Properties in Transformation 185
General Purpose Grammar Rules 185
Syntax for Helper Operations 186
Syntax for Constructors 186
Syntax for Mapping Operations 187
Expressions 187
Syntax for Defining Explicitly Metamodel Contents 190
Typedefinitions 191

B Metamodelle in serialisierter Darstellung 193
B.1 Deklaration der Metamodelle als QVT-Datentypen 193
B.2 Ecore-Repräsentation 195
SimpleUML 195
SimpleRDBM 197
B.3 Benutzung der Ecore-Metamodelle 200

C Operational Mappings-Beispiele 201
C.1 UmlToRdbm 201
PackageToSchema 201
ClassToTable 202
AssociationToTable 204
Das Transformationsscript UmlToRdbm 205
C.2 UML2EJB 210
transformPackages 210
transformClasses 211
Das Transformationsscript UML2EJB 213

D Standardbibliotheken 219
D.1 QVT-Standardbibliothek 219
Vordefinierte QVT-Datentypen 219
Methoden auf Transformation 220
Methoden auf Model 220
Methoden auf Status 221
Methoden auf Object 222
Methoden auf Element 222
Methoden auf Dictionary 223
Methoden auf List 224
Methoden auf Integer 225
Methoden auf String 225
D.2 Die wichtigsten OCL-Standardfunktionen 229
OCL-Standardfunktionen auf Sammlungen 229
OCL-Iterator-Funktionen 231

Glossar 233

Abkürzungsverzeichnis 251

Quellenverzeichnis 255
Literatur 255
Referenzen im Internet 258

Index 261

Abbildungsverzeichnis

Abb.1.1: Der klassische Anwendungsentwicklungsprozess 2
Abb.1.2: SWE – Analyse, Spezifikation und Implementierung 3
Abb.1.3: Übersicht über die Diagramme der UML 6
Abb.1.4: Ein modellgetriebener Anwendungsentwicklungsprozess 10
Abb.1.5: Der MDA-Entwicklungsprozess 14
Abb.1.6: Das MDA-Transformationspattern 15
Abb.1.7: Ein exemplarisches MDA-Transformationspattern 17
Abb.1.8: Ein einfaches Metamodell für die Modellierungssprache `Shapes` 18
Abb.1.9: Architektur der QVT-Sprachen 20
Abb.1.10: Model-To-Model/Model-To-Text-Abgrenzung 23
Abb.1.11: Die Architektur der QVT-Entwicklungsumgebung 26
Abb.2.1: Transformation – UML-Klassendiagramme nach ERM-Schemata 28
Abb.2.2: Das Metamodell `SimpleUML` als UML-Klassendiagramm 29
Abb.2.3: Das Wohnungsbaukreditgeschäft als `SimpleUML`-Modell 31
Abb.2.4: Das Metamodell `SimpleRDBM` 32
Abb.2.5: Das Wohnungsbaukreditgeschäft als SimpleRDBM-Modell 33
Abb.2.6: `SimpleUML` im Topcased Ecore/UML-Editor 47
Abb.2.7: Generierung der Metamodelle 48
Abb.2.8: Deployment der Metamodelle 49
Abb.2.9: Als Eclipse-Plugins veröffentlichte Metamodelle 50
Abb.2.10: Das `Darlehen`-Modell im `SimpleUML` 51
Abb.3.1: Die *Operational Mappings*-Plattform SmartQVT 56
Abb.3.2: Ein exemplarisches QVT-Projekt 58
Abb.3.3: QVT-Optionen im Kontextmenü 59
Abb.3.4: SmartQVT-Ausführungskonfiguration 60
Abb.3.5: Eclipse-Konfiguration einer SmartQVT-Applikation 61
Abb.3.6: Eclipse-Plugin-Konfiguration 62
Abb.3.7: Aufbau von *Operational Mappings*-Transformationen 67
Abb.4.1: Transformationspattern der Transformation `Uml2Rdbm` 118

Abb.4.2: Das Package darlehen – ein simples UML-Diagramm 118
Abb.4.3: Schema darlehen im SimpleRDBM – erste Lösung 125
Abb.4.4: Immobilie und Person mit aufgelösten komplexen Spalten 129
Abb.4.5: Das Package darlehen mit einer Spezialisierung von Person 132
Abb.4.6: Das Datenbankschema nach Auflösung der Spezialisierung 134
Abb.4.7: Immobilie, Konto und Person mit Primärschlüsseln 138
Abb.4.8: Schema darlehen mit Assoziationstabellen 140
Abb.4.9: Das Fachklassenmodell des privaten Wohnungsbaukreditgeschäfts 156
Abb.4.10: Die Transformation UML2EJB im MDA-Pattern 157
Abb.4.11: Das Metamodell UML2 im Kontext *Element* 159
Abb.4.12: Das Metamodell UML2 im Kontext *Classifier* 160
Abb.4.13: Der ***Operational QVT Editor*** 162
Abb.4.14: Konfiguration einer Operational QVT-Transformation 163
Abb.4.15: Die Komponenten des Ziel-Darlehensmodells 169
Abb.4.16: Detailansicht der *SessionBean*-Komponente SB_Konto 174
Abb.4.17: UML-Profil mit Stereotyp <<persistent>> 176
Abb.4.18: Anwendung des Stereotyps <<persistent>> im UML-Modell 177
Abb.C.1: Das *Operational Mappings*-Script UmlToRdbms im Überblick 202
Abb.C.2: Die *Mapping*-Operation ClassToTable im Überblick 203
Abb.C.3: Die *Mapping*-Operation AssociationToTable im Überblick 204
Abb.C.4: Die Transformation UML2EJB 210
Abb.C.5: Das *Mapping* von Paketen 211
Abb.C.6: Das *Mapping* von Fachklassen 212

1 Einführung

1.1 Eine kurze Geschichte der modellgetriebenen Software-Entwicklung

Die moderne Software-Entwicklung ist zunehmend mit der Aufgabe konfrontiert, immer kompliziertere Anforderungen aus der realen Welt mit immer vielfältigeren Anwendungssystemen und Software-Lösungen zu unterstützen. Der Entwickler steht dabei grundsätzlich vor dem Problem, dass er zum einen seine eigenen komplexen Technologien beherrschen muss, zum anderen die Sachverhalte, Strukturen und Gegebenheiten der realen Welt verstehen muss, mit der er es zu tun hat. Um diesen grundlegenden Problemen, die Mitte der sechziger Jahre zu dem Begriff Software-Krise geführt haben, zu begegnen, hat sich im Lauf der Zeit die Einsicht ergeben, den Software-Entwicklungsprozess aus einer Anwendungsprogrammierung herauszuheben und zunehmend nach ingenieursmäßigen Grundsätzen zu gestalten. Und damit wurde auf der berühmten *NATO Conference on Software Engineering* in Garmisch-Partenkirchen, 1968, der Begriff *Software Engineering* [Bau68, Bau93] aus der Taufe gehoben.

Analog zu den traditionellen Ingenieursdisziplinen hat man es zu einem der wesentlichen Prinzipien des *Software Engineerings* gemacht, sich bei der Entwicklung von Software nicht unmittelbar mit der Fertigung, der Programmierung, zu beschäftigen, sondern die Sachverhalte und Strukturen der realen Welt in mehreren aufeinander folgenden Abschnitten zu erschließen und mit abstrahierenden Darstellungstechniken wie Skizzen, Diagrammen oder Modellen zu beschreiben. Einen typischen, wenn auch etwas vereinfacht dargestellten Entwicklungsprozess, in dem die Erstellung von Software über mehrere Phasen erfolgt, zeigt das Diagramm in Abbildung 1.1, in dem die Phasen nach den Rollen der mitwirkenden Personen gegliedert sind.

Üblicherweise beginnt die Entwicklung mit einer Beschreibung der betrieblichen Sachverhalte und Gegebenheiten. Dies liegt in der Verantwortung der fach-

S. Nolte, *QVT - Operational Mappings*, Xpert.press, DOI 10.1007/978-3-540-92293-3_1,
© Springer-Verlag Berlin Heidelberg 2010

lichen Experten, wobei ein Analytiker bereits zuarbeiten und unterstützen kann. In der folgenden Phase vertieft ein Analytiker die betrieblichen Beschreibungen und neben der Klärung weiterer fachlicher Fragen erfolgt eine Formalisierung für den folgenden Entwicklungsschritt. Im Rahmen der Konzeption fertigt ein Designer den Entwurf des zu implementierenden Systems an, welcher in der Konstruktion vom Entwickler umgesetzt wird.

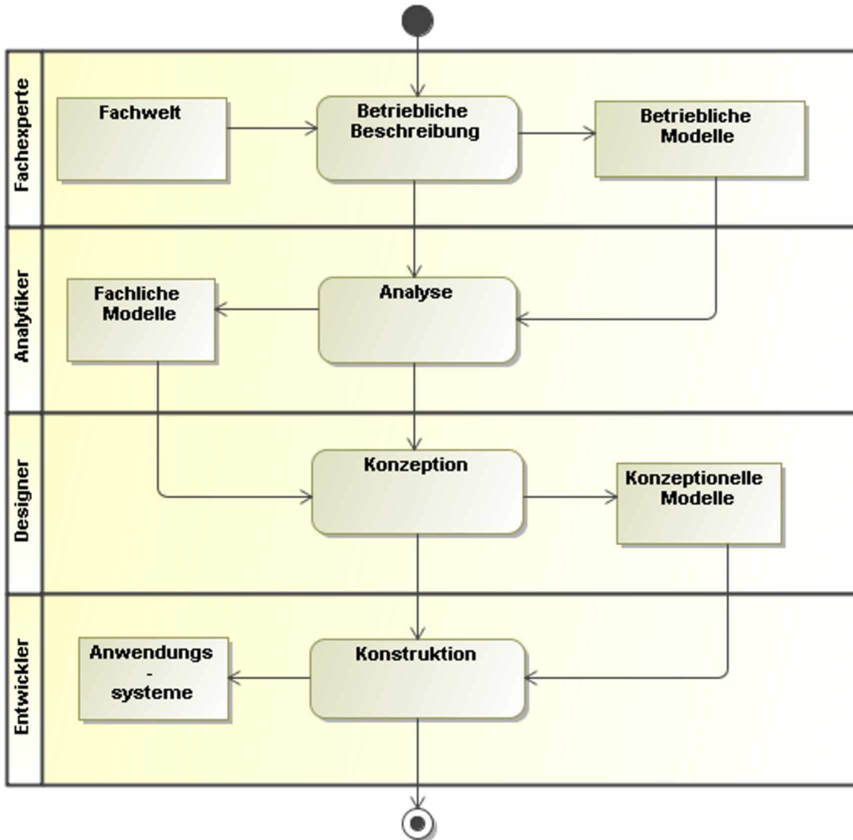

Abb. 1.1: Der klassische Anwendungsentwicklungsprozess

Abbildung 1.2 stellt noch einmal etwas konkreter die Kooperation in den frühen Phasen dar und zeigt die üblicherweise dabei eingesetzten Mittel. In den frühen Phasen der „Beschreibung und Analyse der betriebliche Prozesse" kooperieren Experten der betrachteten Fachwelt mit Analytikern, um die fachlichen Gegebenheiten in ersten betrieblichen Modellen zu erfassen und darzustellen. Der Analytiker setzt dabei noch semi-formale Modellierungstechniken ein, insbesondere die Sprachen und Darstellungsweisen, die in der Fachwelt präsent sind, Fachsprachen oder spezielle domänenspezifische Sprachen.

Im weiteren Entwicklungsprozess und mit zunehmender Abstraktion werden immer formalere Sprachen und Techniken angewendet, letztendlich bis hin zum Einsatz von Programmiersprachen und plattformspezifischen Technologien (Abbildung 1.2). In weiteren Analyse- und Designprozessen muss der Analytiker sich immer wieder mit den Mitwirkenden in den folgenden Phasen auseinandersetzen, um die fachlichen Modelle zu überarbeiten und zu fertigen DV-Lösungen fortzuentwickeln.

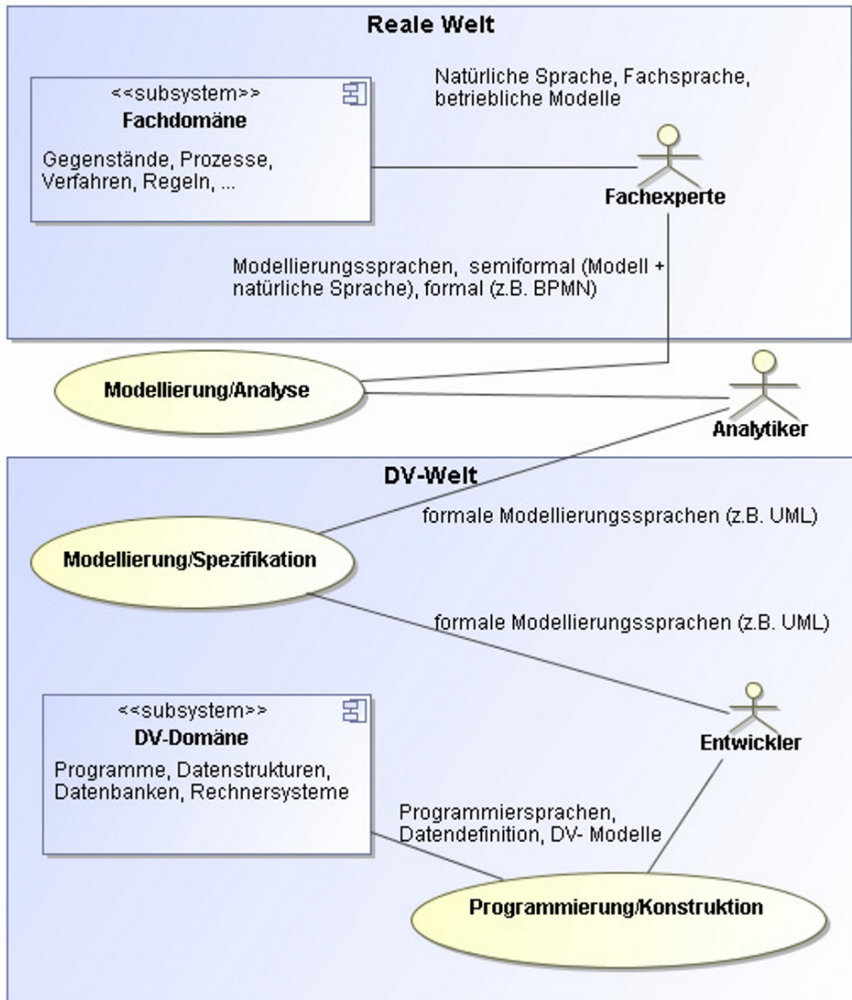

Abb. 1.2: SWE – Analyse, Spezifikation und Implementierung

1.1.1 Strukturierte Software-Entwicklung

Bereits in den frühen sechziger Jahren haben Vordenker wie Carl Adam Petri Modellierungstechniken entwickelt, mit deren Hilfe komplexe Steuerungsvorgänge beschrieben und untersucht werden konnten: die berühmten Petrinetze [Pet62, Pet63, Rei91]. Einen weiteren Ansatz, Techniken der strukturierten Programmierung [Wir93] in graphischer Form abzubilden, haben Isaac Nassi und Ben Shneiderman 1972/73 vorgeschlagen: die Struktogramme oder auch Nassi/Shneiderman-Diagramme [NSD73]. Etwa zeitgleich wurde die Abbildung von imperativen Ablaufmustern in Programmablaufplänen entwickelt. Beide Techniken, Sachverhalte der strukturierten Programmierung in Diagrammen darzustellen, flossen in die Normungen und Standards des *Deutschen Institutes für Normung* (DIN) und der *International Standardization Organization* (ISO) ein:

- Struktogramme - DIN66261 [DIN85],
- Programmablaufpläne - DIN66001 [DIN83], ISO5807.

In den folgenden Jahren wurden von Tom DeMarco und anderen Modelle und Darstellungstechniken herausgearbeitet, die ausgehend von der strukturierten Programmierung hin zu einer globaleren strukturierten Anwendungsentwicklung führten [Dem79, Mcn88, Mar88].

Auch auf der Seite der Datenmodellierung und des Datenbankdesigns fanden Entwicklungen statt, indem zum Beispiel beginnend mit Edgar F. Codd das konzeptionelle Datenbankdesign losgelöst von netzwerkartigen physikalischen Datenbanksystemen auf einer einfachen mathematischen Basis, der Relationentheorie, begründet wurde [Cod70, Cod90]. Auf dieser Grundlage beschrieb Peter P. Chen 1976 das berühmte *Entity Relationship Model*: ein erster graphischer Modellierungsansatz für eine datenzentrierte, objektorientierte Sicht der Dinge [Che76].

1.1.2 Objektorientierte Software-Entwicklung

Ab dann war es kein weiter Weg mehr, die aus der objektorientierten Programmierung stammenden Abstraktionstechniken Datenkapselung und *Information Hiding* [Par72] im Sinne einer objektorientierten Analyse und Designs weiterzuführen. Bertrand Meyer veröffentlichte 1988 die *Object-Oriented Software Construction* [Mey88], Peter Coad und Edward Yourdon 1991 *Object-Oriented Analysis* [Coa91a] und *Object-Oriented Design* [Coa91b] und James Rumbeaugh et al. *Object-Oriented Modeling and Design Techniques* [Rum91]. Grady Booch brachte 1994 *Object-Oriented Analysis And Design With Applications* [Boo91] heraus. All diese Autoren hatten eine mehr oder weniger an den Gegenständen der realen Welt – eben den Objekten – ausgerichtete Sicht auf die Dinge. Einen etwas anderen Ansatz verfolgte Ivar Jacobson, der mit der Veröffentlichung des *Object-Oriented Software-Engineering* [Jac92] eine Betrachtung der Anwendungsfälle *(Use Cases)* einer realen Welt in den Mittelpunkt einer Analyse stellte.

Diese Auflistung ist sicher nicht vollständig. So sind zum Beispiel Autoren wie Larry Constantine, James Odell, Sally Shlaer und Stephen Mellor, Harel und andere etwas in den Hintergrund gerückt. Aber es reicht soweit, um zur *Unified Modeling Language* (UML) als der zurzeit zentralen Modellierungssprache zu kommen. Mitte der 90er Jahre hat die Firma Rational die „Amigos" Booch, Rumbeaugh und etwas später Jacobson unter einem Dach zusammengebracht mit der Zielsetzung, auf der Basis der bestehenden Konzepte und Lösungen im Umfeld der Software-Technologie ein vereinheitlichtes Modell zu erarbeiten. Das Ergebnis war die UML [Boo05a, Boo05b, UML]. Die UML, die mittlerweile in der Version 2.2 vorliegt [UML2], ist eine Modellierungssprache, die eine Vielzahl von Sprachmitteln und Diagrammen zur Verfügung stellt, um strukturelle und funktionale Sachverhalte zu beschreiben. In Abbildung 1.2 kann man bereits erkennen, dass die UML insbesondere in der Welt der Anwendungsentwickler universell in mehreren der Entwicklungsaktivitäten eingesetzt werden kann.

Auch außerhalb der Informatik, zum Beispiel in den wirtschaftswissenschaftlichen Fächern, hat es Einsichten und Entwicklungen gegeben, Sachverhalte in der realen Welt mit Hilfe von Modellen und abstrakten Darstellungstechniken zu beschreiben, etwa im Bereich der Aufbau- und Ablauforganisation von betrieblichen Systemen oder bei der Analyse und Optimierung von betrieblichen Prozessen [Oes95]. Diesen Weg möchte ich nun nicht wie oben aufzeigen. Erwähnenswert ist jedoch, dass auch dies in Modellierungssprachen gemündet ist, die formal sind, zum Beispiel die aus der *Business Process Modeling Initiative* [BPMI] hervorgegangene *Business Process Modeling Notation* [BPMN], die sich mittlerweile ebenfalls im Standardisierungsverfahren der *Object Management Group* (OMG) befindet.

1.2 Die berühmten Akronyme der OMG

Nach dem Abriss der Historie der Modellierungssprachen sollen nun erst einmal ein paar Sätze zur OMG und deren „Produkten" folgen. Die OMG ist ein unabhängiges Gremium, welches 1989 unter Beteiligung eines Herstellerkonsortiums mit der Zielsetzung gegründet worden ist, objektorientierte Technologien zu standardisieren [OMG].

Die OMG hat nicht den Auftrag, Lösungen und Anwendungssysteme zu entwickeln und auf den Markt zu bringen, sondern nur Konzepte und Spezifikationen. Die Ergebnisse und Veröffentlichungen sind frei und können von jedem Hersteller und Anbieter von Systemen genutzt werden, um entsprechende Werkzeuge zu produzieren. Eines der ersten OMG-Projekte bestand darin, einen Ansatz zur Homogenisierung und Vereinheitlichung von verteilten Objekten zu entwickeln. Daraus sind die *Common Request Broker Architecture* [COR] und die *Data Distribution Services* [DDS] für Echtzeitsysteme entstanden. Ein weiteres Produkt, welches hier von zentraler Bedeutung ist und dem wir uns nun etwas ausführlicher widmen wollen, ist die UML.

1.2.1 UML – Unified Modeling Language

Etwa mit dem Stand UML 1.1 wurde die UML 1997 der *Object Management Group* (OMG) mit dem Auftrag übergeben, die Sprache als Standard zu spezifizieren und weiter zu pflegen. Die UML besteht aus einer Menge von Sprachmitteln zur Beschreibung von strukturellen und funktionalen Sachverhalten von Gegenständen einer realen Welt. Zur Darstellung werden dreizehn Diagrammtypen angeboten, sechs Diagramme für Struktur, sieben für Verhalten (Abbildung 1.3):

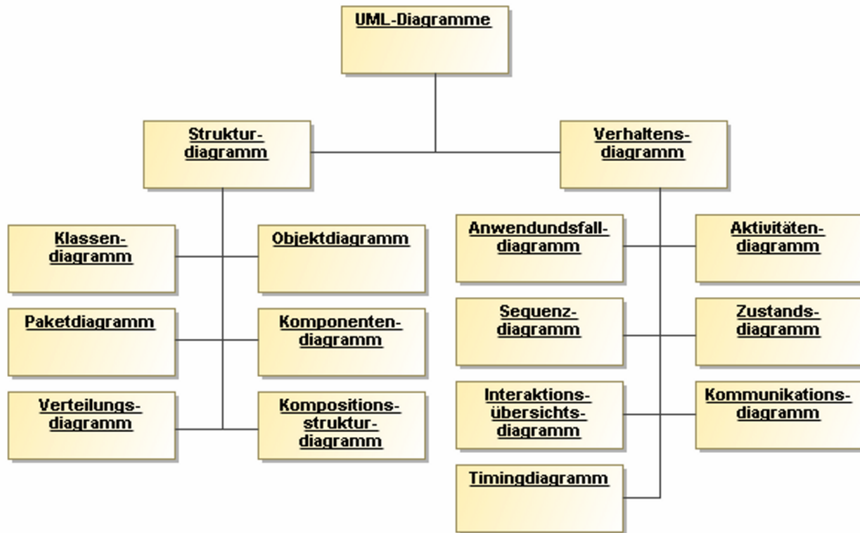

Abb. 1.3: Übersicht über die Diagramme der UML

- Strukturdiagramme
  Strukturdiagramme sind die Diagrammtypen, die zur Darstellung des Aufbaus von Systemen dienen. Es handelt sich hierbei im Wesentlichen um Klassendiagramme und Objektdiagramme. Außerdem zählen dazu weitere Diagramme, die unter dem Oberbegriff Architekturdiagramme zusammengefasst werden können, zum Beispiel Paketdiagramme, Komponentendiagramme, Verteilungsdiagramme.
  - Klassendiagramm
    Mit dem Klassendiagramm wird die Struktur eines zu entwerfenden Systems modelliert, indem die Gegenstandstypen des Systems – allgemeiner: der betrachteten realen Welt – in ihrem Aufbau und ihren Beziehungen untereinander als Klassen abgebildet werden. Gegenstände haben in der Regel beschreibende Merkmale, die als Attribute der Klassen modelliert werden. Außerdem besitzen sie funktionale Eigenschaften, die als Methoden oder Operationen der Klassen abgebildet werden.

  - Objektdiagramm
    Ein Objektdiagramm bietet die Möglichkeit, ein modelliertes System durch die Instantiierung seiner Klassen zu einem bestimmten Zeitpunkt zu betrachten. Es ist die Modellierung eines „Schnappschusses" von einem modellierten System.
  - Paketdiagramm
    Pakete erlauben eine Gruppierung von strukturellen Elementen eines Modells. Ein Paketdiagramm gibt die Möglichkeit, die modellierte Struktur eines Systems zu gliedern und bestimmte Elemente in sinnvollen Gruppen (Paketen) zusammenzufassen.
  - Kompositionsstrukturdiagramm
    Ein Kompositionsstrukturdiagramm ermöglicht explizit die Modellierung von „Teile/Ganzes-Beziehungen". Derartige „Teile/Ganzes-Beziehungen" werden als Kompositionsstrukturen bezeichnet. Kompositionsstrukturen sind häufige Beziehungen zwischen Klassen, die im Klassendiagramm mit Aggregationen oder Kompositionen dargestellt werden.
  - Komponentendiagramm
    Wie auch das Kompositionsstrukturdiagramm ist das Komponentendiagramm eine Option zur Darstellung von fachlichen Architekturen. Ein Komponentendiagramm dient der Darstellung und Betrachtung der Bestandteile eines modellierten Systems, welche auf einem DV-System – oder allgemeiner: auf einer zugrundeliegenden Plattform – betrieben werden sollen. Als Komponente in diesem Sinne wird ein UML-Element bezeichnet, das eine austauschbare Einheit eines Systems darstellt.
  - Verteilungsdiagramm
    Ein Verteilungsdiagramm zeigt die Zuordnung von Komponenten – in Form von speziellen SW-Artefakten, zum Beispiel Dateien, Archiven – auf Betriebseinheiten eines Systems. Die Einheiten werden als Knoten oder Geräte (`Devices`) bezeichnet. Ein Verteilungsdiagramm ordnet also die im Wesentlichen aus Software bestehenden Komponenten den Hardware-Einheiten einer IT-Infrastruktur zu. Dabei wird systemnahe Software – Betriebssysteme, Applikationsserver, DB-Server – im Allgemeinen den Hardware-Einheiten zugerechnet.
- Verhaltensdiagramme
  Verhaltensdiagramme sind die Diagrammtypen, die zur Darstellung des funktionalen Verhaltens von Systemen dienen. Es handelt sich hierbei zum Beispiel um Anwendungsfalldiagramme, Aktivitätendiagramme und Zustandsdiagramme. Außerdem zählen dazu die Diagramme, die unter dem Oberbegriff Interaktionsdiagramme zusammengefasst sind, also Sequenzdiagramm, Kommunikationsdiagramme und weitere.
  - Anwendungsfalldiagramm
    Anwendungsfälle sind die systemgestützten Vorgänge oder Prozesse eines betrieblichen Umfelds, einer betrieblichen Organisationseinheit. Anwendungsfälle beschreiben das System so, wie es sich aus der Sicht von bestimmten Akteuren darstellt oder darstellen soll. In einem etwas weiter ge-

fassten, aber durchaus üblichen Verständnis kann man Anwendungsfälle mit Geschäftsprozessen gleichsetzen.

- Aktivitätendiagramm
  Mit Aktivitätendiagrammen werden die betrieblichen Prozesse und Aktivitäten der betrachteten realen Welt in ihren zeitlichen und logischen Abläufen modelliert. Aktivitätendiagramme zählen zu den zentralen Diagrammtypen, die zur Modellierung von Prozessen und betrieblichen Abläufen dienen. Die zeitlichen Abläufe ergeben sich aus dem Kontrollfluss, die logischen Abläufe aus dem Datenfluss.
- Zustandsdiagramm
  Ein Zustandsdiagramm ist ein weiteres Modell, um Verhalten zu modellieren – in Form von Zuständen und Zustandsübergängen. Dabei geht es im Allgemeinen um das Verhalten von Klassen. Das Verhalten einer Klasse wird mit Zuständen modelliert, die sie im Laufe ihrer „Lebenszeit" (*Lifecycle*) annehmen kann. Zwischen den einzelnen Zuständen kann es Zustandsübergänge geben, die durch Ereignisse ausgelöst werden.
- Sequenzdiagramm
  Mit Sequenzdiagrammen wird der Informationsaustausch zwischen beliebigen Kommunikationspartnern innerhalb eines Systems – oder zwischen Systemen generell – modelliert. Die Grundelemente einer Interaktion sind Kommunikationspartner, die durch Lebenslinien repräsentiert sind, und Nachrichten, die von einem Partner – dem Sender – zu einem anderen Partner – dem Empfänger – geschickt werden.
- Kommunikationsdiagramm
  Mit Kommunikationsdiagrammen wird der Nachrichtenaustausch zwischen Komponenten von komplexeren Systemen modelliert. Die Kommunikationspartner sind die Einheiten einer komplexen Struktur; dies sind zum Beispiel Komponenten oder Rollen beziehungsweise Akteure.
- Interaktionsübersichtsdiagramm
  Interaktionsübersichtsdiagramme kommen in modellierten Systemen zum Tragen, die aus einem Großteil an Interaktionsdiagrammen bestehen. Mit Interaktionsübersichtsdiagrammen ist eine Gruppierung und übersichtlichere Anordnung dieser Diagramme möglich.
- Timingdiagramm
  Ein Timingdiagramm zeigt das zeitliche Verhalten von Klassen in einem modellierten System. Timingdiagramme sind also, wie auch andere Verhaltensdiagrammtypen, Klassen zugeordnet, die ein in diesem Fall zeitliches Verhalten haben.

Mit der Veröffentlichung der Version 1.1 ist die UML dahingehend erweitert worden, dass die *Object Constraint Language* (OCL) der Firma IBM eingeflossen ist [Kle03a, Kle03b, OCL]. Seit der Version 1.3 hat durch die Spezifikation des *XML Metadata Interchanges* (XMI) eine Vereinheitlichung des Austauschformates stattgefunden, so dass ein einfacherer Austausch zwischen den Modellierungswerkzeugen unterschiedlicher Hersteller möglich ist [XMI].

Und schließlich ist mit der Version 2.0 die UML konsequent auf der Basis eines UML-Metamodells formal spezifiziert worden. Damit ist die UML zu einer Basis für das MDA-Konzept geworden.

1.2.2 MDA – Model Driven Architecture

In der Abbildung 1.1 kann man bereits erkennen, dass ein zeitgemäßer Anwendungsentwicklungsprozess in jeder Phase immer wieder Modellierung bedeutet. Jede Aktivität der Anwendungsentwicklung umfasst ein stetiges Abstrahieren und Abbilden von Erkenntnissen in einer bestimmten formalen Weise.

Die Erkenntnisse der frühen Phasen werden gemeinsam mit Experten der Fachwelt analysiert und in fachliche Modelle überführt, zum Beispiel Geschäftsprozessmodelle oder Ereignis/Prozessketten. Diese Modelle werden aufgegriffen und in folgenden Phasen in anderen Modellen, zum Beispiel mit Hilfe der UML, dargestellt, bis zuletzt auf der Basis der vorliegenden Modelle eine Implementierung, also eine Überführung in Datenbankschemata, Programme, Konfigurationen, graphische Oberflächen etc. erfolgen kann. Das an sich ist schon in gewisser Weise ein Vorantreiben der Entwicklung durch Modellierung, also modellgetriebene Software-Entwicklung (MDSD).

Der modellgetriebene Vorgehensprozess besteht nicht allein aus Modellierungsaktivitäten. Ein wesentlicher Teil der Entwicklung der letzten Jahre liegt zum Beispiel darin, Softwarekomponenten auf der Basis von Modellen generativ zu erstellen. Diesen Gedanken, dass die Software-Entwicklung der Zukunft aus den Schritten Modellieren, Generieren, Installieren, Konfigurieren besteht, hat man schon mit der strukturierten Software-Entwicklung verfolgt und mit den *Computer Aided Software Engineering* (CASE)-Werkzeugen bis zu einer gewissen Reife gebracht. Allerdings haben sich auch Grenzen gezeigt. Noch ist die Aktivität der Programmierung nicht wegzudenken. Doch nach wie vor wird an der Idee, die Routinetätigkeiten der Codierung durch generative Techniken zu unterstützen, intensiv und durchaus mit Erfolg gearbeitet [Cza00, Sta07, AMDA, OAW].

MDA – mittlerweile ein eingetragenes Warenzeichen der OMG – und MDSD verfolgen prinzipiell die gleichen Ziele. MDA geht jedoch einen Schritt weiter und schlägt vor, dass ein modellgetriebenes Vorgehen in einem ständigen Wechsel von Modellierungs- und Transformationsschritten erfolgt. Das heißt, sofern eine Modellierungsaktivität bis zu einem gewissen Ergebnis bearbeitet worden ist, wird das Modell mit einem Transformationsschritt in eine nächste Phase überführt, wo die Modellierung mit den entsprechenden Mitteln und Techniken fortgesetzt werden kann. Auch die Umsetzung in Code ist in diesem Sinne eine Modellierungsaktivität. Abbildung 1.4 zeigt anschaulich diesen Prozess.

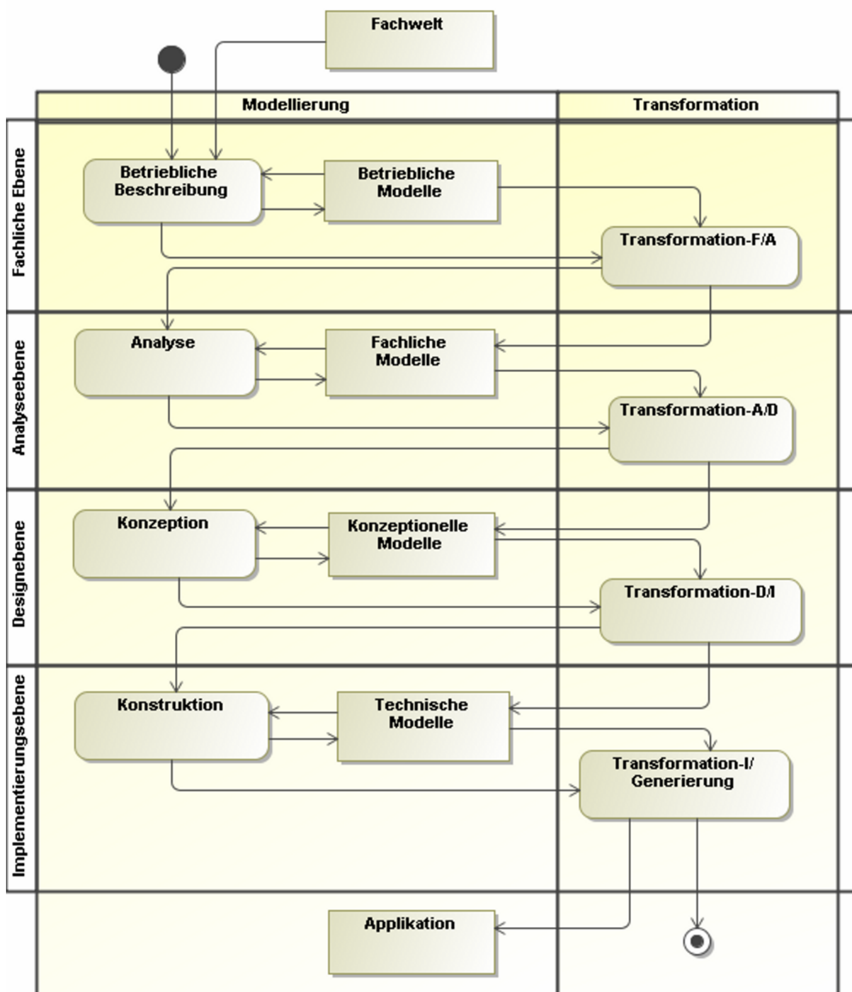

Abb. 1.4: Ein modellgetriebener Anwendungsentwicklungsprozess

Die Phasen beziehungsweise die Entwicklungsebenen, dargestellt durch die horizontalen Bereiche, sind im Grunde genommen die des einfachen phasenorientierten Vorgehensmodells, wie wir es kennen (Abbildung 1.1). In dem MDA-Verständnis gibt es jedoch zwei weitere Säulen, hier dargestellt durch die Partitionen `Modellierung` und `Transformation`. Nach der Bearbeitung eines Modells in einer Phase geht dies in den Transformationsbereich über, wo dann eine Transformation eines Modells als Ausgangspunkt für die nächste Modellierungsphase vorgenommen wird. Diese Anschauung ist natürlich noch etwas einfach, denn Transformation ist durchaus auch in derselben Ebene zugelassen, und es muss nicht immer ein *Top-down*-Transformieren „von oben nach unten“ sein.

MDA dehnt den Aspekt der modellgetriebenen Generierung auf die Entwicklung von Architekturen aus. Um dies besser zu verstehen, müssen wir uns jedoch die Definitionen der OMG aus dem MDA-Konzept etwas genauer vornehmen. Dies soll hier, auch wenn es für ein Gesamtverständnis wichtig ist, jedoch nur im Überblick getan werden. Zur Vertiefung möchte ich auf die weitere Literatur zum Thema MDA verweisen [Gru06, Pet06], insbesondere auch auf das einleitende Kapitel von [Nol09] und die Originalspezifikation der OMG [MDA], die allerdings nicht immer einfach zu lesen ist.

Grundbegriffe der Model Driven Architecture

Im Folgenden sind einige Definitionen aufgeführt, die in dieser Form auch in [Nol09] und in den dort angegebenen Quellen zu finden sind:

- System

  Ein System ist ein aus Teilen zusammengesetztes und strukturiertes Ganzes. Systeme haben eine Funktion, erfüllen einen Zweck und verfügen über eine Architektur [IEEE 1471]. Damit können nahezu alle konkreten oder abstrakten „Dinge“ subsumiert werden, die eine Struktur haben. Im Kontext der modellgestützten Entwicklung von IT-Systemen interessieren uns natürlich alle betrieblichen oder technischen Systeme, in denen IT-Lösungen eingesetzt werden sollen, wie auch die IT-Systeme selbst, die zur Entwicklung der IT-Lösungen herangezogen werden.

- Plattform

  Eine Plattform ist eine Ausführungsumgebung für ein Anwendungssystem. Sie stellt dazu spezielle Schnittstellen bereit, wie zum Beispiel ein Betriebssystem, ein Datenbanksystem oder die Laufzeitumgebung einer Programmiersprache.

  Nach dem Verständnis der OMG ist eine Plattform
  „eine kohärente Menge von Subsystemen und Technologien, die von auf dieser Plattform lauffähigen Anwendungen benutzt werden können. Die Benutzung erfolgt durch die Verwendung genau spezifizierter Schnittstellen und Gebrauchsmuster ohne Kenntnis darüber, wie die über die Schnittstellen angebotenen Dienste implementiert sind“.

  Die Plattform als Ausführungsumgebung von Anwendungssystemen kann kaskadierend sein. Das heißt, ein auf einer bestimmten Plattform laufendes Anwendungssystem kann selbst wieder Plattform für ein anderes System sein. So ist zum Beispiel das auf der Hardware eines Rechnersystems laufende Betriebssystem eine mögliche Plattform für ein Datenbanksystem und dies wiederum eine Plattform für eine datenbankgestützte Applikation.

- Modell

  Ein Modell ist die Beschreibung oder Abbildung eines Systems und seiner Umgebung unter einem gewissen Blickwinkel der Betrachtung. (Der Begriff *purpose* der OMG-Definition wird hier mehr in Richtung *aspect* oder *viewpoint* ausgedehnt, vgl. [Gru06].) Ein Modell wird oft, zumindest in frühen Phasen der Anwendungsentwicklung, in einer semi-formalen Weise repräsentiert, bestehend aus graphischen Diagrammen mit erläuternden natürlichsprachlichen Kommentierungen. Ein Modell beschreibt immer die Sachverhalte einer Realität in einer abstrakten graphischen oder textuellen Form. Modelle können auch als maßstabgerechte Nachbildungen einer Realität repräsentiert sein. Das ist in dem MDA-Kontext allerdings nicht relevant.

- Modellierung

  Die Beschreibung eines Systems in Form eines Modells wird als Modellierung bezeichnet; Modellierung ist ein zentraler Gedanke des MDA-Ansatzes. Modellierung ist die konkrete Beschreibung der realen Welt beziehungsweise des Ausschnittes der realen Welt, der Domäne, die Gegenstand der aktuellen Betrachtung ist. In diesem Kontext wird die Darstellung in Modellen ausschließlich mit formalen Modellierungssprachen vorgenommen, zum Beispiel der UML.

- Architektur

  Der nächste wesentliche Aspekt ist der der Architektur und – natürlich – des architekturgetriebenen Vorgehens.

  Nach der [IEEE1471] ist Architektur
  „die fundamentale Organisation eines aus untereinander in Beziehung stehenden Komponenten zusammengesetzten Systems und dessen Umgebung. Das System hat eine Ordnung und ist in einer evolutionären Entwicklung entworfen und realisiert."

  Das heißt, die Architektur eines Systems besteht aus dem System in seinem Aufbau an sich und aus den diversen abstrakten Repräsentationsformen des evolutionären Entwicklungsprozesses.

  Nach dem Verständnis der OMG ist die
  „Architektur eines Systems die abstrakte Spezifikation seiner Bestandteile – Parts –, der verbindenden Elemente – Konnektoren – und der Regeln für die Interaktion zwischen den Teilen eines Systems über die definierten Konnektoren".

Der Begriff MDA ist nun nicht einfach in diesen Katalog einzuordnen. MDA bedeutet intuitiv die systematische Entwicklung von stabilen, tragfähigen IT-Architekturen, also modellgetriebene Architekturentwicklung. Dies geschieht, wie wir oben in Abbildung 1.4 sehen können, durch eine iterierende Abfolge von Mo-

dellierungs- und Transformationsschritten. In den Entwicklungsebenen findet Modellierung statt. Der Übergang von einer Entwicklungsebene zur folgenden wird jeweils durch eine Transformation der Modelle unterstützt. Transformationen ermöglichen jedoch nicht nur eine Überführung von Modellen, sondern auch Prüfung von deren Gültigkeit im Sinne einer formalen Modellierung, wodurch gewissermaßen die Tragfähigkeit der Systeme konzeptionell hergestellt werden kann.

Ein willkommener Nebeneffekt eines MDA-orientierten Vorgehens ist es demnach, dass bei jedem Phasenübergang, von einer Entwicklungsebene zu einer anderen, unter formalen Gesichtspunkten gültige Ausgangsmodelle übergeben werden. In gewisser Weise kann auch die semantische, sachlogische Validität von Modellen kontrolliert und gewahrt werden, nämlich durch die Ergänzung und Prüfung von formalen Gültigkeitsbedingungen zum Beispiel mit der *Object Constraint Language*.

Die drei Grundziele der modellgetriebenen Architekturentwicklung [MDA03] sind:

1. Portabilität,
   die größtmögliche Unabhängigkeit eines Systems von möglichen Betriebsplattformen,
2. Interoperabilität,
   die Fähigkeit eines Systems, möglichst nahtlos mit anderen Systemen zusammenzuwirken,
3. Wiederverwendbarkeit,
   das Qualitätsmerkmal eines Systems, möglichst umfassend in möglichst vielen unterschiedlichen Kontexten verwendet werden zu können.

Der MDA-Ansatz versucht, diese grundlegenden Ziele zu erreichen, indem eine konsequente Trennung der Spezifikation eines Systems von dessen Implementierung auf einer speziellen Plattform vorgenommen wird. So beginnt man mit einer Modellierung, die gänzlich losgelöst von entwicklungsspezifischen Sachverhalten ist, *Computational Independent Modeling* (CIM). Dem folgt eine bereits konzeptionellere Modellierung, die immer noch bestimmte Entwicklungs- und Betriebsplattform unberücksichtigt lässt, *Platform Independent Modeling* (PIM). Im Weiteren erfolgt eine Überführung der Spezifikation auf eine oder mehrere beliebige zugrundeliegende Plattform(en), wo eine weitere Modellierung unter plattformspezifischen Gesichtspunkten stattfindet, *Platform Specific Modeling* (PSM). Zuletzt wird eine Überführung des plattformspezifischen Modells in die Implementierungsschicht vorgenommen, *Implementation Modeling* (IM).

Man kann bereits erkennen, dass MDA im Sinne der Architekturentwicklung auch ein spezielles Vorgehen beschreibt, so dass es naheliegt, das MDA-Schichtenmodell in einem Vorgehensmodell zu integrieren (Abbildung 1.5). Weitere Überlegungen dazu finden sich in [Pet06].