Xpert.press

Bernhard Rumpe

Modellierung mit UML

2. Auflage

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.

Bernhard Rumpe

Modellierung mit UML

Sprache, Konzepte und Methodik

2. Auflage

Prof. Dr. Bernhard Rumpe
RWTH Aachen
Informatik/Software Engineering
Ahornstr. 55
52062 Aachen
Deutschland
http://www.se-rwth.de

ISSN 1439-5428
ISBN 978-3-642-22412-6 e-ISBN 978-3-642-22413-3
DOI 10.1007/978-3-642-22413-3
Springer Heidelberg Dordrecht London New York

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 2004, 2011
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 GmbH, Heidelberg

Gedruckt auf säurefreiem Papier

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

Vorwort

Vorwort zur 2ten Auflage

Vor zehn Jahren war absehbar, dass agile Methoden sich zumindest für einen Teilbereich der Softwareentwicklung durchsetzen würden, auch wenn sie damals von vielen Entwicklern noch eher belächelt wurden. Mittlerweile sind agile Methoden ein etablierter Bestandteil des Portfolios der Softwaretechnik und wurden an vielen Stellen ergänzt und für mehrere Domänen angepasst.

Parallel hat die Unified Modelling Language ihren Siegeszug angetreten und hat heute praktisch alle anderen echten Modellierungssprachen in sich aufgesogen oder verdrängt, wobei wir Matlab/Simulink nicht als echte Modellierungssprache sondern als grafische Programmiersprache zählen wollen. Die UML ist einerseits groß und leidet weiterhin an den vielen Optionen und Interpretationsmöglichkeiten, die aufgrund ihrer vielen Einsatzgebiete auch nicht so ohne weiteres reduziert werden können. Stattdessen ist es wohl besser ein expliziteres Variabilitätsmodell für syntaktische, methodische und semantische Unterschiede zu erstellen und durch geeignete Auswahl auf einzelne Projekte zu konfigurieren [Grö10].

Noch erfolgreicher hat sich die Programmiersprache Java sowohl als primäre Web- und Business-System-Sprache, als auch als Lehrsprache für Informatikstudenten durchgesetzt.

In diesem sowie auch dem darauf aufbauendem Buch „Agile Modellierung mit UML" werden die UML als auch Java konsolidiert, in Maßen ergänzt und gleichzeitig weiterentwickelt. UML liegt in Version 2.3 und Java in Version 6 vor. Die in diesem Buch vorgestellte UML/P stellt zwar eine eigenständige Fassung, ein sogenanntes Profil, dar, wurde aber durch die Änderungen von UML 1.4 nach UML 2.3 dennoch an einigen Stellen angepasst. Da Java als Ziel von Generierungs- und Testvorgängen zum Einsatz kommt, ist es natürlich von Interesse, auf die neuen Möglichkeiten von Java wie zum Beispiel den Generics oder dem assert-Statement einzugehen.

Die Kluft zwischen den Welten der modellbasierten Softwareentwicklung mit der UML und den agilen Methoden hat sich trotz oder vielleicht gerade wegen des Erfolgs beider Ansätze nicht wirklich geschlossen. Während agile Methoden durchaus gerne Code generieren statt von Hand schreiben wollen, sehen viele Entwickler im Moment noch die Hürde zur Generierung als relativ groß an. Dies liegt häufig an der Unhandlichkeit bzw. Schwergewichtigkeit der Generierungsprozesses und des relativ großen initialen Aufwands zur Einführung von Generierungswerkzeugen in den Entwicklungsprozess. Diese Lücke gilt es noch zu schließen.

An der Erstellung der ersten und der Überarbeitung zur zweiten Fassung dieses Buchs haben eine Reihe von Personen direkt oder indirekt mitgewirkt. Mein besonderer Dank gilt Manfred Broy für die Unterstützung, die dieses Buch erst ermöglicht hat und den Mitarbeitern und Studierenden, insbesondere Christian Berger, Marita Breuer, Angelika Fleck, Hans Grönniger, Sylvia Gunder, Tim Gülke, Arne Haber, Christoph Herrmann, Roland Hildebrandt, Holger Krahn, Thomas Kurpik, Markus Look, Shahar Maoz, Philip Martzok, Anonio Navarro Pérez, Class Pinkernell, Dirk Reiss, Holger Rendel, Jan Oliver Ringert, Martin Schindler, Mark Stein, Christopher Vogt, Galina Volkova, Steven Völkel und Ingo Weisenmöller, die dieses Buch als Grundlage ihrer Arbeit einsetzen oder geholfen haben es für die zweite Auflage zu ergänzen und zu verbessern. Gerne danke ich dem ehemaligen bayerischen Minister für Wissenschaft, Forschung und Kunst Hans Zehetmair für die Verleihung des Habilitationsstipendiums und meinem geschätzten Kollegen und Vorgänger Prof. Dr.-Ing. Manfred Nagl für eine wohlwollende Unterstützung beim Aufbau des Lehrstuhl in Aachen.

Herzlicher Dank gilt meinen Freunden, Kolleginnen und Kollegen, wissenschaftlichen Mitarbeitern, sowie den Studierenden für konstruktive Diskussionen, Mitarbeit an dem Anwendungsbeispiel und Reviews von Zwischenständen dieses Buchs in erster Auflage aus München: Samer Alhunaty, Hubert Baumeister, Markus Boger, Peter Braun, Maria Victoria Cengarle, David Cruz da Bettencourt, Ljiljana Döhring, Jutta Eckstein, Andreas Günzler, Franz Huber, Jan Jürjens, Ingolf Krüger, Konstantin Kukushkin, Britta Liebscher, Barbara Paech, Markus Pister, Gerhard Popp, Jan Philipps, Alexander Pretschner, Mattias Rahlf, Andreas Rausch, Stefan Rumpe, Robert Sandner, Bernhard Schätz, Markus Wenzel, Guido Wimmel und Alexander Wisspeintner.

Bernhard Rumpe

Aachen im Juni 2011

Vorwort zur 1ten Auflage

Der Entwurf großer Software Systeme ist eine der großen technischen Herausforderungen unserer Zeit. Umfang und Komplexität von Software haben mittlerweile Größenordnungen erreicht, die alle bekannten Ansätze und Methoden ihrer Entwicklung an ihre Grenzen bringen.

Vor diesem Hintergrund haben auch die Softwareentwickler das in den Ingenieurwissenschaften altbewährte Rezept der Modellbildung stärker für sich entdeckt. In den letzten Jahren ist unter dem Stichwort modellbasierter Softwareentwicklung eine große Zahl unterschiedlicher Ansätze entstanden, die eine umfangreiche Modellbildung zur Unterstützung der Entwicklung von Softwaresystemen zum Ziel haben. Modellbildung erlaubt es, wichtige Eigenschaften und Aspekte eines zu analysierenden oder zu erstellenden Softwaresystems gezielt modellhaft darzustellen. Ein Anliegen dabei ist eine angemessene Abstraktion, die zu einer Komplexitätsreduktion und einer besseren Beherrschbarkeit von Softwaresystemen führt. Trotz aller Fortschritte auf diesem Gebiet und seiner durchaus gegebenen Einsatzreife für die Praxis stehen noch viele ungelöste Fragen für die Forschung offen.

Ein kritischer Faktor der Modellbildung ist natürlich der zusätzliche Aufwand in der Entwicklung. Hier stellt sich die Frage, wie weit man den Aufwand bei der Modellbildung überhaupt treiben sollte und wie man die oft schwergewichtigten, modellbasierten Vorgehensweisen mit genügend Flexibilität versehen kann, so dass sie die Profile der durchgeführten Projekte besser berücksichtigen.

Neben der Modellorientierung ist ein weiterer Trend in den letzten Jahren im Software Engineering der Einsatz sogenannter agiler Methoden, insbesondere unter dem Stichwort „extreme Programming". Hierunter werden leichtgewichtige Vorgehensmodelle für die Software Entwicklung verstanden, die eine Reduzierung der Softwarebürokratie sicherstellen und eine viel höhere Flexibilität in der Softwareentwicklung erlauben. Für Projekte bestimmten Profils können agile Methoden ein bedeutend effektiveres Vorgehen ermöglichen. Voraussetzung dafür sind jedoch entsprechend hinreichend kompetente Entwickler sowie ein deutlich begrenzter Projektumfang. So können agile Methoden erfolgreich nur in kleinen Projekten eingesetzt werden mit nur einer Handvoll Entwickler über einen überschaubaren Zeitraum, so dass für eine schnelle Kommunikation Rückkopplung im Projekt tatsächlich funktionieren kann.

Zunächst scheint es, als dass modellbasierte Ansätze mit ihrer starken Systematik und ihrer bewußten - von der eigentlichen Codierung losgelösten Modellierungstechnik - den agilen Methoden, die in der Regel codezentriert sind, nicht vereinbar sind. Die vorliegende Arbeit zeigt eindrucksvoll, dass es doch möglich ist, modellbasierte Ansätze mit agilen Methoden zu kombinieren und dies unter Einsatz wohlbekannter Modellierungssprachen wie der UML. Dazu muss allerdings sorgfältig überlegt werden, welche UML-Konstrukte als Modellierungs-, Test- und Implementierungsbeschreibungs-

mittel eingesetzt werden können und wie methodisch vorgegangen werden soll.

Eine Antwort auf diese Frage leistet die Arbeit von Herrn Rumpe. Herr Rumpe setzt sich gleichermassen zum Ziel, relevante, praktische Ansätze, wie agiles Vorgehen und die weit verbreitete Sprache UML einzusetzen, ohne dabei aber auf saubere wissenschaftliche Fundierung und gut dokumentiertes Vorgehen zu verzichten. Deutlich wird insbesondere dargestellt, welche Programmkonstrukte der UML geeignet sind, um beispielsweise Testfälle rigoros zu entwickeln oder über perfekte Regeln eine evolutionäre Weiterentwicklung anzustoßen.

Die vorliegende Arbeit demonstriert, wie die beiden recht unterschiedlichen Paradigmen der agilen Methoden und der Modellorientierung doch miteinander korrespondieren und sich ergänzen. Es entsteht ein Ansatz, der gleichermaßen dem Anspruch einer praxisnahen, gut einsetzbaren Vorgehensweise, wie dem Anspruch einer sauberen wissenschaftlichen Fundierung gerecht wird.

Der beiliegende Text ist sehr gut lesbar, ohne dabei den Anspruch aufzugeben, eine sorgfältige inhaltliche und fachliche Darstellung zu leisten. Die Vorgehensweise, die Herr Rumpe hier vorschlägt, hat er selbst in einer Reihe von kleineren Projekten erfolgreich erprobt.

Die Arbeit stellt damit einen wertvollen Beitrag dar, der für Praktiker eine gute Handlungsanleitung gibt und Ihnen zusätzliche Informationen zeigt, wie sie aktuelle Trends der Softwaretechnik - wie agiles Vorgehen und modellbasierte Entwicklung - erfolgreich und mit guten Ergänzungen miteinander kombinieren können. Für Studierende wird eine umfassende Einführung in das Themengebiet und eine solide Fundierung geleistet.

Das vorliegende und das darauf aufbauende Buch „Agile Modellierung mit UML" sind somit gleichermaßen für Praktiker geeignet, die an einem entsprechenden Ansatz für ihre Entwicklungsprojekte interessiert sind, wie auch für auf praktische Fragestellungen ausgerichtete Vorlesungen, die aber nicht auf einen grundlegenden wissenschaftlichen Anspruch verzichten möchten.

Manfred Broy

Garching im Februar 2004

Inhaltsverzeichnis

1 Einführung 1
1.1 Ziele der beiden Bücher 2
1.2 Überblick 3
1.3 Notationelle Konventionen 4
1.4 Einordnung der UML/P 5
1.4.1 Bedeutung und Anwendungsbereiche der UML 5
1.4.2 UML-Sprachprofile 6
1.4.3 Die Notationen in der UML/P 7
1.4.4 Modellbegriff und Modellbasierung 8
1.5 Ausblick: Agile Modellierung mit UML 12

2 Klassendiagramme 15
2.1 Bedeutung der Klassendiagramme 16
2.2 Klassen und Vererbung 19
2.2.1 Attribute 19
2.2.2 Methoden 22
2.2.3 Vererbung 23
2.2.4 Interfaces 24
2.3 Assoziationen 25
2.3.1 Rollen 26
2.3.2 Navigation 26
2.3.3 Kardinalität 27
2.3.4 Komposition 27
2.3.5 Abgeleitete Assoziationen 28
2.3.6 Assoziationsmerkmale 29
2.3.7 Qualifizierte Assoziation 30
2.4 Sicht und Repräsentation 31
2.5 Stereotypen und Merkmale 34
2.5.1 Stereotypen 36
2.5.2 Merkmale 37
2.5.3 Einführung neuer Elemente 38

3 Object Constraint Language ... 41
3.1 Übersicht über OCL/P ... 43
3.1.1 Der Kontext einer Bedingung ... 43
3.1.2 Das **let**-Konstrukt ... 46
3.1.3 Fallunterscheidungen ... 48
3.1.4 Grunddatentypen ... 49
3.2 Die OCL-Logik ... 50
3.2.1 Die boolesche Konjunktion ... 50
3.2.2 Zweiwertige Semantik und Lifting ... 52
3.2.3 Kontrollstrukturen und Vergleiche ... 54
3.3 Container-Datenstrukturen ... 55
3.3.1 Darstellung von Mengen und Listen ... 56
3.3.2 Mengen- und Listenkomprehension ... 58
3.3.3 Mengenoperationen ... 61
3.3.4 Listenoperationen ... 64
3.3.5 Container-Operationen ... 66
3.3.6 Flachdrücken von Containern ... 68
3.3.7 Typisierung von Containern ... 69
3.3.8 Mengen- und listenwertige Navigation ... 71
3.3.9 Qualifizierte Assoziation ... 76
3.3.10 Quantoren ... 77
3.3.11 Spezialoperatoren ... 82
3.4 Funktionen in OCL ... 85
3.4.1 Queries ... 85
3.4.2 ≪OCL≫-Methoden ... 89
3.4.3 Methodenspezifikation ... 91
3.4.4 Bibliothek von Queries ... 104
3.5 Ausdrucksmächtigkeit der OCL ... 105
3.5.1 Transitive Hülle ... 105
3.5.2 Die Natur einer Invariante ... 109
3.6 Zusammenfassung ... 111

4 Objektdiagramme ... 113
4.1 Einführung in Objektdiagramme ... 116
4.1.1 Objekte ... 116
4.1.2 Attribute ... 118
4.1.3 Links ... 118
4.1.4 Qualifizierte Links ... 120
4.1.5 Komposition ... 121
4.1.6 Merkmale und Stereotypen ... 123
4.2 Bedeutung eines Objektdiagramms ... 125
4.2.1 Unvollständigkeit und Exemplarizität ... 125
4.2.2 Prototypische Objekte ... 126
4.2.3 Instanz versus Modellinstanz ... 127
4.3 Logik der Objektdiagramme ... 129

4.3.1 Namen für ein Diagramm ... 130
4.3.2 Bindung von Objektnamen ... 130
4.3.3 Integration von Objektdiagramm und OCL ... 132
4.3.4 Anonyme Objekte ... 133
4.3.5 OCL-Bedingungen im Objektdiagramm ... 134
4.3.6 Abstrakte Objektdiagramme ... 135
4.4 Methodische Verwendung von Objektdiagrammen ... 137
4.4.1 Komposition von Objektdiagrammen ... 137
4.4.2 Negation ... 138
4.4.3 Alternative Objektstrukturen ... 139
4.4.4 Objektdiagramme in einer Methodenspezifikation ... 139
4.4.5 Objekterzeugung ... 141
4.4.6 Gültigkeit von Objektdiagrammen ... 142
4.4.7 Initialisierung von Objektstrukturen ... 143
4.5 Zusammenfassung ... 145

5 Statecharts ... 147
5.1 Eigenschaften von Statecharts ... 149
5.2 Automatentheorie und Interpretation ... 150
5.2.1 Erkennende und Mealy-Automaten ... 151
5.2.2 Interpretation ... 153
5.2.3 Nichtdeterminismus als Unterspezifikation ... 155
5.2.4 ε-Transitionen ... 156
5.2.5 Unvollständigkeit ... 156
5.2.6 Lebenszyklus ... 157
5.2.7 Beschreibungsmächtigkeit ... 158
5.2.8 Transformationen auf Automaten ... 159
5.3 Zustände ... 160
5.3.1 Zustandsinvarianten ... 162
5.3.2 Hierarchische Zustände ... 167
5.3.3 Start- und Endzustand ... 170
5.4 Transitionen ... 171
5.4.1 Bedingungen innerhalb der Zustandshierarchie ... 171
5.4.2 Start- und Endzustand in der Zustandshierarchie ... 172
5.4.3 Stimuli für Transitionen ... 174
5.4.4 Schaltbereitschaft ... 176
5.4.5 Unvollständiges Statechart ... 179
5.5 Aktionen ... 183
5.5.1 Prozedurale und beschreibende Aktionen ... 183
5.5.2 Aktionen in Zuständen ... 185
5.5.3 Zustandsinterne Transitionen ... 190
5.5.4 do-Aktivität ... 190
5.6 Statecharts im Kontext der UML ... 191
5.6.1 Vererbung von Statecharts ... 191
5.6.2 Transformation von Statecharts ... 192

5.6.3 Abbildung in die OCL ... 205
5.7 Zusammenfassung ... 207

6 Sequenzdiagramme ... 209
6.1 Konzepte der Sequenzdiagramme ... 211
6.2 OCL in Sequenzdiagrammen ... 215
6.3 Semantik eines Sequenzdiagramms ... 217
6.4 Sonderfälle und Ergänzungen für Sequenzdiagramme ... 222
6.5 Sequenzdiagramme in der UML ... 225
6.6 Zusammenfassung ... 228

A Sprachdarstellung durch Syntaxklassendiagramme ... 229

B Java ... 237

C Die Syntax der UML/P ... 245
C.1 UML/P-Syntax Übersicht ... 245
C.2 Klassendiagramme ... 246
C.2.1 Kernteile eines Klassendiagramms ... 247
C.2.2 Textteile eines Klassendiagramms ... 248
C.2.3 Merkmale und Stereotypen ... 250
C.2.4 Vergleich mit dem UML-Standard ... 251
C.3 OCL ... 254
C.3.1 Syntax der OCL ... 254
C.3.2 Unterschiede zu dem OCL-Standard ... 257
C.4 Objektdiagramme ... 260
C.4.1 Kontextfreie Syntax ... 261
C.5 Statecharts ... 263
C.5.1 Abstrakte Syntax ... 264
C.5.2 Vergleich mit dem UML-Standard ... 267
C.6 Sequenzdiagramme ... 269
C.6.1 Abstrakte Syntax ... 269
C.6.2 Vergleich mit dem UML-Standard ... 270

D Anwendungsbeispiel: Internet-basiertes Auktionssystem ... 273
D.1 Auktionen als E-Commerce Applikation ... 274
D.2 Die Auktionsplattform ... 275

Literatur ... 279

Index ... 289

1 Einführung

Das Streben nach Wissen ist eine
natürliche Veranlagung aller Menschen.

Aristoteles

Die Softwaretechnik hat sich in den letzten Jahren zu einer wirkungsvollen Ingenieursdisziplin entwickelt. Aufgrund der kontinuierlich anwachsenden Komplexität ihrer Aufgabenstellungen und Diversität der Anwendungsdomänen konnte ein *Portfolio von Softwareentwicklungstechniken* gebildet werden, das für jede Anwendungsdomäne, Kritikalität und Komplexität des zu entwickelnden Systems eine weitgehend maßgeschneiderte Auswahl an geeigneten Vorgehensweisen und Konzepten bietet. Techniken für Projekt-, Konfigurations-, Varianten- und Qualitätsmanagement, Software Produktlinien, Vorgehensmodelle, Spezifikationstechniken, Analyse- und Entwurfsmuster und „Best Practices" für spezielle Aufgabenstellungen sind nur einige der Elemente dieses Portfolios.

In diesem Portfolio stehen zum einen konkurrierende Ansätze mit jeweils problemspezifischen Vorteilen zur Verfügung. Zum anderen ermöglicht und bedingt die Weiterentwicklung der eingesetzten Sprachen, Frameworks und Werkzeuge eine kontinuierliche Ergänzung und Erweiterung des Portfolios auch in methodischer Hinsicht. Programmiersprachen wie Java, ausgereifte Klassenbibliotheken und die permanent verbesserten Softwareentwicklungswerkzeuge erlauben heute Vorgehensweisen, die noch vor wenigen Jahren undenkbar waren. So ist beispielsweise die werkzeuggestützte Weiterentwicklung oder Modifikation einer bereits im Einsatz befindlichen Softwarearchitektur, mittlerweile wesentlich einfacher geworden.

Weiterführendes Material:

http://www.se-rwth.de/mbse

B. Rumpe, *Modellierung mit UML*, 2. Aufl., Xpert.press, DOI 10.1007/978-3-642-22413-3_1,
© Springer-Verlag Berlin Heidelberg 2011

Die sich schnell wandelnde Technologie, die von den Anwendern beispielsweise im E-Service-Bereich erwartete Flexibilität und Erweiterbarkeit der Systeme sowie die hohe Kritikalität von Geschäftsanwendungen machen es notwendig, Vorgehensweisen und die dabei benutzten Entwicklungstechniken kontinuierlich zu optimieren und anzupassen. Nur so lässt sich unter Nutzung der vorhandenen Softwareentwicklungstechniken und der gegebenen zeitlichen und personellen Ressourcen in flexibler Weise ein qualitativ hochwertiges und für die Wünsche des Kunden geeignetes System entwickeln und kontinuierlich ergänzen.

Die Verbreitung des Internets erlaubt auch die zunehmende Integration von Geschäftsanwendungen über Unternehmensgrenzen hinweg, sowie die Einbindung der Anwender über Rückkopplungsmechanismen sozialer Netzwerke, so dass besonders im Bereich internetbasierter Software komplexe Netzwerke von E-Service- und E-Business-Anwendungen entstehen. Dafür sind adäquate Softwareentwicklungstechniken notwendig. In diesem Bereich wird vor allem Objekttechnologie genutzt und zur Modellierung die sich derzeit zum Standard entwickelnde Unified Modeling Language (UML) eingesetzt.

1.1 Ziele der beiden Bücher

Mission Statement: Es ist ein Kernziel, für das genannte Portfolio einige grundlegende Techniken zur modellbasierten Entwicklung zur Verfügung zu stellen. Dabei wird in diesem Buch eine Variante der UML vorgestellt, die speziell zur effizienten Entwicklung qualitativ hochwertiger Software und Software-basierter Systeme geeignet ist.

UML-Standard: Der UML-Standard muss sehr viele Anforderungen aus unterschiedlichen Gegebenheiten heraus erfüllen und ist daher notwendigerweise überladen. Viele Elemente des Standards sind für unsere Zwecke nicht oder nicht in der gegebenen Form sinnvoll, während andere Sprachkonzepte fehlen. Deshalb wird in diesem Buch ein angepasstes und mit UML/P bezeichnetes Sprachprofil der UML vorgestellt. UML/P wird dadurch für die vorgeschlagenen Entwicklungstechniken im Entwurf, in der Implementierung und in der Wartung optimiert und so in agilen Entwicklungsmethoden besser einsetzbar.

Dieses Buch konzentriert sich vor allem auf die Einführung des Sprachprofils. In einem zweiten Buch „Agile Modellierung mit UML“ werden darüber hinausgehend vor allem modellbasierte Vorgehenstechniken Generierung, Testfalldefinition und Evolution beschrieben.

Die UML/P ist als Ergebnis mehrerer Grundlagen- und Anwendungsprojekte entstanden. So wurde zum Beispiel die in Anhang D beschriebene Anwendung soweit möglich unter Verwendung der hier beschriebenen Prinzipien entwickelt. Das beschriebene Auktionssystem ist auch deshalb ideal zur Demonstration der in den beiden Büchern entwickelten Techniken, weil

Veränderungen des Geschäftsmodells oder der Unternehmensumgebung in dieser Anwendungsdomäne besonders häufig sind. Flexible und dennoch qualitativ hochwertige Softwareentwicklung ist für diesen Bereich essentiell.

Objektorientierung und Java: Für neue Geschäftsanwendungen wird heute primär Objekttechnologie eingesetzt. Die Existenz vielseitiger Klassenbibliotheken und Frameworks, die vorhandenen Werkzeuge und nicht zuletzt der weitgehend gelungene Sprachentwurf begründen den Erfolg der Programmiersprache Java. Das in diesem Buch beschriebene UML-Sprachprofil UML/P und die darauf aufbauenden Entwicklungstechniken werden daher auf Java aufgebaut.

Brücke zwischen UML und agilen Methoden: Gleichzeitig bilden die beiden Bücher zwischen den noch nicht so gut integrierten Ansätzen der agilen Methoden und der Modellierungssprache UML eine elegante Brücke. Agile Methoden wie zum Beispiel Extreme Programming besitzen eine Reihe von interessanten Techniken und Prinzipien, die das Portfolio der Softwaretechnik für bestimmte Projekttypen bereichern. Merkmale dieser Techniken sind der weitgehende Verzicht auf Dokumentation, die Konzentration auf Flexibilität, Optimierung der Time-To-Market und Minimierung der notwendigen personellen Ressourcen bei gleichzeitiger Sicherung der geforderten Qualität. Damit sind agile Methoden für die Ziele dieses Buchs als Grundlage gut geeignet.

Agile Vorgehensweise auf Basis der UML/P: Die UML wird als Notation für eine Reihe von Aktivitäten, wie Geschäftsfallmodellierung, Soll- und Ist-Analyse sowie Grob- und Fein-Entwurf in verschiedenen Granularitätsstufen eingesetzt. Die Artefakte der UML stellen damit einen wesentlichen Grundstein für die Planung und Kontrolle von Meilenstein-getriebenen Softwareentwicklungsprojekten dar. Deshalb wird die UML vor allem in plan-getriebenen Projekten mit relativ hoher Dokumentationsleistung und der daraus resultierenden Schwerfälligkeit eingesetzt. Nun ist die UML aber kompakter, semantisch reichhaltiger und besser geeignet, komplexe Sachverhalte darzustellen, als eine Programmiersprache. Sie bietet dadurch für die Modellierung von Testfällen sowie für die transformationelle Evolution von Softwaresystemen wesentliche Vorteile. Auf Basis einer Diskussion agiler Methoden und der darin enthaltenen Konzepte wird in Buch 2 daher eine agile Methode skizziert, die das UML/P-Sprachprofil als Grundlage für viele Aktivitäten nutzt, ohne die Schwerfälligkeit typischer UML-basierter Methoden zu importieren.

1.2 Überblick

Kapitel 2 beinhaltet eine Definition der Bestandteile von Klassendiagrammen, eine Diskussion des Einsatzes von Sichten und Repräsentationen sowie einen Vorschlag zur Definition von Stereotypen und Merkmalen.

Kapitel 3 stellt die syntaktisch an Java angepasste Form der Object Constraint Language (OCL) in allen semantischen Facetten vor. Für Spezifikationszwecke wird eine zweiwertige Logik eingeführt und die Beschreibungsmächtigkeit der OCL diskutiert. Konstrukte zur Mengenkomprehension, zur Einführung lokaler Funktionen sowie spezieller OCL-Operatoren für das Flachdrücken von Datenstrukturen und die Bildung der transitiven Hülle einer Assoziation werden vorgestellt.

Kapitel 4 beinhaltet die Einführung der Objektdiagramme als eigenständige Notation in der UML/P. Eine beidseitige Integration von OCL und Objektdiagrammen erlaubt die prädikative Verwendung der Diagramme und gleichzeitig die Beschreibung komplexer Zusammenhänge innerhalb eines Diagramms durch die OCL. Die Logik-Operatoren der OCL werden genutzt, um unerwünschte Situationen, Alternativen und die Komposition von Objektdiagrammen zu beschreiben.

Kapitel 5 gibt eine detaillierte Einführung in die Statecharts der UML/P. Dabei werden zunächst einfache Automaten als semantisches Modell studiert und die dabei gewonnenen Erkenntnisse in Bezug auf Nichtdeterminismus, Unterspezifikation, Vervollständigung und Beschreibungsmächtigkeit auf die UML/P-Statecharts übertragen. Für die Beschreibung der Vorbedingungen wird OCL und für die Aktionen werden wahlweise OCL und Java eingesetzt. Eine Sammlung von Transformationsregeln erlaubt es, vereinfachte Statecharts zu erhalten, die in OCL übersetzt werden können oder sich zur Codegenerierung eignen.

Kapitel 6 beschreibt eine einfache Form von Sequenzdiagrammen, die es erlaubt, lineare Abläufe zu formulieren.

Anhänge A-C beschreiben die abstrakte Syntax der UML/P.

Anhang D beschreibt das gewählte Anwendungsbeispiel aus dem E-Commerce-Bereich. Das Internet-basierte Auktionssystem wird in Teilen des Buchs vielfach für Beispiele herangezogen.

1.3 Notationelle Konventionen

In diesem Buch werden mehrere Diagrammarten und textuelle Notationen eingeführt. Damit sofort erkennbar ist, welches Diagramm oder welche textuelle Notation jeweils dargestellt ist, wird abweichend von der UML 2.3 rechts oben eine Marke in einer der in Abbildung 1.1 dargestellten Formen angegeben. Diese Form ist auch zur Markierung textueller Teile geeignet und flexibler als die UML 2.3-Markierung. Eine Marke wird einerseits als Orientierungshilfe und andererseits als Teil der UML/P eingesetzt, da ihr der Name des Diagramms und Diagramm-Eigenschaften in Form von Stereotypen beigefügt werden können. Vereinzelt kommen Spezialformen von Marken zum Einsatz, die weitgehend selbsterklärend sind.

Die textuellen Notationen wie Java-Code, OCL-Beschreibungen und textuelle Teile in Diagrammen basieren ausschließlich auf dem ASCII-Zeichen-

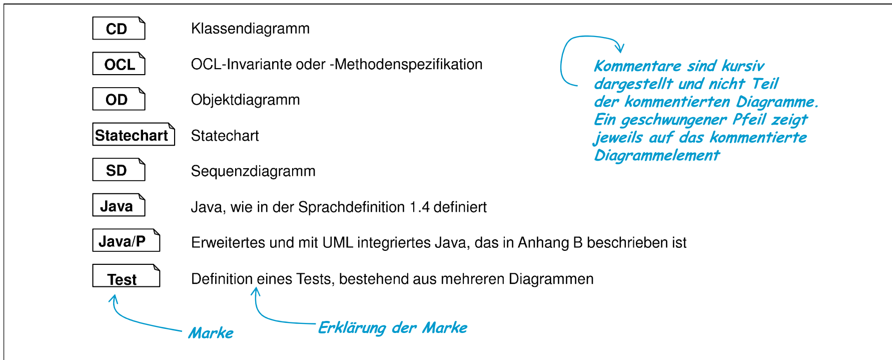

Abbildung 1.1. Marken für Diagramme und Textteile

satz. Zur besseren Lesbarkeit werden einzelne Schlüsselwörter hervorgehoben oder unterstrichen. Für Stereotypen, Transformationen, Testmuster und Refactoring-Regeln wird in 2.19 eine Schablone vorgestellt, die eine systematische Beschreibung des eingeführten Elements erlaubt. Im Text werden folgende Sonderzeichen genutzt:

- Die Repräsentationsindikatoren „…“ und „©“ sind formaler Teil der UML/P und beschreiben, ob die in einem Diagramm dargestellte Repräsentation vollständig ist.
- Stereotypen werden in der Form «Stereotypname» angegeben. Merkmale haben die Form {`Merkmalsname=Wert`} oder {`Merkmalsname`}.
- Nichtterminale werden als ⟨Name⟩ dargestellt. Bei einer Verwendung in anderen Abschnitten wird die Abbildung Definitionsstelle des Nichtterminals mit angegeben. Beispiel: ⟨OCLConstraint$_{\mathrm{C.7}}$⟩.

1.4 Einordnung der UML/P

1.4.1 Bedeutung und Anwendungsbereiche der UML

Graphische Notationen haben gegenüber textuellen Darstellungsformen speziell bei der Kommunikation zwischen Entwicklern eine Reihe von Vorteilen. Sie erlauben dem Betrachter einen schnellen Überblick zu erhalten und erleichtern das Erfassen von miteinander in Beziehung stehenden Systemteilen. Aufgrund ihrer zweidimensionalen Natur besitzen graphische Beschreibungstechniken jedoch auch Nachteile. Genannt seien hier der sehr viel größere Platzverbrauch, also die geringere *Informationsdichte*, die insbesondere bei großen Modellen leicht zum Verlust des Überblicks führt und die allgemein als schwieriger angesehene präzise Definition der Syntax und Semantik einer graphischen Sprache.

Mit der Durchdringung des objektorientierten Programmierparadigmas in nahezu alle Bereiche der Software- und Systementwicklung und der parallel immer komplexer werdenden Systeme sind eine Reihe objektorientierter Modellierungsansätze definiert worden.

Die Unified Modeling Language (UML) [OMG10a] ist der erfolgreiche Versuch, die unterschiedlichen Notationen zu vereinheitlichen und damit eine Standard-Modellierungssprache für die Softwareentwicklung zu entwerfen. Die UML hat mittlerweile einen hohen Verbreitungsgrad. Wesentlich war dabei die Trennung zwischen der Vorgehensweise und der zugrunde liegenden Notation. Die UML ist dazu gedacht, als Beschreibungstechnik für möglichst alle Anwendungsbereiche der Softwareentwicklung zur Verfügung zu stehen. Entsprechend ist auch das in diesem Buch definierte UML/P-Sprachprofil zum Teil methodenneutral, obwohl es sich in besoderem Maße für generative Projekte mit Java als Zielsprache eignet. Dies zeigen auch neuere Bücher, die sich auch mit der Beziehung zwischen der UML und einer Programmiersprache wie etwa Java beschäftigen [Lan09, Lan05].

Mit der UML ist eine Integration eines Teils der bisherigen Vielfalt an Modellierungssprachen erreicht worden. Syntaktische Unterschiede wurden harmonisiert und Konzepte aus verschiedenen Bereichen in die Gesamtsprache integriert. Obwohl dadurch eine sehr große, teilweise überladene Sprache entstanden ist, kann davon ausgegangen werden, dass die UML zumindest ein Jahrzehnt eine wesentliche Rolle als Sprachstandard beanspruchen wird.

1.4.2 UML-Sprachprofile

Die UML wird mittlerweile nicht mehr als in allen syntaktischen und semantischen Einzelheiten vollständig definierte Sprache, sondern als Sprachrahmen oder als Familie von Sprachen verstanden [CKM+99, Grö10, GRR10], die es aufgrund von Erweiterungsmechanismen und semantischen Variationsmöglichkeiten erlaubt, *Sprachprofile* auszubilden, die dem jeweiligen Einsatzzweck angepasst werden können. Damit erhält die UML Charakteristika einer Umgangssprache wie zum Beispiel Deutsch, die es ebenfalls erlaubt, das Vokabular in Form von Fachsprachen und Dialekten anzupassen.

Bereits in [OMG99] wurden die wesentlichen Anforderungen für ein Profil-Konzept für die UML festgelegt und in [CKM+99] diskutiert, wie sich dies auf die Ausprägung unternehmens- oder projektspezifischer Sprachprofile auswirkt.

[Grö10, GRR10] zeigt wie die Organisation syntaktischer und semantischer Variabilitäten eines Teils der UML in Form von Features und Sprachkonfigurationen dargestellt und für die Konfiguration einer Sprache passend zum Projekt eingesetzt werden kann.

Beispiele für Sprachprofile sind die Spezialisierung der UML auf Echtzeitsysteme [OMG09], auf Enterprise Distributed Object Computing

(EDOC) [OMG04], Multimedia-Anwendungen [SE99b, SE99a] und Frameworks [FPR01]. Die Erweiterbarkeit der UML wird durch mehrere Mechanismen auf verschiedenen Ebenen erreicht. Neues Vokabular wird durch Benennung von Klassen, Methoden, Attributen oder Zuständen direkt im Modell eingeführt. Profile bieten zusätzlich „light-weight"-Erweiterungen der UML-Syntax, wie die in Abschnitt 2.17 diskutierten Stereotypen und Merkmale, und „heavy-weight"-Erweiterungen mit neuen Modellierungskonstrukten.[1]

Laut [OMG99] ist das Konzept zur Profildefinition für die UML unter anderem dafür vorgesehen:

- Präzise Definition von Merkmalen, Stereotypen und Bedingungen (Constraints) ist möglich.
- Die Beschreibung von Semantik in natürlicher Sprache ist zugelassen.
- Ein spezifischeres Profil kann ein allgemeineres Profil für die gewünschte Einsatzform anpassen.
- Die Kombination von Profilen erlaubt die gleichzeitige Verwendung mehrerer Profile.
- Mechanismen zum Management der Kompatibilität von Profilen werden angeboten.

Der Wunsch nach leichter Austauschbarkeit und Kombinierbarkeit von Sprachprofilen ist jedoch nicht leicht zu erfüllen. Im Gegenteil können werkzeugbasierte Sprachprofile normalerweise nur dann kombiniert werden, wenn diese explizit aufeinander abgestimmt sind.

1.4.3 Die Notationen in der UML/P

Die UML/P besteht, wie in Abbildung 1.2 illustriert, aus sechs Teilnotationen. Dabei fehlen einige Teilnotationen des UML-Standards. UML/P ist ein Sprachprofil, das besonders die Aktivitäten *Entwurf*, *Implementierung* und *Weiterentwicklung* unterstützt, weil UML/P auch als vollständige Programmiersprache verwendet werden kann. Deshalb wurde das Sprachprofil mit dem Suffix „/P" für „Programmier-geeignet" gewählt.

Die Eignung zur Programmiersprache ist nicht zuletzt auf die Integration von Java-Codestücken in die UML/P und die Anpassung textueller Teile der UML/P auf die Java-Syntax zurückzuführen.

Die im vorhergehenden Abschnitt 1.4.2 diskutierte Notwendigkeit zur Einführung von weiteren, spezialisierenden Sprachprofilen wird in UML/P durch die Konkretisierung der Definition von Stereotypen und Merkmalen ermöglicht. Beide Formen zur Anpassung von Sprachelementen werden zur Definition der UML/P selbst genutzt, stehen aber auch für weitere Anpassungen zur Verfügung, so dass die UML/P als Ausgangsbasis für die Defini-

[1] Umgangssprachen erlauben ebenfalls die Prägung neuer Vokabeln durch Begriffsdefinitionen. Neue Modellierungskonstrukte würden der Erweiterung der Grammatik entsprechen und sind in anderen Sprachen aber nicht üblich.

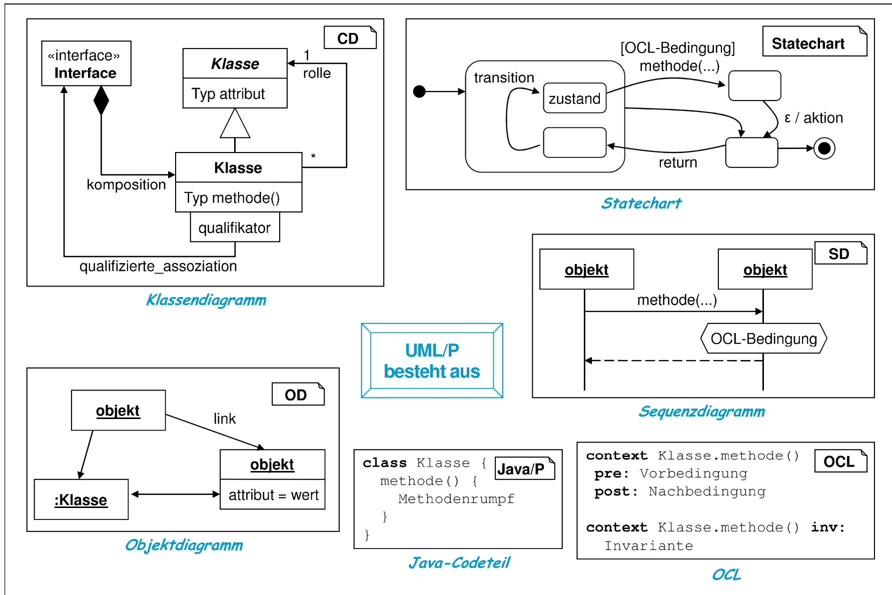

Abbildung 1.2. Die Teilsprachen der UML/P

tion weiterer, anwendungs-, domänen- oder technikspezifischer Sprachprofile geeignet ist.

1.4.4 Modellbegriff und Modellbasierung

Der Modellbegriff

Der Begriff „Modell“ wird in der Softwaretechnik für eine Reihe verschiedener Konzepte eingesetzt. So gibt es unter anderem Produktmodelle, Vorgehensmodelle oder Testmodelle. Eine gute Kategorisierung dieser Begriffe ist in [SPHP02, Sch00] zu finden. Abbildung 1.3 gibt eine Übersicht über allgemeine Definitionen des Begriffs „Modell“. Generell anerkannt ist die Abstraktion, die ein Modell gegenüber dem modellierten Gegenstand besitzt, indem zum Beispiel Details weggelassen werden. Sinnvoll, aber nicht in allen Definitionen Voraussetzung ist auch der zielorientierte Einsatz eines Modells.

Uneinigkeit herrscht im Allgemeinen über die Granularität eines Modells. So spricht [Bal00] einerseits von einem vollständigen *Produktmodell* und assoziiert damit eine Sammlung von Diagrammen und andererseits vom Modell als *Artefakt*, das ein Modell eher einem einzelnen Diagramm gleichsetzt. Auch in diesem Buch wird der Begriff „Modell“ in einem weiteren Sinn verwendet und auch ein Klassendiagramm oder ein Statechart als Modell eines Ausschnitts des zu realisierenden Systems bezeichnet.

Der Modellbegriff in der Literatur:

- Ein *Modell* ist seinem Wesen nach eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems (nach [Sta73]).
- Ein *Modell* ist eine Abstraktion eines Systems mit der Zielsetzung das Nachdenken über ein System zu vereinfachen, indem irrelevante Details ausgelassen werden (nach [BD00].
- Ein *Modell* ist eine „vereinfachte, auf ein bestimmtes Ziel hin ausgerichtete Darstellung der Funktion eines Gegenstands oder des Ablaufs eines Sachverhalts, die eine Untersuchung oder eine Erforschung erleichtert oder erst möglich macht“ [Bal00].
- In der Softwaretechnik ist ein *Modell* eine idealisierte, vereinfachte, in gewisser Hinsicht ähnliche Darstellung eines Gegenstands, Systems oder sonstigen Weltausschnitts mit dem Ziel, bestimmte Eigenschaften des Vorbilds daran besser studieren zu können (nach [HBvB$^+$94]).

Abbildung 1.3. Begriffsdefinition: Modell

Ein Modell hat grundsätzlich Bezug zu einem *Vorbild* oder *Original*. In der Softwaretechnik werden aber Modelle häufig vor dem Original gebildet. Außerdem kann aufgrund der Immaterialität von Software aus einem Modell durch *automatisches* Hinzufügen von Details das vollständige System ohne manuelles Zutun entstehen.

Modellbasierte Softwareentwicklung

In Abschnitt 2.4 wird der Begriff *Sicht* als dem Entwickler zugängliche Repräsentation zum Beispiel eines *Produktmodells* identifiziert. Die dabei stattfindende zweistufige Modellabstraktion vom System über das vollständige Modell zur Entwicklersicht ist in der Größe des Systems begründet. Ein vollständiges Produktmodell hat normalerweise eine Komplexität, die es nicht mehr ohne weiteres erlaubt, Zusammenhänge zu erkennen. Deshalb werden mit den Sichten Ausschnitte des Produktmodells gebildet, die bestimmte Aspekte betonen, andere aber auslassen. Eine Sicht ist ebenfalls ein Modell und als solches zielgerichtet. Eine Sicht wird gebildet, um eine „Story“ zu kommunizieren. Ein Produktmodell kann als die Summe seiner Sichten verstanden werden. Demgegenüber hat zum Beispiel [SPHP02] einen engeren Modellbegriff, indem es Sichten nicht als eigenständige Modelle betrachtet. Entsprechend werden alle dort diskutierten Test- und Refactoring-Techniken direkt auf dem alles umfassenden Produktmodell formuliert.

Während diese Abstraktionsstufen der Modellbildung aus Entwicklersicht allgemein anerkannt sind, herrschen bei den Werkzeugherstellern heute zwei Ansätze zur technischen Realisierung vor:

- Die *Modellbasierung* erfordert, dass ein vollständiges und konsistentes Modell des Systems im Werkzeug verwaltet wird, und erlaubt nur Ent-

wicklungsschritte, die an diesem Modell und allen seinen darin enthaltenen Sichten in konsistenter Weise vorgenommen werden.
- Die *Dokumentorientierung* erlaubt es, jede einzelne Sicht als eigenständiges Dokument zu bearbeiten. Inkonsistenzen innerhalb sowie zwischen Dokumenten werden zunächst gestattet und erst beim Aufruf von entsprechenden Analysewerkzeugen erkannt.

Beide Ansätze haben spezifische Vor- und Nachteile. Die Vorteile des modellbasierten Ansatzes sind:

- Automatische Sicherung der Konsistenz eines Modells ist nur in der modellbasierten Form zu erhalten. Im dokumentorientierten Ansatz kostet die Analyse Zeit, die zum Beispiel die Codegenerierung verlangsamt.
- Werkzeuge sind einfacher zu implementieren, da sie keine Techniken wie Verschmelzung mehrerer Klassendiagramme oder Statecharts für dieselbe Klasse anbieten müssen.

Dem stehen folgende Vorteile des dokumentorientierten Ansatzes und Nachteile der Modellbasierung gegenüber:

- Aus den Erfahrungen mit den syntaxgesteuerten Editoren für Programmiersprachen hat sich gezeigt, dass Unterstützung hilfreich ist, während syntaxgesteuerte Editoren den Gedankenfluss und die Effizienz des Entwicklers stören. Im dokumentorientierten Ansatz lassen sich vorübergehende Inkonsistenzen und syntaktisch fehlerhafte Dokumente besser tolerieren.
- In großen Projekten mit mehreren Entwicklern sind bei modellbasierten Werkzeugen Maßnahmen zu treffen, die die permanente Konsistenz des gleichzeitig bearbeiteten Modells herstellen. Dazu gehören ein gemeinsames Repository mit Synchronisations- oder Locking-Mechanismen. Während erstere Ineffizienzen mit sich bringen, untersagen letztere einen gemeinsamen Modellbesitz und verhindern Agilität.
- Eine permanente Synchronisation der Modelle über ein Repository verbietet die lokale Erprobung von Alternativen. Deshalb muss ein Transaktionskonzept, eine Versionskontrolle oder ein ähnlicher Mechanismus vom Repository angeboten werden, wodurch faktisch ebenfalls das Problem der Integration von Modellversionen entsteht.
- Demgegenüber sind im dokumentorientierten Ansatz Integrationstechniken parallel bearbeiteter Modellteile zu verwenden, wie sie etwa bei Versionsverwaltungen eingesetzt werden. Diese Integration ist notwendig, wenn der Entwickler seine lokal bearbeitete Fassung in die Versionsverwaltung zurückspielt und damit für andere Projektteilnehmer publiziert. Lokale Experimente bleiben damit folgenlos, wenn sie nicht publiziert werden.
- Für selbst entwickelte Spezialwerkzeuge ist es im Allgemeinen einfacher, einzelne, dateibasierte Dokumente zu bearbeiten, als komplette, im Repository abgespeicherte und einer Versions- oder Transaktionskontrolle unterliegende Modelle.

- Ein inkrementeller, modularer Ansatz zur Verarbeitung von Modellen, insbesondere bei der Generierung kann die Effizienz in der Entwicklung deutlich erhöhen, weil so nur Modelle eingelesen und dafür Code neu generiert werden muss, wenn sie sich verändert haben. Dies erfordert aber eine Modularität für UML-Modelle, in dem Sinn, dass die zwischen Modellen auszutauschende Information im Sinne von Schnittstellen zu klären und analog zu Programmiersprachen auch eigenständig abzulegen sind.

In der Praxis dürfte sich jedoch ein synergetischer Kompromiss beider Ansätze als der ideale Weg herauskristallisieren. Im Bereich der Bearbeitung von Programmiersprachen mit integrierten Entwicklungsumgebungen (IDE's) zeichnet sich dies bereits ab. Eine IDE beinhaltet einen Editor mit syntaxgesteuerten Hervorhebungen, Navigation und Ersetzungsmöglichkeiten bis hin zu automatischen Analysen und Codegenerierung im Hintergrund. Die Ablage der Informationen erfolgt allerdings artefaktbasiert in einzelnen Dateien, die gegebenenfalls durch zusätzliche und automatisch erstellte Tabellen unterstützt werden. Damit bleiben die einzelnen Dateien auch für andere Werkzeuge zugänglich, aber beim Entwickler entsteht der Eindruck einer modellbasierten Entwicklungsumgebung. Vorteilhaft ist auch, dass die Entwickler selbst wählen können, welche Dateien und damit welchen Teil des „Modells" sie im Werkzeug laden und bearbeiten möchten.

Model Driven Architecture (MDA)

Der „Model Driven Architecture"-Ansatz (MDA) [OMG03, PM06, GPR06] ist eine Weiterführung der Standardisierungsideen der Object Management Group (OMG), der unter anderem auf der UML basiert. Eine der Kernideen dieses Ansatzes ist im ersten Schritt der Entwicklung die Definition *plattformunabhängiger* Modelle der Geschäftsanwendung mit der UML. Davon getrennt erfolgt im zweiten Schritt die Abbildung dieses plattformunabhängigen Modells auf eine Realisierung mit konkreter Hardware, vorgegebenen Betriebssystemen, Middleware- und Framework-Komponenten.

Dadurch wird die Entwicklung des plattformunabhängigen UML-Modells von plattformspezifischen Konzepten entkoppelt. Die Implementierung besteht dann aus einer Abbildung des plattformunabhängigen auf ein plattformspezifisches UML-Modell, das in einem entsprechenden UML-Sprachprofil formuliert wird. Dafür sollen zum Beispiel Corba-spezifische UML-Profile zur Verfügung stehen. Dann erfolgt die möglichst weit automatisierte Abbildung dieses Modells in eine Implementierung und entsprechende Schnittstellendefinitionen. Neben den technologiespezifischen Abbildungen werden in der MDA auch Standardisierungsbemühungen für Anwendungsdomänen einbezogen. Dazu gehören zum Beispiel die XML-basierten Kommunikationsstandards für E-Commerce, Telekommunikation oder dem Datenmodell für die Finanzindustrie.

MDA basiert einerseits auf der Beobachtung, dass Geschäftsanwendungen durchschnittlich sehr viel länger leben als technologische Plattformen und daher des öfteren eine Migration von Anwendungen notwendig ist. Andererseits basiert MDA auf der Hoffnung, damit sowohl die Wiederverwendung beziehungsweise Evolution applikationsspezifischer Modelle für ähnliche Anwendungen als auch die Interoperabilität zwischen Systemen zu vereinfachen.

In seiner Gänze ist MDA ein Ansatz, der sowohl die Werkzeuge zur Softwareentwicklung als auch die Vorgehensweise zu deren Definition revolutionieren will und sich insbesondere mit der unternehmensweiten und unternehmensübergreifenden Vernetzung von Systemen auseinandersetzt [DSo01, GPR06]. Interessanterweise ist zwar an eine signifikante Reduktion des Aufwands zur Softwareentwicklung durch Generierung beabsichtigt, jedoch werden dazu passende Methodiken nur wenig diskutiert.

Auch der im zweiten Band diskutierte Ansatz kann als eine Konkretisierung eines Teils der MDA verstanden werden. Im Gegensatz zur MDA soll aber kein a priori alles umfassender Ansatz unter Einbeziehung beispielsweise von Metamodellierung, allen verfügbaren Middleware-Techniken oder der Interoperabilität zwischen Applikationen vorgestellt werden. Stattdessen wird hier im Sinne von XP eher die einfache, aber effektivere Lösung vorgeschlagen, in der nur die Schnittstellen bedient, nur die Middleware-Komponenten eingesetzt und nur die Betriebssysteme beachtet werden, für die das System *jetzt* zu entwickeln ist. Es ist anzunehmen, dass die Verfügbarkeit von standardisierten Abbildungen von plattformunabhängigen Modellen auf die jeweilige Technologie für weite Bereiche unwahrscheinlich sein wird. Im Allgemeinen werden diese Abbildungen auf Basis vorgefertigter Muster selbst zu entwickeln und daher Codegeneratoren entsprechend zu parametrisieren sein. Entsprechend sollte Einfachheit vor die Bedienung unnötiger Standards gestellt werden.

1.5 Ausblick: Agile Modellierung mit UML

Um die Effizienz in einem Projekt zu steigern, ist es notwendig, den Entwicklern effektive Notationen, Techniken und Methoden zur Verfügung zu stellen. Weil das primäre Ziel jeder Softwareentwicklung das lauffähige und korrekt implementierte Produktionssystem ist, sollte der Einsatz der UML nicht nur zur Dokumentation von Entwürfen dienen. Stattdessen ist die automatisierte Umsetzung in Code durch *Codegeneratoren*, die Definition von *Testfällen* mit der UML/P zur Qualitätssicherung und die Evolution von UML-Modellen mit *Refactoring*-Techniken essentiell.

Die Kombination von Codegenerierung, Testfallmodellierung und Refactoring bietet wesentliche Synergie-Effekte. Deshalb werden im zweiten bei Springer erschienenen Buch „Agile Modellierung mit UML“ diese Techniken auf Basis der UML/P beschrieben. Beide Bände sind inhaltlich aufeinander abgestimmt.

Agile Modellierung: Es werden einige wesentliche Grundelemente agiler Softwareentwicklungsmethoden herausgearbeitet und auf Basis der dabei gewonnenen Erkenntnisse eine agile modellbasierte Entwicklungsmethode skizziert. Kern agiler Entwicklungsmethoden ist die Nutzung von Modellen als Darstellungs- und Diskussionsmittel, insbesondere aber auch zur Programmierung und Testfalldefinition durch Codegenerierung und zur Planung von Evolutionsschritten durch modellbasiertes Refactoring.

Codegenerierung: Zur effizienten Erstellung eines Systems ist eine gut parametrisierte Codegenerierung aus abstrakten Modellen essentiell. Die diskutierte Form der Codegenerierung erlaubt die kompakte und von der Technik weitgehend unabhängige Entwicklung von anwendungsspezifischen Modellen. Erst bei der Generierung werden technologieabhängige Aspekte wie zum Beispiel Datenbankanbindung, Kommunikation oder GUI-Darstellung hinzugefügt. Dadurch wird die UML/P als Programmiersprache einsetzbar und es entsteht kein konzeptueller Bruch zwischen Modellierungs- und Programmiersprache. Allerdings ist es wichtig, ausführbare und abstrakte Modelle im Softwareentwicklungsprozess explizit zu unterscheiden und jeweils adäquat einzusetzen.

Modellierung automatisierbarer Tests: Die systematische und effiziente Durchführung von Tests ist ein wesentlicher Bestandteil zur Sicherung der Qualität eines Systems. Ziel ist dabei, dass Tests nach ihrer Erstellung automatisiert ablaufen können. Codegenerierung wird daher nicht nur zur Entwicklung des Produktionssystems, sondern insbesondere auch für Testfälle eingesetzt, um so die Konsistenz zwischen Spezifikation und Implementierung zu prüfen. Der Einsatz der UML/P zur Testfallmodellierung ist daher ein wesentlicher Bestandteil einer agilen Methodik. Dabei werden insbesondere Objektdiagramme, die OCL und Sequenzdiagramme zur Modellierung von Testfällen eingesetzt.

Evolution mit Refactoring: Die diskutierte Flexibilität, auf Änderungen der Anforderungen oder der Technologie schnell zu reagieren, erfordert eine Technik zur systematischen Anpassung des bereits vorhandenen Modells beziehungsweise der Implementierung. Die Evolution eines Systems in Bezug auf neue Anforderungen oder einem neuen Einsatzgebiet sowie der Behebung von Strukturdefiziten der Softwarearchitektur erfolgt idealerweise durch Refactoring-Techniken. Ein weiterer Schwerpunkt ist daher die Fundierung und Einbettung der Refactoring-Techniken in die allgemeinere Vorgehensweise zur Modelltransformation und die Diskussion, welche Arten von Refactoring-Regeln für die UML/P entwickelt oder von anderen Ansätzen übernommen werden können. Besonders betrachtet werden dabei Klassendiagramme, Statecharts und OCL.

Sowohl bei der Testfallmodellierung als auch bei den Refactoring-Techniken werden aus den fundierenden Theorien stammende Erkenntnisse dargestellt und auf die UML/P transferiert. In diesem Buch werden diese Konzepte anhand zahlreicher praktischer Beispiele erklärt und in Form von Testmustern und Refactoring-Techniken für UML-Diagramme aufbereitet.

2

Klassendiagramme

Ein Sachverhalt ist denkbar, heißt:
Wir können uns ein Bild von ihm machen.

Ludwig Wittgenstein

Klassendiagramme bilden das architekturelle Rückgrat vieler Systemmodellierungen. Deshalb werden in diesem Kapitel die in der UML/P definierten Klassendiagramme mit den Kernelementen *Klasse*, *Attribut*, *Methode*, *Assoziation* und *Komposition* eingeführt. Im Abschnitt über *Sichten* und *Repräsentationen* werden Einsatzvarianten von Klassendiagrammen diskutiert. Es wird außerdem gezeigt, wie mit *Stereotypen* und *Merkmalen* Modellierungskonzepte für projektspezifische Problemstellungen angepasst werden.

2.1 Bedeutung der Klassendiagramme 16
2.2 Klassen und Vererbung 19
2.3 Assoziationen 25
2.4 Sicht und Repräsentation 31
2.5 Stereotypen und Merkmale 34

B. Rumpe, *Modellierung mit UML*, 2. Aufl., Xpert.press, DOI 10.1007/978-3-642-22413-3_2,
© Springer-Verlag Berlin Heidelberg 2011

Klassendiagramme bilden nach wie vor die mit Abstand wichtigste und am meisten genutzte Modellierungstechnik der UML. Historisch entstanden sind Klassendiagramme aus Anleihen von der Entity/Relationship-Modellierung [Che76] und der graphischen Darstellung von Modulen, die ihrerseits von Datenfluss-Diagrammen [DeM79] beeinflusst wurden. Klassendiagramme beschreiben die Struktur eines Softwaresystems und bilden daher die erste behandelte Kernnotation für die objektorientierte Modellierung.

Im Anhang C.2 wird ergänzend die hier präsentierte Form der Klassendiagramme mit dem UML-Standard verglichen und die Syntax der Klassendiagramme präzisiert.

2.1 Bedeutung der Klassendiagramme

Objektorientierte Systeme beinhalten eine hohe Dynamik. Dadurch wird die Modellierung der Strukturen eines Systems zu einer komplexen Aufgabe in der objektorientierten Softwareentwicklung. Klassendiagramme beschreiben diese Struktur beziehungsweise Architektur eines Systems, auf der nahezu alle anderen Beschreibungstechniken basieren. Klassendiagramme und die darin modellierten Klassen haben jedoch eine Vielfalt von Aufgaben.

Modellierung von Struktur

In einer objektorientierten Implementierung wird der Code in Form von Klassen organisiert. Ein Klassendiagramm stellt daher eine Übersicht über die Code-Struktur und seine inneren Zusammenhänge dar. Weil Programmierern das Konzept *Klasse* aus der Programmierung bekannt ist, sind die in der Modellierung genutzten Klassendiagramme auch leicht verständlich und kommunizierbar. Klassendiagramme werden zur Darstellung der strukturellen Zusammenhänge eines Systems eingesetzt und bilden so das Skelett für fast alle weiteren Notationen und Diagrammarten, da diese sich jeweils auf die in Klassendiagrammen definierten Klassen und Methoden abstützen. Auch deshalb bilden Klassendiagramme ein essentielles – wenn auch nicht einziges – Beschreibungsmittel zur Modellierung von Softwarearchitekturen und Frameworks.

Klassen in Analyse, Design und Implementierung

In der Analyse werden Klassendiagramme genutzt, um Konzepte der realen Welt zu strukturieren. Demgegenüber werden Klassendiagramme bei der Erstellung von Entwurfsdokumenten und in der Implementierung vor allem zur Darstellung einer strukturellen Sicht des Softwaresystems genutzt. Die in der Implementierungssicht dargestellten Klassen sind tatsächlich im implementierten System wieder zu finden. Klassen der Analyse werden dafür

oft signifikant modifiziert, durch technische Aspekte ergänzt oder ganz weggelassen, weil sie z.B. nur zum Systemkontext gehören.

Eines der Defizite der UML entsteht aus der nicht optimalen Möglichkeit, den Diagrammen explizit einen Verwendungszweck zuzuordnen. Wird der Standpunkt eingenommen, dass ein Klassendiagramm eine Implementierung widerspiegelt, so kann die Semantik eines Klassendiagramms relativ einfach und verständlich erklärt werden. Diesen Standpunkt nehmen eine Reihe von Einführungsbüchern in die Modellierung mit Klassen beziehungsweise der UML ein [Mey97, Fow00]. Außerdem wird dieser Standpunkt oft auch durch Werkzeuge impliziert. Fusion [CAB+94] stellt demgegenüber eine explizite Abgrenzung zwischen zum System gehörigen und externen Klassen zur Verfügung und demonstriert so, dass die Modellierung von nicht-softwaretechnischen Konzepten mit Klassendiagrammen möglich und sinnvoll ist.

Das Sprachprofil UML/P ist implementierungsorientiert. Deshalb ist die nachfolgende Bedeutungserklärung von Klassendiagrammen auf Basis des dadurch modellierten Java-Codes für diesen Einsatz ideal.

Aufgabenvielfalt einer Klasse

In der objektorientierten Programmierung und stärker noch der Modellierung haben Klassen eine Vielzahl von Aufgaben. Primär dienen sie zur *Gruppierung* und *Kapselung* von Attributen und dazugehörigen Methoden zu einer konzeptuellen Einheit. Durch Vergabe eines *Klassennamens* können *Instanzen* der Klasse an beliebigen Stellen im Code erzeugt, gespeichert und weitergereicht werden. Klassendefinitionen dienen daher gleichzeitig als *Typsystem* und als *Implementierungsbeschreibung*. Sie können (im Allgemeinen) beliebig oft in Form von *Objekten* instanziiert werden.

In der Modellierung wird eine Klasse auch als *Extension*, also als die Menge aller zu einem bestimmten Zeitpunkt existierenden Objekte, verstanden. Durch die explizite Verfügbarkeit der Extension in der Modellierung kann zum Beispiel die Einhaltung einer Invariante für jedes existierende Objekt einer Klasse beschrieben werden.

Weil die Anzahl der Objekte in einem System potentiell unbeschränkt ist, ist die Katalogisierung der Objekte in endlich viele Klassen notwendig. Dadurch wird eine endliche Aufschreibung eines objektorientierten Systems erst ermöglicht. Klassen stellen damit eine *Charakterisierung der möglichen Strukturen* eines Systems dar. Diese Charakterisierung beschreibt gleichzeitig auch notwendige Strukturformen, ohne jedoch eine konkrete Objektstruktur festzulegen. Deshalb gibt es normalerweise unbeschränkt viele unterschiedliche Objektstrukturen, die einem Klassendiagramm genügen. In der Tat entspricht jedes korrekt laufende System einer sich weiterentwickelnden Sequenz von Objektstrukturen, bei der zu jedem Zeitpunkt die aktuelle Objektstruktur dem Klassendiagramm genügt.

Im Gegensatz zu den Objekten haben Klassen jedoch während der Laufzeit eines Systems in vielen Programmiersprachen keine direkt manipulierbare Repräsentation. Ausnahmen hierzu bilden etwa Smalltalk, das Klassen ebenfalls als Objekte repräsentiert und dadurch uneingeschränkte reflektive Programmierung erlaubt.[1] Java ist demgegenüber restriktiver, denn es erlaubt nur lesenden Zugriff auf den Klassencode. Generell sollte reflektive Programmierung nur sehr sparsam eingesetzt werden, weil eine Wartung des Systems aufgrund der reduzierten Verständlichkeit sehr viel komplexer wird. Deshalb wird im weiteren Verlauf auf reflektive Programmierung nicht weiter eingegangen.

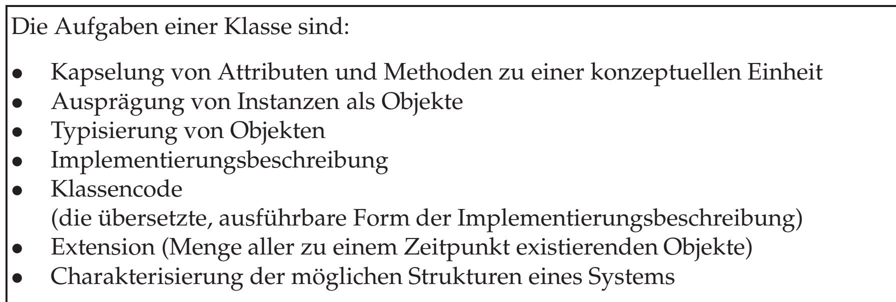

Abbildung 2.1. Aufgabenvielfalt einer Klasse

Metamodellierung

Für die Beschreibung einer diagrammatischen Sprache hat sich aufgrund ihrer zweidimensionalen Darstellungsform die *Metamodellierung* [CEK^{+}00, RA01, CEK01, Béz05, GPHS08, JJM09, AK03] als Präsentationsform durchgesetzt und damit die für Text üblichen Grammatiken abgelöst. Ein *Metamodell* definiert die abstrakte Syntax einer graphischen Notation. Spätestens seit der UML-Standardisierung ist es üblich, als Metamodell-Sprache selbst eine einfache Form von Klassendiagrammen einzusetzen. Dieser Ansatz hat den Vorteil, dass nur eine Sprache erlernt werden muss. Wir diskutieren Metamodellierung im Anhang A und nutzen eine Variante der Klassendiagramme um die graphischen Anteile der UML/P darzustellen.

[1] In Smalltalk manifestiert sich eine Klasse zur Laufzeit als normales Objekt, das wie andere Objekte manipuliert werden kann. Der Inhalt eines solchen Objekts ist allerdings eine Beschreibung von Struktur und Verhalten der diesem Klassen-Objekt zugeordneten Instanzen. Siehe [Gol84].