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

Agile Modellierung mit UML

Codegenerierung, Testfälle, Refactoring

Mit 160 Abbildungen und 30 Tabellen

Springer

Bernhard Rumpe
Institut für Software Systems Engineering
Technische Universität Braunschweig
Mühlenpfordtstr. 23
38106 Braunschweig
e-mail: b.rumpe@tu-bs.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISSN 1439-5428
ISBN 3-540-20905-0 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Satz: Druckfertige Daten des Autors
Herstellung: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3142/YL - 5 4 3 2 1 0

Vorwort

Softwaresysteme sind heutzutage in der Regel komplexe Produkte, für deren erfolgreiche Produktion der Einsatz ingenieurmäßiger Techniken unerlässlich ist. Diese nun mittlerweile mehr als 30 Jahre alte und häufig zitierte Erkenntnis hat dazu geführt, dass in den letzten drei Jahrzehnten innerhalb der Informatik im Gebiet des Software Engineering intensiv an Sprachen, Methoden und Werkzeugen zur Unterstützung des Softwareerstellungsprozesses gearbeitet wird. Trotz großer Fortschritte hierbei muss allerdings festgestellt werden, dass im Vergleich zu anderen, durchgängig viel älteren Ingenieursdisziplinen noch viele Fragen unbeantwortet sind und immer neue Fragestellungen auftauchen.

So macht auch ein oberflächlicher Vergleich zum Beispiel mit dem Gebiet des Bauwesens schnell deutlich, dass dort internatonale Standards eingesetzt werden, um Modelle von Gebäuden zu erstellen, die Modelle zu analysieren und anschließend die Modelle in Bauten zu realisieren. Hierbei sind dann auch die Rollen- und Aufgabenverteilungen allgemein akzeptiert, so dass etwa Berufsgruppen wie Architekten, Statiker sowie Spezialisten für den Tief- und Hochbau existieren.

Eine derartige modellbasierte Vorgehensweise wird zunehmend auch in der Softwareentwicklung favorisiert. Dies bedeutet insbesondere, dass in den letzten Jahren international versucht wird, eine allgemein akzeptierte Modellierungssprache festzulegen, so dass etwa wie im Bauwesen, von einem Software-Architekten erstellte Modelle von einem „Software-Statiker" analysiert werden können, bevor sie von Spezialisten für die Realisierung, also Programmierern in ausführbare Programme umgesetzt werden.

Diese standardisierte Modellierungssprache ist die Unified Modeling Language, die in einem schrittweisen Prozess durch ein international besetztes Konsortium stetig weiterentwickelt wird. Aufgrund der vielfältigen Interessenlagen im Standardisierungsprozess ist mit der aktuellen Version 2.0 der UML eine Sprachfamilie entstanden, deren Umfang, semantische Fundierung und methodische Verwendung noch viele Fragen offen lässt.

Diesem Problem hat sich Herr Rumpe in den letzten Jahren in seinen wissenschaftlichen und praktischen Arbeiten gewidmet, deren Ergebnisse er nun in zwei Büchern einer breiten Leserschaft zugänglich macht. Hierbei hat Herr Rumpe das methodische Vorgehen in den Vordergrund gestellt. Im Einklang mit der heutigen Erkenntnis, dass leichtgewichtige, agile Entwicklungsprozesse insbesondere in kleineren und mittleren Entwicklungsprojekten große Vorteile bieten, hat Herr Rumpe Techniken für einen agilen Entwicklungsprozess entwickelt. Auf dieser Basis hat er dann eine geeignete Modellierungssprache definiert, in dem er ein so genanntes Sprachprofil für die UML definiert hat. In diesem Sprachprofil UML/P hat er die UML geeignet abgespeckt und an einigen Stellen so abgerundet, dass nun eine handhabbare Version der UML insbesondere für einen agilen Entwicklungsprozess vorliegt.

Herr Rumpe hat diese Sprache UML/P ausführlich in dem diesem Buch vorangehenden Buch „Modellierung mit UML“ erläutert. Das Buch bietet eine wesentliche Grundlage für das hier vorliegende Buch, deren Inhalt allerdings auch in diesem Buch noch einmal kurz zusammengefasst wird. Das hier vorliegende Buch mit dem Titel „Agile Modellierung mit UML“ widmet sich nun in erster Linie dem methodischen Umgang mit der UML/P.

Hierbei behandelt Herr Rumpe drei Kernthemen einer modellbasierten Softwareentwicklung. Dies sind

- die Codegenerierung, also der automatisierte Übergang vom Modell zu einem ausführbaren Programm,
- das systematische Testen von Programmen mithilfe einer modellbasierten, strukturierten Festlegung von Testfällen sowie
- die Weiterentwicklung von Modellen durch den Einsatz von Transformations- und Refactoring-Techniken.

Alle drei Kernthemen werden von Herrn Rumpe zunächst systematisch aufgearbeitet und die zugrunde liegenden Begriffe und Techniken werden eingeführt. Darauf aufbauend stellt er dann jeweils seinen Ansatz auf der Basis der Sprache UML/P vor. Diese Zweiteilung und klare Trennung zwischen Grundlagen und Anwendungen machen die Darstellung außerordentlich gut verständlich und bieten dem Leser auch die Möglichkeit, diese Erkenntnisse unmittelbar auf andere modellbasierte Ansätze und Sprachen zu übertragen.

Insgesamt hat dieses Buch einen großen Nutzen sowohl für den Praktiker in der Softwareentwicklung, für die akademische Ausbildung im Fachgebiet Softwaretechnik als auch für die Forschung im Bereich der modellbasierten Entwicklung der Software. Der Praktiker lernt, wie er mit modernen modellbasierten Techniken die Produktion von Code verbessern und damit die Qualität erheblich steigern kann. Dem Studierenden werden sowohl wichtige wissenschaftliche Grundlagen als auch unmittelbare Anwendungen der dargestellten grundlegenden Techniken vermittelt. Dem Wissenschaftler bietet das Buch einen umfassenden Überblick über den heutigen Stand der Forschung in den drei Kernthemen des Buchs.

Das Buch stellt somit einen wichtigen Meilenstein in der Entwicklung von Konzepten und Techniken für eine modellbasierte und ingenieurmäßige Softwareentwicklung dar und bietet somit auch die Grundlage für weitere Arbeiten in der Zukunft. So werden praktische Erfahrungen mit dem Umgang der Konzepte ihre Tragbarkeit validieren. Wissenschaftliche, konzeptionelle Arbeiten werden insbesondere das Thema der Modelltransformation etwa auf der Basis von Graphtransformationen genauer erforschen und darüber hinaus das Gebiet der Modellanalyse im Sinne einer Modellstatik vertiefen.

Ein derartiges vertieftes Verständnis der Informatik-Methoden im Bereich der modellbasierten Softwareentwicklung ist eine entscheidende Voraussetzung für eine erfolgreiche Kopplung mit anderen ingenieurmäßigen Methoden etwa im Bereich von eingebetteten Systemen oder im Bereich von intelligenten, benutzungsfreundlichen Produkten. Die Domänenunabhängigkeit der Sprache UML/P bietet auch hier noch viele Möglichkeiten.

Gregor Engels

Paderborn im September 2004

Inhaltsverzeichnis

1 Einführung 1
- 1.1 Ziele und Inhalte von Band 1 2
- 1.2 Ergänzende Ziele dieses Buchs 4
- 1.3 Überblick 6
- 1.4 Notationelle Konventionen 7

2 Kompakte Übersicht zur UML/P 9
- 2.1 Klassendiagramme 10
  - 2.1.1 Klassen und Vererbung 10
  - 2.1.2 Assoziationen 11
  - 2.1.3 Repräsentation und Stereotypen 14
- 2.2 Object Constraint Language 15
  - 2.2.1 Übersicht über OCL/P 15
  - 2.2.2 Die OCL-Logik 18
  - 2.2.3 Container-Datenstrukturen 19
  - 2.2.4 Funktionen in OCL 26
- 2.3 Objektdiagramme 28
  - 2.3.1 Einführung in Objektdiagramme 29
  - 2.3.2 Komposition 30
  - 2.3.3 Bedeutung eines Objektdiagramms 31
  - 2.3.4 Logik der Objektdiagramme 32
- 2.4 Statecharts 33
  - 2.4.1 Eigenschaften von Statecharts 33
  - 2.4.2 Darstellung von Statecharts 37
- 2.5 Sequenzdiagramme 44

3 Prinzipien der Codegenerierung 49
- 3.1 Konzepte der Codegenerierung 52
  - 3.1.1 Konstruktive Interpretation von Modellen 54
  - 3.1.2 Tests versus Implementierung 56
  - 3.1.3 Tests und Implementierung aus dem gleichen Modell . 59

3.2 Techniken der Codegenerierung ... 60
3.2.1 Plattformabhängige Codegenerierung ... 60
3.2.2 Funktionalität und Flexibilität ... 63
3.2.3 Steuerung der Codegenerierung ... 67
3.3 Semantik der Codegenerierung ... 68
3.4 Parametrisierung eines Werkzeugs ... 71
3.4.1 Implementierung von Werkzeugen ... 71
3.4.2 Darstellung von Skripttransformationen ... 73

4 Transformationen für die Codegenerierung ... 77
4.1 Übersetzung von Klassendiagrammen ... 78
4.1.1 Attribute ... 78
4.1.2 Methoden ... 82
4.1.3 Assoziationen ... 85
4.1.4 Qualifizierte Assoziation ... 90
4.1.5 Komposition ... 93
4.1.6 Klassen ... 95
4.1.7 Objekterzeugung ... 98
4.2 Übersetzung von Objektdiagrammen ... 102
4.2.1 Konstruktiv eingesetzte Objektdiagramme ... 102
4.2.2 Beispiel einer konstruktiven Codegenerierung ... 104
4.2.3 Als Prädikate eingesetzte Objektdiagramme ... 106
4.2.4 Objektdiagramm beschreibt Strukturmodifikation ... 109
4.2.5 Objektdiagramme und OCL ... 112
4.3 Codegenerierung aus OCL ... 112
4.3.1 OCL-Aussage als Prädikat ... 113
4.3.2 OCL-Logik ... 115
4.3.3 OCL-Typen ... 117
4.3.4 Typen als Extension ... 119
4.3.5 Navigation und Flattening ... 120
4.3.6 Quantoren und Spezialoperatoren ... 121
4.3.7 Methodenspezifikation ... 122
4.3.8 Vererbung von Methodenspezifikationen ... 125
4.4 Ausführung von Statecharts ... 125
4.4.1 Methoden-Statecharts ... 127
4.4.2 Umsetzung der Zustände ... 128
4.4.3 Umsetzung der Transitionen ... 132
4.5 Übersetzung von Sequenzdiagrammen ... 136
4.5.1 Sequenzdiagramm als Testtreiber ... 136
4.5.2 Sequenzdiagramm als Prädikat ... 138
4.6 Zusammenfassung zur Codegenerierung ... 140

5 Grundlagen des Testens ... 143
5.1 Einführung in die Testproblematik ... 144
5.1.1 Testbegriffe ... 145
5.1.2 Ziele der Testaktivität ... 147
5.1.3 Fehlerkategorien ... 149
5.1.4 Begriffsbestimmung für Testverfahren ... 150
5.1.5 Suche geeigneter Testdaten ... 152
5.1.6 Sprachspezifische Fehlerquellen ... 153
5.1.7 UML/P als Test- und Implementierungssprache ... 155
5.1.8 Eine Notation für die Testfalldefinition ... 158
5.2 Definition von Testfällen ... 161
5.2.1 Operative Umsetzung eines Testfalls ... 161
5.2.2 Vergleich der Testergebnisse ... 163
5.2.3 Werkzeuge JUnit und VUnit ... 166

6 Modellbasierte Tests ... 171
6.1 Testdaten und Sollergebnis mit Objektdiagrammen ... 172
6.2 Invarianten als Codeinstrumentierungen ... 175
6.3 Methodenspezifikationen ... 177
6.3.1 Methodenspezifikationen als Codeinstrumentierung ... 177
6.3.2 Methodenspezifikationen zur Testfallbestimmung ... 178
6.3.3 Testfalldefinition mit Methodenspezifikationen ... 181
6.4 Sequenzdiagramme ... 182
6.4.1 Trigger ... 183
6.4.2 Vollständigkeit und Matching ... 185
6.4.3 Nicht-kausale Sequenzdiagramme ... 186
6.4.4 Mehrere Sequenzdiagramme in einem Test ... 186
6.4.5 Mehrere Trigger im Sequenzdiagramm ... 187
6.4.6 Interaktionsmuster ... 188
6.5 Statecharts ... 189
6.5.1 Ausführbare Statecharts ... 190
6.5.2 Statechart als Ablaufbeschreibung ... 192
6.5.3 Testverfahren für Statecharts ... 194
6.5.4 Überdeckungsmetriken ... 196
6.5.5 Transitionstests statt Testsequenzen ... 199
6.5.6 Weiterführende Ansätze ... 200
6.6 Zusammenfassung und offene Punkte beim Testen ... 201

7 Testmuster im Einsatz ... 207
7.1 Dummies ... 210
7.1.1 Dummies für Schichten der Architektur ... 211
7.1.2 Dummies mit Gedächtnis ... 212
7.1.3 Sequenzdiagramm statt Gedächtnis ... 214
7.1.4 Abfangen von Seiteneffekten ... 215
7.2 Testbare Programme gestalten ... 215

7.2.1 Statische Variablen und Methoden ... 216
7.2.2 Seiteneffekte in Konstruktoren ... 219
7.2.3 Objekterzeugung ... 219
7.2.4 Vorgefertigte Frameworks und Komponenten ... 220
7.3 Behandlung der Zeit ... 222
7.3.1 Simulation der Zeit im Dummy ... 224
7.3.2 Variable Zeiteinstellung im Sequenzdiagramm ... 224
7.3.3 Muster zur Simulation von Zeit ... 227
7.3.4 Timer ... 228
7.4 Nebenläufigkeit mit Threads ... 229
7.4.1 Eigenes Scheduling ... 230
7.4.2 Sequenzdiagramm als Scheduling-Modell ... 231
7.4.3 Behandlung von Threads ... 232
7.4.4 Muster für die Behandlung von Threads ... 234
7.4.5 Probleme der erzwungenen Sequentialisierung ... 235
7.5 Verteilung und Kommunikation ... 237
7.5.1 Simulation der Verteilung ... 237
7.5.2 Simulation von Singletons ... 239
7.5.3 OCL-Bedingungen über mehrere Lokationen ... 241
7.5.4 Kommunikation simuliert verteilter Prozesse ... 242
7.5.5 Muster für Verteilung und Kommunikation ... 244
7.6 Zusammenfassung ... 245

8 Refactoring als Modelltransformation ... 247
8.1 Einführende Beispiele für Transformationen ... 248
8.2 Methodik des Refactoring ... 253
8.2.1 Technische und methodische Voraussetzungen für Refactoring ... 253
8.2.2 Qualität des Designs ... 255
8.2.3 Refactoring, Evolution und Wiederverwendung ... 256
8.3 Theorie der Modelltransformationen ... 258
8.3.1 Modelltransformationen ... 258
8.3.2 Semantik einer Modelltransformation ... 259
8.3.3 Beobachtungsbegriff ... 266
8.3.4 Transformationsregeln ... 270
8.3.5 Korrektheit von Transformationsregeln ... 272
8.3.6 Ansätze der transformationellen Softwareentwicklung 274

9 Refactoring von Modellen ... 277
9.1 Quellen für UML/P-Refactoring-Regeln ... 278
9.1.1 Definition und Darstellung von Refactoring-Regeln ... 280
9.1.2 Refactoring in Java/P ... 282
9.1.3 Refactoring von Klassendiagrammen ... 288
9.1.4 Refactoring in der OCL ... 294
9.1.5 Einführung von Testmustern als Refactoring ... 296

9.2 Additive Methode für Datenstrukturwechsel 299
9.2.1 Vorgehensweise für den Datenstrukturwechsel 299
9.2.2 Beispiel: Darstellung von Geldbeträgen 302
9.2.3 Beispiel: Einführung des Chairs im Auktionssystem . . . 306
9.3 Zusammenfassung der Refactoring-Techniken 314

10 Zusammenfassung und Ausblick 317

Literatur 321

Index 331

1

Einführung

Der wahre Zweck eines Buches ist,
den Geist hinterrücks zum
eigenen Denken zu verleiten.

Christopher Darlington Morley

Jüngste Beispiele zeigen eindrucksvoll, wie teuer falsche oder fehlerhafte Software beim Absturz einer Rakete oder bei Maut-Projekten werden können.

Um die stetig wachsende Komplexität von Software-basierten Projekten und Produkten sowohl im Bereich betrieblicher Informations- als auch eingebetterer Systeme beherrschbar zu machen, wurde in den letzten Jahren ein wirkungsvolles Portfolio an Konzepten, Techniken und Methoden entwickelt, die die Softwaretechnik zu einer erwachsenen Ingeniersdisziplin heranreifen lassen.

Das Portfolio ist zwar noch nicht vollständig ausgereift, muss aber insbesondere in dem derzeitigen industriellen Softwareentwicklungsprozess sehr viel mehr noch etabliert werden. Die Fähigkeiten moderner Programmiersprachen, Klassenbibliotheken und vorhandener Softwareentwicklungswerkzeuge erlauben uns heute den Einsatz von Vorgehensweisen, die noch vor kurzer Zeit nicht realisierbar schienen.

Als Teil dieses Portfolios behandelt dieses Buch eine auf der UML basierende Methodik, bei der vor allem Techniken zum praktischen Einsatz der UML im Vordergrund stehen. Als wichtigste Techniken werden dabei

- Generierung von Code aus Modellen,
- Modellierung von Testfällen und
- Weiterentwicklung durch Refactoring von Modellen

erkannt und in diesem Buch studiert.

Dabei basiert dieser Band 2 sehr stark auf dem ersten Band „Modellierung mit UML. Sprache, Konzepte und Methodik.“ [Rum04c], in dem das Sprachprofil UML/P detailliert erklärt ist. Es ist daher zu empfehelen, bei der Lektüre dieses Bands [Rum04b] den ersten Band [Rum04c] griffbereit zu

halten, obwohl Teile von Band 1 in kompakter Form in Kapitel 2 wiederholt werden.

1.1 Ziele und Inhalte von Band 1

Gemeinsames Mission Statement beider Bände: Es ist ein Kernziel, für das genannte Portfolio grundlegende Techniken zur modellbasierten Entwicklung zur Verfügung zu stellen. Dabei wird in Band 1 eine Variante der UML vorgestellt, die speziell zur effizienten Entwicklung qualitativ hochwertiger Software und Software-basierter Systeme geeignet ist. Darauf aufbauend enthält dieser Band Techniken zur Generierung von Code, von Testfällen und zum Refactoring der UML/P.

UML-Standard: Der UML 2.0-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 ganz 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.

Band 1 konzentriert sich vor allem auf die Einführung des Sprachprofils und einer allgemeinen Übersicht zur vorgeschlagenen Methodik.

Die UML/P ist als Ergebnis mehrerer Grundlagen- und Anwendungsprojekte entstanden. Insbesondere das in Anhang D, Band 1 dargestellte Anwendungsbeispiel wurde soweit möglich unter Verwendung der hier beschriebenen Prinzipien entwickelt. Das Auktionssystem ist auch deshalb zur Demonstration der in den beiden Büchern entwickelten Techniken geeignet ideal, 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 UML-Sprachprofil UML/P und die darauf aufbauenden Entwicklungstechniken werden daher auf Java zugeschnitten.

Brücke zwischen UML und agilen Methoden: Gleichzeitig bilden die beiden Bücher zwischen den eher als unvereinbar geltenden Ansätzen der agilen Methoden und der Modellierungssprache UML eine elegante Brücke. Agile Methoden und insbesondere Extreme Programming besitzen eine Reihe von interessanten Techniken und Prinzipien, die das Portfolio der Softwaretechnik für bestimmte Projekttypen bereichern. Merkmale dieser Tech-

niken sind der weitgehende Verzicht auf Dokumentation, die Konzentration auf Flexibilität, Optimierung der Time-To-Market und Minimierung der verbrauchten Ressourcen bei gleichzeitiger Sicherung der geforderten Qualität. Damit sind agile Methoden für die Ziele dieses Buchs als Grundlage gut geeignet.

Neue 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 normale 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 Band 1 eine neue agile Methode skizziert, die das UML/P-Sprachprofil als Grundlage für viele Aktivitäten nutzt, ohne die Schwerfälligkeit typischer UML-basierter Methoden zu importieren.

Die beschriebenen Ziele wurden in Band 1 in folgenden Kapitel umgesetzt:

1 Einführung

2 Agile und UML-basierte Methodik
beschreibt die Vorgehensweise zum Einsatz der UML/P im Kontext vorhandener agiler Methoden.

3 Klassendiagramme
führt Form und Verwendung von Klassendiagrammen ein.

4 Object Constraint Language
diskutiert eine syntaktisch auf Java angepasste, sprachlich erweiterte und semantisch konsolidierte Fassung der textuellen Beschreibunsgsprache OCL.

5 Objektdiagramme
diskutiert Sprache und methodischen Einsatz der Objektdiagramme sowie deren Integration mit der OCL-Logik, um damit eine "Logik der Diagramme" zu ermöglichen, in der unerwünschte Situationen, Alternativen und Kombinationen beschrieben werden können.

6 Statecharts
beinhaltet neben der Einführung der Statecharts eine Sammlung von Transformationen zur deren semantikerhaltender Vereinfachung.

7 Sequenzdiagramme
beschreibt Form, Bedeutung und Verwendung von Sequenzdiagrammen.

A Sprachdarstellung durch Syntaxklassendiagramme
bietet eine Kombination aus Extended-Backus-Naur-Form (EBNF) und spezialisierten Klassendiagrammen zur Darstellung der abstrakten Syntax (Metamodell) der UML/P.

B Java
beschreibt die abstrakte Syntax des genutzten Teils von Java.

C Syntax der UML/P
beschreibt die abstrakte Syntax der UML/P.

D Anwendungsbeispiel: Internetbasiertes Auktionssystem
erläutert Hintergrundinformation zu dem in beiden Bänden verwendeten Beispiel des Auktionssystems.

1.2 Ergänzende Ziele dieses Buchs

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 dabei wesentliche Synergie-Effekte, die zum Beispiel bei der Qualitätssicherung, bei der erhöhten Wiederverwendung und der gesteigerten Fähigkeit zur Weiterentwicklung beitragen.

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 Modellen. für spezifische Domänen und Anwendungen. 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. Abbildung 1.1 skizziert die ersten beiden Ziele dieses Buchs.

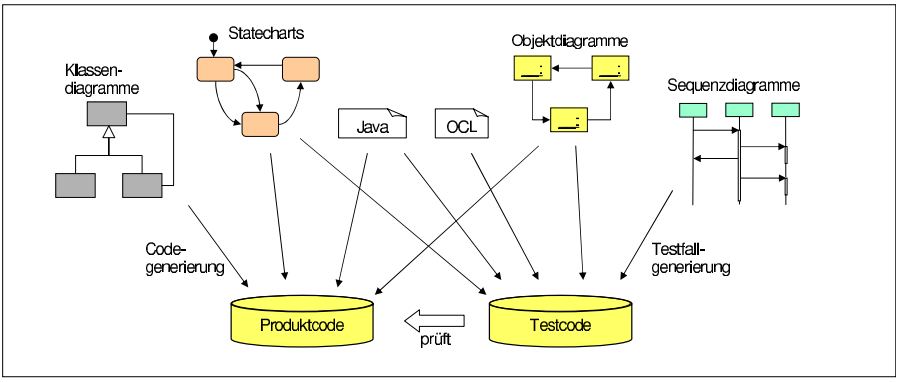

Abbildung 1.1. Notationen der UML/P

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. Ziel des Buchs ist es, diese Konzepte anhand zahlreicher praktischer Beispiele zu erklären und in Form von Testmustern beziehungsweise Refactoring-Techniken für UML-Diagramme aufzubereiten.

Model Driven Architecture (MDA): Initiiert durch das Industriekonsortium OMG[1] wird derzeit an der Intensivierung des werkzeugbasierten Einsatzes der UML in der Softwareentwicklung gearbeitet. Die als MDA be-

[1] Die Object Management Group (OMG) ist für die Definition der UML in Form einer „Technical Recommendation“ zuständig.

zeichnete Technik bietet im Bereich Codegenerierung ähnliche Charakteristiken wie der hier diskutierte Ansatz. Jedoch fehlt eine geeignet angepasste Methodik. Der in diesem Buch vorgestellte Ansatz zum Refactoring bietet dazu eine adäquate Ergänzung zur „horizontalen“ Transformation.

Abgrenzung: Mit den behandelten Techniken konzentriert sich dieses Buch vor allem auf die Unterstützung der Entwurfs-, Implementierungs- und Wartungsaktivitäten, allerdings ohne diese in allen Facetten abzudecken. Wichtig, aber unbehandelt sind auch Techniken und Notationen zur Erhebung und zum Management von Anforderungen, zur Projektplanung und -durchführung, zur Kontrolle sowie zum Versions- und Änderungsmanagement. Stattdessen wird an geeigneten Stellen auf entsprechende weiterführende Literatur verwiesen.

1.3 Überblick

Abbildung 1.2 erlaubt einen guten Überblick über den Matrix-artigen Aufbau weiter Teile beider Bände. Während dieser Band sich verstärkt um methodische Fragestellungen kümmert wird die UML/P im Band 1 erklärt.

	Allgemeines	Klassen-diagramme	OCL	Objekt-diagramme	Statecharts	Sequenz-diagramme
Einführung und Diskussion	*Kapitel 2 (Band 1)*	*Kapitel 3 (Band 1)*	*Kapitel 4 (Band 1)*	*Kapitel 5 (Band 1)*	*Kapitel 6 (Band 1)*	*Kapitel 7 (Band 1)*
Syntax	*Anhang A & B & C.1 (Band 1)*	*Anhang C.2 (Band 1)*	*Anhang C.3 (Band 1)*	*Anhang C.4 (Band 1)*	*Anhang C.5 (Band 1)*	*Anhang C.6 (Band 1)*
Codegenerierung	Kapitel 3	Kapitel 4.1	Kapitel 4.3	Kapitel 4.2	Kapitel 4.4	Kapitel 4.5
Testfallmodellierung	Kapitel 5	(Kapitel 7)	Abschnitte 6.2 & 6.3	Abschnitt 6.1	Abschnitt 6.5	Abschnitt 6.4
Refactoring	Kapitel 8	Abschitte 9.1 & 9.2	Abschnitt 9.1	(Abschnitt 9.2)	*Abschnitt 6.6 (Band 1)*	(Abschnitt 9.2)

Abbildung 1.2. Übersicht über den Inhalt beider Bände

Das Kapitel 2 gibt eine kurze und kompakte Zusammenfassung von Band 1, [Rum04c]. Dabei wird vor allem das dort eingeführte Sprachprofil der UML/P sehr kompakt und daher unvollständig vorgestellt.

Die Kapitel 3 und 4 diskutieren prinzipielle und technische Problemstellungen zur Codegenerierung. Dies beinhaltet die Architektur und die Steuerung eines Codegenerators genauso wie die Frage, welche Teile der UML/P für Test- beziehungsweise für Produktionscode geeignet sind. Darauf aufbauend wird ein Mechanismus zur Beschreibung von Codegenerierung eingeführt. Ergänzend dazu wird anhand ausgewählter Teile der verschiedenen UML/P-Notationen gezeigt, wie daraus Test- und Produktionscode ge-

neriert werden kann. Dabei werden sowohl Alternativen diskutiert als auch die Kombination von Transformationen demonstriert.

Die Kapitel 5 und 6 erörtern die aus der Literatur bekannten Begriffsbildungen für Testverfahren und die beim Testen zu beachtenden Besonderheiten, die aufgrund der Nutzung der UML/P als Test- und Implementierungssprache entstehen. Es wird beschrieben, wie eine Architektur für die automatisierte Testausführung aussieht und wie UML/P-Diagramme eingesetzt werden, um Testfälle zu definieren.

Kapitel 7 zeigt anhand von Testmustern die Verwendbarkeit der UML/P-Diagramme zur Definition von Testfällen. Diese Testmuster enthalten Erfahrungswissen zur Definition von testbaren Programmen insbesondere für funktionale Tests von verteilten und nebenläufigen Softwaresystemen.

Die Kapitel 8 und 9 diskutieren Techniken zur Transformation von Modellen und Code und stellen damit Refactoring als semantikerhaltende Transformation auf eine fundierte Basis. Für Refactoring wird ein expliziter und praktisch verwendbarer Beobachtungsbegriff eingeführt und es wird diskutiert, wie vorhandene Refactoring-Techniken auf die UML/P übertragen werden können. Schließlich wird eine *additive Vorgehensweise* vorgeschlagen, die größere Refactorings unter Verwendung von OCL-Invarianten unterstützt.

1.4 Notationelle Konventionen

In diesem Buch werden mehrere Diagrammarten und textuelle Notationen genutzt. Damit sofort erkennbar ist, welches Diagramm oder welche textuelle Notation jeweils dargestellt ist, wird abweichend von der UML 2.0 rechts oben eine Marke in einer der in Abbildung 1.3 dargestellten Formen angegeben. Diese Form ist auch zur Markierung textueller Teile geeignet und flexibler als die UML 2.0-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-Zeichensatz. Zur besseren Lesbarkeit werden einzelne Schlüsselwörter hervorgehoben oder unterstrichen.

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`}.

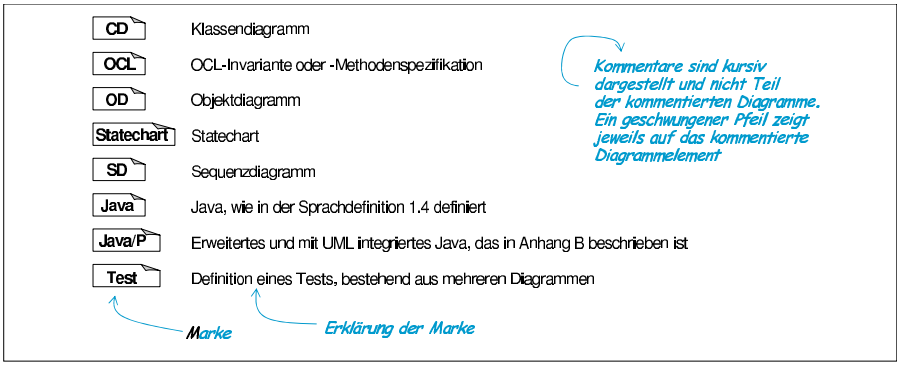

Abbildung 1.3. Marken für Diagramme und Textteile

Danksagung

Wie auch an Band 1 haben an der Erstellung dieses Buchs eine Reihe von Personen direkt oder indirekt mitgewirkt. Neben einem Verweis auf die Danksagung von Band 1 möchte ich insbesondere meinen Kolleginnen und Kollegen an der Technischen Universität Braunschweig für die freundliche Aufnahme sowie meinen Mitarbeiterinnen und Mitarbeitern für die hervorragende Unterstützung bei der Fertigstellung dieses Buchs bedanken. Im Kontext des gerade in Aufbau befindlichen neuen Instituts für Software Systems Engineering ist es durchaus ambitioniert zwei Bücher parallel zu einer Reihe neuer Vorlesungen, Seminaren und Praktika fertigzustellen.

2

Kompakte Übersicht zur UML/P

Die Grenzen meiner Sprache
sind die Grenzen meiner Welt.
Ludwig Wittgenstein

Dieses Kapitel beinhaltet eine kompakte Zusammenfassung der in Band 1 [Rum04c] eingeführten UML/P. Diese Zusammenfassung beschreibt einige, aber nicht alle Besonderheiten und Abweichungen des UML/P-Profils vom UML 2.0 Standard. Für eine genauere Lektüre sei Band 1 empfohlen. Kenner der UML können dieses Kapitel bei Bedarf später nachschlagen. Die verwendeten Beispiele zur Einführung der Sprache beziehen sich im Wesentlichen auf die in Band 1 beschriebene Auktionssystem-Anwendung.

2.1 Klassendiagramme **10**
2.2 Object Constraint Language **15**
2.3 Objektdiagramme **28**
2.4 Statecharts **33**
2.5 Sequenzdiagramme **44**

2.1 Klassendiagramme

Klassendiagramme bilden das architekturelle Rückgrat vieler Systemmodellierungen. Dementsprechend haben Klassendiagramme und darin besonders Klassen eine Reihe von Aufgaben zu erfüllen (Abbildung 2.1): Klassendiagramme beschreiben die Struktur beziehungsweise Architektur eines Systems, auf der nahezu alle anderen Beschreibungstechniken basieren. Das Konzept *Klasse* ist durchgängig in Modellierung und Programmierung eingesetzt und bietet daher ein Rückgrat für Traceability von Anforderungen und Fehlern entlang der verschiedenen Aktivitäten eines Projekts. Klassendiagramme bilden das Skelett für fast alle weiteren Notationen und Diagrammarten, da diese sich jeweils auf die in Klassendiagrammen definierten Klassen und Methoden stützen. So werden in der Analyse Klassendiagramme genutzt, um Konzepte der realen Welt zu strukturieren, während in Entwurf und Implementierung Klassendiagramme vor allem zur Darstellung einer strukturellen Sicht des Softwaresystems genutzt werden.

Die Aufgaben einer Klasse sind:

- Kapselung von Attributen und Methoden zu einer konzeptuellen Einheit
- Ausprägung von Instanzen als Objekte
- Typisierung von Objekten
- Implementierungsbeschreibung
- Klassencode (die übersetzte, ausführbare Form der Implementierungsbeschreibung)
- Extension (Menge aller zu einem Zeitpunkt existierenden Objekte)
- Charakterisierung der möglichen Strukturen eines Systems

Abbildung 2.1. Aufgabenvielfalt einer Klasse

2.1.1 Klassen und Vererbung

Die Abbildung 2.2 enthält eine Einordnung der wichtigsten Begriffe für Klassendiagramme.

In Abbildung 2.3 ist ein einfaches Klassendiagramm bestehend aus einer Klasse zu sehen. Je nach Grad der Detaillierung können Attribute und Methoden der Klassen weggelassen oder nur teilweise angegeben werden. Auch Typen von Attributen und Methoden, Sichtbarkeitsangaben und weitere Modifikatoren sind optional.

Zur Strukturierung von Klassen in überschaubare Hierarchien wird die Vererbungsbeziehung eingesetzt. Abbildung 2.4 demonstriert Vererbung und Interface-Implementierung anhand der Gemeinsamkeiten mehrerer Nachrichtenarten des in Kapitel D, Band 1 beschriebenen Auktionssystems. Die Erweiterung von Interfaces ist darin nicht abgebildet. Sie wird wie die Vererbung zwischen Klassen dargestellt.

Klasse. Eine Klasse besteht aus einer Sammlung von Attributen und Methoden, die den Zustand und das Verhalten ihrer *Instanzen* (*Objekte*) festlegt. Klassen sind durch Assoziationen und Vererbungsbeziehungen miteinander verknüpft. Ein *Klassenname* erlaubt es, die Klasse zu identifizieren.

Attribut. Die Zustandskomponenten einer Klasse werden als Attribute bezeichnet. Sie beinhalten grundsätzlich *Name* und *Typ*.

Methode. Die Funktionalität einer Klasse ist in Methoden abgelegt. Eine Methode besteht aus einer *Signatur* und einem *Rumpf*, der die Implementierung beschreibt. Bei einer *abstrakten* Methode fehlt der Rumpf.

Modifikator. Zur Festlegung von Sichtbarkeit, Instanziierbarkeit und Veränderbarkeit des modifizierten Elements können die Modifikatoren `public`, `protected`, `private`, `readonly`, `abstract`, `static` und `final` auf Klassen, Methoden und Attribute angewandt werden. Für die ersten vier genannten Modifikatoren gibt es in UML/P die graphischen Varianten „`+`“, „`#`“ und „`-`“ und „`?`“.

Konstanten sind als spezielle Attribute mit den Modifikatoren `static` und `final` definiert.

Vererbung. Stehen zwei Klassen in Vererbungsbeziehung, so vererbt die *Oberklasse* ihre Attribute und Methoden an die *Unterklasse*. Die Unterklasse kann weitere Attribute und Methoden hinzufügen und Methoden *redefinieren* – soweit die Modifikatoren dies erlauben. Die Unterklasse bildet einen *Subtyp* der Oberklasse, der es nach dem *Substitutionsprinzip* erlaubt, Instanzen der Unterklasse dort einzusetzen, wo Instanzen der Oberklasse erforderlich sind.

Interface. Ein Interface (Schnittstelle) beschreibt die Signaturen einer Sammlung von Methoden. Im Gegensatz zur Klasse werden keine Attribute (nur Konstanten) und keine Methodenrümpfe angegeben. Interfaces sind verwandt zu abstrakten Klassen und können untereinander ebenfalls in einer Vererbungsbeziehung stehen.

Typ ist ein Grunddatentyp wie `int`, eine Klasse oder ein Interface.

Interface-Implementierung ist eine der Vererbung ähnliche Beziehung zwischen einem Interface und einer Klasse. Eine Klasse kann beliebig viele Interfaces implementieren.

Assoziation ist eine binäre Beziehung zwischen Klassen, die zur Realisierung struktureller Information verwendet wird. Eine Assoziation besitzt einen *Assoziationsnamen*, für jedes Ende einen *Rollennamen*, eine *Kardinalität* und eine Angabe über die *Navigationsrichtungen*.

Kardinalität. Die Kardinalität (Multiplicity) wird für jedes Assoziationsende angegeben. Sie ist von der Form „`0..1`“, „`1`“ oder „`*`“ und beschreibt, ob eine Assoziation in dieser Richtung optional oder eindeutig ist beziehungsweise mehrfache Bindung erlaubt.

Abbildung 2.2. Begriffsdefinition für Klassendiagramme

2.1.2 Assoziationen

Eine Assoziation dient dazu, Objekte zweier Klassen in Beziehung zu setzen. Mithilfe von Assoziationen können komplexe Datenstrukturen gebildet werden und Methoden benachbarter Objekte aufgerufen werden. Abbildung 2.5

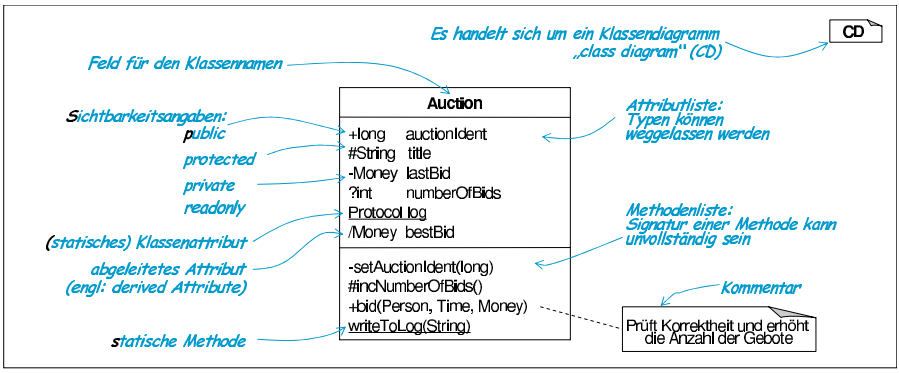

Abbildung 2.3. Eine Klasse im Klassendiagramm

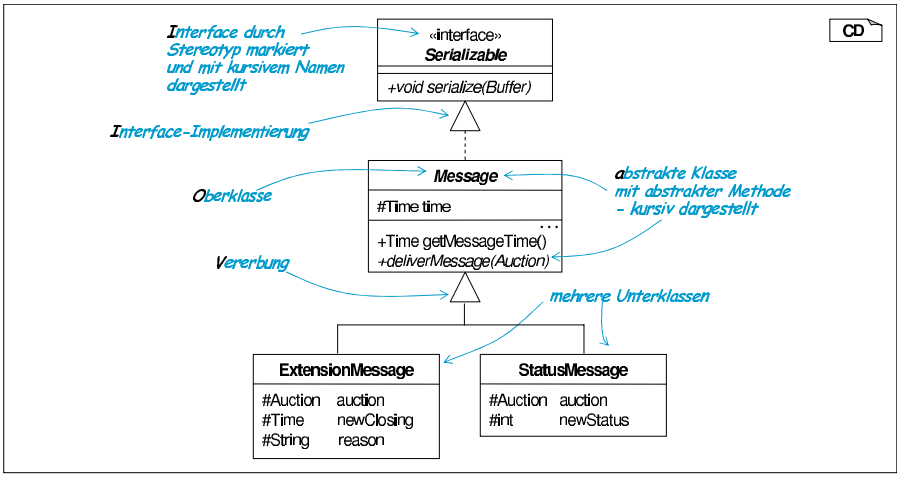

Abbildung 2.4. Vererbung und Interface-Implementierung

beschreibt einen Ausschnitt aus dem Auktionssystem mit mehreren Assoziationen in unterschiedlichen Formen.

Eine Assoziation besitzt im Normalfall einen *Assoziationsnamen* und für jedes der beiden Enden je eine *Assoziationsrolle*, eine Angabe der *Kardinalität* und eine Beschreibung der möglichen *Navigationsrichtungen*. Einzelne Angaben können im Modell auch weggelassen werden, wenn sie zur Darstellung des gewünschten Sachverhalts keine Rolle spielen und die Eindeutigkeit nicht verloren geht.

Assoziationen können grundsätzlich uni- oder bidirektional sein. Ist keine explizite Pfeilrichtung angegeben, so wird von einer bidirektionalen Assoziation ausgegangen. Formal werden die Navigationsmöglichkeiten in dieser Situation als unspezifiziert und damit nicht als eingeschränkt betrachtet.

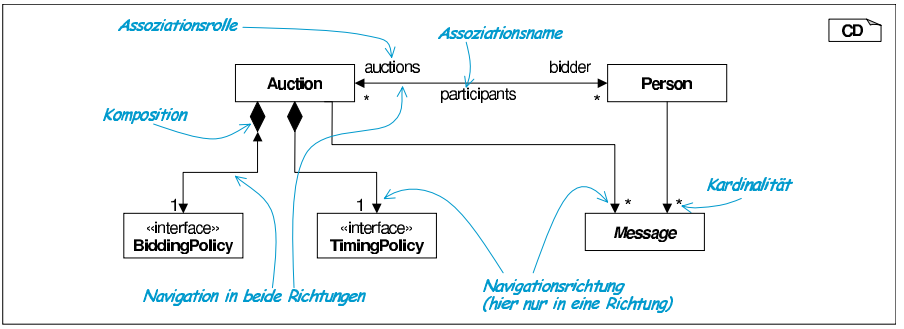

Abbildung 2.5. Klassendiagramm mit Assoziationen

Eine besondere Form der Assoziation ist die *Komposition*. Sie wird durch eine ausgefüllte Raute an einem Assoziationsende dargestellt. In einer Komposition sind die *Teilobjekte* in einer starken auch den Lebenszyklus betreffenden Abhängigkeit vom *Ganzen*.

Der UML-Standard bietet eine Reihe zusätzlicher Merkmale für Assoziationen an, die Assoziationseigenschaften genauer regeln. Abbildung 2.6 beinhaltet einige häufig eingesetzte Merkmale, wie zum Beispiel {`ordered`}, das einen geordneten Zugriff mit Index erlaubt. Mit {`frozen`} wird angezeigt, dass nach der Initialisierung eines Auktionsobjekts die beiden Assoziationen zu den Policy-Objekten nicht mehr verändert werden. Mit {`addOnly`} wird modelliert, dass in der Assoziation nur Objekte hinzugefügt werden dürfen, das Entfernen jedoch verboten ist.

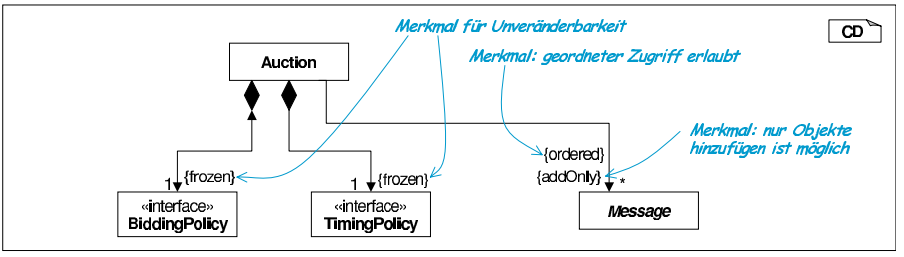

Abbildung 2.6. Merkmale für Assoziationen

Qualifizierte Assoziation bieten die Möglichkeit, aus einer Menge von zugeordneten Objekten mithilfe des *Qualifikators* ein einzelnes Objekt zu selektieren. Abbildung 2.7 zeigt mehrere auf unterschiedliche Weise qualifizierte Assoziationen.

Während in explizit qualifizierten Assoziationen die Art des Qualifikators wählbar ist, stehen in geordneten Assoziationen ganzzahlige Intervalle beginnend mit der `0` zur Verfügung.

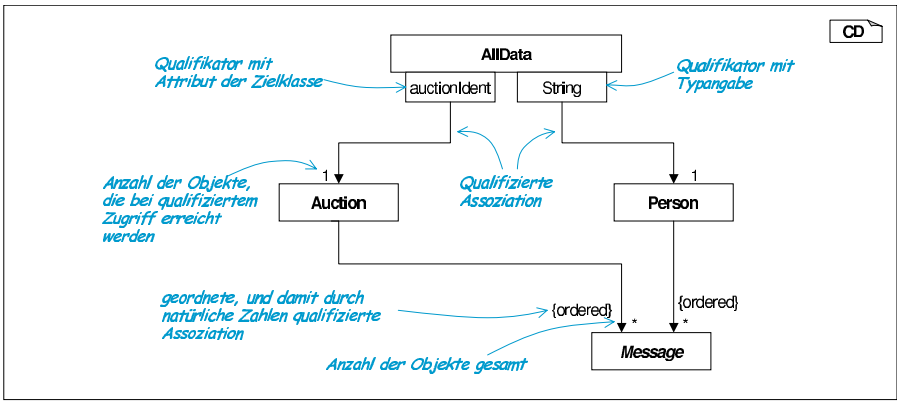

Abbildung 2.7. Qualifizierte Assoziationen

2.1.3 Repräsentation und Stereotypen

Klassendiagramme haben oft das Ziel, die für eine bestimmte Aufgabe notwendige Datenstruktur einschließlich ihrer Zusammenhänge zu beschreiben. Eine vollständige Liste aller Methoden und Attribute ist für einen Überblick dabei eher hinderlich. Ein Klassendiagramm stellt daher meist eine unvollständige *Sicht* des Gesamtsystems dar. So können einzelne Klassen oder Assoziationen fehlen. Innerhalb der Klassen können Attribute und Methoden weggelassen oder unvollständig dargestellt werden.

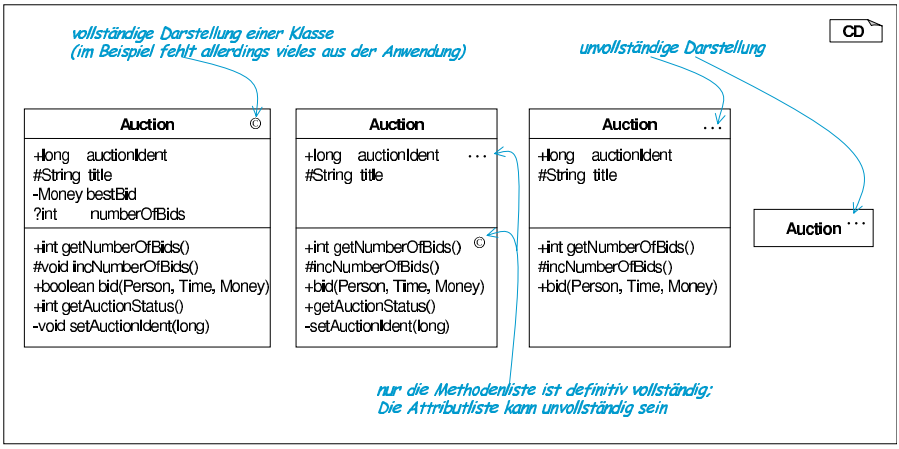

Abbildung 2.8. Formen der Darstellung einer Klasse

Um einem UML-Diagramm anzusehen, ob die darin enthaltene Information vollständig ist, werden in UML/P die Repräsentationsindikatoren

„. . .“ zur Markierung unvollständiger Information und „©“ zur Darstellung vollständiger Information verwendet (Abbildung 2.8).

Die Indikatoren „©“ und „. . .“ wirken nicht auf die Klasse selbst, sondern auf ihre Darstellung innerhalb des Klassendiagramms. Ein „©“ beim Klassennamen besagt, dass sowohl die Attribut- als auch die Methodenliste vollständig ist. Demgegenüber bedeutet der Unvollständigkeits-Indikator „. . .“, dass die Darstellung unvollständig sein *kann*.

Die beiden Repräsentationsindikatoren sind nur eine spezielle From von Merkmalen mit eigener syntaktischer Darstellung. Die UML bietet sehr allgemein die Möglichkeit, Modellelemente mit Stereotypen und Merkmalen zu markieren (Abbildung 2.9), die deren allgemeine Syntax die Formen «stereotyp», {`merkmal`} oder {`merkmal=wert`} hat.

Stereotyp. Ein Stereotyp klassifiziert Modellelemente wie beispielsweise Klassen oder Attribute. Durch einen Stereotyp wird die Bedeutung des Modellelements spezialisiert und kann so beispielsweise bei der Codegenerierung spezifischer behandelt werden. Ein Stereotyp kann eine Menge von Merkmalen besitzen.

Merkmal. Ein Merkmal beschreibt eine Eigenschaft eines Modellelements. Ein Merkmal wird notiert als Paar bestehend aus *Schlüsselwort* und *Wert*. Mehrere solche Paare können zu einer Liste zusammengefasst werden.

Modellelemente sind die (wesentlichen) Bausteine der UML-Diagramme. Beispielsweise hat das Klassendiagramm als Modellelemente Klassen, Interfaces, Attribute, Methoden, Vererbungsbeziehungen und Assoziationen.
Merkmale und Stereotypen können auf Modellelemente angewandt werden, sind aber selbst keine Modellelemente.

Abbildung 2.9. Begriffsdefinition Merkmal und Stereotyp

2.2 Object Constraint Language

Die Object Constraint Language (OCL) ist eine eigenschaftsorientierte Modellierungssprache, die eingesetzt wird, um Invarianten sowie Vor- und Nachbedingungen von Methoden zu modellieren. Die in der UML/P enthaltene OCL/P ist eine syntaktsich an Java angepasste und erweiterte Variante des OCL-Standards.

2.2.1 Übersicht über OCL/P

Abbildung 2.10 erläutert die wichtigsten Begriffe der OCL.

Eines der herausragenden Merkmale von OCL-Bedingungen ist ihre grundsätzliche Einbettung in einen Kontext, bestehend aus UML-Modellen. Das Klassendiagramm in Abbildung 2.11 stellt einen solchen Kontext bereit, in dem zum Beispiel folgende Aussage formuliert werden kann:

Bedingung. Eine Bedingung ist eine boolesche Aussage über ein System. Sie beschreibt eine Eigenschaft, die ein System oder ein Ergebnis besitzen soll. Ihre Interpretation ergibt grundsätzlich einen der Wahrheitswerte **true** oder **false**.

Kontext. Eine Bedingung ist in einen Kontext eingebettet, über den sie Aussagen macht. Der Kontext wird definiert durch eine Menge von in der Bedingung verwendbaren *Namen* und ihren Signaturen. Dazu gehören Klassen-, Methoden- und Attributnamen des Modells und insbesondere im Kontext einer Bedingung explizit eingeführte Variablen.

Interpretation einer Bedingung wird anhand einer konkreten Objektstruktur vorgenommen. Dabei werden die im Kontext eingeführten Variablen mit Werten beziehungsweise Objekten belegt.

Invariante beschreibt eine Eigenschaft, die in einem System zu jedem (beobachteten) Zeitpunkt gelten soll. Die Beobachtungszeitpunkte können eingeschränkt sein, um zeitlich begrenzte Verletzungen zum Beispiel während der Ausführung einer Methode zu erlauben.

Vorbedingung einer Methode charakterisiert die Eigenschaften, die gelten müssen, damit diese Methode ein definiertes und sinnvolles Ergebnis liefert. Ist die Vorbedingung nicht erfüllt, so wird über das Ergebnis keine Aussage getroffen.

Nachbedingung einer Methode beschreibt, welche Eigenschaften nach Ausführung der Methode gelten. Dabei kann auf Objekte in dem Zustand zurückgegriffen werden, der unmittelbar vor dem Methodenaufruf (zur „Zeit" der Interpretation der Vorbedingung) gültig war. Nachbedingungen werden anhand zweier Objektstrukturen interpretiert, die die Situationen vor und nach dem Methodenaufruf darstellen.

Methodenspezifikation ist ein Paar bestehend aus Vor- und Nachbedingung.

Query ist eine von der Implementierung angebotene Methode, deren Aufruf keine Veränderung des Systemzustands hervorruft. Es dürfen neue Objekte als Aufrufergebnis erzeugt werden. Allerdings dürfen diese nicht mit dem Systemzustand durch Links verbunden sein. Dadurch sind Queries ohne Seiteneffekte und können in OCL-Bedingungen verwendet werden.

Abbildung 2.10. Begriffsdefinitionen zur OCL

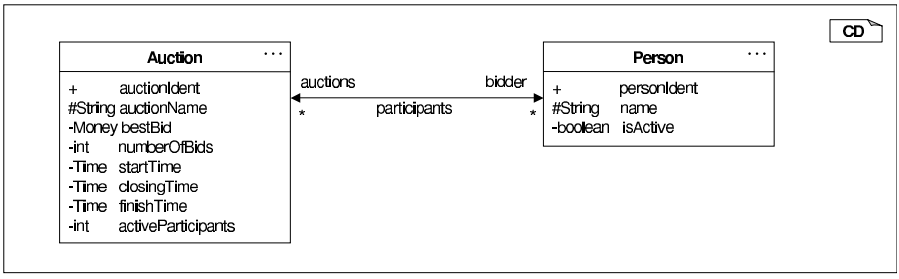

Abbildung 2.11. Ausschnitt des Auktionssystems

OCL

```
context Auction a inv Bidders2:
  a.activeParticipants == { p in a.bidder | p.isActive }.size
```

Die Aussage mit dem Namen `Bidders2` wird quantifiziert über alle Objekte `a` der Klasse Auktion. Darin enthalten sind Navigationsausdrücke wie `a.bidder`, die es erlauben über Assoziationen zu navigieren. Eine *Mengenkomprehension* `{p in ...| ...}` selektiert alle aktiven Personen einer Auktion. Diese Mengenkomprehension ist eine komfortable Erweiterung gegenüber dem OCL-Standard. Sie wird später in mehreren Formen ausführlicher besprochen.

Wird in einem Kontext statt einem expliziten Namen nur die Klasse festgelegt, so wird implizit der Name `this` als vereinbart angenommen. In diesem Fall kann auf Attribute auch direkt zugegriffen werden. *Geschlossene Bedingungen* besitzen einen leeren Kontext:

```
inv:
  forall a in Auction:
    a.startTime <= Time.now() implies
      a.numberOfBids == 0
```

Mithilfe des **let**-Konstrukts können Zwischenergebnisse einer Hilfsvariablen zugewiesen werden, um diese im Rumpf des Konstrukts gegebenenfalls mehrfach zu nutzen:

```
context  inv SomeArithmeticTruth:
  let middle = (a+b)/2
  in
    a<=b  implies  a<=middle  and  middle<=b
```

Das **let**-Konstrukt kann auch dazu genutzt werden, Hilfsfunktionen zu vereinbaren:

```
context Auction a inv Time2:
  let min(Time x, Time y) = (x<=y) ? x : y
  in
    min(a.startTime, min(a.closingTime,a.finishTime))
           == a.startTime
```

Wie bereits gesehen, kann die Fallunterscheidung `.?.:.` aber auch die verbose, äquivalente Fassung `if-then-else` eingesetzt werden. Im Gegensatz zur imperativen Fallunterscheidung sind immer der `then`- und der `else`-Zweig anzugeben.

Eine spezielle Form der Fallunterscheidung erlaubt die Behandlung von Typkonversionen, wie sie bei Subtyphierarchien gelegentlich auftreten. OCL/P bietet dafür eine typsichere Konstruktion an, die eine Kombination einer Typkonversion und einer Abfrage nach dessen Konvertierbarkeit darstellt:

```
context Message m inv:
  let Person p = m instanceof BidMessage ? m.bidder : null
  in ...
```

Zusätzlich zu einer normalen Fallunterscheidung wird im `then`-Zweig der Fallunterscheidung der Typ der Variable `m` auf den Subtyp gesetzt und ermöglicht dadurch die Selektion `m.bidder`. Die Typsicherheit bleibt trotz Konversion erhalten.

Als Grunddatentypen stehen die aus Java bekannten Sorten `boolean`, `char`, `int`, `long`, `float`, `byte`, `short` und `double` und darauf arbeitende Operatoren ohne Seiteffekte zur Verfügung. Ausgeschlossen sind damit die Inkrementoperatoren `++` und `--` sowie alle Zuweisungsformen. Neu sind die booleschen Operatoren `implies` und `<=>`, die zur Darstellung von Implikationen und Äquivalenzen dienen, und die Postfixoperatoren `@pre` und `**`.

Priorität	Operator	Assoziativität	Operand(en), Bedeutung
14	`@pre`	links	Wert des Ausdrucks in der Vorbedingung
	`**`	links	Transitive Hülle einer Assoziation
13	`+, -, ~`	rechts	Zahlen
	`!`	rechts	Boolean: Negation
	`(type)`	rechts	Typkonversion (Cast)
12	`*, /, %`	links	Zahlen
11	`+, -`	links	Zahlen, String (+)
10	`<<, >>, >>>`	links	Shifts
9	`<, <=, >, >=`	links	Vergleiche
	`instanceof`	links	Typvergleich
	`in`	links	Element von
8	`==, !=`	links	Vergleiche
7	`&`	links	Zahlen, Boolean: striktes und
6	`^`	links	Zahlen, Boolean: xor
5	`|`	links	Zahlen, Boolean: striktes oder
4	`&&`	links	Boolesche Logik: und
3	`||`	links	Boolesche Logik: oder
2,7	`implies`	links	Boolesche Logik: impliziert
2,3	`<=>`	links	Boolesche Logik: äquivalent
2	`? :`	rechts	Auswahlausdruck (if-then-else)

Tabelle 2.12: Prioritäten der OCL-Operatoren

Der Datentyp `String` wird wie in Java nicht als Grunddatentyp, sondern als standardmäßig zur Verfügung stehende Klasse verstanden.

2.2.2 Die OCL-Logik

Die richtige Definition der einer Bedingungssprache zugrunde liegenden Logik ist für einen praxistauglichen Einsatz der Sprache wichtig. Deshalb wurde für OCL/P eine zweiwertige Logik und ein spezieller Umgang mit dem