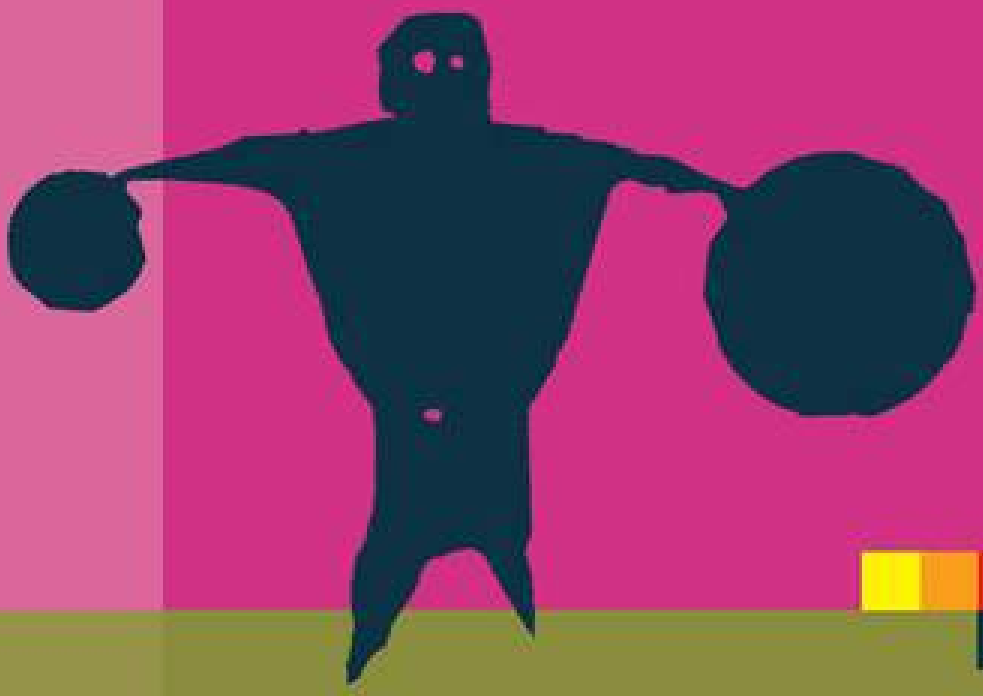

Xpert.press

Bernhard Rumpe

Agile Modellierung mit UML

2. Auflage

Springer Vieweg

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

2. Auflage

Springer Vieweg

Prof. Dr. Bernhard Rumpe
Aachen, Deutschland

ISSN 1439-5428
ISBN 978-3-642-22429-4 e-ISBN 978-3-642-22430-0
DOI 10.1007/978-3-642-22430-0

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

Springer Vieweg

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist eine Marke von Springer DE.
Springer DE ist Teil der Fachverlagsgruppe Springer Science+Business Media
www.springer-vieweg.de

Vorwort

Vorwort zur 2ten Auflage

Da dieses das zweite Buch zur agilen Softwareentwicklung mit der UML ist, der geneigte Leser Buch 1 [Rum11] daher wahrscheinlich kennt und das Vorwort in [Rum11] für beide Bücher gilt, erlaube ich mir an dieser Stelle auf Buch 1 zu verweisen. Darin ist Folgendes dargelegt:

- Agile Methoden und modellbasierte Methoden sind beide erfolgreiche Vorgehensweisen.
- Aber eine Annäherung beider Vorgehensweisen hat bisher nicht stattgefunden.
- Auf Basis der Grundidee, Modelle statt Programmiersprachen zu verwenden besteht aber genau dazu die Möglichkeit.
- Das vorliegende Buch liefert dazu einen Beitrag in Form der UML/P.
- In der zweiten Auflage wurde die UML/P aktualisiert und auf die UML 2.3 sowie Java Version 6 angepasst.

Viel Spaß bei der Nutzung des Buchs beziehungsweise seiner Inhalte.

Bernhard Rumpe

Aachen im März 2012

Weiterführendes Material:

http://mbse.se-rwth.de

Vorwort zur 1ten Auflage

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 internationale 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 Agile und UML-basierte Methodik 9
2.1 Das Portfolio der Softwaretechnik 11
2.2 Extreme Programming (XP) 13
2.3 Ausgesuchte Entwicklungspraktiken 20
2.3.1 Pair Programming 20
2.3.2 Test-First-Ansatz 21
2.3.3 Refactoring 24
2.4 Agile UML-basierte Vorgehensweise 25
2.5 Zusammenfassung 32

3 Kompakte Übersicht zur UML/P 33
3.1 Klassendiagramme 34
3.1.1 Klassen und Vererbung 34
3.1.2 Assoziationen 35
3.1.3 Repräsentation und Stereotypen 38
3.2 Object Constraint Language 38
3.2.1 OCL/P-Übersicht 39
3.2.2 Die OCL-Logik 42
3.2.3 Container-Datenstrukturen 43
3.2.4 Funktionen in OCL 50
3.3 Objektdiagramme 52
3.3.1 Einführung in Objektdiagramme 52
3.3.2 Komposition 55
3.3.3 Bedeutung eines Objektdiagramms 55
3.3.4 Logik der Objektdiagramme 56

3.4 Statecharts ... 57
3.4.1 Eigenschaften von Statecharts ... 57
3.4.2 Darstellung von Statecharts ... 61
3.5 Sequenzdiagramme ... 67

4 Prinzipien der Codegenerierung ... 73
4.1 Konzepte der Codegenerierung ... 76
4.1.1 Konstruktive Interpretation von Modellen ... 78
4.1.2 Tests versus Implementierung ... 80
4.1.3 Tests und Implementierung aus dem gleichen Modell ... 83
4.2 Techniken der Codegenerierung ... 85
4.2.1 Plattformabhängige Codegenerierung ... 85
4.2.2 Funktionalität und Flexibilität ... 88
4.2.3 Steuerung der Codegenerierung ... 91
4.3 Semantik der Codegenerierung ... 92
4.4 Flexible Parametrisierung eines Codegenerators ... 94
4.4.1 Implementierung von Werkzeugen ... 95
4.4.2 Darstellung von Skripttransformationen ... 98

5 Transformationen für die Codegenerierung ... 103
5.1 Übersetzung von Klassendiagrammen ... 104
5.1.1 Attribute ... 104
5.1.2 Methoden ... 108
5.1.3 Assoziationen ... 111
5.1.4 Qualifizierte Assoziation ... 116
5.1.5 Komposition ... 119
5.1.6 Klassen ... 121
5.1.7 Objekterzeugung ... 126
5.2 Übersetzung von Objektdiagrammen ... 129
5.2.1 Konstruktiv eingesetzte Objektdiagramme ... 129
5.2.2 Beispiel einer konstruktiven Codegenerierung ... 131
5.2.3 Als Prädikate eingesetzte Objektdiagramme ... 133
5.2.4 Objektdiagramm beschreibt Strukturmodifikation ... 136
5.2.5 Objektdiagramme und OCL ... 139
5.3 Codegenerierung aus OCL ... 139
5.3.1 OCL-Aussage als Prädikat ... 140
5.3.2 OCL-Logik ... 142
5.3.3 OCL-Typen ... 144
5.3.4 Typ als Extension ... 146
5.3.5 Navigation und Flattening ... 147
5.3.6 Quantoren und Spezialoperatoren ... 148
5.3.7 Methodenspezifikation ... 149
5.3.8 Vererbung von Methodenspezifikationen ... 152
5.4 Ausführung von Statecharts ... 152
5.4.1 Methoden-Statecharts ... 154

5.4.2 Umsetzung der Zustände ... 155
5.4.3 Umsetzung der Transitionen ... 160
5.5 Übersetzung von Sequenzdiagrammen ... 163
5.5.1 Sequenzdiagramm als Testtreiber ... 164
5.5.2 Sequenzdiagramm als Prädikat ... 165
5.6 Zusammenfassung zur Codegenerierung ... 168

6 Grundlagen des Testens ... 171
6.1 Einführung in die Testproblematik ... 173
6.1.1 Testbegriffe ... 173
6.1.2 Ziele der Testaktivitäten ... 174
6.1.3 Fehlerkategorien ... 177
6.1.4 Begriffsbestimmung für Testverfahren ... 178
6.1.5 Suche geeigneter Testdaten ... 179
6.1.6 Sprachspezifische Fehlerquellen ... 180
6.1.7 UML/P als Test- und Implementierungssprache ... 182
6.1.8 Eine Notation für die Testfalldefinition ... 186
6.2 Definition von Testfällen ... 188
6.2.1 Operative Umsetzung eines Testfalls ... 188
6.2.2 Vergleich der Testergebnisse ... 190
6.2.3 Werkzeug JUnit ... 193

7 Modellbasierte Tests ... 197
7.1 Testdaten und Sollergebnis mit Objektdiagrammen ... 198
7.2 Invarianten als Codeinstrumentierungen ... 201
7.3 Methodenspezifikationen ... 203
7.3.1 Methodenspezifikationen als Codeinstrumentierung ... 203
7.3.2 Methodenspezifikationen zur Testfallbestimmung ... 204
7.3.3 Testfalldefinition mit Methodenspezifikationen ... 207
7.4 Sequenzdiagramme ... 208
7.4.1 Trigger ... 209
7.4.2 Vollständigkeit und Matching ... 211
7.4.3 Nicht-kausale Sequenzdiagramme ... 212
7.4.4 Mehrere Sequenzdiagramme in einem Test ... 212
7.4.5 Mehrere Trigger im Sequenzdiagramm ... 213
7.4.6 Interaktionsmuster ... 214
7.5 Statecharts ... 215
7.5.1 Ausführbare Statecharts ... 216
7.5.2 Statechart als Ablaufbeschreibung ... 217
7.5.3 Testverfahren für Statecharts ... 220
7.5.4 Überdeckungsmetriken ... 222
7.5.5 Transitionstests statt Testsequenzen ... 225
7.5.6 Weiterführende Ansätze ... 226
7.6 Zusammenfassung und offene Punkte beim Testen ... 227

8 Testmuster im Einsatz 233
8.1 Dummies 236
8.1.1 Dummies für Schichten der Architektur 237
8.1.2 Dummies mit Gedächtnis 238
8.1.3 Sequenzdiagramm statt Gedächtnis 240
8.1.4 Abfangen von Seiteneffekten 241
8.2 Testbare Programme gestalten 241
8.2.1 Statische Variablen und Methoden 242
8.2.2 Seiteneffekte in Konstruktoren 245
8.2.3 Objekterzeugung 245
8.2.4 Vorgefertigte Frameworks und Komponenten 246
8.3 Behandlung der Zeit 249
8.3.1 Simulation der Zeit im Dummy 250
8.3.2 Variable Zeiteinstellung im Sequenzdiagramm 250
8.3.3 Muster zur Simulation von Zeit 253
8.3.4 Timer 254
8.4 Nebenläufigkeit mit Threads 255
8.4.1 Eigenes Scheduling 256
8.4.2 Sequenzdiagramm als Scheduling-Modell 257
8.4.3 Behandlung von Threads 258
8.4.4 Muster für die Behandlung von Threads 260
8.4.5 Probleme der erzwungenen Sequentialisierung 261
8.5 Verteilung und Kommunikation 263
8.5.1 Simulation der Verteilung 263
8.5.2 Simulation von Singletons 265
8.5.3 OCL-Bedingungen über mehrere Lokationen 267
8.5.4 Kommunikation simuliert verteilte Prozesse 268
8.5.5 Muster für Verteilung und Kommunikation 270
8.6 Zusammenfassung 271

9 Refactoring als Modelltransformation 273
9.1 Einführende Beispiele für Transformationen 274
9.2 Methodik des Refactoring 280
9.2.1 Technische und methodische Voraussetzungen für Refactoring 280
9.2.2 Qualität des Designs 281
9.2.3 Refactoring, Evolution und Wiederverwendung 283
9.3 Modelltransformationen 284
9.3.1 Formen von Modelltransformationen 284
9.3.2 Semantik einer Modelltransformation 285
9.3.3 Beobachtungsbegriff 292
9.3.4 Transformationsregeln 297
9.3.5 Korrektheit von Transformationsregeln 298
9.3.6 Ansätze der transformationellen Softwareentwicklung 300
9.3.7 Transformationssprachen 303

10 Refactoring von Modellen 305
10.1 Quellen für UML/P-Refactoring-Regeln 306
10.1.1 Definition und Darstellung von Refactoring-Regeln . . . 308
10.1.2 Refactoring in Java/P 310
10.1.3 Refactoring von Klassendiagrammen 316
10.1.4 Refactoring in der OCL 322
10.1.5 Einführung von Testmustern als Refactoring 324
10.2 Additive Methode für Datenstrukturwechsel 328
10.2.1 Vorgehensweise für den Datenstrukturwechsel 328
10.2.2 Beispiel: Darstellung von Geldbeträgen 331
10.2.3 Beispiel: Einführung des Chairs im Auktionssystem . . . 335
10.3 Zusammenfassung der Refactoring-Techniken 343

11 Zusammenfassung und Ausblick 347

Literatur 353

Index 369

1

Einführung

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

Christopher Darlington Morley

Viele Projekte zeigen mittlerweile eindrucksvoll, wie teuer falsche oder fehlerhafte Software werden kann.

Um die stetig wachsende Komplexität von Software-basierten Projekten und Produkten sowohl in den Bereichen betrieblicher oder administrativer Informations- und Websysteme als auch bei Cyber-Physischen Systemen wie Auto, Flugzeug, Produktionsanlagen, E-Health- und mobilen Systemen beherrschbar zu machen, wurde in den letzten Jahren ein wirkungsvolles Portfolio an Konzepten, Techniken und Methoden entwickelt, die die Softwaretechnik zu einer erwachsenen Ingenieursdisziplin heranreifen lassen.

Das Portfolio ist noch keineswegs ausgereift, muss aber insbesondere in dem derzeitigen industriellen Softwareentwicklungsprozess noch sehr viel mehr 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.

Weiterführendes Material:

http://mbse.se-rwth.de

B.Rumpe, *Agile Modellierung mit UML*, 2. Aufl., Xpert.press
DOI 10.1007/978-3-642-22430-0_1, © Springer-Verlag Berlin Heidelberg 2012