

# Qualitätssicherung für objektorientierte Software: Anforderungsermittlung und Test gegen die Anforderungsspezifikation

Fachbereich Informatik der  
FernUniversität - Gesamthochschule - in Hagen

Zur Erlangung des Grades eines  
Doktors der Naturwissenschaften  
(Dr. rer. nat.)

angenommene  
Dissertation  
von

Dipl.-Inform. Mario Winter

Erster Berichterstatter:	Prof. Dr. H.-W. Six
Zweiter Berichterstatter:	Prof. Dr. A. Poetsch-Heffter
Datum der mündlichen Prüfung:	10. September 1999

**dissertation.de**

**Verlag im Internet**

Sonderausgabe des Werkes mit der ISBN-Nummer: 3-89825-031-8

**dissertation.de**

**Verlag im Internet**

Leonhardtstr. 8-9

D-14 057 Berlin

Email: [dissertation.de@snafu.de](mailto:dissertation.de@snafu.de)

Internetadresse: <http://www.dissertation.de>

# Wann beginnt das Testen?

Geleitwort von Prof. Dr. Andreas Spillner

Die ersten Programme in der Frühzeit der Computertechnik (ab 1940) wurden sicherlich getestet oder — besser formuliert — vor deren Einsatz ausprobiert. Auftraggeber, Anwender, Programmierer und Tester waren in einer Person vereint. Der Anwendungsbereich der Programme war auf rein mathematische Berechnungen beschränkt.

Mit zunehmender Leistung der Rechner wurden die dadurch erbrachten Möglichkeiten auch für Kunden interessant. Es gab einen Auftraggeber. Aufgabe der Rechner war immer noch das Rechnen, allerdings oft mit einem starken Anwendungsbezug: zum Beispiel der möglichst exakten Vorausberechnung der Flugbahnen von Geschossen.

Die Rechner wurden schneller; die Aufgaben, die jetzt mit ihnen erledigt werden konnten, umfangreicher und anspruchsvoller. Viele Auswertungen und (Volks-)Zählungen wurden jetzt von den elektronischen „Helfern“ übernommen. Eine Person konnte die Realisierung der gestellten Aufgabe nicht mehr allein bewältigen. Es mussten „Dinge“ besprochen, vereinbart und festgehalten werden. Vielleicht wurde die Prüfung von Programmen oder Teilen davon bereits zu diesem Zeitpunkt von anderen Personen vorgenommen. Die Gruppe der Tester entstand.

Der Test, die Überprüfung, ob eine gestellte Aufgabe entsprechend umgesetzt und realisiert wurde, bedarf der genauen Kenntnis der gestellten Aufgabe. Was soll berechnet werden, was ist das erwartete Ergebnis, und wie kann die Überprüfung vorgenommen werden? Das „Wie“ beschäftigt die Forschung seit den 70er Jahren. Ein unstrukturierter Test, ein Test „aus dem Bauch“ heraus, ist leider immer noch in der Praxis eine häufig anzutreffende Test-„Methode“.

Heutzutage kann von einer Konsolidierung bei den Testmethoden für prozedurale Software ausgegangen werden. Es ist klar, welche Verfahren bei welcher Art von Software zur welchen Fehlerhinweisen führen bzw. am besten geeignet sind, Mängel und Fehler aufzudecken. Neue Herausforderungen entstehen durch das objektorientierte Vorgehen. Testmethoden für prozedural entwickelte Programme lassen sich nur bedingt zur Prüfung objektorientierter Systeme einsetzen. Neue, andere Ansätze sind zwingend erforderlich.

In den zurückliegenden Jahren wurde auch zunehmend deutlich, dass die Überlegungen zum Testen nicht erst am Ende der Implementierung beginnen dürfen. Wenn nicht beispielsweise schon bei der Konzeption der Systemstruktur überlegt wird, wie sie die Testbarkeit beeinflusst, werden nicht testbare Systeme entworfen und realisiert.

Mario Winter verknüpft in der Arbeit beide Aspekte: Test von objektorientierter Software und Überlegungen hierzu in der frühen Entwicklungsphase „Anforderungsermittlung“. Die UML (Unified Modelling Language) hat sich als Standard-Darstellungsmethode für die Entwicklung von objektorientierten Systemen etabliert. Sie dient Herrn Winter als Grundlage, um zu einem frühen Zeitpunkt Testüberlegungen anzustellen und, neben der Vorbereitung auf das „eigentliche“ dynamische Testen (mittels Programmausführung), schon in dieser Entwicklungsphase Fehler und Mängel aufzudecken. Er verfolgt somit einen Ansatz, der leider noch immer die Ausnahme ist.

Generell ist für die Zukunft zu wünschen — oder besser zu fordern — dass bei neuen Entwicklungen zur Verbesserung des Softwareentwicklungsprozesses, wie es die objektorientierte Vorgehensweise und auch die Verwendung der UML darstellen, die Testaktivitäten bereits frühzeitig in die Überlegungen einbezogen werden. Erst wenn diese Forderung konsequent umgesetzt werden wird, hat das Testen und Prüfen auch bei der Konzeption von neuen Verfahren und Vorgehensweisen die Bedeutung, die es in der Praxis bereits heute innehält: eine sehr hohe.

Prof. Dr. Andreas Spillner

Hochschule Bremen

Sprecher der GI-Fachgruppe 2.1.7 „Testen, Analysieren und Verifizieren von Software“

Bremen im Oktober 1999

# Danksagung

Diese Arbeit entstand in den Jahren 1995-1999 am Lehrgebiet Praktische Informatik III im Fachbereich Informatik der FernUniversität in Hagen, in dem ich — nach langjähriger industrieller und universitärer Projektarbeit in der Software-Entwicklung — seit 1994 tätig bin.

Mein Dank gilt allen, die mir bei der Erstellung dieser Arbeit zur Seite gestanden haben — zuerst und ganz besonders aber meinem Betreuer Prof. Dr. Hans-Werner Six für die interessante Themenstellung und seine Bereitschaft, mich in den vielen Diskussionen neben der Vermittlung des Sinns für das Wesentliche auch in die Grundregeln wissenschaftlichen Publizierens eingeweiht zu haben. Herrn Prof. Dr. Arnd Poetsch-Heffter danke ich für die spontane Übernahme des Korreferats und seine hilfreichen und kritischen Bemerkungen zur Arbeit.

Ich danke meinen Kolleginnen und Kollegen an der FernUniversität für ihre ständige Gesprächsbereitschaft, besonders Dr. Bernd-Uwe Pagel für die lebendigen Dialoge in den frühen Stadien der Arbeit und Dr. Georg Kösters für seine Geduld und seinen Humor, mit denen er mich in unzähligen Diskussionen vor Um- und Irrwegen bewahrt hat.

Von unschätzbarem Wert bei der Abfassung der Dissertation waren mir die langjährige Vertrautheit in die industrielle Problematik der Qualitätssicherung, in die mich die Mitglieder des Arbeitskreises „Testen objektorientierter Programme“ (TOOP) der GI-Fachgruppe 2.1.7., „Test, Analyse und Verifikation von Software“ (TAV) eingeführt haben. Mein Dank gilt insbesondere Harry Sneed und Prof. Dr. Andreas Spillner, die das Testen von Software in Deutschland überhaupt erst „hoffähig“ gemacht haben.

Zusätzlich danke ich den Studierenden der FernUniversität (Cornelia Wulf, Gabriele Mohl, Thorsten Ritter), die im Rahmen Ihrer Diplomarbeiten die Konzepte der Arbeit implementiert und validiert haben. Das häufig mutierende Manuskript wurde von Brigitte Wellmann mit engelhafter Geduld korrektur gelesen. Weiterer „Testleser“ war Dr. Markus Müller.

Last but not least danke ich meiner Frau Petra, die es verstanden hat und versteht, die Launen eines nur allzuoft an die Arbeit denkenden Ehemanns zu ertragen und unseren lieben Kindern Mischa und Jonas verbunden mit der Entschuldigung dafür, dass sie ihren Papa in der letzten Phase der Arbeit oft nur sporadisch zu Gesicht bekamen. Sie haben mir die Energie und das Selbstvertrauen gegeben, diese Arbeit zu Ende zu führen.



# Inhaltsverzeichnis

<b>I</b>	<b>Einleitung</b>	<b>1</b>
<b>1</b>	<b>Motivation und Überblick</b>	<b>2</b>
1.1	Begriffe und Gebiete der Qualitätssicherung	4
1.2	Geschichte	7
1.3	Überblick über die Arbeit	9
<b>2</b>	<b>Entwicklungstätigkeiten</b>	<b>12</b>
2.1	Geschäftsprozesse und Anforderungsermittlung	14
2.2	Entwurf und Implementation	24
2.3	Test und Administration	26
<b>3</b>	<b>Probleme der Qualitätssicherung für objektorientierte Anwendungen</b>	<b>27</b>
3.1	Formale Methoden	28
3.2	Struktureller white-box Test	30
3.3	Black-box Test gegen die Anforderungsspezifikation	33
3.4	Relevante Arbeiten: Test gegen objektorientierte Anforderungsspezifikationen	36
<b>4</b>	<b>Probleme der Anforderungsermittlung mit Use Cases</b>	<b>41</b>
4.1	Fallbeispiel Bankautomat	41
4.2	Offene Fragen und Kritik	46
4.3	Relevante Arbeiten: Präzisierung von Use Cases	50
4.4	Zwischenfazit	54
<b>II</b>	<b>Anforderungsermittlung</b>	<b>57</b>
<b>5</b>	<b>Grundlegende Konzepte</b>	<b>58</b>
5.1	Use Case Schritte	61
5.2	Use Case Schrittgraphen	64
5.3	SCORES-Metamodell (I)	68
5.4	Modellierungsregeln (I)	69
5.5	Darstellungskonventionen	71
<b>6</b>	<b>Semantik</b>	<b>73</b>
6.1	Schritte in Use Case Schrittgraphen	73
6.2	Interpretation und Entscheidbarkeit	78
6.3	Modellierungsregeln (II)	80

---

6.4	Makroschritte, „extends“, „uses“ und Generalisierung . . . . .	82
<b>7</b>	<b>Kopplung von Use Cases und Klassenmodell . . . . .</b>	<b>85</b>
7.1	Elemente des Klassenmodells . . . . .	86
7.2	Klassenbereiche . . . . .	87
7.3	Wurzelklasse und Wurzeloperation . . . . .	89
7.4	SCORES-Metamodell (II) . . . . .	90
7.5	Vollständigkeitsmetriken . . . . .	91
7.6	Exkurs: Use Case Schrittgraphen und Aktivitätsdiagramme . . . . .	96
<b>8</b>	<b>Methodisches Vorgehen . . . . .</b>	<b>98</b>
8.1	Funktionale Zerlegung . . . . .	98
8.2	Externes Verhalten . . . . .	99
8.3	Struktureller Aufbau . . . . .	100
8.4	Internes Verhalten . . . . .	100
8.5	Administrative Tätigkeiten . . . . .	101
8.6	Zwischenfazit . . . . .	102
<b>III</b>	<b>Validierung und Verifikation der Anforderungsspezifikation . . . . .</b>	<b>103</b>
<b>9</b>	<b>Validierung . . . . .</b>	<b>104</b>
9.1	Reviewtechniken . . . . .	107
9.2	Funktionale Zerlegung . . . . .	109
9.3	Externes Verhalten . . . . .	111
9.4	Scores-Metamodell (III) . . . . .	113
<b>10</b>	<b>Validierungsmetriken . . . . .</b>	<b>115</b>
10.1	Use Case Schrittüberdeckung . . . . .	116
10.2	Use Case Kantenüberdeckung . . . . .	117
10.3	Use Case Szenario-Überdeckung . . . . .	118
10.4	Grenze-Inneres- und Pfadüberdeckung . . . . .	119
<b>11</b>	<b>Verifikation . . . . .</b>	<b>125</b>
11.1	Struktureller Aufbau . . . . .	125
11.2	Externes vs. internes Verhalten . . . . .	127
11.3	Episoden . . . . .	129
11.4	Simulationsregeln . . . . .	132
11.5	Objektorientierte Walk-Throughs . . . . .	133
11.6	Vollständiges Scores-Metamodell . . . . .	135
<b>12</b>	<b>Verifikationsmetriken . . . . .</b>	<b>137</b>
12.1	Minimale Mehrfach-Bedingungsüberdeckung . . . . .	137



---

12.2	Vollständige und polymorphe Operationsüberdeckung .....	140
<b>13</b>	<b>Verfolgbarkeit</b>	<b>147</b>
13.1	Konzepte .....	148
13.2	Elemente der SCORES-Anforderungsspezifikation .....	151
<b>IV</b>	<b>Test gegen die Anforderungsspezifikation</b>	<b>157</b>
<b>14</b>	<b>Probleme und Ziele</b>	<b>158</b>
14.1	Probleme .....	158
14.2	Ziele .....	159
<b>15</b>	<b>Black-box Test</b>	<b>161</b>
15.1	Überblick .....	161
15.2	Testfallermittlung .....	162
15.3	Methodisches Vorgehen .....	165
<b>16</b>	<b>SCORES grey-box Test</b>	<b>169</b>
16.1	Klassen-Botschaftsdiagramm der Anforderungsspezifikation .....	169
16.2	Klassen-Botschaftsdiagramm der Implementation .....	170
16.3	Verfolgbarkeit und Klassen-Botschaftsdiagramme .....	175
16.4	Testfallerweiterung .....	176
16.5	Methodisches Vorgehen .....	177
<b>17</b>	<b>Umgebungsaufbau und Testdatengenerierung</b>	<b>182</b>
17.1	Generierung von Objektkonstellationen .....	182
17.2	Komplexität .....	184
17.3	Interaktionsdaten .....	186
<b>18</b>	<b>Endekriterien, Ausführung und Auswertung</b>	<b>187</b>
18.1	Test- und Testendekriterien .....	187
18.2	Testausführung .....	189
18.3	Testauswertung .....	190
<b>V</b>	<b>Werkzeugunterstützung</b>	<b>193</b>
<b>19</b>	<b>Konzepte</b>	<b>194</b>
19.1	Entwicklung .....	194
19.2	Architektur .....	195
<b>20</b>	<b>SCORESTOOL</b>	<b>198</b>
20.1	Anforderungsspezifikation .....	198

---

20.2	Validierung . . . . .	200
20.3	Verifikation . . . . .	204
20.4	Test . . . . .	205
 <b>VI Resümee und Ausblick</b>		<b>209</b>
<b>21</b>	<b>Resümee</b>	<b>210</b>
21.1	Zusammenfassung und Ergebnisse . . . . .	210
21.2	Erfahrungen . . . . .	214
<b>22</b>	<b>Ausblick</b>	<b>215</b>
22.1	Laufende Arbeiten . . . . .	215
22.2	Zukünftige Arbeiten und offene Fragen. . . . .	216
 <b>Literatur</b>		<b>219</b>
 <b>Anhang A Klassen-Botschaftsdiagramme</b>		<b>236</b>
 <b>Anhang B Syntax der Test-Skriptsprache</b>		<b>252</b>
 <b>Index</b>		<b>255</b>
 <b>Kurzbiographie des Autors</b>		<b>259</b>

# Abbildungsverzeichnis

Abb. 1.1	Gebiete der Qualitätssicherung	6
Abb. 1.2	Fokus der Arbeit (Grafik nach [PagSix94])	10
Abb. 2.1	Der Rational Unified Process (nach [JBR99])	12
Abb. 2.2	Geschäftsprozess, WFMS und Anwendungen	15
Abb. 2.3	Die vier Tätigkeiten der Anforderungsermittlung (aus: [Poh96])	18
Abb. 2.4	Elemente des UML Use Case Diagramms	22
Abb. 2.5	Darstellung der drei Objektarten nach Jacobson	22
Abb. 2.6	Interaktionsdiagramm der Operation „ <b>ArrayStack.push</b> “	23
Abb. 2.7	Teilprozesse und Modelle in OOSE ([JCJ+92])	25
Abb. 2.8	Von der Anforderungsermittlung zum Entwurf: OOSE (nach [JCJ+92])	25
Abb. 2.9	Testtätigkeiten (nach [Grimm95])	26
Abb. 3.1	Problematik des Klassentests (nach [Sneed96])	31
Abb. 3.2	Object-Z basierte Testmethode von [CLS+98]	39
Abb. 3.3	Herkömmlicher (black-box) Systemtest	40
Abb. 4.1	Use Case Geld Abheben: Textuelle Spezifikation (übersetzt aus [JCJ+92])	43
Abb. 4.2	Use Case Diagramm „Bankautomat“	44
Abb. 4.3	Vereinfachtes Objektdiagramm zum Use Case Geld Abheben	44
Abb. 4.4	Interaktionsdiagramm zum Use Case Geld Abheben	45
Abb. 4.5	Die drei Informationsarten bei der Anforderungsermittlung ([PohHau97])	46
Abb. 5.1	Die Abstraktionslücke zwischen Use Case Modell und Klassenmodell	59
Abb. 5.2	Das Kosten-„Teufelsquadrat“ der Anforderungsermittlung	60
Abb. 5.3	Spezifikation des Use Case Anmelden	62
Abb. 5.4	Textuelle Spezifikation einiger Use Case Schritte	64
Abb. 5.5	Bedingungen der Schritte im Use Case Schrittgraph Anmelden	67
Abb. 5.6	Kanten im Use Case Schrittgraph Anmelden	68
Abb. 5.7	Use Case Schrittgraph Metamodell	68
Abb. 5.8	Use Case Schrittgraphen Anmelden (a) und Geld Abheben (b)	72
Abb. 5.9	(Erfolgreiche) Szenarien zu den Use Cases Anmelden (a) und Geld Abheben (b)	72
Abb. 6.1	Semantik 1, einfachster Fall	73
Abb. 6.2	Semantik 2: kein Endschrift, ein Folgeschritt	74
Abb. 6.3	Semantik 3: Endschrift, ein Folgeschritt	74
Abb. 6.4	Semantik 4: kein Endschrift, mehrere sich ausschliessende Folgeschritte	75
Abb. 6.5	(a) unabhängige Folgeschritte (b) abkürzende Notation	76
Abb. 6.6	Semantik 5: kein Endschrift, „Parallel“ ausführbare Folgeschritte	76

Abb. 6.7 Semantik 6: Endschrift, mehrere Folgeschritte	77
Abb. 6.8 Semantik 7: Makroschritt	78
Abb. 6.9 Nachbedingungen der im Szenario Erfolgreiche Anmeldung besuchten Use Case Schritte	79
Abb. 6.10 Bedingungen der im Szenario Erfolgreiche Anmeldung besuchten Kanten	79
Abb. 6.11 Illustration der Semantik von «uses» (a) und «extends» (b)	83
Abb. 7.1 Interaktionsschritt mit Klassenbereich und Wurzeloperation Op_1	90
Abb. 7.2 Erweitertes Use Case Metamodell	91
Abb. 7.3 Klassenmodell des Bankautomaten	94
Abb. 7.4 Use Case Anmelden: Klassenbereich	95
Abb. 7.5 Use Case Schrittgraph mit Klassenbereichen, Wurzelklassen und Wurzeloperationen	95
Abb. 9.1 Exemplarische Checklistenpunkte für die Validierung	110
Abb. 9.1 Scores Metamodell mit Test-Szenario	114
Abb. 10.1 Beispiele zur Use Case Schritt- (a), -Kanten (b) und Szenarioüberdeckung (c)	117
Abb. 10.2 Use Case Schrittgraph (a) und fünf textuell beschriebene Szenarien (b)	121
Abb. 10.3 Sequenzdiagramm zum Test-Szenario Online-Anmeldung, Use Case Anmelden	122
Abb. 10.4 Überdeckung des Use Case Schrittgraphen Anmelden	122
Abb. 10.5 Benutzerfreundliche Sicht für das ATM Test-Szenario Auszahlung OK	123
Abb. 10.6 Bildschirmskizzen für das ATM Test-Szenario Anmeldung OK	124
Abb. 11.1 Exemplarische Checklistenpunkte für die Verifikation	134
Abb. 11.1 Sequenzdiagramm, Episode mit Wurzelklasse <b>Transaktion</b> und Wurzeloperation führePlanAus135	
Abb. 11.2 Vollständiges Scores Metamodell	136
Abb. 12.1 Klassendiagramm und Episode mit redefinierter Operation	141
Abb. 12.2 Episode zum Schritt PIN Prüfen im Test-Szenario Online-Anmeldung	145
Abb. 12.3 Episode zum Schritt PIN Prüfen im Test-Szenario Offline-Anmeldung	146
Abb. 13.1 Verfolgbarkeit für den Grey-box Systemtest	148
Abb. 13.2 Vertikale Verfolgbarkeit (nach [Davis90])	149
Abb. 13.3 UML Auxiliary Elements — Dependencies and Templates ([OMG97])	150
Abb. 13.4 Einfache strukturelle Entwurfsänderungen	154
Abb. 14.1 Vom Test der Anforderungsspezifikation zum System- und Abnahmetest	159
Abb. 15.1 Struktur der Test-Elemente	162
Abb. 15.2 Testskript für die Klasse Stack	164
Abb. 15.3 Algorithmus Black-box Test	165
Abb. 15.4 Black-box Test	166
Abb. 15.5 <i>Gerichteter azyklischer Graph zur Inter-Use Case-Referenzfunktion</i>	167
Abb. 15.6 Black-box Testfälle zum Use Case Anmelden	168
Abb. 16.1 KBDA für einige Operationen im (Domänen-) Klassenmodell des Bankautomaten	171
Abb. 16.2 Java Code (a) und Klassen-Botschaftsdiagramm für die Implementation (b)	174
Abb. 16.3 KBDI vs. KBDA	175
Abb. 16.4 Algorithmus Grey-box Test	178

---

Abb. 16.5 SCORES grey-box Testfallgenerierung mit dem KBD	179
Abb. 16.6 SCORES Grey-box Test	180
Abb. 16.7 Grey-box Testfälle	181
Abb. 17.1 Beispiel zur Generierung von Objektkonstellationen	183
Abb. 17.2 Einfache Beispiele zur polynomialen Transformation von SAT auf GOK	185
Abb. 17.3 Komplexeres Beispiel zur polynomialen Transformation von SAT auf GOK	186
Abb. 17.4 Interaktionsdaten zum Test-Skript Anmeldung OK	186
Abb. 19.1 SCORESTOOL: Grundlegender Aufbau (vgl. [Kösters97])	196
Abb. 20.1 SCORESTOOL Klassendiagramm-Editor	199
Abb. 20.2 SCORESTOOL Use Case Spezifikations-Editor	199
Abb. 20.3 SCORESTOOL Use Case Schrittgraph-Editor	200
Abb. 20.4 SCORESTOOL Objektkonstellations-Editor	201
Abb. 20.5 SCORESTOOL Szenario-Browser: textuelle Sicht	202
Abb. 20.6 SCORESTOOL Szenario-Browser: GUI-Sicht	203
Abb. 20.7 SCORESTOOL Szenario-Browser: Metrik-Sicht	203
Abb. 20.8 SCORESTOOL Episoden-Editor	204
Abb. 20.9 Testfall- und Testdaten-Generierung	205
Abb. 20.10 SCORESTOOL Szenario-Browser: Szenario-Generierung	206
Abb. 21.1 SCORES: Übergang vom Use Case Modell zum Klassenmodell	211
Abb. 21.2 SCORES: Abdeckung der drei Informationsarten (nach [PohHau97])	212
Abb. 21.3 Qualitätssicherung mit SCORES	213
Abb. 22.1 Vollständiger GUI-basierter Systemtest	217
Abb. 22.2 Beziehungsgeflechte: Modular (a) und Objektorientiert (b)	240
Abb. 22.4 Funktion PolyKBD <sub>I</sub>	244
Abb. 22.3 Algorithmus Generiere KBD <sub>I</sub>	245



# Teil I

## Einleitung

Teil I besteht aus vier Kapiteln. Im ersten Kapitel motivieren wir zunächst die Thematik „Qualitätssicherung in der Softwareentwicklung“ und geben einen Überblick über die Arbeit. Danach skizzieren wir in Kapitel 2 die Tätigkeiten bei der Entwicklung objektorientierter Software und stellen damit den Bezugsrahmen für die Arbeit her.

In Kapitel 3 wenden wir uns der Qualitätssicherung in der objektorientierten Softwareentwicklung zu. Da die Komplexität hier in den Interaktionen zwischen Objekten liegt, erweisen sich strukturelle (white-box) Klassentests als wenig aussagekräftig. Aufgrund des z.T. sehr hohen Aufwandes ist ihr Einsatz nicht effektiv. Für funktionale (black-box) Tests der Anwendung gegen die Anforderungsspezifikation sind im Falle objektorientierter Anforderungsspezifikationen neue Verfahren notwendig — womit wir bei der Kernthematik der Arbeit angelangt sind.

In Kapitel 4 listen wir Schwachpunkte der objektorientierten Anforderungsspezifikation mit Use Cases und Klassenmodellen auf, welche die Qualitätssicherung bei der Anforderungsermittlung und den Test der Anwendung gegen die Anforderungsspezifikation betreffen. In diesem Zusammenhang stellen wir auch relevante Arbeiten anderer Autoren zu diesen Themen vor. Ein Zwischenfazit beendet diesen Teil.

# Kapitel 1

## Motivation und Überblick

*Irren ist menschlich!*  
*[überliefert]*

Am 4. Juni 1996 endete der Jungfernflug der Ariane 5 Rakete („Flug 501“) 40 Sekunden nach dem Start in einer Höhe von rund 3700 m mit der Selbstzerstörung der Rakete. Der offizielle Untersuchungsbericht beschreibt den Hergang folgendermaßen:

The launcher started to disintegrate at about  $H_0 + 39$  seconds because of high aerodynamic loads due to an angle of attack of more than 20 degrees that led to separation of the boosters from the main stage, in turn triggering the self-destruct system of the launcher. This angle of attack was caused by full nozzle deflections of the solid boosters and the Vulcain main engine. These nozzle deflections were commanded by the On-Board Computer (OBC) software on the basis of data transmitted by the active Inertial Reference System (SRI 2). Part of these data at that time did not contain proper flight data, but showed a diagnostic bit pattern of the computer of the SRI 2, which was interpreted as flight data. The reason why the active SRI 2 did not send correct altitude data was that the unit had declared a failure due to a software exception.

The OBC could not switch to the back-up SRI 1 because that unit had already ceased to function during the previous data cycle (72 milliseconds period) for the same reason as SRI 2.

The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected [...] the failure was due to a systematic software design error. [Lions97]

Tatsächlich hatte man Teile der Lageregelungssoftware des SRI unverändert aus dem Vorgängermodell — der Ariane 4 — übernommen, ohne ausreichend zu berücksichtigen, dass diese eine völlig andere Starttrajektorie aufweist. Entsprechende Versuchsläufe wurden aus Kostengründen nur mit eigens erstellten Software-Simulationen der SRI's durchgeführt.



Immer wieder melden die Medien solche und ähnliche Nachrichten über (mehr oder weniger katastrophale) Software-Fehler, wobei die Auswirkungen mit dem zunehmenden Grad der „Computerisierung“ unserer Alltagswelt für eine rasch wachsende Anzahl von Menschen spürbar werden. Sei es die Regelung des Herzschrittmachers, die Einkommenssteuererklärung mit dem Heimcomputer oder der Bordcomputer des Airbus A-320 — überall zieht unsichtbar im Hintergrund Software an den Fäden der Geschehnisse. Aufgrund ihrer zunehmend höheren Komplexität werden nicht geplante und unvorhersehbare Wechselwirkungen wahrscheinlicher. Nancy Leveson stellt fest:

We are building systems today — and using computers to control them — that have potential for large-scale destruction of life and environment. More than ever, software engineers and systems developers, as well as their managers, must understand the issues and develop the skills needed to anticipate and prevent accidents before they occur. Professionals should not require a catastrophe to happen before taking action. [Leveson95]

Versuchen wir, die Ursachen von Software-Fehlern wie dem oben Illustrierten aufzuspüren, so sehen wir uns verschiedensten Argumenten gegenübergestellt. Nach [Nuseibeh97] pointieren wir aus dem Blickwinkel verschiedener Projektbeteiligter mögliche Erklärungen:

**Der Programmierer** spricht von einem klaren Programmierfehler. Im Falle der Ariane-5 wurde eine Ausnahme in der SRI-Software aufgrund einer Konvertierung einer 64-bit Real-Zahl in eine 16-bit Integer-Zahl nicht behandelt. Eine bessere Programmierpraxis hätte diesen Fehler klar verhindert.

**Der Entwurfsspezialist** spricht von einem klaren Entwurfsfehler. Die Entwurfsspezifikation berücksichtigte lediglich Hardware-Fehler, so dass der Software-Fehler nicht behandelt und infolgedessen die SRI-2 Einheit stillgelegt wurden. Ebenso erging es der (mit der gleichen Software ausgestatteten) Ersatz-Einheit SRI-1. Ein besserer Systementwurf hätte diesen Fehler klar verhindert.

**Der Anforderungsermittler** spricht von einem klaren Fehler bei der Anforderungsermittlung. Die Anforderungen der Ariane-5 weichen erheblich von denen des Vorgängermodells ab. Wären die geänderten Anforderungen bis zur Implementation nachverfolgbar gewesen, so hätte man erkennen können, dass bei der Ariane-5 die Funktionalität der SRI nach dem Start überflüssig ist. Eine verfolgbare Anforderungsspezifikation hätte diesen Fehler klar verhindert.

**Der Tester** spricht von einem klaren Fehler im Testprozess. So gab es z.B. keinen closed-loop Test der SRI-Einheiten gegen die geänderte Starttrajektorie der Ariane-5. Ein besserer Test hätte diesen Fehler klar verhindert.

**Der Projektmanager** spricht von einem klaren Managementfehler. So wurden z.B. zu den Reviews keine externen Spezialisten herangezogen, und die Dokumentation war inkonsistent. Ein besseres Projektmanagement hätte diesen Fehler klar verhindert.

Letztendlich ist jedes dieser Argumente richtig — ohne adäquate Qualitätssicherung entsteht „fehlerfreie“ Software nur aus „fehlerfreien“ Tätigkeiten. Da der Mensch jedoch im Allgemeinen dazu neigt, Fehler zu machen, ist die Qualitätssicherung bei allen Tätigkeiten der Softwareentwicklung notwendig.

Nach diesem Beispiel eines Software-Fehlers und seiner Auswirkungen präzisieren wir einige Begriffe und betrachten die geschichtliche Entwicklung der Qualitätssicherung in der Softwareentwicklung. Im letzten Abschnitt geben wir dann einen Überblick über die vorliegende Arbeit.

## 1.1 Begriffe und Gebiete der Qualitätssicherung

Neben dem Standard-Vokabular des Software-Engineering (vgl. [IEEE610.90]) verwenden wir in dieser Arbeit spezielle Begriffe<sup>1</sup> der Qualitätssicherung, die wir im Folgenden kurz erläutern. Hierbei lehnen wir uns an [PagSix94] und [Riedemann97] an.

*Prüfgegenstände* sind die zugrundegelegten Betrachtungseinheiten. Zu den Prüfgegenständen bei der Softwareentwicklung gehören z.B. die Anforderungsspezifikation, die Software-spezifikation(en), Diagramme, Quellcode, die ausführbare Anwendung und die Testdokumente selbst.

Eine wichtige Unterscheidung erfolgt zwischen einem Fehler und seiner Ursache. Ein *Fehler* ist das Abweichen eines berechneten, beobachteten oder gemessenen Werts oder eines Zustands von dem entsprechenden spezifizierten richtigen Wert bzw. Zustand (z.B. die Ausgabe eines falschen Funktionswerts). Die *Fehlerursache* kann dabei unbekannt sein. Außerdem können mehrere Fehler dieselbe Ursache haben.

Weiter differenzieren wir zwischen einem Defekt und einem Ausfall:

- ❑ Bei einem *Defekt* weicht ein Merkmal des Prüfgegenstands von seiner Spezifikation ab. Defekte können, müssen aber nicht als Fehlerursache die Funktionstüchtigkeit des Prüfgegenstands beeinträchtigen (z.B. fehlerhaft realisierte Anforderungen, aber auch mangelhafte Lesbarkeit, Verletzung von Dokumentationsrichtlinien).
- ❑ Ein *Ausfall* bedeutet, dass der Prüfgegenstand aufgrund eines Defekts nicht mehr in der Lage ist, die geforderte Funktion auszuführen (z.B. „Absturz“, Livelock- oder Deadlock-Zustand).

Die Tätigkeiten Prüfen, Debuggen, Verifikation und Validierung werden folgendermaßen abgegrenzt:

- ❑ *Prüfen* ist das gezielte Suchen nach Fehlern und Defekten des Prüfgegenstands. Geprüft werden sämtliche Artefakte, die bei der Softwareentwicklung entstehen. Auch


---

<sup>1</sup> Begriffe der Objektorientierung werden von uns vorausgesetzt (s. z.B. [BHJ+98]).

Dokumente, die Prüfungen unterworfen werden, gehören hierzu. Prüfen ist eine Kontrollfunktion im Software-Entwicklungsprozess. Sie umfasst nicht die Fehlerkorrektur. Daneben wird validierendes und falsifizierendes Prüfen<sup>1</sup> unterschieden. Ersteres hat die Frage „Was kann der Prüfgegenstand?“, letzteres die Frage „Was kann der Prüfgegenstand nicht?“ als Ausgangspunkt.

- ❑ *Debuggen* ist das Beheben von Fehlern. Zu einem festgestellten Fehler wird zunächst die Fehlerursache aufgespürt und anschließend korrigiert.
- ❑ *Validierung* bezeichnet die Überprüfung einer allgemeinen Aussage mit positivem Ausgang. So wird z.B. die Behauptung, dass eine Anwendung bei Eingabe bestimmter Werte ein spezifiziertes Ergebnis liefert, durch Ausführung der Anwendung validiert.
- ❑ Unter der *Verifikation* versteht man die Prüfung eines Artefakts gegen seine Spezifikation. Wir fassen den Begriff der Verifikation in dieser Arbeit im Sinne der angelsächsischen Literatur („bauen wir das System richtig“, [Boehm84]) wesentlich weiter als die meisten deutschsprachigen Autoren, die unter der Verifikation eher mathematische Techniken zum Korrektheitsbeweis (z.B. des Quellcodes der Anwendung) auf der Grundlage einer formalen Semantik der verwendeten (Programmiersprache) gegen eine formale Spezifikation verstehen (s. auch Teil IV).

Die *Qualitätssicherung* ist die Gesamtheit aller die Softwareentwicklung begleitenden Tätigkeiten zur Vermeidung von Defekten, zu denen außer Prüfen, Debuggen, Validierung und Verifikation auch die Planung und Kontrolle dieser Tätigkeiten gehören.

Die Qualitätssicherung bzw. das „Testen“ im weitesten Sinne kann in die Gebiete Theorie, Verfahren, Werkzeuge, Evaluierung und Management unterteilt werden ([Binder96a]). Innerhalb dieser Gebiete nehmen wir die in Abb. 1.1 gezeigte weitere Unterteilung vor (vgl. [Beizer90][PagSix94][Binder96a][Riedemann97]). Die in dieser Arbeit angesprochenen Gebiete der Qualitätssicherung (QS) sind in Abb. 1.1 mit  gekennzeichnet.

- ❑ Die *Testtheorie* als Grundlage jeder Testmethode benötigt zunächst Beobachtungen zur Testbarkeit der betrachteten Klasse von Prüfgegenständen. Basierend auf diesen Beobachtungen, bisherigen Erfahrungen, „gesundem Misstrauen“ und Experimenten werden geeignete *Fehlerhypothesen* aufgestellt. Jede Fehlerhypothese stellt eine Annahme bezüglich der Fehlerwahrscheinlichkeit einer Klasse von Prüfgegenständen relativ zu einem bestimmten Aspekt dar.
- ❑ Aufbauend auf Fehlerhypothesen werden *Testmodelle* z.B. anhand der Struktur des Prüfgegenstands (z.B. Kontroll- oder Datenfluss, strukturelles „white-box“ Vorgehen) oder seiner Spezifikation (funktionales „black-box“ Vorgehen) entwickelt, aus denen mit entsprechenden Testverfahren konkrete Testfälle für die zu testenden Prüf-

---

<sup>1</sup> Falsifizierendes Prüfen entspricht der modernen Interpretation des Begriffs *Testen* (vgl. [Myers79][Beizer90]).

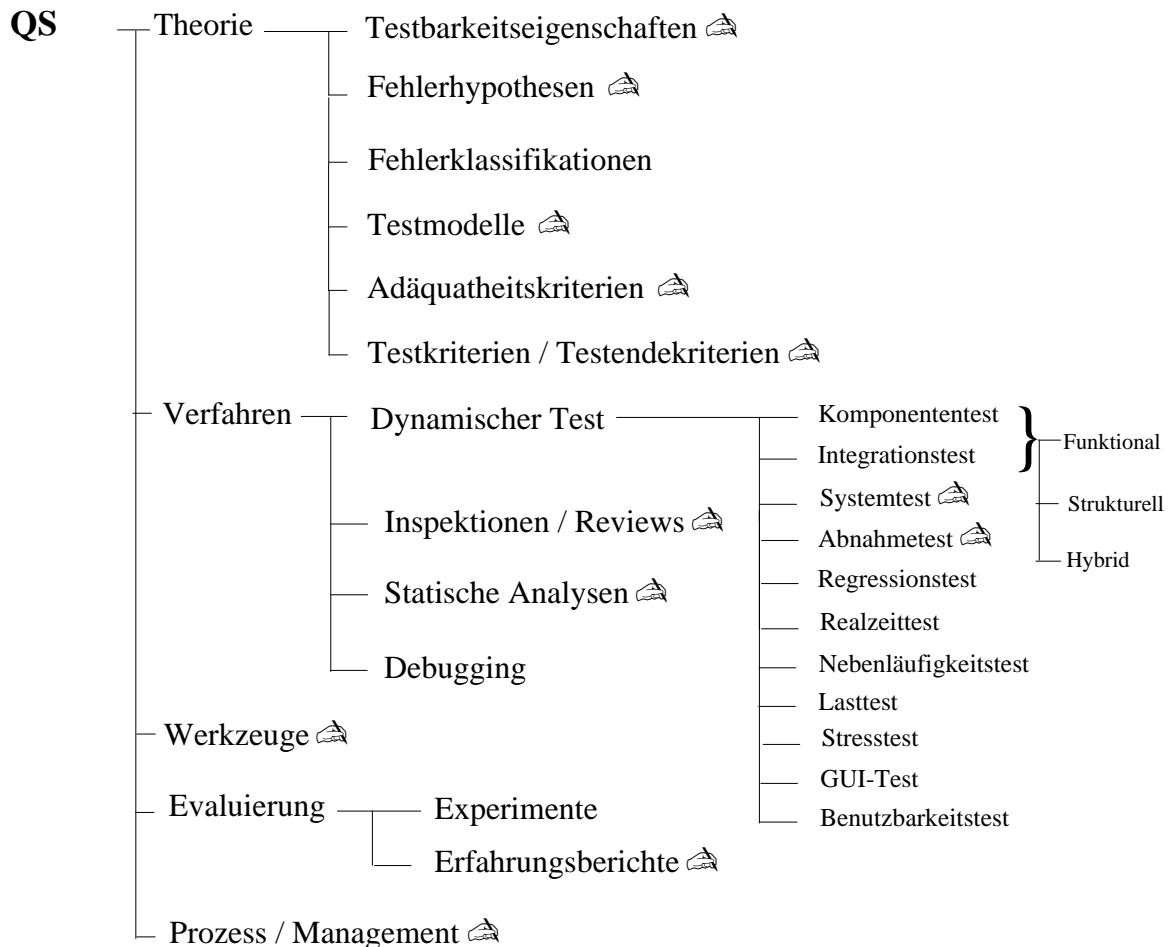


Abb. 1.1 Gebiete der Qualitätssicherung

gegenstände abgeleitet werden. Ein *Adäquatheitskriterium* gibt für eine Menge von Testfällen bzw. Testdaten an, ob sie den Prüfgegenstand bzgl. einer Fehlerhypothese oder eines Testmodells „ausreichend“ testet (vgl. [Weyucker86]). Ein algorithmisch auswertbares Adäquatheitskriterium, welches z.B. in einer prozentualen Angabe mündet, bezeichnen wir auch als *Testkriterium*. So ist z.B. die „Anweisungsüberdeckung“ ein Testkriterium zum Testmodell „Kontrollflussgraph“ bzw. dem Testverfahren „Kontrollflussbasiertes Testen“ (vgl. [Myers79]). Ein *Testendekriterium* oder *Abbruchkriterium* ist eine Menge von Testkriterien sowie ggf. weiteren Kenngrößen zusammen mit entsprechenden vorgegebenen Minimalanforderungen wie z.B. „95%-ige Anweisungsüberdeckung“, welche die Beantwortung der Frage nach einer „ausreichenden“ Prüfung dient. Unter einem *Testfall* versteht man eine z.B. aus der Spezifikation (*black-box Test*) oder dem Quellcode der Anwendung (*white-box Test*) abgeleitete Menge von Eingabedaten inklusive der zugehörigen Ergebnisse, welche die Überprüfung einer bestimmten Eigenschaft des Prüfgegenstands ermöglicht. *Test-*

*daten* bilden die Teilmenge, mit welcher der Prüfgegenstand tatsächlich getestet wird. Ein *Testorakel* ermittelt (automatisch) das erwartete Ergebnis eines Testfalls aus der Spezifikation, gegen die geprüft wird.

- Die Klassifikation von *Testverfahren* erfolgt nach der Granularität des Prüfgegenstands, der Sichtweise auf den Prüfgegenstand und der ggf. mit einem validierenden Test nachzuweisenden Eigenschaft. Man unterscheidet den *Komponententest* vom *Integrationstest* und *Systemtest* sowie zusätzlich den unter Einbeziehung der Endbenutzer durchgeführten *Abnahmetest* und den nach Änderungen am Prüfgegenstand ggf. notwendigen *Regressionstest*. Hinsichtlich der Sichtweise auf den Prüfgegenstand gibt es funktionale und strukturelle Verfahren, wobei auch Mischformen bzw. hybride Tests möglich sind. Der *Realzeittest* weist temporale Eigenschaften des Prüfgegenstands wie z.B. die Reaktionszeit auf ein Ereignis oder die Ausführungsdauer einer bestimmten Operation nach. Der *Nebenläufigkeitstest* betrachtet die Ausführung des Prüfgegenstands zusammen mit weiteren konkurrenten Anwendungen. Der *Lasttest* untersucht den Prüfgegenstand hinsichtlich extremer Quantitäten bestimmter Daten, der *Stresstest* hinsichtlich extremer Quantitäten bestimmter Ereignisse. Mit dem *GUI-Test* wird die Abbildung innerer Zustände des Prüfgegenstands auf die Benutzungsschnittstelle, mit dem *Benutzbarkeitstest* die ergonomische Adäquatheit des Prüfgegenstands geprüft.
- *Testwerkzeuge* unterstützen die Tester bezüglich einzelner oder mehrerer Testverfahren sowie bei den administrativen Tätigkeiten.
- *Evaluationen* bewerten bekannte oder neue Methoden und Verfahren im praktischen Einsatz oder versuchen, bestimmte Hypothesen über dieselben mit kontrollierten Experimenten bzw. empirischen Methoden zu bestätigen oder zurückzuweisen.
- Letztendlich gehören der *Testprozess* und das *Testmanagement* zu den administrativen Punkten der Qualitätssicherung.

Wir skizzieren nun noch kurz die historische Entwicklung der Software-Qualitätssicherung sowie den diesbezüglichen Stand in der industriellen Praxis.

## 1.2 Geschichte

Gelperin und Hetzel unterscheiden von den Anfängen der Programmierung bis 1988 fünf „Epochen“ der Software-Qualitätssicherung ([GelHet88]):

...-1956 In der *Debugging-Epoche* bereiteten Hardware-Fehler weitaus größere Probleme als Fehler in der Software. Im Wesentlichen bestanden die qualitätssichernden Tätigkeiten im konzentrierten „Durchlesen“ der Programme durch den Entwickler und in einer darauffolgenden Erprobungsphase. Debugging und Prüfen waren nicht voneinander unterschieden.

**1957-1978** In der *Demonstrations-Epoche* wurden Debugging und Prüfen erstmals als unterschiedliche Tätigkeiten herausgestellt. Prüfungen wurden als „erfolgreich“ bezeichnet, wenn sie die Funktionstüchtigkeit des Programms nachwiesen.

**1979-1982** Eingeleitet durch das richtungsweisende Buch von Glenford Myers rückte in der *Destruktions-Epoche* das Aufzeigen von Fehlern in den Mittelpunkt der Prüf-Tätigkeiten ([Myers79]). Da der Entwickler die Fehlerfreiheit seines Programms anstrebt, sollten speziell geschulte Tester das Programm unbarmherzig unter die Lupe nehmen. Tests wurden als „erfolgreich“ bezeichnet, wenn sie die Funktionsuntüchtigkeit des Programms nachwiesen.

**1983-1987** In der *Epoche der Evaluierung* wurde die Qualitätssicherung erstmals auf alle Tätigkeiten der Softwareentwicklung ausgedehnt. Jeder Entwicklungstätigkeit wurden entsprechende qualitätssichernde Tätigkeiten zugeordnet. Qualitätssicherung wurde jedoch immer noch eher als *Qualitätskontrolle* angesehen, d.h., Produkte wurden *nach* ihrer Fertigstellung geprüft.

**1988-1991** Während der *Epoche der Fehlervermeidung* versuchte man durch einen Mix an konstruktiven und analytischen Maßnahmen, Fehler möglichst schon an ihrem Entstehungsort zu verhindern, d.h., Produkte wurden *während* ihrer Fertigstellung geprüft.

Bis dahin war das Ziel der Qualitätssicherung immer noch die Freiheit des Produkts von Defekten. Anfang der neunziger Jahre trat die produktzentrierte Sicht zu Gunsten einer prozesszentrierten Sicht in den Hintergrund. Normen wie die ISO 9000 ([ISO9001-3.92]) und standardisierte, vergleichbare Prozess-Evaluierungen wie im Prozess-Reifegradmodell (*capability maturity model*, CMM [Paulk91]) beherrschten die Szene. Während die ISO-9000 sich auf die Dokumentation innerhalb des Qualitätssicherungssystems konzentriert, zielt das CMM auf Maßnahmen zur Prozessverbesserung (vgl. [Paulk95]). Zur Zeit werden die produktzentrierte und die prozesszentrierte Sicht in umfassenden, qualitätszentrierten Software-Entwicklungsprozessen vereinigt (vgl. [JBR99]).

Zum momentanen Stand der Praxis in der Software-Qualitätssicherung bemerken Cusumano und Selby:

Weder wir noch Microsoft glauben, dass mittlerweile alle Probleme gelöst seien und kein Produkt mehr verspätet oder fehlerfrei ausgeliefert würde. Dies kann ohnehin kein Software-Hersteller von sich behaupten. [CusSel96]

Müller und Wiegmann von der Universität Köln untersuchten 1998 den „Stand der Praxis“ der Prüf- und Testprozesse in der Softwareentwicklung in Deutschland ([MülWie98]). Bei der Auswertung ihrer Fragebögen trafen sie auf eine Reihe alarmierender Fakten:

- ☐ Eine spezialisierte Testgruppe fehlt in den meisten Firmen.

- ☐ Hauptsächlich wird in den „späten“ Phasen der Softwareentwicklung geprüft; die wichtigen „frühen“ Phasen wie Anforderungsermittlung und Entwurf werden in Bezug auf die Qualitätssicherung vernachlässigt.
- ☐ 12 von 74 Softwarehäusern testen auch in der Realisierungsphase nicht oder nur teilweise.
- ☐ Nur bei wenigen Firmen erfolgt eine detaillierte Zeit- und Ressourcenplanung. Daher kommt es zu Engpässen, durch die nur ein Teil der vorgesehenen Prüfungen durchgeführt werden können.
- ☐ Nur ein Drittel der Firmen unterstützen Schulungen im Bereich der Qualitätssicherung oder stellen Mitarbeiter der Fachabteilungen für die Tests frei.
- ☐ Bei über zwei Dritteln der Firmen wird der Soll-Ist Vergleich bei den Tests nicht regelmäßig durchgeführt.
- ☐ Bezüglich formaler Kriterien zur Messung des Testfortschritts wird die „Funktionsabdeckung“ am häufigsten verwendet.

Insgesamt stellen Müller und Wiegmann fest, dass in der Qualitätssicherung bei der Softwareentwicklung die Schere zwischen Forschung und Praxis weit auseinanderklafft. Eine wichtige Aufgabe ist es also, an industrielle Erfordernisse angepasste Methoden und Verfahren zur Qualitätssicherung bei der Softwareentwicklung anzugeben.

Hinzu kommen die überproportional mit der Zeit zwischen Fehlerentstehung und -entdeckung steigenden Kosten der Fehlerentdeckung und -behebung. Je später ein Fehler entdeckt wird, desto höher ist der Aufwand für seine Behebung, da eine rasch wachsende Zahl von Entwicklungsprodukten betroffen ist (vgl. Tab. 2.1, S. Seite 18). Es treten also Methoden und Verfahren zur Qualitätssicherung für die „frühen“ Phasen wie z.B. die Anforderungsermittlung in den Vordergrund.

### 1.3 Überblick über die Arbeit

In Kapitel 2 skizzieren wir die Tätigkeiten bei der Entwicklung objektorientierter Software und stellen damit den Bezugsrahmen für die Arbeit her. Hierbei konzentrieren wir uns auf die frühen Tätigkeiten, insbesondere auf die Anforderungsermittlung. In Kapitel 3 wenden wir uns der Qualitätssicherung in der objektorientierten Softwareentwicklung zu. Da die Komplexität hier in den Interaktionen zwischen Objekten liegt, erweisen sich strukturelle (white-box) Klassentests als wenig aussagekräftig. Aufgrund des z.T. sehr hohen Aufwandes ist ihr Einsatz nicht effektiv. Für funktionale (black-box) Tests der Anwendung gegen die Anforderungsspezifikation sind im Falle objektorientierter Anforderungsspezifikationen neue Verfahren notwendig. In Kapitel 4 beleuchten wir daher als den diesbezüglichen „State of the Art“ die objektorientierte Anforderungsspezifikation mit Use Cases und Klassenmodellen. Wir listen anhand eines Fallbeispiels Schwachpunkte auf, welche die Qualitätssicherung bei

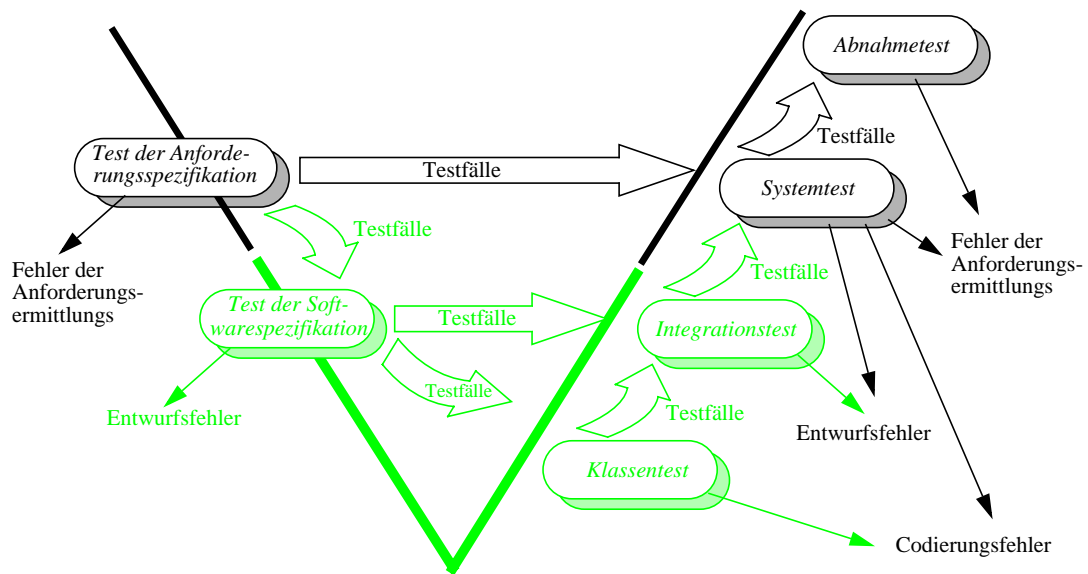


Abb. 1.2 Fokus der Arbeit (Grafik nach [PagSix94])

der Anforderungsermittlung und den Test der Anwendung gegen die Anforderungsspezifikation betreffen und die wir mit dieser Arbeit beheben wollen. In diesem Zusammenhang stellen wir auch relevante Arbeiten anderer Autoren zu diesen Themen vor.

Nach diesen „motivierenden“ Worten geben wir nun einen Überblick in Form eines „Wegweisers“ durch die weiteren Teile der vorliegenden Arbeit. Im Mittelpunkt steht SCORES (Systematic Coupling of Requirements Specifications), unsere qualitätszentrierte Methode zur Anforderungsermittlung. Die Arbeit gliedert sich in die in Abb. 1.2 hervorgehobenen drei jeweils aufeinander aufbauenden Themenbereiche

- ❑ Präzisierung und Verfeinerung der Anforderungsspezifikation mit Use Cases,
- ❑ Qualitätssicherung bei der Anforderungsermittlung und
- ❑ Ableitung von Testfällen aus der Anforderungsspezifikation.

Im zweiten Teil stellen wir die Konkretisierung und Spezifikation von Anforderungen mit SCORES vor. In Anbetracht der aufgedeckten Schwachpunkte der „klassischen“ Use Case Analyse nach Jacobson führen wir Use Case Schrittgraphen zur Präzisierung und Verfeinerung von Use Cases als zentrales Modellierungskonzept von SCORES ein. Use Case Schrittgraphen verschmelzen funktionale Aspekte der Anforderungen mit ablaufbezogenen Aspekten, wobei sich Szenarien nahtlos in die erhaltene kombinierte Sicht einfügen. Diese Sicht wird darüber hinaus mit den strukturellen Aspekten, also dem Klassenmodell gekoppelt. In Kapitel 8 geben wir methodische Hinweise zum Vorgehen bei der Anforderungsermittlung mit SCORES.



In Teil III befassen wir uns mit der Qualitätssicherung bei der Anforderungsermittlung mit SCORES. Wir zeigen, wie Analytiker und Tester mit Benutzern und Domänen-Experten zunächst Use Cases (bzw. Schrittgraphen) und wichtige Teile des Klassenmodells validieren. Zusätzlich geben wir eine Methode zur rigorosen Prüfung bzw. „Verifikation“ des Klassenmodells gegen Use Case Schrittgraphen und umgekehrt an. Granularität und Semantik der SCORES-Anforderungsspezifikation erlauben die Formulierung von Testkriterien sowohl zur Bestimmung des Test-Endes als auch zur Generierung von Testfällen. Da sogenannte SCORES grey-box Testfälle im Systemtest interne Interaktionen prüfen, zeigen wir in Kapitel 13, wie die Anforderungsspezifikation und die bei ihrer Prüfung protokollierte Information über den Entwurf in die Implementation hin (nach-) verfolgt wird.

Nach der Implementation wird die Anwendung gegen die Anforderungsspezifikation getestet. Der (dynamische) Test der Anwendung gegen die SCORES-Anforderungsspezifikation ist Inhalt von Teil IV. Nach einem Überblick leiten wir als angestrebten „Mehrwert“ der Anforderungsermittlung mit SCORES in Kapitel 15 zunächst reine black-box Testfälle ab. Da white-box Klassen-Tests für objektorientierte Software nicht effektiv sind und black-box Testverfahren die fehlerträchtigen komplexen internen Interaktionen objektorientierter Anwendungen nicht adäquat prüfen, erweitern wir in Kapitel 16 die black-box Testfälle zu sogenannten SCORES grey-box Testfällen. Wir prüfen hierbei die internen Interaktionen der Anwendung gegen die (bei der Verifikation protokollierte) systeminterne Information der Anforderungsspezifikation, die wir dafür geeignet im sogenannten Klassen-Botschaftsdiagramm zusammenfassen. In Kapitel 17 gehen wir auf die Generierung von Objektkonstellationen als Testdaten ein. Wir zeigen, dass die Generierung solcher Objektkonstellationen ein NP-hartes Problem ist und geben Hinweise auf entsprechende Heuristiken. Im letzten Kapitel von Teil IV skizzieren wir die Testausführung und -auswertung.

Abschließend skizzieren wir in Teil V die im Rahmen der Arbeit implementierte Werkzeugunterstützung und schließen in Teil VI mit einer Bewertung der erzielten Ergebnisse und einem Ausblick auf laufende und zukünftige Arbeiten.

# Kapitel 2

## Entwicklungstätigkeiten

In diesem Kapitel skizzieren wir typische Tätigkeiten in der objektorientierten Softwareentwicklung. Wir orientieren uns an dem aus *Objectory* ([JCJ+92]) hervorgegangenen *Rational Unified Process* (RUP, [Kruchten99][JBR99]), der die Vorteile herkömmlicher Vorgehensmodelle wie z.B. des (iterierten) Phasenmodells oder der evolutionären Softwareentwicklung (vgl. [PagSix94]) in sich zu vereinigen sucht. Wir zeigen dann, wie Geschäftsprozesse und Workflow-Modelle den Kontext für die zu entwickelnden Anwendungen bilden und gehen auf die Anforderungsermittlung, den Entwurf und die Implementation sowie (kurz) auf die Qualitätssicherung ein. Letztere ist Gegenstand des nächsten Kapitels.

Kruchten und Jacobson beschreiben den RUP anhand der zwei Dimensionen Zeit und Tätigkeiten (Abb. 2.1):

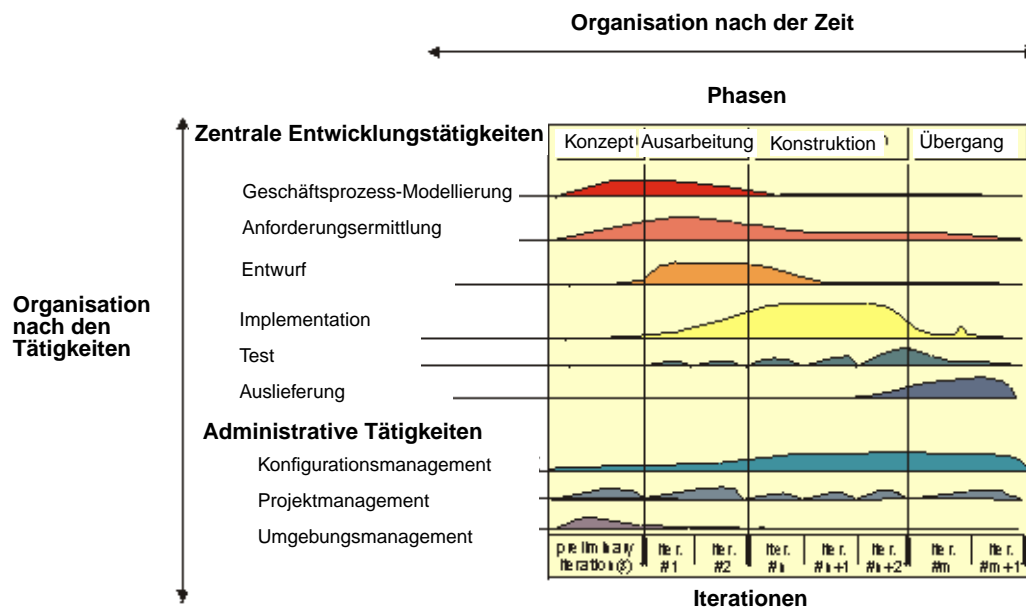


Abb. 2.1 Der Rational Unified Process (nach [JBR99])

- Die horizontale Achse repräsentiert die *Zeit* und zeigt die Dynamik der einzelnen Tätigkeiten, also das „Wann“. Die Zeitachse ist in Zyklen, Phasen und Iterationen eingeteilt. Der Lebenszyklus objektorientierter Software wird im Allgemeinen für jede Generation bzw. Version einer Anwendung durchlaufen. Im *RUP* ist der Lebenszyklus in die vier aufeinanderfolgende Phasen Konzeptualisierung, Entwurf, Konstruktion und Übergang unterteilt.
- Auf der vertikalen Achse werden die einzelnen Aspekte der Tätigkeiten, also das „Wie“, „Was“ und „Wer“ abgetragen. Hier sehen wir auf der obersten Ebene die Tätigkeiten. Zu jeder Tätigkeit geben wir die erzeugten bzw. bearbeiteten Entwicklungsprodukte, die jeweiligen Aktivitätsträger („Worker“) und die „Workflows“ der einzelnen Tätigkeit an.

Jede Phase wird mit wohldefinierten Meilensteinen abgeschlossen, bei denen auf der Grundlage der bisherigen erreichten Ziele Entscheidungen für das weitere Vorgehen getroffen werden. Innerhalb der Phasen werden die Tätigkeiten iterativ ausgeführt, wobei jede Iteration in einer lauffähigen (internen) Teilversion der Anwendung mündet (Inkrement). Da jede Iteration bestimmte Teil-Funktionalitäten implementiert, die wiederum mit Use Cases beschrieben werden (vgl. Teil II), wird der Prozess als *Use Case-gesteuert, inkrementell und iterativ* charakterisiert (s. [JBR99]).

Zur präzisen Spezifikation der durch die verschiedenen Tätigkeiten erstellten Produkte bzw. Modelle wird eine hinreichend ausdrucksstarke Notation benötigt. Nachdem Anfang der neunziger Jahre über 30 verschiedene objektorientierte „Methoden“ mit unterschiedlichen Notationen existierten, vereinigten die bekannten „Methodologen“ Grady Booch und James Rumbaugh sowie ab Ende 1995 auch Ivar Jacobson die Notationen ihrer Methoden OOA/D, OMT und OOSE ([Booch94][RBP+91][JCJ+92]). Es entstand die Unified Modeling Language (UML), deren Version 1.1 Ende 1997 von der Object Modeling Group „standardisiert“ wurde ([OMG97]). Zur Zeit wird die UML bezüglich des Metamodells bzw. der Semantik sowie für die Modellierung von Geschäftsprozessen überarbeitet ([OMG-AT98][OMG99]).

Die Spezifikation der Anforderungen mit Use Cases bzw. Use Case Diagrammen ist Gegenstand der Kapitel 2 und 4. Die darüber hinaus in dieser Arbeit verwendeten Modellsichten bzw. Diagramme der UML sind in [OMG97] definiert. Bei den im Rahmen dieser Arbeit verwendeten deutschen Begriffen zur UML haben wir uns an die deutschsprachige Literatur zur UML orientiert (vgl. z.B. [Kahlbrandt98], s. auch [BHJ+98]). In manchen Fällen hat sich inzwischen die Situation eingestellt, dass im Deutschen sowohl der englische Originalbegriff als auch ein deutsches Synonym verwendet wird. Als Beispiel sei hier *use case* genannt, was man mit *Anwendungsfall* übersetzt oder — so wie in dieser Arbeit — auch in anderen deutschsprachigen Veröffentlichungen zu *Use Case* „assimiliert“. Auch zur Benutzung des Wortes *Akteur* als Übersetzung von *Actor* konnten wir uns nicht durchringen und verwenden als Kompromiss den Begriff *Aktor*.

## 2.1 Geschäftsprozesse und Anforderungsermittlung

In diesem Abschnitt betrachten wir zunächst kurz Geschäftsprozesse und Workflow-Modelle und zeigen Anknüpfungspunkte für die Anforderungsermittlung einzelner Anwendungen auf. Hierbei lehnen wir uns an die Arbeiten von Scheer, Jablonski und Sinz et al. sowie an das „Workflow Reference Model“ der WPMC (Workflow Management Coalition) an ([Scheer98a][Jablonski95][FerSin95][WPMC97]).

Eine für unsere Zwecke ausreichende Definition des Begriffes „Geschäftsprozess“ (*Business Process*) gibt Scheer:

Allgemein ist ein Geschäftsprozess eine zusammengehörige Abfolge von Unternehmensverrichtungen zum Zweck einer Leistungserstellung. Ausgang und Ergebnis des Geschäftsprozesses ist eine Leistung, die von einem internen oder externen „Kunden“ angefordert und abgenommen wird. [Scheer98a]

Geschäftsprozesse sind somit direkter Gegenstand betriebswirtschaftlicher Betrachtungen, für die Ziele wie z.B. „Optimierung des zeitlichen Aufwandes“ definiert werden und die der Kostenrechnung unterliegen. Wichtig ist das tiefe Verständnis der Anwendungssituation, was insbesondere eine große Anschaulichkeit der Modelle für Geschäftsprozesse bedingt.

Workflow-Modelle dienen als Grundlage für die Automatisierung von Geschäftsprozessen. Im „Workflow Reference Model“ der WPMC finden wir die folgende Definition:

**Workflow.** The computerized facilitation or automation of a business process, in whole or part. [WPMC97]

Workflow-Modelle müssen aufgrund ihrer Implementationsnähe präzise sein, also jedes Detail eines Geschäftsprozesses eindeutig beschreiben und eine konsistente Gesamtbeschreibung liefern. Oft wird der Begriff Business Process Reengineering synonym für die Analyse und Spezifikation von Workflows gebraucht (vgl. [HamCha94]). Integrierte Systeme sowohl zur Spezifikation als auch zur Ausführung von Workflows werden Workflow Management Systeme genannt:

**Workflow Management System.** A system that completely defines, manages, and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic. [WPMC97]

Workflow-Modelle spiegeln verschiedene sachliche Aspekte wider (nach [Jablonski95]):

- ☐ der *organisatorische Aspekt* zeigt, *wer* etwas ausführt bzw. ausführen kann (und darf);
- ☐ der *funktionale Aspekt* spezifiziert, *was* ausgeführt wird;
- ☐ der *operationale Aspekt* gibt an, *wie* etwas ausgeführt wird (z.B. computergestützt durch spezielle Anwendungen oder manuell);

- ❑ mit dem *ablauf- oder verhaltensbezogenen Aspekt* wird beschrieben, *wann* etwas ausgeführt wird, also der „Kontrollfluss“ im Workflow spezifiziert;
- ❑ die Daten, also das *Womit*, werden durch den *informationsbezogenen Aspekt* beschrieben;
- ❑ der *kausale Aspekt* letztendlich gibt an, *warum* bzw. aufgrund welcher Intra-Workflow-Abhängigkeiten etwas ausgeführt wird.

Weiterhin enthalten Workflow-Modelle auch nichtsachliche Aspekte wie z.B. den *historischen Aspekt* zur Protokollierung von Tätigkeiten und den *transaktionalen Aspekt* zur Koordination von (nebenläufigen) Tätigkeiten.

Der die Tätigkeiten in einem Unternehmen beschreibende Geschäftsprozess wird also mit einem Workflow-Modell in einem WFMS spezifiziert, welches den Geschäftsprozess dann durch die koordinierte Ausführung verschiedener Anwendungen automatisiert. Diesen Zu-

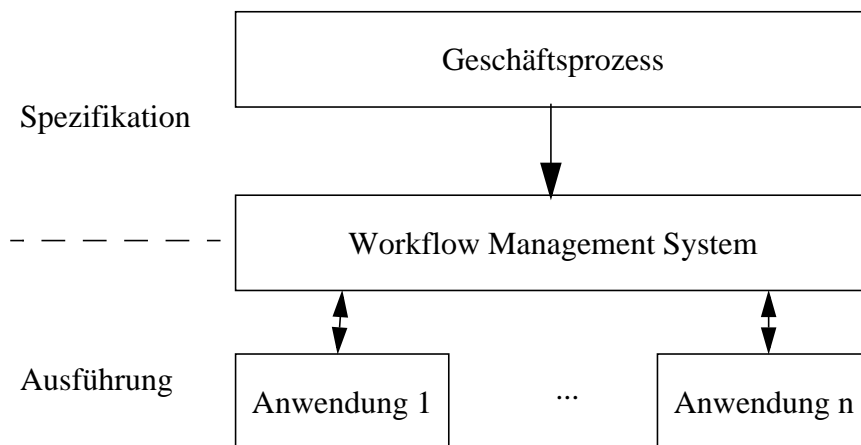


Abb. 2.2 Geschäftsprozess, WFMS und Anwendungen

sammenhang zeigt Abb. 2.2.

Engen wir nun den Blickwinkel auf eine bestimmte, innerhalb eines Workflows auszuführende Anwendung ein, so können wir bestimmte Sichten bzw. Aspekte zunächst als „vorgegebene“ und relativ stabile Randbedingungen ansehen. Einige Aspekte wie z.B. die Organisation und die Ziele des Geschäftsprozesses ergeben hervorragende Einstiegspunkte in die Anforderungsermittlung für eine bestimmte Anwendung: Anhand der Organisationsstruktur ermitteln wir, wer mit der Anwendung welche Ziele erreichen soll. Die Informations- bzw. Datensicht gibt die Struktur der Daten an, welche die Anwendung lesen bzw. erzeugen soll. Dies kann z.B. in Form von Dateiformaten oder Datenbankschemata oder aber (bei manueller Dateneingabe) umgangssprachlich bzw. mit Formularen erfolgen.

Fassen wir zusammen: zu Beginn einer Entwicklung erfolgt eine Ist-Analyse des Geschäftsfeldes (*business analysis*). Danach wird der angestrebte Geschäftsprozess (*envisioned busi-*

ness process) als Soll-Zustand modelliert (*business reengineering*, vgl. [HamCha94]). Für die einzelnen Aufgaben des Geschäftsprozesses werden der Grad (automatisch, rechnergestützt, manuell, ...) sowie die Art (Standard- oder Individualsoftware, ...) der DV-Unterstützung festgelegt und ggf. im Workflow-Managementsystem modelliert. Letztendlich werden die den einzelnen Aufgaben entsprechenden Anwendungen angepasst bzw. neu entwickelt. Wir kommen somit zur Anforderungsermittlung für eine Anwendung.

## Anforderungsermittlung

In der *Anforderungsermittlung* (*requirements engineering*, RE) werden informelle Ideen hinsichtlich der Ziele, Funktionen und Beschränkungen eines Softwareprodukts zu einer präzisen Spezifikation ausgearbeitet und in der Anforderungsspezifikation sorgfältig dokumentiert. Da die Zufriedenheit der Benutzer mit dem System in zunehmendem Maße als ausschlaggebendes Kriterium zur Feststellung der Systemqualität herangezogen wird, gewinnen das richtige Verständnis für die Anforderungen und Bedürfnisse des Benutzers, deren Spezifikation, Dokumentation und Validierung an Bedeutung.

Der IEEE Standard 610, „Glossary of Software Engineering Terminology“, definiert die *Anforderungsermittlung* (*requirements analysis*) als:

**Requirements analysis.** (1) The process of studying user needs to arrive at a definition of system, hardware or software requirements. (2) The process of studying and refining system, hardware or software requirements. [IEEE610.90]

Zum Begriff der *Anforderung* (*requirement*) lesen wir in der IEEE 610:

**Requirement.** (1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. (3) A documented representation of a condition or capability as in (1) or (2). [IEEE610.90]

Nach dem IEEE Standard 830, „Guide to Software Requirements Specifications“ werden bei der Spezifikation von Anforderungen solche, die das externe beobachtbare Verhalten betreffen (*functional<sup>1</sup> requirements*), von technischen Anforderungen und Qualitätsanforderungen (*non-functional requirements*) unterschieden ([IEEE830.93]):

### ☐ Externes Verhalten

- Funktionen oder Eigenschaften (Operationsfolgen, alternative Abläufe und Ausnahmebehandlungen, zu beachtende Geschäftsregeln, ...)
- Externe Schnittstellendefinitionen (Ein- Ausgabedaten, Datenbankanbindungen, ...)

---

<sup>1</sup> „Funktionale Anforderungen“ werden auch als „behavioral requirements“ [Davis90] oder „operative Anforderungen“ [PagSix94] bezeichnet.

## □ Nichtfunktionale Anforderungen

- Technische Anforderungen (Performanz, Last, Umgebungseinschränkungen, Entwurfseinschränkungen, ...)
- Qualitätsanforderungen (Nutzbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit, ...)

Die Anforderungsspezifikation (Software Requirements Specification, SRS) soll als Ergebnis der Anforderungsermittlung präzise Vorgaben für den Entwurf, die Implementation und den Test der zu entwickelnden Anwendung liefern. Die Anforderungsspezifikation soll nur beinhalten, *was* die Anwendung leisten soll, nicht aber, *wie* diese Leistung erbracht wird. Der IEEE Standard 830 fordert für die Anforderungsspezifikation zusätzlich:

A good software requirements specification is unambiguous, complete, verifiable, consistent, modifiable, traceable, and usable during the operation and maintenance phase. [IEEE830.93]

Die Anforderungsspezifikation als wichtiger Bestandteil des Vertrages zwischen Auftraggeber und Softwareentwickler muss für alle beteiligten Partner verständlich formuliert sein. Um aus dem Problembereich die relevanten Anforderungen zu extrahieren und präzise, verständlich und konsistent zu formulieren, ist eine enge Zusammenarbeit zwischen Analytikern und Benutzern erforderlich. Die Gründe hierfür sind vielfältig. Im Allgemeinen sind Analytiker zu Beginn der Anforderungsermittlung nicht ausreichend mit dem Problembereich und seiner spezifischen Terminologie vertraut, während Benutzer und Auftraggeber wenig Kenntnisse über die technisch orientierte Welt der elektronischen Datenverarbeitung und ihre Fachbegriffe besitzen. Aus den Wünschen und häufig vagen Vorstellungen des Auftraggebers entwickeln sich erst allmählich konkrete Vorgaben an die Anwendung, die wieder und wieder überarbeitet und gemeinsam validiert werden. Wenn die Vorstellungen des Auftraggebers und der Benutzer nicht richtig ermittelt und in der Anforderungsspezifikation entsprechend fixiert sind, kann keine Produktakzeptanz erreicht werden.

Darüber hinaus stellt die Anforderungsspezifikation den Ausgangspunkt für die weiteren Entwicklungstätigkeiten dar und hat somit auf alle Teilaspekte der Softwareentwicklung einen zum Teil erheblichen Einfluss (vgl. [Faulk97]). Die Anforderungsspezifikation muss als hierfür taugliche Basis ausreichend präzise sein.

Ohne eine sorgfältig durchgeführte Anforderungsermittlung werden fehlerhafte, fehlende, widersprüchliche oder mehrdeutige Anforderungen oftmals erst in der letzten Entwicklungsphase, d.h. beim System- oder Abnahmetest, entdeckt. Fehler, die das Außenverhalten des Produkts betreffen, werden beim praktischen Einsatz vom Benutzer rasch festgestellt und lassen ihn an der Eignung des Produkts zweifeln.

Tab. 2.1 gibt einen Überblick der relativen Kostenfaktoren zur Behebung eines Fehlers der Anforderungsspezifikation, der bei einer bestimmten Tätigkeit der Softwareentwicklung entdeckt wird. Die Kosten steigen überproportional, da mit fortschreitender Entwicklung nach der Entdeckung des Fehlers immer mehr auf dem fehlerhaften Teil der Anforderungsspezifi-

<i>Tätigkeit</i>	<i>relativer Kostenfaktor</i>
Anforderungsermittlung	1-2
Entwurf	5
Implementation	10
System- und Abnahmetest	20-50
Einsatz und Wartung	200

Tab. 2.1 Relative Kostenfaktoren zur Korrektur eines RE-Fehlers (nach [Faulk97])

kation basierende Entscheidungen bzw. Entwicklungsprodukte betroffen sind und korrigiert werden müssen.

Pohl identifiziert innerhalb der Anforderungsermittlung das Extrahieren, Konkretisieren, Spezifizieren sowie Validieren und Verifizieren von Anforderungen als die wichtigsten Tätigkeiten ([Pohl96]). Abb. 2.3 zeigt die gegenseitige Beeinflussung dieser vier Tätigkeiten zusammen mit potentiell involvierten Personengruppen (*Stakeholder*).

Die Anforderungsermittlung startet mit der *Extraktion (Elicitation)* der Anforderungen, den Wünschen und Einschränkungen an das zu erstellende Softwareprodukt. Ziel dieser fortlauf-

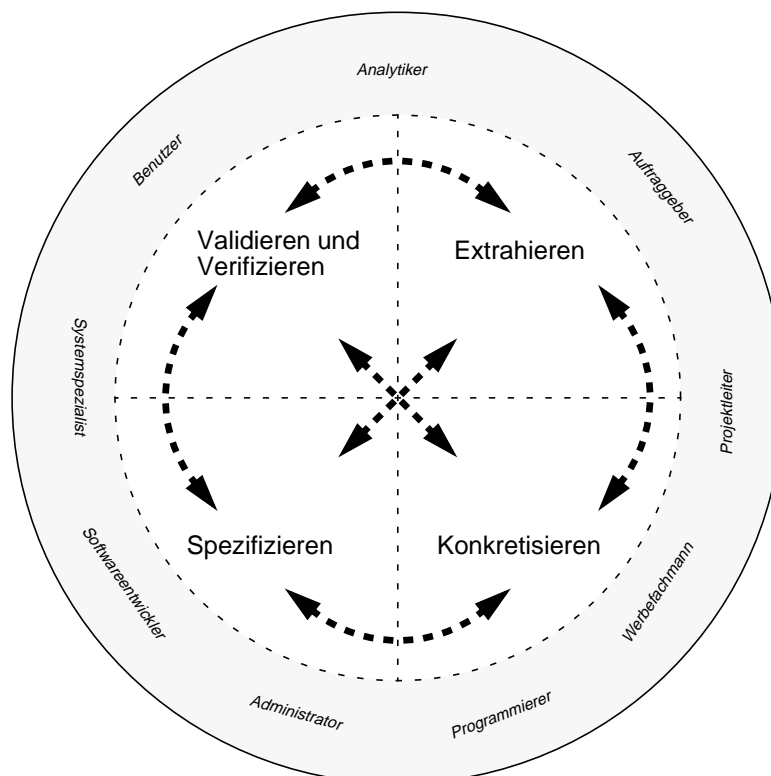


Abb. 2.3 Die vier Tätigkeiten der Anforderungsermittlung (aus: [Poh96])



fenden Tätigkeit ist es, die zum Teil verborgenen Anforderungen an das zu entwickelnde Softwareprodukt explizit zu machen, und zwar in einer Form, die allen involvierten Personen verständlich ist. Die *Konkretisierung (Negotiation)* der Anforderungen hat das Ziel, hinsichtlich der Anforderungen an das Produkt eine weitgehende Übereinstimmung zwischen allen involvierten Personen zu erzielen. Allgemein wird als Hauptziel bei der *Spezifikation* der Anforderungen die Ableitung einer möglichst formalen Produktspezifikation angesehen, die als Referenzdokument bei den nachfolgenden Entwicklungstätigkeiten eingesetzt wird. Hierzu muss die Dokumentation der Anforderungen für die beteiligten Personengruppen eine Reihe von Modellen, mit unterschiedlichen graphischen Darstellungen und Abstraktionsebenen bereitstellen, beispielsweise formale Spezifikationen für Softwareentwickler, graphische Darstellungen für Manager oder umgangssprachliche Beschreibungen für Benutzer. Beim *Validieren und Verifizieren* der Anforderungen ist nicht nur sicherzustellen, dass die Anforderungsspezifikation vollständig ist und mit den Erwartungen der Benutzer übereinstimmt, sondern es muss auch gewährleistet werden, dass die spezifizierten Anforderungen konsistent und korrekt sind. Beim Validieren der Anforderungen ist eine enge Zusammenarbeit zwischen Benutzer und Analytiker erforderlich, denn ein versierter Analytiker kann zwar widersprüchliche Anforderungen in der Anforderungsspezifikation entdecken, prinzipiell können aber nur Auftraggeber bzw. Benutzer validieren, ob die Anforderungsspezifikation mit ihren Erwartungen an das zu erstellende Softwareprodukt übereinstimmt.

Zusammenfassend halten wir fest: viele Schwierigkeiten bei der Erstellung brauchbarer Anforderungsspezifikationen sind in der Tatsache zu suchen, dass die Anforderungsspezifikation zwei Aufgaben mit gegensätzlichen Zielen erfüllen soll. Auf der einen Seite soll sie informell, intuitiv und für Nicht-Informatiker verständlich sein, damit Benutzer und Auftraggeber mit Analytikern und Testern validieren können, „*that we are building the right product*“ ([Boehm84]). Auf der anderen Seite soll die Anforderungsspezifikation präzise genug sein, um Analytikern, Testern und Entwicklern als Ausgangsbasis für Prüfungen bezüglich der Frage „*that we are building the product right*“ ([Boehm84]) dienen kann.

## Anforderungsermittlung mit Use Cases

Die Anforderungsspezifikation präzisiert die sachlichen Aspekte eines Geschäftsprozesses in Hinblick auf eine bestimmte Anwendung. Strukturierte Methoden wie die „Moderne Strukturierte Analyse“ (MSA, vgl. [Yourdon89]) und — mehr noch — die ersten objektorientierten Methoden wie die objektorientierte Analyse (*Object-Oriented Analysis*, OOA [CoaYou90]) oder der objektorientierte Entwurf (*Object-Oriented Design*, OOD [WBW+90]) betonen die strukturellen, eher daten- bzw. informationsorientierten Aspekte — das Klassenmodell. Erst die von Ivar Jacobson Anfang der 90er Jahre eingeführten Use Cases ändern den Stellenwert der funktionalen Sicht in der objektorientierten Welt signifikant ([JCJ+92]). Erstmals werden Anforderungen in Form des gewünschten, extern beobachtba-

ren Verhaltens als primäres Element der Projektplanung verwendet, an dem sich die einzelnen Entwicklungstätigkeiten orientieren.

Bezüglich der organisatorischen Aspekte kategorisiert Jacobson menschliche oder maschinelle Benutzer, die mit der Anwendung interagieren, in Form sogenannter Aktoren:

Actors model anything that needs information exchange with the system. [JCJ+92]

Präziser als diese etwas kurze Beschreibung definiert die UML den Begriff des Aktors:

An actor is a role of object or objects outside of a system that interacts directly with it as part of a coherent work unit (a use case). An Actor element characterizes the role played by an outside object; one physical object may play several roles and therefore be modelled by several actors. [OMG97] (Notation Guide)

Aktoren sind im Metamodell der UML Unterklasse der Metaklasse **Classifier** und können somit generalisiert werden. Jeder Aktor kann mindestens die Rollen aller ihn generalisierenden Aktoren spielen:

Two or more actors may have commonalities, i.e., communicate with the same set of use cases in the same way. This commonality is expressed with generalizations to another (possibly abstract) actor, which models the common role(s). An instance of an heir can always be used where an instance of the ancestor is expected. [OMG97] (Semantics Guide)

Generalisiert ein Aktor  $a$  einen Aktor  $a'$ , so schreiben wir abkürzend  $a' \leq a$ .

Wir kommen nun zu den funktionalen Aspekten. Einfach gesprochen spezifiziert jeder Use Case eine in sich abgeschlossene (Teil-) Funktionalität, die bei der Benutzung der Anwendung für die beteiligten Aktoren einen „messbaren“ Fortschritt oder ein bestimmtes (Teil-) Ergebnis innerhalb des Geschäftsprozesses erbringt. Jacobson beschreibt einen Use Case als

... a specific way of using the system by using some part of the functionality [...]. Each use case constitutes a complete course of events initiated by an actor and it specifies the interaction that takes place between the actor and the system. [JCJ+92].

Operationale und verhaltensbezogene Aspekte der Use Cases werden textuell spezifiziert, wobei Jacobson einen mehr oder weniger „prosaischen“ Stil vorschlägt. Jeder Use Case beschreibt für eine klar umrissene Aufgabe den „normalen“ Ablauf sowie alternative und z.B. aufgrund von Ausnahmesituationen notwendige Varianten:

The use case construct is used to define the behaviour of a system or other semantic entity without revealing the entity's internal structure. Each use case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity. [OMG97] (Semantics Guide)

Jede konkrete Ausführung eines Use Case stellt eine Sequenz von Ereignissen dar, die als *Szenario*<sup>1</sup> bezeichnet wird:

**Scenario.** A specific sequence of actions that illustrates behaviours. A scenario may be used to illustrate an interaction. [OMG97] (Semantics Guide)

Bei der Anforderungsermittlung können prinzipiell zwei Fälle eintreten, die eine weitere Strukturierung des Use Case Modells notwendig machen.

1. Die Vielzahl der Use Cases macht das Modell unüberschaubar.
2. Einzelne Use Cases werden sehr komplex.

Zunächst können Use Cases ebenso wie Aktoren als Unterklasse der Metaklasse **Classifier** generalisiert werden. Im Falle vieler kleinerer Use Cases können in der UML Use Cases — wie alle anderen Modellierungselemente auch — zu Paketen zusammengefasst werden. Dies verbessert zwar den Überblick, gibt aber keine Hilfe bei der Vermeidung von Redundanzen und bei der Darstellung von Abhängigkeiten zwischen verschiedenen Use Cases. Hierzu und für den Fall komplexer Use Cases gibt Jacobson zwei als *extends* und *uses* bezeichnete spezielle Generalisierungsbeziehungen zwischen Use Cases an.

Zwei Use Cases werden durch die *extends*-Beziehung verknüpft, wenn sie ähnlich sind, jedoch einer der beiden Use Cases eine erweiterte Funktionalität beschreibt:

An *extends* relationships from use case A to use case B indicates that an instance of use case B may include (subject to specific conditions specified in the extension) the behaviour specified by A. [OMG97] (Notation Guide)

Die *uses*-Beziehung wird verwendet, wenn mehrere Use Cases eine bestimmte Teilfunktionalität benötigen. Diese notwendige Teilfunktionalität wird zur Redundanzvermeidung in einem separaten Use Case beschrieben, der von den anderen Use Cases verwendet wird:

A *uses* relationship from use case A to use case B indicates that an instance of the use case A will also include the behaviour as specified by B. [OMG97] (Notation Guide)

Sowohl die *uses*- als auch die *extends*-Beziehung fügen also zusätzliche Funktionalität in eine Basisfunktionalität ein. Im Falle von *uses* ist die zusätzliche Funktionalität unabdingbar, im Falle von *extends* ist sie optional, d.h. eine Instanziierung des erweiterten Use Cases ergibt auch ohne die zusätzliche Funktionalität sinnvolle Szenarien.

Zur überblicksartigen Darstellung der Use Cases für eine Anwendung benutzt Jacobson das *Use Case Diagramm*. In ihm werden (menschliche und maschinelle) Aktoren als „Strichmännchen“ visualisiert. Jeder Use Case wird durch eine Ellipse dargestellt, die mit dem Namen des Use Cases gekennzeichnet wird. Die an einem Use Case beteiligten Aktoren werden durch (ungerichtete oder gerichtete) Assoziationen mit der dem Use Case entsprechenden Ellipse verbunden. Geschlossene Pfeile (Generalisierungen) verbinden zwei Use Cases, wenn

---

<sup>1</sup> Typische, meist informell beschriebene Szenarien zur Illustration funktionaler Anforderungen bilden seit Mitte der neunziger Jahre einen eigenen Forschungsgegenstand (vgl. z.B. [Carroll95][WPJ+98]).

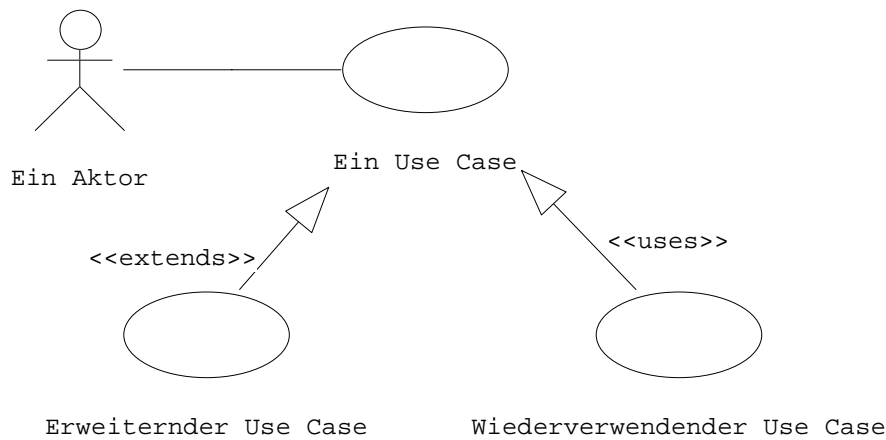


Abb. 2.4 Elemente des UML Use Case Diagramms

sie in einer uses- oder einer extends-Beziehung zueinander stehen; hierbei kennzeichnen wir an dem Pfeil, um welche der beiden Beziehungen zwischen Use Cases es sich handelt (Abb. 2.4).

Bezüglich der informationsbezogenen Aspekte empfiehlt Jacobson, für jeden Use Case ein separates Objektdiagramm zu erstellen, welches die am Use Case partizipierenden Objekte und ihre Abhängigkeiten darstellt:

For each use case you prepare an object diagram, which shows the objects participating in the use case and their dependencies. [Jacobson95]

Er schlägt vor, die Funktionalität der Use Cases in Anlehnung an die drei Perspektiven Präsentation, Information und Verhalten auf Objekte aus den drei Kategorien Schnittstelle, Entitäten und Kontrolle zu verteilen, deren Symbole Abb. 2.5 zeigt ([JCJ+92]):

- ❑ *Schnittstellen-Objekte (interface objects<sup>1</sup>)* verkapseln die funktionalen Teile der Anwendung, die unmittelbar von der Anwendungsumgebung abhängen. Hauptaufgabe der Schnittstellen-Objekte ist, die Tätigkeiten von Aktoren in anwendungsinterne Er-

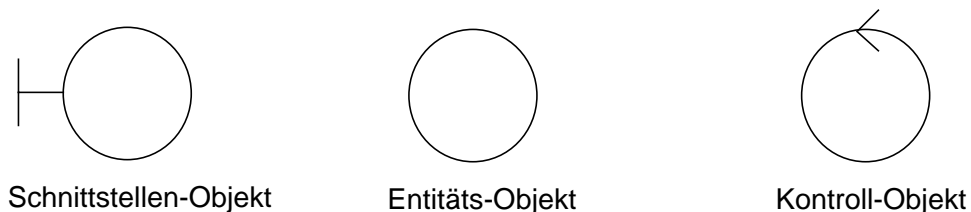


Abb. 2.5 Darstellung der drei Objektarten nach Jacobson

<sup>1</sup> In [JGJ97] bezeichnet Jacobson Schnittstellen-Objekte passender als *Begrenzer-Objekte (boundary objects)*.

eignisse zu übersetzen, und umgekehrt anwendungsinterne Ereignisse und Zustände, an denen die jeweiligen Aktoren interessiert sind, in eine dem Akteur präsentierbare Form zu überführen.

- ❑ *Entitäts-Objekte (entity objects)* modellieren Daten, welche die Anwendung über längere Zeit verwendet. Typischerweise überleben solche Daten den Ablauf einzelner Use Cases. Neben den eigentlichen Daten kapseln Entitäts-Objekte auch solche Funktionalitäten, die ihre Informationen unmittelbar betreffen.
- ❑ *Kontroll-Objekte (control objects)* kapseln die „verbleibende“ Funktionalität komplexer Use Cases, die sich nicht eindeutig zu Objekten der beiden anderen Kategorien zu teilen lässt — typischerweise also transaktionsbezogene Funktionalität, (Teil-) Funktionalität, die von mehreren Use Cases verwendet wird, oder aber Funktionalität, welche eine Brücke von Interface-Objekten zu Entitäts-Objekten bildet.

Zur Spezifikation des dynamischen Ablaufs verwendet Jacobson Interaktionsdiagramme ähnlich dem UML-Sequenzdiagramm:

An interaction diagram shows how participating objects interact to offer the use case; one is required for each concrete use case. Interaction takes place, when objects send stimuli to one another. As you draw the diagram, you also define all the stimuli sent, including the parameters. [Jacobson95]

Auf der linken Seite des Interaktionsdiagramms (an der „Anwendungsgrenze“) beschreiben entsprechende Texte aus der Use Case Beschreibung den Ablauf bzw. die Ereignisse. Jedem am Use Case beteiligten Objekt wird eine vertikale Linie im Diagramm zugeordnet; vertikale Rechtecke auf diesen Linien veranschaulichen die Ausführung einer Operation. Horizontale Pfeile spiegeln die im Verlauf einer Operation gesendeten Botschaften wider, die dementsprechend zur Ausführung einer Operation im Zielobjekt der Botschaft führen.

*Beispiel 2.1.* Abb. 2.6 zeigt exemplarisch ein Interaktionsdiagramm für den Ablauf der Operation push in einem Objekt der Klasse **ArrayStack** (vgl. [Meyer97]).

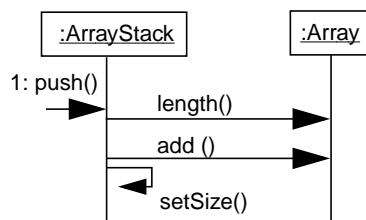


Abb. 2.6 Interaktionsdiagramm der Operation „**ArrayStack**.push“

## 2.2 Entwurf und Implementation

Beginnend mit einigen Überlegungen zur Software-Architektur betrachten wir in diesem Abschnitt exemplarisch den Entwurf objektorientierter Software mit Jacobsons *OOSE* (*object-oriented software engineering*), der Methode des „Objectory“ Software-Entwicklungsprozesses ([JCJ+92], s. a. Kapitel 2).

In der Welt des objektorientierten Software-Engineering herrscht mittlerweile Übereinstimmung darin, dass „gute“ Entwürfe — und insgesamt erfolgreiche Projekte — nur auf der Grundlage einer guten Architektur möglich sind (vgl. [Booch94][JGJ97][Meyer97][Züllighoven98]). Sogenannte „Freistil-Entwürfe“ führen erfahrungsgemäß nur bei kleinen Projekten zu lauffähigen Anwendungen (vgl. [Hatton98][Lauesen98]). Rein transformatorische Entwürfe auf der Grundlage formaler Spezifikationen sind zumindest heute nur in eng eingegrenzten Domänen wie z.B. bei eingebetteten Echtzeitsystemen möglich (vgl. [Lauesen98][TheGot98]).

Da wir diese Arbeit im Umfeld kommerzieller Anwendungen (z.B. Informationssysteme) ansiedeln, konzentrieren wir uns auf „manuelle“ Entwürfe, bei denen die Entwurfsfreiheit durch die Architektur sowie die Entwurfsmethodik eingeschränkt wird (vgl. [Shaw94] [ShaGar96] [GHJ+94][BMR+97]). Wie wir sehen werden, mündet der Entwurf mit *OOSE* in eine Architektur mit den drei Schichten Benutzungsschnittstelle, Anwendungslogik und Datenhaltung (s. z.B. [ShaGar96][BMR+97]). Allgemein halten wir fest: je klarer die Architekturvorgabe ist und je disziplinierter sie eingehalten wird, desto schematischer und nachvollziehbarer wird der Entwurf und die Implementation.

Jacobsons *OOSE* stützt sich im Prinzip auf fünf Modelle (Abb. 2.7):

- ❑ das *Anforderungsmodell* (requirements model) spezifiziert mit Use Cases die funktionalen Anforderungen an die zu entwickelnde Anwendung;
- ❑ das *Analysemodell* (analysis model) spezifiziert mit einem Klassenmodell möglichst robust und leicht änderbar die strukturellen Anforderungen;
- ❑ das *Entwurfsmodell* (design model) verfeinert das Analysemodell im Hinblick auf die gewählte Implementations-Umgebung;
- ❑ das *Implementationsmodell* (implementation model) besteht aus dem ausführlich dokumentierten Quellcode und den zusätzlichen Dateien zur Anwendungs-Erstellung (*build*);
- ❑ das *Testmodell* (test model) spezifiziert die Komponenten-, Integrations- und Systemtestfälle.

Die fünf Modelle werden mit den Tätigkeiten Anforderungsermittlung, Konstruktion und Test iterativ und inkrementell erstellt.

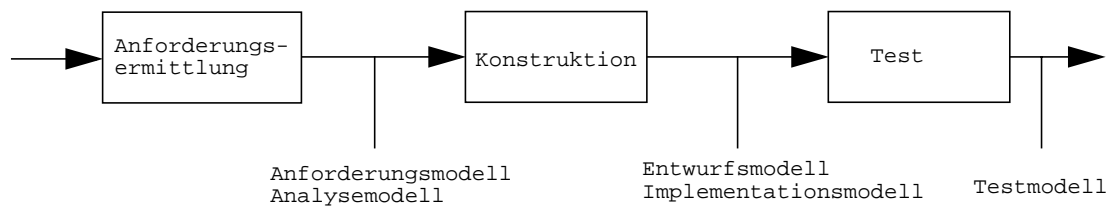


Abb. 2.7 Teilprozesse und Modelle in OOSE ([JCJ+92])

Jacobson verteilt die in Use Cases spezifizierte Funktionalität bezüglich der Perspektiven Präsentation, Information und Verhalten auf Objekte im Analysemodell. Hierbei unterscheidet Jacobson Schnittstellenobjekte, Entitätsobjekte und Kontrollobjekte. Diese Modelle dienen der Kommunikation sowohl mit dem Auftraggeber als auch mit den Entwicklern.

Die auf den Modellen der Anforderungsermittlung aufbauende *Konstruktion* besteht aus dem Entwurf und der Implementation. Im Wesentlichen besteht die Grundidee des Entwurfs nach OOSE im Hinzufügen einer vierten Perspektive, der Implementations-Umgebung (Abb. 2.8). Analyseklassen werden zu Teilsystemen des Entwurfs (*Blöcke*, vgl. [JCJ+92]). Auch Jacobson übernimmt also das Analysemodell als erste Näherung des Entwurfsmodells. Er legt besonderen Wert auf die Verfolgbarkeit zwischen den Modellen:

The first attempt at a design model can be made mechanically, based on the analysis model. The transformation is made so that initially each analysis object becomes a block. [...] As each analysis object is traceable to a block, changes introduced in the analysis model will be local in the design model and thus also in the source code. Note that the traceability is bidirectional, that is, it also goes the other way — we can trace a class in the source code back to the analysis and see whatever gave rise to it. [JCJ+92]

Nach dieser „initialen Entwurfstransformation“ werden Verfeinerungen der Blöcke<sup>1</sup> und Erweiterungen um implementationsspezifische Klassen und Teilsysteme durchgeführt.

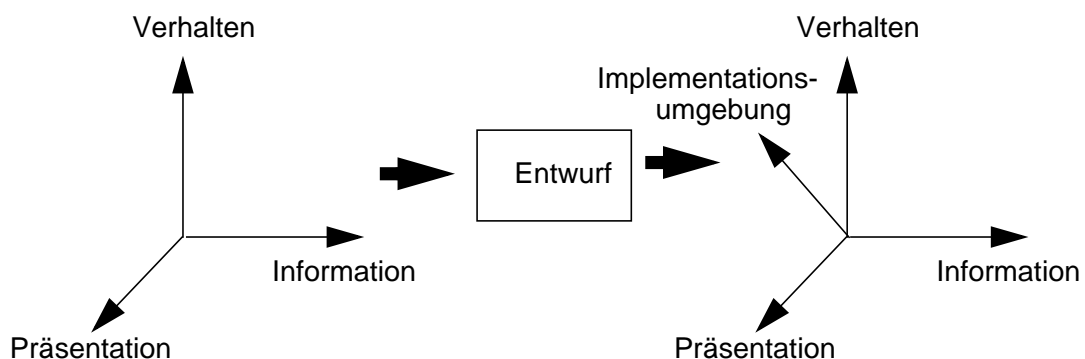


Abb. 2.8 Von der Anforderungsermittlung zum Entwurf: OOSE (nach [JCJ+92])

## 2.3 Test und Administration

Dem Stand der Technik der Qualitätssicherung für objektorientierte Anwendungen ist das nächste Kapitel gewidmet, so das wir hier lediglich noch einmal festhalten, dass sich gerade in der objektorientierten Softwareentwicklung die Qualitätssicherung über die gesamte Projektdauer erstreckt ([PagSix94], vgl. auch Abb. 1.2 und Abb. 2.1). Je nach Projektfortschritt bzw. Phase verschiebt sich ihr Fokus von der Prüfung der Anforderungsspezifikation bis hin zum (System- und Abnahme-) Test gegen die Anforderungsspezifikation. Abb. 1.2 auf Seite 10 zeigt dies angelehnt an ein „V-Modell“ der Softwareentwicklung (vgl. [PagSix94]). So sollte im Idealfall z.B. die Qualitätssicherung der Anforderungsspezifikation zu Testfällen für die Softwarespezifikation und den System- und Abnahme-Test führen. Abb. 2.9 zeigt die verschiedenen Testtätigkeiten und ihren logischen Zusammenhang.

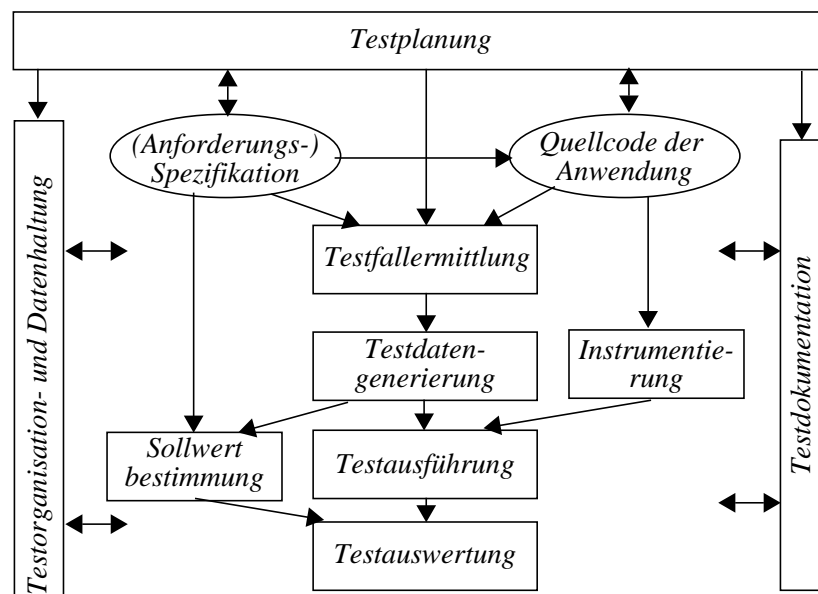


Abb. 2.9 Testtätigkeiten (nach [Grimm95])

Administrative Tätigkeiten sind z.B. die Auslieferung, das Konfigurationsmanagement, das Projektmanagement und das Umgebungsmanagement. Dem für diese Arbeit wichtigen Konzept der (Nach-) Verfolgbarkeit der Anforderungsspezifikation über den Entwurf in die Implementation als Teilaspekt des Konfigurationsmanagements wenden wir uns am Ende von Teil III zu. Einige Hinweise bzw. Verfahren für das Projekt- bzw. Testmanagement und speziell für den Testumgebungs Aufbau finden sich in den Teilen II (Kapitel 8), III und IV.

---

1 Oft wird ein Block zu einer Schnittstelle für eine Menge von (die Analyseklasse implementierenden) Entwurfs- bzw. Implementationsklassen. Hierfür gibt es ein spezielles Entwurfsmuster, die *Fassade* (s. [GHJ+94]).



# Kapitel 3

## Probleme der Qualitätssicherung für objektorientierte Anwendungen

*Things had literally crawled to halt in system testing, with each new bugfix generating enough new bugs that almost no forward progress was achieved.*  
[Glass98]

Während zu Beginn der 80er Jahre die ersten objektorientierten Programmiersprachen industriell verwendet wurden und sich ab 1985 methodische Ansätze zur objektorientierten Softwareentwicklung durchzusetzen begannen, blieb die Qualitätssicherung in der Objektorientierung lange Zeit unbeachtet. Viele populäre Methodologen verlieren kein Wort hierüber (vgl. [CoeYou90][WBW+90][WalNer95]). Grady Booch schreibt lediglich:

... the use of object-oriented design doesn't change any basic testing principles; what does change is the granularity of the units tested. [Booch94]

Jacobson behauptet:

The testing of a system which has been developed with an object-oriented method does not differ considerably from the testing of a system developed by any other method. [JCJ+92]

Bei James Rumbaugh lesen wir sogar:

Both testing and maintenance are simplified by an object-oriented approach... [RBP+91]

Die anfänglich oft vertretene Meinung, dass objektorientierte Software einen erheblich reduzierten Prüfungsaufwand erfordert und bekannte Prüfverfahren unmodifiziert verwendet werden können, hat sich allerdings nicht erfüllt. Erste Anzeichen für einen erhöhten Qualitätssicherungsbedarf bei der Entwicklung objektorientierter Software finden wir bei Perry und Kaiser, die bei der Untersuchung des Wiederverwendungspotentials objektorientierter Software 1990 feststellen:

... we have uncovered a flaw in the general wisdom about object-oriented languages — that "proven" (that is well-understood, well-tested, and well-used) classes can be reused as superclasses without retesting the inherited code. [PerKai90]

### Der bekannte Test-Spezialist Boris Beizer bemerkt letztendlich

... it costs a lot more to test oo-software than to test ordinary software — perhaps four or five times as much. [...] inheritance, dynamic binding, and polymorphism creates testing problems that might effect a testing cost so high that it obviates the advantages. [Beizer94]

Die Objektorientierung ist heute ein weitverbreitetes Entwicklungsparadigma. Allerdings wird ihre Eignung nach der anfänglichen Begeisterung in neuester Zeit auch angezweifelt ([Hatton98]). Betrachten wir die objektorientierte Softwareentwicklung aus dem Blickwinkel der Qualitätssicherung, so stellen sich Fragen wie z.B.

- ☐ Welche Fehler treten bei objektorientierter Software auf?
- ☐ Welche Modelle sind brauchbar, um auf mögliche Fehlerquellen schließen zu können?
- ☐ Mit welchen Verfahren können Tests für objektorientierte Modelle, Spezifikationen und Implementationen ermittelt werden?
- ☐ Inwiefern müssen geerbte Methoden in der Unterklasse erneut geprüft werden?
- ☐ Können Testfälle wiederverwendet werden?
- ☐ Wie sind Polymorphismus und dynamische Methodenbindungen zu testen?
- ☐ Wie werden objektorientierte Testfälle und Testdaten repräsentiert?

Vor diesem Hintergrund geben wir nach der Festlegung einiger Grundbegriffe in den nächsten Abschnitten einen Überblick darüber, welche Gebiete der Qualitätssicherung durch die Objektorientierung überhaupt berührt werden und die Entwicklung und den Einsatz von speziellen Verfahren erforderlich machen.

## 3.1 Formale Methoden

Wiederholt wird in der Literatur behauptet, dass die einzelnen Operationen bei der objektorientierten Programmierung wesentlich einfacher sind als in der herkömmlichen, prozeduralen Programmierung (vgl. [Meyer85][PerKai90][Hatton98]). Die Tatsache, dass die heutzutage (oft objektorientiert) entwickelten Anwendungen ein Vielfaches der Komplexität der noch vor zehn Jahren mit herkömmlichen Methoden durchgeführten Projekte haben, stellt uns somit vor die Frage, wo sich die höhere Komplexität objektorientierter Anwendungen niederschlägt? Die offensichtliche Antwort ist: in den (internen) Interaktionen, also der Kommunikation zwischen Objekten (vgl. [Binder96a][HKR+97][Winter98], s. auch Anhang A.2).

Ein schon seit Anfang der siebziger Jahre beschrittener Weg zur Komplexitätsbeherrschung ist die Verkapselung von Daten und Operationen in abstrakten Datentypen (ADT, vgl. [LisZil75][GutHor78]). Dieser Weg geht vom Geheimnisprinzip aus ([Parnas72]): Realisierungsdetails werden hinter wohldefinierten Schnittstellen verborgen, die z.B. mit Verträgen (*contracts*), also Vor- und Nachbedingungen jeder Operation und Invarianten des ADT spezifiziert wird (vgl. [Meyer97]). Von der Korrektheit der einzelnen Teile einer Anwendung wird dann auf die Korrektheit der vollständigen Anwendung geschlossen (vgl. [Poetzsch97]).

Die Objektorientierung erweitert diese Konzepte um die Vererbung und den damit verbundenen Polymorphismus, so dass im Prinzip eine in Bezug auf die Anzahl von Operationen exponentielle Anzahl von internen Interaktionen möglich ist, die beim Einsatz herkömmlicher Spezifikationsmethoden auch alle zu verifizieren bzw. zu testen sind (s. Anhang A.2). Wegner bemerkt hierzu:

Object-oriented programs are more expressive than procedure-oriented programs in providing clients with continuing services over time: they are specified by marriage contracts that cannot be expressed by procedure-oriented sales contracts. [Wegner97]

Und weiter:

Object-oriented programming is not compositional; composite structures of interactive components have emergent behavior, such that the whole is greater than the sum of its parts. [WegGol99]

Aus diesen Gründen werden in aktuellen Arbeiten formale Methoden bzw. Verifikationskalküle für objektorientierte (Programmier-) Sprachen entwickelt. Erst das Zusammenspiel mächtiger Sprachen für Schnittstellenspezifikationen und Implementationen mit entsprechenden Kalkülen und universellen Techniken zur Spezifikation und zum automatischen Beweis der Korrektheit in einer logik-basierten Programmierumgebung wird den Einsatz formaler Methoden in der objektorientierten Softwareentwicklung praktikabel machen (vgl. [Poetzsch97]).

Daneben behindert die unabdingbare Forderung nach „benutzerfreundlichen Sichten“ bei der Anforderungsermittlung den Einsatz formaler Methoden, da die unterschiedlichen Abstraktions- und Granularitätsniveaus den Übergang von informellen, für den Benutzer verständlichen Dokumenten zu formalen Spezifikationen erschweren.

Zur Zeit spielen daher formale Methoden in der industriellen Softwareentwicklung noch keine große Rolle und werden fast ausschließlich für bestimmte Teile hoch-sicherheitskritischer Anwendungen eingesetzt (vgl. [DHP+99]). Auch wenn die komplette Anwendung formal spezifiziert und verifiziert wäre, reicht dies für die Qualitätssicherung nicht aus. In Riedemann's Worten:

Die Programmverifikation hat folgende Vorteile und Nachteile bzw. Probleme:

Vorteile: Das Programm wird (bezüglich der Eingabe- und Ausgabezusicherungen) als korrekt für alle Eingabefälle bewiesen. (Dies gilt für den Quellcode, für den Objektcode gilt dies natürlich

nur, falls der Compiler dieselbe Programmsemantik hat wie das Validationswerkzeug).

Nachteile/Probleme: [...] Ein als korrekt bewiesenes Programm kann dennoch falsche Resultate liefern: i.) Die Semantik der Programmiersprache kann falsch modelliert sein. ii.) Der Compiler, das Betriebssystem oder die Hardware können anders als angenommen arbeiten [...] iii.) Die Eingabe- und Ausgabezusicherungen in ihrer prädikatenlogischen Form können die eigentliche (meist verbale) Anforderungsspezifikation des Auftraggebers falsch wiedergeben. [Riedemann97]

Nach diesen Ausführungen zur prinzipiellen Notwendigkeit qualitätssichernder Tätigkeiten gerade in der objektorientierten Softwareentwicklung betrachten wir nun (dynamische) Verfahren zur Prüfung von (ausführbaren) Programmen. Hierbei gehen wir zunächst auf strukturelle Tests (white-box Tests) und danach auf Tests gegen die (Anforderungs-) Spezifikation (black-box Tests) ein.

## 3.2 Struktureller white-box Test

Kommen wir zum white-box Test, also zu dynamischen Prüfungen gegen strukturelle Eigenschaften der Anwendung (bzw. des Codes). Als Prüfgegenstände sind in der objektorientierten Softwareentwicklung nach ihrer Granularität

- ☐ Methoden,
- ☐ Klassen,
- ☐ Cluster<sup>1</sup>,
- ☐ Teilsysteme und
- ☐ die gesamte Anwendung

zu unterscheiden (vgl. [Binder96a][Siegel96][Sneed96]). Obwohl viele Techniken aus der Qualitätssicherung in der prozeduralen bzw. modularen Softwareentwicklung auch auf objektorientierte Software anwendbar sind ([Binder96a]), konfrontiert uns letztere auch hier mit speziellen Problemen, welche neue Lösungen erfordern (s. auch Anhang A.2). Für dynamische Prüfungen stehen wir prinzipiell vor zwei Extremen bei der Vorgehensweise:

- ☐ Wir prüfen alle Methoden separat, danach die Interaktionen von Methoden innerhalb einer Klasse, und letztlich die Interaktionen zwischen Methoden verschiedener Klassen.
- ☐ Wir führen direkt Integrations- bzw. (partielle) Systemtests durch und testen somit von Beginn an die Interaktionen zwischen Methoden verschiedener Klassen.

Betrachten wir zunächst die Prüfung der einzelnen Methoden. Solche Prüfungen sind für objektorientierte Software schwierig, da die Methoden einer Klasse über die Instanzvariablen

---

<sup>1</sup> Ein *Cluster* ist eine Gruppe logisch zusammengehörender Klassen [Meyer97].

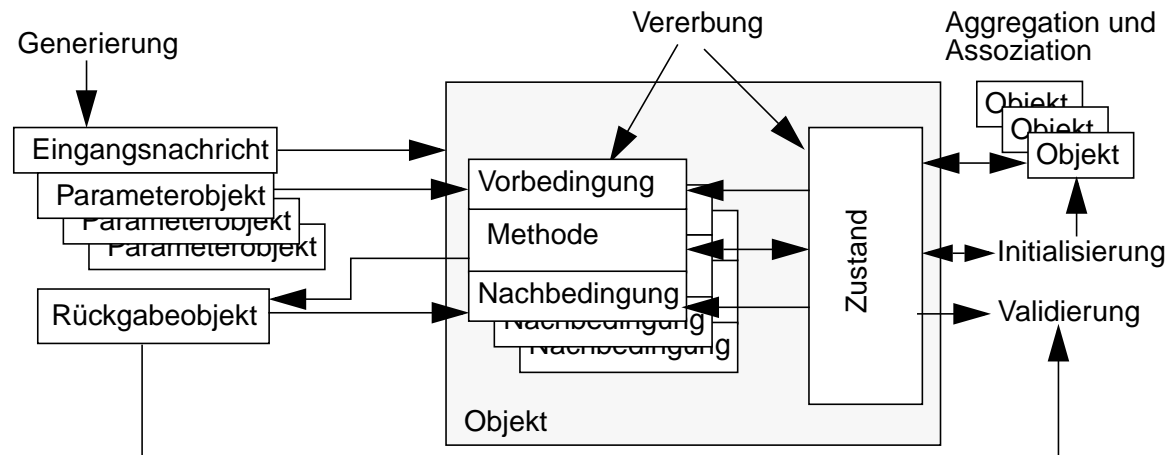


Abb. 3.1 Problematik des Klassentests (nach [Sneed96])

gekoppelt sind und darüber hinaus meistens weitere Methoden der eigenen oder anderer Klassen benutzen. Prüfverfahren aus der prozeduralen bzw. modularen Programmierung sind eingeschränkt<sup>1</sup> anwendbar, jedoch oft wenig aufschlussreich, weil die Methoden im Allg. nur eine geringe strukturelle Komplexität haben (vgl. [Binder96a]). Daher gilt die Klasse bzw. das einzelne Objekt im Normalfall als der kleinste Prüfgegenstand (s. z.B. [McGreKor94] [Binder96a][Sneed95]).

Auch bei der Prüfung einer einzelnen Klasse bzw. eines isolierten Objekts treten Schwierigkeiten auf (Abb. 3.1):

- ❑ Es müssen Eingangsnachrichten generiert werden — diese enthalten oft Objekte als Parameter. Für die Klassen der Parameterobjekte ergibt sich durch die Vererbung und den damit verbundenen Polymorphismus eine mit der Anzahl ihrer Unterklassen exponentiell wachsende Anzahl von Kombinationen.
- ❑ Der Prüfgegenstand sowie alle Parameterobjekte untereinander müssen initialisiert, also in einen bestimmten Zustand gebracht werden.
- ❑ Sieht das Klassenmodell für den Prüfgegenstand bzw. die Parameterobjekte Assoziations- oder Aggregationsbeziehungen zu anderen Objekten vor, sind auch diese geeignet zu initialisieren.
- ❑ Bei der Erstellung des erwarteten Ergebnisses bei der Testausführung sind neben dem Rückgabewert der aufgerufenen Methode auch die Parameter und Ergebnisse sowie die korrekte Reihenfolge der bei dem Test ausgeführten Methoden zu berücksichtigen (vgl. [JorEri94]).

<sup>1</sup> Zum Test einer Operation muss ein Objekt der die Operation definierenden Klasse instanziiert werden, es gibt also keine Operationentreiber im herkömmlichen Sinn.

- ❑ Zur Überprüfung von Verträgen, also Vor- und Nachbedingungen von Operationen sowie Klasseninvarianten sind Zugriffe auf den Zustand des Prüfgegenstands als auch auf die (Zustände der) Botschaftsparameter notwendig.
- ❑ Im Unterschied zu prozedural implementierten Anwendungen muss bei der Überdeckungsmessung für dynamische Tests objektorientiert implementierter Anwendungen unterschieden werden, ob die Angaben sich auf Klassen oder Objekte beziehen. Beispiel: Eine Klasse **K** bietet zwei Operationen M1 und M2 an. Instanziiert man nun zwei Objekte O1 und O2 von **K** und verwendet nach O1.M1 dann O2.M2, ist dann die Abdeckung 50% oder 100%?
- ❑ Eine ähnliche Unterscheidung muss im Falle der Vererbung bezüglich der Klasse des die Operation ausführenden Objekts getroffen werden.

Strauß und Hörnke stellen dementsprechend fest:

In C++ sind die Klassen tendenziell sehr stark vernetzt, so dass die Eingabe der Daten zu den Instanzen und die Instanzenverwaltung zeitaufwendig und unhandlich war. [StrHör98]

Methodische Unterstützung zum Klassentest geben z.B. die Arbeit von Jüttner und die meisten der von Binder bzw. Kung et al. betrachteten Arbeiten ([Jüttner93][Binder96a][KHG98]). Barbey leitet aus einer algebraischen Spezifikationssprache und hierarchischen Petrinetzen Testfälle für den Klassentest ab ([Barbey97]). Siepmann und Newton beschreiben ein Werkzeug zum Klassen- und Integrationstest für Smalltalk-80 Programme ([SieNew94]). Rüppel entwickelt ein generisches Werkzeugkonzept für den Klassentest ([Rüppel96]).

Der Vorteil des Prüfens von Aufrufsequenzen besteht in dem geringeren Aufwand bei der Erstellung der Testumgebung sowie der Testfälle und Testdaten, da ein „Aufruf“ eine ganze Reihe von Operationen prüfen kann (vgl. [JorEri94]). Daneben werden nicht nur die einzelnen Operationen, sondern direkt auch ihr Zusammenspiel geprüft. Als Nachteil sehen wir den aufgrund der vorgegebenen (partiellen) Reihenfolge der Operationen höheren Kommunikationsaufwand verbunden mit dem geringeren Potential der „Parallel-Arbeit“ bei Entwicklungs- und Test-Tätigkeiten. Darüber hinaus besteht die Gefahr der Fehlermaskierung, d.h. nachfolgende Operationen in einem Test verhindern die Entdeckung eines Fehlers in einer Operation (vgl. [Spillner90]).

Im Gegensatz zu modularer Software, wo größtenteils Baumstrukturen der Benutzungsbeziehungen auftreten, sind in objektorientierten Anwendungen oft Graphen mit starken Zusammenhangskomponenten (*cluster*) und zyklischen Abhängigkeiten<sup>1</sup> zu beobachten (vgl. [Overbeck94][Binder96a][KHG98], s. auch Anhang A.2). Für einen 30 Klassen umfassenden Ausschnitt der Container-Klassen in der Smalltalk-80 Klassenbibliothek ([GolRob89]) wur-

---

<sup>1</sup> Zyklische Abhängigkeiten treten häufig bei der Verwendung von Entwurfsmustern ([GHJ+94]) auf (vgl. [Winter98]).

de beispielsweise ein 26 Klassen enthaltender Cluster gemessen<sup>1</sup> ([Mohl97][Winter98]). Eine dies berücksichtigende Integrationsstrategie kann zu einem um über 80% verminderten Aufwand für die Erstellung von Stubs und Treibern führen, erscheint jedoch ohne Werkzeugunterstützung kaum sinnvoll (vgl. [Overbeck94]).

In Frameworks bzw. Klassenbibliotheken liegt oft eine Mischung abstrakter und instanzierbarer Klassen vor (vgl. [Johnson97]). Die Nutzung erfordert somit eine Mischung von Unterklassenbildung („Vererbungsnutzung“) und Delegation bzw. Methodenaufrufen („Anwendungsnutzung“), was den Test weiter erschwert (vgl. [Berard93] [Jüttner93] [Overbeck94]).

Wiederverwendbare Klassen werden so allgemein wie möglich gehalten, um eine möglichst breite Benutzbarkeit zu gewährleisten. Daher müssen beim Test wiederverwendbarer Klassen strengere Maßstäbe angelegt werden, wobei im Prinzip alle möglichen Wiederverwendungen der Klasse in Benutzungs- und in Vererbungsbeziehungen berücksichtigt werden müssen. Da dies in der Regel nicht möglich ist, ist bei jeder Wiederverwendung zu prüfen, ob der Anwendungskontext weitere Tests der wiederverwendeten Klasse notwendig macht. Bei der Wiederverwendung wird jedoch oft nur ein kleiner Teil der Funktionalität der Klasse benutzt, so dass eine vollständige Überdeckung der wiederverwendeten Klasse im Rahmen der Anwendung nur schwer erreicht wird.

Fassen wir zusammen: Aufgrund der Verkapselung von Methoden und Attributen in Klassen, „kleinen Methoden“ und der damit verbundenen erhöhten Komplexität des Zusammenspiels von Methoden sowie der größeren Ausdruckstärke durch Vererbung und Polymorphismus sind systematische Prüfungen in der objektorientierten Softwareentwicklung deutlich komplexer als z.B. in der modularen Softwareentwicklung (vgl. [Jüttner93][Binder96a]). White-box Tests auf der Granularität einzelner Klassen oder gar einzelner Methoden erweisen sich in der Praxis zunehmend als wenig aussagekräftig, da die Komplexität in den Interaktionen zwischen verschiedenen Objekten liegt (vgl. z.B. [JorEri94][HKR+97][Wegner97]). Darüber hinaus sind solche Tests mit sehr hohem Aufwand verbunden, so dass ihr gewinnbringender Einsatz nicht gegeben ist (vgl. [Sneed95][StrHör98]). Wir gehen daher im nächsten Abschnitt näher auf den black-box Test einer Anwendung gegen die Anforderungsspezifikation ein.

### 3.3 Black-box Test gegen die Anforderungsspezifikation

In der Literatur über objektorientiertes Testen wurden bisher hauptsächlich strukturelle oder auf formalen Spezifikationen basierende funktionale Verfahren für den Klassentest sowie der Integrationstest betrachtet (vgl. [Binder96a][KHG98]). Da in der industriellen Praxis haupt-

---

<sup>1</sup> Da Smalltalk-80 eine ungetypte objektorientierte Programmiersprache ist und die Messungen ohne eine vorgeschaltete Typinferenz erhoben wurden, stellen die angegebenen Werte Obergrenzen dar (vgl. [PalSch94]).

sächlich in den „späten“ Phasen der Softwareentwicklung geprüft wird (vgl. Kapitel 1), konzentrieren wir uns auf den Test der Anwendung gegen die Anforderungsspezifikation (System- und Abnahmetest).

Mit dem häufig gehörten Argument, dass einer Anwendung ihre Implementationssprache „von Außen nicht anzusehen ist“, wurde der System- und Abnahmetest als Test gegen die Anforderungsspezifikation für objektorientierte Anwendungen bisher kaum behandelt. In Binders Worten:

Established practices for system testing can be transferred to object-oriented implementations with no loss of generality. [Binder96a]

Anforderungen an objektorientierte Anwendungen sind jedoch zunehmend mit objektorientierten Methoden und Modellen spezifiziert, gegen die getestet werden muss. Dies lässt die „Übertragbarkeit herkömmlicher Verfahren“ auf objektorientierte Anwendungen problematisch erscheinen.

Zunächst präzisieren wir die grundsätzliche Problemstellung mit einigen allgemeinen Definitionen und Erläuterungen zum System- und Abnahmetest sowie Arbeiten zum Test gegen „herkömmliche“ Anforderungsspezifikationen und skizzieren dann in Abschnitt 3.4 die relevanten Arbeiten zum Test gegen objektorientierte Anforderungsspezifikationen.

Der IEEE Standard 610 definiert den Systemtest wie folgt:

**System testing.** Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. [IEEE610.90]

Spezifischer wird Beizer, der jedoch über diese Definition hinaus nicht auf den Systemtest eingeht:

**System, System Testing** — A system is a big component. System testing is aimed at revealing bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects. System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery. [Beizer90]

Oft wird der Systemtest mit dem Test quantifizierbarer Merkmale der nicht-funktionalen Anforderungen wie z.B. dem Last- und Stresstest zusammengefasst, diese aber explizit vom Abnahmetest unterschieden. Wir lesen bei Pagel und Six:

Sind die Integrationsarbeiten abgeschlossen und liegt das System vollständig montiert vor, erfolgt der Systemtest. Dabei werden neben Tests der operativen Anforderungen und der Schnittstellen zu externen Systemen insbesondere Leistungstests in Form von Laufzeit- und Stresstest durchgeführt, welche die Effizienzanforderungen validieren und das Verhalten (z.B. die Robustheit) des Systems unter extremer Belastung untersuchen. Stresstest testen das Systems über einen längeren Zeitraum unter Spitzenbelastungen (z.B. Transaktionen oder Arithmetik) oder konfrontieren es



mit einer umfangreichen Datenmenge (Massentest). [...] Der Systemtest ist nicht zu verwechseln mit dem Abnahmetest, der vom Auftraggeber auf seiner eigenen Plattform vorgenommen wird und eine anders gelagerte Zielsetzung hat. [PagSix94]

Im Weiteren stellen wir zunächst einige „klassische“ und dann die bekannten „objektorientierten“ Ansätze zum Test der Anwendung gegen die Anforderungsspezifikation vor.

**Celentano et al.** geben eine Methode zum System- und Abnahmetest an ([CGL81]). Sie betonen die Entwicklersicht beim Systemtest im Gegensatz zur Benutzersicht beim Abnahmetest sowie die Notwendigkeit, neben der reinen Funktionalität — den „Aufgabenhierarchien“ — auch die erlaubten und nicht erlaubten Abläufe zu testen, unterscheiden also positives und negatives Testen. Den Hauptunterschied zwischen dem System- bzw. Abnahmetest und anderen Testtätigkeiten sehen Celentano et al. in folgenden Punkten:

- Der System- bzw. Abnahmetest testet gegen die Anforderungsspezifikation, der Komponenten und Integrationstest gegen Entwurfsdokumente.
- Die Sichtweise ist im System- und Abnahmetest eher funktional; Komponenten und Integrationstest konzentrieren sich auf die innere Struktur der Anwendung.
- Der Systemtest wird hauptsächlich vom QS-Team durchgeführt; der Abnahmetest in der Regel vom Benutzer bzw. Kunden; Komponenten und Integrationstest werden eher von Entwicklern durchgeführt.
- Bezüglich der Qualitätsmerkmale liegt das Hauptaugenmerk des System- bzw. Abnahmetests auf der Benutzbarkeit, Zuverlässigkeit, Robustheit und Performanz, während sich der Komponenten und Integrationstest auf die Korrektheit der Anwendung konzentriert.

Auch **Hetzel** geht ausführlich auf den System- und Abnahmetest ein ([Hetzel88]). Der Systemtest beginnt, wenn der Integrationstest beendet ist und endet, wenn die Anwendung vertrauensvoll genug erscheint, um vom Kunden abgenommen zu werden. Hetzel differenziert den Systemtest in einen anforderungsbasierten, einen performanzbasierten und einen entwurfsbasierten Teil:

- Der *anforderungsbasierte Systemtest* zeigt systematisch die Verfügbarkeit aller Funktionen und ihre Übereinstimmung mit der Spezifikation auf. Die Vorgehensweise orientiert sich dabei am Benutzer, d.h. es werden keine systeminternen Informationen verwendet. Die Testfälle werden aus dem Benutzerhandbuch oder der Anforderungsspezifikation abgeleitet.
- Der *performanzbasierte Systemtest* prüft systematisch die Performanzeigenschaften der Anwendung und zeigt, dass die entsprechenden Anforderungen erfüllt sind. Zuerst wird die Anwendung mit den spezifizierten Daten- bzw. Transaktionsvolumina getestet, danach bis hin zum Abbruch der Anwendung.

- Der *entwurfsbasierte Systemtest* erzielt eine systematische Überdeckung der Entwurfsstrukturen der Anwendung. Hierfür schlägt Hetzel die Verwendung entsprechender spezifikationsbasierter Testfälle aus dem Klassen- und Integrationstest oder die Modellierung des Transaktionsflusses in der gesamten Anwendung vor und wendet auf das resultierende Modell strukturelle Testverfahren an. Sinnvollerweise sollte zunächst die strukturelle Überdeckung (Codeüberdeckung oder Überdeckung der durch die Verfolgbarkeit identifizierten Entwurfselemente) während der anforderungs- und performanzbasierten Systemtests gemessen werden. Danach identifiziert der Tester die Teile der Anwendung, für die weitere Testfälle benötigt werden.

Der Abnahmetest soll nach Hetzel zeigen, dass die Anwendung bereit bzw. geeignet für den Einsatz in der Produktion ist. Hierbei fokussieren die Tester wieder auf die Anforderungen; der Abnahmetest wird in der Regel unter der Regie des Auftraggebers durchgeführt und soll diesen davon überzeugen, dass die Anwendung der Anforderungsspezifikation entspricht.

**Hsia und Kung** geben ein szenariobasiertes Verfahren zur Erstellung eines Benutzungsmodells (*usage model*) für den Abnahmetest an ([HsiKun97]). Aus der textuellen Anforderungsspezifikation werden manuell Sequenzen externer Ereignisse (*events*) zur Erfüllung einer funktionalen Anforderung modelliert. Diese Ereignissequenzen (*scenarios*) werden in einem Zustands-Übergangsdiagramm vereinigt, aus dem dann weitere Ereignissequenzen für den Abnahmetest generiert werden. Dieses Vorgehen ist analog zum statistischen Benutzungstest (*statistical usage testing*) (vgl. [Lyu96] [Oshana98][RegRun98]).

Zusammenfassend gesagt prüft der anforderungsbasierte Test die Anwendung gegen ihre Anforderungsspezifikation. Zusätzlich prüft der entwurfsbasierte Systemtest systematisch die Anwendung gegen den Entwurf. In dieser Arbeit konzentrieren wir uns auf diese beiden Arten von System- und Abnahmetests, da die Prüfung gegen nicht-funktionale Anforderungen wie z.B. der Performanz- und Stresstest oder der Benutzbarkeitstest in der Tat unabhängig von der Art der Anforderungsspezifikation bzw. der Entwicklungsmethode sind. Wir betrachten daher im Folgenden die bezüglich der Fragestellung „Test gegen die (funktionale) Anforderungsspezifikation“ relevanten Arbeiten.

### 3.4 Relevante Arbeiten: Test gegen objektorientierte Anforderungsspezifikationen

**Binder** skizziert eine Strategie zum Use Case-basierten Systemtest, welche jeden Use Case entsprechend seiner Benutzungshäufigkeit (*operational profile*) prüft ([Binder96b]). Er empfiehlt ähnlich wie Poston (s. Unten), die für den konkreten Ablauf eines Use Case erforderliche Information in sogenannten operationalen Variablen (*operational variables*) zu spezifizieren, die er dann in einer Art Entscheidungstabelle (*operational*

*relation*) zusammenfasst und zur Testfallableitung verwendet. Für rigorose Tests (*integrity verification*) schlägt Binder vor, die tatsächlichen Abläufe mit den in Interaktionsdiagrammen spezifizierten zu vergleichen (*thread coverage*).

**McGregor** gibt methodische Hinweise zur Planung des Tests gegen die (mit Use Cases spezifizierten) funktionalen Anforderungen [McGregor97]. Jedem Use Case werden ein Nutzungsprofil (*use profile*) sowie bestimmte Kritikalitäten zugeordnet, mit denen die Zuteilung von Ressourcen gesteuert wird. Konkrete Techniken zur Testfallermittlung werden nicht angegeben.

**Poston** beschreibt Möglichkeiten zur Generierung funktionaler Tests aus Signaturen der Operationen in Klassenmodellen ([Poston94]) sowie aus um Signaturen für die „Eingabeparameter“ erweiterten Sequenzdiagrammen zu Use Cases ([Poston98]). Beiden Ansätzen liegen Methoden des funktionalen Testens von Operationen zugrunde (Äquivalenzklassenbildung, Bedingungsüberdeckung, s. [Beizer90][Riedemann97]).

**Balzert** schreibt zum Systemtest:

Der Systemtest ist der abschließende Test der Software-Entwickler und Qualitätssicherer in der realen Umgebung [...] — ohne den Auftraggeber. [...] Im Unterschied zum Integrationstest ist beim Systemtest nur das „Äußere“ des Systems sichtbar, d.h. die Benutzungsoberfläche und andere externe Schnittstellen des Systems. [...] Basis für den Systemtest ist die Produktdefinition, bestehend aus Pflichtenheft, Produktmodell (z.B. OOA-Modell), Konzept der Benutzungsoberfläche und Benutzerhandbuch. [Balzert98]

Speziell zum sogenannten „Funktionstest“ rät Balzert dem Qualitätssicherer lediglich, zu überprüfen, ob alle in der Anforderungsspezifikation (Produktdefinition) geforderten Funktionen vorhanden und wie vorgesehen realisiert sind. Hierfür sollen aus dem Pflichtenheft die Testsequenzen übernommen und/oder mit funktionalen Testverfahren systematisch und vollständig hergeleitet werden. Liegt ein Produktmodell, z.B. in Form eines OOA-Modells vor, dann können aus diesem Modell die externen Operationen, d.h. die Operationen, die aus Benutzersicht aufrufbar sind, entnommen werden. Ist bei der Anforderungsspezifikation bereits das Konzept der Benutzungsoberfläche erstellt worden, dann können daraus auch die möglichen Funktionen für den Funktionstest ermittelt werden ([Balzert98]). Wie dies allerdings durchgeführt werden soll, wird nicht weiter geschildert.

**Strauß und Hörnke** geben einen Erfahrungsbericht zum Thema „Testen in grossen objektorientierten Projekten“ ([StrHör98]). Sie berichten hierbei vom Testen bei der Erstellung von Individualsoftware und erläutern die konkrete Umsetzung und Erweiterung von bekannten Methoden zur Testfallentwicklung anhand einer Spezifikation. Sie stellen die Testüberdeckung und die Testökonomie in den Vordergrund. Für den funktionalen Systemtest verwenden auch Strauß und Hörnke Entscheidungstabellen. Sie schreiben:

Nach unserer Erfahrung ist es nur mit den (späteren) Benutzern eines Systems — den fachlichen Experten — möglich, komplexe praxisrelevante Testfälle zu finden, ohne gleichzeitig eine Masse von nicht-praxisrelevanten Tests zu produzieren. Deshalb benötigen wir in unserem Kontext eine Darstellungsform, die auch von den Benutzern verstanden wird, damit wir diese in die Testfallentwicklung einbeziehen können. [StrHör98]

Allerdings treten bei der Verwendung der Entscheidungstabellen folgende Probleme auf:

- Die Erstellung konsistenter Testfälle ist nicht immer sichergestellt, d.h. die Kombination unverträglicher Testprüfpunkte lässt sich bei der Testfallbildung prinzipiell nicht vermeiden.
- Die Dokumentation von Abhängigkeiten zwischen Varianten verschiedener Testkriterien (Verfolgbarkeit) wird nicht besonders unterstützt, was für die Qualitätssicherung und die Pflege der Testfälle problematisch sein kann.

Zusätzlich geben Strauß und Hörnke einen Ansatz zur Automatisierung von Tests über die Benutzungsschnittstelle an, der jedoch deutliche Schwächen insbesondere beim Test solcher Anforderungen (Geschäftsvorfälle) hat, die unterschiedliche Eingabereihenfolgen unterstützen. Da in den erstellten Testskripten die Reihenfolge fest vorgegeben war, wurde ein „Mastertestskript“ jeweils kopiert und für die Reihenfolgetests manuell angepasst.

**Chang et al.** präsentieren einen Ansatz zum „high level test“ für objektorientierte Anwendungen, der auf einer formalen Spezifikation in Object-Z (vgl. [DKR+91][CarSto94]) und Nutzungsprofilen (*usage profiles*) basiert ([CLS+98]). Sie schreiben:

The behavior of a software system is specified in an object-oriented formal specification. A state model provides a complementary representation of the dynamic behavior. In the model, a state represents the cumulative results of the system behavior. Probability distributions are used to derive the anticipated operation sequences of a program from the state model. An enhanced state transition diagram (ESTD) is used to describe the state model, which incorporates hierarchy, usage and parameter information. This paper describes the construction of state transition diagrams (STDs) based on the formal specification, and the derivation of test scenarios from the ESTD. [CLS+98]

Einen Überblick über die Methode gibt Abb. 3.2. Die manuell auszuführenden Tätigkeiten sind durch die stilisierte ausführende Person gekennzeichnet, wobei insbesondere die Konstruktion der Zustandsautomaten aus den Object-Z Spezifikationen bisher nur manuell, unter Einsatz von Heuristiken möglich ist:

The derivation guideline [for the ESTD] requires extensive human effort. A subset of the Object-Z grammar is being investigated to study the possibility of constructing a parser to recognize schema dependency in the specification. This will be a major step towards a semi-automated tool for the ESTD derivation. [CLS+98]

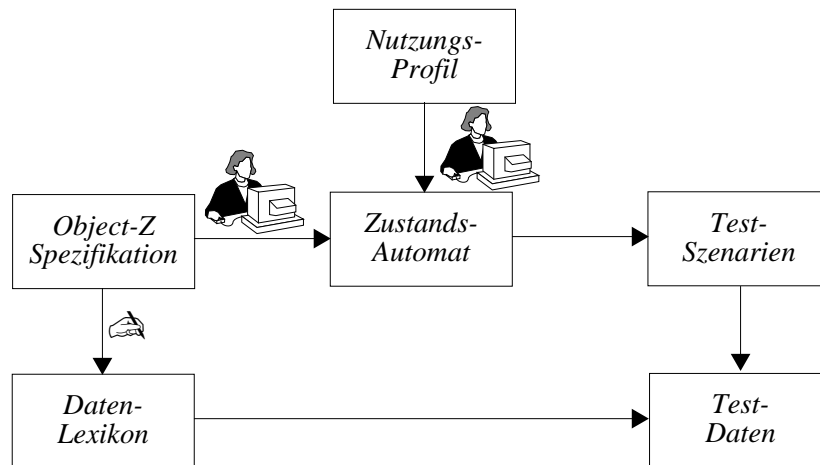


Abb. 3.2 Object-Z basierte Testmethode von [CLS+98]

Ähnlich zu dem weiter oben besprochenen herkömmlichen Verfahren von Hsia und Kung ([HsiKun97]) ist es bei dem Ansatz von Chang et al. problematisch, dass die Anforderungsspezifikation die möglichen Ereignisse bzw. Elemente der Benutzungsschnittstelle und somit Realisierungsdetails enthält. Zusätzlich enthalten die aus Zustandsautomaten und dem Datenlexikon generierten Testdaten nur „atomare“ Daten bzw. Ereignisse und berücksichtigen somit keine persistenten Objekte. Daneben ist die Schachtelung von Testfällen nicht möglich, da jede Sequenz einen vollständigen Pfad durch die Anwendung beschreibt.

Abb. 3.3 stellt die den bekannten Verfahren zugrundeliegende Vorgehensweise bei der Ausführung von Systemtests im grau hinterlegten Bereich dar — der Tester bzw. das Testwerkzeug stimulieren und validieren die Anwendung über die Ein- und Ausgabemöglichkeiten der (grafisch-interaktiven) Benutzungsoberfläche. Bei dieser Vorgehensweise steht der Tester bis jetzt vor unbeantworteten Fragen wie z. B.

- ☐ Welche konkreten Abläufe sind für jeden Use Case zu testen?
- ☐ Mit welchen von der Benutzungsoberfläche angebotenen Operationen ist eine gegebene Anforderung auszuführen und zu überprüfen?
- ☐ Wie ist die Anwendung zu initialisieren, d.h. wie kann der für die zu testende Anforderung erforderliche Zustand der Anwendung und/oder der persistenten Objekte hergestellt werden?
- ☐ Welche Operationen des Domänen-Klassenmodells entsprechen welcher Funktionalität — und welchen Operationen der Implementation?
- ☐ Wie kann der Zustand der Anwendung und/oder der persistenten Objekte nach der Testausführung beobachtet werden?

- In welchen Zuständen müssen sich die Anwendung und/oder die persistenten Objekte nach der Testausführung befinden?

Zusammenfassend halten wir fest, dass für die Qualitätssicherung in der objektorientierten Softwareentwicklung bisher größtenteils Verfahren für den (dynamischen) Klassentest entwickelt wurden. Beim funktionalen Test der Anwendung gegen die Anforderungsspezifikation bleiben Verfahren aus der prozeduralen Welt nur zum Teil relevant, denn objektorientierte Anwendungen müssen zunehmend gegen objektorientierte Anforderungsspezifikationen getestet werden. Zu statischen Prüfungen gibt es für objektorientierte Spezifikationen so gut wie keine methodische Unterstützung, so dass zusätzlich zu fehlenden Verfahren für den Test gegen objektorientierte Anforderungsspezifikationen auch ein Defizit bezüglich der Qualitätssicherung bei der Anforderungsermittlung selbst besteht. Wir beleuchten daher im nächsten Kapitel unter dem Gesichtspunkt der „Testbarkeit“ den Stand der Technik bei der Anforderungsermittlung — die „Use Case Analyse“.

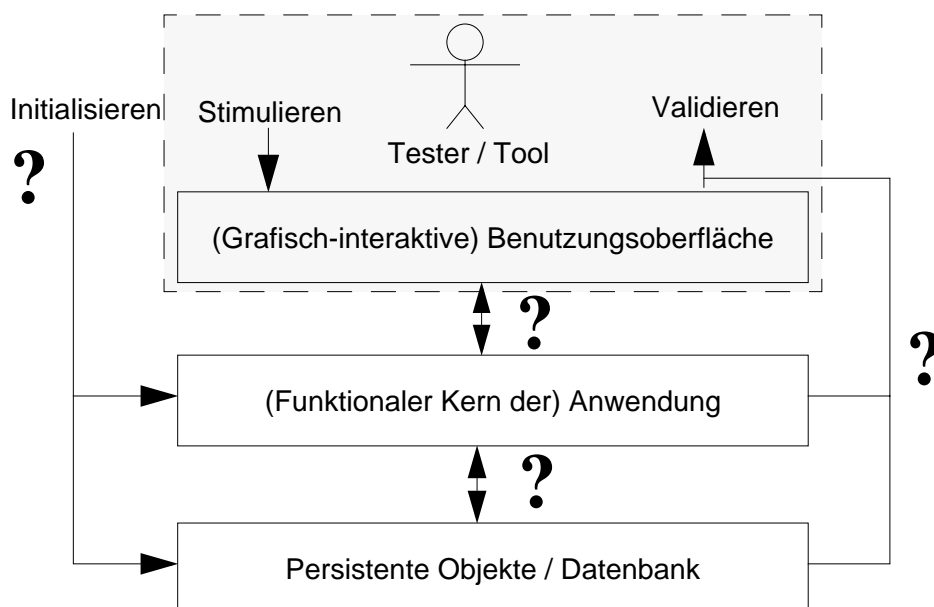


Abb. 3.3 Herkömmlicher (black-box) Systemtest

# Kapitel 4

## Probleme der Anforderungsermittlung mit Use Cases

*The hardest single part of building a system is deciding exactly what to build*  
[Brooks87]

Wir interessieren uns im Rahmen dieser Arbeit für die DV-Unterstützung bereits modellierter (relativ stabiler) Geschäftsprozesse. Unsere Betrachtungen zielen daher auf die Anforderungsermittlung für eine einzelne, neu zu entwickelnde Anwendung ab (*envisioned application*). Hierzu haben wir in Kapitel 2 bereits die Use Case Analyse nach Jacobson skizziert ([JCJ+92]). Anhand eines Fallbeispiels listen wir in diesem Kapitel einige Schwachpunkte des Use Case Konzepts in Bezug auf den Test gegen die Anforderungsspezifikation auf und diskutieren relevante Arbeiten zu ihrer Behebung.

### 4.1 Fallbeispiel Bankautomat

Als durchgehendes Anwendungsbeispiel dieser Arbeit betrachten wir die Arbeitsweise eines *Bankautomaten* (vgl. [WBW+90][Jacobson95]). Wir beginnen mit einer informellen Produktskizze:

An dem Bankautomaten können Kunden ohne die Beteiligung eines Bankangestellten bestimmte Transaktionen vornehmen. Vor der Eingabe einer Kreditkarte zeigt der Automat eine Eingabeaufforderung. Nach der Eingabe einer Kreditkarte wird diese gelesen. Nicht lesbare Karten werden eingezogen, nicht zum Zugriff berechtigende Karten direkt ausgeworfen. Ist die Karte lesbar und berechtigt zum Zugriff auf den Automaten, so fordert der Automat die Eingabe der Geheimzahl (PIN). Ist der Zentralrechner verfügbar, so wird die PIN dorthin übermittelt und von diesem geprüft, wobei direkt auch etwaige Sperrvermerke des Karteninhabers geprüft werden. Ansonsten prüft der Automat die PIN anhand der Kartendaten. Wird eine falsche PIN eingegeben, so wird dieser Versuch gezählt; nach drei Fehlversuchen wird die Karte gesperrt und ausgeworfen.

Nachdem die richtige PIN eingegeben wurde, werden verschiedene Verarbeitungsmöglichkeiten angezeigt (wie Kontostand anzeigen, Geld abheben, Geld einzahlen, Geld zwischen verschiedenen Konten transferieren etc.). Die Auswahl dieser Punkte verursacht einen Zugriff auf einen Zentralrechner zur Abfrage bzw. Veränderung der Kontostände. Ist der Zentralrechner nicht verfügbar, so kann nur Geld bis zu einem bestimmten Betrag ausgezahlt werden und die anderen Verarbeitungsmöglichkeiten sind gesperrt. Führt der Automat eine Aktion durch, so wird eine Beschäftigt-Anzeige ausgegeben. Jede ausgewählte Aktion kann durch die Abbruch-Taste gestoppt werden, Geldbeträge müssen mit der Bestätigungstaste endgültig festgelegt werden.

Soll Geld ausgezahlt werden, so muss zuvor die Kreditkarte entnommen werden. Anschließend wird das Geld ausgegeben (stets 100-Mark Scheine bis auf den letzten, dieser wird in kleineren Scheinen ausgegeben — z.B. 50,20,20,10). Nach Abschluss einer Transaktion druckt der Bankautomat einen Beleg mit bestimmten Informationen über die Transaktion und dem bzw. den aktuellen Kontoständen. Der Bankautomat wird regelmäßig von einem geschulten Bediener gewartet, d.h. es werden Ressourcen wie Geld und Druckerpapier nachgefüllt, schmutzgefährdete Teile gereinigt und etwaige Störungen beseitigt. Hierfür ist ein besonderer Bedienermodus vorzusehen.

Beginnen wir die Use Case Analyse nach Jacobson ([JCJ+92]). Aus der Produktskizze ermitteln wir die drei Akteure *Bankkunde*, *Bediener* und *Zentralrechner*. Wir betrachten zunächst den Akteur *Bankkunde* näher: Der Bankkunde muss sich mit seiner Kreditkarte und der Geheimnummer gegenüber dem Automaten identifizieren, bevor er eine der Transaktionen „Geld Abheben“, „Geld Einzahlen“ und „Geld Transferieren“ durchführen kann. Wir erhalten dementsprechend für den Akteur *Bankkunde* drei Use Cases. Für den Akteur *Bediener* modellieren wir zunächst einen Use Case Administration. Der (maschinelle) Akteur *Zentralrechner* partizipiert an den drei (den Transaktionen entsprechenden) Use Cases Geld Abheben, Geld Einzahlen und Geld Transferieren.

Spezifizieren wir beispielsweise den Use Case Geld Abheben textuell, so erhalten wir (ohne Anspruch auf Vollständigkeit) die in Abb. 4.1 dargestellten Haupt- und alternativen Abläufe. Bei der Spezifikation der weiteren Use Cases für den Akteur *Bankkunde* stellen wir fest, dass wir für jeden der drei Use Cases Geld Abheben, Geld Einzahlen und Geld Transferieren die Anmeldung des Bankkunden am Bankautomaten spezifizieren müssten. Da die Anmeldung für sich alleine den beobachtbaren Fortschritt „Angemeldet“ für den Akteur *Bankkunde* liefert, modellieren wir Sie als eigenständigen Use Case Anmelden, der von den anderen drei Use Cases benutzt wird — wir verwenden die uses-Beziehung<sup>1</sup>, wobei der Use Case Anmelden als *abstrakter Use Case* bezeichnet wird ([JCJ+92]). Das vollständige Use Case Diagramm zeigt Abb. 4.2.

Wenden wir uns dem strukturellen Modell zu. Ein rudimentäres Objektdiagramm für den Bankautomaten zeigt Abb. 4.3. Wir stellen uns den normalen Ablauf des Use Cases Geld Ab-

---

<sup>1</sup> Die Anmeldung alleine könnte allerdings z.B. der Validierung der Karte bzw. der PIN durch den Bankkunden dienen, so dass in diesem Falle auch der Einsatz der extends-Beziehung denkbar ist. Wir folgen in diesem Beispiel Jacobson (vgl. [Jacobson95]).



**USE CASE** Geld Abheben**Normaler Ablauf:**

- Der Bankautomat zeigt eine Begrüssungsmitteilung an
- Der Bankkunde gibt seine Karte ein
- Der Bankautomat liest den Code der Karte vom Magnetstreifen und prüft ihn auf Zulässigkeit
- Ist der Code zulässig, fragt der Bankautomat den Bankkunden nach der persönlichen Identifikationsnummer (PIN)
- Der Bankautomat wartet auf die Eingabe der PIN
- Der Bankkunde gibt seine PIN ein
- Ist die PIN korrekt, fragt der Bankautomat den Bankkunden nach der gewünschten Transaktion
- Der Bankautomat wartet auf die Eingabe der Transaktion
- Der Bankkunden wählt die Transaktion „Geld Abheben“ aus
- Der Bankautomat sendet die PIN zum Zentralrechner und fragt nach den Konten des Bankkunden
- Der Bankautomat zeigt die erhaltenen Kontonummern an und wartet auf die Auswahl eines Kontos durch den Bankkunden
- Der Bankkunde wählt eine Kontonummer aus und gibt den auszuszahlenden Betrag ein
- Der Bankautomat sendet die Kontonummer aus und den auszuszahlenden Betrag zum Zentralrechner
- Der Bankautomat präpariert die Banknoten zur Ausgabe
- Der Bankautomat wirft die Karte aus
- Der Bankkunde entnimmt die Karte
- Der Bankautomat druckt den Auszahlungsbeleg
- Der Bankautomat gibt die Banknoten aus
- Der Bankkunde entnimmt die Banknoten

**Alternative Abläufe:**

**Unlesbare Karte:** Ist der Code der Karte unlesbar, so zieht der Bankautomat die Karte mit einem Alarmton ein

**Unzulässige Karte:** Ist der Typ der Karte falsch, so wirft der Bankautomat die Karte aus

**Falsche PIN:** Ist die eingegebene PIN falsch, so wird dieser Versuch gezählt; nach drei Fehlversuchen wird die Karte gesperrt und ausgeworfen

**Unzulässige Transaktion:** Erklärt der Zentralrechner die angeforderte Transaktion für unzulässig, wird eine dementsprechende Meldung ausgegeben und die Karte ausgeworfen

**Abbruch durch den Bankkunden:** Der Bankkunde kann die laufende Transaktion jederzeit abbrechen, was zum Auswurf der Karte und Abbruch der begonnenen Transaktion führt

*Abb. 4.1 Use Case Geld Abheben: Textuelle Spezifikation (übersetzt aus [JCJ+92])*

heben vor. An diesem Ablauf partizipieren die in Abb. 4.3 gezeigten acht Objekte. Die ersten drei — **Kartenleser**, **Transaktion** und **Bedienpult** — sind die am Use Case Anmelden beteiligten Objekte (der vom Use Case Geld Abheben über die uses-Beziehung benutzt wird). An letzterem sind die anderen fünf — **Auszahlung**, **ZentralrechnerSchnittstelle**, **Belegdrucker**, **Geldausgabe** und **Bargeld** — beteiligt.

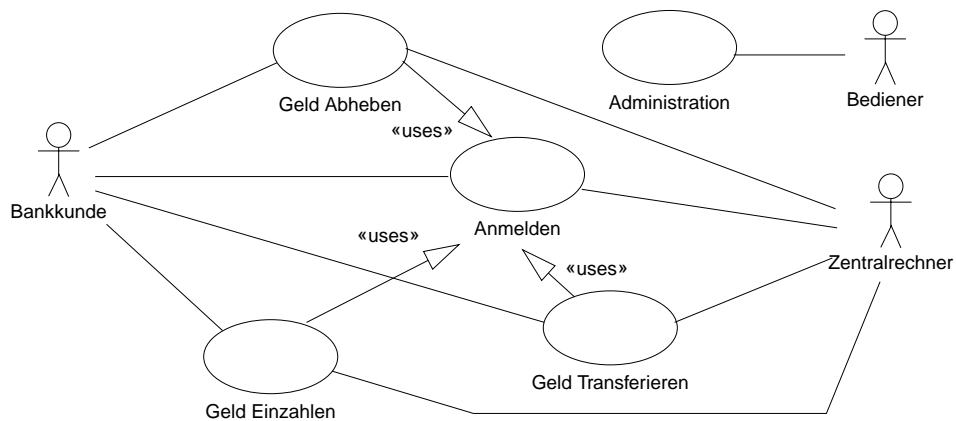


Abb. 4.2 Use Case Diagramm „Bankautomat“

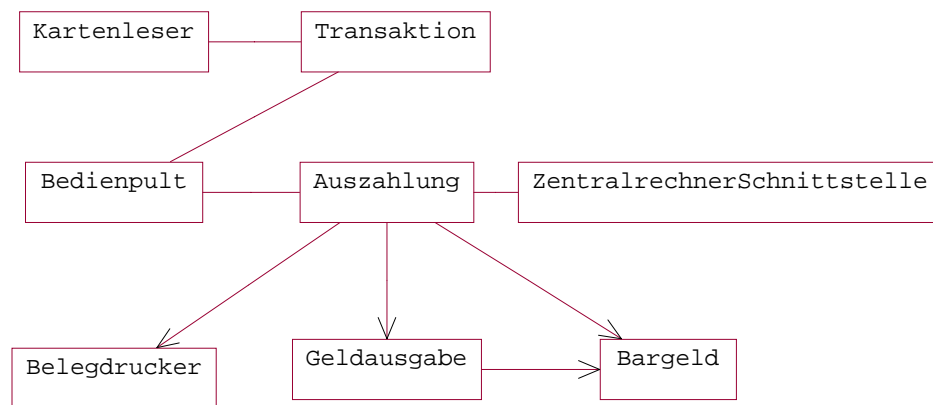


Abb. 4.3 Vereinfachtes Objektdiagramm zum Use Case Geld Abheben

Ein Interaktionsdiagramm für den normalen Ablauf des Use Cases Geld Abheben zeigt Abb. 4.4. Im Diagramm sind fast alle Ereignisse direkt durch eine Operation an der Benutzungsschnittstelle (Klasse **Bedienpult**) ausgelöst und keine komplexen internen Abläufe modelliert.

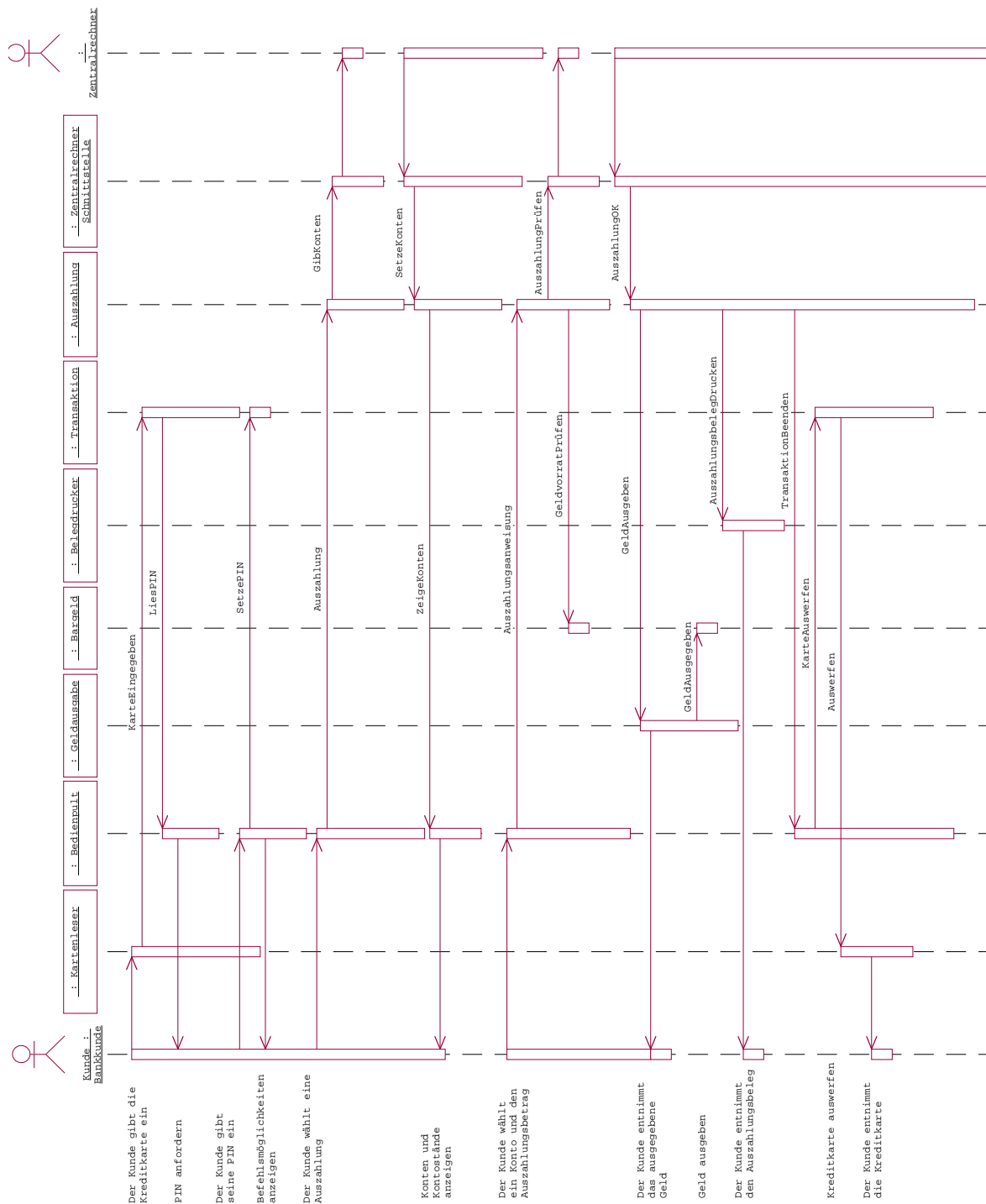


Abb. 4.4 Interaktionsdiagramm zum Use Case Geld Abheben

## 4.2 Offene Fragen und Kritik

Nach der begeisterten Aufnahme des Use Case Konzepts in der Fachwelt traten in der Praxis einige Mängel zutage, die Meyer veranlassen zu behaupten:

Except with a very experienced design team (having built several successful systems of several thousand classes each in a pure OO language), do not rely on use cases as a tool for object-oriented analysis and design. [Meyer97]

Wir listen diese Mängel getrennt bezüglich der Ausdrucksstärke und Anwendbarkeit von Use Cases, ihrer Struktur und ihrer Anwendbarkeit im Test gegen die Anforderungsspezifikation auf.

### Ausdrucksstärke und Anwendbarkeit

Modelle für Anforderungsspezifikationen sollten die in Abb. 4.5 gezeigten drei „Informationsarten“ erfassen können (vgl. [PohHau97][Paech98]):

- ❑ *kontextuelle Information* beschreibt das Umfeld der Anwendung, also z.B. Teilbereiche von Unternehmen und Anknüpfungspunkte an Geschäftsprozesse bzw. Workflows,
- ❑ *Interaktionsinformation* beschreibt die Interaktionen der Anwendung mit Benutzern oder benachbarten Systemen (externe Interaktionen) und

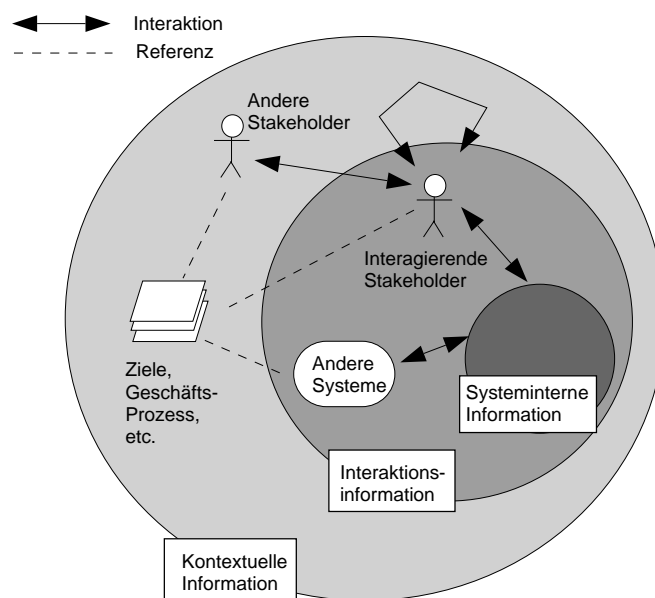


Abb. 4.5 Die drei Informationsarten bei der Anforderungsermittlung ([PohHau97])

- *systeminterne Information* beschreibt Interna der Anwendung selbst (Zustände von Domänenklassen, interne Interaktionen zwischen Instanzen von Domänenklassen).

Jacobsons Use Case Analyse beinhaltet im Wesentlichen Interaktionsinformation, die in erster Linie das Zusammenspiel der Anwendung mit den *Stakeholdern* (von der Anwendung betroffene Personen, „menschliche“ Aktoren) und anderen Anwendungen (maschinelle Aktoren) beschreibt (vgl. [PohHau97]). Kontextuelle Information wie z.B. das Zusammenspiel der Stakeholder untereinander sowie mit anderen Anwendungen wird nicht in Use Cases erfasst:

A use case is [...] manifested by sequences of messages exchanged among the system and one or more outside interactors (called actors) together with the actions performed by the system.  
[OMG99] (Notation Guide)

Durch die Beschränkung auf die Interaktionsinformation kann der Analytiker bei der Use Case Modellierung nach Jacobson Teile der Anforderungen wie z.B. die Einbettung der Anwendung in ihren Kontext oder die von einem Use Case betroffenen Teile der Anwendung bzw. des strukturellen Modells höchstens als Kommentar angeben. Aufgrund des relativ hohen Abstraktionsgrads tritt insbesondere die Problematik auf, dass die Funktionalität eines Use Cases sich nicht unmittelbar auf einzelne Domänenklassen und dann über den Entwurf in die Implementation verfolgen lässt. Dies hat zu der Ansicht geführt, dass solche Funktionalität ganz vom Klassenmodell getrennt und auf „freie“ Operationen ([Lauesen98]) bzw. Kontroll-Objekte ([JCJ+92]) abgebildet wird; sicherlich eine Ansicht, die von vielen Vertretern der objektorientierten Softwareentwicklung nicht geteilt wird (vgl. [CoaYou91][Meyer97][WalNer95][WBW+90]).

## Struktur

Betrachten wir als nächstes den Zusammenhang zwischen einem Use Case als „Klasse“ und einem Szenario als Use Case „Instanz“ (vgl. [JCJ+92]). Ein Use Case soll alle Szenarien ausdrücken, also alle möglichen konkreten Abläufe zur Erledigung der Aufgabe unter Benutzung der Anwendung. Jacobson sieht jedoch das „Instanzieren“ eines Use Cases im Wesentlichen als die Überführung der textuellen Spezifikation in jeweils ein Interaktionsdiagramm für den normalen Ablauf und die alternativen Abläufe. Insofern müssen alle möglichen Abläufe auch textuell spezifiziert sein, die textuelle Spezifikation beinhaltet also dieselbe Information wie die Menge aller Interaktionsdiagramme bzw. Szenarien. Ein Interaktionsdiagramm spezifiziert genau eine konkrete „Ausführung“ des Use Cases. Auch eine Menge von Interaktionsdiagrammen kann nicht alle möglichen Abläufe erfassen (vgl. [PTA94][RAB96]).

Zudem repräsentieren viele Use Cases komplexe Aufgaben. Die Zerlegung solcher Aufgaben erfolgt über *uses-* und *extends-*Beziehungen zwischen Use Cases. Diese Beziehungen lassen sich nicht auf die Interaktionsdiagramme fortsetzen, so dass letztere entweder unvollständig

sind oder aber bei der Darstellung des vollständigen Ablaufs schnell unübersichtlich werden (siehe z.B. Abb. 4.4). Dementsprechend sollte eine geeignete Use Case-Verfeinerung Bausteine zur Modellierung des Kontrollflusses beinhalten und alle möglichen Abläufe in einem Diagramm ausdrücken können.

In der Anforderungsspezifikation wird nicht nur spezifiziert, welche Funktionalität benötigt wird, sondern auch wann bzw. in welchen *Reihenfolgen* und mit welchen *Auswirkungen*. Hierzu werden z.B. Datenflussdiagramme (s. z.B. [Yourdon89][RBP+91][PagSix94]) oder Informationsfluss- bzw. Kooperationsdiagramme ([KWR96][Hasselbring98]) verwendet. Die textuelle Spezifikation von Use Cases nach Jacobson sowie Interaktionsdiagramme erscheinen hierzu nicht ausreichend.

Kommen wir zu dem von Jacobson vorgeschlagenen „Metamodell“ der Use Cases. Insbesondere die unklare bzw. fehlende Semantik der von Jacobson angegebenen *uses*- und *extends*-Beziehungen zwischen Use Cases führte zu nicht unerheblicher Konfusion sowohl bei Benutzern als auch Analytikern. Rumbaugh merkt an:

I am not convinced that Jacobsons *extends* and *uses* relationships are fundamentally different. [...] Both can be treated as special cases of a directed *adds* relationship between a base case and an additional case. [Rumbaugh94]

Diese Überzeugung hat jedoch nicht dazu geführt, dass die entsprechenden Definitionen in der UML geändert wurden. Alternative Abläufe sowie erfolgreiche und nicht erfolgreiche Szenarien sind weiterhin nur schwer in der Use Case Beschreibung identifizierbar (vgl. [Cockburn97][Hurlbut98]). Cockburn berichtet dementsprechend aus seiner Beratungspraxis von mehr als 18 verschiedenen Interpretationen bzw. Arten von Use Case Modellierungen ([Cockburn97]). Cockburns Fazit:

...people are using whatever they think of as a use case to good effect. [Cockburn97].

Das von Jacobson für die Modellierung struktureller Anforderungen vorgeschlagene Analysemodell entspricht eher dem Entity-Relationship Modell (vgl. z.B. [Yourdon89]) als denn einem voll ausgeprägten Klassenmodell. Dadurch bleiben die Auswirkungen eines Use Cases unklar. Im Extremfall wird für jedes Interaktionsdiagramm eine andere Objektkonstellation zugrundegelegt, was die Validierung des gesamten Use Cases verkompliziert. Die Aufteilung der oft verkapselten komplexen Funktionalität auf Schnittstellen-, Entitäts- und Kontroll-Objekte ist im Modell nicht nachzuvollziehen, da Jacobson keine „innere Struktur“ für Use Cases vorschlägt.

## Anwendbarkeit im Test gegen die Anforderungsspezifikation

Wie unterstützen Jacobsons Use Cases nun den Test gegen die Anforderungsspezifikation? Jacobson schreibt:

In the use case test you perform an operation test of the basic courses of the use case, that is, the expected flow of events. [...] The tests of the odd courses are all other flows of events. [...] Normally we test all use cases one by one. Those use cases with an extends association are tested after testing the use cases where it is to be inserted. [...] When we test the system the use cases should be tested in parallel, both in step and out of step. We should also stress the system by running several use cases at the same time. [...] Each test is now divided into subtests with different conditions. We can consequently decompose the test hierarchically. [JCJ+92]

In einer empirischen Untersuchung über den aktuellen Stand der Verwendung von Szenarien in fünfzehn industriellen Projekten stellen Weidenhaupt et al. zunächst Folgendes fest:

[...] that while many companies express interest on Jacobson's use case approach, actual scenario<sup>1</sup> usage often falls outside what is described in textbooks and standard methodologies. [WPJ+98]

Alle befragten Entwickler sehen die Notwendigkeit, Systemtests auf der Grundlage von Szenarien zu erstellen und damit dem Benutzer zu demonstrieren, dass die entwickelte Anwendung die Anforderungen erfüllt. Allerdings sieht die Praxis anders aus. Weidenhaupt et. al. schreiben hierzu:

We found that current practice rarely satisfies this demand. Often, the scenarios<sup>1</sup> developed during requirements engineering and system design were out of date by the time system testing began. Most projects therefore lacked a systematic approach for defining test cases based on scenarios. [WPJ+98]

Tests gegen die Anforderungsspezifikation nur durch manuelles „Nachspielen“ der Szenarien bzw. der in Interaktionsdiagrammen modellierten Abläufe sind einerseits schwer zu automatisieren. Der Tester steht vor Fragen wie „was sind die zulässigen Eingabewerte?“, „wie sieht der Anwendungszustand vor dem Szenario aus (inkl. der persistenten Objekte)?“. Andererseits erlaubt die fehlende Struktur textuell spezifizierter Use Cases nicht, objektive Testenkriterien zu formulieren. Die Frage, wann ein Use Case ausreichend getestet ist, bleibt unbeantwortet.

Bezüglich der Generierung von Testfällen aus Use Cases lassen wir zum Ende unserer Aufzählung noch einmal Poston zu Wort kommen:

However, use cases fall short of their intended purpose, because they do not provide enough information for test case or script generation. Testers must go outside use cases to find enough information to create test cases. [Poston98]

Im nächsten Abschnitt betrachten wir relevante Arbeiten zur Präzisierung von Use Cases bzw. der funktionalen Aspekte der Anforderungen.

---

<sup>1</sup> Anders wie bei Jacobson und der UML ([JCJ+92][OMG97]) umfasst das Konzept „Szenario“ bei Weidenhaupt et al. ebenso wie bei Carroll das des Use Cases ([WPJ+98][Carroll95]).

### 4.3 Relevante Arbeiten: Präzisierung von Use Cases

Aus der Fülle der Arbeiten greifen wir nun solche auf, die Ansätze zur Lösung eines der im letzten Abschnitt dargestellten Probleme bei der Anwendung des Use Case Konzepts geben.

**Potts et al.** stellen eine Methode zur Anforderungsermittlung vor, die insbesondere den zu bestimmten Anforderungen führenden Prozess unterstützen und dokumentieren soll ([PTA94]). Das zugrundeliegende *Inquiry Cycle Model* basiert auf einem Hypertext Framework, welches den laufenden Prozess der Anforderungsermittlung im Zyklus aus Anforderungsdokumentation, -diskussion und -evolution abbildet. Szenarien als Grundlage für Diskussionen zur Ermittlung, Konkretisierung und Validierung der Anforderungen werden auf zwei Komplexitäts- bzw. Abstraktionsebenen repräsentiert. Vollständige Szenarien (*complete scenarios*) unterstützen die sequentielle bzw. bedingte Ausführung von Teilszenarien; feiner aufgelöste Teilaktivitäten können als sogenannte Episoden in verschiedenen Szenarien wiederverwendet werden. Zusätzlich können Szenarien entweder mit Typen (Rollen, Klassen) oder Instanzen (konkreten Personen, Objekten) mehr oder weniger generisch beschrieben werden, wobei die Autoren sagen:

Introducing instances into scenarios has the drawback of doubling their size and possibly introducing irrelevant details but make scenarios more concrete, which may make requirements discussions easier and help resolve conflicts more quickly. [...] Scenario analysis and the inquiry cycle model complement each other. [PTA94]

Interessant ist die Verfolgbarkeit der Anforderungsspezifikation zu den Informationsquellen (Vor-Verfolgbarkeit, vgl. Kapitel 13). Hierzu werden Änderungen der Anforderungen mit ihren treibenden Fragen und Antworten verknüpft. „Vollständige Szenarien“ bei Potts et al. entsprechen Use Cases, „Episoden“ verkapseln komplexere Abläufe und dienen zur hierarchischen Strukturierung der Anforderungsspezifikation. Da Potts et al. die „Objekt-Identifikation“ eher als Entwurfstätigkeit sehen, werden strukturelle Aspekte nicht betrachtet.

**Cockburn** klassifiziert zunächst bekannte Use Case Modellierungsansätze anhand der vier Dimensionen Zweck (*purpose*), Inhalt (*contents*), Pluralität (*plurality*) und Struktur (*structure*) ([Cockburn97]). Er schlägt dann selbst ein hierarchisches Use Case Modell zur Anforderungsspezifikation vor (Zweck), in dem Use Cases normsprachlich modelliert (Inhalt), mehrere Szenarien in einem Use Case erfasst (Pluralität) und Use Cases semiformal strukturiert werden (Struktur). Die Struktur der Use Cases orientiert sich dabei an den mit dem Einsatz der Anwendung verfolgten Zielen (*goals*, vgl. [LamWil98]), wobei Cockburn für jeden Use Case erfolgreiche von nicht erfolgreichen Szenarien unterscheidet.



**Regnell et al.** entwickeln ein Use Case Modell mit drei Hierarchieebenen ([RAB96]). Die Umgebungsebene (*environmental level*) entspricht Jacobsons Use Case Modell, eine Zwischenebene (*structure level*) definiert zusammenhängende Teilabläufe eines Use Cases, genannt Episoden, deren Semantik über Vor- und Nachbedingungen spezifiziert wird. Episoden bei Regnell et al. sind als funktional hierarchisch strukturierte Teilabläufe am ehesten mit den von uns vorgestellten Makroschritten zu vergleichen und werden zeitlich über Konstrukte verknüpft, die an Message Sequence Charts (MSC, vgl. [ITU93]) angelehnt sind. Auf der Ereignissebene (*event level*) wird der Nachrichtenaustausch genau eines Aktors mit der Anwendung (betrachtet als Black Box) in zeitlicher Reihenfolge mittels MSC modelliert. Alle Use Cases eines Aktors können in einem sogenannten *Synthesized Usage Model* zusammengefasst werden, welches alle möglichen Interaktionen dieses Aktors mit der Anwendung darstellt (vgl. [RKW95]). Regnell et al. schreiben hierzu:

The synthesized usage model is an abstraction mechanism that conceals the detailed protocol of interactions. [RKW95]

Aufgrund des Detaillierungsgrads und der Beschränkung auf einen Akteur pro Use Case erscheint dieses Modell eher zugeschnitten auf die Spezifikation von Mensch-Maschine oder Maschine-Maschine Schnittstellen und die Ableitung von Testfällen für den statistischen Benutzungstest (*statistical usage testing*, vgl. [RegRun98]).

**Rolland und Achour** kombinieren die Ansätze von Regnell, Cockburn und Dano ([RAB96][Cockburn97][DBB97]) und entwickeln eine Methodik zur textuellen Spezifikation von Use Cases, die auf sogenannten linguistischen Strukturen basiert ([RolAch98]). Anhand dieser Strukturen wird die (umgangs-) sprachliche Beschreibung von Geschäftsvorfällen analysiert und strukturiert. Rolland und Achour schreiben:

A use case specification comprises contextual information of the use case, its change history, the complete graph of possible pathways, attached requirements and open issues. The proposed approach delivers a use case specification as an unambiguous natural language text. This is done by a stepwise and guided process which progressively transforms initial and partial natural language descriptions of scenarios into well structured, integrated use case specifications. [RolAch98]

**Lamsweerde und Willemet** beschreiben einen wissensbasierten Ansatz zur Inferenz deklarativer Anforderungsspezifikationen aus textuellen Beschreibungen von Szenarien ([LamWil98]). Sie verfolgen ebenso wie Rolland und Achour das Ziel einer konsistenten und vollständigen (textuellen) Spezifikation der Use Cases:

Scenarios are in general partial, procedural, and leave required properties about the intended system implicit. In the end such properties need to be stated in explicit, declarative terms for consistency/completeness analysis to be carried out. A formal method is proposed for supporting the process of inferring specifications of system goals and requirements inductively from interaction

scenarios provided by stakeholders. The method is based on a learning algorithm that takes scenarios as examples/counterexamples and generates a set of goal specifications in temporal logic that covers all positive scenarios while excluding all negative ones. [LamWil98]

Beide Ansätze können komplementär zu dieser Arbeit zur Extraktion und Konkretisierung sowie bei der textuellen Spezifikation von Use Cases benutzt werden.

**Hurlbut** erhebt Szenarien (im Sinne konkreter Abläufe eines Use Cases) zur Modellierungselementen und heftet sie an sogenannte adaptive Use Cases (*adaptive use cases*, [Hurlbut98]). Er orientiert sich hierbei an dem Workflow Referenzmodell (vgl. [WFMC97]) und fokussiert dementsprechend auf Geschäftsprozesse. Innerhalb eines Szenarios können Erweiterungspunkte definiert werden, an denen alternative Abläufe bzw. Ausnahme- und Sonderabläufe eingeschoben werden können. Eine klare Semantik für die Szenarien und die Einschübe ist nicht angegeben.

**Kösters, Six und Voss** konzentrieren sich auf die modellbasierte Entwicklung von Programmen mit ausgeprägter (grafischer) Benutzungsoberfläche und geben mit FLUID eine Vorgehensweise zur kombinierten Ermittlung von Domänen- und Benutzungsschnittstellenanforderungen an ([KSV96]), die z. Zt. von Homrighausen und Voss für den Entwurf erweitert wird ([HomVos97]). Kern von FLUID sind vier miteinander gekoppelte Modelle:

- Das *Aufgabenmodell* klärt die Frage, was ein Benutzer mit der Anwendung bewerkstelligen will. Zur Strukturierung werden Aufgaben in Teilaufgaben zerlegt sowie Wechsel zwischen logisch oder organisatorisch zusammengehörigen Aufgaben modelliert.
- Zur Problemweltmodellierung wird die objektorientierten Analyse ([CoaYou90]) mit dem Ergebnis eines *Domänen-Klassenmodells* verwendet.
- Anforderungen an die Benutzungsschnittstelle werden im *Oberflächen-Analysemodell* (*user interface analysis model*, UIA-Modell) festgehalten, wobei hauptsächlich das Aufgabenmodell in einer auf direkt-manipulative Oberflächen zugeschnittenen Weise zu einer verbindlichen Vorgabe für die weitere Entwicklung präzisiert wird.
- Semantische Anteile der Benutzungsschnittstelle und die Vermittlung zwischen ihren Objekten und dem funktionalen Kern der Anwendung werden im *Oberflächen-Entwurfsmodell* (*user interface design model*, UID-Modell) spezifiziert. Hier werden auch wesentliche Teile der Architektur der Anwendung festgelegt.

Auch **Rosson** beschreibt, wie die ineinander verwobene Analyse von Benutzungs- und sogenannten „Objekt-Interaktions-Szenarien“ das Verständnis der gegenseitigen Beschränkungen der Anforderungen und des Software-Entwurfs verbessern kann ([Rosson99]). Wir lesen:

User tasks and the software designed to support them are independent. Tasks set requirements for new systems; as systems are developed, their software and hardware characteristics create constraints or opportunities for the tasks. The paper argues for a more direct integration of the design of user tasks and their corresponding software design, an integration provided through object-oriented analysis and design (OOAD) of user interaction scenarios. [Rosson99]

Beide Ansätze ergänzen einerseits die Spezifikation der Oberfläche und ermöglichen andererseits die Nach-Verfolgung der Domänenklassen der Anforderungsspezifikation zu den Entwurfsklassen. Wir sprechen den zweiten Aspekt im Verlauf der Arbeit noch an.

**Glinz** entwickelt ein auf dem Zustandsdiagramm basierendes integriertes, formales Modell zur ausführbaren Spezifikation und Komposition von Szenarien ([Glinz95]). Ein Szenario ist als ein Statechart (mit genau einem Ein- und Ausgang) modelliert. Glinz gibt Regeln zur Komposition von Statecharts in ein integriertes Modell an, welches das Verhalten der Anwendung modelliert. Aufbauend auf der Semantik für Statecharts (vgl. [HarNaa96]) und den Kompositionsregeln leitet Glinz die Semantik des integrierten Szenarienmodells ab und gibt Regeln zu seiner Analyse und Verifikation z.B. in Hinblick auf deadlocks, Konsistenz und Vollständigkeit an. Abschließend skizziert Glinz, wie Klassen- bzw. Objektmodelle in das Szenarienmodell integriert werden könnten. Aufgrund der Einschränkung auf die systeminterne Information und des Detaillierungsgrades erscheinen die Modelle eher für die Entwurfstätigkeiten als zur Anforderungsermittlung geeignet. Die vorgeschlagene Integration von Klassen- bzw. Objektmodellen in das Szenarienmodell stellt Objekte als Zustände bzw. Statecharts dar und ist insbesondere zur Generierung von Prototypen vorgesehen.

**Ziegler** entwickelt basierend auf der Vereinigung von Aufgabenhierarchien und Zustandsmaschinen eine Vorgehensweise zum Entwurf von Anwendungen mit grafischen Benutzungsschnittstellen ([Ziegler96]). Zur Modellierung von Aufgaben und Prozessen führt Ziegler die sog. *Task Object Charts* ein, welche besondere Darstellungsmittel für Aufgabenhierarchien, nebenläufige Bearbeitung und flexible Aufgabenstellungen bereitstellt. Durch die Verwendung von Objektzuständen zur Steuerung von Aufgabenabläufen wird eine Kopplung zwischen Objekt- und Aufgabenmodell angestrebt, mit welcher der Entwurf von Dialogstrukturen in systematischer Weise abgeleitet wird.

**Züllighoven** beschreibt einen Leitbild-orientierten, hauptsächlich mit den Metaphern „Werkzeug“, „Automat“ und „Material“ arbeitenden Ansatz zur objektorientierten Anwendungsentwicklung ([Züllighoven98]). Zentrale Idee dieses Ansatzes ist,

... die Gegenstände und Konzepte des Anwendungsbereichs als Grundlage des softwaretechnischen Modells zu nehmen. Das Ergebnis soll eine enge Korrespondenz zwischen dem anwendungsfachlichen Begriffsgebäude und der Softwarearchitektur sein. Diese Strukturähnlichkeit hat zwei entscheidende Vorteile: Die Anwender finden die Gegenstände ihrer Arbeit und die Begriffe ihrer Fachsprache im Anwendungssystem repräsentiert. Sie können entsprechend ihre Arbeit in gewohnter Weise organisieren. Die Entwickler können Softwarekomponenten und Anwendungs-

konzepte bei fachlichen und softwaretechnischen Änderungen zueinander in Beziehung setzen und somit die wechselseitigen Abhängigkeiten erkennen. [Züllighoven98]

Nach unserer Begriffsbildung deckt der Ansatz von Züllighoven die funktionale Sicht (Werkzeuge und Automaten), die Informationssicht (Materialien) und — mit dem Leitbild des Arbeitsplatzes — auch Teile der organisatorischen sowie der Leistungssicht ab. Als fachliches Modell zur Ist-Analyse dient dabei in erster Linie das Datenwörterbuch, aus dem sich die Objekte des Anwendungsbereichs herauskristallisieren sollen. Diese werden dann — ggf. nach Änderungen am Geschäftsprozess — in Domänenklassen transformiert bzw. auf solche abgebildet, welche das softwaretechnische Modell bilden und später im Rechner realisiert werden sollen. Die Ablauf- oder Verhaltenssicht ist nicht abgedeckt.

Fassen wir zusammen: Während die rein textuelle Formulierung der Use Cases bei Cockburn (s. [Cockburn97]) die Validierung erschwert, ist die von Regnell et al. (s. [RAB96]) schon auf relativ hoher Abstraktionsebene verwendete MSC-Notation zu sehr auf Anwendungsgebiete wie z.B. die Telekommunikation beschränkt. Beide Arbeiten vernachlässigen genauso wie Jacobson ([JGJ97]), Potts ([PTA94]), Rolland und Achour ([RolAch98]), Lamsweerde und Willemet ([LamWil98]) und Hurlbut ([Hurlbut98]) die strukturellen Anteile der Anforderungsdokumente.

Aufgrund der unterschiedlichen Ansätze lassen sich die meisten Arbeiten nicht kombinieren (vgl. [Carroll95][PohHau97]). Eine Ausnahme bilden die auf Zustandsmaschinen (state charts) basierenden Arbeiten von Glinz und Ziegler. Die implizite Hypothese, Funktionalität und Verhalten kompletter Anwendungen unter Verwendung lediglich einer einzigen (neuen? formalen?) Notation spezifizieren zu können, gerät jedoch immer mehr in das Kreuzfeuer von Kritikern (vgl. [FMD97][Wiegers98]):

- ☐ Keines der bekannten monolithischen Modelle kann alle Aspekte realer Anwendungen erfassen.
- ☐ Je komplexer die Modelle werden, desto weniger wahrscheinlich ist die Möglichkeit, Anwendungen (realer Größe) damit vollständig spezifizieren zu können und
- ☐ desto weniger werden Analytiker und Entwickler solche Modelle lesen, geschweige denn erstellen können.

Die angesprochenen Probleme sowie weitere Forderungen an Anforderungsspezifikationen werden in Teil II dieser Arbeit angegangen.

## 4.4 Zwischenfazit

Wir haben in Abschnitt 1.2 gesehen, dass in der industriellen Praxis hauptsächlich in den „späten“ Phasen der Softwareentwicklung — und dort relativ unsystematisch — geprüft wird. Zur Verbesserung dieser Situation sind die folgenden Problematiken zu behandeln:

- ❑ Es fehlen Verfahren zur Qualitätssicherung bei der „objektorientierten“ Anforderungsermittlung.
- ❑ In der Literatur sind hauptsächlich Verfahren zur Generierung von Testfällen und Testdaten aus formalen Spezifikationen angegeben.
- ❑ Der Test von Reihenfolgebedingungen [Riedemann97] ist problematisch, da die meisten Techniken zur Spezifikation auf rein „funktionale“ Spezifikationen hinauslaufen.
- ❑ Die Komplexität objektorientierter Anwendungen liegt in den Interaktionen (vgl. Anhang A.2), so dass die in der Praxis oft verwendeten white-box Testdekriterien wenig aussagekräftig sind. Benötigt werden neue Testkriterien zur Abdeckung der Interaktionen sowie neue, darauf abzielende Testverfahren.
- ❑ Bekannte Verfahren zur Qualitätssicherung bei der Anforderungsermittlung und zum (System- und Abnahme-) Test gegen die Spezifikation beziehen sich auf „herkömmliche“ Techniken und Modelle zur Spezifikation. Daher sind diese Verfahren bei der Prüfung der Anwendung gegen eine objektorientierte Spezifikation nicht anwendbar.

Grundsätzlich gilt, dass die Produkte aller Tätigkeiten bei der Softwareentwicklung einerseits selbst qualitätsgesichert und andererseits für den Test abhängiger Produkte herangezogen werden müssen. Mit Blick auf Abb. 1.2 (s. Seite 10) argumentieren wir folgendermaßen für die Auswahl der Qualitätssicherung bei der Anforderungsermittlung und den (System- und Abnahme-) Test gegen die Spezifikation als Inhalt dieser Arbeit:

- ❑ Die Entwicklung und damit die qualitätssichernden Tätigkeiten beginnen mit der Anforderungsermittlung bzw. der Prüfung der Spezifikation.
- ❑ Aufgrund des Kosten/Nutzen-Verhältnisses (Entstehungsort, Aufdeckungswahrscheinlichkeit und Behebungskosten von Fehlern) versprechen die Anforderungsermittlung und darauf aufbauenden Tests gegen die Spezifikation die größte Effizienz und Akzeptanz in der industriellen Praxis (Tab. 2.1, s. Seite 18).

Die grundsätzliche Fragestellung dieser Arbeit lautet somit:

***„Wie wird eine objektorientierte, Use Case-basierte Spezifikation geprüft bzw. erst prüfbar gestaltet und wie können Testfälle für den Test gegen die geprüfte Spezifikation abgeleitet werden?“***



# Teil II

## Anforderungsermittlung

In diesem Teil stellen wir die Konzepte von SCORES vor (Systematic Coupling of Requirements Specifications), unserer qualitätszentrierten Methode zur Anforderungsermittlung. SCORES präzisiert und verfeinert das Use Case Konzept und ermöglicht die direkte Kopplung mit dem Klassenmodell.

In den Kapiteln 5 und 6 führen wir die Konzepte und die Semantik von Use Case Schrittgraphen ein. Als zentrales Modellierungskonzept von SCORES verschmelzen Use Case Schrittgraphen funktionale und ablaufbezogene Aspekte der Anforderungen. Szenarien fügen sich nahtlos in die resultierende kombinierte Sicht ein, die dann in Kapitel 7 mit den strukturellen Aspekten der Anforderungen — dem Klassenmodell — gekoppelt wird.

In Kapitel 8 geben wir methodische Hinweise zum Vorgehen bei der Anforderungsermittlung mit SCORES.

# Kapitel 5

## Grundlegende Konzepte

Wir stellen zu Beginn dieses Kapitels die Ziele zusammen, die wir mit der in diesem Teil beschriebenen Präzisierung und Verfeinerung des Use Case Konzepts erreichen wollen. Zunächst einmal formulieren wir einige Ziele bezüglich der Ausdrucksstärke.

- ❑ Use Cases werden hauptsächlich statisch, funktional beschrieben. Damit konkrete Szenarien hinlänglich spezifizierbar sind, müssen die Erweiterungen ablaufbezogene „Kontrollflussaspekte“ abdecken (vgl. [PTA94][RAB96]).
- ❑ Zur Redundanzvermeidung wird eine hierarchische Strukturierung der Use Cases gefordert (vgl. [Cockburn97][JGJ97][RAB96]).
- ❑ Die Beziehungen zwischen Use Cases (uses, extends, Generalisierung) müssen auf der Verfeinerung fortsetzbar sein.
- ❑ Das resultierende Modell muss systeminterne, interaktions- und kontextuelle Information ausdrücken können ([PohHau97], vgl. Abb. 4.5).
- ❑ Das Klassenmodell beschreibt die Domänenobjekte, welche Verantwortlichkeiten zur Bearbeitung der Use Cases haben, in Hinblick auf ihre anwendungsinterne Darstellung (vgl. [Meyer97][WBW+90]). Daher kommt der systeminternen Information besondere Bedeutung zu.
- ❑ Die Stärken des ursprünglichen Ansatzes — wenige Metaobjekte, leichtverständliche grafische Darstellung, Konzentration auf das Wesentliche — müssen für die Anwendung des resultierenden Konzepts erhalten bleiben, da sie die Hauptsäule des herausragenden Erfolgs des Use Case Konzepts bilden.

Es folgen einige Ziele hinsichtlich der Verfolgbarkeit (vgl. [GotFin94]).

- ❑ Gefordert ist ein klarer, nachvollziehbarer Übergang zwischen funktionalen und strukturellen Aspekten der Anforderungsspezifikation, also zwischen Use Cases und dem Klassenmodell (s. auch Abb. 5.1).



- Darüber hinaus müssen die Elemente der Anforderungsspezifikation über den Entwurf in die Implementation verfolgbar sein (vgl. [GotFin94]).

Zum Schluss unserer Aufzählung beziehen wir uns auf Aspekte der Qualitätssicherung.

- Damit die weitestgehende Validierung der Anforderungen durch die Benutzer (vgl. [Boehm84][Gerrard94][MMM+97][PTA94]) gewährleistet ist, muss das resultierende Modell die Ableitung benutzerfreundlicher Sichten erlauben (vgl. [Hasselbring98][Züllighoven98]).
- Die in verschiedenen Modellen wie z.B. Use Cases und Klassenmodellen spezifizierten Aspekte der Anforderungen müssen gegeneinander geprüft werden können.
- Die Anforderungsspezifikation muss für den Übergang zu formalen Spezifikationen im Entwurf sowie zur Unterstützung von Testtätigkeiten ausreichend formal bzw. zumindest formalisierbar sein.
- Der Präzisierungsgrad bei der Anforderungsspezifikation muss z.B. die Ableitung von Testfällen und die Anwendung von Testendekriterien ermöglichen (s. z.B. [Myers79][Beizer90][Riedemann97]).
- Zur Automatisierung von Tests sind z.B. umgangssprachlich beschriebene Testfälle (z.B. [JCJ+92]) nicht ausreichend. Wir zitieren Riedemann:

Das spezifikations- und entwurfsorientierte Testen leidet darunter, dass meist eine formale Basis fehlt. Daher sind formale Spezifikations- und Entwurfsverfahren zu entwickeln bzw. für das Testen nutzbar zu machen, z.B. zum (möglichst) automatischen Generieren der Testfälle und Testdaten. [Riedemann97]

Wie wir in Kapitel 4 gezeigt haben, wurden einige dieser Anforderungen an ein Konzept zur Verfeinerung und Präzisierung von Use Cases bereits in anderen Arbeiten berücksichtigt. Allerdings deckt kein bekannter Ansatz sämtliche Anforderungen ab. Da die verschiedenen Ansätze zudem größtenteils nicht miteinander kombinierbar bzw. erweiterbar sind, versuchen

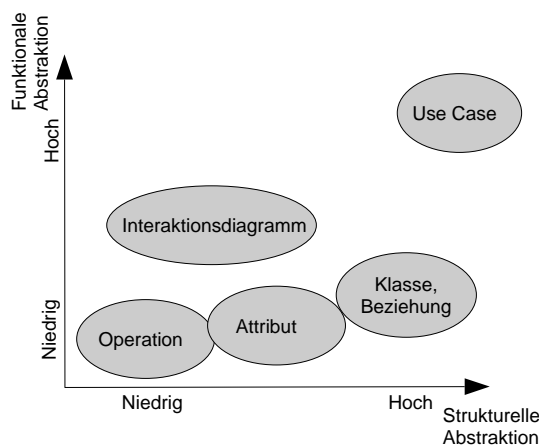


Abb. 5.1 Die Abstraktionslücke zwischen Use Case Modell und Klassenmodell

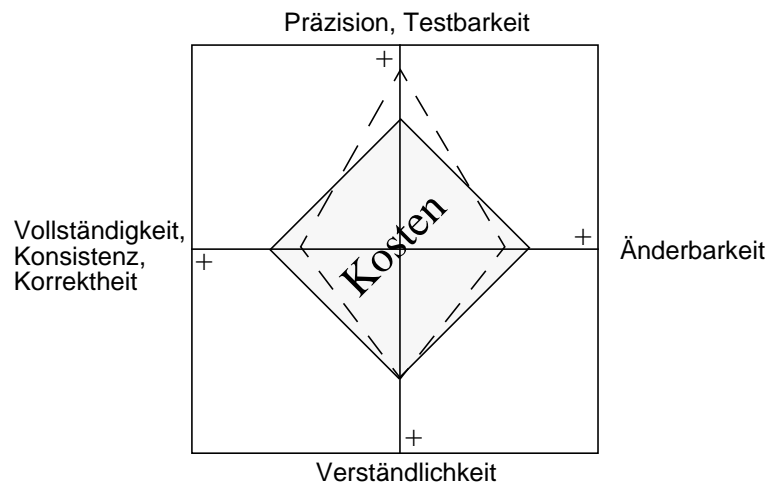


Abb. 5.2 Das Kosten-„Teufelsquadrat“ der Anforderungsermittlung

wir daher in den folgenden Kapiteln, die interessantesten Ideen in einem flexiblen, ausdrucksstarken Konzept zu vereinigen und mit eigenen Vorschlägen zur Erfüllung aller oben genannten Anforderungen zu vervollständigen.

Es bleibt zu Bemerkem, dass präzisere Anforderungsspezifikationen zunächst nicht „kostenneutral“ erstellt werden können. Abb. 5.2 zeigt z.B., wie bei gegensätzlichen Forderungen die Erhöhung der Präzision und Testbarkeit bei gleichbleibenden Kosten zu einer verminderten Erfüllung anderer Forderungen führt. Ein höherer Einsatz bei der Anforderungsermittlung ergibt jedoch eine besser oder überhaupt erst prüfbare Anforderungsspezifikation. Zusätzlich sind im weiteren Verlauf der Produktentwicklung gegen diesen Einsatz der „äußerliche Mehrwert“ z.B. einer höheren Gebrauchstauglichkeit und der „innerliche Mehrwert“ eines gegen die Anforderungsspezifikation prüfbaren Produkts sowie einfach ableitbarer oder — im Idealfall — generierbarer System- und Abnahmetestfälle aufzurechnen.

In Hinblick auf die Werkzeugunterstützung führen wir in diesem Kapitel ein Metamodell für die neu eingeführten Elemente ein, welches wir dann in den folgenden Kapiteln schrittweise erweitern. Die eher formale Darstellung soll die informelle Sicht auf Use Cases keinesfalls ersetzen, sondern lediglich präzisieren. Weitere Sichten wie z.B. die unerlässliche informelle Benutzersicht werden (automatisch) aus der werkzeuginternen Darstellung abgeleitet.

In diesem und den weiteren Kapiteln werden wir neue Konzepte jeweils mit kleinen, „kontextfreien“ Beispielen illustrieren. Daneben ergänzen wir am Schluss jedes Kapitels entsprechende Teile des Fallbeispiels „Bankautomat“.

## 5.1 Use Case Schritte

Wir verschmelzen in diesem Kapitel funktionale und ablaufbezogene Aspekte der Anforderungen zu einem verfeinerten Use Case Konzept, das wir dann mit den strukturellen Aspekten der Anforderungen — dem Domänen-Klassenmodell — koppeln, in dem Domänenklassen, ihre Attribute, Operationen und ihre Beziehungen untereinander modelliert werden (s. [OMG97]). Die dort spezifizierten Operationen stellen eine weitere funktionale Sicht dar; nun aber auf einer tieferen Abstraktionsstufe und oft ohne direkten Bezug auf die extern beobachtbaren Aufgaben und Abläufe.

Zunächst erweitern wir das aus Abschnitt 2.1 schon bekannte Use Case-Konzept um eine präzise Aufgabenspezifikation.

**Definition 5.1** Ein *Use Case* beschreibt die Ausführung einer wohldefinierten Aufgabe. Wir charakterisieren jeden Use Case durch

- ❑ einen im Anwendungskontext eindeutigen *Namen*,
- ❑ eine informelle *Beschreibung* der Aufgabe,
- ❑ eine Menge *A* von *Aktoren*, die bei der Bearbeitung des Use Cases mitwirken bzw. am Ergebnis des Use Cases interessiert sind,
- ❑ eine *Vor-* und eine *Nachbedingung*, welche die Annahmen bzw. das Ergebnis der Aufgabe des Use Cases kennzeichnen.

❑

Für die Menge der einem Use Case *uc* zugeordneten Aktoren schreiben wir auch  $A(uc)$ .

Als funktionales Konzept haben Use Cases kein „Gedächtnis“. Wir entleihen daher das Vertragskonzept der (objektorientierten) Programmierung (s. z.B. [Meyer97][WBW+90]) und spezifizieren die Aufgaben und insbesondere den Kontext bzw. Zustand, in dem sie ausgeführt werden, deskriptiv mit Vor- und Nachbedingungen über Domänen-Objekten bzw. den Gegenständen der Anwendung (vgl. [Züllighoven98]), wobei wir uns eng am Domänen-Klassenmodell anlehnen (vgl. Abschnitt 7.1 sowie [OMG97]). Die Vorbedingung gibt die minimalen Voraussetzungen an, mit der die im Schritt beschriebene Aufgabe begonnen werden kann. Die Nachbedingung gibt mögliche Auswirkungen bzw. das Ergebnis der Aufgabenbearbeitung an.

Zu Beginn der Anforderungsermittlung spezifizieren wir diese Bedingungen informell; in Hinblick auf eine automatische Auswertung bzw. Prüfung können wir sie dann in der Anforderungsspezifikation bis hin zu formalen mehrsortigen prädikatenlogischen oder (OCL<sup>1</sup>-) Ausdrücken über dem Klassenmodell präzisieren.

---

<sup>1</sup> OCL = Object Constraint Language, s. [OMG97].

*Beispiel 5.1.* Abb. 5.3 zeigt die Spezifikation für den Use Case Anmelden des Bankautomaten. Die verschiedenen, mit **ODER** verknüpften Bedingungen der Nachbedingung geben die möglichen Ausgänge bzw. Ergebnisse der Aufgabenbearbeitung an. In den einzelnen Bedingungen benutzen wir zunächst die im Verlauf der Bearbeitung gewonnene Information wie z.B. „Karte unlesbar“. Daneben erkennen wir auch bestimmte Zustände von Anwendungsobjekten (z.B. „Kartenleser gesperrt“ oder „PIN dreimal falsch eingegeben“).

**Use Case Anmelden In Modell Bankautomat**

**Beschreibung** Nach Eingabe der Karte durch den Bankkunde liest der Bankautomat den Code vom Magnetstreifen und prüft ihn auf Zulässigkeit. Bei zulässigem Code fordert der Bankautomat die Eingabe der Identifikationsnummer (PIN). Der Bankkunde gibt die PIN ein. Der Bankautomat überprüft die PIN. Wird eine falsche PIN eingegeben, so wird der Versuch gezählt; nach drei Fehlversuchen wird die Karte gesperrt. Nach der Eingabe der korrekten PIN fragt der Bankautomat nach der gewünschten Transaktion.

**Aktoren** Bankkunde, Zentralrechner

**Vorbedingung** Kartenleser betriebsbereit, Bedienpult gesperrt

**Nachbedingung** Karte im Kartenleser, Kartenleser gesperrt, Karte lesbar, Code gültig, PIN gelesen, PIN OK, Bedienpult auswahlbereit

**ODER** Karte nicht lesbar, Karte eingezogen

**ODER** Code nicht gültig, Karte ausgeworfen

**ODER** PIN dreimal falsch eingegeben, Karte gesperrt, Karte ausgeworfen

**UND** Kartenleser betriebsbereit, Bedienpult gesperrt

**END** Anmelden

Abb. 5.3 Spezifikation des Use Case Anmelden



Die Summe der Use Cases stellt eine relativ abstrakte, funktionale Sicht auf die Anforderungen dar, die insbesondere in Hinblick auf das Klassenmodell sowie die Ableitung von Testfällen nicht hinreichend ist. Wir verfeinern daher jeden Use Case analog zu einer Aufgabenhierarchie (vgl. [KSV96][Rosson99]) in einzelne, Teilaufgaben entsprechende Use Case Schritte. Zusätzlich halten wir fest, welcher der dem Use Case zugeordneten Aktoren einen bestimmten Use Case Schritt bearbeitet und spezifizieren die (Teil-) Aufgaben der Schritte analog zur Aufgabe des Use Cases mit Vor- und Nachbedingungen. Wie schon bei Use Cases beschreiben die Vor- und Nachbedingungen die Aufgaben der Use Case Schritte über den Zustand von ihnen betroffener Realweltobjekte.

**Definition 5.2** Ein *Use Case Schritt* (eines Use Cases uc) ist definiert durch

- ☐ einen im Kontext des Use Cases uc eindeutigen *Namen*,
- ☐ die textuelle *Beschreibung* der (Teil-) Aufgabe des Schritts,
- ☐ eine Menge  $A \subseteq A(uc)$  von in die (Teil-) Aufgabe des Schritts involvierten *Aktoren*,
- ☐ eine *Vor-* und eine *Nachbedingung*,

- die *Schrittart*  $SA \in \{\text{Kontext, Interaktion, Makro}\}$ .

□

Bei der Abgrenzung der einzelnen Teilaufgaben interessiert uns, inwiefern eine Teilaufgabe Anwendungsunterstützung erhalten soll. Die beiden Schrittarten Kontextschritt und Interaktionsschritt geben hierauf eine (grobe) Antwort:

- Ein *Kontextschritt* beschreibt eine (Teil-) Aufgabe, die alleine von den Akteuren des Schritts bearbeitet wird, also einem nicht von der Anwendung zu unterstützenden Teil des umgebenden Geschäftsprozesses entspricht.
- Ein *Interaktionsschritt* beschreibt eine (Teil-) Aufgabe, die interaktiv von den Akteuren des Schritts und der Anwendung bearbeitet wird. *Interaktionsschritte* spiegeln also die von der betrachteten Anwendung zu unterstützenden Teile des umgebenden Geschäftsprozesses wider.

Ob eine Teilaufgabe kontextuelle Information oder Interaktionsinformation (s. Abb. 4.5) beinhaltet, geht also unmittelbar aus der Art des Schritts hervor, denn Kontextschritte und Interaktionsschritte decken gerade die entsprechende Informationsart ab. In Kapitel 7 zeigen wir, dass Interaktionsschritte Anknüpfungspunkte für die (grobgranulare) Modellierung der dritten, systeminternen Information bieten. Diese wird dann mit sogenannten Episoden, die wir im Rahmen der Verifikation der Use Case Graphen und des Klassenmodells verwenden, angereichert und verfeinert (s. Kapitel 11).

Makroschritte (Schrittart  $SA = \text{Makro}$ ) ermöglichen eine redundanzfreie, hierarchische Spezifikation der Use Cases und übertragen die *uses*- und *extends*-Beziehungen von Use Cases auf die entsprechenden Use Case Schrittgraphen (s. Abschnitt 6.4). Vorläufig reicht die folgende informelle Beschreibung:

- Jeder *Makroschritt*  $s_{\text{makro}}$  enthält einen Verweis  $\text{RefUC}(s_{\text{makro}})$  auf einen anderen Use Case und beschreibt somit eine (Teil-) Aufgabe indirekt, indem er den referenzierten Use Case sozusagen „aufruft“.

Bei der Bearbeitung eines Makroschritts wird — analog zu einem Prozeduraufruf — der benutzte Use Case Schrittgraph sozusagen vom Makroschritt „aufgerufen“. Sei  $UC$  eine endliche Menge von Use Cases und  $uc \in UC$  ein Use Case Schrittgraph mit der Schrittmenge  $S(uc)$ .

**Definition 5.3** Die *Makroschritt-Referenzfunktion* wird definiert als  $REF: S(uc) \rightarrow UC$  mit  $REF(s) = \emptyset$ , falls  $SA(s) \neq \text{Makro}$  und  $REF(s) = \text{RefUC}(s) \in UC \setminus uc$ , sonst.

□

*Beispiel 5.2.* Im Rahmen des Use Cases Anmelden finden wir z. B. als Kontextschritt das Eingeben der Karte in den Kartenleser des Bankautomaten, als Interaktionsschritt die Eingabe der PIN. Abb. 5.4 zeigt die textuellen Spezifikationen der Use Case Schritte Karte Einführen

sowie Karte Lesen des Use Cases Anmelden und den Use Case Makroschritt Anmelden im Use Case Geld Abheben.

```

Kontextschritt Karte Einführen In Use Case Anmelden
  Beschreibung Der Bankkunde führt die Karte in den Kartenleser ein.
  Aktoren Bankkunde
  Vorbedingung Kartenleser betriebsbereit
  Nachbedingung Karte im Kartenleser, Kartenleser gesperrt
END Karte Einführen

Interaktionsschritt Karte Lesen In Use Case Anmelden
  Beschreibung Der Kartenleser liest den Magnetstreifen der Karte
  Aktoren -
  Vorbedingung Karte im Kartenleser, Kartenleser gesperrt
  Nachbedingung Kartenleser gesperrt, Karte lesbar
    ODER Kartenleser gesperrt, NICHT Karte lesbar
END Karte Lesen

Makroschritt Anmelden In Use Case Geld Abheben
  Beschreibung Der Bankkunde identifiziert sich gegenüber dem Bankautomaten.
    Nach der Identifikation zeigt der Bankautomat die möglichen Transaktionen an.
  Aktoren Bankkunde, Zentralrechner
  Vorbedingung Kartenleser betriebsbereit
  Nachbedingung Karte im Kartenleser, Kartenleser gesperrt, Karte lesbar,
    Code gültig, Bedienpult auswahlbereit
    ODER Karte ausgeworfen, Kartenleser betriebsbereit
    ODER Karte eingezogen, Kartenleser betriebsbereit
END Anmelden

```

*Abb. 5.4 Textuelle Spezifikation einiger Use Case Schritte*



## 5.2 Use Case Schrittgraphen

Die bis jetzt erreichte rein funktionale Zerlegung der Aufgabe eines Use Cases reicht zur Spezifikation der möglichen Abläufe bei der Bearbeitung dieser Aufgabe noch nicht aus. Anstelle der bei der Aufgabenmodellierung üblichen statischen Anordnung der (Teil-) Aufgaben nach Benutzungsbeziehungen (vgl. [KSV96][Rosson99]) führen wir nun eine die Abläufe beleuchtende Relation auf den Use Case Schritten ein. Diese gibt an, in welchen möglichen Reihenfolgen die (Teil-) Aufgaben eines Use Cases bearbeitet werden können. Wir modellieren sozusagen den „Kontrollfluss“ eines Use Cases.

Hierzu streben wir eine der strukturierten Programmierung entsprechende Ausdruckskraft an, müssen also in der Lage sein, die sequentielle, alternative und die wiederholte Bearbeitung der Aufgaben zu modellieren. In Analogie zu verallgemeinerten Flussdiagrammen (vgl. [Weihrauch87]) ordnen wir jedem Use Case Schritt eine Menge möglicher Nachfolgeschritte

zu. Zusätzlich markieren wir genau einen Schritt des Use Cases als sogenannten *Startschritt* und eine nichtleere Teilmenge der Schritte eines Use Cases als sogenannte *Endschritte*.

Präzisieren wir diese Überlegungen, so kommen wir zur folgenden Definition.

**Definition 5.4** Ein *Use Case Schrittgraph* (zu einem Use Case  $uc$ ) ist ein 4-Tupel  $UCG = \{S, s_0, SE, \sigma\}$  mit:

1.  $S$  ist eine nichtleere Menge von Use Case Schritten (*Knoten*);
2.  $s_0 \in S$  ist der *Startschritt* des Use Case Schrittgraphen;
3.  $SE \subseteq S$  ist die nichtleere Menge der *Endschritte* des Use Case Schrittgraphen;
4.  $\sigma \subseteq S \times S$  ist die Menge der gerichteten *Kanten* des Use Case Schrittgraphen. Jede Kante  $e = (s, s') \in \sigma$  wird mit einer *Zusicherung*<sup>1</sup> oder *Übergangsbedingung*  $c(e)$  annotiert, die angibt, unter welcher Bedingung  $s'$  als Folgeschritt ausgewählt wird.

□

Die Beschränkung auf genau einen Startschritt für jeden Use Case Schrittgraphen folgt aus der grundlegenden Eigenschaft, dass jeder Use Case eine (und nur eine) klar umrissene Aufgabe beschreibt. Der Startschritt spiegelt sozusagen den „Auslöser“ bzw. das auslösende Ereignis der Aufgabenausführung wider und wird insofern meistens von einem Akteur ausgeführt. Erst nach dieser ersten Reaktion sollten alternative Abläufe, d.h. Verzweigungen im Use Case Schrittgraphen auftreten.

Die mit dem Startschritt beginnenden und mit einem Endschritt abgeschlossenen Pfade durch den Use Case Schrittgraphen spiegeln genau die möglichen Szenarien wider, die bei der Bearbeitung der dem Use Case zugeordneten Aufgabe auftreten können.

**Definition 5.5** Ein *Szenario*  $Sz$  eines Use Case Schrittgraphen  $ucs = \{S, s_0, SE, \sigma\}$  ist ein Tupel  $Sz = (a_0, \dots, a_n)$ ,  $n > 0$  von Use Case Schritten, so das gilt:

1.  $a_i \in S$ ;
2.  $a_0 = s_0$  und  $a_n \in SE$ ;
3.  $a_{i+1} \in \sigma(a_i)$  für alle  $0 \leq i < n$ .

□

Seien  $UC$  eine endliche Menge von Use Cases und  $uc \in UC$  ein Use Case mit der Schrittmenge  $S(uc)$ . Wir erhalten eine „Benutzungsrelation“ auf Use Cases, indem wir jedem Use Case die Menge der von den Makroschritten seines Schrittgraphen referenzierten Use Cases zuordnen. Aufbauend auf Definition 5.3 (Makroschritt-Referenzfunktion) und Definition 5.4 kommen wir zur folgenden Definition 5.6.

---

<sup>1</sup> Wir erinnern daran, dass wir Zusicherungen entweder informell, umgangssprachlich oder formal als (mehrsortige) prädikatenlogische Ausdrücke z.B. in OCL angeben.

**Definition 5.6** Wir definieren die *Inter-Use Case Referenzfunktion*  $REF: UC \rightarrow 2^{UC}$  zu  $REF(uc) = \bigcup_{s \in S(uc)} REF(s)$ .

□

Wir behandeln Makroschritte ausführlich in den Abschnitten 6.3 und 6.4.

**Zur Notation.** Im Weiteren werden wir, wenn keine Mißverständnisse zu befürchten sind, die Begriffe Use Case und Use Case Schrittgraph synonym verwenden. Weiterhin bezeichnen wir wie üblich mit  $\mathbb{N}$  die natürlichen, mit  $\mathbb{Z}$  die ganzen und mit  $\mathbb{R}$  die reellen Zahlen.

- $S|_{\text{Interaktion}}$  für  $\{s \in S \mid SA(s) = \text{Interaktion}\}$  für die Menge aller Interaktionsschritte eines Use Case Schrittgraphen.
- Für die Menge aller Folgeschritte eines Use Case Schritts  $s$  schreiben wir auch  $\sigma(s)$  und nennen  $\sigma$  auch *Folgeschrittrelation*.
- Für  $s \in \sigma(s')$  bzw.  $(s, s') \in \sigma$  nennen wir  $s$  einen *Vorgängerschritt* von  $s'$  und umgekehrt  $s'$  *Nachfolgeschritt* von  $s$ .
- Schritte aus der Menge  $S \setminus SE$  nennen wir *innere Schritte* des Use Case Schrittgraphen.
- Schritte aus  $SE$ , die keinen Nachfolgeschritt haben, bezeichnen wir als *Blattschritte* des Use Case Schrittgraphen.

Zu den obigen Definitionen folgen — nach einem Beispiel — noch einige Anmerkungen und Modellierungsregeln.

**Beispiel 5.3.** Abb. 5.5 zeigt textuelle Spezifikationen der Schritte des Use Case Schrittgraphen Anmelden (die Vor- und Nachbedingungen sind noch umgangssprachlich formuliert). Es ist deutlich zu erkennen, dass sich die meisten Bedingungen auf Eigenschaften oder „Zustände“ von Domänenobjekten beziehen, wie z.B. Kartenleser betriebsbereit oder Karte lesbar.

```

Kontextschritt Karte Einführen In Use Case Anmelden
  Vorbedingung Kartenleser betriebsbereit
  Nachbedingung Karte im Kartenleser UND Kartenleser gesperrt
END Karte Einführen

Interaktionsschritt Karte Lesen In Use Case Anmelden
  Vorbedingung Karte im Kartenleser UND Kartenleser gesperrt
  Nachbedingung (Karte lesbar UND (Code gültig ODER NICHT Code gültig) ODER
    NICHT Karte lesbar)
END Karte Lesen

Interaktionsschritt PIN Anfordern In Use Case Anmelden
  Vorbedingung Karte im Kartenleser UND Kartenleser gesperrt UND Code gültig
  Nachbedingung PIN gelesen ODER Abbruch
END PIN Anfordern

Interaktionsschritt PIN Prüfen In Use Case Anmelden
  Vorbedingung Karte im Kartenleser UND Kartenleser gesperrt UND Karte lesbar UND
    Code gültig UND PIN gelesen
  Nachbedingung PIN OK ODER NICHT PIN OK

```



```

END PIN Prüfen

Interaktionsschritt Auswahl Anzeigen In Use Case Anmelden
  Vorbedingung Karte im Kartenleser UND Kartenleser gesperrt UND Karte lesbar UND
    Code gültig UND PIN gelesen UND PIN OK
  Nachbedingung Bedienpult auswahlbereit
END Auswahl Anzeigen

Interaktionsschritt Karte Schreiben In Use Case Anmelden
  Vorbedingung Karte im Kartenleser UND Kartenleser gesperrt UND Karte lesbar UND
    Code gültig UND (PIN heute dreimal falsch ODER Abbruch)
  Nachbedingung WENN PIN heute dreimal falsch DANN Karte gesperrt
END Karte Schreiben

Interaktionsschritt Karte Anbieten In Use Case Anmelden
  Vorbedingung Karte im Kartenleser UND Kartenleser gesperrt
  Nachbedingung Karte angeboten UND Timer läuft
END Karte Anbieten

Kontextschritt Karte Entnehmen In Use Case Anmelden
  Vorbedingung Karte angeboten UND Kartenleser gesperrt UND Timer läuft
  Nachbedingung (Karte entnommen UND Kartenleser betriebsbereit UND Timer gestoppt)
    ODER (Karte angeboten UND Timer >= 60)
END Karte Entnehmen

Interaktionsschritt Karte Einziehen In Use Case Anmelden
  Vorbedingung (Karte angeboten UND Kartenleser gesperrt UND Timer >= 60)
    ODER (Kartenleser gesperrt UND NICHT Karte lesbar)
  Nachbedingung Karte eingezogen UND Timer gestoppt UND Kartenleser betriebsbereit
END Karte Einziehen

```

*Abb. 5.5 Bedingungen der Schritte im Use Case Schrittgraph Anmelden*

Bezüglich der Kantenbedingungen ist anzumerken, dass diese sich nicht auf „Ereignisse“ wie z.B. bei Zustandsmaschinen (Statecharts, vgl. [HarNaa96][OMG97]) beziehen, sondern auf den Zustand „der Anwendung“ bzw. bestimmter Domänenobjekte. Abhängig vom Stand der Aufgabenbearbeitung ermöglichen sie die Auswahl der nächsten (Teil-) Aufgabe. Abb. 5.6 zeigt die textuellen Spezifikationen der Kanten des Use Case Schrittgraphen Anmelden.

```

Kante Karte Einführen - Karte Lesen In Use Case Anmelden
  Bedingung Karte im Kartenleser
END Karte Einführen - Karte Lesen

Kante Karte Lesen - PIN Anfordern In Use Case Anmelden
  Bedingung Karte lesbar UND Code gültig
END Karte Lesen - PIN Anfordern

Kante Karte Lesen - Karte Anbieten In Use Case Anmelden
  Bedingung NICHT Code gültig
END Karte Lesen - Karte Anbieten

Kante Karte Lesen - Karte Einziehen In Use Case Anmelden
  Bedingung NICHT Karte lesbar
END Karte Lesen - Karte Einziehen

Kante PIN Anfordern - PIN Prüfen In Use Case Anmelden
  Bedingung PIN gelesen
END PIN Anfordern - PIN Prüfen

```

**Kante** PIN Anfordern - Karte Schreiben **In Use Case** Anmelden  
**Bedingung** Abbruch gewählt  
**END** PIN Anfordern - Karte Schreiben  
**Kante** PIN Prüfen - PIN Anfordern **In Use Case** Anmelden  
**Bedingung** NICHT PIN OK **UND** NICHT PIN heute dreimal falsch  
**END** PIN Prüfen - PIN Anfordern  
**Kante** PIN Prüfen - Auswahl Anzeigen **In Use Case** Anmelden  
**Bedingung** PIN OK  
**END** PIN Prüfen - Auswahl Anzeigen  
**Kante** PIN Prüfen - Karte Schreiben **In Use Case** Anmelden  
**Bedingung** PIN heute dreimal falsch  
**END** PIN Prüfen - Karte Schreiben  
**Kante** Karte Schreiben - Karte Anbieten **In Use Case** Anmelden  
**Bedingung** True  
**END** Karte Schreiben - Karte Anbieten  
**Kante** Karte Anbieten - Karte Entnehmen **In Use Case** Anmelden  
**Bedingung** True  
**END** Karte Anbieten - Karte Entnehmen  
**Kante** Karte Entnehmen - Karte Einziehen **In Use Case** Anmelden  
**Bedingung** Timer >= 60  
**END** Karte Entnehmen - Karte Einziehen

Abb. 5.6 Kanten im Use Case Schrittgraph Anmelden



### 5.3 SCORES-Metamodell (I)

In Hinblick auf die Implementation des vorgeschlagenen hierarchischen Use Case Modells in einem CASE-Tool geben wir das in Abb. 5.7 gezeigte (vorläufige) UML-Metamodell für die Use Case Schrittgraphen entsprechenden Primitive von SCORES an. Wir verwenden dabei an englische Begriffe angelehnte Bezeichner, wobei die Übertragung auf die bei den entsprechenden Konzepten verwendeten deutschen Bezeichner problemlos möglich sein sollte.

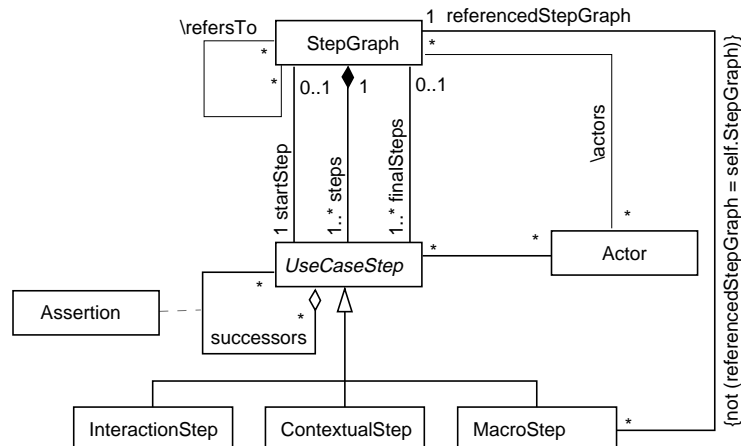


Abb. 5.7 Use Case Schrittgraph Metamodell

Betrachten wir die Assoziation `referencedStepGraph`, welche die Klassen **MacroStep** und **StepGraph** verbindet und die Makroschritt-Referenzfunktion modelliert (s. Definition 5.3). Der Constraint `{not (referencedStepGraph = self.StepGraph)}` verbietet direkte rekursive Bezüge von einem Makroschritt auf den diesen Schritt enthaltenden Use Case (Schrittgraphen). Die abgeleitete selbstrekursive Assoziation `\refersTo` der Klasse **StepGraph** überträgt die durch die Makroschritte bedingten Abhängigkeiten auf die entsprechenden Use Case Schrittgraphen und modelliert die Inter-Use Case Referenzfunktion (s. Definition 5.6). Die abgeleitete Assoziation von der Klasse **StepGraph** zur Klasse **Actor** wird aus der entsprechenden Assoziation der zugeordneten Objekte der Klasse **UseCaseStep** abgeleitet. Letztendlich beinhaltet die Assoziationsklasse **Assertion**, welche die Assoziation `successors` attributiert, die Bedingung zur Auswahl eines Folgeschritts.

## 5.4 Modellierungsregeln (I)

Formale Definitionen allein können nicht sicherstellen, dass die ihnen genügenden Modelle auch sinnvoll sind. Zusätzlich geben wir deshalb hier und in weiteren Abschnitten einige Regeln an, welche zur Modellierung sinnvoller Use Case Schrittgraphen hilfreich sind. Die Regeln können z.B. als OCL-Ausdrücke im Metamodell oder als Prädikate über den Modellierungselementen formuliert werden. Wir beschränken uns bei den Regeln auf letzteres und verwenden die OCL erst später zur Formulierung der auf dem SCORES-Metamodell aufbauenden Metriken (vgl. Kapitel 10 und 12).

Natürlich muss die Konstruktion (temporär) bezüglich der Regeln inkonsistenter Modelle erlaubt sein bzw. die Einhaltung der Regeln nur auf Anforderung geprüft werden. Auf eine dementsprechende Werkzeugunterstützung gehen wir in Teil V ein.

Betrachten wir zunächst die Modellierung von Aktoren. Wir erinnern uns daran, dass Aktoren generalisiert werden können und einigen uns auf folgende, die Darstellung vereinfachende Regel.

**Regel 5.4.1 [Use Case-Aktoren]** Zu einem Use Case bzw. Use Case Schritt werden immer nur die am höchsten in der Vererbungshierarchie stehenden Aktoren angegeben.

Da ein Kontextschritt nur von Aktoren bearbeitet wird, erhalten wir die folgende Regel.

**Regel 5.4.2 [Kontextschritt-Aktoren]** Ein Kontextschritt hat mindestens einen Actor:

$$\forall s \in S(uc): SA(s) = \text{Kontext} \Rightarrow A(s) \neq \emptyset.$$

Umgekehrt liegt es im Ermessen des Modellierers, ob er z.B. bei zeitabhängigen Aufgaben die „Systemuhr“ als Actor modelliert oder aber als „in die Anwendung“ integriert betrachtet und somit eine zeitabhängige Aufgabe als Schritt ohne Actor modelliert. Interaktionsschritte (sowie Makroschritte) können, müssen aber keine Aktoren haben.

Wir kommen nun zu einigen strukturellen Eigenschaften der Use Case Schrittgraphen. Zunächst sollen sämtliche Schritte des Use Case Schrittgraphen eng mit der Aufgabe des Use Cases zusammenhängen, also auch auf irgend einem Pfad vom Startschritt aus erreichbar sein. Hierzu formulieren wir die zwei folgenden Modellierungsregeln.

**Regel 5.4.3 [Schrittgraph-Wurzelschritte]** Jeder Use Case Schrittgraph muss eine nichtleere Menge von Schritten enthalten, von denen aus mindestens alle anderen<sup>1</sup> Schritte des Use Case Schrittgraphen erreichbar sind:

$$\exists s \in S(uc): \sigma^*(s) \supseteq S(uc) \setminus s. \quad ^2$$

Wir nennen jeden Schritt mit der in Regel 5.4.3 angegebenen Eigenschaft *Wurzelschritt* des Use Case Schrittgraphen. Es folgt, dass jeder Use Case Schrittgraph ein gerichteter Wurzelgraph und damit insbesondere zusammenhängend ist. Da jedes Szenario, also jeder Pfad durch den Use Case Schrittgraphen den Startschritt als erste Reaktion auf den „Auslöser“ bzw. das auslösende Ereignis der Aufgabenausführung enthalten muss, sollte der Startschritt offensichtlich ein Wurzelschritt sein.

**Regel 5.4.4 [Schrittgraph-Startschritt]** Der Startschritt  $s_0$  eines Use Cases  $uc$  muss ein Wurzelschritt sein:  $\sigma^*(s_0) \supseteq S(uc) \setminus s_0$ .

Ebenso muss jedes Szenario, welches einen Schritt ohne Folgeschritte enthält, mit diesem Schritt enden. Wir erhalten die nächste Modellierungsregel.

**Regel 5.4.5 [Schrittgraph-Endschritte]** Alle Schritte ohne Nachfolgeschritt müssen in der Menge der Endschritte des Use Case Schrittgraphen enthalten sein:

$$\forall s \in S(uc): \sigma(s) = \emptyset \Rightarrow s \in SE(uc).$$

Der Fall  $SE(uc) = \emptyset$  ist definitorisch ausgeschlossen. Wir müssen allerdings beachten, dass Regel 5.4.5 Endschritte mit Nachfolgeschritten erlaubt — Szenarien können in ihrem Verlauf mehrere Endschritte durchlaufen. Solange es noch möglich ist, die Aufgabe erfolgreich zu bearbeiten, werden Alternativen und Ausnahmebehandlungen versucht. Im Extremfall kann  $SE(uc) = S(uc)$  sein.

Use Case Schritte und Schrittgraphen entsprechen einer funktionalen Zerlegung, bei der Aufgaben auf verschiedenen Abstraktionsgraden modelliert werden. Zyklische Benutzungen bzw. Referenzen durch Makroschritte deuten hierbei erfahrungsgemäß oft auf eine unsaubere Zerlegung oder auf vorweggenommene Navigationsmöglichkeiten der Benutzungsschnittstelle hin.

Analog zur Benutzungsstruktur in Modulgraphen (siehe z.B. [PagSix94]) fordern wir auch bei der Modellierung von Use Case Schrittgraphen, dass die Inter-Use Case Referenzfunktion

1 Die Formulierung „mindestens alle anderen Schritte“ macht nur im Falle genau eines Schritts mit dieser Eigenschaft Sinn. Dieser darf den Eingangsgrad 0 haben.


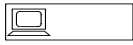

2 Der Stern „\*“ bezeichnet hier den transitiven Abschluß einer Relation.

$REF: UC \rightarrow 2^{UC}$  azyklisch sein muss. Diese Vereinbarung vereinfacht insbesondere Betrachtungen zur Terminierung der mit Use Case Schrittgraphen beschriebenen Aufgaben (vgl. [HOR98]).

Regel 5.4.6 [Makro-Struktur] Der transitive Abschluss der Inter-Use Case Referenzfunktion ist irreflexiv:  $\forall uc \in UC: uc \notin REF^*(uc)$ .



## 5.5 Darstellungskonventionen

Wir kommen nun zur Darstellung von Use Case Schrittgraphen. Use Case Schritte werden mit einem der Schrittart entsprechenden Symbol gezeichnet (Tab. 5.1).

Symbol	Bedeutung
	Kontextschritt
	Interaktionsschritt
	Makroschritt

Tab. 5.1 Symbole der Use Case Schritte

Nachfolgeschritte werden mit ihrem Vorgängerschritt durch einen Pfeil mit geschlossener, ausgefüllter Spitze verbunden. Die dem Use Case Schritt zugeordneten Aktoren werden wie im Use Case Diagramm dargestellt (vgl. [OMG97]). Verläuft die Kommunikation zwischen Aktor und Schritt nur in einer Richtung, deuten wir dies mit einem Pfeil mit offener Spitze an. Den Startschritt markieren wir mit einem doppelt gezeichneten Rahmen, alle Endschritte eines Use Case Schrittgraphen durch ausgefüllte Dreiecke links oben und rechts unten im Symbol (Tab. 5.2).

Markierung	Bedeutung
	Startschritt
	Endschritt

Tab. 5.2 Start- und Endschritt-Markierung

*Beispiel 5.4.* Wir greifen das Beispiel des Bankautomaten wieder auf. Abb. 5.8 (a) und (b) zeigen Use Case Schrittgraphen zu den Use Cases Anmelden und Geld Abheben.

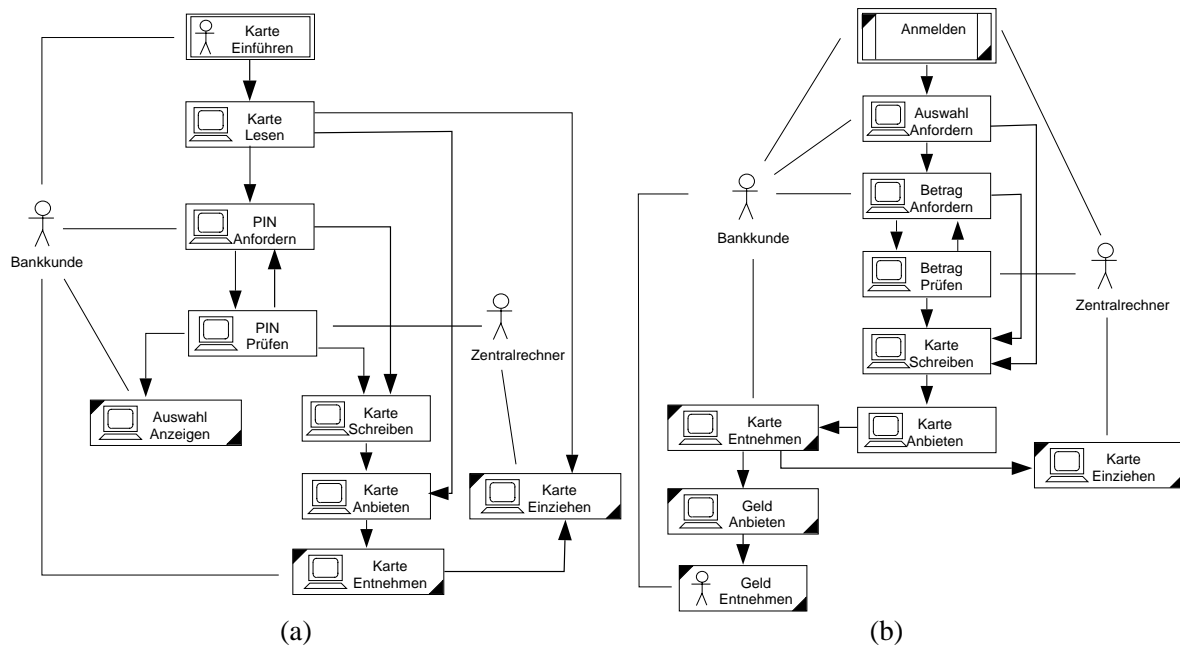


Abb. 5.8 Use Case Schrittgraphen Anmelden (a) und Geld Abheben (b)

Szenarien visualisieren wir mit Sequenzdiagrammen. Makroschritte wie der Schritt Anmeldung im Szenario in Abb. 5.9 (b) werden hierbei „komprimiert“ dargestellt und sollen ein Szenario des referenzierten Use Cases (hier Anmelden) symbolisieren (*probe*, vgl. [JCJ+92], [HitKap98]).

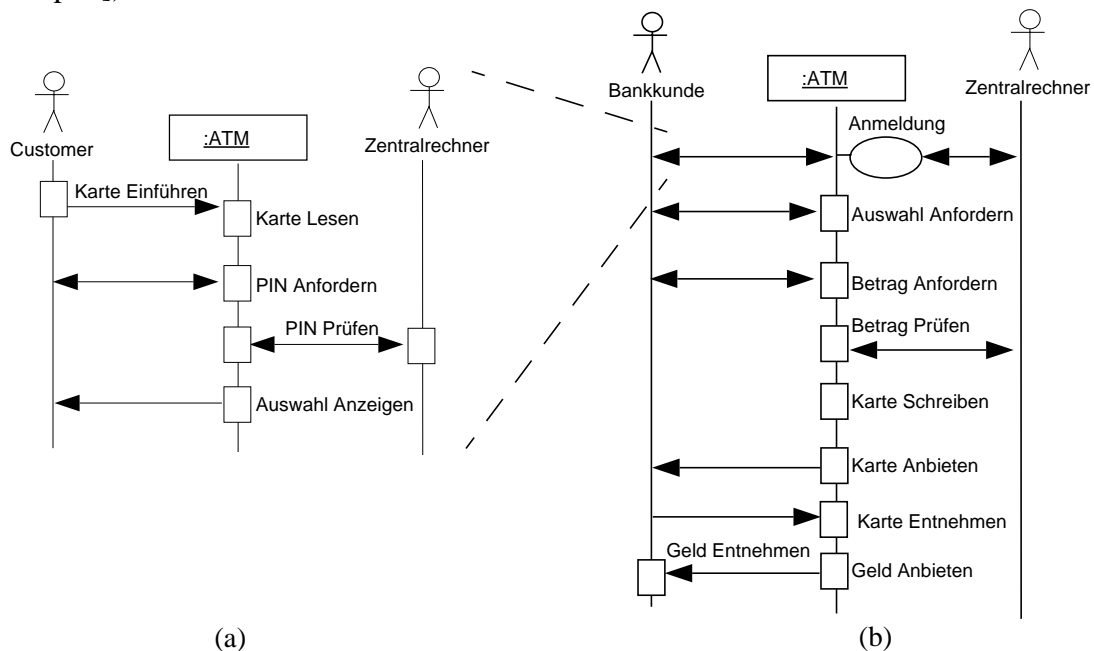


Abb. 5.9 (Erfolgreiche) Szenarien zu den Use Cases Anmelden (a) und Geld Abheben (b)



# Kapitel 6

## Semantik

Wir skizzieren in diesem Abschnitt eine operationale oder — treffender gesagt — „ablaufbezogene“ Semantik für Use Case Schrittgraphen. Hierzu geben wir analog zu Berechnungen in Flussdiagrammen (vgl. [Weihrauch87]) zu den möglichen Schrittarten und der Anzahl der Folgeschritte (keiner, einer oder mehrere Nachfolgeschritte) in Pseudocode an, welches Verhalten der Schritt im Kontext eines Use Case Schrittgraphen bewirkt.

Umgangssprachlich können wir die Semantik der Schritte im Use Case Schrittgraph wie folgt beschreiben: Die Vorbedingung eines Schritts gibt die minimalen Voraussetzungen an, mit der die Bearbeitung seiner Aufgabe begonnen werden kann. Die Nachbedingung bestimmt zusammen mit den Kantenbedingungen die Auswahl eines nächsten Schritts oder — im Falle von Endschritten — ggf. das Ende der Bearbeitung.

### 6.1 Schritte in Use Case Schrittgraphen

Wir beginnen mit Kontext- und Interaktionsschritten und behandeln danach Makroschritte. In den Beschreibungen ist der betrachtete Schritt jeweils schwarz, ohne eingehende Kanten gezeichnet; die für den Kontext des Schritts im Use Case Schrittgraphen maßgeblichen Schritte sind grau gezeichnet und können von beliebiger Schrittart sein.

- Kontext- oder Interaktionsschritt, kein Folgeschritt (Abb. 6.1).

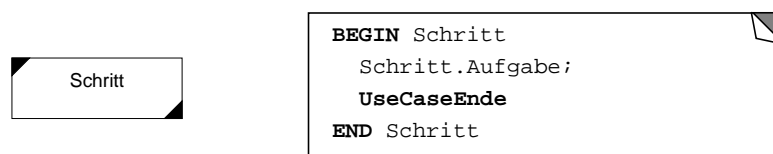


Abb. 6.1 Semantik 1, einfachster Fall

Der Schritt beschreibt eine elementare Aufgabe, welche entweder die mit dem Use Case beschriebene Aufgabe erfolgreich beendet oder aber für den Fall, dass die Aufgabe des Use Cases nicht mehr erfolgreich abgearbeitet werden kann, sozusagen „Aufräumarbeiten“ verrichtet. Jede Bearbeitung der Aufgabe des Use Cases, in der die Teilaufgabe des Schritts ausgeführt wird, muss also mit diesem Schritt enden (vgl. Re-

gel 5.4.5). Wir deuten diesen Sachverhalt in Abb. 6.1 durch das Schlüsselwort **UseCaseEnde** an.

- Kontext- oder Interaktionsschritt, ein Folgeschritt, kein Endschrift (Abb. 6.2).

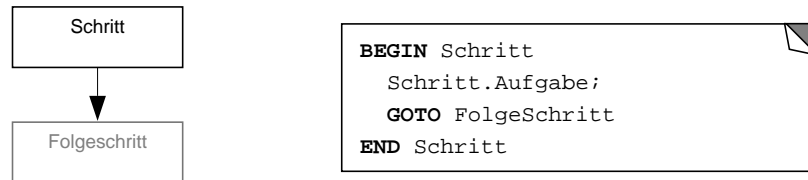


Abb. 6.2 Semantik 2: kein Endschrift, ein Folgeschritt

Der Schritt beschreibt eine elementare Teilaufgabe, nach der die im Folgeschritt beschriebene Aufgabe bearbeitet werden muss. Dies wird durch das Schlüsselwort **GOTO** gekennzeichnet (Abb. 6.2 rechts).

- Kontext- oder Interaktionsschritt, ein Folgeschritt, Endschrift (Abb. 6.3).

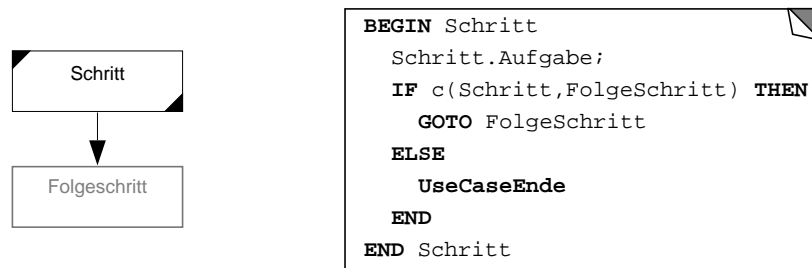


Abb. 6.3 Semantik 3: Endschrift, ein Folgeschritt

Der Schritt beschreibt eine elementare Teilaufgabe, mit der ein Szenario entweder endet, oder aber, wenn die Bedingung  $Pre$  erfüllt ist, mit der im Folgeschritt beschriebenen Aufgabe fortgesetzt werden muss. Die Bedingung  $c(Schritt, FolgeSchritt)$  darf hier nicht schon alleine von der Nachbedingung  $Post_{Schritt}$  des Schritts impliziert werden, da Schritt sonst kein echter Endschrift wäre und der vorherige Fall vorliegt. Meistens wird  $c(Schritt, FolgeSchritt)$  aus um weitere Bedingungen ergänzten Teilen der Nachbedingung  $Post_{Schritt}$  des Schritts Schritt zusammengesetzt sein.

Wir kommen nun zu Schritten, denen mehrere Folgeschritte zugeordnet sind, und unterscheiden die folgenden drei Fälle:

- Kontext- oder Interaktionsschritt, mehrere Folgeschritte, Übergangsbedingungen der ausgehenden Kanten schliessen sich gegenseitig aus, kein Endschrift (Abb. 6.4). Für  $1 \leq i \neq j \leq n$  ist  $c(Schritt, FolgeSchritt_i) \wedge c(Schritt, FolgeSchritt_j) \Leftrightarrow False$ . Unabhängig vom tatsächlichen Ergebnis des Schritts muss natürlich immer mindestens einer der Folgeschritte ausgewählt werden können, d.h. es gilt  $c(Schritt, FolgeSchritt_1) \vee \dots \vee c(Schritt, FolgeSchritt_n) \Leftrightarrow True$ . Ist dies nach der Bearbeitung der Aufgabe nicht



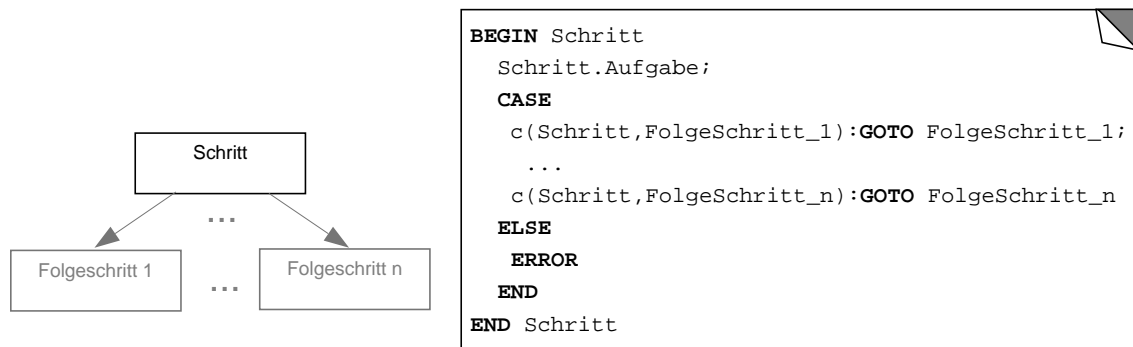


Abb. 6.4 Semantik 4: kein Endschrift, mehrere sich ausschliessende Folgeschritte

der Fall, d.h. keine der Bedingungen der vom Schritt ausgehenden Kanten ist erfüllt, ist entweder ein Spezifikationsfehler entdeckt worden, oder der Use Case Schritt muss als Endschrift markiert werden. Wir deuten diesen Sachverhalt wie in Abb. 6.7 durch das Schlüsselwort **Error** an.

- Kontext- oder Interaktionsschritt, mehrere Folgeschritte, Übergangsbedingungen der ausgehenden Kanten schliessen sich nicht gegenseitig aus, kein Endschrift. Es gibt  $1 \leq i \neq j \leq n$  mit  $c(\text{Schritt}, \text{FolgeSchritt}_i) \wedge c(\text{Schritt}, \text{FolgeSchritt}_j)$  ist erfüllbar. Solche Fälle weisen auf unabhängige, im Prinzip parallel bearbeitbare Aufgaben oder — technisch gesehen — nebenläufige Prozesse hin. Wir betrachten den Spezialfall, dass für  $1 \leq m < n$  mehrere Folgeschritte Folgeschritt\_1 bis Folgeschritt\_m unabhängig voneinander ausgewählt, aber allesamt vor weiteren Folgeschritten Folgeschritt\_{m+1} bis Folgeschritt\_n ausgeführt werden müssen. Es ergibt sich eine wie in Abb. 6.5 (a) gezeigte Struktur. In diesem Fall können wir irgendeine willkürliche Reihenfolge der Schritte 1 bis m modellieren oder aber einen „künstlichen“, die Schritte Folgeschritt\_1 bis Folgeschritt\_m zusammenfassenden Schritt „Schritt“ einführen, der die Schritte Folgeschritt\_1 bis Folgeschritt\_n als Folgeschritte hat und zu dem die Schritte 1 bis m wieder „unbedingt“ zurückverzweigen (d.h. es gelte  $\sigma(\text{FolgeSchritt}_i) = \{\text{Schritt}\}$  für jeden der Schritte Folgeschritt\_1 bis Folgeschritt\_m und für  $1 \leq i \leq m$  sei  $c(\text{FolgeSchritt}_i, \text{Schritt}) = \text{True}$ ).

Da solche Ablaufstrukturen häufig auftreten, führen wir als abkürzende grafische Notation die in Abb. 6.5 (b) dargestellte Variante ein. Der „Synchronisationsbalken“ unter dem Use Case Schritt „Schritt“ deutet an, dass die mit einer Linie mit Doppelpfeil verbundenen Schritte Folgeschritt\_1 bis Folgeschritt\_m unabhängig voneinander alle ausgeführt sein müssen, bevor einer der Schritte Folgeschritt\_{m+1} bis Folgeschritt\_n

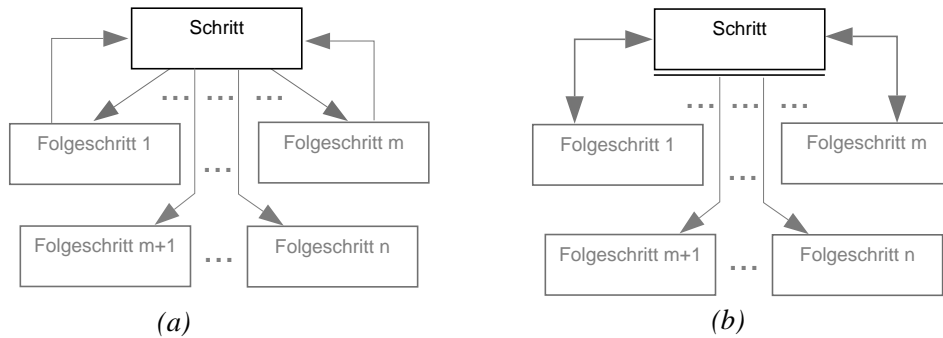
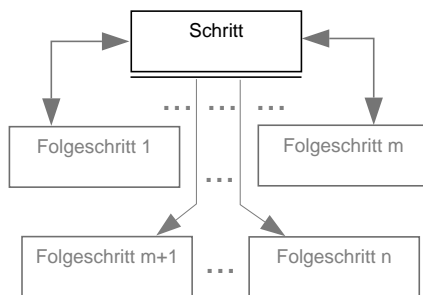


Abb. 6.5 (a) unabhängige Folgeschritte (b) abkürzende Notation

ausgeführt wird. Semantisch spiegelt sich dieser Sachverhalt darin wider, dass bestimmte Kombinationen der Nachbedingungen der Use Case Schritte Folgeschritt\_1 bis Folgeschritt\_m jeweils eine oder mehrere Vorbedingungen der Schritte Folgeschritt\_m+1 bis Folgeschritt\_n erfüllen. Für diesen Spezialfall vereinbaren wir die in Abb. 6.6 skizzierte Semantik<sup>1</sup>. Für einen Schritt x gebe hierbei das Prädikat `executed(x)` an, ob die Aufgabe von x innerhalb des aktuellen Ablaufs bereits ausgeführt ist.



```

BEGIN Schritt
  Schritt.Aufgabe;
  WHILE NOT (executed(FolgeSchritt_1) AND ...
    ... AND executed(FolgeSchritt_m)) DO
    CASE
      NOT executed(FolgeSchritt_1): FolgeSchritt_1;
      ...
      NOT executed(FolgeSchritt_m): FolgeSchritt_m
    END
  END;
  CASE
    c(Schritt,FolgeSchritt_m+1):GOTO FolgeSchritt_m+1;
    ...
    c(Schritt,FolgeSchritt_n):GOTO FolgeSchritt_n
  ELSE
    ERROR
  END
END Schritt

```

Abb. 6.6 Semantik 5: kein Endschrift, „Parallel“ ausführbare Folgeschritte

<sup>1</sup> Wir hätten auch die in der Literatur oft verwendete bewachte (nichtdeterministische) Anweisung z.B. in der Form

**DO NOT** `executed(FolgeSchritt_1): FolgeSchritt_1` | **NOT** `executed(FolgeSchritt_2): FolgeSchritt_2` | ... | **NOT** `executed(FolgeSchritt_m): FolgeSchritt_m` **OD**

zur Spezifikation des „zusammenfassenden“ Schritts benutzen können ([Dijkstra76]). Da wir jedoch bei der **CASE**-Anweisung sowieso keine Reihenfolge für die Prüfung der Bedingungen voraussetzen, erschien dies nicht als zwingend.

Wir sehen von einer weitergehenden Modellierung möglicher Nebenläufigkeiten ab und verweisen in diesem Zusammenhang auf UND-Zustände in Zustandsdiagrammen (vgl. [HarNaa96][OMG97]) und auf Petri-Netze (vgl. [DesObe96]). Abgesehen vom oben gezeigten Spezialfall modellieren wir die betreffenden Schritte in irgendeiner Reihenfolge als Sequenz. Technisch schlagen sich diesbezügliche Anforderungen an die Mehrbenutzerfähigkeit bzw. „Multiple Sessions“ auf Einplatz-Systemen im Transaktionsmechanismus (vgl. [VosGrH93]) bzw. den Modalitäten zwischen Elementen der Benutzungsschnittstelle nieder (vgl. [KSV96] [VosNen98]).

- Kontext- oder Interaktionsschritt, mehrere Folgeschritte und Endschrift (Abb. 6.7).

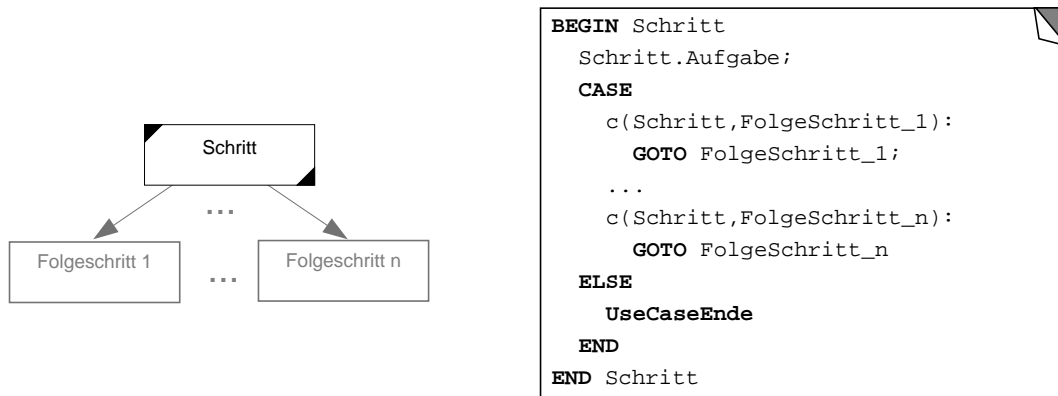


Abb. 6.7 Semantik 6: Endschrift, mehrere Folgeschritte

Dieser Fall unterscheidet sich lediglich aufgrund des zusätzlichen **ELSE**-Teils in der **CASE**-Anweisung vom vorigen. Es darf hier der Fall eintreten, dass nach der Bearbeitung der Aufgabe keine der Bedingungen  $c(\text{Schritt}, \text{FolgeSchritt}_1)$  bis  $c(\text{Schritt}, \text{FolgeSchritt}_n)$  zutrifft. In diesen Fällen wird die Bearbeitung des Use Cases beendet.

Kommen wir nun zur Semantik der Makroschritte. Jeder Makroschritt  $s_{\text{makro}}$  referenziert einen Use Case Schrittgraphen  $\text{RefUC}(s_{\text{makro}})$ . Wir betrachten den Fall, dass der Makroschritt kein Endschrift ist und mehrere Nachfolger hat.

- Makroschritt, mehrere Folgeschritte, Übergangsbedingungen der ausgehenden Kanten schliessen sich gegenseitig aus, kein Endschrift (Abb. 6.8) .

Den „Aufruf“ bzw. die Ausführung des referenzierten Use Cases deuten wir mit dem Schlüsselwort **DO** an. Im Sinne des „Design by Contract“ (s. [Meyer97]) vereinbaren wir, dass die Vorbedingung des referenzierten Use Cases von der des Makroschritts impliziert werden muss und umgekehrt die disjunktiv verknüpften Nachbedingungen der Endschritte des referenzierten Use Cases die Nachbedingung des Makroschritts implizieren<sup>1</sup>, es gelten also die beiden Implikationen  $\text{Pre}_{s_{\text{makro}}} \Rightarrow \text{Pre}_{\text{REF}(s_{\text{makro}})}$  und

$$\left( \bigvee_{s \in SE(\text{REF}(s_{\text{makro}}))} \text{Post}_s \right) \Rightarrow \text{Post}_{s_{\text{makro}}}.$$

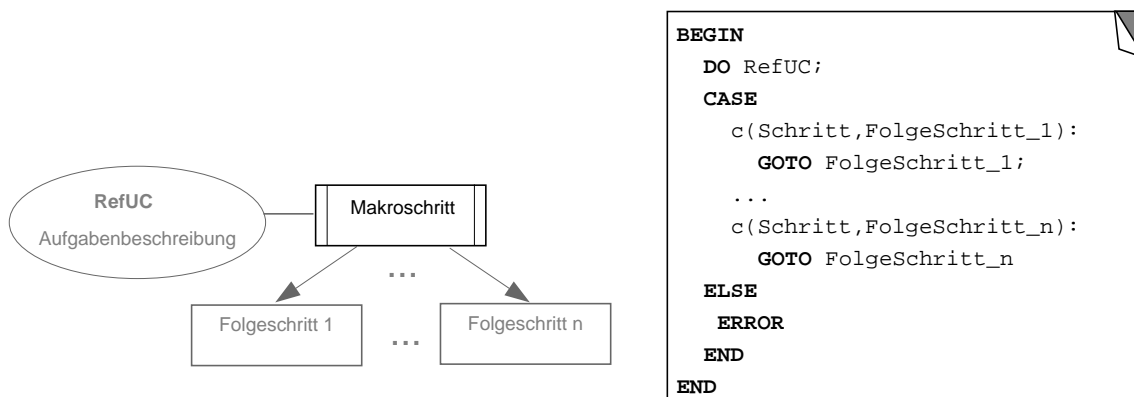


Abb. 6.8 Semantik 7: Makroschritt

Somit gilt die für Kontext- und Interaktionsschritte definierte Semantik auch für entsprechend im Schrittgraph eingebettete Makroschritte. Wir werden weitergehende Nebenbedingungen und Modellierungsregeln insbesondere zur Beleuchtung von Fragen wie z.B. „Welche Abläufe ergeben sich bei erfolgreicher/nicht erfolgreicher Bearbeitung der Aufgabe des referenzierten Use Cases?“ in Abschnitt 6.3 angeben.

## 6.2 Interpretation und Entscheidbarkeit

Wir interpretieren einen Use Case Schrittgraphen folgendermaßen: Beginnend mit der (Teil-) Aufgabe des Startschritts und konkreten Wertebelegungen für die in der Vorbedingung des Use Cases bzw. der des Startschritts vorkommenden Variablen (i.e. Instanzen von Domänenklassen) werden die (Teil-)Aufgaben der Schritte bearbeitet. Nach jeder Aufgabe wird gemäß der im vorigen Abschnitt definierten Semantik anhand des Ergebnisses (und damit, wie wir in Kapitel 11 zeigen, des Zustands, in dem sich die betrachteten Objekte des Anwendungsbereichs befinden) einer der Folgeschritte ausgewählt oder die Bearbeitung des Schrittgraphen ggf. beendet (Endschritt). Jede solcher Bearbeitungen spiegelt sich in einem Szenario wider. Die Wirkung eines Szenarios wird durch die Wertebelegungen für die in der Nachbedingung des Use Cases sowie der aller im Szenario ausgeführten Use Case Schritte bzw. der Übergangsbedingungen der Kanten vorkommenden Variablen konkretisiert. Alle so möglichen Szenarien definieren die Semantik des Use Case Schrittgraphen.

Die Bedingungen werden formal als (mehrsortige) prädikatenlogische Ausdrücke z.B. in OCL angeben. Aus der Unentscheidbarkeit der Prädikatenlogik folgt, dass Fragen wie z.B. der gegenseitige Ausschluss von Übergangsbedingungen der ausgehenden Kanten eines Use Case Schritts oder die Erfüllbarkeit einzelner Bedingungen i. Allg. nicht entscheidbar sind. Darüber hinaus ist natürlich auch das Terminierungsproblem in Use Case Schrittgraphen

---

<sup>1</sup> In der Praxis sind die Vor- und Nachbedingung des Makroschritts oft gleich denen des referenzierten Use Cases.

nicht entscheidbar. Wir nehmen hier den pragmatischen Standpunkt ein, dass — gerade bei der Anforderungsermittlung — die konkreten, praktisch auftretenden Bedingungen in der Regel entscheidbar sind und setzen somit im Weiteren stillschweigend voraus, dass die Bedingungen sich nicht widersprechenden, d.h. die Konjunktionen bzw. Disjunktionen nicht zu Tautologien degenerieren.

Wir illustrieren die Interpretation von Use Case Schrittgraphen an einem Beispiel.

*Beispiel 6.1.* Abb. 5.8 (a) zeigt den Schrittgraph des Use Case Anmelden im Fallbeispiel Bankautomat. Eine erfolgreiche Bearbeitung des Use Cases führt zur Identifikation des Kunden und Anzeige der Auswahlmöglichkeiten, wie es Abb. 5.9 (a) zeigt. Die Wirkung des Szenarios ermitteln wir, indem wir die Nachbedingungen der im Szenario besuchten Schritte sowie die Bedingungen der durchlaufenen Kanten betrachten (Abb. 6.9 sowie Abb. 6.10).

```

Kontextschritt Karte Einführen In Use Case Anmelden
  Nachbedingung Karte im Kartenleser UND Kartenleser gesperrt
Interaktionsschritt Karte Lesen In Use Case Anmelden
  Nachbedingung (Karte lesbar UND (Code gültig ODER NICHT Code gültig) ODER
    NICHT Karte lesbar)
Interaktionsschritt PIN Anfordern In Use Case Anmelden
  Nachbedingung PIN gelesen ODER Abbruch
Interaktionsschritt PIN Prüfen In Use Case Anmelden
  Nachbedingung PIN OK ODER NICHT PIN OK
Interaktionsschritt Auswahl Anzeigen In Use Case Anmelden
  Nachbedingung Bedienpult auswahlbereit
END Auswahl Anzeigen

```

*Abb. 6.9 Nachbedingungen der im Szenario Erfolgreiche Anmeldung besuchten Use Case Schritte*

```

Kante Karte Einführen - Karte Lesen In Use Case Anmelden
  Bedingung Karte im Kartenleser
END Karte Einführen - Karte Lesen
Kante Karte Lesen - PIN Anfordern In Use Case Anmelden
  Bedingung Karte lesbar UND Code gültig
END Karte Lesen - PIN Anfordern
Kante PIN Anfordern - PIN Prüfen In Use Case Anmelden
  Bedingung PIN gelesen
END PIN Anfordern - PIN Prüfen
Kante PIN Prüfen - Auswahl Anzeigen In Use Case Anmelden
  Bedingung PIN OK
END PIN Prüfen - Auswahl Anzeigen

```

*Abb. 6.10 Bedingungen der im Szenario Erfolgreiche Anmeldung besuchten Kanten*

In Worten ergibt sich für den Use Case Anmelden, dass nach Ablauf des Szenarios Erfolgreiche Anmeldung die Karte im Kartenleser, der Kartenleser für weitere Karten gesperrt und die Karte lesbar sowie der Code der Karte gültig ist. Die PIN wurde eingegeben und als richtig

geprüft. Nach dem Szenario ist das Bedienpult auswahlbereit, d. h. die angebotenen Dienste werden angezeigt und der Bankkunde kann seine Auswahl treffen.

□

## 6.3 Modellierungsregeln (II)

Zusätzlich zu den Definitionen und (Meta-) Klassendiagrammen und der oben vorgestellten Semantik von Use Case-Schritten und -Schrittgraphen geben wir in diesem Abschnitt noch einige weitere Modellierungsregeln für semantisch sinnvolle Use Case Schrittgraphen an. Wir kommen also nach den bisherigen „syntaktischen“ Regeln zu eher „semantischen“ Regeln, die insbesondere den Gebrauch der Vor- und Nachbedingungen klären. Während die „syntaktischen“ Regeln strukturelle Eigenschaften beschreiben und zum Standardrepertoire der Werkzeugunterstützung zählen, verkörpern die eher „semantischen“ Regeln sozusagen „Best Practices“ bei der Anforderungsermittlung (u.a. [IEEE830.93][Meyer85][FMD97][Fowler97a]) und sind nur eingeschränkt durch Werkzeuge prüfbar (vgl. [LamWil98]).

Zuerst einmal kann ein Szenario nur dann beginnen, wenn die Vorbedingung des Startschritts erfüllt ist. Damit dies zu Beginn der Bearbeitung jedes Use Cases gilt, geben wir die folgende Regel an.

**Regel 6.3.1 [Use Case-Vorbedingung]** Die Vorbedingung eines Use Cases  $uc$  muss die Vorbedingung seines Startschritts  $S(uc)$  implizieren:  $Pre_{uc} \rightarrow Pre_{S(uc)}$ .

Natürlich sollte es für einen Use Case kein Szenario geben, in dem die Aufgabe des Use Cases nicht abgearbeitet wird. Dies präzisieren wir sogleich in

**Regel 6.3.2 [Use Case-Nachbedingung]** Zu einem Use Case darf es kein Szenario geben, für das die UND-Verknüpfung aller<sup>1</sup> Nachbedingungen der im Szenario „ausgeführten“ Use Case Schritte nicht die Nachbedingung des Use Cases impliziert:

$$\forall sz \in \text{Szenarien}(uc): \bigwedge_{s \in S(sz)} Post_s \rightarrow Post_{uc}.$$

Die nächste Regel ist notwendig für die „Ausführbarkeit“ bzw. „liveness“ der Use Case Schrittgraphen (vgl. [DesObe96][HOR98]). Ist ein Use Case Schritt innerer Schritt (hat also mindestens einen Folgeschritt), so muss in jedem Fall nach der Bearbeitung seiner Aufgabe (mindestens) einer seiner Folgeschritte ausgewählt werden können:

**Regel 6.3.3 [Innerer Schritt-Nachbedingung]** Die Nachbedingung jedes (nicht als Endschrift markierten) inneren Schritts  $s$  eines Use Case Schrittgraphen  $uc$  muss die Disjunktion der Bedingungen seiner ausgehenden Kanten implizieren:

$$\forall s \in S(uc) \setminus SF(uc): Post_s \rightarrow \bigvee_{e \in \sigma(uc)} c(e).$$

---

<sup>1</sup> Aufeinander folgende Zustandsänderungen von Domänenobjekten behandeln wir in Abschnitt 11.3.

Zum Schluss geben wir noch einige Modellierungsregeln zum sinnvollen Gebrauch von Makroschritten an. Zunächst zwei Regeln über „erlaubte Aufrufe“. Use Case Schritte und Use Case Schrittgraphen entsprechen einer funktionalen Zerlegung, wobei Makroschritte die Benutzung eines (Sub-) Use Cases durch einen oder mehrere (Super-) Use Cases ermöglichen. Wir betrachten nun das Verhältnis der Aktoren des Makroschritts zu denen des referenzierten Use Cases. Die folgende Regel stellt sicher, dass die Aktoren des referenzierten Use Cases die Aufgabe des Makroschritts bearbeiten können.

**Regel 6.3.4 [Makro-Aktoren]** Die Aktoren eines Makroschritts müssen konform zu den Aktoren des referenzierten Use Cases sein:

$$\forall s \in S(uc): SA(s) = \text{Makro} \Rightarrow A(s) \subseteq_1 A(REF(s)).$$

Im Weiteren wollen wir die Anzahl der Folgeschritte eines Makroschritts nicht beschränken, sondern lediglich das Zusammenspiel von Makroschritten und den von ihnen referenzierten Use Cases regeln. Es lassen sich drei Fälle unterscheiden.

Als Erstes betrachten wir den Fall, dass ein Makroschritt innerer Schritt ist (und somit mindestens einen Nachfolgeschritt hat, s. Regel 5.4.5). Für jedes Szenario des referenzierten Use Cases müssen wir in der Lage sein, einen Folgeschritt des Makroschritts anzugeben, mit dem die Aufgabenbearbeitung fortgesetzt werden kann.

**Regel 6.3.5 [Makro-Innerer Schritt]** Ist ein Makroschritt  $s$  innerer Schritt, so muss die Nachbedingung jedes Endschriffs  $sf$  des referenzierten Use Cases  $REF(s)$  die Bedingung von mindestens einer ausgehenden Kante des Makroschritts implizieren:

$$\begin{aligned} \forall s \in S(uc): s \notin SE(uc) \wedge SA(s) = \text{Makro} \Rightarrow \\ \forall sf \in SE(REF(s)) \exists e \in \sigma(s): \text{Post}_{sf} \rightarrow c(e). \end{aligned}$$

Ist der Makroschritt  $s$  ein Blattschritt (also Endschrift ohne Nachfolgeschritte), so muss jedes  $s$  enthaltende Szenario des Use Cases in einem Endschrift des referenzierten Use Case  $REF(s)$  enden. Insbesondere muss also die Nachbedingung jedes Endschriffs von  $REF(s)$  die Nachbedingung von  $s$  implizieren.

**Regel 6.3.6 [Makro-Blattschritt]** Ist ein Makroschritt  $s$  Blattschritt (und somit natürlich Endschrift), so muss die Nachbedingung jedes Endschriffs  $sf$  des von  $s$  referenzierten Use Cases die Nachbedingung von  $s$  implizieren:

$$\begin{aligned} \forall s \in S(uc): SA(s) = \text{Makro} \wedge \sigma(s) = \emptyset \Rightarrow \\ \forall sf \in SE(REF(s)): \text{Post}_{sf} \rightarrow \text{Post}_s. \end{aligned}$$

Als letztes betrachten wir einen Makroschritt  $s$ , der Endschrift mit mindestens einem Nachfolgeschritt sei. In jedem  $s$  enthaltenden Szenario wird zunächst der von  $s$  referenzierte Use Case  $RefUC(s)$  „bearbeitet“. Abhängig von der Nachbedingung des erreichten Endschriffs von  $RefUC(s)$  müssen wir entscheiden können, ob das Szenario mit  $s$  endet oder aber in einem der Folgeschritte von  $s$  weitergeführt wird. Die folgende Regel 6.3.7 ergänzt Regel 6.3.3.

Regel 6.3.7 [Makro-Endschritt] Ist ein Makroschritt  $s$  Endschritt, aber kein Blattschritt, so muss die Nachbedingung mindestens eines Endschriffs  $sf$  des referenzierten Use Cases  $REF(s)$  nicht die Disjunktion aller Bedingung der ausgehenden Kanten des Makroschritts implizieren:

$$\begin{aligned} \forall s \in S(uc): s \in SE(uc) \wedge SA(s) = \text{Makro} \wedge \sigma(s) \neq \emptyset \Rightarrow \\ \exists sf \in SE(REF(s)): \neg(\text{Post}_{sf} \rightarrow \bigvee_{e \in \sigma(s)} c(e)). \end{aligned}$$

## 6.4 Makroschritte, „extends“, „uses“ und Generalisierung

Die in Use Cases bzw. Use Case Schritten modellierten Funktionalitäten haben unterschiedliche Komplexität, die von einfachen Informationseingaben bis hin zu komplizierten Interaktionen reicht (vgl. [Cockburn97]). Bei der Verfeinerung unterschiedlicher Use Cases ergeben sich oft ähnliche Teilaufgaben, die wir wiederum mit Use Cases bzw. Use Case Schrittgraphen modellieren. Im Use Case Modell werden solche Abhängigkeiten mit uses- und extends-Beziehungen ausgedrückt, deren Benutzung jedoch aufgrund der unklaren bzw. fehlenden Semantik problematisch erscheint (vgl. [Rumbaugh94][Cockburn97]).

Zur Präzisierung dieser Beziehungen haben wir das Konzept des Makroschritts eingeführt. Ein Makroschritt  $s_{\text{makro}}$  beschreibt eine (Teil-)Aufgabe eines Use Cases indirekt, indem er auf einen anderen, den benutzten Use Case Schrittgraphen  $RefUC(s_{\text{makro}})$  verweist. Mit dem Makroschritt-Konzept oder — genauer — mit der Inter-Use Case Referenzfunktion<sup>1</sup> und der Semantik von Use Case Schrittgraphen geben wir eine Semantik für die von Jacobson vorgeschlagenen extends- und uses-Beziehungen zwischen Use Cases an. Im Weiteren seien  $A$  und  $B$  Use Cases, und  $AG$  bzw.  $BG$  die  $A$  bzw.  $B$  verfeinernden Use Case Schrittgraphen.

**Definition 6.1**  $AG$  *referenziert*  $BG$ , wenn für alle Szenarien  $sz$  von  $AG$  gilt:  $sz$  enthält einen Makroschritt  $s_{\text{makro}}$  mit  $RefUC(s_{\text{makro}}) = B$ . Gilt  $A$  uses  $B$ , dann muss gelten:  $AG$  referenziert  $BG$  (Abb. 6.11 (a)).

□

**Definition 6.2**  $BG$  *erweitert*  $AG$ , wenn es ein Szenario  $sz$  von  $AG$  gibt, für das gilt:  $sz$  enthält einen Makroschritt  $s_{\text{makro}}$  mit  $RefUC(s_{\text{makro}}) = B$ . Gilt  $B$  extends  $A$ , dann muss gelten:  $BG$  erweitert  $AG$  (Abb. 6.11 (b)).

□

Die Semantik von uses und extends ist somit über Makroschritte und die von ihnen referenzierten Use Cases klar definiert. Die *Erweiterungspunkte* eines Use Case (vgl. [OMG97])

---

<sup>1</sup> „Rekursive“ Makroschritte bzw. Zyklen in der Inter-Use Case Referenzfunktion schließen wir aus; siehe Regel 5.4.6.



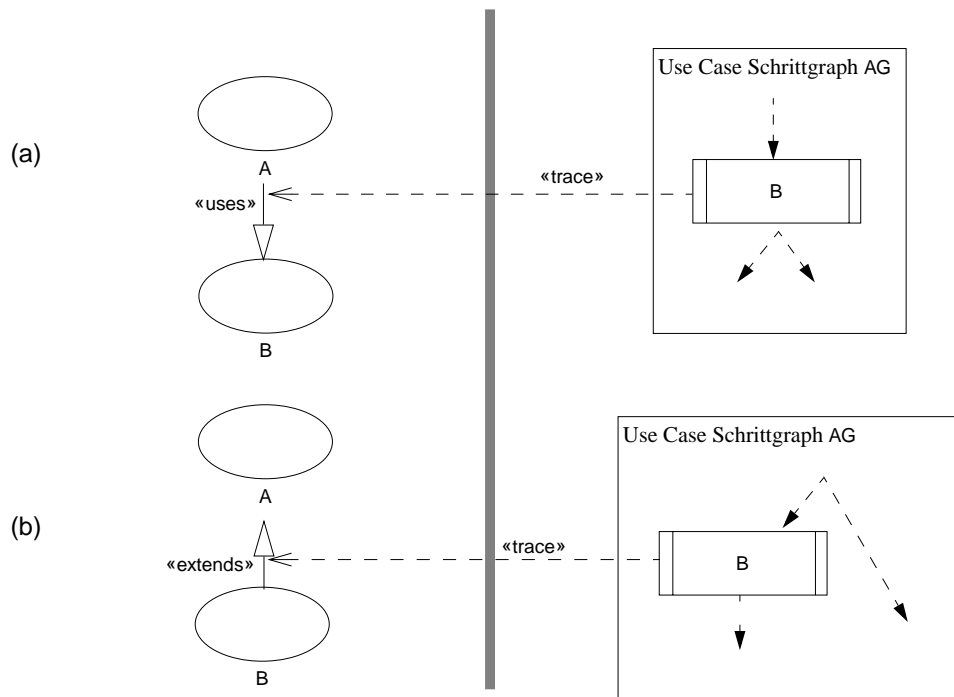


Abb. 6.11 Illustration der Semantik von «uses» (a) und «extends» (b)

sind durch Makroschritte in seinem Schrittgraph repräsentiert. Damit sind die in Kapitel 4 aufgezeigten Schwächen der beiden Beziehungen behoben.

Wir übertragen nun noch die *Generalisierung* von Use Cases auf die entsprechenden Use Case Schrittgraphen. Neben der Konformität der Vor- und Nachbedingungen sollen im spezialisierenden Schrittgraphen auch mindestens die Szenarien des generalisierten Use Case Schrittgraphen möglich sein. Wir werden in dieser Arbeit keine weiteren Einschränkungen bzgl. der Generalisierung von Use Case Schrittgraphen machen und erhalten:

**Definition 6.3** Es ist  $BG \leq AG$ , wenn für alle Szenarien  $sz_A$  von AG gilt: es gibt ein Szenario  $sz_B$  von BG, welches von  $sz_A$  generalisiert<sup>1</sup> wird. Gilt  $B \leq A$ , dann muss gelten:  $BG \leq AG$ . □

Die Ähnlichkeit der Generalisierung und der uses- sowie der extends-Beziehung ist unverkennbar, da sie sozusagen „erlaubte“ Änderungen bzw. Abwandlungen eines Use Cases beschreiben. Während jedoch bei der Generalisierung (ähnlich zu Jacobsons Interpretation der extends-Beziehung, vgl. [JCJ+92]) nur die Konformität der Vor- und Nachbedingungen gefordert ist und der Schrittgraph des spezialisierenden Use Cases beliebige weitere Aufgaben

<sup>1</sup>  $sz_A$  generalisiert  $sz_B$ , wenn  $sz_A$  durch Streichung endlich vieler Schritte aus  $sz_B$  entsteht und für jeden verbleibenden Makroschritt  $sm_B$  in  $sz_B$  und den entsprechenden Makroschritt  $sm_A$  in  $sz_A$  gilt:  $RefUC(sm_B) \leq RefUC(sm_A)$ .

enthalten bzw. Szenarien ermöglichen kann, sind bei der von uns präzisierten *uses-* und *extends-*Beziehungen (im Sinne der UML, vgl. [OMG99]) Änderungen bzw. Spezialisierungen nur noch für ganz bestimmte, durch die Erweiterungspunkte — hier Makroschritte — angezeigte Teile möglich.

Fassen wir die Ergebnisse dieses Teils der Arbeit zusammen: Use Case Schrittgraphen kombinieren die funktionalen Anforderungen, also das *Was* bzw. die Aufgaben unserer Anwendung, mit der Ablauflogik, also dem *Wann* und *Warum*. Als funktionales Konzept haben Use Cases natürlich kein Gedächtnis — wir haben daher Aufgaben und insbesondere den Kontext bzw. Zustand, in dem sie ausgeführt werden, rein deskriptiv anhand von Vor- und Nachbedingungen über Realwelt- bzw. Domänenklassen spezifiziert.

Die im Klassenmodell spezifizierten Operationen der Domänenklassen als eine weitere funktionale Sicht haben aufgrund ihres geringeren Abstraktionsgrades oft keinen direkten Bezug zu extern beobachtbaren Aufgaben und Abläufen. In anderen Worten sind die funktionalen Anforderungen oft über die Klassen und Operationen hinweg „verwischt“.

Es ergibt sich — ebenso wie bei den in ER- und Datenflussdiagrammen getrennten strukturellen und funktionalen Sichten der strukturierten Analyse (vgl. [Yourdon89]) — eine „Lücke“, die sich in Fragestellungen wie z.B. „Erfüllt das Klassenmodell die funktionalen Anforderungen?“ oder „Wie spezifiziert man die funktionalen Anforderungen im Klassenmodell?“ niederschlägt. Zur Beantwortung solcher Fragen werden wir in den nächsten beiden Kapiteln Use Case Schrittgraphen und Klassenmodelle koppeln.

# Kapitel 7

## Kopplung von Use Cases und Klassenmodell

*No approach [...] can be complete unless it accounts both for the function and the objects parts. [Meyer97]*

Wir spezifizieren mit Use Cases bzw. Use Case Schrittgraphen verhaltensbezogene (funktionale und ablaufbezogene) Aspekte der Anforderungen. Strukturelle bzw. „datenbezogene“ Aspekte werden im Klassenmodell spezifiziert. Da die Anforderungsermittlung auf den Soll-Zustand abzielt, beschreiben Domänenklassen im Kontext dieser Arbeit Objekte, die wir in dem betrachteten Geschäftsprozess *nach* der Einführung unserer Anwendung finden. Adäquat zu den Zielen der Anforderungsermittlung spezifiziert auch das Domänen-Klassenmodell das *Was* der Anwendung. Operationen in Domänenklassen sollen also nicht vorschreiben, wie bestimmte (Teil-) Aufgaben implementiert werden sollen, sondern präzisieren lediglich auf einer feingranulareren Ebene, welche Wirkung sie (auf Instanzen von Domänenklassen) haben.

Eines der Hauptprobleme von Ansätzen, die auf mehreren Modellen basieren, besteht darin, die involvierten Modelle konsistent zueinander aufzustellen und die Konsistenz bei Änderungen bzw. Erweiterungen zu erhalten (vgl. z.B. [RBP+91][Berard93][Behringer97]). Hierzu müssen die jeweiligen Modellierungselemente einander zugeordnet werden. Als Problem erweist sich hierbei die „Abstraktionslücke“ zwischen dem Use Case Modell und dem Domänen-Klassenmodell (s. S. 59, Abb. 5.1).

Um diese Lücke zu überbrücken, haben wir die Granularität von Use Cases auf die von Schritten in Use Case Schrittgraphen heruntergebrochen. Dies ermöglicht es, die Spezifikationen der verhaltensbezogenen und der strukturellen bzw. datenbezogenen Aspekte der Anforderungen aneinander koppeln — wir koppeln also Use Case Schrittgraphen und das Klassenmodell. Die resultierende integrierte Anforderungsspezifikation beschreibt sowohl die (flüchtigeren) funktionalen und ablaufbezogenen Aspekte der Anforderungen als auch deren (stabilere) strukturelle Fassetten. Wir nähern uns somit nach der bereits im Use Case

Schrittgraphen ausdrückbare kontextuelle- und Interaktionsinformation auch der systeminternen Information.

Im Wesentlichen identifizieren wir Interaktionsschritte als mögliche Verbindungspunkte zwischen Use Cases und Klassenmodell, denn

1. die Verantwortlichkeit der (Teil-) Aufgabe eines Interaktionsschritts muss auf Elemente des Klassenmodells abgebildet werden können (vgl. [WBW+90]) und
2. die erfolgreiche „Ausführung“ eines Use Case Schritts ist nur auf bestimmten, durch die Aufgabe bzw. die Vor- und Nachbedingung des Schritts eingeschränkten Objektkonstellationen möglich, die zum Klassenmodell konsistent sein müssen.

Nach einigen notationellen Vereinbarungen bezüglich der Elemente des Klassenmodells behandeln wir diese beiden Aspekte dann getrennt in den Abschnitten 7.2 bis 7.3.

## 7.1 Elemente des Klassenmodells

Wir treffen zuerst einige Vereinbarungen zur Referenzierung von Elementen im Klassenmodell, wobei wir uns an Meyer und Poetzsch-Heffter anlehnen (vgl. [Meyer97][Poetzsch97]).

Mit  $\mathbf{K}$  bezeichnen wir die Menge der in einem Klassenmodell enthaltenen Klassen, wobei wir „Standardklassen“ ([CoaYou90]) bzw. „primitive Typen“ ([GJS96]) wie z.B. **Boolean**, **Integer**, **Real** und **String** je nach Bedarf als in  $\mathbf{K}$  enthalten betrachten. Für eine Klasse  $\mathbf{k}$  aus  $\mathbf{K}$  bezeichnen wir mit  $\mathbf{k.attributes}$  die Menge aller in  $\mathbf{k}$  definierten Attribute.

Der Ausdruck  $\mathbf{k.operations}$  bezeichne die Menge aller in der Klasse  $\mathbf{k}$  definierten Operationen. Für eine Klasse  $\mathbf{k}$  und eine Operation  $o$  schreiben wir, falls gilt  $o \in \mathbf{k.operations}$ , abkürzend auch  $\mathbf{k}::o$ . OPS sei die Menge aller Operationen im Klassenmodell, also  $OPS \equiv \bigcup_{k \in \mathbf{K}} k.operations$ . Jede Operation  $o \in OPS$  wird durch ihren Namen  $o.name$  und ihre Signatur  $o.signature \in \mathbf{K}^{n+1}$  syntaktisch spezifiziert, wobei  $n \in \mathbb{N}$  die Anzahl der Parameter von  $o$  angibt und jede Operation genau einen Rückabewert liefert. Sei  $n > 0$ . Für  $o.signature = (\mathbf{k}_0, \dots, \mathbf{k}_n)$  sind  $\mathbf{k}_0, \dots, \mathbf{k}_{n-1}$  die Typen bzw. Klassen der Parameter und  $\mathbf{k}_n$  diejenige des Rückgabewerts der Operation. Abkürzend schreiben wir auch  $o.name(\mathbf{k}_0, \dots, \mathbf{k}_{n-1}):\mathbf{k}_n$ . Für  $s = (\mathbf{k}_0, \dots, \mathbf{k}_n) \in \mathbf{K}^{n+1}$  bezeichnen wir mit  $\{(\mathbf{k}_0, \dots, \mathbf{k}_n)\}$  die Menge der in  $s$  vorkommenden Klassen, d.h.  $\{(\mathbf{k}_0, \dots, \mathbf{k}_n)\} \equiv \{\mathbf{k} \in \mathbf{K} \mid \exists i \in N_n: \mathbf{k} = \mathbf{k}_i\}$ .

Neben den Signaturen spezifizieren wir die Semantik von Operationen durch Vor- und Nachbedingungen, die wir (in OCL) mit  $o.pre$  und  $o.post$  ansprechen. Die Semantik einer Klasse wird durch die Angabe von Klasseninvarianten<sup>1</sup> spezifiziert: Jedes Objekt der Klasse erfüllt vor und nach der Ausführung einer Operation die Klasseninvariante. Wir bezeichnen die Invariante einer Klasse  $\mathbf{k}$  mit  $\mathbf{k.invariant}$ . Vor- und Nachbedingungen sowie Klasseninvarianten

---

<sup>1</sup> Die in Invarianten und Vor- und Nachbedingungen verwendeten Operationen werden als Seiteneffektfrei vorausgesetzt bzw. angenommen (vgl. [Meyer97][OMG97]).

notieren wir, wie schon bei der Spezifikation von Use Cases und Use Case Schritten, entweder umgangssprachlich oder in OCL, wobei wir auch die von den betroffenen Klassen angebotenen Operationen verwenden. Bezüglich der OCL verweisen wir auf den OMG-Standard UML 1.1 ([OMG97]).

Ist eine Klasse  $l$  Unterklasse einer Klasse  $k$ , schreiben wir auch  $l \leq k$ . Ist  $l \neq k$  und  $l \leq k$ , so nennen wir  $l$  *eigentliche Unterklasse* von  $k$  und schreiben  $l < k$ . Folgt zusätzlich für alle  $k_s$  aus  $l \leq k_s \leq k$  entweder  $l = k_s$  oder  $k_s = k$ , so nennen wir  $l$  *direkte Unterklasse* von  $k$  und schreiben hierfür  $l \ll k$ . Stehen die beiden Klassen in keiner Vererbungsbeziehung zueinander, so schreiben wir  $l \nabla k$ . Für zwei Operationen  $l::o$  und  $k::o$  mit  $l::o.name = k::o.name$  und  $l::o.signature = k::o.signature$  sagen wir  $l::o$  *redefiniert (überschreibt)*  $k::o$ .

Für den Fall von im Rahmen einer Vererbung redefinierten Operationen beleuchten z.B. [FNZ96][Meyer97][Poetzsch97] in welcher Beziehung die Signaturen sowie Spezifikationen (Vor- und Nachbedingungen, Klasseninvarianten) der überschreibenden Operation zu denen der Überschriebenen stehen sollten. Wir setzen voraus, dass in Anforderungsspezifikationen ausnahmslos die konforme Vererbung<sup>1</sup> verwendet wird und geben — wie bei Aktoren — jeweils nur die am höchsten in der Vererbungshierarchie stehenden Klasse an.

Für eine Klasse  $k$  aus  $K$  ermitteln wir mit  $k.allAttributes$  die Menge aller die Objekte der Klasse  $k$  beschreibenden neu definierten, überschriebenen oder unverändert geerbten Attribute und mit  $k.allOperations$  dementsprechend die Menge aller in der Klasse  $k$  bekannten Operationen. Entsprechend den obigen Ausführungen ist eine Menge  $B$  von Klassen konform zu einer Menge  $A$  von Klassen, wenn jede in  $B$  enthaltene Klasse konform zu mindestens einer in  $A$  enthaltenen Klasse ist. In diesem Fall schreiben wir  $A \subseteq_1 B$ .

## 7.2 Klassenbereiche

Wir beschreiben die Aufgaben von Use Cases und Use Case Schritten durch Vor- und Nachbedingungen: Die Vorbedingung gibt die Voraussetzungen an, mit der die Aufgabe begonnen werden kann; die Nachbedingung gibt mögliche Auswirkungen bzw. Ergebnisse der Aufgabenbearbeitung an. Es stellt sich nun die Frage, ob die Bedingungen mit konkreten Objekten oder auf der Klassenebene spezifiziert werden sollen. Genauer gesagt müssen wir entscheiden, ob wir für jeden Use Case und jeden Use Case Schritt eine konkrete Objektkonstellation oder aber lediglich die Klassen eventuell involvierter Objekte und eine möglichst minimale Menge von Beschränkungen vorgeben sollen, die angeben, welche Objektkonstellationen vor bzw. nach Bearbeitung der Aufgabe vorliegen können.

---

<sup>1</sup> Eine Klasse  $B$  wird *konform* zu einer Klasse  $A$  genannt, wenn  $B$  jede Operation von  $A$  enthält, die Klasseninvariante von  $B$  die der Klasse  $A$  impliziert und die Vorbedingung jeder Operation von  $A$  die der entsprechenden Operation von  $B$  impliziert sowie umgekehrt die Nachbedingung jeder von  $B$  redefinierten Operation die der entsprechenden Operation von  $A$  impliziert (vgl. [FNZ96][Meyer97][Poetzsch97]).

Betrachten wir hierzu noch einmal die Beispiele in den vorherigen Kapiteln. Die Bedingungen beschreiben einerseits den Zustand konkreter Anwendungs- bzw. Realweltobjekte vor und nach der Bearbeitung der Aufgabe, also z.B. „Bankautomat Betriebsbereit“ oder „Karte Lesbar“. Andererseits werden auch über den Zustand einzelner Objekte hinaus Aussagen z.B. der Art „Es existiert ein (Objekt der Klasse) **Konto** mit ...“ oder „PIN dreimal falsch eingegeben“ gemacht.

Ein Use Case ist eine (statische) Beschreibung von (mehreren) möglichen Abläufen bzw. Szenarien, die bei der Bearbeitung einer Aufgabe auftreten können. Insbesondere soll ein Use Case Szenarien für den normalen Ablauf, alternative Abläufe und Ausnahmeabläufe beinhalten. Jedem Szenario liegt eine bestimmte Objektkonstellationen zugrunde. Als abstrahierende funktionale Beschreibung kann ein Use Case die Menge möglicher Objektkonstellationen (als „Zustand“ einer Anwendung bzw. als „Manifestation“ ihres bisherigen Ablaufs) nur abstrahierend, durch Angabe der Klassen und ggf. weiterer Bedingungen beschreiben.

In der Anforderungsermittlung bevorzugt man daher die Modellierung auf der Klassenebene (vgl. [CoaYou90] [Meyer97][RBP+91][WalNer95]). Dabei darf jedoch nicht vergessen werden, dass für den Benutzer — wenn überhaupt — nur konkrete Konstellationen von Realweltobjekten aus seinem Anwendungsgebiet verständlich sind (vgl. [Hasselbring98] [Züllighoven98]). Die Anforderungsspezifikation muss solche Sichten berücksichtigen, damit die Validierbarkeit durch den Benutzer gewährleistet ist. Wir kommen auf diese Aspekte in Kapitel 9 zurück.

Der Modellierung auf Klassenebene entsprechend bestimmen wir aus den (Teil-) Aufgabenbeschreibungen der Use Cases bzw. ihrer Schrittgraphen sowie der einzelnen Use Case Schritte zunächst jeweils bestimmte Mengen von Klassen, die wir als *Klassenbereiche* bezeichnen. Ein Klassenbereich enthält alle Klassen, deren Instanzen von der (Teil-) Aufgabe betroffen sein können, d.h. die erzeugt, gelöscht oder aber in ihrem Zustand geändert und zur Formulierung der Bedingungen für die Auswahl und Ausführung weiterer (Teil-) Aufgaben benötigt werden.

**Definition 7.1** Seien  $K$  die Menge aller Klassen eines Klassenmodells und  $uc = \{S, s_0, SE, \sigma\}$  ein Use Case Schrittgraph. Die Abbildung  $B: S \rightarrow 2^K$  ordne jedem Use Case Schritt eine Menge möglicher in die Aufgabe des Schritts involvierter Klassen zu.  $B(s)$  heißt *Klassenbereich* des Schritts  $s$ ; die Vereinigung der Klassenbereiche aller Schritte eines Use Case Schrittgraphen bezeichnen wir als *Klassenbereich des Use Case (Schrittgraphen)* und erweitern dementsprechend die obige Abbildung auf Use Cases zu  $B: UC \rightarrow 2^K$ ,  $B(uc) = \bigcup_{s \in S(uc)} B(s)$ .  $\square$

Hierbei betrachten wir im Klassenbereich eines Kontextschritts „reale“ Domänenobjekte und im Klassenbereich eines Interaktionsschritts deren anwendungsinterne Darstellung.

Wie wir in Kapitel 11 zeigen, sind die Klassenbereiche bei der „Simulation“ der Use Case Schrittgraphen während der Verifikation der Anforderungsspezifikation Grundlage für ein dem Sichtbarkeitsbereich bei Programmiersprachen (vgl. [Poetzsch91] [Meyer97]) bzw. den in einem Vertrag involvierten Klassen ([WBW+90]) entsprechendes Konzept. Wir geben noch zwei Regeln zur sinnvollen Gestaltung der Klassenbereiche an.

Die erste Regel gibt an, dass die in den Vor- und Nachbedingungen der Aufgabenbeschreibungen verwendeten Klassen Element der jeweiligen Klassenbereiche sein müssen.

**Regel 7.2.1 [Use Case (Schritt)-Klassenbereich]** Der Klassenbereich eines Use Case Schrittgraphen bzw. Use Case Schritts muss alle in seiner Beschreibung und der Vor- und Nachbedingung vorkommenden Klassen bzw. Objekte enthalten<sup>1</sup>.

Für die sinnvolle Modellierung der Klassenbereiche im Fall von Makroschritten geben wir die folgende unmittelbar einsichtige Regel an, die eine notwendige Bedingung zur Ausführung der Aufgabe des Makroschritts durch den „Aufruf“ des referenzierten Use Case Schrittgraphen formuliert.

**Regel 7.2.2 [Makro-Klassenbereich]** Der Klassenbereich des Makroschritts muss konform zu dem Klassenbereich des referenzierten Use Case Schrittgraphen sein:

$$\forall s \in S(uc): SA(s) = \text{Makro} \Rightarrow B(s) \subseteq B(REF(s)).$$

In konkreten Objektkonstellationen dürfen dementsprechend Instanzen von Unterklassen der in einem Klassenbereichen angegebenen Klassen vorkommen. Wir werden diesen Sachverhalt in Kapitel 15, „Black-box Test“, benutzen. Weitere Hinweise dafür, welche Klassen durch die Abbildung  $B$  den Klassenbereichen der Kontext- oder Interaktionsschritten zugeordnet werden, geben wir in Kapitel 8.

## 7.3 Wurzelklasse und Wurzeloperation

Die bisherige Zuteilung der Klassen zu Klassenbereichen der Use Cases und Use Case Schritte sowie die Beschreibung der Aufgaben gibt einen Überblick, welche Klassen bzw. Instanzen welcher Klassen von der (Teil-) Aufgabe betroffen sein können und ermöglicht eine grobe Abschätzung der Wirkungen einer Aufgabenausführung. Wir könnten diese Wirkung weiter präzisieren, indem wir in der Spezifikation des Use Cases die möglicherweise betroffenen (elementaren) Entitäten des Klassendiagramms angeben, z.B. „Assoziation  $a$  zwischen zwei Objekten der Klassen **A** und **B** aufgebaut“ oder „Attribut  $x$  einer Instanz der Klasse **Y** definiert“. Damit wäre allerdings die Kapselung der Klassen durchbrochen und immer noch keine echte Integration der funktionalen Sichten der Use Cases und des Klassenmodells

---

<sup>1</sup> Formal angeben bzw. automatisch prüfen können wir diese Regel nur für den Fall von in OCL oder einer anderen formalen Sprache notierten Bedingungen, sehen also in diesem Fall von einer Präzisierung der Regel ab.



Abb. 7.1 Interaktionsschritt mit Klassenbereich und Wurzeloperation Op\_1

hergestellt. Wir wollen die Verantwortlichkeiten der Aufgaben „objektorientiert“ auf Operation(en) von Klasse(n) verteilen. In diesem Sinne ordnen wir jedem Interaktionsschritt zunächst eine bestimmte Klasse im Klassenmodell und eine ihrer Operationen zu.

**Definition 7.2** Seien  $K$  die Menge aller Klassen eines Klassendiagramms und  $uc$  ein Use Case Schrittgraph. Die Abbildung  $WK: S(uc)|_{\text{Interaktion}} \rightarrow K$  ordnet jedem Interaktionsschritt  $s$  eine für seine Aufgabe verantwortliche Klasse  $WK(s) \in B(s)$  zu (d.h. die Klasse muss im Klassenbereich des Schritts enthalten sein). Wir nennen  $WK(s)$  die *Wurzelklasse* von  $s$ . Wir definieren zusätzlich eine Abbildung  $WO: S|_{\text{Interaktion}} \times K \rightarrow OPS$  mit  $WO(s, k) = o$ , für  $k = WK(s)$  und  $o \in k.operations$ , die jedem Interaktionsschritt eine für die Abbildung der Aufgabe im Domänen-Klassenmodell verantwortliche Operation der Wurzelklasse zuordnet. Diese Operation nennen wir *Wurzeloperation* des Interaktionsschrittes.

□

Analog zum Verhältnis der Spezifikationen eines Makroschritts und des von ihm referenzieren Use Cases sollte die Spezifikation eines Interaktionsschritts von der Spezifikation seiner Wurzeloperation befriedigt werden, d.h. die Vorbedingung des Interaktionsschritts impliziert die der Wurzeloperation, und deren Nachbedingung impliziert die des Interaktionsschritts.

Zur Darstellung der Klassenbereiche von Use Case Schritten erweitern wir die Use Case Schritt-Symbole um eine Sektion, in der wir die Klassen im Klassenbereich des Schritts textuell aufführen (Abb. 7.1). Hierbei kennzeichnen wir die Wurzelklasse des Schritts durch ein vorangesetztes @ und führen optional, vom Klassennamen durch einen Punkt abgetrennt, noch den Namen der Wurzeloperation auf. Wie in Abb. 7.4 dargestellt, visualisieren wir Klassenbereiche von Use Cases mit Kollaborationen (vgl. [OMG97]).

## 7.4 SCORES-Metamodell (II)

Wir erweitern das in Abb. 5.7 gezeigte Metamodell für Use Cases um entsprechende Primitive zur Modellierung von Wurzelklassen sowie Wurzeloperationen und erhalten das UML-Klassendiagramm in Abb. 7.2. Der schon in Abb. 5.7 gezeigte Teil des Metamodells ist **abgeschwächt** (bzw. bei Farbdruk rot) gezeichnet. Die erforderliche Assoziation für die Klassenbereiche (Assoziation scope) ist der abstrakten Klasse **ValidationClassifier** zugeordnet, von der die Klassen **StepGraph** und **UseCaseStep** erben. Objekte der Klassen **StepGraph** und **UseCaseStep** können dementsprechend mit Objekten der Klasse **Class** verbunden wer-



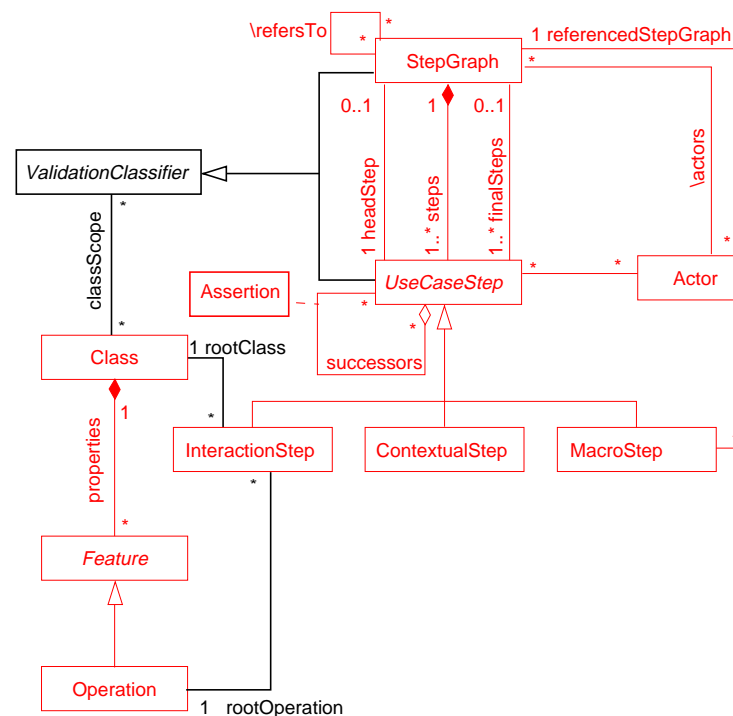


Abb. 7.2 Erweitertes Use Case Metamodell

den. Wurzelklasse und Wurzeloperation werden durch die Assoziationen **rootClass** und **rootOperation** zwischen Objekten der Klassen **InteractionStep** und **Class** bzw. **Operation** abgebildet.

## 7.5 Vollständigkeitsmetriken

Metriken dienen dem quantitativen Verständnis von Entwicklungsprodukten und sind mittlerweile integraler Bestandteil der objektorientierten Softwareentwicklung und des Software-Engineering allgemein (vgl. [Henderson96]). Hierbei werden Produktmetriken von Prozessmetriken unterschieden. Während erstere sozusagen als „Schnappschuss“ den Stand des Produkts zu einem bestimmten Zeitpunkt widerspiegeln, geben letztere statistische Tendenzen der Entwicklung über einen Zeitraum an und benötigen zu ihrer Interpretation zusätzliche Modelle. In Hendersons Worten:

Process contains a time rate of change and requires a process model to fully understand what is happening. [Henderson96]

Wir beschränken uns in dieser Arbeit auf Produktmetriken und wollen in diesem Abschnitt mit „Vollständigkeitsmetriken“ messen, wie eng das Klassenmodell und die Use Cases bzw. Use Case Schrittgraphen der Anforderungsspezifikation miteinander gekoppelt sind. Validie-

rungs- und Verifikationsmetriken in Form von Testkriterien geben wir in Kapitel 10 und 12 an.

Mit dem dreifachen Gleichheitszeichen „ $\equiv$ “ kennzeichnen wir die Definitionsgleichung einer Funktion oder einer Metrik. Wir definieren die Metriken (ebenso wie die der in Kapitel 10 und 12 angeführten Validierungs- und Verifikationsmetriken) mit mengentheoretischen bzw. prädikatenlogischen Ausdrücken sowie OCL-Ausdrücken. Wir beziehen uns in diesen Ausdrücken auf die SCORES-Modellierungsprimitive und verweisen diesbezüglich auf das (partielle) Metamodell in Abb. 7.2.

Die OCL ermöglicht es, Anfragen an ein Modellierungselement zu richten. Für die Elemente des Klassenmodells legen wir zusätzlich die UML-Semantikdefinitionen zugrunde (vgl. [OMG97]). Mengen von Modellierungselementen zurückgebende OCL-Anfragen identifizieren wir dabei mit den entsprechenden Rückgabemengen. Dementsprechend verknüpfen wir solche OCL-Ausdrücke auch mit mengentheoretischen Operatoren.

*Beispiel 7.1.* Für einen Use Case Schritt `ucs` liefert der OCL-Ausdruck `ucs.Bereich` den Klassenbereich des Schritts, also eine Menge mit Elementen vom UML-Typ **Class**. Im Kontext eines bestimmten Modells ergibt der Ausdruck

$$\bigcup_{v \in \text{ValidationClassifier.allInstances}} v.\text{ClassScope}$$

die Menge aller Klassen, die im Bereich (Metassoziations `ClassScope`) zumindest eines Use Cases oder Use Case Schritts enthalten sind. Der Ausdruck

$$\text{ValidationClassifier.allInstances} \rightarrow \text{size}$$

ergibt die Anzahl aller im Modell enthaltenen Instanzen vom Typ **ValidationClassifier**.

□

Im Folgenden bezeichne `uc` einen Use Case Schrittgraphen und **UC** die Menge aller Schrittgraphen der Anforderungsspezifikation.

Als erste, einfache Metrik messen wir mit der *einfachen Klassenüberdeckung*  $c_{0}^{\text{KM},K}: \mathbf{K} \times \mathbf{UC} \rightarrow \mathbb{R}$ , welcher Prozentsatz der Klassen im Klassenmodell als Wurzelklasse der Interaktionsschritte in den Use Case Schrittgraphen vorkommen. Zunächst ermitteln wir mit der Operation **StepGraph::rootClasses():Set of Class** die Wurzelklassen der Schritte eines Use Case Schrittgraphen.

$$\text{uc.rootClasses} \equiv \bigcup_{s \in \text{uc.Steps} \cap \text{InteractionStep.allInstances}} s.\text{rootClass} \quad (7.1)$$

Hiermit können wir die einfache Klassenüberdeckung angeben zu

$$c_0^{KM,K} \equiv \frac{|\bigcup_{uc \in \text{UseCase.allInstances}} uc.rootClasses|}{\text{Class.allInstances} \rightarrow \text{size}} \quad (7.2)$$

Es ist  $c_0^{KM,K} = 100\%$ , wenn jede Klasse als Wurzelklasse in mindestens einem Interaktionsschritt der Use Case Schrittgraphen vorkommt.

Von hier an verzichten wir auf die Angabe detaillierter Signaturen für die definierten Operationen bzw. Funktionen, da alle Metriken den Wertebereich  $\mathbb{R}$  haben und der Definitionsbereich jeweils durch die betroffenen Modellierungs- bzw. Validierungselemente festgelegt wird.

Sicherlich werden nicht alle Klassen als Wurzelklassen verwendet, d.h. im Normalfall wird keine 100%-ige einfache Klassenüberdeckung erreicht. Als *vollständige Klassenüberdeckung*  $c_1^{KM,K}$  geben wir daher an, welcher Prozentsatz aller Klassen in zumindest einem Klassenbereich eines Use Case Schritts enthalten ist. Zuvor ermitteln wir wieder, welche Klassen in den Klassenbereichen der Schritte eines Use Cases enthalten sind.

$$uc.ClassScope \equiv \bigcup_{s \in uc.Steps} s.classScope \quad (7.3)$$

Die vollständige Klassenüberdeckung ergibt sich dann zu:

$$c_1^{KM,K} \equiv \frac{|\bigcup_{uc \in \text{UseCase.allInstances}} uc.ClassScope|}{\text{Class.allInstances} \rightarrow \text{size}} \quad (7.4)$$

Es ist  $c_1^{KM,K} = 100\%$ , wenn jede Klasse in mindestens einem Klassenbereich eines Schritts der Use Case Schrittgraphen vorkommt.

Als weitere Metrik messen wir mit der *einfachen Operationsüberdeckung*  $c_0^{KM,O}$ , welcher Prozentsatz der Operationen im Klassenmodell Wurzeloperationen von Use Case Schritten sind. In einem ersten Schritt ermitteln wir für einen Use Case die Wurzeloperationen seiner Schritte.

$$uc.rootOperations \equiv \bigcup_{s \in uc.Steps \cap \text{InteractionStep.allInstances}} s.rootOperation \quad (7.5)$$

Hiermit können wir die einfache Operationsüberdeckung angeben zu

$$c_{0}^{KM,O} \equiv \frac{|\bigcup_{uc \in \text{UseCase.allInstances}} uc.rootOperations|}{|\text{Operation.allInstances}| \rightarrow \text{size}} \quad (7.6)$$

Mit diesen und ähnlichen Metriken können wir die Belastbarkeit der „Brücke“ zwischen Use Cases bzw. deren Schrittgraphen und dem Klassenmodell zeigen. Der Eingangs geforderte Übergang zwischen den Modellen bzw. den Sichten der Anforderungen ist somit geschaffen.

*Beispiel 7.2.* Zur Verdeutlichung der eingeführten Begriffe *Klassenbereich*, *Wurzelklasse* und *Wurzeloperation* betrachten wir den Use Case Anmelden des Bankautomaten. Abb. 7.3 skizziert das im Weiteren zugrundeliegende (teilweise unvollständige) Klassenmodell. Den Use Case Anmelden zusammen mit den in seinem Klassenbereich enthaltenen Klassen zeigt Abb. 7.4.

Die Kopplung von Use Case Schrittgraph und Klassenmodell beginnt mit dem Kontextschritt Kreditkarte Eingeben. Dieser setzt auf den ersten Blick lediglich das physikalische Vorhandensein des (funktionsbereiten Kartenlesers im) Bankautomaten voraus. Wir ordnen diesem Schritt also den Klassenbereich {**Kartenleser**} zu. Der Interaktionsschritt Kreditkarte Lesen

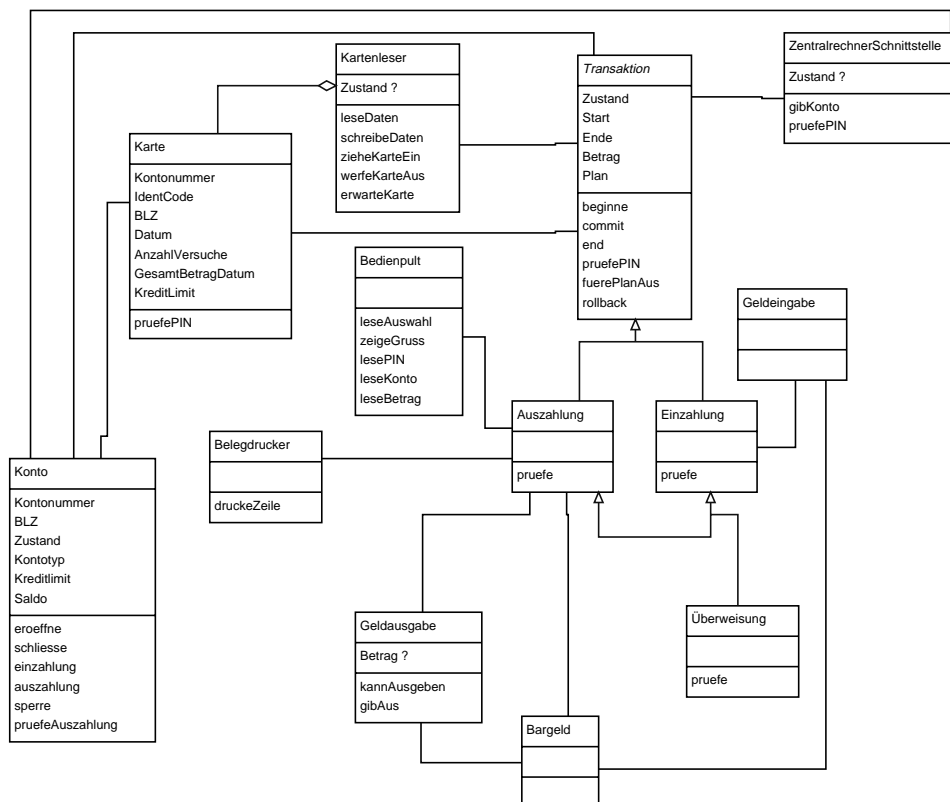


Abb. 7.3 Klassenmodell des Bankautomaten

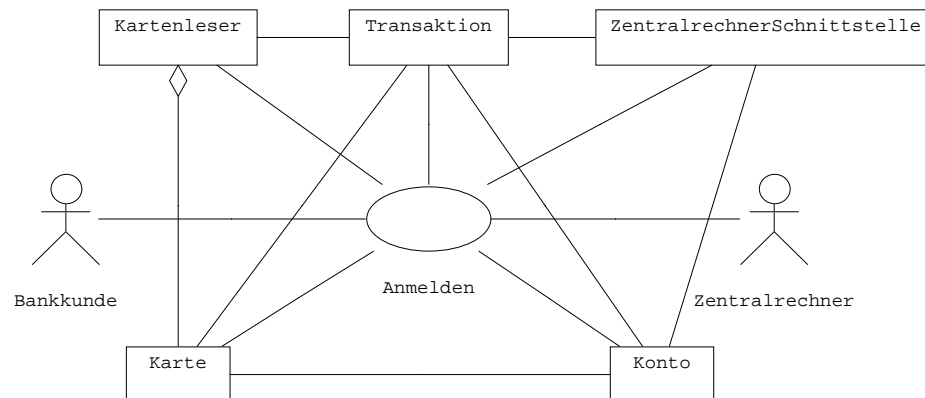


Abb. 7.4 Use Case Anmelden: Klassenbereich

beginnt eine Transaktion und setzt einen betriebsbereiten Kartenleser voraus, der die im Leseschlitz befindliche (physikalische) Kreditkarte liest und mit den Daten der Kreditkarte ein Objekt der Klasse **Karte** erzeugt. Wir ordnen diesem Schritt also den Klassenbereich {**Transaktion**, **Kartenleser**, **Karte**} zu. Ähnlich verfahren wir mit den übrigen Schritten des Use Case Schrittgraphen und erhalten die in Abb. 7.5 den Schritten zugeordneten Klassenbereiche, Wurzelklassen und Wurzeloperationen.

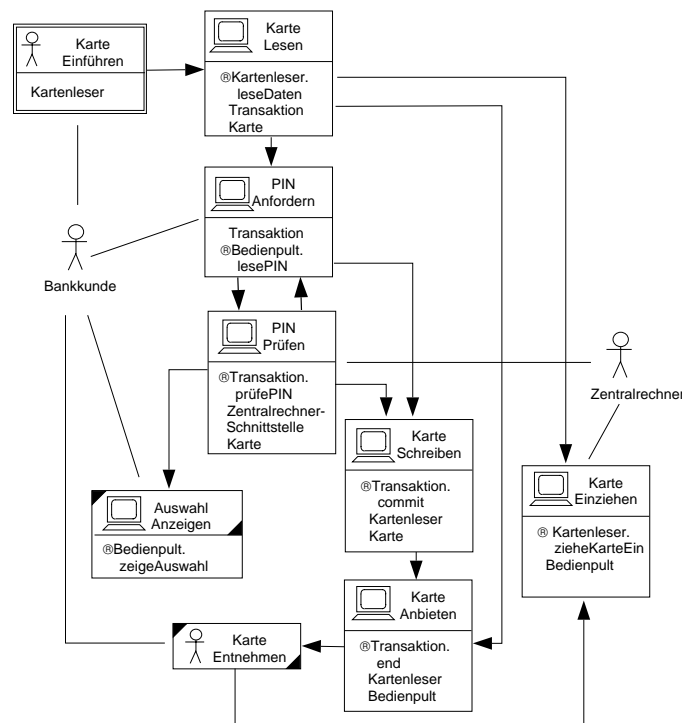


Abb. 7.5 Use Case Schrittgraph mit Klassenbereichen, Wurzelklassen und Wurzeloperationen

Wir ermitteln nun die Werte der Vollständigkeitsmetriken. Von den 13 Klassen (Abb. 7.3) sind 3 als Wurzelklassen zugeteilt (**Kartenleser**, **Bedienpult** und **Transaktion**). Die einfache Klassenüberdeckung ergibt sich somit zu  $c_0^{KM,K} = \frac{3}{13} = 23,08\%$ .

In den Klassenbereichen kommen zusätzlich noch die Klassen **Karte** und **Zentralrechner-Schnittstelle** vor. Wir erhalten also eine Klassenüberdeckung von  $c_1^{KM,K} = \frac{5}{13} = 38,46\%$ .

Von den 29 im Klassenmodell definierten Operationen sind 6 als Wurzeloperationen zugeteilt, so dass wir eine einfache Operationsüberdeckung von erhalten  $c_0^{KM,O} = \frac{6}{29} = 20,69\%$ . □

## 7.6 Exkurs: Use Case Schrittgraphen und Aktivitätsdiagramme

In der im September 1997 (zeitgleich zu unseren Entwicklungen, s. [KPW97]) veröffentlichten Version 1.1 der UML ist das Metamodell der Verhaltenssicht (Common Behavior) erweitert worden ([OMG97]). Die Autoren schlagen den Einsatz von Aktivitätsdiagrammen zur Modellierung einzelner Operationen im Klassenmodell bis hin zur Spezifikation von vollständigen Workflows vor. Das Aktivitätsdiagramm weist Ähnlichkeiten zum Use Case Schrittgraphen auf, da beide den Kontrollfluss von Aktivität zu Aktivität beschreiben. Wir lesen:

Activity diagrams accomplish much of what people want from DFDs, and then some; activity diagrams are also usefull for modeling workflow. The activity diagram [...] is similar to the work flow diagrams developed by many sources including many pre-OO sources. [OMG97] (Summary)

The activity diagram is a variation of a state machine in which the states represent the performance of actions or subactivities and the transitions are triggered by the completion of the actions or subactivities. [OMG97] (Semantics)

In anderen Worten: das Aktivitätsdiagramm ist eine spezielle Zustandsmaschine, in der Zustände zu Aktivitäten redefiniert sind und Übergänge das Ende einer Aktivität darstellen.

Wir wollen mit Use Case Schrittgraphen das (interaktive) Verhalten *mehrerer* Entitäten (Aktoren und Anwendung) modellieren. Das Aktivitätsdiagramm (und andere Arten von Zustandsmaschinen) dient in erster Linie zur Modellierung von prozeduralem (reaktiven) Verhalten *einer* Entität (Operation, Klasse, Anwendung, Betrieblicher Prozess, ...). Wir lesen in der UML:

The entire activity diagram is attached (through the model) to a class or to the implementation of an operation or a use case. The purpose of this diagram is to focus on flows driven by internal processing (as opposed to external events). Use activity diagrams in situations where all or most of the events represent the completion of internally-generated actions (that is, procedural flow of control). [OMG97]

Somit können in einem Aktivitätsdiagramm entweder kontextuelle, interaktions- oder systeminterne Information, nicht aber mehrere Informationsarten gleichzeitig abgedeckt werden.

Zusätzlich zu diesen eher konzeptuellen Mängeln können wir Use Case Schrittgraphen aus weiteren Gründen nicht ohne Änderungen des UML-Metamodells auf Aktivitätsdiagramme abbilden:

- ❑ Referenzen von Zuständen (Aktivitäten) auf vollständige Aktivitätsdiagramme sind nicht erlaubt. Daher lassen sich die *extends*- und *uses*-Beziehungen nicht auf Aktivitätsdiagramme fortsetzen, was die Ausdrucksstärke des Modells stark beeinträchtigen würde. Die Möglichkeit, in sogenannten Untermaschinen-Zuständen (Submachine-States) auf vollständige Zustandsdiagramme zu verweisen, ist eher auf syntaktischer Ebene zu sehen und wird semantisch auf eine textuelle Ersetzung des Untermaschinen-Zustands durch das referierte Zustandsdiagramm zurückgeführt. Hierarchien sind auf ein „Ober“-Aktivitätsdiagramm beschränkt und können nicht zur Redundanzvermeidung verwendet werden.
- ❑ Zustände sind nicht Unterklasse der UML-Metaklasse **Classifier**. Damit können einzelnen Zuständen keine Aktoren oder Elemente des Klassenmodells zugeordnet werden<sup>1</sup>, was die feingranulare (Nach-) Verfolgbarkeit der Anforderungen während des Entwurfs bis hin zur Implementation erschwert.

Diese — und weitere — Probleme wurden von der „UML Task Force“ der Object Modeling Group (OMG) erkannt und sollen in späteren Versionen der UML verbessert werden (vgl. [OMG-AT98]). Nach entsprechenden Änderungen können wir die von uns vorgeschlagenen Use Case Schrittgraphen mit Aktivitätsdiagrammen und den drei Typen von Use Case Schritten entsprechenden Stereotypen modellieren. Die in dieser Arbeit eingeführte Semantik und Methodik wäre dann vollständig mit der UML verträglich, was insbesondere der Forderung nach einer einheitlichen Modellierungssprache entgegenkommt (vgl. [Wiegers98]).

---

<sup>1</sup> Die sogenannten „Schwimmbahnen“ (Swimlanes) in Aktivitätsdiagrammen dienen als rein graphisches Element lediglich zur visuellen Zuteilung von Aktivitäten zu beteiligten Aktoren, haben jedoch keine weitere Semantik und spiegeln sich auch nicht im Metamodell wieder.

# Kapitel 8

## Methodisches Vorgehen

*Most methodologists now agree that user-centred analysis is the best way to solve the right problem. [Rumbaugh94]*

Geeignete Modellierungselemente sind zwar notwendige, aber keine hinreichenden Voraussetzungen für eine im industriellen Alltag anwendbare Software Engineering-Methode. Für den erfolgreichen praktischen Einsatz einer Methode stellen Hinweise zum methodischen Vorgehen und eine geeignete Werkzeugunterstützung wichtige Ergänzungen der Modellierungskonzepte dar. Ohne diese Hilfestellungen wird eine Software Engineering-Methode in der heutigen Zeit kaum Akzeptanz finden. Deshalb ergänzen wir in diesem Kapitel unsere bisherigen Ausführungen durch einige Bemerkungen zum methodischen Vorgehen bei der Spezifikation der Anforderungen. Wir betrachten hierbei separat die vier Aspekte funktionale Zerlegung, externes Verhalten, strukturelle Zerlegung und internes Verhalten. Besonderen Wert legen wir bei der funktionalen Zerlegung und der Spezifikation des externen Verhaltens der Anwendung auf die Einbeziehung der (zukünftigen) Benutzer.

### 8.1 Funktionale Zerlegung

Die Anwendung soll den z. B. in einer Produktskizze grob beschriebenen (Teil von einem) Geschäftsprozess unterstützen. Anhand der organisatorischen Aspekte dieses Geschäftsprozesses werden entsprechende Benutzer bzw. von der Anwendung „betroffene“ Personen oder Personengruppen ermittelt. Für diesen Kreis von Personen (und ggf. anderen Anwendungen) bilden wir anhand von Aufgaben und Verantwortlichkeiten Rollen, die wir dann wiederum zu Aktoren abstrahieren. Ergebnis dieser Tätigkeit sind Aktoren und Aktorhierarchien mit ihren jeweiligen in den Geschäftsprozess eingebetteten Aufgaben.

Die Aufgaben der einzelnen Aktoren werden anhand der funktionalen Sicht des Geschäftsprozesses zu Use Cases gruppiert (vgl. Kapitel 2). Analytiker und bestimmte Aktoren reprä-



sentierende Benutzer erstellen gemeinsam die textuelle Beschreibung der Aufgaben der Use Cases (vgl. [RolAch98]). Zusätzlich werden die Aufgaben funktional in Teilaufgaben zerlegt, die wiederum Aktoren zugeordnet werden. Dieser Prozess wird fortgesetzt, bis die resultierenden Teilaufgaben von (normalerweise) einem Akteur in einer zeitlich und räumlich zusammenhängenden Aktion unter Beachtung nur weniger, einfacher Geschäftsregeln bearbeitbar sind. Analytiker und Anwender spielen hierzu z.B. einfache Geschäftsvorfälle durch und gleichen die funktionale Zerlegung entsprechend an. Jede dieser Teilaufgaben bildet einen Use Case Schritt.

Während dieser Tätigkeiten sammeln Analytiker Begriffe und Definitionen und fassen sie in einem „normsprachlichen“, organisationsweit gültigen Datenlexikon zusammen, welches im weiteren Verlauf der Anforderungsermittlung fortgeschrieben wird (vgl. [Züllighoven98]). Parallel dazu werden anhand der Datenaspekte erste, sozusagen „ER-diagrammartige“ Skizzen des Klassenmodells angefertigt.

## 8.2 Externes Verhalten

Während der funktionalen Zerlegung der Use Cases werden die kontextuelle Information und die Interaktionsinformation in der SCORES-Anforderungsspezifikation erfasst. Analytiker und Benutzer simulieren unter Berücksichtigung der Ablaufaspekte des Geschäftsprozess-Modells z.B. in Rollenspielen wieder gemeinsam grundlegende Geschäftsvorfälle. Die Produktskizze und die operationalen Aspekte des Geschäftsprozesses (vgl. Abschnitt 2.1) geben Hinweise zur Charakterisierung der Schritte in kontextuelle Schritte und Interaktionsschritte. Während kontextuelle Schritte „beliebig“ umfangreiche Aufgaben beschreiben, die in ihrem Verlauf nicht die Benutzung der zu spezifizierenden Anwendung erfordern, orientieren sich Interaktionsschritte in ihrer Granularität an der Verantwortlichkeit einer bestimmten Klasse bzw. einiger weniger, eng gekoppelter Klassen im Klassenmodell („benannte Domänenobjekte“, vgl. [KSV96]). Extern entspricht ein Interaktionsschritt in etwa einer *Szene* ([KSV96]), also einem „Fenster“ bzw. einer „Maske“ der Benutzungsoberfläche (vgl. [VosNen98]).

Aus den durchgespielten (bis jetzt textuell spezifizierten) Geschäftsvorfällen extrahieren die Analytiker die grundlegende Ablaufstruktur, also die Übergangsbedingungen der Kanten in den Use Case Schrittgraphen. Jeder Geschäftsvorfall muss auf ein Szenario, also einen Pfad in einem Use Case Schrittgraph abgebildet werden können. Vor- und Nachbedingungen der Use Cases und Use Case Schritte bzw. die Übergangsbedingungen der Kanten werden zunächst umgangssprachlich notiert und dann weiter formalisiert.

(Teil-) Szenarien, die in mehreren Use Case Schrittgraphen vorkommen, können zu (Sub-) Use Cases zusammengefasst und mit dem Makroschritt-Mechanismus wiederverwendet werden. Auf der anderen Seite bilden wir auch die im Use Case Modell spezifizierten extends- und die uses-Beziehungen auf (Sub-) Use Cases und entsprechende Makroschritte ab. Dari-

ber hinaus können Generalisierungen zwischen Use Cases gebildet und auf die entsprechenden Schrittgraphen übertragen werden.

### 8.3 Struktureller Aufbau

Sind die betrachteten Use Cases und die sie verfeinernden Schrittgraphen stabil, konzentrieren sich die Analytiker auf das Klassenmodell, d.h. auf die „statische“ systeminterne Information. Sie halten zunächst fest, welche „Verantwortlichkeiten“ (*responsibilities*, vgl. [WBW+90]) die Aufgabe eines Schritts umfasst. Hierzu verwenden die Analytiker das Datenlexikon, die textuelle Beschreibung der Aufgabe und die Vor- und Nachbedingung. Erste Kandidaten für Verantwortlichkeiten sind z.B:

- ☐ Verben in den textuellen Spezifikationen.
- ☐ Mehrfach angesprochene „Eigenschaften“ von Domänen-Objekten.
- ☐ Alternative Pfade bzw. Ausnahmebehandlungen in den Use Case Schrittgraphen.

Zur Auswahl von Klassen werden die textuellen Beschreibungen der Use Cases herangezogen ([JCJ+92]). Zusätzlich können (und sollten) die in anderen Methoden zur Anforderungsermittlung gegebenen Hinweise und Ideen zur Auffindung von Klassen beachtet werden, wie z.B. Herauskristallisieren der Schlüsselabstraktionen des Anwendungsbereichs durch Hauptwortanalyse (vgl. [Meyer97][Booch94][CooYou90]), CRC-Karten ([WBW+90]), Leitbilder, Metaphern und Analogien ([Züllighoven98]) und weitere Heuristiken. Weiterhin ergänzen und verfeinern die Analytiker das Klassenmodell, d.h. fügen fehlende Klassen hinzu, vervollständigen Vererbungshierarchien und Objektbeziehungen.

Welche Klassen den Klassenbereichen der Use Case Schritte zugeordnet werden, leiten die Analytiker aus den Beschreibungen und Verantwortlichkeiten der Schritte ab. Ist eine Klasse in der Beschreibung erwähnt, wird sie zunächst dem Klassenbereich hinzugefügt. Auswirkungen der (Teil-) Aufgabe des Schritts auf Instanzen der Klasse werden — soweit ersichtlich — notiert. Mit den der Klasse zugeteilten Verantwortlichkeiten werden Attribute und Operationen ermittelt und spezifiziert. Vereinzelt können Interaktionsschritten schon Wurzeloperationen zugewiesen werden.

### 8.4 Internes Verhalten

Zur Spezifikation des internen Verhaltens (also der „dynamischen“ systeminternen Information) werden die Interaktionsschritte der Use Cases durch die Auswahl einer Wurzelklasse aus dem Klassenbereich und einer entsprechenden Wurzeloperation an das Klassenmodell gekoppelt. Analytiker spielen z.B. die bei der funktionalen Zerlegung dokumentierten Szenarien erneut durch, um dabei die Aufgaben der Interaktionsschritte mit den Verantwortlichkeiten der Klassen bzw. den Operationen abzugleichen. Operationen werden entsprechend den

Aufgaben der Interaktionsschritte angepasst oder im Klassenmodell ergänzt. Besondere Berücksichtigung finden Szenarien, die Sonderfälle behandeln.

In diesem Aufgabenbereich vervollständigen die Analytiker die feingranularen und änderungsanfälligeren Spezifikationen der Operationen. In vielen Fällen deckt die Spezifikation einer komplexen Operation das Fehlen weiterer (interner) Operationen und funktionaler Abhängigkeiten auf.

Die Werte der im vorigen Abschnitt vorgestellten Metriken reflektieren den Stand der Modellierung. Als Faustregel können wir z.B. sagen, dass mit der Validierung der Anforderungsspezifikation (s. Kapitel 9) nicht eher begonnen werden sollte, bis die Klassenüberdeckung  $c_1^{KM,K} = 1$  ist, also jede Klasse in zumindest einem Bereich eines Use Cases bzw. Use Case Schritts enthalten ist.

## 8.5 Administrative Tätigkeiten

In SCORES beinhalten die administrativen Tätigkeiten insbesondere die Vorbereitung der Ressourcenplanung. Hierzu bestimmen die Analytiker und Tester ein Profil für jeden Use Cases z.B. nach folgenden Kriterien (vgl. [Lyu96], [McGregor97]):

- ❑ Die Kritikalität  $C$  repräsentiert die (schlimmst-) möglichen Auswirkungen bei Ausfall oder fehlerhafter Bearbeitung des Use Cases. Sie wird durch Technik-Folgeabschätzungen in Zusammenarbeit mit den Benutzern festgelegt.
- ❑ Das technische Risiko  $R_T$  (wie kompliziert wird die Realisierung des Use Case) wird aus der Komplexität des Schrittgraphen und des Klassenbereichs abgeleitet.
- ❑ Das geschäftliche Risiko  $R_B$  (inwiefern ist der Absatz/die Akzeptanz bei Nichterfüllung gefährdet) wird z.B. durch Befragungen von (prospektiven) Benutzern ermittelt.
- ❑ Das Projektrisiko  $R_P$  (inwiefern gefährdet die Nichterfüllung den weiteren Projektfortschritt) ergibt sich aus der Anzahl der über den Makro-Mechanismus vom betrachteten Use Case abhängigen Use Cases sowie der Grösse des Klassenbereichs.

Die Zuteilung der Profile zu den ersten Skizzen der Use Cases erfolgt aus der Erfahrung früherer Projekte und Risikoabschätzungen des Projektmanagements. Drei Werte *low*, *medium* und *high* reichen erfahrungsgemäß zur Quantifizierung der einzelnen Faktoren des Profils aus, zudem diese auch nicht unabhängig voneinander sind (s. z.B. [Lyu96]). Indem wir diese drei Werte auf die natürlichen Zahlen 1, 2 und 3 abbilden, können wir z.B. mit Formel 8.1 jedem Use Case  $uc$  ein bestimmtes Gewicht  $w_{uc}$  zuweisen.

Für die Planung der Entwicklung übertragen wir die Gewichte der Use Cases über die Klassenbereiche zunächst auf Elemente des Domänen-Klassenmodells und nehmen erste, grobe

$$w_{uc} = \frac{R_T + R_B + R_P}{3} + F \cdot R_B + C \cdot R_T \quad (8.1)$$

Aufwandsschätzungen vor. Hierbei berechnen wir das Gewicht einer Klasse z.B. als Mittelwert oder — konservativer — als Maximum der Gewichte aller Use Cases, in deren Klassenbereich die Klasse vorkommt. Feinere Zuteilungen können z.B. auch noch Wurzelklassen bzw. Wurzeloperationen berücksichtigen. Über die Verfolgungsrelation (s. Kapitel 13) können wir die Gewichte später auf entsprechende Elemente des Entwurfs und der Implementation abbilden. Die am Anfang der Entwicklung grob abgeschätzten Profile der Use Cases werden dann im Laufe der Entwicklung verfeinert, wobei auch die Komplexität der entsprechenden (Entwurfs- und Implementations-) Klassen mit einfließt.

*Beispiel 8.1.* Das Gewicht des Use Cases Anmelden (Abb. 5.3) mit dem Profil  $F = high$ ,  $C = medium$ ,  $R_T = high$ ,  $R_B = high$  und  $R_P = high$  ergibt sich zu

$$w_{\text{anmelden}} = \frac{3+3+3}{3} + 3 \cdot 3 + 2 \cdot 3 = 18$$

□

## 8.6 Zwischenfazit

Wir haben in diesem Teil das Use Case Konzept präzisiert bzw. verfeinert und in Use Case Schrittgraphen mit der Ablaufsicht verschmolzen. Diese ermöglichen den nahtlosen, verfolg-  
baren Übergang von Use Cases zu Domänenklassen. Die SCORES Anforderungsspezifikation deckt die kontextuelle und die Interaktionsinformation ab und berücksichtigt somit auch Aspekte der umfassenden Geschäftsprozesse. Bei der Prüfung der Anforderungsspezifikation wird dann auch die bisher nur rudimentär erfasste systeminterne Information vervollständigt.

Für die Anforderungsspezifikation ermöglicht uns die Syntax und Semantik der SCORES-Elemente Use Case Schritt, Use Case Schrittgraph und Klassenbereich sowie Wurzelklasse und Wurzeloperation die Formulierung von Regeln, die den Modellierungsraum einschränken und zu aussagekräftigeren Anforderungsspezifikationen führen. Darüber hinaus können wir für den Grad der Kopplung des Use Case Modells mit dem Klassenmodell einige Vollständigkeitsmetriken angeben.

Im nächsten Teil betrachten wir die qualitätssichernden Tätigkeiten bei der Anforderungsermittlung. Einerseits muss sichergestellt werden, dass die Anforderungen und Wünsche der Benutzer in der Anforderungsspezifikation fachlich vollständig und richtig berücksichtigt wurden — Analytiker und Benutzer validieren dazu das Use Case Modell. Andererseits prüfen Analytiker und Tester, ob die Anforderungsspezifikation konsistent und formal vollständig ist — hierzu ermöglicht SCORES insbesondere die Verifikation des Klassenmodells gegen die Use Cases. In Teil IV zeigen wir dann, wie sozusagen als „Mehrwert“ der Qualitätssicherung für die SCORES-Anforderungsspezifikation Testfälle für den Test der Anwendung gegen die Anforderungsspezifikation abgeleitet und zum Teil generiert werden können, womit die entwicklungsbegleitende Qualitätssicherung als Hauptziel der Arbeit erreicht ist.

## Teil III

# Validierung und Verifikation der Anforderungsspezifikation

In diesem Teil befassen wir uns mit Qualitätssicherung bei der Anforderungsermittlung mit SCORES. In den Kapiteln 9 und 10 zeigen wir, wie Analytiker und Tester mit Benutzern und Domänen-Experten zunächst Use Cases bzw. Use Case Schrittgraphen und Teile des Klassenmodells validieren. Inhalt der darauf folgenden beiden Kapitel ist die rigorose Prüfung bzw. „Verifikation“ des Klassenmodells gegen die Use Case Schrittgraphen und — teilweise — umgekehrt. Die SCORES-Anforderungsspezifikation erlaubt es dabei, für die Validierung und die Verifikation Metriken bzw. Testkriterien sowohl für Testendekriterien als auch zur Ableitung entsprechender Testfälle anzugeben.

Im Test der Anwendung gegen die Anforderungsspezifikation zielen sogenannte „SCORES grey-box Testfälle“ auf die internen Interaktionen der Anwendung ab. Wir zeigen daher in Kapitel 13, wie die Anforderungsspezifikation und die bei ihrer Prüfung protokollierte Information über den Entwurf in die Implementation hin (nach-) verfolgt werden kann.

# Kapitel 9

## Validierung

*A bug prevented is cheaper than a bug discovered! [Beizer95]*

Die Anforderungsspezifikation ist sowohl bei ihrer Erstellung als auch später immer dann, wenn neue Anforderungen gefunden oder bereits spezifizierte Anforderungen geändert werden, zu validieren und zu verifizieren. Validierung und Verifikation der Anforderungsspezifikation sind also fortlaufende, ineinander greifende Tätigkeiten, die sowohl für Zwischenergebnisse als auch für die „endgültige“ Anforderungsspezifikation durchzuführen sind (vgl. [Boehm84][Cockburn97][Gerrard94]).

Zur präziseren Abgrenzung der Validierung und Verifikation ziehen wir wieder den IEEE Standard 610 zu Rate:

**Validation.** The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

**Verification.** (1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness. [IEEE610.90]

Boehm bringt den Unterschied zwischen Validierung und Verifikation in den zwei folgenden, oft zitierten Fragen auf den Punkt:

Informally, we might define these terms via the following questions:

Verification: "Am I building the product right?"

Validation: "Am I building the right product?" [Boehm84]

Wir kommen nun zu den Defekten, die in der Validierung (und Verifikation) mit Reviews aufgespürt werden sollen. Die häufigsten *Fehler in Anforderungsspezifikationen* lassen sich in drei Kategorien aufteilen (vgl. [Boehm84][Meyer85][Poston87][IEEE830.93][IEEE1233.96]):

- Die *Vollständigkeit* der Anforderungsspezifikation bedeutet, dass jeder Anforderung mindestens ein Element der Anforderungsspezifikation zugeordnet ist (*fachliche Vollständigkeit*). Daneben stellen z.B. auch ungelöste Referenzen und lückenhafte Nummerierungen Unvollständigkeiten dar (*formale Vollständigkeit*).
- Bezüglich der *Konsistenz* der Spezifikation in sich wird z.B. geprüft, ob zwei oder mehr Elemente der Spezifikation im Widerspruch zueinander stehen. Hierzu zählen auch Mehrdeutigkeiten (ein Spezifikationselement lässt mindestens zwei Interpretationen zu). Formale Vollständigkeit und Konsistenz sind notwendig für die *formale Korrektheit* der Anforderungsspezifikation.
- Unter dem Begriff der *fachlichen Korrektheit* werden bei der Anforderungsermittlung alle Unstimmigkeiten der Anforderungsspezifikation mit den eigentlichen Anforderungen bzw. der Domäne zusammengefasst, in der die Anwendung eingesetzt werden soll. Hierunter fallen z.B. verletzte Geschäftsregeln oder Gesetze, fehlerhafte Datendefinitionen oder aber falsch verstandene Benutzerwünsche. Zusätzlich enthält diese Kategorie Überspezifikationen (ein Element, das die Realisierung von Anforderungen beschreibt) und nicht testbare Anforderungen (Wunschdenken, „bells and whistles“, Anforderungen, deren Erfüllung entweder unmöglich oder aber nicht überprüfbar ist).

Die Validierung soll die fachliche Korrektheit und Vollständigkeit sicherstellen, die Verifikation die formale Korrektheit.

Die Anforderungsspezifikation ist der erste „Meilenstein“ in der Entwicklung (oder Erweiterung) einer Anwendung. Daher stehen Analytiker und Tester im Rahmen der Qualitätssicherung vor dem Problem, die Anforderungsspezifikation zumeist nicht z.B. unter Benutzung umfassender Checklisten gegen ein qualitätsgesichertes Ausgangsdokument validieren zu können, sondern nur gegen die oft eher vagen Vorstellungen und Wünsche der Benutzer. Mögliche Methoden zur Validierung sind somit das „Prototyping“ sowie Reviews als manuelle, statische Verfahren zur Prüfung von Entwicklungsprodukten bzw. ihren unterschiedlichen Repräsentationen (vgl. [Boehm84][FreWei90][ABR+94][PagSix94][Gerrard94][WBM96][Winter97]).

Wir betrachten zunächst kurz das Prototyping, das Hsia, Davis und Kung im Rahmen der Anforderungsermittlung folgendermaßen präzisieren:

In context of requirements engineering, prototyping is the construction of an executable system model to enhance the understanding of the problem and identify appropriate and feasible external behaviors for possible solutions. Prototyping is an effective tool to reduce risk on software projects by an early focus on feasibility analysis, identification of real requirements, and elimination of unnecessary requirements. [HDK93]

Im Weiteren orientieren wir uns an [KPS95]. Geht man von Anwendungen mit „geschichteten“ Architekturen aus (vgl. Abschnitt 2.2), so ist es sinnvoll, von horizontalen und vertikalen Prototypen zu sprechen.

- ❑ *Horizontale Prototypen* realisieren eine Architekturschicht des Systems vollständig. Die anderen Schichten werden nur skizzenhaft ausgeführt und soweit simuliert, dass der Prototyp lauffähig ist. Die bekanntesten horizontalen Prototypen sind Oberflächenprototypen, welche die Interaktionen zwischen Benutzer und Anwendung modellieren (vgl. [VosNen98]).
- ❑ *Vertikale Prototypen* fokussieren auf einige wenige Funktionalitäten, die durch alle Schichten hindurch realisiert werden. Hier geht es z.B. darum, unklare Funktionalitäten zu studieren und zu präzisieren oder die technische Machbarkeit zu überprüfen. Die übrigen Funktionalitäten werden wieder nur insoweit implementiert, dass die Anwendung lauffähig ist.

Während die Klassifizierung in horizontale und vertikale Prototypen den modellierten Bereich der Anwendung und den Realisierungsgrad als Beschreibungsmerkmale verwendet, geht es bei der Klassifikation in explorative, experimentelle und evolutionäre Prototypen darum, auf welche Weise der Prototyp entsteht, zu welchem Zweck er erstellt wird und was später mit ihm geschieht.

- ❑ *Exploratives Prototyping* dient dazu, die grundsätzlichen Probleme und die wichtigsten Anforderungen anhand einer lauffähigen Anwendung zu erkunden. Analytiker und Entwickler verschaffen sich Einblick in die Domäne und lernen die Terminologie der Benutzer und deren Wünsche kennen. Im Mittelpunkt stehen also die Klärung von Anforderungen sowie die Ausarbeitung von Lösungsvorschlägen.
- ❑ *Experimentelles Prototyping* sucht nicht nach Lösungsvorschlägen, sondern untersucht eine bereits gefundene Lösung auf ihre Angemessenheit und technische Machbarkeit. Ein solcher Prototyp kann z.B. dazu dienen, Spezifikationen zu ergänzen bzw. zu verfeinern oder den Weg zur Implementation vorzuzeichnen. Dabei kann man sich auf Aspekte wie z.B. das Layout der Benutzungsoberfläche, die Architektur, die Effizienz eines Algorithmus oder die Anbindung einer externen Anwendung konzentrieren.

Experimentelle Prototypen können in manchen Fällen in die Anwendung integriert werden, teilen jedoch meistens das Schicksal explorativer Prototypen, d.h. sie werden „weggeworfen“.

- ❑ *Evolutionäre Prototypen* werden mit der Intention erstellt, nicht weggeworfen zu werden, sondern in einem kontinuierlichen, zyklischen Prozess in die Anwendung zu konvergieren.

Evolutionäre Ansätze müssen in einen dementsprechend ausgerichteten Entwicklungsprozess integriert werden (vgl. [Davis90]). Beschränken wir uns auf die Validierung der Anforderungsspezifikation, so erscheinen von den prototyp-basierten Ansätzen aus Kostengründen nur explorative generierte (Wegwerf-)Prototypen sinnvoll (vgl. [Boehm84][Davis90]). Zur Generierung solcher Prototypen ist der Einsatz formaler Spezifikationen wie beispielsweise



State Charts oder anderweitig spezifizierter Zustandsautomaten (vgl. [Glinz95] [HarNaa96] [HsiKun97]), Petri-Netzen (vgl. [Barbey97][DBB97]), algebraischen bzw. modellbasierten Spezifikationssprachen wie (Object-) Z ([CarSto94]) oder spezieller Techniken zur Spezifikation von Benutzungsschnittstellen (s. [KSV96][Voss97]) notwendig.

Da diese Arbeit die Anforderungsspezifikation mit (verfeinerten) Use Cases und Klassenmodellen zum Inhalt hat und wir keine Anforderungen an den Grad der Formalisierung stellen wollen, betrachten wir als Validierungsmethode im Wesentlichen verschiedene Formen von Reviews<sup>1</sup>. Werden zur Anforderungsspezifikation auch formale Techniken oder spezielle Modelle verwendet, bieten sich generierte (Wegwerf-)Prototypen selbstverständlich zur Ergänzung und Illustration der Reviews an.

## 9.1 Reviewtechniken

In diesem Abschnitt skizzieren wir einige bekannte Reviewtechniken, wobei wir uns eng an [PagSix94] sowie diverse Artikel in [WBM96] anlehnen. Reviews leiten sich aus der „*Have a close look*“-Vorgehensweise ab, mit der normalerweise jeder Autor seine Dokumente überprüfen sollte. Aus der freiwilligen, unstrukturierten Betrachtung von Dokumenten bzw. Entwicklungsprodukten wird ein präzise definierter Vorgang, bei dem der zeitliche und inhaltliche Ablauf sowie die Anzahl und Qualifikationen der beteiligten Personen genau festgelegt sind und der nicht auf die Prüfung textueller Dokumente beschränkt bleibt.

Reviews sollen Defekte im Entwicklungsprodukt aufzeigen. In Abhängigkeit von der beteiligten Personengruppe, den Defekten, die das Review aufdecken soll, ihrer Kritikalität sowie der Kritikalität des zu prüfenden Produkts werden verschiedene Review-Techniken eingesetzt. Am weitesten verbreitet sind das persönliche Review, der Walk-Through und die Inspektion.

- ❑ *Persönliche Reviews* werden im Laufe der Entwicklung mit dem Ziel der Validierung und Verbesserung des Produkts durchgeführt. Möglichkeiten hierzu sind die „Ausführung von Hand“, bei welcher der Entwickler ein Programm (-stück) auf dem Papier durchspielt und dabei die Werte von Variablen etc. tabellarisch nachhält, oder das Prüfen von Dokumenten anhand persönlicher Check- und Fehlerlisten. Mit gutem Erfolg können die Mitarbeiter einer Projektgruppe ihre Produkte auch in gegenseitigen persönlichen Reviews prüfen.
- ❑ Als *Walk-Through* wird eine Zusammenkunft mehrerer Personen mit meistens unterschiedlichen Aufgaben und Kenntnissen bezeichnet. Der Entwickler stellt das Produkt vor und erklärt es anhand einiger durchzuspielender Szenarien, wofür einige nicht zu

---

<sup>1</sup> Wir benutzen „Review“ als Oberbegriff für (statische) manuelle Prüfungen (vgl. [PagSix94][Riedemann97]); die anglo-amerikanische Literatur verwendet hierfür eher den Begriff „Inspektion“ (vgl. [WBM96]).

komplizierte Testfälle vorbereitet werden. Ziel ist die Diskussion der Entwurfsentscheidungen und die Vermittlung von Entwurfswissen zum tieferen Verständnis des Produkts. Die Testfälle selbst spielen dabei keine kritische Rolle; sie dienen eher dazu, den Entwickler gezielt zu Entscheidungen (z.B. zur Ablauflogik) zu befragen. In den meisten Fällen werden mehr Defekte in der Diskussion mit dem Entwickler als durch die Testfälle selbst entdeckt.

- Eine *Inspektion* ist eine von mehreren Personen nach reglementierten Schritten durchgeführte, dokumentierte Prüfung eines Produkts gegen eine begrenzte Menge von Kriterien (s. [Fagan76]). Das Inspektionsteam besteht gewöhnlich aus vier bis acht Personen. Der Moderator, der ein erfahrener Softwareentwickler sein sollte, leitet hauptverantwortlich die Inspektionssitzung und protokolliert alle gefundenen Defekte. Außerdem muss er die anschließende Behebung aller während der Inspektion gefundenen Defekte sicherstellen. Die weiteren Teilnehmer sind der Autor des Dokuments, der Programmdesigner (wenn es sich um ein Codedokument handelt) und ein Testspezialist. Eine Inspektion läuft in den folgenden vier Etappen ab.

**Überblick.** Der Moderator verteilt das zu prüfende Dokument inkl. sämtlicher Vorgängerdokumente, die bindende und damit zu testende Vorgaben enthalten (z.B. die Produktskizze), an die Teilnehmer. (Für nicht mit dem Produkt vertraute Teilnehmer kann optional noch ein kurzer Informationsvortrag über das Produkt und das zu prüfende Dokument gegeben werden.)

**Vorbereitung.** Die Teilnehmer setzen sich intensiv mit dem Stoff auseinander.

**Sitzung.** Der Autor erläutert sein Dokument Schritt für Schritt. Alle in der Diskussion erkannten Defekte werden protokolliert, jedoch nicht korrigiert. Als Hilfsmittel dienen dabei z.B. Checklisten möglicher Defekte.

**Nachbereitung.** In dieser Phase werden die gefundenen Defekte behoben.

Die Sitzung sollte nicht länger als 120 Minuten dauern, da danach die Konzentrationsfähigkeit der Teilnehmer und damit die Produktivität zu stark nachlässt. In dieser Zeit können ca. 250-300 Anweisungen Quelltext inspiziert werden.

Nach Parnas kann der Inspektionserfolg dadurch verbessert werden, dass unterschiedlich qualifizierte Reviewer (z.B. Benutzer, Entwickler, Anwendungsspezialist...) Teile des Prüfgegenstands anhand verschiedener, defektspezifischer Techniken untersuchen (*active design reviews*, [ParWei85]). In mehreren kleineren Treffen der Entwickler mit jeweils einem Reviewer werden die aufgeworfenen Punkte weiter diskutiert und so ein umfassendes Protokoll erstellt.

Eine weitere Verbesserung durch die Unterteilung der Reviews in kleine, aufeinander aufbauende Schritte (*phases*) schlagen Knight und Myers mit *schrittweisen Inspektionen* (*phased inspections*) vor ([KniMye93]). In Einzelinspektorschritten wird das Produkt zuerst

von jeweils einem Reviewer gegen relativ einfache, aber wichtige anwendungsunabhängige Defekte (z.B. Spezifikations-Richtlinien) geprüft. Danach prüfen mehrere Reviewer in sog. Multi-Inspektorschritten das Produkt unabhängig voneinander gegen domänen- oder anwendungsspezifische Defekte (z.B. Geschäftsregeln, Robustheit, Benutzerfreundlichkeit). In einer jeden Multiinspektorphase abschließenden Sitzung vergleichen die Reviewer ihre Ergebnisse und erstellen ein gemeinsames Protokoll. Im Unterschied zu den anderen Review-Techniken, die jeweils einen Teil des Produkts gegen viele Kriterien prüfen, wird in jedem Schritt das gesamte Produkt gegen eine oder einige wenige Eigenschaften geprüft. Experimentell wurden nach der Anwendung solcher Verbesserungen Effizienzsteigerungen von bis zu 35% gemessen (vgl. [PVB95]).

Nach diesem Überblick wenden wir uns nun den Techniken und dem methodischen Vorgehen bei der Validierung der Anforderungsspezifikation zu. Durch das Konzept der Kontextschritte kann in SCORES die Anforderungsspezifikation von der kontextuellen Information bis zur Interaktionsinformation validiert werden (vgl. [KPR+97] [KPW98]). Neben Analytikern, Testern und Domänenexperten sind auch verschiedene Benutzergruppen, also die „Stakeholder“ der entsprechenden Geschäftsprozesse, beteiligt. Sie validieren in schrittweisen Inspektionen und Walk-Throughs die funktionale Zerlegung und das externe Verhalten der Anwendung, wobei das Hauptaugenmerk auf der fachlichen Vollständigkeit und Korrektheit der Anforderungsspezifikation in Bezug auf den Anwendungsbereich liegt. Die Zusammenarbeit aller betroffenen Personengruppen ist eine notwendige Voraussetzung für die erfolgreiche Validierung und sichert darüber hinaus zu, dass die textuellen Teile der qualitätsgesicherten Anforderungsspezifikation in einer von allen Parteien akzeptierten und verstandenen Sprache formuliert sind.

## 9.2 Funktionale Zerlegung

Die Zerlegung der funktionalen und organisatorischen Aspekte der Geschäftsprozesse spiegelt sich in Aktoren sowie den Aufgaben von Use Cases und Use Case Schritten wider. Aus der SCORES Anforderungsspezifikation werden die Aktor-Hierarchie und (anhand des Makroschritt-Konzepts) eine statische Aufgabenhierarchie generiert. Zusätzlich können wir aus den textuellen Beschreibungen der Use Cases und Use Case Schritte auch jeweils auf einen Aktor zugeschnittene Dokumente generieren, die von Benutzern aus den entsprechenden Gruppen bzw. von Domänenexperten mit schrittweisen Inspektionen geprüft werden.

- Personalverantwortliche und Domänenexperten inspizieren die Aktor-Spezifikationen gegen die Organisationssicht der entsprechenden Geschäftsprozesse. Inspektionskriterien sind z.B. die vollständige Abbildung des von der Anwendung betroffenen Teils der Organisationsstruktur, die richtige Zuteilung von Aktoren zu Benutzergruppen und die vollständige Zuteilung von Aufgaben zu Aktoren.

- ❑ Domänenexperten prüfen die statische Aufgaben- bzw. Use Case-Hierarchie gegen die funktionalen Aspekte des Geschäftsprozesses, wobei mögliche Inspektionskriterien z.B. die vollständige Abbildung des Prozesses in den Aufgaben oder die Vollständigkeit der Use Case-Beschreibungen bezogen auf den Geschäftsbereich sind.
- ❑ Einzelne Benutzer prüfen sozusagen als „Instanzen von Akteuren“ die Beschreibung der Use Cases und Use Case Schritte, an denen die entsprechenden Akteure beteiligt sind. Inspektionskriterien sind z.B. die Granularität der einzelnen Use Case Schritte, die richtige Aufteilung in Kontext- und Interaktionsschritte und die Vollständigkeit und Korrektheit der Beschreibung eines Use Case Schritts bezogen auf die entsprechende Geschäftsregel.

Abb. 9.1 zeigt einige exemplarische Checklistenpunkte für die Validierungs-Inspektionen. Umfassendere und konkretere Checklisten sind abhängig von der Anwendungsdomäne und z.B. den personellen und fachlichen Gegebenheiten zu erstellen. Nach den Inspektionen ist die fachliche Vollständigkeit der Anforderungsspezifikation in Bezug auf die Funktionalität und die fachliche Korrektheit der Aufgabenbeschreibungen gegen die Organisatorische Sicht und die Leistungssicht des Geschäftsprozesses geprüft.

#### **Fachliche Vollständigkeit**

- Sind alle Einträge der Produktskizze in Use Cases erfasst?
- Sind alle Geschäftsvorfälle in den Use Cases erfasst?
- Gibt es Lücken in den Use Case Schrittgraphen, also fehlende Schritte oder Kanten?
- Sind für alle Use Case Schritte und alle Kanten Vor- und Nachbedingungen definiert?
- Sind alle betroffenen Klassen im Bereich eines Schritts enthalten?

#### **Fachliche Korrektheit**

- Erfassen die Vorbedingungen der Use Cases erlaubte Geschäftssituationen?
- Beachten die Spezifikationen der Use Cases die bekannten Geschäftsregeln?
- Erfassen die Nachbedingungen der Use Cases erlaubte Geschäftssituationen?
- Ist das erwartete Ergebnis eines Szenarios korrekt in den Nachbedingungen der im Szenario enthaltenen Schritte (insbesondere im Endschrift) wiedergegeben?

#### **Prüfbarkeit**

- Ist die Aufgabe eines Use Cases bzw. Use Case Schritts vage beschrieben?
- Sind die Vor- und Nachbedingungen und die Bedingungen der Kanten prüfbar?
- Ist jeder Use Case zu seinem Ursprung bzw. den Stakeholdern vor-verfolgbar?

#### **Änderbarkeit**

- Ist jeder nicht von einem Makroschritt referenzierte Use Case ohne Einfluss auf die übrigen Use Cases änderbar?

*Abb. 9.1 Exemplarische Checklistenpunkte für die Validierung*

## 9.3 Externes Verhalten

Im zweiten Schwerpunkt der Validierung werden die Use Case Schrittgraphen gegen operationale und kausale Aspekte sowie die Ablaufaspekte des Geschäftsprozesses geprüft. Erfahrungsgemäß führt bei „dynamischen“ bzw. ablaufbezogenen Spezifikationen eine Reihe von Walk-Throughs zu den besten Ergebnissen. Hierbei werden bei der Validierung von Anforderungsspezifikationen konkrete Geschäftsvorfälle als angemessenes Medium zur Validierung des extern zu beobachtenden bzw. erwünschten Verhaltens verwendet (*business scenarios*, vgl. [Boehm84][Gerrard94][PTA94][WPJ+98]). SCORES-Anforderungsspezifikationen sind hierbei besonders „testbar“, da sich konkrete Geschäftsvorfälle bzw. Szenarien unmittelbar als Pfade durch einen Use Case Schrittgraphen wiederfinden ([KPR+97]).

In SCORES validieren Benutzer, Domänenexperten und Analytiker somit die Use Case Schrittgraphen, indem sie in Walk-Throughs anhand einer angenommenen, konkreten (Geschäfts-) Situation und Aufgabenstellung die entsprechenden Pfade im Use Case Schrittgraph durchlaufen. Häufig entdeckte Fehler sind hierbei fehlende, falsche oder mehrdeutige Schritte, Geschäftsregeln verletzende Vor- und Nachbedingungen der Schritte bzw. Übergangsbedingungen der Kanten, und die unvollständige Behandlung von Ausnahmesituationen.

Zur Vorbereitung eines Walk-Throughs leiten Analytiker und Tester aus dem Klassenmodell anhand der Vorbedingung des Use Cases und seinem Klassenbereich eine initiale Objektkonstellation ab, welche die angenommene Geschäftssituation widerspiegelt (s. auch Abschnitt 17.1). Zu Beginn des Walk-Throughs prüfen Benutzer und Domänenexperten, ob die zum Start des Geschäftsvorfalles notwendige Information bzw. Situation gegeben ist. Hierzu vergleichen Analytiker gemeinsam mit den Benutzern die jeweilige Objektkonstellation mit realen Konstellationen in der Problemwelt.

Zur Unterstützung der Benutzer — die im Allg. mit Klassenmodellen bzw. Objektkonstellationen wenig vertraut sind — können Metaphern, in Multimediassequenzen festgehaltene „Realwelt-Szenen“ oder einfache domänenspezifische Piktogramme verwendet werden (vgl. [Züllighoven98] [MCP+98] [Hasselbring98]). Zur weiteren Erhöhung der Anschaulichkeit lassen sich einzelnen Szenarioschritten auch manuell erstellte oder mit einem GUI-Werkzeug skizzierte Layouts von Fenstern oder Bildschirmmasken zuordnen. Wir erhalten auf diese Weise Näherungen an „nicht-operationale“ Prototypen der Anwendung (auch „mock-up“ genannt). Diese Elemente werden dann in „animierten“ Walk-Throughs zusammen mit den aufbereiteten Objektkonstellationen dargestellt.

Die Beteiligten simulieren bei einem Walk-Through den Use Case Schritt für Schritt unter Berücksichtigung des jeweiligen (Anwendungs-) Zustands, d.h. der jeweiligen Objektkonstellation. Hierbei konzentrieren sich die Benutzer und Domänenexperten auf die textuellen Spezifikationen der Schritte bzw. der Übergangsbedingungen und sichern ihre Korrektheit zu. Am Ende der Simulation (d.h., wenn kein weiterer Schritt mehr notwendig bzw. möglich ist) wird geprüft, ob ein Endschrift im Use Case Schrittgraph erreicht ist und der sich erge-

bende Ablauf und die resultierende Objektkonstellation mit dem erwarteten Geschäftsvorfall verglichen. Falsche, fehlende und mehrdeutige Schritte werden zusammen mit dem Pfad und Kommentaren protokolliert, korrigiert und erneut validiert. Geschäftsvorfälle werden so zu Testfällen der Anforderungsspezifikation.

Da hierbei auch die Verantwortlichkeiten der beteiligten Klassen geprüft werden, sind somit (implizit) auch zumindest Teile des Klassenmodells validiert. Hauptaugenmerk richten die Analytiker auf fehlende, unvollständige oder überflüssige Klassen und Beziehungen, Mehrdeutigkeiten und die Adäquatheit des vorgeschlagenen Klassenmodells hinsichtlich der Use Cases.

Die bei der Validierung Schritt für Schritt durchlaufenen Szenarien werden von den Analytikern bzw. Testern protokolliert. Wir nennen solche protokollierten Szenarien *Test-Szenarien*. Hiermit wird deren Wiederverwendung bei der Verifikation und im Test sowie auch das erneute Durchspielen nach Änderungen an der Anforderungsspezifikation ermöglicht. Wir benötigen zur Werkzeugunterstützung wieder ein etwas formaleres Gerüst.

**Definition 9.1** Ein *Test-Szenarioschritt*  $ts$  beschreibt die Ausführung eines Use Case Schritts  $ucs$  eines Use Case Schrittgraphen  $uc$  und ist charakterisiert durch

- ❑ eine erläuternde *Beschreibung*,
- ❑ einen Verweis  $ts.ucs \in uc.steps$  auf den ihm zugeordneten *Use Case Schritt*  $ucs$ ,
- ❑ im Fall, dass der zugeordnete Use Case Schritt  $ucs$  ein Makroschritt ist, optional einen Verweis auf ein Test-Szenario des referenzierten Use Cases.

❑

Entsprechend der bisherigen Definition des Begriffs „Szenario“ als Pfad durch den Use Case Schrittgraphen ist ein Test-Szenario eine Folge von Test-Szenarioschritten, welche die konkrete „Ausführung“ eines einzelnen, bestimmten Use Cases beschreiben. Präziser:

**Definition 9.2** Sei  $uc$  ein Use Case Schrittgraph. Ein *Test-Szenario* zum Use Case Schrittgraphen  $uc$  ist ein 4-Tupel  $TS = \{ST, s_0, s_e, \sigma\}$  mit:

1.  $ST$  ist eine nichtleere Menge von Test-Szenarioschritten;
2.  $s_0 \in ST$  ist der *Startschritt* des Test-Szenarios;
3.  $s_e \in S$  ist der *Endschritt* des Test-Szenarios;
4.  $\sigma: ST \setminus s_e \rightarrow ST$  ist die *Folgeschrittfunktion*, die jedem Test-Szenarioschritt mit Ausnahme des Endschriffs genau einen *Folgeschritt* zuordnet.

❑

Bei einem Makroschritt stehen wir vor der Wahl, die Test-Szenarien auf den referenzierten Use Case Schrittgraphen auszudehnen oder den Schritt im Kontext eines Test-Szenarios als „atomare“ Aufgabe zu betrachten. Um die Übersichtlichkeit und Machbarkeit des Ansatzes

zu wahren, bevorzugen wir die zweite Möglichkeit und validieren einen Makroschritt über seine Vor- und Nachbedingung sowie die des referenzierten Use Case „als Ganzes“.

Natürlich muss jedes Test-Szenario auch tatsächlich ein Szenario zu dem Use Case sein, also einem Pfad durch den Use Case Schrittgraphen entsprechen, welcher mit dem Startschritt beginnt und in einem Endschritt endet. Wir formulieren sogleich drei dementsprechende (Validierungs-) Regeln.

**Regel 9.3.1 [Test-Szenario-Startschritt]** Der Startschritt  $s_0$  eines Test-Szenarios  $ts$  zum Use Case Schrittgraph  $uc$  muss dem Startschritt des Use Case Schrittgraphen zugeordnet sein:  $ucs(s_0(ts)) = s_0(uc)$ .

**Regel 9.3.2 [Test-Szenario-Folgeschritt]** Die Folgeschrittfunktion eines Test-Szenarios  $ts$  zum Use Case Schrittgraph  $uc$  muss verträglich zur Folgeschrittfunktion des Use Case Schrittgraphen sein:  $\forall st \in ST(ts) \setminus s_e: ucs(\sigma(st)) \in \sigma(ucs(st))$ .

**Regel 9.3.3 [Test-Szenario-Endschritt]** Der Endschritt  $s_e$  eines Test-Szenarios  $ts$  zum Use Case Schrittgraph  $uc$  muss einem Endschritt des Use Case Schrittgraphen zugeordnet sein:  $ucs(s_e(ts)) \in SE(uc)$ .

Wir werden Test-Szenarien während der Verifikation um weitere, systeminterne Information anreichern. In Teil IV zeigen wir dann, wie solche Test-Szenarien für den dynamischen Test der Anwendung gegen die Anforderungsspezifikation wiederverwendet werden.

## 9.4 Scores-Metamodell (III)

In diesem Abschnitt erweitern wir die in den Abbildungen Abb. 5.7 und Abb. 7.2 dargestellten partiellen Sichten des SCORES Metamodells um Primitive zur Modellierung von Test-Szenarien und erhalten das in Abb. 9.1 gezeigte UML-Klassendiagramm. Die Klasse **TestScenario** ist Unterklasse von **ValidationClassifier** und „erbt“ somit ihren Klassenbereich. Die Folgeschrittfunktion wird durch die Beziehung *successor* modelliert, deren Multiplizitäten die möglichen Strukturen der Szenarien auf lineare Listen einschränken.

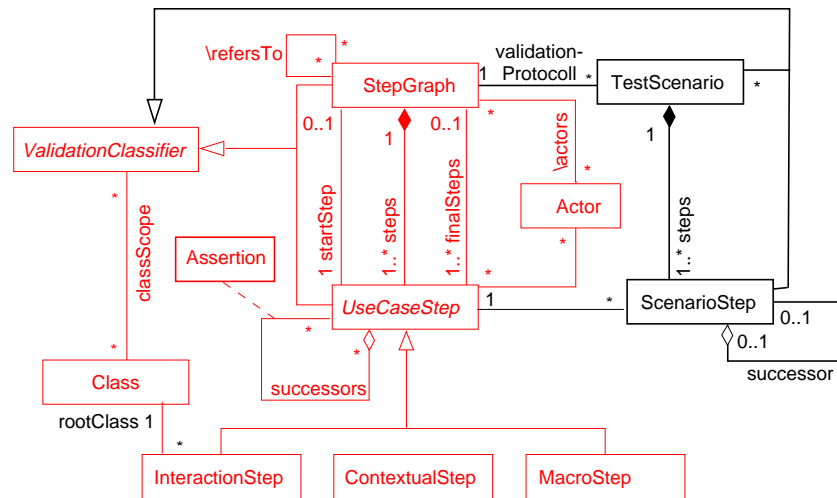


Abb. 9.1 SCORES Metamodell mit Test-Szenario



# Kapitel 10

## Validierungsmetriken

*Question: What do you do when you see a graph?*

*Answer: Cover it! [Beizer95]*

Validierung und Verifikation sowie das Testen von Programmen weisen eine Reihe ähnlicher Probleme auf. Unter diesen Problemen ist eines der schwierigsten die Frage nach dem Ende der Prüfung. Einerseits soll die Prüfung eine solide Basis für die weitere Entwicklung oder den Einsatz des Produkts sicherstellen, andererseits können Prüfungen — zumindest im Sinne von stichprobenartigen „Tests“ — niemals die Korrektheit garantieren, sondern nur das „Vertrauen“ in das Produkt erhöhen (vgl. [Dijkstra76]). Somit ist eine möglichst objektive Kosten/Nutzen Abwägung wünschenswert. Dementsprechend wurden z.B. für das Testen von Programmen viele Testkriterien angegeben, welche — zusammengefasst zu Testendekriterien — den Tester bei der Ermittlung des richtigen Zeitpunkts zum Abbruch der Prüfungen unterstützen (vgl. [Myers79][Beizer90][Riedemann97]). Auch die Validierungs- und Verifikationstätigkeiten bei der Anforderungsermittlung werden bei der Anwendbarkeit solcher quantitativen Maße besser planbar und kontrollierbar.

Granularität und Semantik der SCORES Anforderungsspezifikation erlauben es, bekannte Testkriterien für die Prüfung von Programmen auf die Qualitätssicherung bei der Anforderungsermittlung zu übertragen. Da die Validierung (sowie Teile der Verifikation) auf dem Konzept des Use Case Schrittgraphen aufbauen, können wir z.B. auf bekannte „Überdeckungskriterien“ des kontrollflussbasierten sowie des funktionalen Testens von Programmen zurückgreifen.

Wie bei den qualitätssichernden Tätigkeiten in der Anforderungsermittlung unterteilen wir auch die Metriken in Validierungs- und Verifikationsmetriken. Validierungsmetriken messen in erster Linie die Überdeckung der funktionalen Aspekte der Anforderungsspezifikation. Sie geben also Antwort auf die Frage, wie rigoros die Use Case (Schrittgraphen) validiert wurden. Verifikationsmetriken beurteilen anhand funktionaler und struktureller Aspekte die Ve-

rifikation und geben damit Auskunft, wie tief das Zusammenspiel von Use Case Schrittgraphen und Klassenmodell geprüft ist. Wie bei den Modellen für den dynamischen Test von Anwendungen (vgl. [Beizer90][Riedemann97]) verwenden wir die Metriken sowohl zur Ableitung von Testfällen als auch in Form von Validierungs- bzw. Verifikations-Endekriterien.

Wir beginnen mit einfachen, auf dem Use Case Schrittgraphen aufbauenden strukturellen Metriken (Use Case-Schritt- und Kantenüberdeckung). Als komplexere strukturelle Metriken definieren wir dann die Use Case-Szenario-, -Grenze-Inneres und -Pfadüberdeckung. Mit der Use Case-Schrittüberdeckung konzentrieren wir uns auf die Aufgaben und somit die funktionale Zerlegung. Die anderen Überdeckungen zielen auf die Übergangsbedingungen und somit auf die „Ablauflogik“ der Use Case Schrittgraphen. Wir definieren die Metriken wieder mit OCL-Operationen (zur OCL s. auch Abschnitt 7.5). In den Ausdrücken verwenden wir die SCORES-Primitive und verweisen diesbezüglich auf das Metamodell (Abb. 9.1.). Im Folgenden sei  $uc$  ein Use Case (Schrittgraph) und  $ts$  ein Test-Szenario von  $uc$ .

## 10.1 Use Case Schrittüberdeckung

Die *Use Case Schritt-Überdeckung*  $c_0^{uc}$  entspricht der Anweisungsüberdeckung  $c_0$  des white-box Tests (vgl. [Riedemann97]). Die Operation  $uc.validationProtocol$  ermittelt zunächst die Menge aller für den Use Case protokollierten Test-Szenarien. Für ein bestimmtes Test-Szenario ermittelt die OCL-Operation  $visitedUseCaseSteps$  die vom Test-Szenario überdeckten Schritte des Use Cases:

$$\begin{aligned} &ts.visitedUseCaseSteps \\ &\equiv \{ucs \in uc.steps \mid \exists ts.s \in ts.steps : ts.s.UseCaseStep = ucs\} \end{aligned} \quad (10.1)$$

Für  $uc$  ist dann die Use Case Schritt-Überdeckung

$$c_0^{uc} \equiv \frac{|\bigcup_{ts \in uc.validationProtocol} ts.visitedUseCaseSteps|}{uc.steps \rightarrow size} \quad (10.2)$$

Es ist  $c_0^{uc} = 1$ , wenn jeder Schritt im Use Case Schrittgraph durch mindestens einen Schritt in einem Test-Szenario überdeckt ist.

*Beispiel 10.1.* In dem in Abb. 10.1 (a) skizzierten Use Case Schrittgraph sind die von Szenarien überdeckten Schritte fett hervorgehoben. Nach (10.2) wurde eine Use Case Schritt-Überdeckung von  $c_0^{uc} = \frac{4}{5} = 80\%$  erreicht.

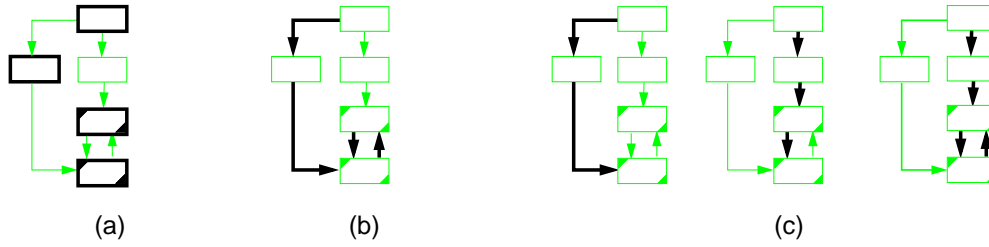


Abb. 10.1 Beispiele zur Use Case Schritt- (a), -Kanten (b) und Szenarioüberdeckung (c)

□

## 10.2 Use Case Kantenüberdeckung

Als zweite, mächtigere Metrik stellen wir die *Use Case Kanten-Überdeckung*  $c_1^{uc}$  vor, die das Pendant zur Zweigüberdeckung  $c_1$  eines Kontrollflussgraphen ist (vgl. [Riedemann97]). Wir definieren die zwei Hilfsfunktionen **StepGraph::allEdges** zur Ermittlung der Kantenmenge eines Use Case Schrittgraphen  $uc$  (10.3) und **TestScenario::visitedUseCaseEdges**

$$uc.allEdges \equiv \{(ucs_1, ucs_2) \in uc.steps \times uc.steps \mid (ucs_2 \in ucs_1.successors)\} \quad (10.3)$$

zur Ermittlung der von einem Test-Szenario  $ts$  überdeckten Kanten des Use Case Schrittgra-

$$ts.visitedUseCaseEdges \equiv \{uce \in uc.allEdges \mid \exists tss_1, tss_2 \in ts.steps: \\ (tss_2 = tss_1.successor) \wedge (tss_1.UseCaseStep, tss_2.UseCaseStep) = uce\} \quad (10.4)$$

phen (10.4). Für einen Use Case  $uc$  ergibt sich damit die Kanten-Überdeckung zu

$$c_1^{uc} \equiv \frac{|\bigcup_{ts \in uc.validationProtocol} ts.visitedUseCaseEdges|}{uc.allEdges \rightarrow size} \quad (10.5)$$

Es ist  $c_1^{uc} = 1$ , wenn jede Kante im Use Case Schrittgraph in mindestens einem Test-Szenario durchlaufen ist.

*Beispiel 10.2.* In dem in Abb. 10.1 (b) skizzierten Schrittgraph sind die von Test-Szenarien überdeckten Kanten fett hervorgehoben. Nach (10.5) ist eine Use Case Kanten-Überdeckung von  $c_1^{uc} = \frac{4}{6} = 66, \overline{66}\%$  erreicht.

□

## 10.3 Use Case Szenario-Überdeckung

Die Use Case Kantenüberdeckung ist nicht ausreichend, da einerseits bei vollständiger Kantenüberdeckung nicht unbedingt alle Endschr tte des Use Case Schrittgraphen auch schon als Endschr tt in einem Test-Szenario vorkommen. Andererseits haben wir Zyklen im Schrittgraph, die zur wiederholten Bearbeitung von Aufgaben f hren k nnen, bisher  berhaupt noch nicht betrachtet. Als n chstes definieren wir daher die *Use Case Szenario- berdeckung*  $cSz^{uc}$ .

Zur Beschreibung von Pfaden im Use Case Schrittgraph ben tigen wir zun chst einige weitere Begriffe und Hilfsfunktionen. Sei  $n \in \mathbb{N} \setminus \{0\}$ . Zun chst ermittle die Operation **StepGraph::allPaths** (integer):SetOfSequenceOfUseCaseStep (10.6) alle Pfade der L nge  $n$  (d.h. mit  $n$  Kanten)

$$uc.allPaths(n) \equiv \{(ucs_0, \dots, ucs_n) \in uc.steps^{n+1} \mid \forall 0 \leq i < n: ucs_{i+1} \in \sigma(ucs_i)\} \quad (10.6)$$

und die Operation **StepGraph::allPaths**():SetOfSequenceOfUseCaseStep (10.7) alle Pfade in einem Use Case Schrittgraph  $uc$ .

$$uc.allPaths \equiv \bigcup_{1 \leq i \leq \infty} uc.allPaths(i) \quad (10.7)$$

Wir nennen einen Pfad *einfach*, wenn kein Schritt im Pfad mehrfach vorkommt, d.h.  $ucs_i \neq ucs_j$  f r  $0 \leq i < j < n$ . Nat rlich sind wir bei der Validierung nicht an beliebigen Pfaden im Use Case Schrittgraph interessiert, sondern an *Szenarien*, also Pfaden, die mit dem Startschritt des Schrittgraphen beginnen und in einem der Endschr tte abgeschlossen sind. Operation **allScenarios** (10.8) ermittelt genau die Menge der in einem Use Case Schrittgraph m glichen Szenarien.

$$\begin{aligned} uc.allScenarios \\ \equiv \{(ucs_0, ucs_1, \dots, ucs_n) \in uc.allPaths \mid ucs_0 = uc.startStep \wedge ucs_n \in uc.finalSteps\} \end{aligned} \quad (10.8)$$

Sei  $ts$  ein Test-Szenario zum Use Case  $uc$  und  $(tss_0, tss_1, \dots, tss_n)$  die Folge der Schritte von  $ts$ . Es gelte also  $tss_0 = ts.steps.first$  und  $tss_{i+1} = \sigma(tss_i)$  f r  $0 \leq i < n$ . Wir ermitteln zun chst den ein-eindeutig dem Test-Szenario entsprechenden Pfad im Use Case Schrittgraph und definieren hierf r Operation **ts.path** (10.9).

$$ts.path \equiv (ucs_0, ucs_1, \dots, ucs_n) \in uc.allPaths \mid \forall i \in \mathbb{N}^n: ucs_i = tss_i.UseCaseStep \quad (10.9)$$

Zus tzlich ermittelt die Operation **ts.isComplete** (10.10), ob der dem Test-Szenario entsprechende Pfad auch tats chlich ein Szenario im Use Case Schrittgraph darstellt, also mit dem Startschritt beginnt und in einem Endschr tt endet.

$$\begin{aligned} ts.isComplete &\equiv tss_o.UseCaseStep = ts.UseCase.startStep \wedge \\ tss_n.UseCaseStep &\in uc.finalSteps \end{aligned} \quad (10.10)$$

Die Use Case Szenario-Überdeckung  $c_{Sz}^{uc}$  ermittelt nun den Prozentsatz aller von einem Test-Szenario überdeckten<sup>1</sup> möglichen Szenarien im Use Case Schrittgraph zu

$$c_{Sz}^{uc} \equiv \frac{|\{ts \in uc.ValidationProtocol \mid ts.isComplete\}|}{uc.allScenarios \rightarrow size} \quad (10.11)$$

Es ist  $c_{Sz}^{uc} = 1$ , wenn jedes mögliche Szenario des Use Case Schrittgraphen, also jeder Pfad, der mit dem Startschritt beginnt und in einem Endschrift endet, auch als Test-Szenario protokolliert ist.

*Beispiel 10.3.* In den drei in Abb. 10.1 (c) gezeigten Skizzen eines Schrittgraphen sind die von Test-Szenarien überdeckten Kanten fett hervorgehoben. Nach (10.5) ist eine Use Case Szenario-Überdeckung von  $c_1^{uc} = \frac{4}{\infty} = 0\%$  erreicht, da der Zyklus aus den beiden unteren (End-)Schritten prinzipiell unendlich viele Szenarien ermöglicht. Wären z.B. aufgrund einer Geschäftsregel in jedem Szenario maximal 3 Ausführungen des oberen Endschrifts erlaubt, ergäbe sich eine Use Case Szenario-Überdeckung von  $c_1^{uc} = \frac{3}{13} = 23,08\%$ , da nun über den linken Zweig 7 und über den rechten Zweig 6 Pfade durch den Schrittgraphen möglich sind.

□

## 10.4 Grenze-Inneres- und Pfadüberdeckung

Die Use Case Szenarioüberdeckung ist nur für azyklische Use Case Schrittgraphen vollständig zu erfüllen und muss in der Praxis geeignet eingeschränkt werden. Als praktisch relevantere strukturelle Metrik definieren wir daher im Folgenden die *Use Case Grenze-Inneres-Überdeckung*  $c_{GI}^{uc}$ . Diese Metrik konzentriert sich gerade auf die Zyklen im Use Case Schrittgraph und entspricht der (schwachen)  $c_{GI}$ -Überdeckung des kontrollflussbezogenen Testens (vgl. [Riedemann97]).

Wir benötigen für die Definition der Grenze-Inneres-Überdeckung einige weitere Begriffe und Hilfsfunktionen zur Beschreibung und Verknüpfung von Pfaden und insbesondere von Zyklen im Use Case Schrittgraph, die wir der eigentlichen Definition (10.18) voranstellen. Ein *Zyklus* im Use Case Schrittgraph ist ein Pfad, der an seinem Startschritt endet, für den also  $ucs_0 = ucs_n$  gilt. Die Menge aller Zyklen in einem Use Case Schrittgraph  $uc$  berechnet Operation  $uc.allCycles$ .

$$uc.allCycles \equiv \{(ucs_0, ucs_1, \dots, ucs_n) \in uc.allPaths \mid ucs_0 = ucs_n\} \quad (10.12)$$

<sup>1</sup> Vereinfachend sagen wir, dass ein Test-Szenario einen Pfad im Use Case Schrittgraph überdeckt, wenn der Pfad Teilpfad des dem Test-Szenario entsprechenden Szenarios ist.

Für zwei Pfade  $v = (v_0, v_1, \dots, v_m)$  und  $w = (w_0, w_1, \dots, w_n)$  mit  $0 < m \leq n$  des Use Case Schrittgraphen  $uc$  definieren wir den „Teilpfad“-Operator  $\subseteq: uc.allPaths \times uc.allPaths \rightarrow \text{bool}$  zu

$$v \subseteq w \equiv \begin{cases} \text{true, falls } \exists 0 \leq i \leq n-m: (w_i, \dots, w_{i+m}) = v \\ \text{false, sonst} \end{cases} \quad (10.13)$$

d.h. es ist genau dann  $v \subseteq w = \text{true}$ , wenn  $v$  vollständig in  $w$  enthalten ist. Das Zeichen  $\perp$  beude den *leeren Pfad*, es gelte  $\perp \subseteq v$  für alle Pfade  $v \in uc.allPaths$ . Für den Spezialfall, dass für den End- bzw. Startschritt der Pfade  $v$  und  $w$  gilt  $v_n = w_0$ , definieren wir den „Verkettungs-Operator“  $.;: uc.allPaths \times uc.allPaths \rightarrow uc.allPaths$  zu

$$v, w \equiv \begin{cases} (v_0, \dots, v_n, w_1, \dots, w_m), \text{ falls } (v_n = w_0) \\ \perp, \text{ sonst} \end{cases} \quad (10.14)$$

Für  $i \in \mathbb{N}$  bezeichnen wir mit  $v^i$  die  $i$ -fache Verkettung eines Pfades  $v$  mit sich selbst, es ist also  $v^i = v, \dots, (i\text{-mal}) \dots, v$  und  $v^0 = \perp$ . Zusätzlich seien wie üblich  $v^* = \bigcup_{i \in \mathbb{N}} v^i$  die Menge aller aus beliebig vielen Verkettungen von  $v$  mit sich selbst bestehenden Pfade (einschließlich des leeren Pfades) und  $v^+ = v^* \setminus \{\perp\}$ .

Hiermit können wir nun mit der Operation  $uc.allBoundaryCycles$  alle Zyklen im Use Case Schrittgraph ermitteln, die genau einmal von einem Test-Szenario „überdeckt“ werden.

$$\begin{aligned} uc.allBoundaryCycles &\equiv \{c \in uc.allCycles \mid \exists ts \in uc.ValidationProtocol: \\ &c \subseteq ts.path \wedge \forall p \in c^+ \setminus \{c\}: \neg(p \subseteq ts.path)\} \end{aligned} \quad (10.15)$$

Ebenso ermitteln wir mit der Operation  $uc.allInteriorCycles$  alle Zyklen im Use Case Schrittgraph, die zwei- oder mehrmals von einem Test-Szenario überdeckt werden.

$$uc.allInteriorCycles \equiv \{c \in uc.allCycles \mid \exists ts \in uc.ValidationProtocol: \exists p \in c^+: p \subseteq ts.path\} \quad (10.16)$$

Zusätzlich ermitteln wir noch alle Zyklen im Use Case Schrittgraph, die von mindestens einem Test-Szenario nicht überdeckt, also „umgangen“ werden (10.17).

$$uc.allExtPaths \equiv \{c \in uc.allCycles \mid \exists ts \in uc.ValidationProtocol: \neg \exists p \in c^+: p \subseteq ts.path\} \quad (10.17)$$

Nach diesen Vorüberlegungen sind wir in der Lage, für einen Use Case (Schrittgraph)  $uc$  die *Use Case Grenze-Inneres-Überdeckung* zu definieren:

$$c_{GI}^{uc} \equiv \frac{|uc.allBoundaryCycles \cap uc.allInteriorCycles \cap uc.allExtPaths|}{uc.allCycles \rightarrow \text{size}} \quad (10.18)$$

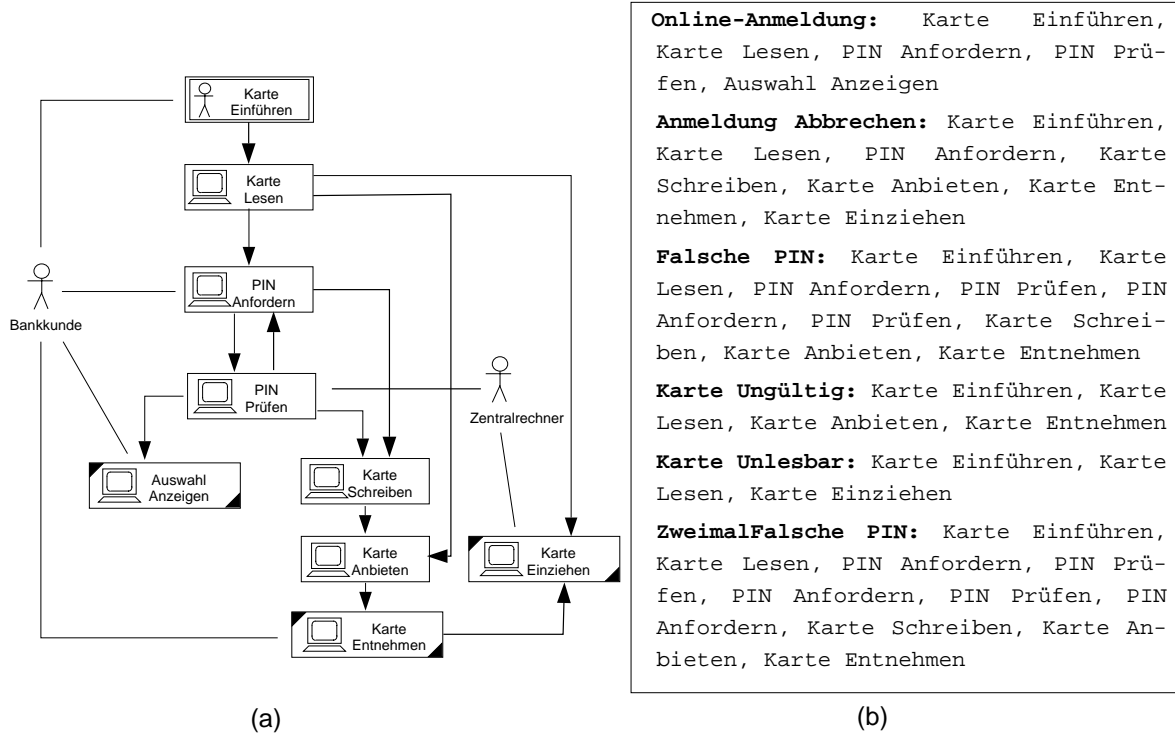


Abb. 10.2 Use Case Schrittgraph (a) und fünf textuell beschriebene Szenarien (b)

Wir erhalten  $c_{GI}^{uc} = 1$ , wenn für jeden Zyklus im Use Case Schrittgraph jeweils mindestens ein Test-Szenario existiert, das den Zyklus einmal, genau einmal und mindestens zweimal durchläuft.

Letztendlich können wir als mächtigste (wiederum nur für azyklische Use Case Schrittgraphen vollständig zu erfüllende) Validierungsmetrik die *Use Case Pfad-Überdeckung*  $c_{\infty}^{uc}$  definieren:

$$c_{\infty}^{uc} \equiv \frac{\left| \bigcup_{ts \in uc.ValidationProtocol} ts.path \right|}{uc.allPaths \rightarrow size} \quad (10.19)$$

**Beispiel 10.4.** Wir illustrieren die strukturellen Metriken dieses Abschnitts an dem in Abb. 10.2 (a) gezeigten Schrittgraph des Use Cases Anmelden. Abb. 10.2 (b) skizziert hierzu textuell einige Szenarien. Die Use Case-Schrittüberdeckung erreichen wir bereits mit den beiden Szenarien Online-Anmeldung (Abb. 10.3 bzw. Abb. 10.4 (a)) und Anmeldung Abbrechen (Abb. 10.4 (b)). Die fehlenden Kanten vom Schritt PIN Prüfen zurück zum Schritt PIN Anfordern und vom Schritt PIN Anfordern zurück zum Schritt Karte Schreiben decken wir mit dem Szenario Falsche PIN ab (Abb. 10.4 (c)). Dementsprechend deckt das Szenario Karte Ungültig die Kante vom Schritt Karte Lesen zum Schritt Karte Schreiben ab. Zur 100%-igen Erfüllung der Use Case-Kantenüberdeckung durchläuft das Szenario Karte Unlesbar noch die Kante vom Schritt Karte Lesen zum Schritt Karte Einziehen. Zur 100%-igen Grenze-Inneres Überdeckung fehlt noch eine Wiederholung des durch die beiden Kanten zwischen den Schritten

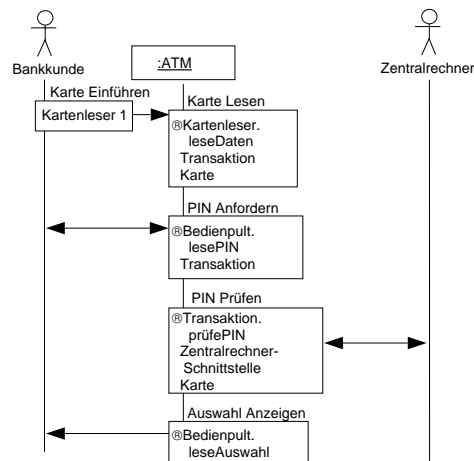


Abb. 10.3 Sequenzdiagramm zum Test-Szenario Online-Anmeldung, Use Case Anmelden

PIN Anfordern und PIN Prüfen gebildeten Zyklus. Dies erreichen wir durch zweimalige Fehleingabe der PIN, wie es im Szenario ZweimalFalsche PIN gezeigt ist. Schon für diesen Use Case Schrittgraphen sind die vollständige Szenarioüberdeckung oder gar die Pfadüberdeckung aufgrund des Zyklus theoretisch unerreichbar<sup>1</sup>. Abb. 10.5 zeigt eine mit Piktogrammen animierte Sicht des Test-Szenarios Auszahlung OK im Use Case Geld Abheben (s. auch Abb. 5.8 (b)). Die Interaktionen verlaufen zeitlich gesehen von oben nach unten. Für Szenarien, in denen viele Aktoren involviert sind, ist neben dieser Darstellung auch eine an das UML-Kollaborationsdiagramm angelehnte Sicht möglich, bei der z.B. eine Nummerierung die zeitliche Reihenfolge der Interaktionen verdeutlicht. Abb. 10.6 zeigt ein mit Skizzen der Benutzungsoberfläche animiertes Test-Szenario zum Use Case Anmelden.

□

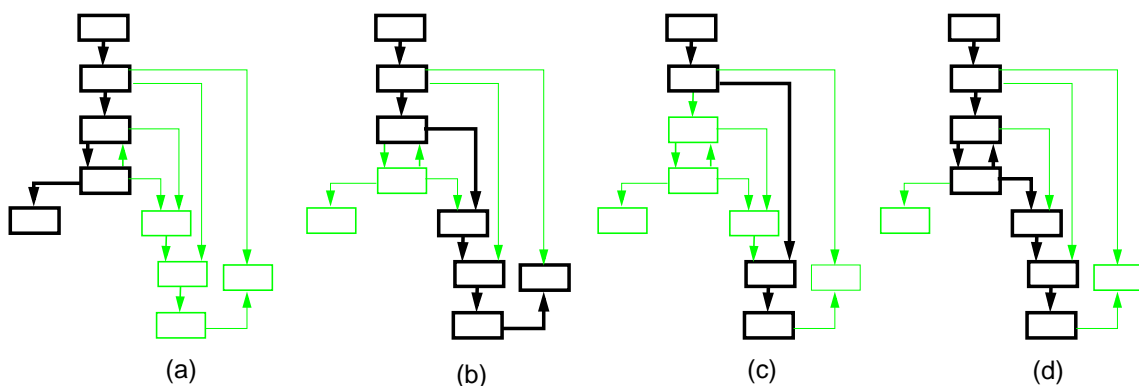


Abb. 10.4 Überdeckung des Use Case Schrittgraphen Anmelden

<sup>1</sup> Praktisch ist die vollständige Szenarioüberdeckung erreichbar, da bei dreimalig innerhalb von 24 Stunden aufeinanderfolgender Fehleingabe der PIN die Karte gesperrt wird. Der Grenzfall „viermalige Fehleingabe der PIN“ in mehr als 24 Stunden muss natürlich trotzdem getestet werden!



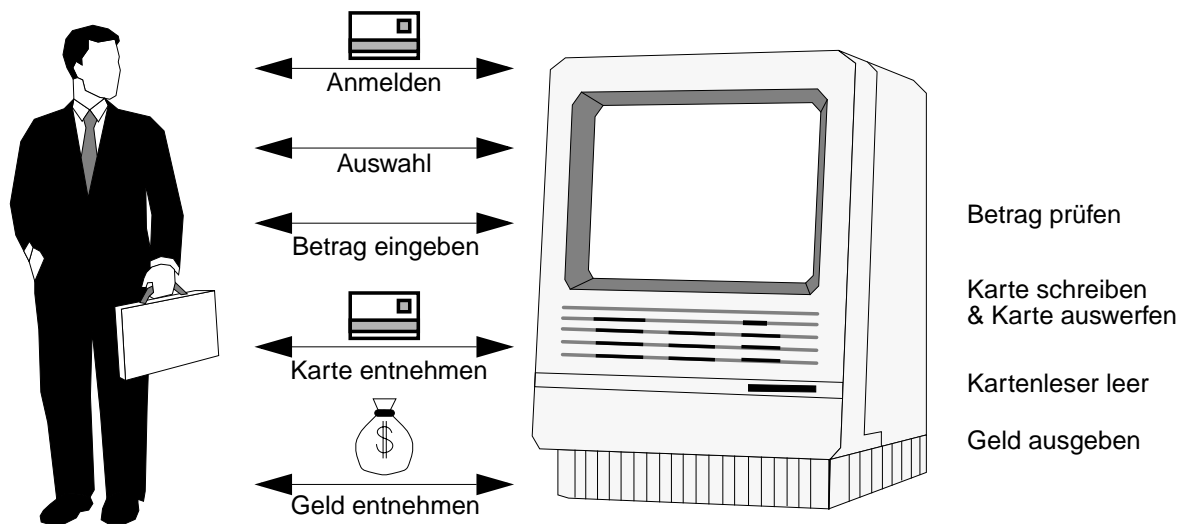


Abb. 10.5 Benutzerfreundliche Sicht für das ATM Test-Szenario Auszahlung OK

Zum Ende dieses Kapitels beleuchten wir noch die Möglichkeiten und Grenzen bzw. das Ziel der Validierung in Abhängigkeit vom angestrebten bzw. zur Ableitung der Testfälle verwendeten Testkriterium bzw. von der verwendeten Metrik. Test-Szenarien, die auf die Use Case Schritt-Überdeckung  $c_0^{uc}$  hin gerichtet sind, können nur Fehler in den einzelnen Schritten selbst, nicht aber in allen möglichen Abläufen entdecken. Diese Tests zielen also auf die Hauptabläufe und lassen alternative Abläufe und Ausnahmebehandlungen oft unberücksichtigt. Insofern handelt es sich um positive Prüfungen zum Nachweis der Funktionalität.

Test-Szenarien im Hinblick auf die Use Case Kanten-Überdeckung  $c_1^{uc}$  durchlaufen alle möglichen Verzweigungen im Use Case Schrittgraph mindestens einmal. Sie testen also die Übergangsbedingungen der Kanten und damit auch auf die möglichen Alternativen und Ausnahmen im Use Case, so dass diese Tests auf die Robustheit der Anwendung abzielen. Bertolino und Marré geben ein Verfahren zur Generierung von Testfällen für Kontrollflussgraphen an ([BerMar93]). Natürlich setzen solche Verfahren die automatische Auswertbarkeit der Bedingungen voraus.

Auch die Use Case Grenze-Inneres-Überdeckung  $c_{GI}^{uc}$  zielt auf iterativ durchzuführende (Teil-) Aufgaben sowie bestimmte Ausnahmen im Use Case. Insbesondere das Umgehen von Zyklen im Use Case Schrittgraph bedeutet oft, dass z.B. „keine Daten zu bearbeiten bzw. vorhanden“ sind. Daneben werden jedoch auch die Hauptabläufe bei sich wiederholenden Teilaufgaben geprüft. Die zwei- oder mehrmalige Ausführung der Zyklen ist als erster Schritt in Richtung des Lasttests zu sehen, der dann zur (teilweisen) Use Case Szenario-Überdeckung bzw. sogar zur Use Case Pfad-Überdeckung  $c_{\infty}^{uc}$  führt. Da hierbei jedoch auch Pfade im Use Case berücksichtigt werden, die keinen Szenarien und insofern keinen vollständigen Abläufen in der Realwelt entsprechen, ist die Use Case Pfad-Überdeckung nur theoretisch, sozusagen in ihrer Abschlusseigenschaft als obere Schranke, interessant.

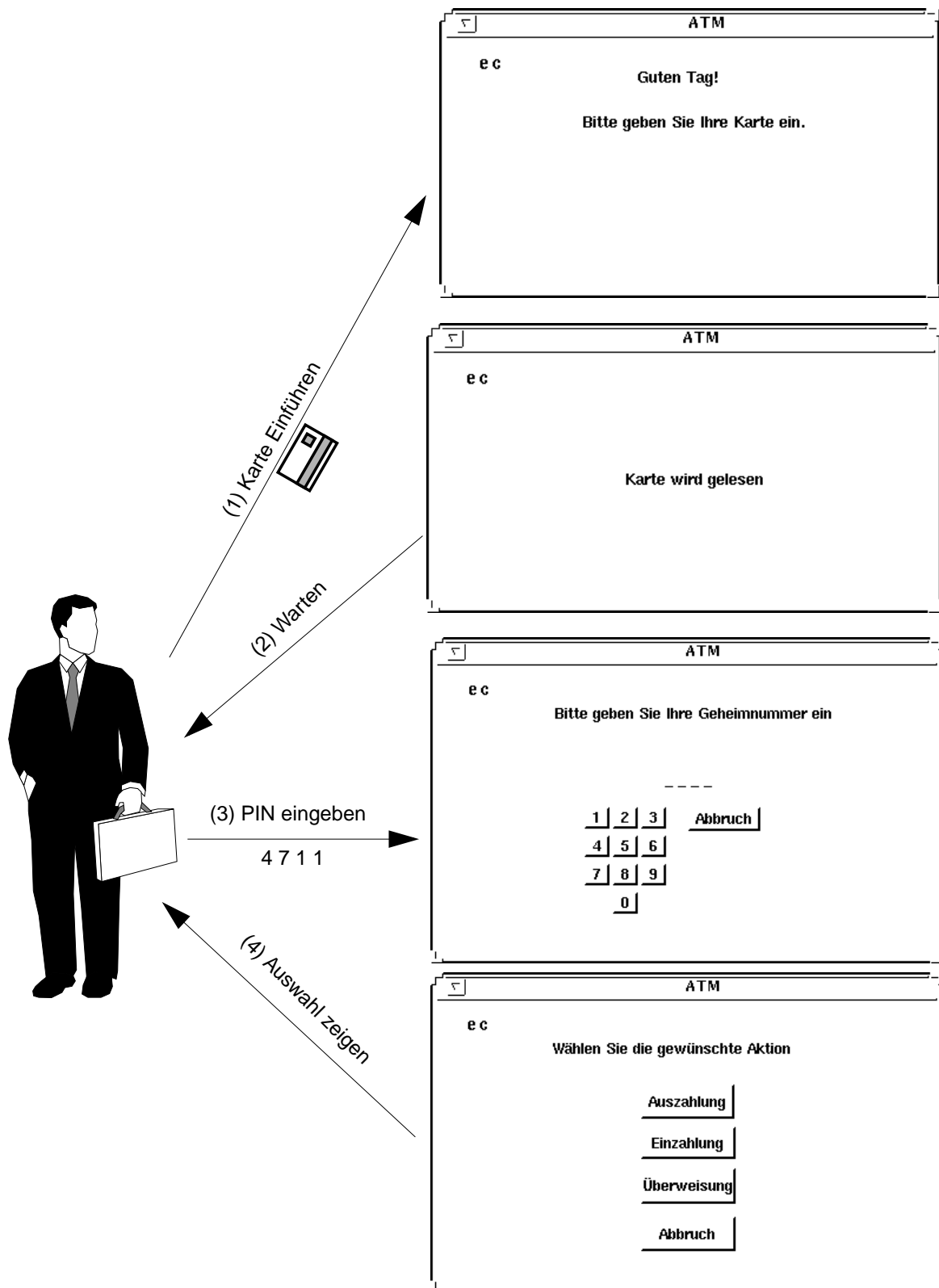


Abb. 10.6 Bildschirmskizzen für das ATM Test-Szenario Anmeldung OK

# Kapitel 11

## Verifikation

Validierung und Verifikation der Anforderungsspezifikation sind ähnliche Tätigkeiten, wobei letztere auf einer erheblich detaillierteren und formaleren Ebene arbeitet. Die Verifikation zielt auf die Konsistenz und (formale) Korrektheit der Anforderungsspezifikation ab. Aus diesem Grund können Benutzer (und auch viele Domänen-Experten) im Allgemeinen nicht bei der Verifikation mitwirken und Analytiker und Tester verbleiben als die einzig ausführenden Parteien. Diese verifizieren den strukturellen Aufbau sowohl der Use Case Schrittgraphen als auch des Klassenmodells sowie dessen „internes Verhalten“ und damit die Frage, wie die funktionalen Anforderungen im Domänen-Klassenmodell abgebildet werden.

### 11.1 Struktureller Aufbau

Die Verifikation in SCORES beinhaltet zunächst eine Serie separater schrittweiser Inspektionen der Use Case Schrittgraphen und des Klassenmodells. Ziel der Inspektionen ist die Aufdeckung inkonsistenter und/oder mehrdeutiger sowie unvollständiger Spezifikationen.

Analytiker konzentrieren sich bei der Verifikation von Use Case Schrittgraphen auf die korrekte Umsetzung der textuell angegebenen Vor- und Nachbedingungen in prädikatenlogische bzw. OCL-Ausdrücke. So muss z.B. die Bearbeitung der Aufgabe eines Use Case Schritts  $s$  für alle Objektkonstellationen und ggf. Eingaben, die laut der Vorbedingung von  $s$  und dem Klassenmodell erlaubt sind (s. auch Kapitel 17), ein Ergebnis erbringen, welches der Nachbedingung von  $s$  und dem Klassenmodell genügt. Um die Bedingungen der von  $s$  ausgehenden Kanten im Use Case Schrittgraph zu verifizieren, suchen die Analytiker für jede Kante zumindest eine laut Vorbedingung von  $s$  erlaubte Objektkonstellation, für welche nach der Ausführung von  $s$  die Bedingung der betrachteten Kante erfüllt ist. Zusätzlich wird im Falle eines Interaktionsschritts der Klassenbereich und die Spezifikation der Wurzeloperation geprüft. Auch „globale“ Eigenschaften wie z.B. die Erreichbarkeit der einzelnen Schritte und die Terminierung der Abläufe sind Gegenstand der Verifikation und bei ausreichender For-

malisierung der Bedingungen teilweise automatischen Prüfungen zugänglich (vgl. [DesObe96][HOR98]).

Im Zentrum der Inspektionen des strukturellen Aufbaus des Klassenmodells stehen die Generalisierungen und die Assoziations-, Aggregations- und Kompositionsbeziehungen (vgl. [MMB+96][Siegel96][Winter97]). Einige Kriterien z.B. für die Vererbungshierarchie sind:

- ☐ Ist dokumentiert, ob es sich um Vererbung im Sinne einer Generalisierung/Spezialisierung oder um Vererbung für Wiederverwendung handelt?
- ☐ Ist die Vererbungsbeziehung konzeptuell notwendig oder lässt sich der erwünschte Effekt auch durch eine Benutzungsbeziehung bzw. mittels Aggregation erreichen?
- ☐ Sind Blätter der Vererbungshierarchie abstrakte Klassen? Wenn ja, warum? Können Sie eliminiert werden?
- ☐ Erben abstrakte Klassen von konkreten Klassen? Wenn ja, warum? Können Sie eliminiert werden?
- ☐ Gibt es Klassen, die keine Attribute und/oder Dienste neu definieren bzw. redefinieren? Wenn ja, warum? Können Sie eliminiert werden?

Zusätzlich bei mehrfacher Vererbung:

- ☐ Ist der gewünschte Effekt einer wiederholten Vererbung klar dokumentiert?
- ☐ Realisiert die Klasse mehrere Konzepte bzw. Aufgaben? Wenn ja, warum?

Anhand der Assoziationen teilen wir das Klassenmodell in mehrere Gruppen gekoppelter Klassen (Cluster) auf. Grob gesagt erkennen wir solche Cluster daran, dass zwischen Klassen aus unterschiedlichen Clustern keine Aggregationen und Kompositionen existieren. Wir inspizieren z.B. gegen folgende Kriterien:

- ☐ Sind die erkennbaren Cluster explizit als solche dokumentiert?
- ☐ Sind die dokumentierten Cluster entkoppelt, d.h. bestehen zwischen Klassen aus verschiedenen Clustern keine Aggregationen und Kompositionen?
- ☐ Existieren zwischen Klassen aus verschiedenen Clustern wenige, klar dokumentierte Assoziationen, welche die notwendigen Navigationen zum „Botschaftsaustausch“ ermöglichen?

Nach den Inspektionen sind sowohl das Use Case Modell als auch das Klassenmodell konsistent und eindeutig, also formal korrekt. In der Validierung wurde das Use Case Modell hinsichtlich seiner fachlichen Korrektheit und Vollständigkeit untersucht, so dass es nun vollständig geprüft ist. Da während der Validierung nur Teile des Klassenmodells geprüft wurden, können wir seine Vollständigkeit und fachliche Korrektheit noch nicht zusichern. Es verbleibt also zu verifizieren, ob das Klassenmodell bezüglich der (validierten und verifizierten) Use Cases vollständig und korrekt ist.

Hierzu prüfen wir, ob das Klassenmodell die Aufgaben im Use Case Modell adäquat widerspiegelt, wobei der durch die Klassenbereiche, Wurzelklassen und der Wurzeloperationen hergestellte nahtlose und nach-verfolgbare Übergang von den Use Case Schritten zu Operationen und Klassen im Klassenmodell eine ausschlaggebende Rolle spielt.

## 11.2 Externes vs. internes Verhalten

In der Validierung werden Test-Szenarien in Form von Geschäftsvorfällen als Pfade im Use Case Schritgraph oder — genauer gesagt — als „konkrete, Abläufe“ geprüft. Analytiker und Tester untersuchen nun, ob und wie die validierten „extern beobachtbaren“ Auswirkungen der Szenarien aufgrund des Zustands von Anwendungsobjekten bzw. der Anwendung selbst zustandekommen können. In anderen Worten: Sie prüfen, ob das strukturelle Anforderungsmodell, also das Klassenmodell, die funktionalen Aspekte der Anforderungen, also die Use Cases, auch wirklich reflektiert bzw. „ausführen“ kann. Ausgehend von den validierten und in Inspektionen verifizierten Use Case Schritgraphen werden dafür die Elemente des Klassenmodells im Einzelnen untersucht.

Mit der Angabe des Klassenbereichs, der Wurzelklasse und der Wurzeloperation für jeden Use Case Interaktionsschritt ist zunächst — in Verbindung mit den Vor- und Nachbedingungen des Schritts — festgehalten, welche Operation welcher Domänenklasse im Klassendiagramm für die betrachtete (Teil-) Aufgabe verantwortlich ist. Was diese Operation allerdings in Abhängigkeit von ihrem Ausführungskontext genau bewirkt und ob bzw. wie sie sich auf weiteren, in den Domänenklassen definierten Operationen abstützt, ist in unseren bisherigen, zumindest bezüglich des Klassenmodells eher „statischen“ Betrachtungen noch nicht ausgedrückt. Zusätzlich ist bis jetzt unklar, wie sich der „globale Zustand“ der Anwendung manifestiert und wie er sich während der Bearbeitung eines Use Cases ändert.

Ohne dieses Wissen müssten wir jedoch zur präzisen Spezifikation der Wurzeloperation die Kapselung der einzelnen Klassen durchbrechen und in der Vor- und Nachbedingung von Wurzeloperationen Informationen aufnehmen, die nicht im Verantwortungsbereich der jeweiligen Wurzelklasse liegen. Behringer nennt diesen Sachverhalt in ihrer Dissertation als Hauptnachteil der auf mehreren, getrennten Modellen basierenden „Matrix-Ansätze“<sup>1</sup>:

Though encapsulation is one of the key concepts of object-orientation, [in matrix-approaches] there is no encapsulation of data and functions into objects. Data and functions are not modelled on the same level of granularity. The data is modelled on the level of individual objects, the functions on the level of the whole system. The scenarios, system operations or use cases are orthogo-

---

<sup>1</sup> Unter dem Namen „Matrix-Ansatz“ fasst Behringer alle objektorientierten Methoden wie z.B. OOSE ([JCJ+92]) und OMT ([RBP+91]) zusammen, welche die funktionale Sicht und die Datensicht getrennt modellieren und in denen diese Sichten nur durch eine — implizite oder explizite — Abhängigkeitsmatrix zwischen Funktionen und Daten aufeinander abgebildet werden.

nal to objects. They are purely functional abstractions and could just as well be used as input for structured design. Instead of a system of interacting objects, the analysis model specifies a matrix over data entities and global functions. There is no modularisation. Other concepts such as specialisation and aggregation are used for the object model but not for the scenario model. [Behringer97]

Während Behringer diese Problematik mit einem „Ein-Modell-Ansatz“ angeht, welcher das „globale Verhalten“ der Anwendung mit um Kontrollflusskonstrukte<sup>1</sup> angereicherten Interaktionsdiagrammen beschreibt, spezifizieren wir in SCORES die Anforderungen mit mehreren Modellen. Maßgeblich für diese Entscheidung sind die angestrebte Ausgewogenheit und „Objektorientierung“ der Anforderungsspezifikation, da rein verhaltensorientierte Modelle erfahrungsgemäß zu einer funktionalen Zerlegung und damit oft zu instabilen Strukturen führen (vgl. [PagSix94]). Auch Firesmith schreibt in Bezug auf rein funktionale, z.B. nur auf Use Cases basierende Modellierungsansätze:

A major functional abstraction can cause the numerous problems with functional decomposition that object technology was to avoid [...] Any decomposition based on use cases scatters the features of the objects and classes among the individual use cases [...] This scattering of objects to use cases leads to the Humpty Dumpty effect, in which [it is] unlikely to put the objects and classes back together without a massive expenditure of time and effort. [Firesmith98]

In SCORES spezifizieren wir dementsprechend die externe Funktionalität bzw. das extern beobachtbare Verhalten der Anwendung mit Use Case Schrittgraphen. Dies entspricht einerseits dem oft verkündeten Grundsatz, Geschäfts-Regeln und -Prozesse nicht in den Objekten zu „vergraben“ (vgl. [JCJ+92][Maring96]), und erhält andererseits über die Kopplung der Use Case Schrittgraphen zum Klassenmodell die notwendigen „objektorientierten“ Anteile.

Wir haben im letzten Kapitel mit Test-Szenarien konkrete, extern beobachtbare Abläufe bei der Bearbeitung der Aufgabe eines Use Cases simuliert bzw. beschrieben (Interaktions- und kontextuelle Information). Die oft nicht unmittelbar zu beobachtende interne Funktionalität spiegelt sich in den Operationen und den internen Interaktionen zwischen Instanzen der Domänenklassen des Klassenmodells wider (systeminterne Information). Natürlich können wir den internen Ablauf einzelner Operationen der Domänenklassen nicht vollständig bzw. allgemein angeben, ohne die Operationen durch Angabe des *Wie* z.B. in Form von (Pseudo-) Code überzuspezifizieren. Wir konzentrieren uns daher auf einzelne, konkrete Abläufe von Operationen im Kontext eines Test-Szenarios. Dabei reichern wir sozusagen die Schritte des Test-Szenarios um systeminterne Information an (s. auch Abb. 4.5).

---

1 Behringer's um Kontrollflusskonstrukte angereicherten Szenariendiagramme münden in einer Notation ähnlich der von Regnell ([RAB96]) verwendeten Erweiterung der MSC-Notation aus [ITU93].

## 11.3 Episoden

Bei dem Durchspielen eines Test-Szenarios liegt das Hauptaugenmerk auf der systeminternen Information, also den aufgrund der Ausführung eines Schritts ausgelösten Operationen im Klassenmodell. Innerhalb von Test-Szenarien spielen Analytiker und Tester für jeden Szenarioschritt, der einem Interaktionsschritt im Use Case zugeordnet ist, die Ausführung der Wurzeloperation des Interaktionsschritts durch. Da die zuerst aufgerufene Operation offensichtlich die Wurzeloperation des Interaktionsschritts ist, dient sie quasi als „Einstiegspunkt“ in das Klassenmodell.

In den meisten Fällen senden Objekte während der Ausführung von Operationen Botschaften an weitere Objekte (oder sich selbst), welche dann wiederum die Ausführung anderer Operationen bewirken. Dies entspricht einem (Aufruf-) Baum von Operationen im Klassenmodell, den wir als *Episode* bezeichnen. Episoden protokollieren somit systeminterne Information, die später einerseits mit entsprechenden Verifikationsmetriken ausgewertet wird und andererseits im Test der Anwendung gegen die Anforderungsspezifikation die Grundlage der SCORES grey-box Testfälle darstellt (s. Kapitel 16).

Episoden spiegeln die Ausführung der dem Interaktionsschritt zugeordneten Wurzeloperation in der angenommenen Objektkonstellation wider. Im Einzelfall entscheiden Analytiker und Tester, ob sie die Episode bis hin zu elementaren Attributzugriffen oder aber nur bis hin zu Aufrufen von wohlverstandenen Operationen modellieren bzw. verfolgen und protokollieren. Auch hier benötigen wir für die Werkzeugunterstützung wieder etwas Formalismus.

Zur Vereinfachung des Klassenmodells bzw. der Simulation von Episoden vereinbaren wir zunächst für jede Domänenklasse zusätzlich zu ihren explizit spezifizierten fachlichen „Domänen-Operationen“ implizit einige „Standard-Operationen“ (vgl. [CoeYou91][PagSix94][Poetzsch97]):

- ☐ new erzeugt eine neue Instanz der Klasse.
- ☐ destroy zerstört eine Instanz der Klasse.
- ☐ read symbolisiert das Lesen von Attributen einer Instanz der Klasse.
- ☐ modify symbolisiert das Setzen bzw. Ändern von Attributen einer Instanz der Klasse.
- ☐ Für jede eine Klasse betreffende Beziehung x (Assoziation, Aggregation und Komposition) bietet die Klasse die Standardoperationen
  - x-connect zum Erstellen,
  - x-disconnect zum Lösen der Beziehung,
  - x-count zur Abfrage der Anzahl aktuell referenzierter Objekte sowie
  - x-query zum Zugriff auf einzelne und x-iterate zum Zugriff auf alle bezogenen Objekte an.

Wir beginnen bei den folgenden Definitionen mit der Simulation der Ausführung einer einzelnen, in einer Klasse im Klassenmodell definierten Operation. Hierbei bezeichne  $\mathbf{K}$  wieder die Menge der Klassen im Klassenmodell (vgl. auch Abschnitt 7.1).

**Definition 11.1** Ein *Episodenschritt*  $es$  beschreibt die Ausführung einer einzelnen Operation und ist charakterisiert durch

- einen Verweis  $es.AK \in \mathbf{K}$  auf die *Klasse* des Objekts, welches den Schritt  $es$  „ausführt“;
- einen Verweis  $es.WO \in es.AK.allOperations$  auf die vom Schritt  $es$  „ausgeführte“ *Operation* im Klassenmodell und
- den *Sichtbarkeitsbereich* (*scope*)  $es.SK \subseteq \mathbf{K}$ .

□

Durch die Angabe der „ausführenden“ Klasse sind auch polymorphe bzw. dynamisch gebundene Aufrufe unverändert geerbter bzw. redefinierter Operationen berücksichtigt. Wir beschreiben den Sichtbarkeitsbereich weiter unten.

Die Benutzung bzw. der „Aufruf“ weiterer Operationen bei der Simulation einer Operationsausführung wird in der nachfolgenden Definition mit sog. Folgeschritten modelliert.

**Definition 11.2** Sei  $tsc$  ein Test-Szenarioschritt mit  $SA(ucs(tsc)) = \text{Interaktion}$  ( $tsc$  verweist also auf einen Interaktionsschritt im Use Case Schrittgraph). Die *Episode* zu  $tsc$  ist ein 3-Tupel  $ep = \{ES, s_0, \sigma\}$  mit:

1.  $ES$  ist eine nichtleere Menge von Episodenschritten;
2.  $s_0 \in ES$  ist der *Startschritt* der Episode;
3.  $\sigma: ES \rightarrow 2^{ES \setminus s_0}$  ist eine injektive Abbildung, die jedem Episodenschritt eine Menge von *Folgeschritten* („Operationsaufrufen“) zuordnet.

□

Eine Episode  $ep$  zu einem Test-Szenarioschritt  $tsc$  simuliert den Ablauf der Wurzeloperation des Interaktionsschritts, auf den  $tsc$  verweist, in einem bestimmten Kontext;  $ep$  modelliert also einen „Aufrufbaum“ oder „Trace“. Daher darf jeder Episodenschritt beliebig viele Folgeschritte, jedoch höchstens einen „Vorgängerschritt“ haben, was durch die Injektivität von  $ep.\sigma$  gesichert ist. Da die Episode insgesamt zusammenhängend sein muss, erhalten wir die folgende Regel ( $E$  bezeichne die Menge aller Episoden).

**Regel 11.3.1 [Episode-Folgeschritte]** Mit Ausnahme des Startschritts einer Episode ist jeder Episodenschritt Folgeschritt von mindestens einem anderen Episodenschritt, d.h.:

$$\forall ep \in E: \forall s \in ES(ep) \setminus s_0: |\{s_v \mid s \in ep.\sigma(s_v)\}| \geq 1.$$



Die Injektivität von  $\sigma$  und die in Definition 11.2 enthaltenen Bedingung, dass der Startschritt  $s_0$  der Episode nicht Folgeschritt irgend eines Schritts ist, sichern zu, dass  $\sigma$  zyklensfrei ist und somit jeder Schritt mit Ausnahme von  $s_0$  Folgeschritt von genau einem anderen Episodenschritt ist. Nehmen wir an, es gäbe einen Zyklus. Der Startschritt kann lt. Definition 11.2 nicht Element des Zyklus sein. Also müsste mindestens ein Schritt der Episode mehr als einen Vorgängerschritt haben, nämlich sowohl den Vorgängerschritt auf dem Pfad zum Startschritt als auch den Vorgängerschritt im Zyklus. Dies widerspricht der Injektivität von  $\sigma$ .

In Anlehnung an den Sichtbarkeitsbegriff für Programmiersprachen (vgl. [Poetzsch91]) kommen wir nun zum *Sichtbarkeitsbereich* eines Episodenschritts. Vereinfacht gesagt verstehen wir hierunter die Klassen (bzw. ihre Operationen, Attribute und Assoziationen), welche der vom Schritt „ausgeführten“ Operation zu einem bestimmten Betrachtungszeitpunkt bekannt sind. Die Veränderung bzw. Erweiterung des Sichtbarkeitsbereichs im Verlauf der simulierten Operationsausführung skizzieren wir zunächst informell:

- Der Sichtbarkeitsbereich eines Episodenschritts ohne Folgeschritte (d.h. unmittelbar nach dem „Aufruf“ der Operation) wird als *initialer Sichtbarkeitsbereich* bezeichnet. Er besteht aus der Klasse des „ausführenden“ Objekts selbst und den Klassen ihrer Attribute. Hinzu kommen die Klassen aller Parameter der dem Episodenschritt zugeordneten Operation.
- Hat der Episodenschritt Folgeschritte (d.h. im Verlauf der Operationsausführung wurden bereits weitere Operationen aufgerufen), so erweitert sich der Sichtbarkeitsbereich um alle Klassen der Ausgabeparameter der den Folgeschritten zugeordneten Operationen (d.h. der benutzten oder „aufgerufenen“ Operationen).
- Im Falle des „Aufrufs“ einer der beiden Standardoperationen x-query (Zugriff auf einzelne Objekte) und x-iterate (Zugriff auf alle bezogenen Objekte) erweitert sich der Sichtbarkeitsbereich um die Klasse, zu der die Assoziation x besteht.

Der initiale Sichtbarkeitsbereich geht statisch aus dem Klassendiagramm hervor. Bei den jeweiligen Operationsaufrufen werden dann die Klassen (und ihre Unterklassen) nachgehalten, um die sich der Sichtbarkeitsbereich erweitert. Wir präzisieren dies sogleich in der folgenden Definition.

**Definition 11.3** Seien ES die Schritte einer Episode  $e$  (zum Szenarioschritt  $tsc$ ) und  $es \in ES$  ein Episodenschritt. Wir definieren den *Sichtbarkeitsbereich* von  $es$  mit Hilfe der folgenden Abbildung  $SK: ES \rightarrow 2^K$ , wobei wir  $es.SK$  für  $SK(es)$  schreiben:

1.  $es.AK \in es.SK$  (Klasse des „ausführenden“ Objekts).
2.  $\{x \in K \mid y \in es.AK.allAttributes: x \leq y.class\} \subseteq es.SK$  (Klassen der Attribute)
3. Ist  $es.WO$  die  $es$  zugeordnete Operation mit  $es.WO.signature = (k_0, \dots, k_n)$ , dann ist  $\{x \in K \mid \exists y \in \{(k_0, \dots, k_{n-1})\}: x \leq y\} \subseteq es.SK$  (Klassen der Parameter).

4. Ist  $s_f \in \sigma(es)$  und die dem Folgeschritt  $s_f$  zugeordnete Operation  $s_f.WO$  eine der Standardoperationen  $x\text{-query}$  und  $x\text{-iterate}$ , und verbindet die Beziehung  $x$  die diese Operation definierende Klasse mit der Klasse  $y$ , so ist  $\{x \in K \mid x \leq y\} \subseteq es.SK$ .
5. Ist  $s_f \in \sigma(es)$ , und gilt für die Signatur der  $s_f$  zugeordneten Operation  $s_f.WO.signature = (k_0, \dots, k_n)$ , dann ist  $\{x \in K \mid x \leq k_n\} \subseteq es.SK$  (Klasse des Rückabewerts der aufgerufenen Operation).
6. Keine andere Klasse aus  $K$  gehört zu  $es.SK$ .

□

Zu beachten ist, dass die Episode über die von den Schritten „ausgeführten“ Operationen zwar Manipulationen auf Instanz- bzw. Objektebene beschreibt, sich selber jedoch ausschließlich auf Elemente des Klassenmodells bezieht. Die bei der „Ausführung“ der Operation in der Episode „sichtbaren“ Objekte und Beziehungen werden also auf entsprechende Elemente im Klassenmodell zurückgeführt.

## 11.4 Simulationsregeln

Analog zu den Modellierungsregeln geben wir auch zur „sinnvollen Simulation“ von Episoden einige Regeln an. Diese betreffen den Sichtbarkeitsbereich, also die im Kontext einer bestimmten Operationsausführung zugreifbaren Klassen.  $E$  bezeichne wieder die Menge aller Episoden.

Bezüglich der „ausführenden Klassen“ bzw. der Sichtbarkeitsbereiche von Episodenschritten orientieren wir uns an statisch getypten objektorientierten Programmiersprachen (vgl. [Meyer97]). Es ergeben sich die folgenden unmittelbar einsichtigen Regeln.

**Regel 11.4.1 [Episode-Wurzelklasse]** Die Klasse des ausführenden Objekts des Startschritts einer Episode ist konform zur Wurzelklasse des dem Test-Szenarioschritt zugeordneten Use Case Schritts:  $\forall e \in E: s_0(e).AK \leq e.tsc.ucs.WK$ .

**Regel 11.4.2 [Episode-Konformität1]** Die ausführende Klasse jedes Folgeschritts muss konform zu einer Klasse im Sichtbarkeitsbereich des „aufrufenden“ Schritts sein:

$$\forall e \in E: \forall es \in S(e): s_f \in \sigma(es) \Rightarrow \exists k \in es.SK: s_f.AK \leq k.$$

**Regel 11.4.3 [Episode-Konformität2]** Die in der Signatur der ausgeführten Operation des Folgeschritts enthaltenen Parameter müssen konform zu Klassen im Sichtbarkeitsbereich des „aufrufenden“ Schritts sein:

$$\forall e \in E: \forall es \in S(e):$$

$$s_f \in \sigma(es) \wedge s_f.WO.signature = (k_0, \dots, k_n) \Rightarrow \{(k_0, \dots, k_{n-1})\} \subseteq_I SK(es).$$

Betrachten wir den Sichtbarkeitsbereich noch unter dem Aspekt der Vererbung. Ist die ausführende Klasse eines Episodenschritts Unterklasse einer Klasse im Sichtbarkeitsbereich des

Vorgängerschritts, so wollen wir wissen, in welcher Klasse die ausgeführte Operation definiert ist.

Es liegt nahe, alle geerbten Eigenschaften quasi mittels "Copy & Paste" einfach in die erben-  
de Klasse zu übernehmen (vgl. [Binder95]). Bei redefinierten Operationen sorgen wir dabei  
durch geeignete Umbenennungen dafür, dass sowohl die geerbte als auch die redefinierte  
Operation innerhalb der Klasse sichtbar sind (vgl. [PalSch94]).

- ❑ Wird eine Operation in einer Unterklasse überschrieben, so wird dem Namen der Ope-  
ration der Name der definierenden Klasse vorangestellt und durch „::“ abgetrennt.

Jede Benutzung der in einer Oberklasse definierten (und in der kontextgebenden Klasse über-  
schriebenen) Operation<sup>1</sup> wird dann analog behandelt:

- ❑ Der Name der Oberklasse wird dem Botschaftsnamen vorangestellt und durch „::“ ab-  
getrennt.
- ❑ Die Operation mit dem so entstehenden Bezeichner wird „benutzt“<sup>2</sup>.

Im Kontext „aufrufender“ Objekte ist nur die redefinierte Operation sichtbar. Die umbenann-  
te Operation kann also nicht in Episodenschritten, denen Instanzen anderer Klassen zugeord-  
net sind, „aufgerufen“ werden. Dieses auch als „Abflachen“ der Vererbungshierarchie  
bekannte Vorgehen (*flattening*, vgl. [Binder95]) führt im schlimmsten Fall zu einer | **K** |-fa-  
chen Vervielfältigung einer Operation.

## 11.5 Objektorientierte Walk-Throughs

Während der Validierung noch konkrete Objektkonstellationen für die Szenarien zugrunde-  
lagen, konzentrieren sich Analytiker und Tester in der Verifikation mehr auf das Klassenmo-  
dell, also auf die systeminterne Information der Anforderungsspezifikation. Wir schlagen die  
folgende Form des *objektorientierten Walk-Through*‘s vor (vgl. auch [Winter97]):

- ❑ Die Objekte jeder beteiligten Klasse werden von einer Person vertreten, die auch den  
Zustand der Objekte (Werte der Attribute, Referenzen auf andere Objekte) nachhält.
- ❑ Beginnend mit dem Ereignis, welches die durchzuspielende Aufgabe auslöst, werden  
anhand der Beschreibung der Operationen Botschaften zwischen den Objekten bzw.  
den sie vertretenden Personen gesendet und so die Episode simuliert und protokolliert.  
Die zugrundeliegende Objektkonstellation kann entweder vorgegeben oder — besser,  
aber zeitraubender — von einem „Wurzelobjekt“ ausgehend innerhalb der Walk-Th-  
roughs selber aufgebaut werden.

---

<sup>1</sup> Entspricht einem Auftreten von **super** in der objektorientierten Programmierung.

<sup>2</sup> Programmiertechnisch: alle Vorkommen von **super** werden durch **self** ersetzt.

**Formale Vollständigkeit**

- Ist jede Klasse in wenigstens einem Klassenbereich enthalten?
- Ist jedem Interaktionsschritt eine Wurzeloperation zugeordnet?
- Sind alle Klassen der Parameter der Wurzeloperation im Klassenbereich des „aufrufenden“ Interaktionsschritts enthalten?
- Ist der Klassenbereich eines Makro-Schritts eine Teilmenge des Klassenbereichs des referenzierten Use Cases?
- Gibt es Referenzen auf nicht existierende Use Cases?
- Gibt es Klassen im Bereich eines Schritts, die nicht im Klassenmodell existieren?
- Gibt es Wurzeloperationen, die nicht in der Wurzelklasse definiert sind?

**Konsistenz**

- Bezieht sich die Spezifikation eines Interaktionsschritts nur auf Klassen in seinem Klassenbereich?
- Impliziert die Vorbedingung eines Interaktionsschritts die der Wurzeloperation?
- Stellen die Nachbedingungen der in einer Episode „ausgeführten“ Operationen die des vom Test-Szenarioschritt referenzierten Interaktionsschritts sicher?
- Falls eine Kante  $e$  des Schrittgraphen von einem Test-Szenario überdeckt ist, ist ihre Bedingung  $c(e)$  in der augenblicklichen Objektkonstellation erfüllt?
- Sind für eine Objektkonstellation verschiedene Szenarien eines Use Case möglich?
- Ist die Disjunktion der Bedingungen aller ausgehenden Kanten eines inneren Schritts von der Nachbedingung des Schritts impliziert?
- Sind verschiedene Episoden einer Wurzeloperation für eine bestimmte Objektkonstellation möglich?
- Ist eine Operation Wurzeloperation verschiedener Schritte (in verschiedenen Use Case Schrittgraphen)?
- Sind für eine Operation widersprüchliche Episoden simuliert worden?
- Sind alle Objektkonstellationen verträglich zum Klassenmodell?
- Treten in den Episoden „Nutzungs-Anomalien“ auf (create-destroy, modify-modify, destroy-read)

*Abb. 11.1 Exemplarische Checklistenpunkte für die Verifikation*

Die Simulation der Episoden deckt fehlerhafte Klassenschnittstellen, fehlende oder in der Multiplizität fehlerhafte Beziehungen und Probleme der Vererbungsstruktur auf. Unter anderem werden bei den Walk-Throughs folgende Punkte beachtet:

- ☐ Sind alle verwendeten Attribute auch initialisiert?
- ☐ Bestehen Referenzen zu den Zielobjekten der von der Operation ausgesendeten Botschaften (Attribut bzw. Parameter der die Operation auslösenden Botschaft)?
- ☐ Sind die aktuellen Parameter der von der Operation ausgesendeten Botschaften bekannt? Stimmen Ihre Typen bzw. „verstehen“ die Zielobjekte die Botschaft?
- ☐ Werden die Rückgabewerte benutzter Operationen adäquat weiterverarbeitet?

Abb. 11.1 zeigt exemplarisch einige Checklistenpunkte für die Verifikation.

Episoden werden in SCORES mit UML-Sequenzdiagrammen (vgl. [OMG97]) visualisiert, die jeweils dem entsprechenden „auslösenden“ Schritt des Test-Szenarios zugeordnet sind. Hierzu betrachten wir ein kleines Beispiel.

*Beispiel 11.1.* Für den Bankautomaten zeigt Abb. 11.1 das Sequenzdiagramm einer Episode.

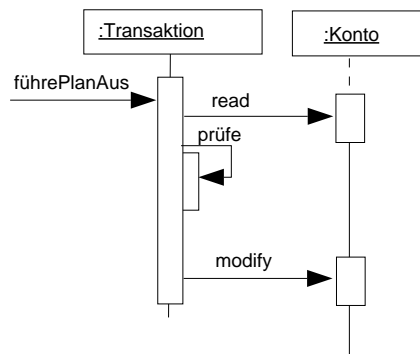


Abb. 11.1 Sequenzdiagramm, Episode mit Wurzelklasse **Transaktion** und Wurzeloperation *führePlanAus*

Dem zugehörigen Use Case Schritt ist die Wurzelklasse **Transaktion** bzw. die Wurzeloperation *führePlanAus* zugeordnet. Der „Aufruf“ der Wurzeloperation *führePlanAus* führt zu einem Aufruf der Standardoperation *read* im Kontext eines Objekts der Klasse **Konto**. Danach wird die Operation *prüfe* der Klasse **Transaktion** aufgerufen — in diesem Falle muss sich aus der betrachteten Objektkonstellation ergeben und im Episodenschritt protokolliert werden, ob sich dieser Aufruf an das gleiche Objekt richtet oder an eine andere Instanz der Klasse **Transaktion** bzw. einer zu ihr konformen Klasse. Letztendlich wird die Standardoperation *modify* im Kontext des Objekts der Klasse **Konto** aufgerufen.

□

## 11.6 Vollständiges Scores-Metamodell

In diesem Abschnitt erweitern wir die in den Abbildungen 5.7, 7.2 und 9.1 dargestellten partiellen Sichten des SCORES Metamodells zum letzten Mal. Wir beziehen Primitive für Episoden ein und erhalten das vollständige SCORES Metamodell in Abb. 11.2. Der Constraint zwischen den Klassen **Episode** und **InteractionStep** deutet an, dass Episoden nur dann modelliert werden, wenn der dem Szenarioschritt zugeordnete Use Case Schritt ein Interaktionschritt ist — nur in diesem Fall wird die Aufgabe ja von der Anwendung unterstützt. Die Klasse **EpisodeStep** ist (ebenso wie nun auch die Klasse **InteractionStep**) Unterklasse der (abstrakten) Klasse **SupportedStep**, so dass sich jeder Episodenschritt auf genau eine Klasse und eine darin enthaltene Operation abstützen kann. Mit der Aggregation *calls* modellieren wir den „Aufruf“ beliebig vieler weiterer Episodenschritte (Folgeschritte). Dies entspricht

der Intention, dass Episoden (sowie die sie visualisierenden Sequenz- und Kollaborationsdiagramme) Aufruf-Bäume von Operationen darstellen.

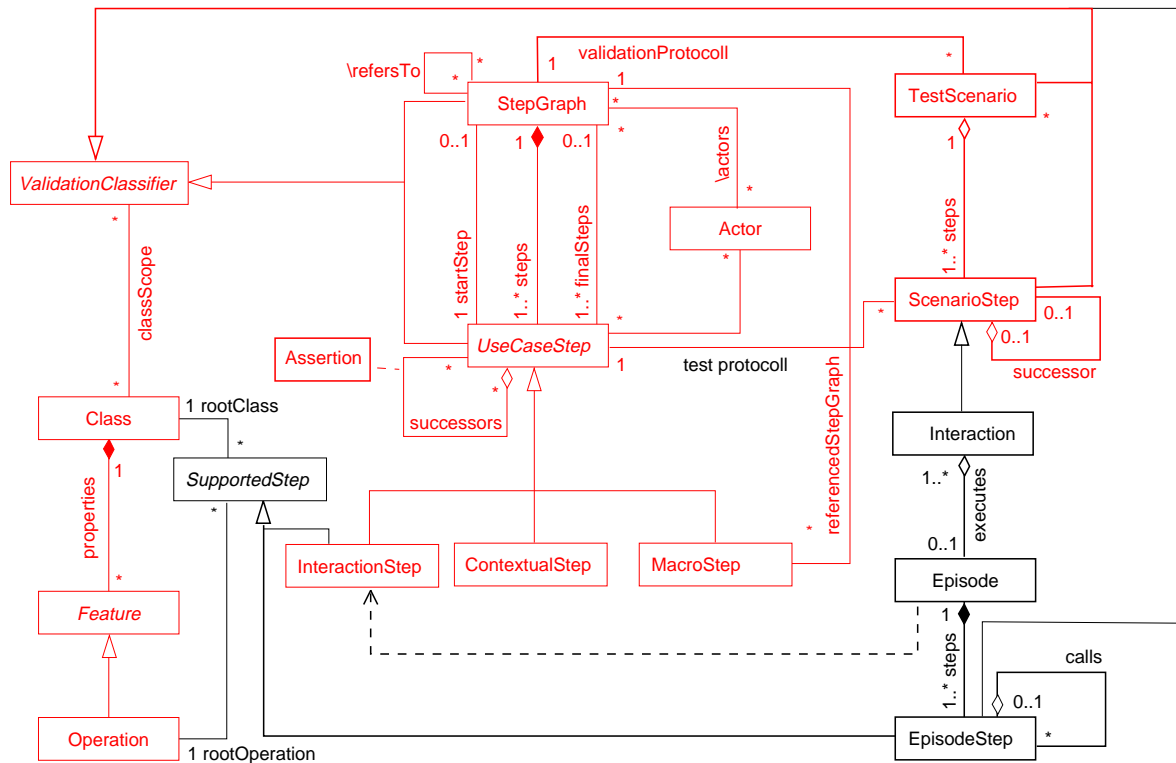


Abb. 11.2 Vollständiges SCORES Metamodell

# Kapitel 12

## Verifikationsmetriken

Während die Validierung hauptsächlich die funktionalen Aspekte der Anforderungsspezifikation, also die Use Cases und ihre Schrittgraphen betrifft, bezieht sich die Verifikation auch auf das Zusammenspiel der funktionalen und strukturellen Aspekte, also zusätzlich auf das Domänen-Klassenmodell. Objekte „verstecken“ ihre Daten bzw. ihren Zustand und lassen Zugriffe nur über wohldefinierte Operationen zu. Dementsprechend erfolgt die Kommunikation zwischen Objekten ausschließlich über Operationsaufrufe (Botschaften), was sich auf die Kommunikation im System fortsetzen lässt.

Die Bedingungen der Schritte und Kanten im Use Case Schrittgraph werden mit der minimalen Mehrfach-Bedingungsüberdeckung (Abschnitt 12.1) verifiziert. Da wir darüber hinaus das „Verhalten“ der Domänenklassen prüfen, zielen die strukturell orientierten Verifikationsmetriken hauptsächlich auf Testmethoden für die im Klassenmodell definierten Operationen ab und geben Analytikern und Testern Kriterien dafür, wie umfassend das Klassenmodell verifiziert wurde (Abschnitt 12.2). Bezüglich der in den Definitionsgleichungen der Metriken verwendeten SCORES-Primitive verweisen wir auf das Metamodell in Abb. 11.2.

### 12.1 Minimale Mehrfach-Bedingungsüberdeckung

Testszenarien alleine nach der Struktur der Use Case Schrittgraphen auszuwählen ist nicht adäquat, da die eigentlichen Spezifikationen, d. h. die Vor- und Nachbedingungen der Use Cases bzw. der einzelnen Use Case Schritte nicht im Mittelpunkt ihrer Ermittlung stehen. Analytiker und Tester können die Vor- und Nachbedingungen unter Ausnutzung folgender Eigenschaften verifizieren:

- Jede Bedingung besteht aus einer prädikatenlogischen Formel bzw. einem boole'schen (OCL-) Ausdruck, meistens in Form einer UND bzw. ODER-Verknüpfung oder einer Implikation zwischen Termen (auch atomare Prädikate genannt, vgl. [PagSix94]) und Ausdrücken, wobei wir zudem Selektoren mit boole'schen Rückgabewerten sowie Vergleiche als Terme ansehen.

- Die Vorbedingung muss vor der Bearbeitung der (Teil-) Aufgabe des Schritts erfüllt sein — dies schränkt die Anzahl der benötigten Testfälle ein und gibt Hinweise auf notwendige Initialisierungen. Insbesondere muss die Vorbedingung des Use Cases vor dem Test erfüllt sein.
- Die Nachbedingung wird von der Aufgabe des Schritts zugesichert, wenn vor ihrer Bearbeitung die Vorbedingung erfüllt ist — die Nachbedingung spezifiziert also einerseits das erwartete Ergebnis der Testfälle. Andererseits können wir versuchen, durch die Auswahl gewisser „Eingangsdaten“ eine bestimmte Belegung der Terme der Nachbedingung zu erzwingen oder sogar die Aufgabe des Schritts quasi so zu „sabotieren“, dass die Nachbedingung trotz erfüllter Vorbedingung nicht erfüllt wird.

Genauso wie in bedingten Anweisungen oder Schleifen innerhalb von Methoden der Kontrollfluss gesteuert wird, „steuern“ die Bedingungen die Traversierung von Kanten und damit die Ausführung oder Nicht-Ausführung der Aufgaben von Schritten.

Nun sollten die Bedingungen der Kanten in verschiedenen Szenarien zu einem Use Case Schrittgraph im Allg. beide Wahrheitswerte — true und false — annehmen, da sonst evtl. unerreichbare Schritte, überflüssige Kanten oder nie ausgeführte bzw. nicht terminierende Schleifen vorliegen. Die Vor- und Nachbedingungen der Schritte jedoch sollen vor- und nach einer Aufgabebearbeitung immer den Wert true annehmen, da im Falle einer verletzten Zusage ja ein Fehler aufgetreten ist. Das Ziel dieses „Vertragstests“ (vgl. [Overbeck94]) ist es also, mit gezielten Szenarien die Nachbedingungen der Schritte negativ zu prüfen, d.h. zu false auszuwerten.

Kontrollflussorientierte Testverfahren unterscheiden zwischen der *einfachen Bedingungsüberdeckung* ( $C_2$ -Test), bei der alle Terme einer Bedingung mindestens einmal den Wert true und einmal den Wert false erhalten müssen, und der *Mehrfach-Bedingungsüberdeckung* ( $C_3$ -Test), welche die Überprüfung aller Wertekombinationen fordert (vgl. [Riedemann97]). Die einfache Bedingungsüberdeckung wird als ein zu schwaches Kriterium angesehen, da sie nicht einmal die Zweigüberdeckung umfasst. Das Problem der Mehrfach-Bedingungsüberdeckung ist der exponentielle Anstieg der Testfälle in Abhängigkeit von der Anzahl der vorkommenden Terme. Für den effektiven Test von Programmen wird daher oft die *minimale Mehrfach-Bedingungsüberdeckung* herangezogen:

Die *minimale Mehrfach-Bedingungsüberdeckung*, die einen guten Kompromiss zwischen der einfachen und mehrfachen Bedingungsüberdeckung darstellt, wertet jedes Prädikat — ob atomar oder nicht atomar — zu beiden Wahrheitswerten aus. Dadurch wird die hierarchische Struktur von Bedingungen berücksichtigt und jeder Schachtelungsstufe die gleiche Aufmerksamkeit gewidmet. [PagSix94]

Nach [Riedemann97] ist die minimale Mehrfach-Bedingungsüberdeckung genau dann erreicht, wenn für jeden boole'schen Ausdruck alle Belegungen der Terme erzwungen wurden,



bei denen die Änderung des Wahrheitswerts eines Terms den Wahrheitswert des zusammengesetzten Ausdrucks ändert.

Beginnen wir mit der Verifikation der Vorbedingungen. Da (positive) „Testfälle“ die Vorbedingungen einhalten müssen, wenden wir die minimale Mehrfach-Bedingungsüberdeckung zunächst eingeschränkt auf insgesamt erfüllte Vorbedingungen an:

- ☐ Liegt ein Term **A** vor, erzeugen wir einen Testfall mit ( $A=\text{true}$ ), liegt ein negierter Term **NICHT A** vor, erzeugen wir einen Testfall mit ( $A=\text{false}$ ).
- ☐ Im Falle von **ODER**-verknüpften Termen **A** und **B** muss jeder Term einmal den Wert **true** und einmal den Wert **false** erhalten, wir erzeugen also für eine Vorbedingung vom Typ **A ODER B** zwei Testfälle, welche die Belegungen ( $A=\text{true}, B=\text{false}$ ) und ( $A=\text{false}, B=\text{true}$ ) erzwingen.
- ☐ Bei **UND**-verknüpfter Terme brauchen wir nur einen Testfall mit ( $A=\text{true}, B=\text{true}$ ) zu erzeugen,
- ☐ im Falle einer Implikation **WENN A DANN B** zwei Testfälle mit ( $A=\text{false}, B=\text{false}$ ) und ( $A=\text{true}, B=\text{true}$ ).
- ☐ Bei geschachtelten Ausdrücken werden alle die Vorbedingung erfüllenden Belegungen geprüft, bei denen die Änderung des Wahrheitswerts eines Terms den Wahrheitswert des zusammengesetzten Ausdrucks ändert

Bezüglich der Nachbedingungen wollen wir auch mögliche Verletzungen erreichen — wir befinden uns ja auf der Jagd nach Fehlern. Wir versuchen laut der minimalen Mehrfachbedingungsüberdeckung Testfälle für folgende Wahrheitswert-Kombinationen der Terme in den Nachbedingungen zu finden:

- ☐ **ODER**-verknüpfte Terme versuchen wir zu ( $\text{false}, \text{false}$ ) als negativen Testfall sowie zu ( $\text{true}, \text{false}$ ) und ( $\text{false}, \text{true}$ ) als positive Testfälle zu erzwingen.
- ☐ **UND**-verknüpfte Terme versuchen wir, zu ( $\text{true}, \text{true}$ ) als positiven Testfall und ( $\text{true}, \text{false}$ ) sowie ( $\text{false}, \text{true}$ ) als negative Testfälle zu erzwingen.
- ☐ Die beiden Terme einer Implikation versuchen wir dementsprechend zu ( $\text{false}, \text{false}$ ) und ( $\text{true}, \text{true}$ ) als positive Testfälle sowie ( $\text{true}, \text{false}$ ) als negativen Testfall zu erzwingen.
- ☐ Bei geschachtelten Ausdrücken werden alle Belegungen geprüft, bei denen die Änderung des Wahrheitswerts eines Terms den Wahrheitswert des zusammengesetzten Ausdrucks ändert

Das Vorgehen erläutern wir in Beispiel 12.2. weiter unten. Für die negativen Testfälle müssen wir — bei korrekten und vollständigen Spezifikationen — die Vorbedingung verletzen, da sonst die Nachbedingung ja „garantiert“ wird. Wir setzen voraus, dass für alle Use Cases und Use Case Schritte bzw. Kanten die minimale Mehrfach-Bedingungsüberdeckung erreicht

wird. Dies führt natürlich in Abhängigkeit von der Komplexität der einzelnen Bedingungen zu einer großen Anzahl ähnlicher Szenarien — bei „operationalen“ Bedingungen in einer formalen Sprache wie z.B. OCL oder sogar in einer Programmiersprache ist die Testfallerzeugung allerdings mit bekannten Techniken automatisierbar (vgl. [Poston94][Poston96]).

## 12.2 Vollständige und polymorphe Operationsüberdeckung

Als eine erste, einfache Überdeckungsmetrik messen wir mit der *vollständigen Operationsüberdeckung*  $c_1^{\text{KM}, \text{O}}$ , welcher Prozentsatz der Operationen mit Episoden simuliert und damit mindestens einmal bei der Verifikation betrachtet wurde. Zuerst ermitteln wir für eine Episode  $e$  die benutzten Operationen (12.1), dann für ein Test-Szenario  $ts$  die Menge der Episoden (12.2) und schließlich die Menge der von einem Test-Szenario „ausgeführten“ Operationen im Klassenmodell (12.3).

$$e.\text{operationsUsed} \equiv \bigcup_{est \in e.\text{steps}} est.\text{rootOperation} \quad (12.1)$$

$$ts.\text{episodes} \equiv \bigcup_{s \in (ts.\text{Steps} \cap \text{InteractionStep.allInstances})} s.\text{executes} \quad (12.2)$$

$$ts.\text{operationsUsed} \equiv \bigcup_{e \in ts.\text{episodes}} e.\text{OperationsUsed} \quad (12.3)$$

Hiermit können wir die vollständige Operationsüberdeckung angeben zu

$$c_1^{\text{KM}, \text{O}} \equiv \frac{\left| \bigcup_{ts \in \text{TestScenario.allInstances}} ts.\text{operationsUsed} \right|}{\text{Operation.allInstances} \rightarrow \text{size}} \quad (12.4)$$

Wir erhalten  $c_1^{\text{KM}, \text{O}} = 1$ , wenn jede Operation im Klassenmodell in zumindest einer Episode verwendet wird.

Diese Metrik können wir in mehrfacher Hinsicht verfeinern. Zum einen entlang der Vererbungshierarchie, wobei wir z.B. verlangen können, dass eine vererbte Operation mindestens dann in einer Episode im Kontext der Unterklassen simuliert wird, wenn sie (in mindestens einer Episode im Kontext der Oberklasse) eine in der Unterklasse redefinierte Operation benutzt (vgl. [PerKai90][HMGF92]). Zum anderen ist es sinnvoll, für die einzelnen Elemente des Klassenmodells dediziert anzugeben, wie weit der Verifikationsprozess gediehen ist. Betrachten wir zunächst ein Beispiel einer Vererbungshierarchie.

*Beispiel 12.1.* Wir betrachten die vier Klassen **A**, **B**, **C** und **D** in Abb. 12.1 (a). Klasse **A** definiert die beiden Operationen *a* und *b*. Die Klassen **B** und **D** sind Unterklassen von Klasse **A**, wobei Klasse **B** Operation *b* redefiniert. Klasse **C** definiere die Operation *c*, Klasse **D** Opera-

tion d. In der in Abb. 12.1 (b) gezeigten Episode mit Wurzelklasse **A** und Wurzeloperation **a** benutzt die Operation **a** zunächst Operation **b**, welche als Rückgabewert ein Objekt der Klasse **C** liefere; **C** kommt damit in den Sichtbarkeitsbereich der Ausführung von Operation **a** in Klasse **A**, die dann die Operation **c** aufrufe. Da Operation **b** in Klasse **B** überschrieben ist, fordern wir, dass eine zusätzliche Episode mit Wurzelklasse **B** und (der unverändert geerbten) Wurzeloperation **a** simuliert wird. Diese soll zeigen, dass die redefinierte Operation **b** den Sichtbarkeitsbereich bei der Ausführung von Operation **a** im Kontext der Klasse **B** tatsächlich um die Klasse **C** (bzw. eines ihrer Objekte) erweitert. Da Klasse **D** keine Operation redefiniert, die in einer im Kontext der Klasse **A** simulierten Episode benutzt wird, ist hier keine zusätzliche Episode notwendig.

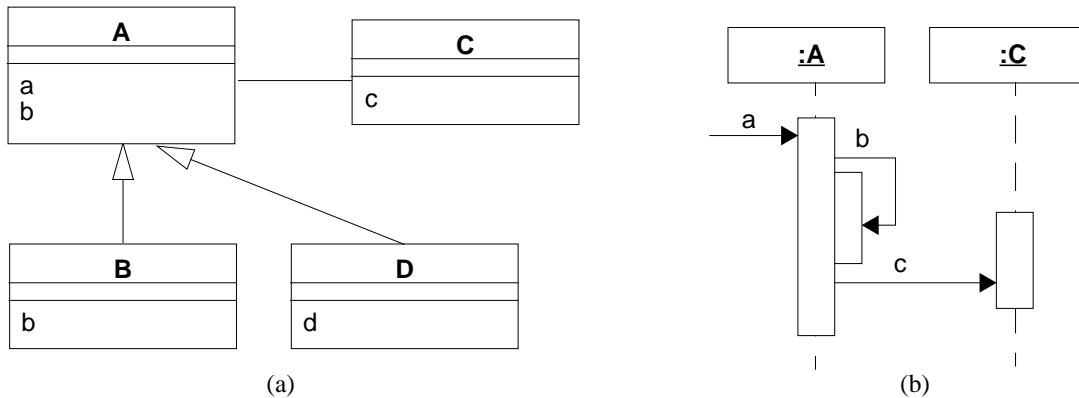


Abb. 12.1 Klassendiagramm und Episode mit redefinierter Operation

□

Für den Fall redefinierter Operationen wollen wir also ermitteln, für welche redefinierten Operationen, die in einer Episode der Oberklasse benutzt werden, in der redefinierenden Unterklasse noch keine Episode simuliert wurde. Wir bezeichnen die entsprechende Metrik als *polymorphe Operationsüberdeckung*  $c_p^{KM, O}$  (12.10). Zunächst ermitteln wir für eine Klasse **k** die Menge der in ihr (neu- oder re-) definierten Operationen (12.5).

$$k.operations \equiv \{p \in k.properties \mid p \in Operation.allInstances\} \quad (12.5)$$

Operation (12.6) bestimmt (rekursiv) die von **k** (unverändert) geerbten Operationen.

$$k.inheritedOperations \equiv \bigcup_{\{sk \in Class.allInstances \mid k \leq sk\}} sk.inheritedOperations \cup sk.operations \quad (12.6)$$

Damit können wir die in der Klasse **k** redefinierten Operationen bestimmen zu

$$k.redefinedOperations \equiv \{op \in k.inheritedOperations \mid \exists rop \in k.operations : (op.name = rop.name) \wedge (op.signature = rop.signature)\} \quad (12.7)$$

Für eine im Klassenmodell definierte Operation  $op$  ermitteln wir nun, ob  $op$  im Kontext seiner definierenden Klasse in einer Episode simuliert wurde.

$$\begin{aligned} op.isSimulated &\equiv \exists es \in EpisodeStep.allInstances : \\ es.rootOperation &= op \wedge es.rootClass = op.Class \end{aligned} \quad (12.8)$$

Zuletzt müssen wir ermitteln, welche dieser redefinierten Operationen in Episodenschritten im Kontext der Oberklassen benutzt wurden. Dies ermittelt Funktion (12.9).

$$\begin{aligned} k.redefinedAndUsedOps &\equiv \{op \in k.redefinedOperations | \\ \exists (e \in EpisodeStep.allInstances : e.rootClass \geq k \wedge e.rootOperation = op) \} \end{aligned} \quad (12.9)$$

Den Prozentsatz der redefinierten und sowohl in Episoden der Oberklasse als auch der Unterklasse benutzten Operationen berechnet die polymorphe Operationsüberdeckung (12.10)

$$\begin{aligned} \frac{KM_c}{p} \equiv \frac{|\bigcup_{c \in Class.allInstances} \{op | (op \in c.redefinedAndUsedOps \wedge op.isSimulated)\}|}{|\bigcup_{c \in Class.allInstances} c.redefinedAndUsedOps|} \end{aligned} \quad (12.10)$$

Zur Illustration der minimalen Mehrfach-Bedingungsüberdeckung sowie dem objektorientierten Walk-Through mit der Simulation von Episoden folgt nun noch jeweils ein Beispiel.

*Beispiel 12.2.* Betrachten wir die Spezifikationen der Schritte und Kanten des Use Case Schrittgraphen Anmelden in Abb. 5.5 bzw. 5.6. Um das Beispiel übersichtlich zu halten, beschränken wir uns auf die vier Schritte Karte Einführen, Karte Lesen, PIN Anfordern und PIN Prüfen. In Tab. 12.1 haben wir zunächst die Vor- und Nachbedingungen dieser vier Schritte nach Konjunktionstermen getrennt aufgelistet<sup>1</sup>. Bei der Festlegung der Testfälle gehen wir iterativ vor, indem wir in Szenarien die Schritte nacheinander abarbeiten, bis der geforderte Überdeckungsgrad für jede Bedingung erreicht ist (oder, bei negativen Testfällen, als nicht erreichbar erkannt ist). Zunächst folgt aus der Vorbedingung des Use Cases Anmelden (Abb. 5.3), dass zu Beginn jedes Szenarios der Kartenleser betriebsbereit und das Bedienpult gesperrt sein muss. Jedes Szenario beginnt mit dem Startschritt, hier also Karte Einführen, dessen Vorbedingung von der des Use Cases impliziert sein muss. „Interaktionsparameter“ des ersten Schritts sind die (physikalische) Karte bzw. die auf dem Magnetstreifen oder Mikrochip gespeicherten Daten.

In Szenario A gehen wir von einer lesbaren, zum Zugriff auf den Automaten berechtigenden Karte aus. Nachdem diese im Kontextschritt Karte Einführen vom Akteur Bankkunde richtig

<sup>1</sup> Tab. 12.1 kann als verkürzte *Entscheidungstabelle* (vgl. [Beizer90]) angesehen werden, bei der Bedingungen direkt auf Aktionen (Szenarien) abgebildet werden. In der vollständigen Entscheidungstabelle werden alle Terme als Bedingungen und die Use Case Schritte als Aktionen eingetragen. Eine Regel entspricht dann der Belegung aller Interaktionsparameter der in einem Szenario ausgeführten Schritte.

<i>Schritt-Bedingungen</i>	<i>Szenarien</i>				
	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
Karte Einführen <b>Vorbedingung</b> Kartenleser betriebsbereit	true	true	true	true	true
Karte Einführen <b>Nachbedingung</b> Karte im Kartenleser <b>UND</b> Kartenleser gesperrt	true	true	true	true	true
Karte Lesen <b>Vorbedingung</b> Karte im Kartenleser <b>UND</b> Kartenleser gesperrt	true	true	true	true	true
Karte Lesen <b>Nachbedingung</b> Karte lesbar <b>UND</b> Code gültig	true	false	true	true	true
Karte Lesen <b>Nachbedingung</b> <b>NICHT</b> Karte lesbar	false	true	false	false	false
Karte Lesen <b>Nachbedingung</b> <b>NICHT</b> Code gültig	false	false	true	false	false
PIN Anfordern <b>Vorbedingung</b> Karte im Kartenleser <b>UND</b> Kartenleser gesperrt <b>UND</b> Karte lesbar <b>UND</b> Code gültig	true	false	false	true	true
PIN Anfordern <b>Nachbedingung</b> PIN gelesen	true	-	-	false	true
PIN Anfordern <b>Nachbedingung</b> Abbruch	false	-	-	true	false
PIN Prüfen <b>Vorbedingung</b> Karte im Kartenleser <b>UND</b> Kartenleser gesperrt <b>UND</b> Karte lesbar <b>UND</b> Code gültig <b>UND</b> PIN gelesen	true	false	-	false	true
PIN Prüfen <b>Nachbedingung</b> PIN OK	true	-	-	-	false
PIN Prüfen <b>Nachbedingung</b> <b>NICHT</b> PIN OK	false	-	-	-	true

Tab. 12.1 Testfallermittlung zur minimalen Mehrfach-Bedingungsüberdeckung

ausgerichtet in den Kartenleser eingeführt wurde, erwarten wir, dass sich die Karte im Kartenleser befindet, und der Kartenleser gesperrt ist. Diese Belegung der Terme in der Nachbedingung erfüllt auch die Vorbedingung des Folgeschritts Karte Lesen, den wir als nächstes ausführen können, da ja auch die Bedingung der entsprechenden Kante (hier trivialerweise) erfüllt ist. Aufgrund der Annahme „lesbare, zum Zugriff auf diesen Automaten berechtigenden Karte“ wird diese vom Kartenleser gelesen, so dass der erste Konjunktionsterm der Nachbedingung erfüllt wird. Da diese Belegung auch die Bedingung der Kante vom Schritt Karte Lesen zum Schritt PIN Anfordern erfüllt, setzt letzterer das Szenario A fort. Interaktionspara-

meter dieses Schritts ist die PIN (als vierstellige Folge von Ziffern), wobei wir in diesem Szenario von der Eingabe der richtigen PIN ausgehen. Auf diese Weise fahren wir fort, bis ein Endschrift des Schrittgraphen erreicht und keine Fortsetzung möglich ist.

Da wir die Nachbedingung des Startschritts Karte Einführen nur durch mechanische Einwirkung verletzen (geknickte Karte...) und insbesondere die beiden Terme der Konjunktion nicht getrennt beeinflussen können, betrachten wir diesen Schritt als „ausreichend getestet“ und konzentrieren uns in Szenario B auf den zweiten Konjunktionsterm der Nachbedingung des Schritts Karte Lesen (also Kartenleser gesperrt **UND NICHT** Karte lesbar). Als Test-Eingabedatum gehen wir von einer entmagnetisierten, nicht lesbaren Karte aus, so dass der zweite Term erfüllt wird; Szenario B endet dementsprechend mit dem Endschrift Karte Einziehen. In einem weiteren Szenario C wird jetzt noch die Eingabe einer lesbaren, aber für diesen Automaten nicht gültigen Karte geprüft. Die Konjunktionsterme der Nachbedingung des Schritts Karte Lesen können wir nicht gleichzeitig zu false zwingen und erachten somit auch diesen Schritt für ausreichend getestet.

Entsprechend betrachten wir mit den Szenarien D und E die Schritte PIN Anfordern und PIN Prüfen. In Szenario D wählen wir als Interaktionsparameter des Schritts PIN Anfordern den Abbruch des Vorgangs, so dass der zweite Konjunktionsterm der Nachbedingung des Schritts PIN Anfordern (Karte im Kartenleser **UND** Kartenleser gesperrt **UND** Karte lesbar **UND** Karte gültig **UND** Abbruch) erfüllt ist und das Szenario mit dem Schritt Karte schreiben fortgesetzt wird (im Beispiel nicht mehr dargestellt). In Szenario E geben wir eine (syntaktisch richtige, aber bezüglich der Karte) falsche PIN ein und erfüllen so den zweiten Konjunktionsterm der Nachbedingung des Schritts PIN Prüfen. Da wir weitere negative Testfälle (physikalisch) nicht ausführen können, betrachten wir die vier Schritte bezüglich der minimalen Mehrfach-Bedingungsüberdeckung ausreichend getestet.

□

*Beispiel 12.3.* Wir betrachten einen Walk-Through durch den Schrittgraphen des Use Cases Anmelden in der Bankautomaten-Fallstudie (vgl. Abb. 5.8 (b)). Das in Abb. 5.9 (b) gezeigte Szenario setzt eine Verbindung zum Zentralrechner voraus (der Bankautomat sei also „Online“). Zusammen mit den in Beispiel 7.2., Abb. 7.5 dargestellten Klassenbereichen und den Vor- und Nachbedingungen der Use Case Schritte erhalten wir das in Abb. 10.3. dargestellte Test-Szenario Online-Anmeldung, bei dem die eingegebene PIN durch den Zentralrechner geprüft werden kann. Mögliche Fragestellungen bei dem Walk-Through sind

- Existiert eine Instanz der Klasse **Kartenleser** zu Beginn des Test-Szenarios?
- Wann wird eine Instanz der Klasse **Transaktion** erzeugt?
- Welchen Zustand hat die Instanz der Klasse **Transaktion**
  - nach der Erzeugung?
  - nach den einzelnen Szenarioschritten?

- nach dem Szenario?

- ☐ Wie kann die Instanz der **Zentralrechnerschnittstelle** die PIN prüfen?
- ☐ Sind die resultierenden Objektkonstellationen sinnvoll?
- ☐ Wo manifestiert sich der „Zustand“ des Bankautomaten?

Wir simulieren noch eine Episode zum Szenarioschritt „PIN prüfen“ Test-Szenario Online-Anmeldung (Abb. 10.3). Für den Fall des verfügbaren Zentralrechners prüft dieser die eingegebene PIN sowie ggf. eine vorherige Sperrung der Karte. Die in der Wurzelklasse **Transaktion** definierte Operation `prüfePIN` benutzt die gleichnamige Operation der Klasse **ZentralrechnerSchnittstelle**, die der Transaktion über eine Beziehung „bekannt“ ist. Wie diese Operation in der **ZentralrechnerSchnittstelle** mit dem zentralen Bankrechner kommuniziert, sei an dieser Stelle nicht betrachtet — die Operation ist also ein Blatt der Episode. Im betrachteten Test-Szenario habe der Benutzer die richtige Geheimnummer eingegeben (Szenarioschritt „PIN anfordern“). Die Zentralrechner-Schnittstelle bestätigt dies; die darauf folgende Standard-Operation `modify` beschreibt das Rücksetzen des Attributs `AnzahlVersuche` des die Kreditkarte darstellenden Objekts der Klasse **Karte** (Abb. 12.2).

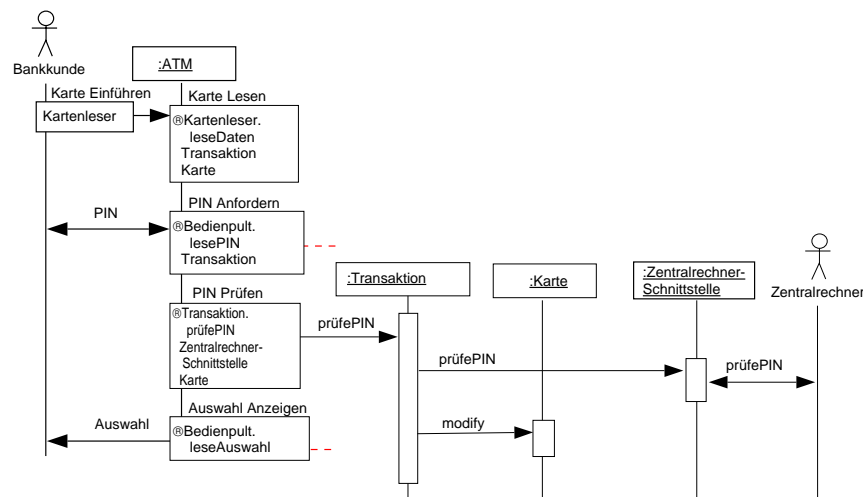


Abb. 12.2 Episode zum Schritt PIN Prüfen im Test-Szenario Online-Anmeldung

Hätten wir angenommen, dass der Bankautomat keine Verbindung zum Zentralrechner aufbauen kann, würde die Episode in Abb. 12.3 zutreffen, in der die Transaktion sich selbst mit

der Operation offline in eben diesen Zustand versetzt und danach die eingegebene PIN durch eine entsprechende Operation der Klasse **Karte** prüfen lässt.

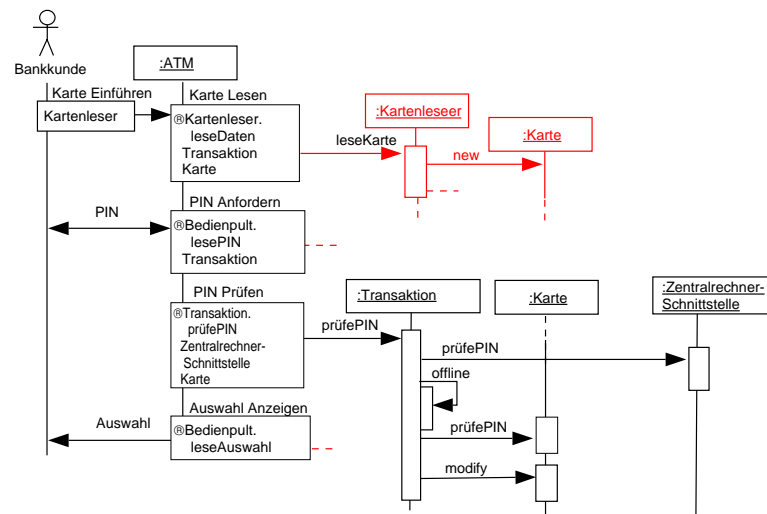


Abb. 12.3 Episode zum Schritt PIN Prüfen im Test-Szenario Offline-Anmeldung



Hiermit sind wir am Ende unserer Ausführungen zu den qualitätssichernden Tätigkeiten bei der Anforderungsermittlung angelangt. Zur besseren Einordnung fasst Tab. 12.2 die verwendeten Begriffe und SCORES-Elemente noch einmal bezogen auf die Tätigkeiten Spezifikation, Validierung und Verifikation zusammen.

	Spezifikation	Validierung (Workflow-Aspekt)	Verifikation
Funktionale Zerlegung	Use Cases, Use Case Schritte	Organisation, Funktionalität	Vor- und Nachbedingungen
Externes Verhalten	Use Case Schrittdiagrammen	Operational, Ablauf, Kausalität, Geschäftsvorfälle	Übergangsbedingungen, Test-Szenarien
Struktureller Aufbau	Klassen, Attribute, Assoziationen	Information, Objektkonstellationen	Klassenmodell
Internes Verhalten	Klassenbereiche, Verantwortlichkeiten	Verantwortlichkeiten von Domänenobjekten	Operationen, Episoden

Tab. 12.2 Einsatz der SCORES-Elemente bei der Anforderungsermittlung

Nach einem „Exkurs“ zur Verfolgbarkeit der Elemente der Anforderungsspezifikation über den Entwurf in die Implementation kommen wir dann im nächsten Teil zum Test der Anwendung gegen die Anforderungsspezifikation.



# Kapitel 13

## Verfolgbarkeit

*Traceability is a tremendously important property in system development!*  
*[JCJ+92]*

In der Anforderungsermittlung haben wir Use Cases zu Use Case Schrittgraphen verfeinert und Use Case Schritte mit Domänenklassen und (Wurzel-) Operationen verknüpft. Zusätzlich haben wir bei der Validierung der Anforderungsspezifikation Test-Szenarien durchgespielt und bei ihrer Verifikation mit Episoden die Ausführung der (Wurzel-) Operationen im Kontext des Test-Szenarios simuliert. Neben der rein funktionalen Wiederverwendung der Test-Szenarien als black-box Testfälle werden wir die in den Episoden enthaltene systeminterne Information als Testorakel sogenannter SCORES grey-box Testfälle für den entwurfsbasierten Systemtest (vgl. [Hetzel88]) verwenden. Diese überprüfen, ob bei der Ausführung eines Test-Szenarioschritts für alle in der Episode durchgespielten Operationen bzw. alle von ihr „berührten“ Elemente des Domänen-Klassenmodells auch die entsprechenden Elemente der Implementation ausgeführt bzw. „berührt“ werden. Notwendige Voraussetzung für die Ableitung von Testfällen für den grey-box Test ist, dass wir für alle Elemente der Anforderungsspezifikation entsprechende Elemente in der Implementation identifizieren können.

Wir müssen die bei der Anforderungsermittlung erfasste systeminterne Information also über alle Entwicklungstätigkeiten hinweg auf die Elemente der Implementation abbilden (Abb. 13.1). Bereits in der Einführung haben wir die Methodik OOSE ([JCJ+92]) skizziert, welche die Elemente des Domänen-Klassenmodells als „Entitätsklassen“ in den Entwurf einbettet. In diesem Kapitel stellen wir zunächst allgemeine Betrachtungen zur Verfolgbarkeit an. Dann betrachten wir in Abschnitt 13.2, wie sich die im Entwurf durchgeführten Änderungen am Klassenmodell auf die Elemente der SCORES-Anforderungsspezifikation auswirken und wie wir die in den Episoden enthaltene systeminterne Information durch den Entwurf verfolgen.

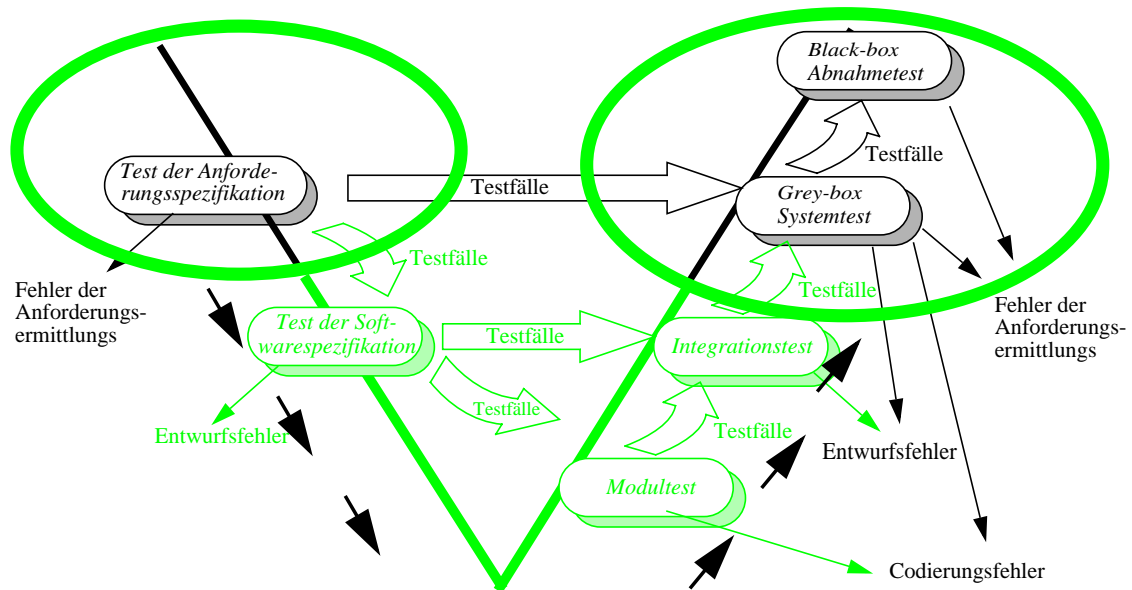


Abb. 13.1 Verfolgbarkeit für den Grey-box Systemtest

## 13.1 Konzepte

Das Identifizieren von aufeinander aufbauenden bzw. (inhaltlich) zusammenhängenden Elementen wird als *Verfolgbarkeit* (*traceability*) bezeichnet (vgl. [IEEE830.93][GotFin94]). Was aber bedeutet Verfolgbarkeit genau, und wie wird sie ermöglicht? Eine Definition gibt die IEEE Norm 830 (Guide to Software Requirements Specifications):

A SRS (software requirements specification) is traceable, if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. [IEEE830.93]

Man kann bei der Verfolgbarkeit zunächst die horizontale und vertikale Verfolgbarkeit unterscheiden (vgl. [Davis90][RamEdw93]).

- ❑ Die *horizontale Verfolgbarkeit* setzt einerseits Elemente der gleichen Entwicklungstätigkeit in einen (zeitlichen) Bezug — man spricht in diesem Zusammenhang auch von *Versionen* eines Artefakts. Andererseits beschreibt die horizontale Verfolgbarkeit, wie bestimmte Modellierungselemente innerhalb einer Entwicklungstätigkeit durch andere (gleichartige Elemente) verfeinert bzw. präzisiert werden.
- ❑ Die *vertikale Verfolgbarkeit* betrachtet die Relationen zwischen Elementen verschiedener Entwicklungstätigkeiten und gibt Antwort auf die Frage, wie ein bestimmtes Element (z.B. eine Klasse im Code) aus einem anderen (z.B. einer Entwurfsklasse) hervorgegangen ist.

Relativ zur Anforderungsspezifikation lassen sich mit der obigen Definition die in Abb. 13.2 gezeigten vier Richtungen der vertikalen Verfolgbarkeit ableiten (nach [Davis90]):

- ❑ Verfolgbarkeit „zurück von den Anforderungen“ impliziert das Wissen, warum ein Element der Anforderungsspezifikation existiert, also eine Referenz zur Quelle der Anforderung (Gesprächsprotokolle, Skizzen, ...).
- ❑ Verfolgbarkeit „vorwärts zu den Anforderungen“ bedeutet, dass jedes der Anforderungsspezifikation vorausgehende Dokument die aus ihm hervorgegangenen Elemente der Anforderungsspezifikation referenziert.
- ❑ Verfolgbarkeit „vorwärts von den Anforderungen“ impliziert das Wissen, welche Elemente der Anwendung eine bestimmte Anforderung befriedigen, also z.B. welche Entwurfselemente ihren Ursprung in einer bestimmten Anforderung haben.
- ❑ Verfolgbarkeit „zurück zu den Anforderungen“ ermöglicht, zu jedem Element der Anwendung festzustellen, welche Anforderung es zu erfüllen hilft.

Bei den ersten beiden Richtungen wird auch von *Vor-Verfolgbarkeit* (*pre-traceability*) gesprochen, bei den letzten beiden dementsprechend von *Nach-Verfolgbarkeit* (*post-traceability*) (vgl. [GotFin94]).

Im weiteren Verlauf dieser Arbeit sind wir im Wesentlichen am Test der (objektorientierten) Anwendung gegen die Anforderungsspezifikation interessiert. Somit steht die Nach-Verfolgbarkeit der Anforderungsspezifikation über den Entwurf in die Implementation im Mittelpunkt. Der eigentliche Entwurf bzw. die Evolution bestehender Entwürfe und/oder Anwendungen ist nicht Gegenstand der Arbeit. Daher betrachten wir im Rahmen dieser Arbeit nicht die horizontale Verfolgbarkeit im Sinne des zeitlichen Bezugs, sondern sehen diese

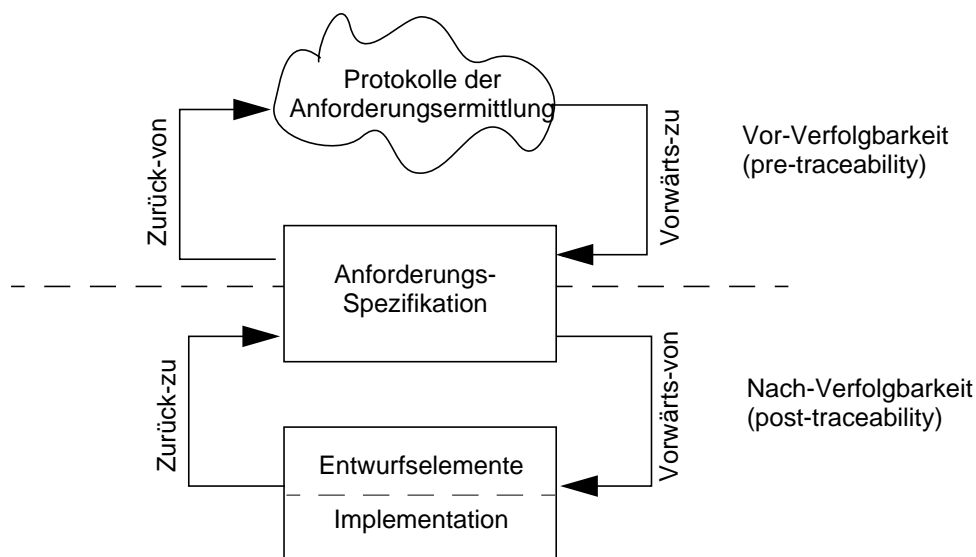


Abb. 13.2 Vertikale Verfolgbarkeit (nach [Davis90])

rein als Gegenstand des Konfigurations-Managements bzw. der Versionsverwaltung (vgl. [Tichy94]). Auch die Vor-Verfolgbarkeit wird in diesem Zusammenhang nicht betrachtet. Wir sprechen somit im Weiteren, wenn keine Verwechslungsgefahr besteht, anstelle von vertikaler Nach-Verfolgbarkeit nur noch von *Verfolgbarkeit*.

Wir vereinbaren nun einige Relationen, mit denen wir später zu den Elementen der Anforderungsspezifikation die entsprechenden Entwurfs- bzw. Implementationselemente ermitteln können. In Anlehnung an die UML verwenden wir Abhängigkeitsbeziehungen zur Modellierung der Verfolgbarkeitsrelationen. In dieser Arbeit genügen uns die zwei Relationen *Verfolgung* (*trace*, [OMG97]) und *Verfeinerung* (*refinement*, [OMG97]) als Unterarten der allgemeinen Relation *Abhängigkeit* (*dependency*, [OMG97]). Im UML-Semantics Guide lesen wir:

A *dependency* indicates a semantic relationship among model elements themselves (rather than instances of them) in which a change to one element may affect or require changes to other elements.

A *trace* is a conceptual connection between two elements or sets of elements that represent a single concept at different semantic levels or from different points of view. However, there is no specific mapping between the elements. The construct is mainly a tool for tracing of requirements. It is also useful for the modeler to keep track of changes to different models.

A *refinement* is a relationship between model elements at different semantics levels, such as analysis and design [...] The derivation cannot necessarily be described by an algorithm; human decisions may be required to produce the clients. [OMG97] (semantics guide)

Abb. 13.3 stellt einen Ausschnitt des UML-Metamodells dar. Objekte der Klassen **Trace** und **Refinement** als Unterklassen von **Dependency** fassen jeweils eine Menge von Modellierungselementen zusammen. Zur Verfolgbarkeit dienen Instanzen der Klasse **Trace**, die Modellierungselemente aus *verschiedenen* Modellen verknüpfen. Kann über eine einfache Zuordnung hinaus angegeben werden, wie die Modellelemente einander entsprechen, wird diese „Verfeinerung“ genannte Abhängigkeit im Attribut Mapping einer Instanz der Klasse **Refinement** festgehalten. Während Instanzen der Klasse **Trace** also lediglich *syntaktische*

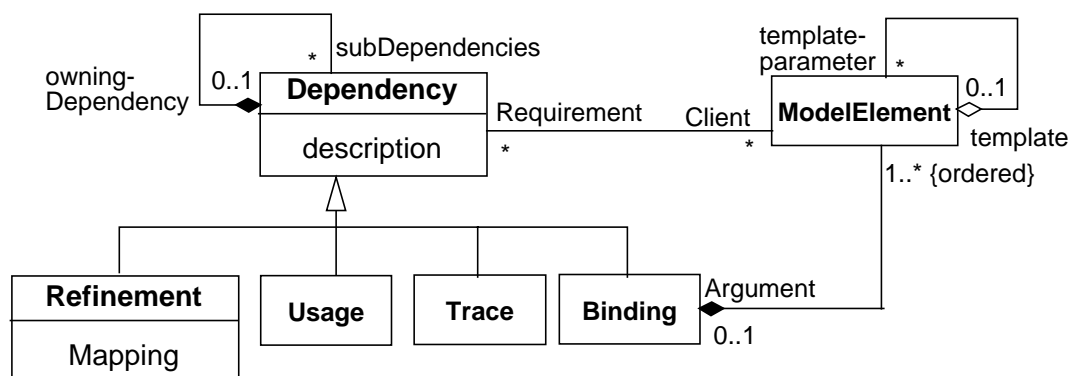


Abb. 13.3 UML Auxiliary Elements — Dependencies and Templates ([OMG97])

*Abhängigkeiten* angeben, erfassen Instanzen der Klasse **Refinement** zusätzlich *semantische Abhängigkeiten*, die z.B. zur Steuerung von Verfolgbarkeits-Anfragen etc. verwendet werden können (vgl. [PinGog96]).

Sei  $\mathbf{M} = \mathbf{ModelElement.allInstances}$  die Menge aller Modellierungselemente. Jede Instanz der Klasse **Dependency** (bzw. einer der angesprochenen Unterklassen) erfasst eine Teilmenge von  $\mathbf{M}$ . Für ein Modellierungselement  $x$  bedeutet jede Assoziation zu einer Instanz  $t$  von **Dependency**, dass  $x$  abhängig von jedem anderen an der Assoziation beteiligten Modellierungselement  $y \in t.Client$  ist. Die Menge aller Instanzen von **Dependency** ergibt mit der Äquivalenz (13.1) eine reflexive, symmetrische Relation  $dependsOn \subseteq \mathbf{M} \times \mathbf{M}$ .

$$dependsOn(x,y) \Leftrightarrow \exists d \in \mathbf{Dependency.allInstances}: x \in d.Client \wedge y \in d.Client \quad (13.1)$$

Ebenso können wir speziellere Relationen *refinedBy* oder *tracesTo* definieren. Sei z.B.  $\mathbf{m}_A$  ein Modell der Anforderungsspezifikation,  $\mathbf{m}_D$  ein Entwurfsmodell und  $\mathbf{m}_I$  die Implementation. Für ein bestimmtes Element der Anforderungsspezifikation  $x \in \mathbf{m}_A$  erhalten wir mit dem Ausdruck (13.2) die Menge aller Abhängigkeiten, die  $x$  enthalten.

$$x.allDependencies \equiv \{d \in \mathbf{Dependency.allInstances} | (x \in d.client)\} \quad (13.2)$$

Die Menge aller Entwurfselemente, die zu  $x$  zurück-verfolgbar sind, erhalten wir dann mit dem Ausdruck (13.3).

$$x.allDesignElements \equiv \{y \in \mathbf{m}_D | dependsOn(x,y)\} \quad (13.3)$$

Dementsprechend ermitteln wir mit (13.4) die Menge aller das Element  $x$  der Anforderungsspezifikation im Entwurf verfeinernden Modellelemente.

$$x.allDesignRefinements \equiv \{y \in \mathbf{m}_D | refinedBy(x,y)\} \quad (13.4)$$

Für unsere Belange genügen diese Ausführungen. Wie bzw. durch wen oder welche Tätigkeit diese Relationen bei der Modellierung gesetzt werden, ist nicht Gegenstand dieser Arbeit. Hinweise und technische Einzelheiten finden sich z.B. in [Corriveau96] und [PinGog96].

## 13.2 Elemente der SCORES-Anforderungsspezifikation

Sowohl OOA/D ([CooYou91]) als auch OOSE ([JCJ+92]) bieten klare Richtlinien, was die Einbettung der Domänenklassen in den Entwurf angeht: In OOA/D werden sie zunächst „eins zu eins“ übernommen, in OOSE spielen sie im Entwurf die Rolle der „Entitätsklassen“. Während jedoch OOA/D die gesamte Funktionalität quasi über die Entwurfsklassen verteilt, kap-

selt OOSE Teile der fachlichen Logik, also die „Use-Case-Funktionalität“, in speziellen Kontroll-Klassen. Beide Entwurfsmethoden spalten die Benutzungsoberfläche ähnlich ab: OOA/D führt den GUI-Bereich ein, OOSE sieht sog. Interface-Klassen als Mediator zwischen der („eigentlichen“) Benutzungsoberfläche und den Kontroll-Klassen vor (s. auch Abschnitt 2.2). Für OOSE gibt Tab. 13.1 einige Beispiele der Verfolgbarkeit von Elementen der Anforderungsspezifikation über den Entwurf in die Implementation.

<i>Analyse</i>	<i>Entwurf</i>	<i>Implementation</i>
Analyseobjekt	Block	Eine oder mehrere Klassen
Verhalten	Operation	Operation
Attribut (der Klasse)	Attribut (der Klasse)	Statische Variable
Attribut (der Instanz)	Attribut (der Instanz)	Instanzvariable
Aggregation	Aggregation	Instanzvariable
Temporäre Assoziation	Temporäre Assoziation	Botschaftsparameter
Interaktion	Stimulus	Botschaft / Prozeduraufruf
Use Case	Interaktionsdiagramm	Botschaftsfolge
Teilsystem	Teilsystem	Paket

Tab. 13.1 Verfolgbarkeit in OOSE (nach [JCJ+92])

Natürlich wundert es nicht, auch in kommerziellen Frameworks (vgl. z.B. [GHJ+94] [BMR+97][Johnson97][Züllighoven98]) ähnliche Entwurfsentscheidungen vorzufinden, da sich Coad und Yourdon ([CoaYou91]) explizit auf das MVC-Framework ([GolRob89]) berufen und auch Jacobson die OOSE auf in kommerziellen Projekten gewonnenen Erfahrungen aufgebaut hat und umgekehrt ([JCJ+92][JGJ97]). Auch Züllighoven schreibt:

Die anwendungsfachliche Modellierung bestimmt die technische. Dies geschieht nicht durch eine einfache Eins-zu-eins-Transformation aller fachlichen Begriffe in Klassen und aller Umgangsformen in Operationen. Trotzdem lässt sich das im Modell des Anwendungsbereichs realisierte Begriffsgebäude ohne Modellbruch und erkennbar strukturähnlich in Klassen und Klassenbeziehungen übertragen. [Züllighoven98]

Wie Cook und Daniels in der Einleitung zu ihrer objektorientierten Entwicklungsmethode Syntropy schreiben, gilt also (vgl. insbesondere auch [CoaYou91][JCJ+92]):

All object-oriented methods are based on the notion that "objects" identified in the problem (analysis) have a meaningful place in the solution (design). [CooDan94]

Im Normalfall werden jedoch in der inkrementellen, iterativen Entwicklung während des Entwurfs auch Elemente der Anforderungsspezifikation geändert. Änderungen, die aufgrund

von geänderten Anforderungen bzw. Änderungen in der Realität notwendig sind, nehmen wir direkt in der Anforderungsspezifikation vor, berücksichtigen sie in den SCORES-Elementen und „reichen sie in den Entwurf durch“. Ist eine „technische“ Klasse ohne Entsprechung in der Realität bzw. in der Anforderungsspezifikation von der Änderung betroffen, so sind (zunächst) keine Änderungen an den SCORES-Elementen notwendig — wir erinnern daran, dass SCORES nur der Anforderungsermittlung „angemessene“ Elemente enthält. Oft führen allerdings nachträglich noch die Änderungen von Beziehungen und Operationen (Erweiterungen, „Umleiten“, Aufrufe von Wurzel-Operationen aus der Benutzungsschnittstelle, ...) zu entsprechenden Änderungen des in den Entwurf übernommenen Domänen-Klassenmodells.

Kurz gesagt sind von der Änderung von Vererbungsbeziehungen und (Wurzel-) Operationen die Klassenbereiche der Use Cases und Use Case Schritte sowie Episoden betroffen. Änderungen von anderen Beziehungen oder Attributen betreffen nur die Episoden, wobei wir in diesem Zusammenhang auch an die Standard-Operationen für Zugriffe auf Beziehungen und Attribute erinnern (s. Abschnitt 11.3).

Im Prinzip lassen sich Änderungen am Klassenmodell auf wenige verfeinernde, verfolgbare Operatoren<sup>1</sup> zurückführen (vgl. [JCJ+92][Opdyke92][Hürsch95][Rumpe96]), welche wir in den SCORES-Elementen folgendermaßen berücksichtigen:

- ❑ Einfügen einer neuen Klasse, Abb. 13.4 (a). SCORES-Elemente können die neue Klasse nicht referenzieren, sind also durch das Einfügen der Klasse (erst einmal) nicht betroffen.
- ❑ Entfernen einer Klasse, Abb. 13.4 (b). Normalerweise sind die SCORES-Elemente nicht (mehr) betroffen, da vorher mit den weiter unten aufgeführten Änderungen die Klasse sozusagen aus dem Entwurf „herausgeschält“ wurde. Referenzieren SCORES-Elemente die Klasse dennoch, so sind nach dem Entfernen der Klasse aus dem Entwurf und den SCORES-Elementen (i.e. Klassen-Bereiche sowie Episoden) erneute Walk-Throughs der betroffenen Test-Szenarien durchzuführen.
- ❑ Aufspalten einer Klasse bzw. Verschmelzen mehrerer Klassen. Diese Operatoren sind auf Kombinationen der beiden ersten Operatoren zurückführbar.
- ❑ Einfügen einer Vererbungsbeziehung, Abb. 13.4 (c). Hat die Oberklasse eine Entsprechung im Domänen-Klassenmodell, so müssen ggf. SCORES-Elemente der Oberklasse im Kontext der Unterklasse erneut verifiziert werden (vgl. [PerKai90][HMGF92]). Natürlich sind später auch dementsprechende „geerbte“ Tests notwendig. Haben Ober- und Unterklasse Entsprechungen im Domänen-Klassenmodell, so sind zusätzlich alle SCORES-Elemente „polymorph“ zu erweitern, die Objektbeziehungen zur Ober-

---

<sup>1</sup> Besonders interessant sind in diesem Zusammenhang semantikerhaltende Änderungen, sog. Refaktorisierungen ([Opdyke92]). Diese sind automatisch auch auf die SCORES-Elemente fortsetzbar und erleichtern so die Wartung der Testfälle.

klasse verwenden. In allen anderen Fällen sind SCORES-Elemente (zunächst) nicht betroffen.

- ❑ Entfernen einer Vererbungsbeziehung, Abb. 13.4 (d). Haben Ober- und Unterklasse Entsprechungen im Domänen-Klassenmodell und sind für die Unterklasse eigene, auf geerbte Elemente der Oberklasse zurückgreifende SCORES-Elemente definiert worden, so werden die SCORES-Elemente entsprechend geändert (Löschen aller geerbten Elemente) und erneut validiert und verifiziert. Darüber hinaus müssen Test-Szenarien der Oberklasse, die mit in der Unterklasse redefinierten Operationen beginnen, für die Unterklasse dupliziert, angepasst und erneut durchgespielt werden (vgl. [PerKai90][HMGF92]). In allen anderen Fällen sind SCORES-Elemente nicht betroffen.
- ❑ Einfügen einer Assoziations-, Aggregations- oder Kompositionsbeziehung, Abb. 13.4 (e). Haben beide Beziehungspartner Entsprechungen im Domänen-Klassenmodell, so sind alle Episoden zu prüfen, in denen die neu in Beziehung gesetzten Klassen vorkommen. Ggf. sind die Signaturen von Operationen zu beschränken, da die benutzte

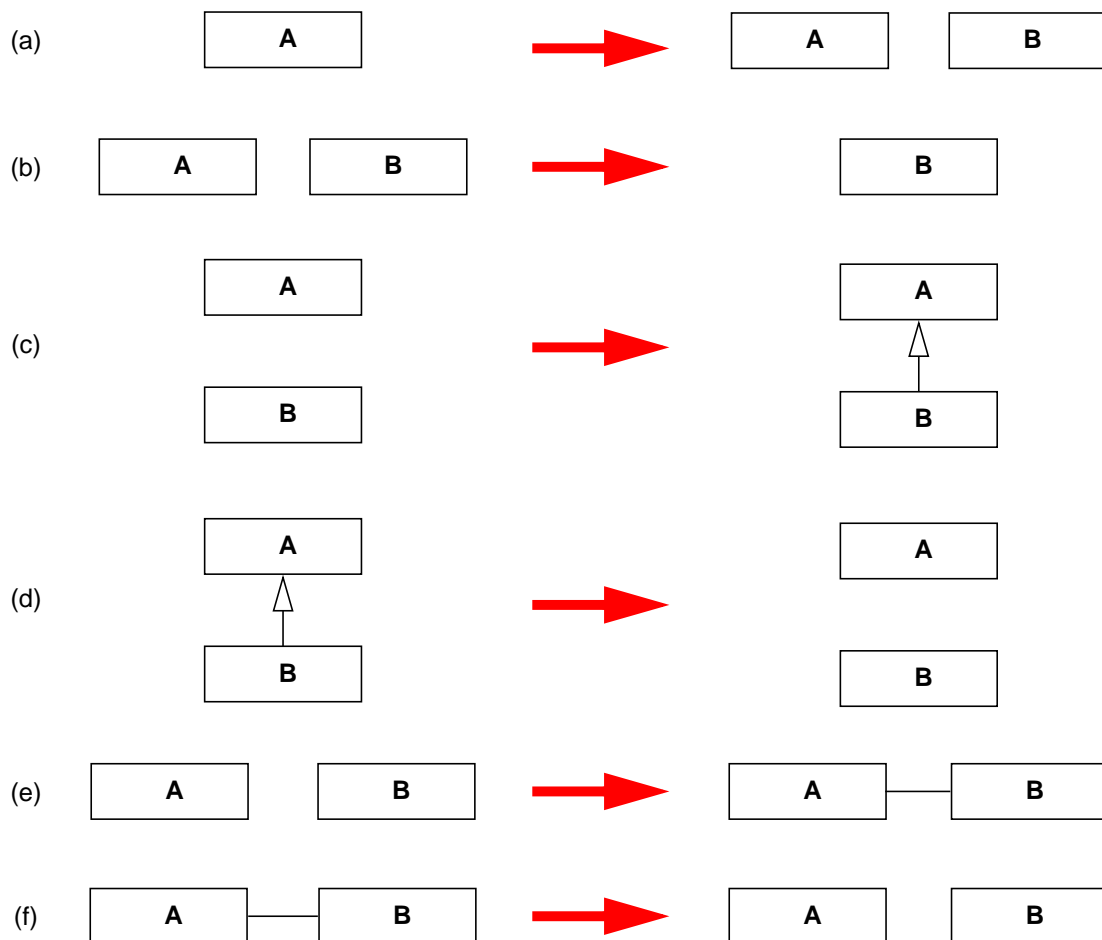


Abb. 13.4 Einfache strukturelle Entwurfsänderungen



Klasse jetzt sozusagen „fest verdrahtet“ (durch entsprechende Standard-Operationen) im Sichtbarkeitsbereich der benutzenden Klasse bzw. Operation ist. Ist einer der beiden Beziehungspartner eine Domänenklasse, so können ggf. weitere Aufrufe zugeordnet werden. In allen anderen Fällen sind SCORES-Elemente nicht betroffen.

- ❑ Entfernen einer Assoziations-, Aggregations- oder Kompositionsbeziehung, Abb. 13.4 (f). Haben beide Beziehungspartner Entsprechungen im Domänen-Klassenmodell, so müssen alle Episoden geprüft werden, welche auf die Beziehung bzw. eine ihr zugeordnete Standard-Operation zugreifen. Ggf. sind die Signaturen von Operationen zu erweitern, um die entsprechende Klasse weiterhin im Sichtbarkeitsbereich zu halten. Ist einer der beiden Beziehungspartner eine Domänenklasse, so können ggf. weitere Aufrufe zugeordnet werden. In allen anderen Fällen sind SCORES-Elemente nicht betroffen.

Zusätzlich betrachten wir noch feingranulare, die Operationen und Attribute betreffende Änderungen.

- ❑ Das Einfügen von Attributen und Operationen wird nur dann berücksichtigt, wenn die geänderten Elemente Entsprechungen in der Realwelt bzw. in der Anforderungsspezifikation haben. Hierbei führt eine neue Wurzeloperation auch zu neuen Test-Szenarien, ansonsten müssen wir ggf. Episoden erweitern.
- ❑ Entfernen von Attributen und Operationen. Attribute sind nur zu beachten, wenn sie in den Spezifikationen explizit angesprochen werden. Wird eine Wurzeloperation entfernt, so müssen die entsprechenden Use Case Schritte, Test-Szenarien und Episoden geändert werden. Handelt es sich um eine im „inneren“ von Episoden verwendete Operation, sind die betroffenen Episoden abzuändern. Ggf. müssen wir die betroffenen Teile der Anforderungsspezifikation erneut verifizieren.

Die Anwendung dieser Operatoren transformiert die Elemente der Anforderungsspezifikation schrittweise in solche des Entwurfs (und der Implementation). Tab. 13.2 zeigt die empfo-

lenen Verfolgbarkeiten (✓) von Elementen der Anforderungsspezifikation und ihren Prüfprotokollen zu Elementen der Implementation (I) bzw. des Tests (T) an.

	I						T				
	<i>Paket</i>	<i>Klasse</i>	<i>Generalisierung</i>	<i>Beziehung</i>	<i>Instanzvariable</i>	<i>Operation</i>	<i>TestSuite</i>	<i>Testskript</i>	<i>Testfall</i>	<i>Testdatum</i>	<i>Testergebnis</i>
<i>Use Case</i>	✓	✓					✓	✓	✓	✓	✓
<i>Use Case Schritt</i>	✓	✓				✓		✓	✓	✓	✓
<i>Test-Szenario</i>	✓	✓				✓		✓	✓	✓	✓
<i>Test-Szenario Schritt</i>	✓	✓				✓			✓	✓	✓
<i>Episodenschritt</i>	✓	✓		✓	✓	✓			✓	✓	✓
<i>Paket</i>	✓	✓					✓	✓	✓	✓	✓
<i>Klasse</i>	✓	✓	✓	✓	✓			✓	✓	✓	✓
<i>Generalisierung</i>		✓	✓	✓	✓	✓		✓	✓		✓
<i>Beziehung</i>		✓		✓	✓	✓			✓	✓	✓
<i>Attribut</i>		✓	✓	✓	✓	✓			✓	✓	✓
<i>Operation</i>		✓	✓	✓	✓	✓			✓	✓	✓

Tab. 13.2 Verfolgbarkeiten zwischen Anforderungsspezifikation, Implementation und Test

Fassen wir zusammen: Aus den verschiedensten Gründen sind im Entwurf Änderungen am Domänen-Klassenmodell notwendig. Durch die Zurückführung von Änderungen auf Operatoren wie den oben genannten sind die Elemente des Domänen-Klassenmodells jedoch über den Entwurf hin bis zur Implementation nachverfolgbar, falls das Domänen-Klassenmodell überhaupt im Entwurf berücksichtigt wird. Dies ist bei den vorherrschenden Entwurfsmethoden der Fall. Wir wissen also, was bei der „Ausführung“ einer Operation im Domänen-Klassenmodell in der Implementation berührt werden sollte — wenn auch nicht unbedingt, in welcher Reihenfolge. Auf letzteren Aspekt gehen wir in den nächsten Kapiteln noch weiter ein.

# Teil IV

## Test gegen die Anforderungsspezifikation

In diesem Teil stellen wir den Test (objektorientierter) Anwendungen gegen objektorientierte Anforderungsspezifikationen vor, dessen spezielle Probleme und Ziele wir zum besseren Überblick in Kapitel 14 repetieren.

Zur Abschöpfung des „Mehrerts“ der SCORES-Anforderungsermittlung leiten wir in Kapitel 15 black-box Testfälle zur Prüfung der Anwendung gegen die Kontextuelle Information und die Interaktionsinformation der Anforderungsspezifikation ab. SCORES ermöglicht es darüber hinaus, die black-box Testfälle zu sogenannten SCORES grey-box Testfällen zu erweitern, welche die internen Interaktionen der Anwendung gegen die bei der Verifikation der Anforderungsspezifikation protokollierte systeminterne Information prüfen. Interne Interaktionen fassen wir dafür in sogenannten Klassen-Botschaftsdiagrammen zusammen (Kapitel 16).

In den beiden letzten Kapiteln dieses Teils gehen wir auf die Komplexität der Generierung von Objektkonstellationen als Testdaten sowie kurz auf die Testausführung und -auswertung ein.

# Kapitel 14

## Probleme und Ziele

In diesem Kapitel zeigen wir die bei dem Test von (objektorientierter) Anwendungen gegen eine objektorientierte Anforderungsspezifikation auftretenden Probleme auf und fassen die Ziele zusammen, die wir in diesem Teil erreichen wollen.

### 14.1 Probleme

In der Literatur sind überwiegend Verfahren zur Prüfung einzelner Klassen (Klassentest) veröffentlicht (vgl. [Binder96a][KHG98]). Diese erweisen sich für objektorientierte Software jedoch zunehmend als wenig effektiv und sind nicht auf den Test komplexer Anwendungen mit hunderten von Klassen erweiterbar (s. Kapitel 3). Rufen wir uns zudem ins Gedächtnis, dass die Komplexität objektorientierter Anwendungen nicht im Quellcode einzelner Methoden, sondern in den Interaktionen liegt, so müssen wir die Effektivität herkömmlicher white-box Testverfahren bzw. struktureller Testkriterien überhaupt in Frage stellen.

Wir erinnern an den Stand der Qualitätssicherung in der industriellen Praxis: systematische Prüfungen von neu erstellten bzw. geänderten Anwendungen werden — wenn überhaupt — überwiegend in späten Phasen der Softwareentwicklung durchgeführt (vgl. [MülWie98]). Aufgrund der o.g. Problematik des white-box Tests spielen hierbei black-box Tests wie z.B. System- und Abnahme-Tests mit „Abdeckung des Pflichtenhefts“ eine prominente Rolle.

Präzise Vorgaben für die Prüfung der Anwendung gegen die (objektorientierte) Anforderungsspezifikation fehlen jedoch, weil die Anforderungsermittlung und insbesondere ihre Qualitätssicherung stiefmütterlich behandelt wird (vgl. [MülWie98]). Aus herkömmlichen Verfahren zur Qualitätssicherung bei der Anforderungsermittlung wie z.B. Reviews und Prototypen (vgl. [Boehm84][Davis90][Gerrard94][Pohl96][WBM96]) ergeben sich darüber hinaus keine unmittelbar verwendbaren Testfälle für den Test der Anwendung gegen die Anforderungsspezifikation. Im Endeffekt stehen die Tester vor der in Abb. 14.1 skizzierten Situation „nicht testbarer Anforderungen“ (vgl. [Poston94]).



runingsspezifikation verwendeten Testkriterien auch beim (dynamischen) Test der Anwendung.

Zusätzlich wollen wir mit den Tests ohne größeren Mehraufwand auch die internen Interaktionen prüfen und gezielt vertraglich geforderte Werte für strukturelle Testkriterien erreichen (Kapitel 16). Im Hinblick hierauf nutzen wir in sogenannten SCORES grey-box Tests die Vorteile der Prüfung des Zusammenspiels mehrerer Methoden mit Aufrufsequenzen unter Minimierung des höheren Aufwands bei der Erstellung der erwarteten Ergebnisse. Hierzu verwenden wir das *Klassen-Botschaftsdiagramm* für die SCORES-Anforderungsspezifikation ( $KBD_A$ ), welches die bei der Verifikation betrachtete systeminterne Information der Episoden geeignet zusammenfasst. Zusätzlich generieren wir auch für die zwischen Implementationsklassen möglichen internen Interaktionen ein *Klassen-Botschaftsdiagramm* ( $KBD_I$ ). Dann geben wir an, wie die systeminterne Information der Anforderungsspezifikation auf die zwischen Implementationsklassen möglichen Interaktionen abgebildet werden kann. Indem wir Episoden quasi als Testorakel verwenden, können wir die Testfälle des black-box Tests erweitern und kommen zu SCORES grey-box Testfällen, die gezielt versuchen, eine bestimmte geforderte strukturelle Überdeckung zu erfüllen.

Für den Test benötigen wir eine geeignete Initialisierung z.B. der persistenten Daten. Dies führt im Falle objektorientierter Daten zum Problem der Generierung von Objektkonstellationen (Kapitel 17). Wir zeigen, dass dieses Problem NP-hart ist und geben Hinweise auf entsprechende Heuristiken.

# Kapitel 15

## Black-box Test

*Black-box testing will shift to testing business solutions. [Pol98]*

Wir wenden uns nun der Testfallermittlung für den Test der Anwendung gegen die Anforderungsspezifikation zu, also im „V-Modell“ (Abb. 14.1, Seite 159) den beiden oberen, mit Fragezeichen markierten Tätigkeiten. In diesem Kapitel leiten wir rein funktionale, auf die kontextuelle Information (Geschäftsprozesse) und die Interaktions-Information bezogene black-box Testfälle ab.

### 15.1 Überblick

Der black-box Test als *anforderungsbasierter Systemtest* zeigt systematisch die Verfügbarkeit aller Funktionen und ihre Übereinstimmung mit der Anforderungsspezifikation auf. Es wird keine systeminterne Information verwendet. Die Testfälle werden aus der Anforderungsspezifikation abgeleitet. Der black-box Test als *Abnahmetest* soll zeigen, dass die Anwendung bereit bzw. geeignet für den Einsatz in der Produktion ist. Er wird in der Regel unter der Regie des Auftraggebers unter Einbeziehung der Benutzer durchgeführt und soll diese davon überzeugen, dass die Anwendung der Anforderungsspezifikation bzw. dem "Pflichtenheft" entspricht. Hierbei werden ausgewählte Test-Szenarien manuell, vornehmlich von Benutzern in ihrer Umgebung, ausgeführt.

Grundlage der black-box Testfälle sind die Use Case Schrittgraphen und die bei der Validierung und Verifikation der Anforderungsspezifikation verwendeten Test-Szenarien, so dass wir die Validierungsmetriken aus Kapitel 10 als Testkriterien für den black-box Test wieder verwenden können. Die prinzipielle Vorgehensweise umfasst folgende Punkte.

- ❑ Aus den einzelnen Schritten eines Test-Szenarios leiten wir *Testfälle* ab. Diese konzentrieren sich auf einzelne Operationen und stellen Aufrufe und — als Testdaten —

Interaktionsparameter zur Verfügung. Die Interaktionsparameter werden aus den Signaturen der Wurzeloperationen im Klassenmodell generiert.

- ❑ Jedes Test-Szenario wird zu einem *Testskript*. Insbesondere die externen Abläufe und der Kontext im Klassenmodell (Objektkonstellationen) sind hier durch die Vor- und Nachbedingungen der Schritte und die Bedingungen der Kanten im Use Case Schrittgraph betrachtet. Testskripte können hierarchisch strukturiert werden.
- ❑ Zu jedem Use Case bzw. Use Case Schrittgraph erstellen wir eine eigene *Testsuite*. Diese dient lediglich als administratives Mittel zur Zuordnung einzelner Test-Skripte zu Use Cases und hat darüber hinaus keine weitere Test Semantik. In der Test-Suite sammeln wir auch die Protokolle bezüglich der mit den Tests erzielten funktionalen und strukturellen Überdeckung.

Wir betrachten die einzelnen Punkte im folgenden Abschnitt genauer. Die resultierende Struktur der SCORES black-box Tests zeigt Abb. 15.1.

## 15.2 Testfallermittlung

Anhand der Aufgabenbeschreibung des Schritts ermitteln wir die vom Benutzer (bzw. Werkzeug) auszuführende Aktion. Das erwartete Ergebnis dieser Aktion ergibt sich aus der konkreten Nachbedingung des Szenarioschritts und der Übergangsbedingung der entsprechenden Kante zum Folgeschritt im Use Case Schrittgraph.

### Interaktionsparameter — atomare Testfälle

Die Objektkonstellationen werden zusammen mit den Interaktionsparametern (Signaturen der Wurzeloperationen) als Eingabemengen der Testfälle ausgewertet. Verfahren zur Ablei-

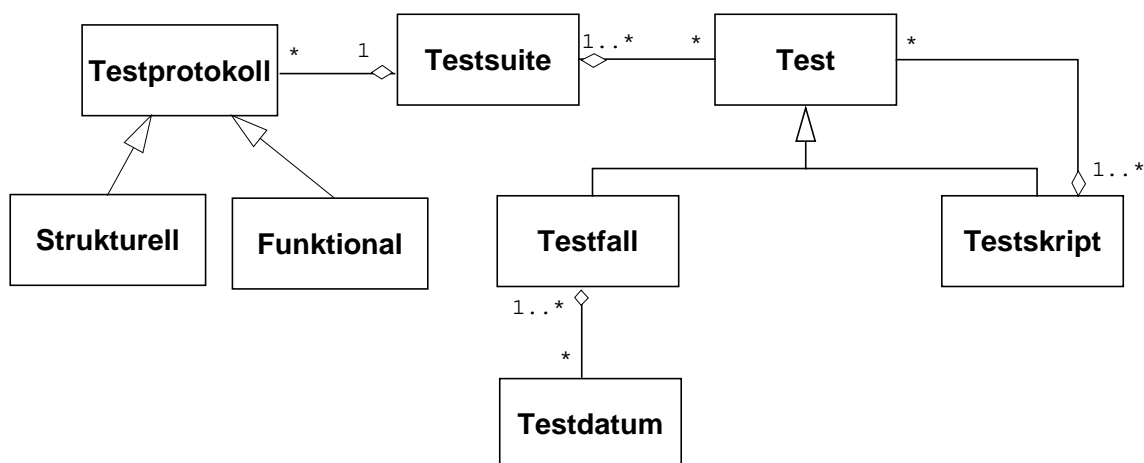


Abb. 15.1 Struktur der Test-Elemente



tung von Testfällen für primitive Interaktionsparameter geben z.B. [Myers79][Beizer90][Beizer95][Poston96] und [Riedemann97]. Hierbei berücksichtigen wir auch die mit der für die Verifikation der Use Case Schrittgraphen durch die minimale Mehrfach-Bedingungsüberdeckung ermittelten Testfälle bezüglich der Vor- und Nachbedingungen der Schritte in den Use Case Schrittgraphen (Abschnitt 12.1).

## Test-Szenario Schritte — Testfälle

Zur Ableitung der Testfälle betrachten wir die Schrittart des dem Test-Szenarioschritt zugeordneten Use Case-Schrittes:

**Interaktionsschritt.** Die Test-Botschaften ergeben sich über die Verfolgbarkeitsrelation aus der Wurzelklasse und -operation. Die Interaktionsparameter ermitteln wir wiederum über die Verfolgbarkeitsrelation aus der/den Signatur/en der Operationen bzw. den Eingabefeldern der Elemente der Benutzungsoberfläche, welche der Wurzeloperation entsprechen. Die entsprechenden Elemente der Benutzungsoberfläche ergeben sich im optimalen Fall aus Entwurfsdokumenten (vgl. [KSV96]). Ansonsten muss die Zuordnung aus einer Analyse der Quellcodes der Anwendung manuell oder semi-automatisch mit Werkzeugen zum Test (grafischer) Benutzungsoberflächen erfolgen. Hierbei hilft eine konsistente Namenswahl, welche Elemente der Oberfläche zu entsprechenden Interaktionsschritten zuordnet. Die Typen der atomaren Interaktionsparameter ermitteln wir dann aus den Attributen der betroffenen Klassen.

**Kontextschritt.** Jeder Kontextschritt beschreibt eine Tätigkeit, die vom Akteur manuell bzw. maschinell ausgeführt wird. Hier werden externe Abläufe simuliert und geprüft, womit die Grenze zum Geschäftsprozess bzw. Business Process Reengineering erreicht ist (vgl. [HamCha94]). Weitere Information ergibt sich ggf. aus dem Klassenbereich des Use Case-Schritts.

**Makroschritt.** Jeder Makroschritt wird zu einem Verweis auf ein Testskript der Test-Suite des referenzierten Use Cases. Wir ermitteln passende Skripts anhand der Vor- und Nachbedingung des dem Makroschritt zugeordneten Schritts im Test-Szenario und der Test-Szenarien des referenzierten Use Cases.

Für alle drei Schrittarten gilt: der Test-Setup wird aus der Vorbedingung des Schritts ermittelt, für das Testorakel ziehen wir die Nachbedingung des Schritts und die Übergangsbedingung der entsprechenden Kante im Use Case Schrittgraph heran. Wir unterscheiden Testfälle für positive und negative Tests: während positive Tests die jeweiligen Vorbedingungen einhalten, verletzen negative Tests ganz bewusst eine Vorbedingung und spezifizieren als erwartetes Ergebnis eine bestimmte Ausnahme (-situation) (vgl. [Berard93]).

Zur textuellen Spezifikation von Testfällen haben wir eine einfache Skriptsprache entwickelt, deren Syntax in Anhang B angegeben ist. Hierzu betrachten wir ein einfaches Beispiel.

*Beispiel 15.1.* Wir beschreiben Testfälle zur Prüfung der Klasse **Stack** mit den Operationen **push** (zufügen eines Elements auf den Stapel), **pop** (entfernen des obersten Elements vom Stapel) und **top** (Zugriff auf das oberste Element des Stapels, vgl. [Meyer97]). Abb. 15.2 zeigt ein entsprechendes Testskript.

```

TESTCASE TFStack1 ROOTCLASS Stack (A, B : Object)
  LOCAL C : Object;
  PRECONDITION A != null AND B != null ;
BEGIN
  CREATE create() ORACLE count == 0;
  push (A) ORACLE count == 1, top == A;
  push (B) ORACLE count == 2, top == B;
  C = top ORACLE count == 2, top == C;
  pop ORACLE count == 1, top == A;
  pop ORACLE count == 0;
  pop EXCEPTION EmptySet "EmptySet";
END

```

Abb. 15.2 Testskript für die Klasse Stack



## Test-Szenario — Testskript

Jeder Test beginnt jeweils mit dem „Hochfahren“ der Anwendung (Test-Setup), wobei wir die Zustände persistenter Objekte aus den konkreten Vorbedingungen des durchzuspielenden Test-Szenarios ermitteln und vor dem Start der Anwendung in der Datenbank setzen oder mit dem Durchspielen „vorgelagerter“ Test-Szenarien (Makroschritte!) direkt mit der Anwendung erzeugen.

In einem Testskript stellen wir anhand der Vor- und Nachbedingungen der Testfälle und der Flussbedingungen der Kanten im Use Case Schrittgraph „passende“ Testfälle zu einem Ablauf zusammen.

Auch hier unterscheiden wir wieder positive und negative Tests: positive Testskripte halten die Bedingungen explizit ein. Gezielte Verletzungen der Constraints für die erlaubten Objektkonstellationen sowie insbesondere der Bedingungen der Schritte und Kanten im Use Case Schrittgraph und der Interaktionsparameter führen zu negativen Testfällen, mit denen wir die Robustheit unserer Anwendung prüfen.

## Use Case und Use Case Schrittgraph — Testsuite

Die Testsuite ordnet die Testskripte und Testfälle bzw. -daten einem bestimmten Use Case zu. Es sind keine Entsprechungen in den Test-Skripten erforderlich, da lediglich eine strukturierende Ebene eingeführt wird. Daher sind in der Syntax für Test-Spezifikationen (Anhang B) auch keine Konstrukte für Testsuiten vorgesehen.

Bei den black-box Tests messen wir die erzielte Überdeckung bezüglich des Use Case Schrittgraphen analog zu den Validierungsmetriken (funktionale Überdeckung, Kapitel 10) und des Quellcodes der Anwendung (strukturelle Überdeckung, Kapitel 16). Zusätzlich werden in black-box Tests auch Reihenfolgebedingungen verwendet (vgl. [Beizer95] [Riedemann97]).

Wie bei strukturellen Programmtests verwenden wir die Validierungsmetriken auch zur Generierung weiterer Test-Szenarien. So ist z.B. die (automatische) Auswertung von Verzweigungsbedingungen und Variablendeklarationen zur Generierung von Testfällen und -daten bekannt (s. [Beizer90][Riedemann97]). Analog zu den Verzweigungsbedingungen werden die Bedingungen der Schritte und Kanten im Use Case Schrittgraph ausgewertet. Analog zu den Deklarationen von Variablen werten wir Attribute der Klassen in den Klassenbereichen und die Signaturen der Wurzeloperationen als Eingaben der Testfälle aus.

### 15.3 Methodisches Vorgehen

Methodisch gehen wir nach dem Algorithmus Black-box Test (Abb. 15.3) vor. Die Reihenfolge der Tests wird durch die invertierte topologische Sortierung der als azyklischen Graph interpretierten Inter-Use Case-Referenzfunktion *REF* bestimmt (s. Definition 5.6 und Regel 5.4.6). Die Testskripte werden im Falle der manuellen Testausführung aus den Beschreibungen der entsprechenden Use Case Schritte generiert. Wird ein Testwerkzeug eingesetzt, so werden die entsprechenden Elemente der Oberfläche mit den dazugehörenden Testdaten in der Skriptsprache des Werkzeugs generiert.

Als Test-Endekriterium wenden wir z.B. „alle Test-Szenarien geprüft“ oder „alle Knoten des Use Case Schrittgraphen überdeckt“ an. Bezüglich weiterer, auf Prozessmetriken wie der

#### Algorithmus Black-box Test

```

    Eingabe: Anforderungsspezifikation;
    Ausgabe: Ausführungsprotokolle;
1   Test-Reihenfolge tre generieren;
2   FORALL uc ∈ UC ORDERED BY tre DO
3       FORALL ts ∈ uc.Test-Szenarien DO {
4           REPEAT
5               Datenbank einrichten;
6               Daten für die Interaktionsparameter der Schritte von ts generieren;
7               Testskript (generieren und) ausführen;
8           UNTIL Test-Endekriterium erfüllt OR Test-Resource für Test-Szenario ausgeschöpft;
        }; /* FORALL */
9       Je nach Ressourcenstand weitere Testfälle ermitteln und ausführen;
END Black-box Test.
```

Abb. 15.3 Algorithmus Black-box Test

*Fehlerrückmeldungsrates*<sup>1</sup> basierender Test-Endekriterien siehe z.B. [Riedemann97] oder [Siegel96].

Offen blieb bisher die Frage, wie die Klassenhierarchie bei der Ausführung der Test-Szenarien berücksichtigt wird? Prinzipiell sind für die Interaktionsparameter alle Kombinationen von Klassen bzw. Unterklassen zu testen. Wenn dies zu einer kombinatorischen Explosion auszuführender Tests führt, sollten wenigstens in gewissem Sinne „repräsentative“ Kombinationen selektiert werden. So sollte jede Klasse mindestens einmal instanziiert und als Interaktionsparameter verwendet werden. Ein systematisches Auswahlverfahren auf der Grundlage des statistischen Entwurfs komplexer Experimente mit orthogonalen Feldern (*Orthogonal Array Test Support*, OATS [Roy90]) geben McDaniel und McGregor an [McDMcG94].

Den prinzipiellen (technischen) Ansatz des black-box Tests der Anwendung zeigt Abb. 15.4.

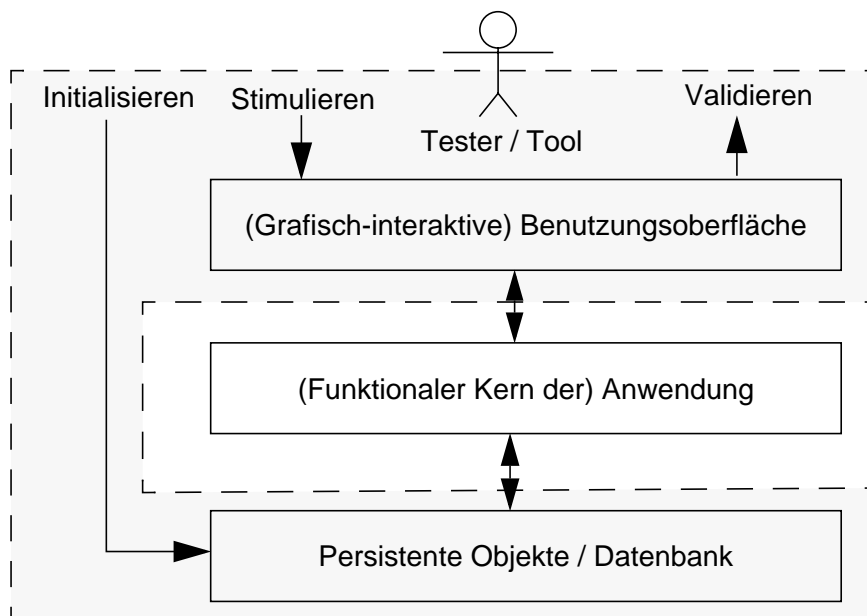


Abb. 15.4 Black-box Test

Nach der Initialisierung der persistenten Daten z.B. mit vorhandenen (anonymisierten) Produktionsdaten wird die Anwendung über die Benutzeroberfläche stimuliert. Nach jedem Test überprüft der Tester bzw. das Test-Werkzeug die resultierenden Ausgaben und damit indirekt den Zustand der Anwendung mit den Möglichkeiten der Benutzeroberfläche (Feldvergleiche, Zustand der Oberflächenelemente, ...).

1 Die Fehlerrückmeldungsrates gibt an, wie viele Fehler pro Zeiteinheit entdeckt werden (vgl. [Riedemann97]).

*Beispiel 15.2.* Für drei Use Cases  $uc\_1$  bis  $uc\_3$  seien  $REF(uc\_1) = \{uc\_2, uc\_3\}$ ,  $REF(uc\_2) = \emptyset$  und  $REF(uc\_3) = \{uc\_2\}$ . Die topologische Sortierung der Knoten des entsprechenden Graphen (Abb. 15.5) ergibt die Test-Reihenfolge ( $uc\_2, uc\_3, uc\_1$ ).

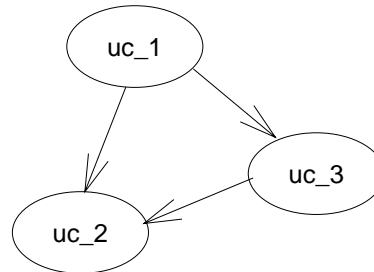


Abb. 15.5 Gerichteter azyklischer Graph zur Inter-Use Case-Referenzfunktion



*Beispiel 15.3.* Abb. 15.6 zeigt ein Testskript mit einigen Testfällen zum Test-Szenario Online-Anmeldung des Use Case Anmelden (Abb. 10.3). Der erste Testfall AnmeldungOK benutzt die beiden Testfälle KarteEinführen und PINAnfordern.

Wir ermitteln die Interaktionsparameter für den Schritt PIN Eingeben im Test-Szenario Online-Anmeldung. Die Wurzeloperation `Transaktion::pruefePIN` erwartet als Eingabeparameter einen Wert vom Typ `integer`<sup>1</sup>. Für diesen Interaktionsparameter ermitteln wir die Äquivalenzklassen `PIN` und `keinePIN`. Für eine Integer-Größe  $i$  bezeichne  $Chars(i)$  die Anzahl der Stellen in der Dezimaldarstellung von  $i$  (ohne führende Nullen). Damit erhalten wir die Äquivalenzklassen  $PIN \equiv \{i \mid i > 0 \wedge Chars(i) = 4\}$  und  $keinePIN \equiv \{i \mid i \notin PIN\} = \{i \mid i \leq 0 \vee Chars(i) \neq 4\}$ .

Die Äquivalenzklasse `PIN` verfeinern wir weiter in die beiden Klassen `GueltigePIN` und `UngueltigePIN`. Die Klasse `GueltigePIN` enthält alle Eingaben, zu denen ein entsprechendes Konto existiert, `UngueltigePIN` enthält alle anderen Eingaben.

---

<sup>1</sup> Wir vernachlässigen hier Eingaben, die überhaupt keiner Integer-Größe entsprechen, da diese über entsprechend prüfende Felder der Benutzungsoberfläche abgefangen werden sollten.

```
TESTCASE AnmeldungOK USECASE Anmelden ( Bankleitzahl: String[8],
    Kontonummer: String[10], Gültigkeitsdatum: Date, Kartennummer: Integer;
    PIN: Integer)
// Der Bankkunde identifiziert sich durch Eingabe der Karte und der PIN.
PRECONDITION
    AccountCreatedOrExists (Bankleitzahl, Kontonummer) AND
    CardNotLocked (Bankleitzahl, Kontonummer, Gültigkeitsdatum, Kartennummer) AND
    ATMReadyAndOnline;
BEGIN
// Karte Eingeben
    KarteEinführen (Bankleitzahl, Kontonummer, Gültigkeitsdatum, Kartennummer)
    ORACLE Kartenleser gesperrt AND Karte im KartenLeserAND Karte lesbar
        AND Code gültig;
// PIN Anfordern
    PINAnfordern (PIN)
    ORACLE PIN OK;
// Auswahl anzeigen
    Bedienpult.MenüAnzeigen()
END.

TESTCASE KarteEinführen ROOTCLASS Kartenleser (Bankleitzahl: String[8],
    Kontonummer: String[10], Gültigkeitsdatum: Date, Kartennummer: Integer)
BEGIN
// Der Bankkunde führt die Karte in den Kartenleser ein.
END

TESTCASE PINAnfordern ROOTCLASS Bedienpult (PIN: Integer)
BEGIN
// Der Bankkunde gibt die PIN ein.
END
```

*Abb. 15.6 Black-box Testfälle zum Use Case Anmelden*



# Kapitel 16

## SCORES grey-box Test

Die black-box Testfälle des vorigen Kapitels prüfen die Anwendung gegen die kontextuelle- und die Interaktionsinformation der Anforderungsspezifikation. Wir wollen darüber hinaus noch prüfen, ob die Implementation der systeminternen Information der Anforderungsspezifikation genügt. Mit den black-box Testfällen können wir dies nur indirekt, indem wir die an der Benutzungsoberfläche (oder ggf. über externe Abfragemöglichkeiten der verwendeten Datenbank) beobachtbaren Zustände und Zustandsänderungen untersuchen.

Verwenden wir nun die bei der Simulation der Episoden festgehaltenen systeminterne Interaktionen als Testmodell, so müssen wir beachten, dass Episoden zunächst nur interne Interaktionen zwischen Domänenklassen widerspiegeln. Wir fassen die systeminterne Information aller Episoden zunächst geeignet im sogenannten Klassen-Botschaftsdiagramm für die Anforderungsspezifikation zusammen (Abschnitt 16.1). In den Abschnitten 16.2 und 16.3 zeigen wir dann, wie diese z.B. mit den in Kapitel 13 vorgestellten Techniken zu „Geschäftsobjekten“ oder „Fachklassen“ (vgl. [Züllighoven98]) der Implementation nachverfolgt werden können. In Abschnitt 16.4 verfeinern wir dann die rein funktionalen, auf externe Interaktionen bezogenen black-box Testfälle zu sogenannten „grey-box“ Testfällen, die analog zu entwurfsbasierten Systemtests (vgl. [Hetzel88]) interne Interaktionen berücksichtigen und gezielt zur Erfüllung struktureller Testkriterien beitragen.

### 16.1 Klassen-Botschaftsdiagramm der Anforderungsspezifikation

Wesentlich für unser Vorgehen ist, dass die komplexen Operationen der Domänenklassen umfassend mit Episoden geprüft wurden (Kapitel 12) und die hierbei explizit gemachte systeminterne Information protokolliert ist. Wir betrachten nun die Gesamtheit der internen Interaktionen zwischen Domänenklassen. Im Fokus liegen also deren Operationen sowie die Aufrufe bzw. „Benutzungs“-Beziehungen zwischen ihnen.

Jeder Episodenschritt repräsentiert die „Ausführung“ einer bestimmten, in einer Domänenklasse deklarierten Operation. Die „Benutzungsbeziehungen“ zwischen den Operationen

sind in der Folgeschrittfunktion der Episoden protokolliert. Wie fassen diese Beziehungen in einem Graph zusammen, den wir das *Klassen-Botschaftsdiagramm für die Anforderungsspezifikation* ( $\text{KBD}_A$ ) nennen:

1. Jeder im Klassenmodell deklarierten Operation entspricht ein Knoten des Graphen.
2. Ist ein Episodenschritt  $s_2$  Folgeschritt eines Episodenschritts  $s_1$ , d.h.  $s_2 \in \sigma(s_1)$ , so fügen wir eine *Benutzungskante* zwischen den entsprechenden Knoten ein.
3. Überschreibt eine Operation eine andere Operation, so fügen wir eine *Vererbungskante* zwischen den entsprechenden Knoten ein. Zusätzlich duplizieren wir in diesem Fall alle Kanten, die in dem der überschriebenen Operation entsprechenden Knoten enden, für den der überschreibenden Operation entsprechenden Knoten. Solche duplizierten und „umgeleiteten“ Kanten stehen bzgl. der ursprünglichen Kante in einer Relation  $P$ , die explizit die Tatsache repräsentiert, dass die entsprechende „Benutzung“ an Objekte aller die Operation (re-) definierender Klassen gerichtet sein können.

Ein Algorithmus zur Generierung des  $\text{KBD}_A$  ist in Anhang A.1 angegeben. Für die Darstellung des  $\text{KBD}_A$  treffen wir folgende Vereinbarungen:

- Wir stellen Operationen durch ein Rechteck mit abgerundeten Kanten dar. Im Rechteck steht der Name der Operation, mit „:“ abgesetzt wird diesem der Name der Klasse, in der die Operation definiert ist, vorangestellt.
- Vererbungskanten werden wie in der UML durch einen geschlossenen, nicht ausgefüllten Pfeil (mit der Spitze zum Knoten der überschriebenen Operation) gezeichnet.
- Benutzungskanten stellen wir durch einen geschlossenen, ausgefüllten Pfeil von der benutzenden- zur benutzten Operation (mit der Spitze zur Zieloperation) dar.
- Gehen Benutzungskanten an redefinierte Operationen polymorph substituierbarer Zielklassen (Relation  $P$ ), so werden ihre Ausgangspunkte zusammengelegt.

*Beispiel 16.1.* Aus den in den Abbildungen 11.1, 12.2 und 12.3 dargestellten Episoden ermitteln wir (mit dem Algorithmus Generiere  $\text{KBDA}$ , s. Anhang A.1) die Benutzungsbeziehungen zwischen Operationen der Klassen **Karte**, **Transaktion**, **Auszahlung** und **Konto** (Abb. 16.1). Wir erkennen, dass die Operation **Transaktion::prüfe** in der Unterklasse **Auszahlung** redefiniert wird. Operation **führePlanAus**, welche in der Klasse **Transaktion** definiert und in der Klasse **Auszahlung** unverändert geerbt ist, benutzt (polymorph je nach der Klasse des ausführenden Objekts) die beiden Operationen **Transaktion::prüfe** und **Auszahlung::prüfe**.

□

## 16.2 Klassen-Botschaftsdiagramm der Implementation

Das  $\text{KBD}_A$  stellt in kompakter Art und Weise die Abhängigkeiten im Domänen-Klassenmodell zusammen. Wir streben analog dazu ein (möglichst einfaches) Test-Modell für die Imp-



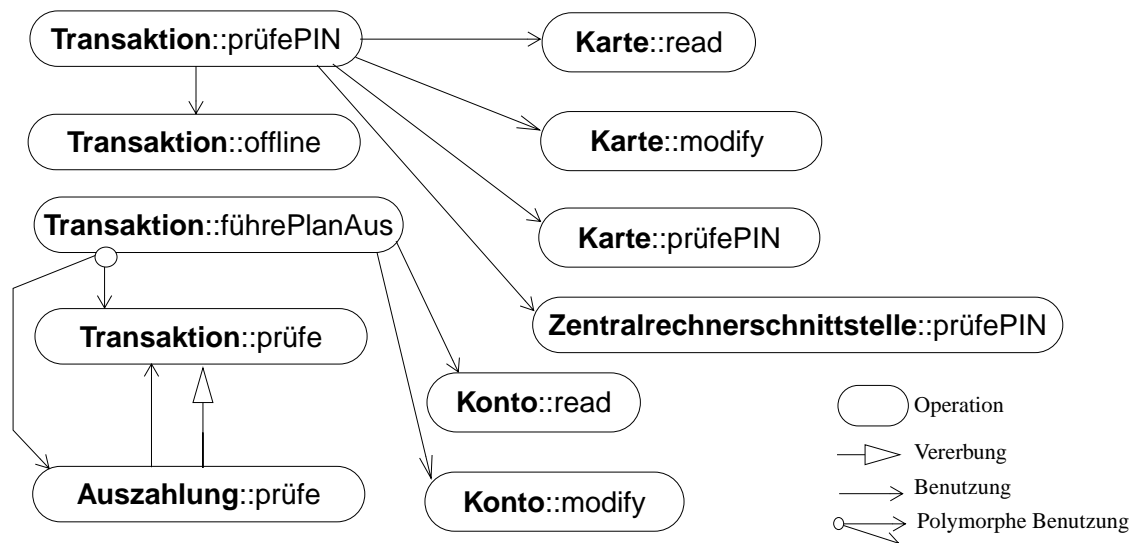


Abb. 16.1  $KBD_A$  für einige Operationen im (Domänen-) Klassenmodell des Bankautomaten

lementation an, welches alle möglichen Interaktionen zwischen Objekten der Implementation unter Beachtung der Vererbungsstruktur erfasst. Hierbei betrachten wir als Interaktionen den „Botschaftsfluss“ sowie mögliche Zustandsänderungen aufgrund einer Botschaft. Weitere Details bezüglich interner Interaktionen in objektorientierten Anwendungen sind in Anhang A.2 ausgeführt.

Im Folgenden skizzieren wir, wie aus dem Quellcode der Anwendung das *Klassen-Botschaftsdiagramm für die Implementation* ( $KBD_I$ ) abgeleitet werden kann, welches genau diesen Zweck erfüllt. In Anhang A.3 ist zusätzlich zur formalen Definition des  $KBD_I$  ein Algorithmus zu seiner Konstruktion aus den Java-Quellcodes der Anwendung angegeben.

Analog zu den im  $KBD_A$  erfassten Benutzungsbeziehungen gehen wir bei der Ableitung des  $KBD_I$  von folgenden Beobachtungen aus:

- Die Quelle einer Botschaft liegt syntaktisch eindeutig identifizierbar in einer (im Kontext eines Objekts ausgeführten) Methode, die wir *Quellmethode* der Botschaft nennen. Das den Kontext angegebende Objekt ist Instanz einer eindeutig bestimmten Klasse.
- Das *Zielobjekt* einer Botschaft ist ein Objekt, dessen Klasse bei ungetypten Sprachen nicht syntaktisch ermittelbar ist; bei getypten Sprachen kann sie eine von mehreren polymorph substituierbaren Klassen sein, die syntaktisch ermittelbar sind. „Versteht“ das Ziel die Botschaft, so entspricht dies der Ausführung einer anhand der Vererbungsstruktur zu ermittelnden Methode, der *Zielmethode* der Botschaft.

- Der aktuelle *Zustand* als Äquivalenzklassen von Werten der Klassen- und Instanzvariablen (vgl. [Binder96a]) bestimmt den Ablauf der Methoden, wobei diese Variablen innerhalb von Methoden gelesen und modifiziert werden.

Als Knoten des  $KBD_I$  kommen prinzipiell Klassen, Objekte, Methoden oder einzelne Anweisungen in Frage. Klassen scheiden als zu grobgranular aus, da Botschaften nur innerhalb von Methoden erzeugt werden. Objekte als Knoten würden das Modell auf die jeweils betrachtete Objektkonstellation einschränken und somit in einem Modell nicht alle möglichen Interaktionen erfassen. Würden wir den Knoten einzelne „Anweisungen“ zuordnen, so müssten wir zusätzlich eine dem Kontrollfluss innerhalb der Methoden entsprechende Kantenart zufügen, die jedoch keiner Botschaft entspräche. Dies würde der Forderung, dass wir uns auf die (durch Botschaften) möglichen Interaktionen zwischen Objekten in einer Anwendung konzentrieren wollen, zuwiderlaufen. Darüber hinaus wäre die Granularität des so entstehenden Graphen zu fein für den Test einer vollständigen Anwendung.

Wir wählen daher „member“ [GJS96] oder „Features“ [Meyer97] als die den Knoten des  $KBD_I$  entsprechende Elemente der Implementation aus, also — wie beim  $KBD_A$  — Methoden und zusätzlich auch Klassen- und Instanzvariablen.

Im Weiteren bezeichnen **A** und **B** Klassen. Falls Verwechslungen ausgeschlossen sind, sagen wir im Kontext des  $KBD$  einfach „Methodenknoten **A**::a“ anstatt „der die Methode a in Klasse **A** repräsentierende Knoten mit dem Namen <**A**::a>“.

Wir betrachten zunächst Aufrufe von Methoden (*direkte Aufruf-Interaktionen*). Eine einfache gerichtete *Botschaftskante* verbindet Methodenknoten **A**::a mit Methodenknoten **B**::b, wenn bei der Ausführung von a in einer Instanz der Klasse **A** eine Botschaft zu einem Objekt der Klasse **B** gesendet werden kann, welche die Ausführung der Methode b bewirkt. In anderen Worten benutzt Methode **A**::a Methode **B**::b.

Als nächstes beziehen wir die Vererbung, den Polymorphismus sowie die damit verbundenen Schlüsselworte bzw. Konstrukte **this** und **super** zur Steuerung der dynamischen Methodenbindung mit ein. Zunächst einmal fügen wir eine als *Vererbungskante* bezeichnete gerichtete, mit der Marke "inheritance" gekennzeichnete Kante von Methodenknoten **B**::b zu Methodenknoten **A**::b, wenn Klasse **B** Unterklasse von **A** ist, d.h. Methode **B**::b redefiniert Methode **A**::b.

Zusätzlich duplizieren wir in diesem Fall alle unmarkierten Botschaftskanten, die in Methodenknoten **A**::a enden, für den Methodenknoten **B**::b. Die duplizierten und „umgeleiteten“ Botschaftskanten stehen (analog zum  $KBD_A$ ) jeweils in einer Relation *P* zur ursprünglichen Kante. Die Relation *P* repräsentiert explizit die Tatsache, dass „Aufrufe“ der Methode b dynamisch an („polymorph substituierbare“) Objekte der Klassen **A** und **B** gebunden werden können.

Jedes Vorkommen eines Methodenaufrufs der Form **this** a in einer Methode x der Klasse **A** wird als mit der Marke "this" gekennzeichnete Botschaftskante von Methodenknoten x zu

dem Methodenknoten dargestellt, welcher der ersten Deklaration oder Redefinition der Methode  $a$  „aufwärts“ in der Vererbungshierarchie der Klasse  $\mathbf{A}$  (inklusive  $\mathbf{A}$ ) entspricht. In der gleichen Art und Weise wird jedes Vorkommen eines Methodenaufrufs der Form **super**  $a$  in einer Methode  $x$  der Klasse  $\mathbf{A}$  als mit der Marke "super" gekennzeichnete Botschaftskante von Methodenknoten  $x$  zu dem Methodenknoten dargestellt, welcher der ersten Deklaration oder Redefinition der Methode  $a$  „aufwärts“ in der Vererbungshierarchie der Klasse  $\mathbf{A}$  (exklusive  $\mathbf{A}$ ) entspricht.

Nun betrachten wir noch die (explizite) Erzeugung von Objekten in der Anwendung. Sei  $A(k_0, \dots, k_n)$  ein Konstruktor der Klasse  $\mathbf{A}$ . Jedes Vorkommen eines Methodenaufrufs der Form  $\text{new } A(k_0, \dots, k_n)$  in einer Methode  $x$  wird als eine mit der Marke "new" gekennzeichnete Botschaftskante von Methodenknoten  $x$  zu dem (Konstruktor-) Methodenknoten mit entsprechender Signatur in der Klasse  $\mathbf{A}$  dargestellt. Wir nennen solche Kanten auch Instanziierungskanten.

Da in Java (wie auch in Eiffel) im Gegensatz zu C++ die „explizite“ Zerstörung von Objekten nicht vorgesehen ist („Garbage Collection“, s. z.B. [GJS96][Meyer97]), gibt es im  $\text{KBD}_I$  bei einer Implementation in Java keine mit der Marke "destroy" gekennzeichneten Kanten. Die explizite Entfernung persistenter Objekte (z.B. aus einer Datenbank) spiegelt sich in der Implementation in Methoden wider, die zurück zu entsprechenden destroy-Standardoperationen im Domänen-Klassenmodell der Anforderungsspezifikation verfolgbar sind.

Zusätzlich berücksichtigen wir auch noch Klassen- bzw. Instanz-Variablen, also die Zustandsspeicher (*direkte Daten-Interaktionen*). Für jede Variable fügen wir dem  $\text{KBD}_I$  einen mit dem qualifizierten Namen der Variablen gekennzeichneten Variablenknoten hinzu. Zugriffe auf die Variable werden dann auf speziell markierte Botschaftskanten abgebildet:

- Eine mit der Marke "uses" gekennzeichnete Kante vom Methodenknoten  $a$  zum Variablenknoten  $x$  bedeutet, dass während der Ausführung von Methode  $a$  nicht-modifizierend auf Variable  $x$  zugegriffen werden kann.
- Analog bedeutet eine Kante mit der Marke "def" vom Variablenknoten  $x$  zum Methodenknoten  $a$ , dass bei der Ausführung von Methode  $a$  die Variable  $x$  modifiziert werden kann. Anders herum ausgedrückt „benutzt“ Variable  $x$  die Methode  $a$  zu ihrer Definition.

Für die Darstellung des  $\text{KBD}_I$  treffen wir folgende, zusätzlich zur Darstellung des  $\text{KBD}_A$  geltende Vereinbarungen:

- Knoten, die Instanz- und Klassen-Variablen zugeordnet sind, zeichnen wir als Parallelogramm, in das wir den qualifizierten Namen der Variablen schreiben.
- Mit „def“ und „uses“ markierte Kanten stellen wir wie Botschaftskanten dar, also durch einen offenen Pfeil.

Wir betrachten zur Illustration ein einfaches Beispiel.

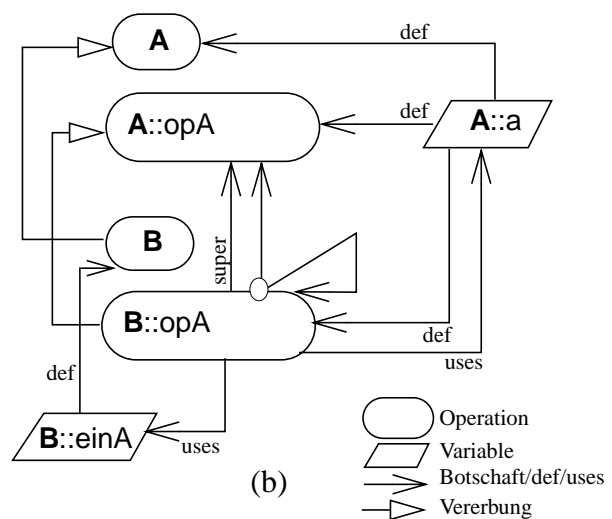
*Beispiel 16.2.* Abb. 16.2 (a) zeigt ein Java Code Fragment mit zwei Klassen **A** und **B**, wobei **B** direkte Unterklasse von **A** ist. Das KBD<sub>I</sub> für die Klassen **A** und **B** zeigt Abb. 16.2 (b). Zunächst sehen wir vier Methodenknoten anstelle von zwei, da die „Default-Konstruktoren“ für beide Klassen berücksichtigt sind (vgl. [GJS96]). Eine mit „def“ markierte Kante verbindet die Variablenknoten mit den jeweiligen Konstruktorknoten.

Mit der Vererbungskante wird die Redefinition der Methode `opA` in der Klasse **B** berücksichtigt. Die Botschaftskante von `B::opA` zu Methode `opA` in Klasse **A** reflektiert den Gebrauch des Schlüsselworts **super** in `B::opA` und die mögliche (polymorphen) Bindungen der Botschaftsquelle `einA.opA`. Die Anweisung `einA.opA` in der Methode `B::opA` führt dementsprechend auch zur selbst-rekursiven Botschaftskante mit dem Ziel `B::opA`.

Die mit „def“ markierten Kanten vom Variablenknoten `A::a` zu den drei Methodenknoten **A**, `A::opA` und `B::opA` zeigen die (mögliche) Modifikation der Instanzvariablen `a` und damit direkte Dateninteraktionen an. Die mit „uses“ markierten, von der Methode `B::opA` ausgehenden Kanten bedeuten die mögliche, nicht-modifizierende Verwendung der Instanzvariablen `einA` und `a` in dieser Methode.

```
class A {
  public integer a;
  public void opA {
    a = 1;
  }
}
class B extends A {
  public A einA;
  public void opA {
    super.opA;
    a = a + 1;
    einA.opA
  }
}
```

(a)



(b)

Abb. 16.2 Java Code (a) und Klassen-Botschaftsdiagramm für die Implementation (b)

Wir erkennen die direkten Aufruf-Interaktionen anhand der Kanten des KBD<sub>I</sub>. Zustandsbedingte Abhängigkeiten zwischen den Methoden (indirekte Interaktionen) spiegeln sich im KBD<sub>I</sub> in Pfaden von einem Methodenknoten zu einem (nicht notwendigerweise verschiede-

nen) Methodenknoten wider, die mindestens eine<sup>1</sup> mit def oder mit uses markierte Kante beinhalten (s. Anhang A.2).

□

## 16.3 Verfolgbarkeit und Klassen-Botschaftsdiagramme

Über die Verfolgbarkeitsrelation können wir das aus der Anforderungsspezifikation generierte  $KBD_A$  auf einen Teil des aus der Implementation generierten  $KBD_I$  abbilden — diesen Teil bezeichnen wir mit  $KBD_{I \rightarrow A}$ . Hierbei entsprechen z.B. jeder Benutzung der Standardoperation modify im  $KBD_A$  eine oder mehrere mit „def“ markierte Kanten des  $KBD_I$ . Benutzungen der Standardoperation x.query im  $KBD_A$  entsprechen im  $KBD_I$  Kanten zu Methoden aus den Implementationsklassen, welche die Beziehung x des Klassendiagramms der Anforderungsspezifikation realisieren. Die Standardoperation query spiegelt sich in mindestens einer mit "new" markierten Kante des  $KBD_I$  wider; die Standardoperation destroy in entsprechenden Methodenknoten des  $KBD_I$ .

Ausgehend von den direkt vom  $KBD_A$  erreichbaren Knoten des  $KBD_I$  erreichen wir transitiv weitere Knoten des  $KBD_I$ ; den resultierenden Teil des  $KBD_I$  nennen wir  $KBD_{I \rightarrow TA}$ . Es verbleibt ggf. ein nicht-leerer Teil-Graph  $KBD_I \setminus KBD_{I \rightarrow TA}$ . Dieser Sachverhalt ist in Abb. 16.3 dargestellt.

Ist  $KBD_I \setminus KBD_{I \rightarrow TA} \neq \emptyset$ , so können verschiedene Umstände vorliegen, von denen wir die wichtigsten kurz erläutern und Maßnahmen zu ihrer Behandlung im grey-box Test angeben:

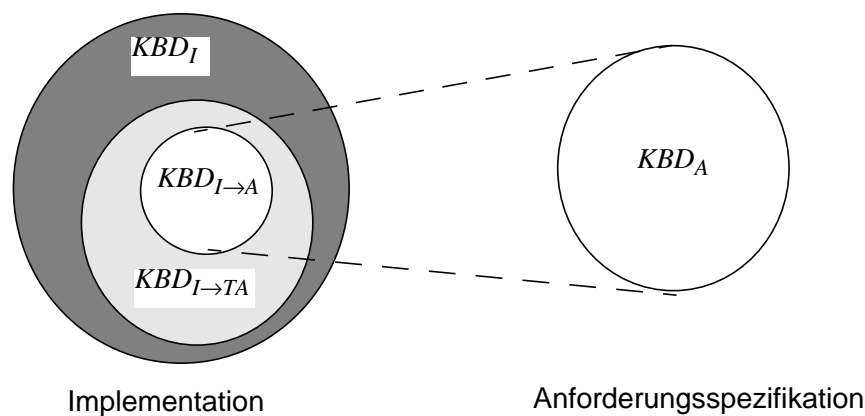


Abb. 16.3  $KBD_I$  vs.  $KBD_A$

<sup>1</sup> Da def- bzw. uses- Kanten immer vom Variablen- zum Methodenknoten bzw. umgekehrt gerichtet sind, enthalten die indirekten Interaktionen entsprechenden Pfade immer gleichviele, jeweils aufeinanderfolgende def- und uses- Kanten.

- ❑ Es kann eine Botschaft erzeugt werden, für die keine zugehörige Operation in den bei der Generierung des  $KBD_I$  berücksichtigten Klassen gefunden wird. In diesem, bei Bibliotheksklassen häufig eintretenden Fall ist die betrachtete Anwendung nicht abgeschlossen. Den Abschluss erreicht man entweder durch Hinzufügen geeigneter weiterer (Bibliotheks-) Klassen oder aber neuer Klassen (bzw. Stellvertreter), welche die betroffenen Methoden definieren.
- ❑ Es existieren Operationen, für die innerhalb des Systems keine entsprechenden Botschaften erzeugt werden. Einerseits kann dieser Fall auftreten, wenn die Anwendung nur bestimmte Aspekte von Bibliotheksklassen verwendet (Meyer's „Shopping List“ Ansatz [Meyer97]). Auf der anderen Seite könnten entsprechende, die Operation aufrufende Bibliotheksklassen bei der Generierung des  $KBD_I$  nicht berücksichtigt sein. Einige Operationen der Anwendung werden z.B. als „Callback“-Operationen von nicht betrachteten Klassen verwendeter (GUI-) Frameworks eingesetzt. In diesem Fall können wir diese Operationen entweder vernachlässigen oder aber die entsprechenden Klassen des Frameworks in unsere Betrachtungen mit einbeziehen. Es ist also zu prüfen, ob wir weitere Klassen bei der Generierung des  $KBD_I$  berücksichtigen müssen, oder ob diese Methoden Wurzeloperationen ggf. neu zu definierender Use Case bzw. Test-Szenarioschritte sind.
- ❑ Es existieren Botschaftskanten im  $KBD_I$ , für die beim Ablauf der Anwendung keine entsprechenden Aufrufe erzeugt werden können. Dies ist z.B. der Fall, wenn in der Anwendung kein Objekt der betroffenen Klasse erzeugt wird oder die Botschaftsquelle keine entsprechende Referenz annehmen kann. Dieser Fall kann nur in ungetypten Sprachen auftreten und z.B. mit Algorithmen zur Typinferenz weitgehend statisch eliminiert werden (vgl. [PalSch94]). Oft kann der Tester in diesem Fall sofort erkennen, welche Interaktionen nicht ausführbar sind, und diese nach Ausführung aller Tests manuell eliminieren.
- ❑ Für Unterklassen im Klassenmodell der Anforderungsspezifikation wurden keine eigenen Test-Szenarien bzw. Episoden angegeben. Mit dem in [HMGF92] vorgestellten Algorithmus können wir die notwendigen Erweiterungen der Test-Szenarien berechnen bzw. zumindest angeben, für welche Unterklassen neue Episoden durchgespielt und im grey-box Test verwendet werden müssen.

## 16.4 Testfallerweiterung

Im SCORES grey-box Test erweitern wir die Testfälle des black-box Tests. Hierfür benutzen wir die bei der Simulation der Episoden „operationalisierten“ Spezifikationen der Wurzeloperationen als Testorakel zur Spezifikation des „erwarteten Ablaufs“ sowie der erwarteten Änderungen an der Objektkonstellation:

- ❑ Für jeden Testfall des black-box Tests ermitteln wir aus den Episoden der Schritte des entsprechenden Test-Szenarios die „berührten“ Elemente des Domänenklassendiagramms.
- ❑ Über die Verfolgbarkeitsrelation ermitteln wir alle Elemente der Implementation, welche zurück zu den „berührten“ Elementen verfolgbar sind.
- ❑ Die von der Episode vorgegebene Reihenfolge übertragen wir auf die Elemente der Implementation und erhalten die bei der Ausführung der Testfälle in der Anwendung erwarteten Abläufe.
- ❑ Objekte, die in der Anwendung vor dem Test bereits existieren müssen, leiten wir aus dem Klassenbereich des dem Test-Szenario Schritt zugeordneten Use Case Schritts ab. Aus den Signaturen der Wurzeloperationen und der von den Episodenschritten ausgeführten Operationen bestimmen wir (als Testorakel) die innerhalb eines Test-Szenarios modifizierten bzw. erzeugten Instanzen von Klassen, die von den Domänenklassen aus verfolgbar sind.

Natürlich können wir nicht erwarten, die so spezifizierten erwarteten Abläufe tatsächlich eins-zu-eins bei der Ausführung wiederzufinden. Nach den Ausführungen im vorigen Abschnitt finden zusätzliche interne Interaktionen statt, die implementationstechnische Details verkörpern. Allerdings sollten die spezifizierten Abläufe von den sich ergebenden Abläufen überdeckt werden bzw. sich in diese einbetten lassen.

## 16.5 Methodisches Vorgehen

Wir gehen beim Scores grey-box Test nach dem Algorithmus Grey-box Test (Abb. 16.4) vor. Die Test-Reihenfolge ermitteln wir wie im black-box Test über die invertierte topologische Sortierung der als azyklischen Graph interpretierten Inter-Use Case-Referenzfunktion *REF*. Dann iterieren wir in dieser Reihenfolge über alle Use Cases und alle Test-Szenarien jedes Use Case (Zeilen 5-15). Zunächst ermitteln wir in den Zeilen 5-10 die „Einstiegspunkte“, also die über die Verfolgbarkeit den Operationen der Episoden direkt zugeordneten Elemente der Implementation und danach (über die transitive Hülle des  $KBD_I$ ) alle von diesen Elementen erreichbaren Elemente der Implementation. Diese Teile der Implementation werden instrumentiert (Zeile 10).

**Algorithmus** Grey-box Test

```

    Eingabe: Anforderungsspezifikation, Implementation;
    Ausgabe: Test-Überdeckungs- und Ausführungsprotokolle;
1   Test-Reihenfolge tre generieren;
2   KBDI aus der Implementation generieren;
3   FORALL uc ∈ UC ORDERED BY tre DO
4       FORALL ts ∈ uc.Test-Szenarien DO {
5           kbdEP := ∅ ;
           /* Über die (Nach-) Verfolgbarkeit „Einstiegspunkte“ in das KBDI ermitteln */
6       FORALL ep ∈ ts.allEpisode DO
7           kbdEP := kbdEP ∪ ep.rootOperation.allImplementationElements ;
           /* Über transitive Hülle des KBDI berührte Elemente der Implementation ermitteln */
8       kbdEP := (KBDI |*kbdEP);
           Berührte Elemente der Implementation instrumentieren;
9       REPEAT
10          Objektkonstellation aufbauen;
11          Daten für die Interaktionsparameter der Schritte von ts generieren;
12          Test-Skript ausführen, hierbei die Wurzeloperationen der Test-
13             Szenarioschritte für die entsprechenden Wurzelobjekte „aufrufen“;
14          UNTIL Test-Endekriterium erreicht OR Test-Ressourcen für Test-Szenario ausgeschöpft;
15      }; /* FORALL */
16   Testfälle für die noch nicht überdeckten Teile des KBDI ermitteln und ausführen;
END Grey-boxTest.

```

Abb. 16.4 Algorithmus Grey-box Test

In der Schleife ab Zeile 11 führen wir dann die Testfälle aus, wobei wir zunächst eine für den Start des Test-Szenarios zulässige (positiver Test) oder nicht zulässige (negativer Test) Objektkonstellation aufbauen (Zeile 12). Danach werden Interaktionsdaten generiert (Zeile 13) und dann die den Schritten des Test-Szenarios zugeordneten Wurzeloperationen ausgeführt bzw. aufgerufen (Zeile 14). Dies erfolgt solange, bis das Test-Endekriterium erfüllt ist (z.B. „Berührte Elemente der Implementation überdeckt“) oder die Test-Ressourcen für das Test-Szenario ausgeschöpft sind (Zeile 15). Aus dem KBD<sub>I</sub> abgeleitete Test-Endekriterien bezüglich der Interaktionen in objektorientierten Anwendungen geben wir in Kapitel 18 an.

In Zeile 16 versuchen wir, falls noch Ressourcen verfügbar sind und es technisch<sup>1</sup> möglich ist, die bisher noch nicht überdeckten Teile der Anwendung auszuführen. Ist nach der Durchführung der so ausgeführten grey-box Tests die geforderte strukturelle Überdeckung noch nicht erreicht (Zeile 16), so ermitteln wir unter Benutzung der Verfolgbarkeitsrelation (Abschnitt 13.2) weitere Testfälle nach der folgenden Vorgehensweise.

1 Nicht überdeckte Teile der Implementation können z.B. technische Ausnahmebehandlungen, aber auch in der Anwendung tatsächlich nicht verwendete Operationen z.B. von wiederverwendeten Klassen sein.



Zunächst berechnen wir über die (invers betrachtete) Verfolgbarkeitsrelation alle Elemente des Domänen-Klassenmodells der Anforderungsspezifikation, die sich vorwärts zu noch nicht überdeckten Elementen der Implementation  $\mathbf{m}_I$  verfolgen lassen. Bezeichnen wir mit  $I$  die noch nicht ausreichend überdeckten Elemente der Implementation und mit  $I_A$  die entsprechenden Elemente des Domänen-Klassenmodells  $\mathbf{m}_A$ , so ermitteln wir  $I_A$  zu

$$I_A = \{x \in \mathbf{m}_A \mid \exists y \in I: \text{refinedBy}(x,y)\}$$

Zu diesen Elementen des Domänen-Klassenmodells ermitteln wir über die Kopplung von Use Cases und dem Klassenmodell der SCORES-Anforderungsspezifikation alle Use Cases, welche noch nicht überdeckte Elemente der Implementation „berühren“.

Mit den Techniken aus Teil III spezifizieren wir für diese Use Cases ggf. neue Test-Szenarien und leiten weitere Testfälle ab. Hierbei konzentrieren wir uns auf solche Interaktionsschritte, zu deren Wurzeloperationen sich nicht überdeckte Elemente der Implementation (ggf. transitiv über das  $\text{KBD}_I$ ) zurück verfolgen lassen. Mit den so abgeleiteten Testfälle verbessern wir gezielt die strukturelle Überdeckung.

Können auch mit dieser Vorgehensweise einige Elemente der Implementation nicht überdeckt werden, so müssen für diese Elemente white-box Tests abgeleitet werden ([Beizer90] [Riedemann97]). Bezüglich dieses Falles erinnern wir auch an die in Abschnitt 16.2 genannten Möglichkeiten nicht-verfolgbarer Elemente der Implementation.

Abb. 16.5 verdeutlicht die Vorgehensweise: Über die Verfolgungsrelation (1) ermitteln wir alle Elemente der Anforderungsspezifikation, die nicht-ausreichend überdeckten Elementen der Implementation entsprechen (2). Für alle Use Cases, die solche Elemente berühren, spezifizieren wir ggf. neue Test-Szenarien und leiten weitere Testfälle ab. Hierbei berücksichtigen wir über (3) alle Interaktionsschritte, deren Wurzeloperationen sich ggf. transitiv über das  $\text{KBD}_I$  zu nicht-ausreichend überdeckten Elementen der Implementation verfolgen lassen

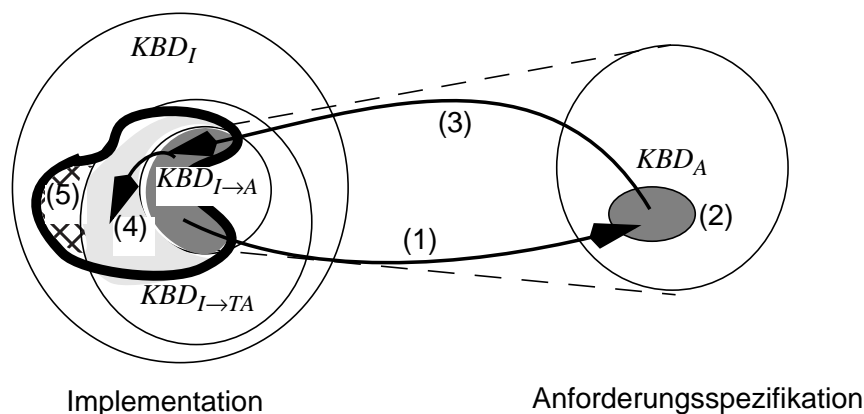


Abb. 16.5 SCORES grey-box Testfallgenerierung mit dem KBD

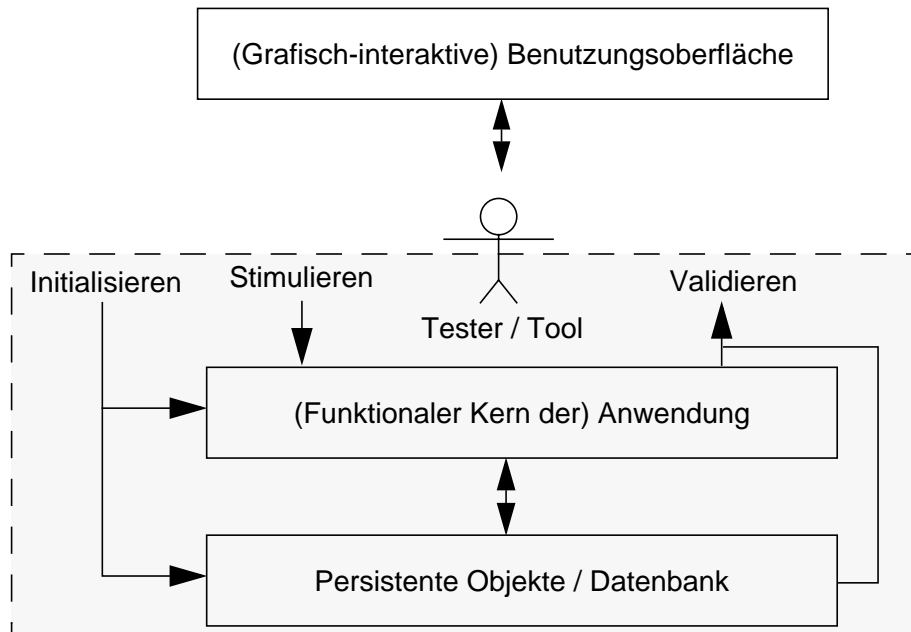


Abb. 16.6 SCORES Grey-box Test

(4). Zum Schluss werden dann noch die so nicht erreichten Elemente der Implementation mit strukturellen Klassentests geprüft (5).

Abb. 16.6 verdeutlicht die technischen Voraussetzungen für den grey-box Test: Gegebenenfalls unter Umgehung der Benutzungsoberfläche wird der funktionale Kern der Anwendung nach der Initialisierung der persistenten Objekte und der Anwendung selbst mit den Wurzelbotschaften der Interaktionsschritte des Test-Szenarios stimuliert und die Ergebnisse anhand der tatsächlichen Abläufe in der Anwendung und der sich ergebenden Objektkonstellation geprüft.

*Beispiel 16.3.* Das in Abb. 15.6 gezeigte Testskript zum Test-Szenario Anmeldung OK des Use Cases Anmelden wird für den grey-box Test erweitert. Hierbei sind zunächst lediglich die in den Episoden simulierten Abläufe in den Test-Orakeln zu spezifizieren. Bei der Testausführung bzw. der Testauswertung wird dann geprüft, ob diese Abläufe sich in die tatsächlich gemessenen Abläufe einbetten lassen.

```

TESTCASE AnmeldungOKGreyBox USECASE Anmelden (
    Bankleitzahl: String[8],
    Kontonummer: String[10],
    Gültigkeitsdatum: Date,
    Kartennummer: Integer;
    PIN: Integer)
// Der BankKunde identifiziert sich durch Eingabe der Karte und der PIN.
PRECONDITION
    AccountCreatedOrExists (Bankleitzahl, Kontonummer) AND
    CardNotLocked (Bankleitzahl, Kontonummer, Gültigkeitsdatum, Kartennummer) AND
    ATMReadyAndOnline;
BEGIN
// Karte Eingeben
    KarteEinführen (Bankleitzahl, Kontonummer, Gültigkeitsdatum, Kartennummer)
    ORACLE Kartenleser.state.asString = "locked", (Kartenleser::leseKarte - Karte::new);
// PIN Anfordern und prüfen
    Transaktion.prüfePIN (PINAnfordern (PIN))
    ORACLE Kartenleser.state.asString = "locked" AND PIN_OK, (Bedienpult::lesePIN -
        (Transaktion::prüfePIN - ZentralrechnerSchnittstelle::prüfePIN - Karte::modify));
// Auswahl anzeigen
    Bedienpult.MenüAnzeigen()
END.

TESTCASE KarteEinführen ROOTCLASS Kartenleser (
    Bankleitzahl : String[8],
    Kontonummer : String[10],
    Gültigkeitsdatum : Date,
    Kartennummer : Integer)
LOCAL
    theCard : Karte;
BEGIN
// Der BankKunde führt die Karte in den Kartenleser ein
    Kartenleser.erwarteKarte();
    theCard := Kartenleser.leseKarte() ORACLE Kartenleser.state.asString = "locked",
    theCard.isEqual(Karte(Bankleitzahl, Kontonummer, Gültigkeitsdatum , Kartennummer))
END

TESTCASE PINAnfordern ROOTCLASS Bedienpult (PIN: Integer)
LOCAL
    thePIN : Integer;
BEGIN
// Der BankKunde gibt die PIN ein
    thePIN := Bedienpult.lesePIN() ORACLE thePIN.isEqual(PIN);
END

```

Abb. 16.7 Grey-box Testfälle



# Kapitel 17

## Umgebungsaufbau und Testdatengenerierung

*Preparation of test data is a major component of the total cost of system testing and acceptance testing*  
[CGL81]

Während die Ableitung von Testfällen und Testdaten bezüglich primitiver Typen als Interaktionsparameter (z.B. integer, real, string) in der „klassischen“ Testliteratur ausführlich beleuchtet ist (vgl. [Beizer90] [Beizer95] [Myers79] [Riedemann97]), wurde die Generierung von Objekten und Objektkonstellationen bisher nur in wenigen speziellen Anwendungsgebieten untersucht (z.B. [Kösters97]). Wir gehen daher nach unserer eher skizzenhaften Betrachtung der Interaktions-Testfälle (Abschnitt 15.2) in diesem Kapitel ausführlicher auf dieses Problem ein.

### 17.1 Generierung von Objektkonstellationen

Bei der Validierung und Verifikation der Anforderungsspezifikation haben wir bei den Walk-Throughs der Test-Szenarien und Simulationen der Episoden konkrete Objektkonstellationen zugrundegelegt. In den Episoden sind mit den Sichtbarkeitsbereichen und den „Aufrufen“ allerdings implizit Beschränkungen auf Objektkonstellationen getroffen worden, in denen die jeweilige Episode „ablaufen“ kann. Diese Beschränkungen ergeben sich aus den Bedingungen der Schritte und Kanten im Use Case Schrittgraph, den Klassenbereichen der Schritte in den Test-Szenarien sowie aus Assoziationen und ihren Multiplizitäten des Klassenmodells. Wir wollen nun diese Beschränkungen dazu verwenden, aus den manuell bei der Qualitätssicherung für die Anforderungsspezifikation erstellten Objektkonstellationen weitere Objektkonstellationen für die Tests der Anwendung zu generieren.

Die grundlegenden Bausteine sind hierbei *abstrakte Objektstrukturen* (s. [Rogotzki97]). Eine abstrakte Objektstruktur schränkt innerhalb eines Teilmodells die Attributwerte der Klassen

durch Wertebereiche und ihre Objektbeziehungen durch Schablonen so ein, dass diese eine Menge semantisch sinnvoller Objekte definieren. Mit Hilfe eines Generators werden nach Vorgabe der abstrakten Objektstruktur dann *konkrete Objektstrukturen* in beliebiger Anzahl erzeugt, wobei die Attributwerte jedes einzelnen Objekts zufällig aus den vordefinierten Wertebereichen ausgewählt werden. Zwei so erzeugte konkrete Objektstrukturen sind nur strukturell isomorph, da die entsprechenden Objekte sich durch ihre Attributwerte unterscheiden.

Weiterhin können zwei Mengen konkreter Objektstrukturen — falls das zugehörige Klassenmodell derartige Beziehungen vorsieht — zu einer einzigen Objektstruktur verknüpft werden, wobei der Generator die jeweils betroffenen Objekte eingeschränkt durch eine Beziehungsschablone und ggf. den Beziehungen zugeordnete Constraints auswählt.

Weitere Objektkonstellationen werden mit abstrakten Objektstrukturen generiert, indem wir die bei der Validierung (manuell) erzeugten Objektkonstellationen variieren und nachfolgende „Konsistenz-Checks“ gegen das Klassenmodell ausführen.

*Beispiel 17.1.* Das in Abb. 17.1 links dargestellte Klassenmodell erlaubt die Generierung von Objektkonstellationen aus jeweils zwei über die Beziehung *verheiratetMit* verbundenen Objekten der Klasse *Person*. In einer Objektkonstellation hat jede *Person* (aus Terminierungsgründen) „keine“ oder aber genau zwei Eltern. Zu der manuell vorgegebenen konkreten Objektstruktur im gestrichelt gezeichneten Rahmen in Abb. 17.1 rechts können nach den

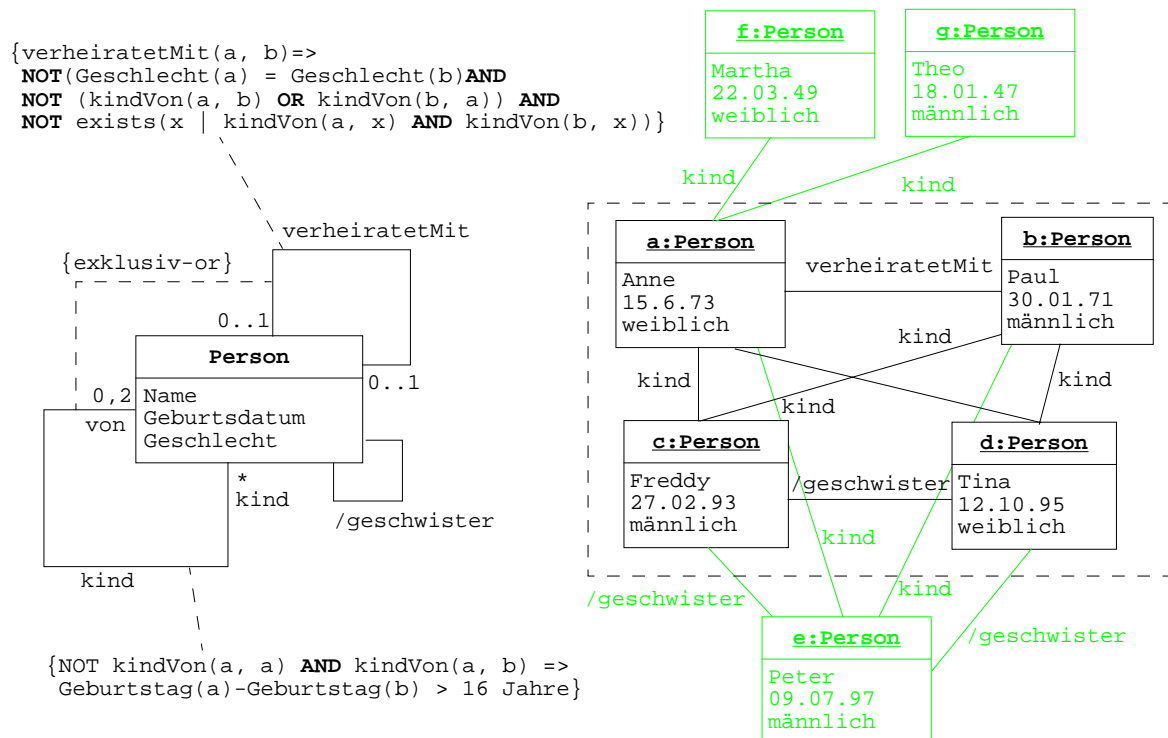


Abb. 17.1 Beispiel zur Generierung von Objektkonstellationen

Constraints der Beziehungen z.B weitere Kinder oder aber Eltern der miteinander verheirateten Personen Anne und Paul generiert werden. Die zusätzlich generierten Objekte bzw. Links sind in Abb. 17.1 rechts grau gezeichnet.



## 17.2 Komplexität

Unsere ersten Ansätze zur Erzeugung semantisch sinnvoller Objektkonstellationen aus Klassenmodellen mit abstrakten Objektstrukturen und Constraints erwiesen sich zunächst als vielversprechend (s. [Rogotzki97]). Jedoch konnten für einige „einfach“ erscheinende Klassenmodelle ohne vom Analytiker oder Tester manuell vorgegebene „Muster“-Objektstrukturen keine Objektkonstellationen generiert werden. Wie sich herausstellte, ist der Grund dafür die inhärente Komplexität des Problems, der wir den Rest dieses Abschnitts widmen. Wir formulieren zunächst das Problem *Generierung von Objektkonstellationen (GOK)* präziser:

*Gegeben* sei ein (UML) Klassenmodell  $KM$  mit einer (Teil-) Menge  $\mathbf{K}$  von Klassen sowie für jede Klasse  $\mathbf{k}$  aus  $\mathbf{K}$  eine Multiplizität  $c_{\mathbf{k}}$  (Anzahl der zu generierenden Instanzen der Klasse  $\mathbf{k}$ ). *Gesucht* ist eine zu  $KM$  konsistente<sup>1</sup> Objektkonstellation.

Die Komplexität von *GOK* ergibt sich aus dem folgenden Satz.

**Satz 17.1** *GOK* ist NP-hart<sup>2</sup>.

**Beweis** (Skizze) Wir wählen als NP-vollständiges Problem das Erfüllbarkeitsproblem für aussagenlogische Formeln in konjunktiver Normalform (*satisfiability*, *SAT* vgl. [GarJoh79]). Zur Reduktion von *SAT* auf *GOK* transformieren Instanzen von *SAT* in drei Schritten auf Instanzen von *GOK*.

1. Zunächst erzeugen wir für jede Aussagenvariable  $a$  zwei Klassen: eine die Variable selbst darstellende Klasse  $\mathbf{Va}$  mit der Kardinalitätsbeschränkung 0..1 — von jeder Klasse darf also höchstens eine Instanz generiert werden — und eine den Wert der Variablen symbolisierende abstrakte<sup>3</sup> Klasse  $\mathbf{Wa}$ , zu deren Objekten (genauer: zu Objekten konkreter Unterklassen) wir das singuläre Objekt der Klasse  $\mathbf{Va}$  in eine 1-1-Beziehung setzen. Wir nennen diese beiden Klassen die *VW-Klassen* zur Aussagenvariable  $a$ . Die Kardinalitätsbeschränkung der Klasse  $\mathbf{Va}$  spiegelt sozusagen die Identität bzw. „Einmaligkeit“ der Aussagenvariable  $a$  wider, die Klasse  $\mathbf{Wa}$  generalisiert die

1 Wir nennen eine Objektkonstellation konsistent zu einem Klassenmodell, wenn sie keine Instanzen abstrakter Klassen enthält und keine der angegebenen Multiplizitäten für Klassen und Beziehungen verletzt.

2 Ein Problem  $p \in \text{NP}$  wird „NP-hart“ genannt, wenn sich irgendein NP-vollständiges Problem auf  $p$  reduzieren läßt (vgl. [GarJoh79]).

3 Die Namen abstrakter Klassen sind kursiv gesetzt.

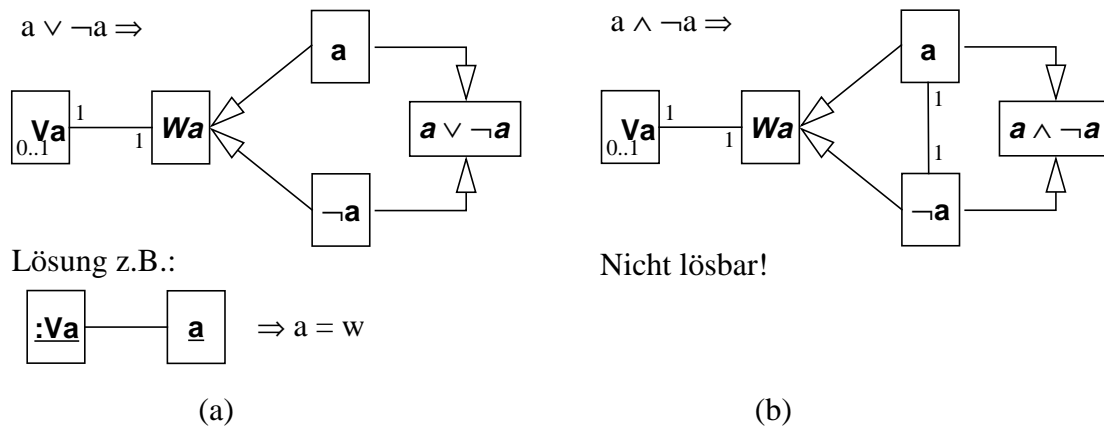


Abb. 17.2 Einfache Beispiele zur polynomialen Transformation von SAT auf GOK

- möglichen Belegungen der Variablen. Durch die 1-1 Beziehung wird sichergestellt, dass die Variable nicht gleichzeitig mit verschiedenen Werten belegt wird (Belegung mit wahr ( $w$ ) und falsch ( $f$ ), „das ausgeschlossene Dritte“).
2. Jedes Literal der Form  $a$  bilden wir auf eine Klasse  $a$  ab, die Unterklasse der Klasse  $Wa$  ist. Dementsprechend wird jedes Literal der Form  $\neg a$  auf eine Klasse  $\neg a$  abgebildet, die wiederum Unterklasse der Klasse  $Wa$  ist. Diese beiden Klassen nennen wir auch *L-Klassen* des entsprechenden Literals, sie entsprechen der Belegung der Aussagenvariablen  $a$  mit  $w$  ( $a$  instanziiert) oder  $f$  ( $\neg a$  instanziiert).
  3. Jeden zusammengesetzten Term bilden wir auf eine abstrakte Klasse (*T-Klasse*) ab, von der die T- bzw. L-Klassen der Teilterme erben. Bei einer Konjunktion fügen wir im Klassenmodell zusätzlich zwischen allen den Teiltermen der Konjunktion entsprechenden T- oder L-Klassen eine ungerichtete 1-1 Beziehung ein. Diese Beziehungen werden zu den L-Klassen vererbt und stellen die wesentlichen Beschränkungen aufgrund der aussagenlogischen Verknüpfungen dar. Zwei einfache Beispiele zur Transformation zeigt Abb. 17.2 (a, b). Ein etwas komplexeres Beispiel stellt Abb. 17.3 vor.

Wir konstruieren aus einer Lösung des *GOK* eine Lösung der entsprechenden Instanz von *SAT*, indem wir für jedes Objekt der generierten Objektkonstellation den Wert des entsprechenden Literals zu  $w$  und den Wert aller anderen Literale zu  $f$  setzen. Für jedes in  $\alpha$  vorkommende Literal-Paar der Form  $(\neg u, u)$  grenzen 1-1 Beziehungen zwischen den entsprechenden V- und L-Klassen gerade solche Lösungen des *GOK* aus, deren zugeordnete Belegungen für beide Literale —  $\neg u$  und  $u$  — den gleichen Wahrheitswert ergeben, also der Variablen  $u$  sowohl  $w$  als auch  $f$  zuordnen. Umgekehrt ergibt jede Lösung der Instanz von *SAT* eine Lösung des *GOK*, indem wir für jedes mit *true* belegte Literal die zugeordnete Klasse instanzieren und dann die entsprechenden Verbindungen setzen. Der vollständige Beweis findet sich in [Winter99].

□

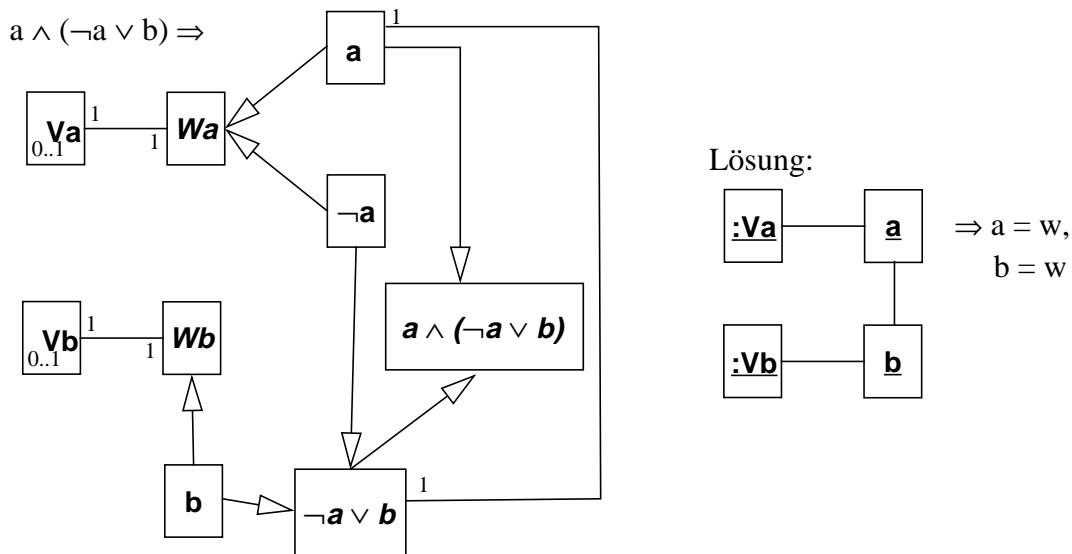


Abb. 17.3 Komplexeres Beispiel zur polynomialen Transformation von SAT auf GOK

## 17.3 Interaktionsdaten

Zur Generierung von (atomaren) Interaktionsdaten existieren in der Literatur eine große Anzahl von Verfahren (s. z.B. [Beizer90][Beizer95][Myers79][Riedemann97][Poston96]). Wir reissen die erforderlichen Schritte also nur kurz an.

- ❑ Im Rahmen der black-box und grey-box Tests erzeugen wir die Interaktionsdaten z.B. mit der Äquivalenzklassenbildung, der Grenzwertanalyse und ähnlichen funktionalen Testverfahren.
- ❑ Für Abnahmetests verwenden wir vorzugsweise (anonymisierte) Produktionsdaten, die durch den Benutzer bereitzustellen sind (vgl. [Hetzl88]). Sind diese nicht vorhanden, so müssen wir in Absprache mit dem Benutzer Parameter zur Generierung der Interaktionsdaten festlegen.

*Beispiel 17.2.* Einen (ausführbaren) Test mit konkreten Interaktionsdaten zum Test-Szenario bzw. -Skript Anmeldung OK des Bankautomat-Beispiels zeigt Abb. 17.4.

```

TEST AnmeldungOK (
    Bankleitzahl = 33050000,
    Kontonummer: 4712042,
    Gültigkeitsdatum: 01.01.2000,
    Kartennummer: 1234;
    PIN: 4656)
  
```

Abb. 17.4 Interaktionsdaten zum Test-Skript Anmeldung OK





# Kapitel 18

## Endekriterien, Ausführung und Auswertung

*One of the most difficult problems in testing is knowing when to stop.*  
[Myers79]

Für die Bewertung einer durchgeführten Prüfung benötigen wir sogenannte Testendekriterien. Wir erwarten von solchen Kriterien quantitative Aussagen über das Risiko, eine geprüfte Anwendung tatsächlich einzusetzen. Im Prinzip sollte ein solches Kriterium also Aussagen der Art „Testende erreicht/nicht erreicht“ ermöglichen. Zusätzlich zu verschiedenen Endekriterien für den Test der Anwendung gegen die objektorientierte Anforderungsspezifikation geben wir in diesem Kapitel noch einige Hinweise zur Ausführung und Auswertung der Tests.

### 18.1 Test- und Testendekriterien

Weyucker stellt eine Reihe von „Axiomen“ zur analytischen Bewertung von Testkriterien auf. Gefordert wird, dass „vernünftig formulierte“ Testkriterien alle Axiome erfüllen sollen ([Weyucker86]). Perry und Kaiser benutzen die Axiome zur Angabe einiger, insbesondere durch das Konzept der Vererbung bedingter, Probleme der Testbarkeit objektorientierter Anwendungen ([PerKai90]). Diese Probleme lassen die Anwendbarkeit herkömmlicher Testkriterien für objektorientierte Anwendungen zweifelhaft erscheinen (vgl. [Binder94][JorEri94][HKR+97]).

Wir schlagen daher in diesem Kapitel einige Testkriterien für den Test von objektorientierten Anwendungen vor. Hierbei unterscheiden wir

- ☐ funktionsbezogene Kriterien, die auf dem Use Case Schrittgraph basieren, von
- ☐ strukturellen Kriterien, die sich auf die Implementation beziehen. Diese unterteilen wir weiter in

- strukturell ablaufbezogene Kriterien, die sich an den Aufruf-Interaktionen orientieren und
- strukturell datenbezogene Kriterien, welche sich auf die Daten-Interaktionen konzentrieren.

## Funktionsbezogen

Selbstverständlich sollte die Anwendung in den System- und Abnahmetests gegen alle Use Cases bzw. Test-Szenarien geprüft werden. Anhand der in den Tests gegen die Anforderungsspezifikation erzielten Überdeckung der Use Case Schrittgraphen messen wir die „funktionale“ Vollständigkeit der Tests. Hierfür verwenden wir die Validierungsmetriken (Use Case Schritt-, Kanten-, Szenarien-, Grenze-Inneres- und Pfad-Überdeckung) sowie die bei der Verifikation benutzte minimale Mehrfach-Bedingungsüberdeckung.

In den Systemtests sollte die 100%-ige Use Case Kantenüberdeckung angestrebt werden, da erst so wichtige alternative Abläufe eines Use Case geprüft werden.

## Strukturell ablaufbezogen

In Analogie zu den kontrollflussbasierten Testkriterien geben wir für objektorientierte Anwendungen folgende, auf dem Klassen-Botschaftsdiagramm der Implementation aufbauende Hierarchie von ablaufbezogenen Testkriterien an:

- ☐  $C_0-0$ , Methodenüberdeckung: Alle Methoden der Anwendung müssen mindestens einmal ausgeführt werden. Jeder Methodenknoten des  $KBD_I$  ist überdeckt.
- ☐  $C_0-1$ , Botschaftsüberdeckung: Jede Botschaftsquelle muss mindestens einmal „gefeuert“ haben. Bildet man Äquivalenzklassen der Botschaftskanten des  $KBD_I$  bezüglich  $P$ , so muss aus jeder Klasse mindestens eine Kante durchlaufen sein.
- ☐  $C_0-2$ , polymorphe Botschaftsüberdeckung: Jede Botschaftsquelle muss für jede mögliche dynamische Bindung mindestens einmal „gefeuert“ haben. Alle Botschaftskanten des  $KBD_I$  sind mindestens einmal durchlaufen.
- ☐  $C_0-3$ , polymorphe Grenze-Inneres-Überdeckung: Alle Zyklen im  $KBD_I$ , die nur Botschaftskanten enthalten, werden 0, 1 und 2 mal traversiert.
- ☐  $C_0-n$ , polymorphe Botschaftsüberdeckung: Alle möglichen Abläufe in der Anwendung sind geprüft worden. Es sind alle Pfade im  $KBD_I$  durchlaufen.

Bei rein objektorientierten Sprachen wie z.B. Smalltalk-80 subsumiert der  $C_0-1$ -Test die Anweisungsüberdeckung  $C_0$ , da jeder (atomare) Anweisung<sup>1</sup> eine Botschaft entspricht. Die 100%-ige  $C_0-2$ -Überdeckung kann als minimales Endekriterium bei der Prüfung objektorientierter Anwendungen angesehen werden, wenn im Klassentest die Zweigüberdeckung für jede Klasse bereits erreicht wurde. Ansonsten sollte die  $C_0-3$ -Überdeckung angestrebt wer-

den, da erst diese auf Iterationen sowie direkt- und indirekt-rekursive Aufrufe abzielt. Die  $C_0$ - $n$ -Überdeckung lässt sich zu 100% nur für zyklenfreie Klassen-Botschaftsdiagramme erreichen und ist somit, analog zur Szenarien- und zur vollständigen Pfadüberdeckung in Use Case Schrittgraphen, für die meisten Anwendungen ohne einschränkende Nebenbedingungen nicht erreichbar (vgl. hierzu Beispiel 10.4.).

## Strukturell datenbezogen

Für daten- bzw. zustandsbezogene Testkriterien betrachten wir die mit „def“ und „uses“ markierten Kanten des  $KBD_I$ . In Analogie zu den datenflussorientierten Kriterien beim Test prozeduraler Programme geben wir die folgenden drei Kriterien an.

- ☐ Mindestens eine der möglichen Definitionen, mindestens eine Benutzung.
- ☐ Alle möglichen Definitionen, mindestens eine Benutzung.
- ☐ Alle möglichen Definitionen und Benutzungen.

Im Prinzip sollten (bei ausreichenden Klassentests) alle möglichen Definitionen und Benutzungen der Variablen schon zu Beginn des Tests der gesamten Anwendung geprüft sein. Mit den SCORES grey-box Tests können gezielt Definitionen und Benutzungen von solchen Variablen geprüft werden, die sich zurück zu Attributen und Assoziationen im Domänen-Klassenmodell verfolgen lassen. Im Falle polymorpher können wir zur Auswahl der Klassen auch hier wieder das Verfahren OATS einsetzen (s. [Roy90] [McDMcG94]).

## 18.2 Testausführung

Mit den konkreten Testdaten und dem ggf. instrumentierten Prüfgegenstand erfolgt in der spezifizierten Testumgebung die *Testausführung*. Hierzu werden die Tests in der angegebenen Reihenfolge ausgeführt und die Testergebnisse protokolliert. Gemäß dem gewählten strukturellen Testkriterium (Anweisungsüberdeckung, Zweigüberdeckung etc.) wird zunächst in der *Instrumentierung* der Prüfgegenstand so um zusätzliche Anweisungen erweitert, dass bei der Testausführung Informationen zur Berechnung der erzielten Überdeckung ausgegeben werden (z.B. in Form von Datei- oder Datenbankeinträgen für jede durchlaufene Anweisung bzw. jede ausgewertete Bedingung). Die Testausführung ist somit der dynamische Test im herkömmlichen Sinn. Dabei werden diverse Testprotokolle produziert, darunter ein Ablaufprotokoll, ein Ergebnisprotokoll und ein Protokoll der erreichten Werte für die Test-

---

<sup>1</sup> Hierbei gehen wir davon aus, dass Zugriffe auf atomare Instanzvariablen in entsprechenden Methoden gekapselt sind. Direkt auf Instanzvariablen zugreifende Anweisungen müssen „klassisch“ mit der Anweisungs- bzw. Zweigüberdeckung und mit entsprechenden datenflussbezogenen Kriterien betrachtet werden.

kriterien („Überdeckungsprotokoll“). Zum Ergebnis gehört auch ein *Testvorfallsbericht*, in dem alle Probleme bzw. Abweichungen vom Soll oder sonstige Besonderheiten notiert sind.

Die Test-Skripte werden hierbei entweder durch ein Werkzeug zur automatischen Ansteuerung der Benutzungsoberfläche ausgeführt (Abb. 15.4, Seite 166) oder aber — z.B. für die grey-box Tests — in eine objektorientierte Programmiersprache transformiert, übersetzt und direkt ausgeführt. Zu den Ergebnissen einer Testausführung gehören in erster Linie die Aussage, ob das tatsächliche Ergebnis der Testausführung dem erwarteten, im Testfall spezifizierten Ergebnis entspricht sowie die Überdeckungsprotokolle. Wir sammeln die Ergebnisse in den entsprechenden Test-Suiten.

### 18.3 Testauswertung

Nach der Durchführung werden die Testergebnisse analysiert und verdichtet. Ziel der *Testauswertung* ist es zu entscheiden, ob ein Prüfgegenstand den Test bestanden hat oder ob der Test nach einer Korrektur der Fehler bzw. Defekte wiederholt werden muss. Die Bewertungsergebnisse dokumentieren sich im *Testbericht*. In der Testauswertung werden die protokollierten Ergebnisse der Testausführung mit den Sollwerten verglichen. Hierbei wird entschieden, ob der Test einen Fehler offenbart hat oder nicht. Außerdem wird geprüft, ob bzw. wie weit das Testendekriterium erreicht ist. Die Prüfung der Testergebnisse kann dabei z.B. mit Druck-Repräsentationen, Zusicherungen, speziell für den Test geschriebenen Auswertemethoden oder aber ausführbaren Spezifikationen erfolgen. Die Testauswertung ist schließlich eine Bewertung der Testergebnisse im Hinblick auf die Testziele. Hier gilt es zu entscheiden, wann der Test zu Ende ist bzw. wann der Test abzurechnen ist. Das Ergebnis ist ein Testabschlußbericht.

Wir nutzen wieder das Klassen-Botschaftsdiagramm. Objekt-Instanziierung (und ggf. -Zerstörung) werden protokolliert und mit entsprechenden Kanten des  $KBD_I$  verglichen. Unbenutzte Funktionen, welche die Angabe realistischer Aussagen zur erzielten strukturellen Überdeckung erschweren, ergeben sich direkt als „Wurzelknoten“ des  $KBD_I$ . Die Interaktionen zwischen Objekten werden wieder auf entsprechende Kanten des  $KBD_I$  abgebildet und gegen das  $KBD_A$  geprüft.

In der *Testdokumentation* werden die Tätigkeiten und ihre Ergebnisse festgehalten. Hierzu gehören die Prüfgegenstände selbst (Methoden, Klassen, Teilsysteme, ...) sowie die jeweils benötigte Testumgebung, die Testfallspezifikation sowie Testdaten und Testergebnisse. Die *Testberichte* für die Auswertung der Prüfung einer objektorientierten Anwendung sind ähnlich gestaltet wie die Berichte über „konventionelle“ Prüfungen. Hierzu gehören Fehlerberichte an die Entwickler und Aufstellungen über erreichte Werte von Metriken wie z.B. der Fehlerrate<sup>1</sup>, der Fehleraufdeckungsrate, der erreichten Testüberdeckung und der Test-Überdeckungsrate. Letztendlich fasst der Testabschlussbericht alle Testereignisse und -ergebnis-

se, eine Fehlerstatistik, eine Beurteilung der Test-Überdeckung und eine Aussage über die erreichte Qualität relativ zur geplanten Qualität zusammen.

Hiermit sind wir am Ende unserer konzeptuellen bzw. methodischen Ausführungen angelangt. Die geforderte Brücke von der Anforderungsermittlung über die Prüfung der Anforderungsspezifikation bis hin zum Test der Anwendung (s. Abb. 14.1 auf Seite 159) wurde errichtet. Grundlegende Pfeiler sind die Verfeinerung und Präzisierung von Use Cases zu Use Case Schrittgraphen, die Kopplung mit dem Klassenmodell, die Validierung und Verifikation der Anforderungsspezifikation und die Übernahme bzw. Erweiterung der Test-Szenarien und Episoden zu black-box bzw. grey-box Testfällen. Zum besseren Überblick stellt Tabelle 18.1 noch einmal die Elemente der SCORES-Anforderungsermittlung den entsprechenden bzw. resultierenden Elementen des Tests gegenüber.

<i>Anforderungsermittlung</i>		<i>Test</i>	
Spezifikation	Use Case Diagramm	Testplan	Management
	Use Case (Schrittgraph)	Testsuite	
	Use Case-Schritt	Testfall	
Validierung	Test-Szenario	Testskript	Black-box
	Test-Szenarioschritt	Testfall	
	Bedingungen	Testdaten, Testorakel	
Verifikation	Episode	Testdaten, Testorakel	Grey-box
	Objektkonstellation	Testumgebung	

*Tab. 18.1 Anforderungsermittlung vs. Test*

Mit dem vorgestellten methodischen Vorgehen haben die Tester konkrete Aufgaben über alle Tätigkeiten hinweg und stehen damit in der Entwicklung nicht mehr „am Ende der Schlange“. Über die direkte Zuordnung der Testelemente zu Elementen aus der Anforderungsermittlung wird der Testfortschritt messbarer und damit der Testprozess steuerbar. Die am Anfang der Entwicklung erforderliche Mehrarbeit (Abb. 5.2, s. Seite 60) z.B. bei der Spezifikation von Use Case Schrittgraphen und der „Formalisierung“ der Bedingungen zahlt sich durch den bei den Tests der Anwendung gegen die Anforderungsspezifikation abgeschöpften Mehrwert aus.

---

1 Die Fehlerrate gibt die Anzahl (gefundener) Fehler pro Implementationseinheit (z.B. Fehler / 1000 Lines of Code, error / KLOC) an (s. [Riedemann97]).



# Teil V

## Werkzeugunterstützung

Die in den bisherigen Teilen der Arbeit vorgestellte Methode SCORES zur Anforderungsermittlung, Validierung und Verifikation der Anforderungsspezifikation und Testfallgenerierung stellt hinsichtlich einer wünschenswerten, umfassenden Werkzeugunterstützung keine Ausnahme dar, so dass wir SCORES durch ein entsprechendes Werkzeug, das SCORESTOOL, ergänzt haben.

In diesem Teil gehen wir daher in Kapitel 19 kurz auf die Konzepte und Ziele der Werkzeugunterstützung bei der Anforderungsermittlung sowie beim Test ein. In Kapitel 20 skizzieren wir das SCORESTOOL, unsere Werkzeugkomponenten für die Anforderungsermittlung, Validierung und Verifikation der Anforderungsspezifikation und die Testfallgenerierung mit SCORES.

# Kapitel 19

## Konzepte

*Die Tatsache, ob zu einer Methode Werkzeugunterstützung existiert, entscheidet fast schon allein über deren Einsatz. [PagSix94]*

Bei der zunehmenden Komplexität heutiger Anwendungen ist eine effektive Softwareentwicklung ohne adäquate Werkzeugunterstützung unmöglich, so dass die Anwendbarkeit und Akzeptanz (neuer) Methoden zur Softwareentwicklung entscheidend von der Verfügbarkeit entsprechender Werkzeuge abhängen (CASE, Computer Aided Software Engineering).

Im Allg. werden besondere Anstrengungen bei der Anforderungsermittlung nur dann als sinnvoll betrachtet, wenn sich die in die Anforderungsspezifikation investierte Arbeit bei nachfolgenden Tätigkeiten (z.B. Erstellung und Validierung der Softwarearchitektur oder der Ableitung von Testfällen) unmittelbar auszahlt (vgl. [WPJ+98][DHP+99]). Wir haben die dementsprechenden Möglichkeiten von SCORES in den vorherigen Teilen der Arbeit aufgezeigt.

Das SCORESTOOL ist im Wesentlichen eine Sammlung von Werkzeugkomponenten welche einerseits die Anforderungsermittlung, andererseits auch die Validierung und Verifikation der Anforderungsspezifikation sowie die Generierung von Testfällen für den Test der Anwendung gegen die Anforderungsspezifikation unterstützen.

Wir skizzieren in den folgenden Abschnitten das Vorgehen bei der Entwicklung und die grundlegende Architektur des SCORESTOOLS und stellen in Kapitel 20 einige ausgewählte SCORESTOOL-Komponenten vor.

### 19.1 Entwicklung

Das SCORESTOOL basiert auf dem GEOOOA-Tool, einem Werkzeug zur Unterstützung der GEOOOA<sup>1</sup> ([Kösters97]). Zu Beginn der Entwicklung des GEOOOA-Tools wurden weitgehend unabhängige Teilwerkzeuge identifiziert. Die Realisierung erfolgte inkrementell, wobei



die einzelnen Teilwerkzeuge in einem iterativen Prototyping-Prozess hauptsächlich durch Studierende der FernUniversität Hagen im Rahmen ihrer Diplomarbeiten implementiert wurden (u.a. [Rogotzki97][Schulte97][Uhl97]).

Das Hauptaugenmerk bei der Entwicklung des GEOOOA-Tools und auch des SCORESTOOLS lag auf der durchgängigen Anwendung objektorientierter Entwicklungsmethoden. Die Ermittlung der (funktionalen) Anforderungen an die Komponenten des SCORESTOOLS erfolgte dabei im Wesentlichen mit Hilfe von SCORES selbst. Spezielle Anforderungen bezüglich der Benutzungsschnittstelle wurden mit FLUID ermittelt und spezifiziert ([KSV96]), der Entwurf orientierte sich an *OOSE* ([JCJ+92], s. auch Abschnitt 2.2) und am *Responsibility Driven Design* ([WBW+90]).

Implementiert wurde das SCORESTOOL in der *Smalltalk*-Programmierungsumgebung VISUALWORKS ([VW98]). Für die Auswahl von VISUALWORKS sprachen neben der umfangreichen Klassenbibliothek und der komfortablen Programmierungsumgebung zwei pragmatische Gründe. Erstens ist die Programmierungsumgebung für Studenten bzw. Diplomanden allgemein verfügbar, da die FernUniversität Hagen über eine Campus-Lizenz verfügt. Zweitens ist VISUALWORKS auf fast allen gängigen Hardware-Plattformen lauffähig, was eine breite Einsetzbarkeit des Werkzeugs gewährleistet.

## 19.2 Architektur

Die Komponenten des SCORESTOOLS sind im Grunde als unabhängige Werkzeuge für die Anforderungsermittlung konzipiert, die eine einfache Anbindung als Fremdwerkzeug an eine offene Software-Entwicklungsumgebung (*SEU*, vgl. [Fuggetta97]) ermöglichen. Abb. 19.1 stellt den grundlegenden Aufbau des SCORESTOOLS dar. Die Architektur baut im Wesentlichen auf der des GEOOOA-Tools auf und lässt sich grob in drei Schichten einteilen, die wir nachfolgend kurz besprechen, wobei wir uns eng an [Kösters97] und [Schulte97] anlehnen.

Auf der obersten Schicht befinden sich einzelne, Analytikern und Testern zugängliche Teilwerkzeuge des SCORESTOOLS. Auf dieser Ebene trennt das SCORESTOOL noch die an der Anforderungsermittlung beteiligten Modellierungs- bzw. Validierungs- und Verifikationswerkzeuge von den externen Schnittstellen, mit welchen beispielsweise die Modelle der Anforderungsspezifikation bzw. die bei der Validierungs- und Verifikation protokollierte Information in — soweit wie möglich — äquivalente Darstellungen transformiert werden, die innerhalb der *SEU* weiter bearbeitet werden können.

**Modellierungswerkzeuge.** Zu den Modellierungswerkzeugen zählen beispielsweise Editoren, Browser oder statische Analysatoren. Ein graphischer Editor und unterschiedliche Spezifikationseditoren dienen zum Erzeugen, Ändern und zur Gestaltung des Layouts

---

1 GEOOOA ist eine objektorientierte Methode zur Anforderungsspezifikation für geographische Informationssysteme ([KPS96][Kösters97]).

z.B. von Klassenmodellen. Viele der SCORES-Modellierungsregeln werden von den statischen Analysatoren bereits bei der Konstruktion der entsprechenden Teile der Anforderungsspezifikation überprüft.

**Validierungs- und Verifikationswerkzeuge.** Hierzu zählen beispielsweise Animationswerkzeuge oder statische Analysatoren. Erstere unterstützen Analytiker und Tester bei der Eingabe und Editierung von Objektmodellen, Test-Szenarien und Episoden. Letztere ermitteln z.B. die aktuellen Werte der verschiedenen Vollständigkeits-, Validierungs-, und Verifikationsmetriken.

**Schnittstellen.** Um die Ergebnisse der Anforderungsermittlung und der Validierung und Verifikation der Anforderungsspezifikation auch in den nachfolgenden Entwicklungsschritten verfügbar zu machen, können die Modelle in unterschiedliche Formate bzw. Modelle transformiert werden. Abhängig von der ausgewählten Zielumgebung, kann das Zielmodell zum Beispiel ein ER- oder objektorientiertes Modell sein. Zu jedem Format existiert eine Menge wohldefinierter Transformationsregeln, die jedes Primitiv des SCORES-Metamodells auf (ein oder mehrere) Primitive des Zielmodells transformieren. Unter Berücksichtigung dieser Regeln interpretiert ein Generator die aktuellen SCORES-Modelle und transformiert sie in das entsprechende Zielmodell.

In der zweiten Schicht befinden sich Komponenten, auf die entweder nahezu alle Teilwerkzeuge der ersten Schicht zugreifen (z.B. Metamodell, Speichermanager), oder Frameworks für Editoren, Browser und Generatoren, die bestimmte „generische“ Teilfunktionalitäten zur Verfügung stellen:

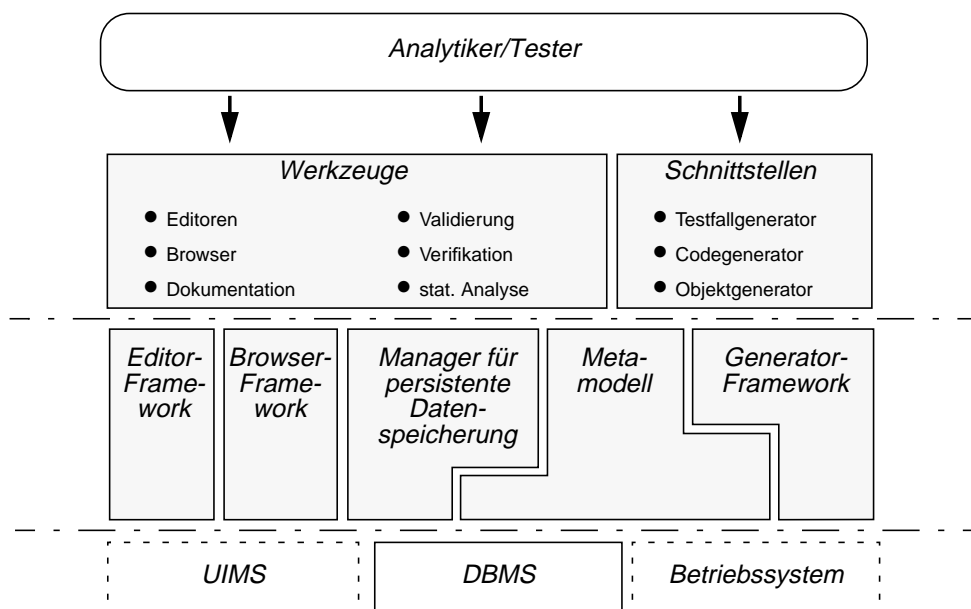


Abb. 19.1 SCORESTOOL: Grundlegender Aufbau (vgl. [Kösters97])

**Metamodell.** Im Zentrum des Werkzeugs steht das SCORES-Metamodell, welches die Syntax und Semantik aller Primitive definiert und die vordefinierten Eigenschaften der Primitive dokumentiert. Im übertragenen Sinne kann das Metamodell als mächtiges, integriertes Datenlexikon angesehen werden, welches eine wohldefinierte Basis für alle Teilwerkzeuge darstellt.

**Manager für persistente Datenspeicherung (MDS).** Das Speichern und Laden von Modellen erfolgt ausschließlich über den MDS des GEOOOA- bzw. des SCORETOOLS. Der MDS stellt eine einheitliche Schnittstelle zu verschiedenen Technologien der persistenten Datenhaltung zur Verfügung.

**Frameworks.** Mit dem Begriff *Framework* wird ein halbfertiges (Teil-)System bezeichnet, das in einem spezifischen Kontext instanziiert werden kann (vgl. [Johnson97][Griffel98]). Beispielsweise definiert das *Generator-Framework* die Architektur für eine Familie von Dokument- und Codegeneratoren sowie Modelltransformatoren. Es enthält die grundlegenden Basisbausteine (z.B. Scanner, Parser, Regel-Interpreter), um anwendungsspezifische Generierungs- bzw. Transformationsregeln zu definieren und Modelle damit auszuwerten. Mit dem *Editor-* und *Browser-Framework* stehen zwei weitere Frameworks zur Bearbeitung der Modellierungprimitive bzw. deren textueller Spezifikationen bereit.

Auf der untersten Ebene befinden sich externe Basiskomponenten, auf denen das WERKZEUG aufsetzt. Zur Speicherung von Modellen können unterschiedliche Datenbanksysteme eingesetzt werden, sofern deren Einsatz durch den MDS unterstützt wird. Die Schnittstellen zum Betriebssystem und zum *User Interface Management System* (UIMS, vgl. [VosNen98]) werden bereits von der Entwicklungsumgebung VISUALWORKS plattform-unabhängig zur Verfügung gestellt, so dass keine weiteren Vorkehrungen für den Einsatz auf unterschiedlichen Plattformen getroffen werden müssen.

# Kapitel 20

## SCORESTOOL

Wir stellen in diesem Kapitel exemplarisch einige ausgewählte Komponenten des SCORESTOOLS vor. Hierbei unterscheiden wir Komponenten zur hauptsächlichen Unterstützung

- ☐ der Spezifikation der Anforderungen,
- ☐ ihrer Validierung und Verifikation und
- ☐ des Tests der Anwendung gegen die Anforderungsspezifikation.

Die unterschiedlichen Prototypen der einzelnen Teilwerkzeuge und die damit durchgeführten Fallstudien haben wiederholt zu einer Überarbeitung bzw. Präzisierung von SCORES geführt. Mittlerweile nähert sich der Prototyping-Prozess seinem Abschluss und das SCORESTOOL stellt die gewünschte Funktionalität fast vollständig zur Verfügung.

### 20.1 Anforderungsspezifikation

Bei der Spezifikation der Anforderungen unterstützt das SCORESTOOL die Erstellung von Use Cases und Klassenmodellen durch entsprechende Editorkomponenten. Die Visualisierung und das grafische Layout der Diagramme erfolgt hierbei auf verschiedenen Abstraktionsstufen. Die textuellen Spezifikationen werden über spezielle Spezifikations-Editoren eingegeben bzw. geändert, wobei spezielle Browser das Editieren und die Verfolgung verschachtelter textueller Spezifikationen erlauben.

**Klassendiagramme.** Klassendiagramme werden in der UML-Notation ([OMG97]) oder wahlweise in der Notation nach Coad und Yourdon ([CoaYou90]) dargestellt. Nach der Markierung eines Modellierungselements können abhängig von seiner Metaklasse weitere Spezifikationseditoren z.B. für die Operationen einer Klasse oder die Multiplizitäten von Beziehungen aktiviert werden. Abb. 20.1 zeigt einen „Screen-Shot“ des Klassendiagramm-Editors.

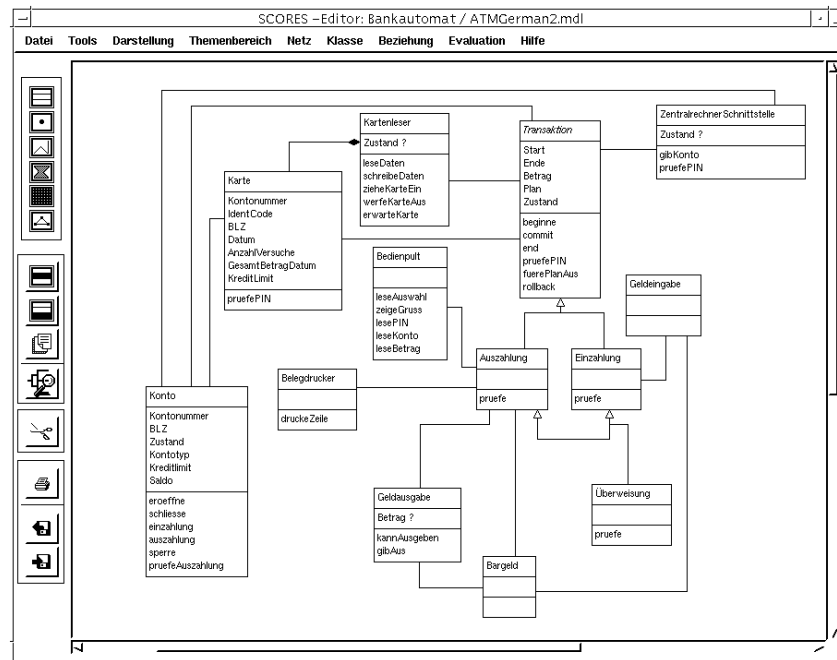


Abb. 20.1 SCORESTOOL Klassendiagramm-Editor

**Use Cases.** Die Spezifikation von Use Cases erfolgt im ScoresTool hauptsächlich textuell, wobei die Eingabe spezieller Werte wie z.B. der Profile (s. Abschnitt 8.5) über entsprechende vorgefertigte Eingabefelder (Widgets, vgl. [VosNen98]) erfolgt. Das Hauptfenster des Spezifikations-Editors für Use Cases ist in Abb. 20.2 gezeigt.

**Use Case Schritte und Use Case Schrittgraphen.** In einem speziellen Graph-Konstruktions- und Visualisierungsframework sind vorgefertigte Werkzeuge zur Erstellung gerichteter Graphen oder Bäume enthalten. Use Case Schritte werden direkt im Use Case

The screenshot shows the 'Bankautomat: Use cases' editor window. The 'Anmelden' use case is selected. The editor displays the following information:

- Name:** Anmelden
- Weight:** 18
- Usage:** high
- Criticality:** medium
- RTechnical:** high
- RBusiness:** high
- RProject:** high
- Vorbedingung:** Kartenleser betriebsbereit, Bedienpult gesperrt
- Beschreibung:** Nach Eingabe der Karte durch den Bankkunde liest der Bankautomat den Code vom Magnetstreifen und prüft ihn auf Zulässigkeit. Bei zulässigem Code fordert der Bankautomat die Eingabe der Identifikationsnummer (PIN). Der Bankkunde gibt die PIN ein. Der Bankautomat überprüft die PIN. falsche PIN eingegeben, so wird der Versuch gezählt: nach drei Fehlversuchen wird die Karte gesperrt.
- Nachbedingung:** Karte ausgeworfen oder Zentralrechner busy, Karte ausgeworfen und Kartenleser betriebsbereit, Bedienpult gesperrt
- IN-Objects:** Kartenleser, Bedienpult
- OUT-Objects:** Karte, Transaktion

Abb. 20.2 SCORESTOOL Use Case Spezifikations-Editor

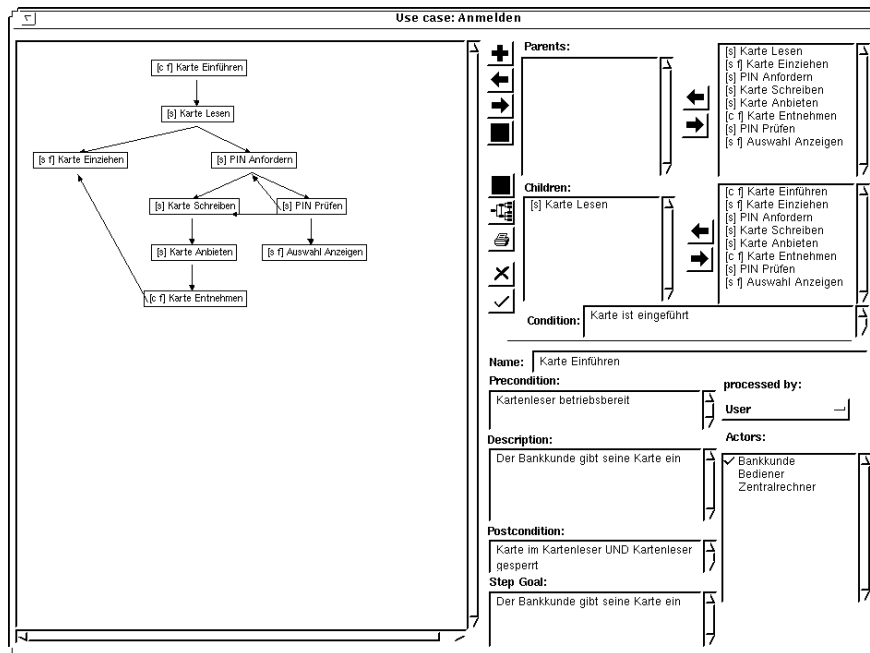


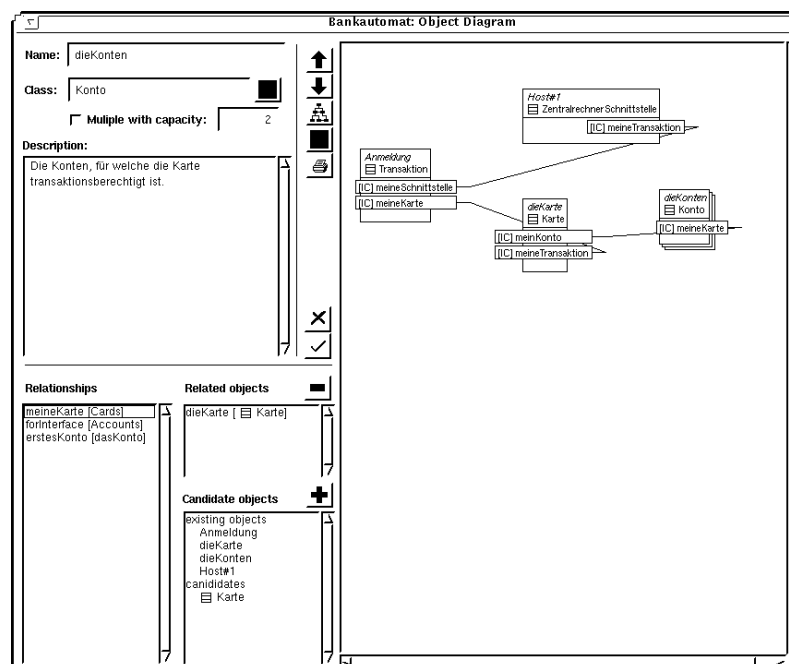
Abb. 20.3 SCORESTOOL Use Case Schrittgraph-Editor

Schrittgraph-Editor spezifiziert und miteinander verbunden. Hierbei können die Vor- und Nachbedingungen der Schritte sowie die Übergangsbedingungen der Kanten (neben der textuellen Spezifikation und der OCL) optional auch in Smalltalk-80 Code eingegeben und auf entsprechenden Objektkonstellationen ausgewertet werden. Abb. 20.3 zeigt den Use Case Schrittgraph-Editor bei der Bearbeitung des Use Case Schrittgraphen Anmelden.

Zusätzlich zu den Editoren sorgen spezielle Browser sowie Verfolgbarkeitshilfen für die konzentrierte Darstellung der gesamten Information für ein selektiertes Element der Anforderungsspezifikation. So können z.B. sämtliche Operationen einer selektierten Klasse sortiert nach der Vererbungsstruktur in einer Liste dargestellt werden, wobei geerbte Operationen nach den jeweils definierenden Klassen zusammengefasst sind.

## 20.2 Validierung

Jede Validierungssitzung bezieht sich auf eine sogenannte Validierungseinheit, welche eine Gruppe von zusammenhängenden Use Cases darstellt. Zunächst können spezielle Sichten auf die Anforderungsspezifikation bezüglich der Validierungseinheit generiert werden, d.h. die mit den Use Cases der Validierungseinheit interagierenden Aktoren, die Schrittgraphen und die in den entsprechenden Klassenbereichen enthaltenen Klassen werden zusammengestellt. Daneben werden diese Elemente anhand des aus den Profilen der Use Cases berechneten Ge-



wichte priorisiert. Die Prioritäten geben Anhaltspunkte für die Reihenfolge und die Intensität der Validierung (bzw. Verifikation) des Elements.

Da z.B. bei Benutzern und Domänenexperten keine entsprechenden Computer- und CASE-Kenntnisse vorausgesetzt werden können, empfiehlt es sich, für solche an der Validierung teilnehmenden Stakeholder vorab textuelle Spezifikationen zu generieren und an diese zu verteilen. Hierfür ermöglicht das SCORESTOOL die Generierung von Spezifikationsauszügen entsprechend der Validierungseinheiten. Im Weiteren skizzieren wir den Objektkonstellations-Editor, die Komponenten des Szenario-Browsers zur Konstruktion und Protokollierung der Test-Szenarien und gehen kurz auf die Validierungsmetriken ein.

**Objektkonstellationen.** Für die Validierung (und Verifikation) erlaubt das SCORESTOOL die manuelle Konstruktion von Objektkonstellationen anhand des Klassenmodells. Ausgehend von einem sogenannten „Wurzelobjekt“ können Analytiker und Tester anhand der Beziehungen der entsprechenden Klasse im Klassenmodell zu weiteren Klassen navigieren und diese instanziiieren. Hierbei werden z.B. Multiplizitätsbeschränkungen der Beziehungen automatisch berücksichtigt. In Abb. 20.4 ist der SCORESTOOL-Objektkonstellations-Editor gezeigt. Ausgehend von der Wurzelklasse **Transaktion** ist zunächst über die Beziehung *meineSchnittstelle* zur Klasse **ZentralrechnerSchnittstelle** das Objekt *Host#1* erzeugt und mit der Transaktion verknüpft worden. Danach wurde ein Objekt der Klasse **Karte** und über die Beziehung *meinKonto* zwischen den Klassen **Karte** und **Konto** einige Instanzen der Klasse **Konto** in die Objektkonstellation eingefügt.

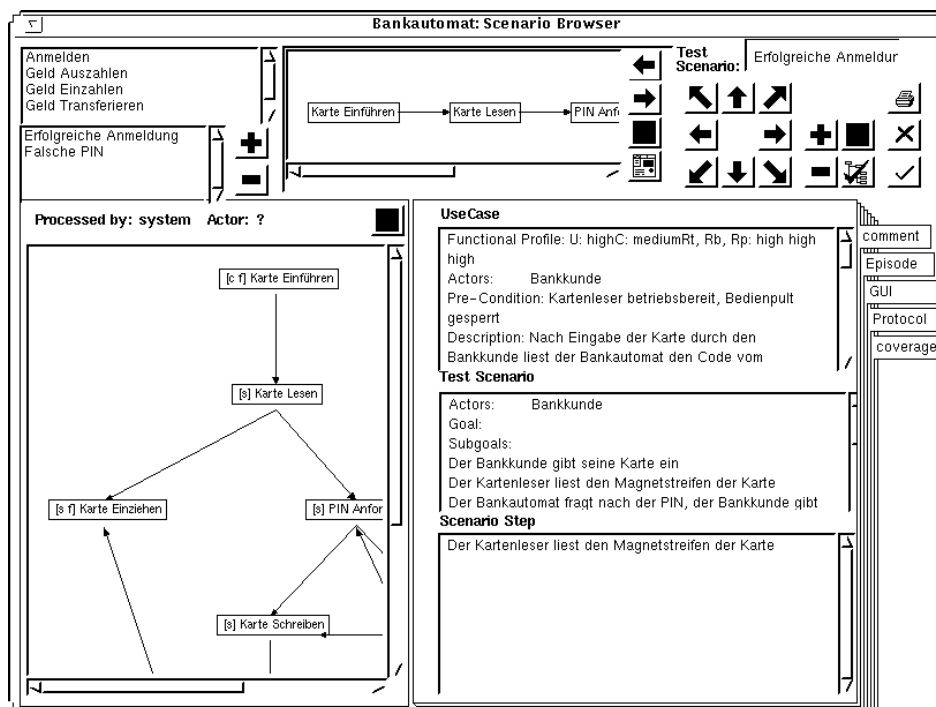


Abb. 20.5 SCORESTOOL Szenario-Browser: textuelle Sicht

**Test-Szenarien.** Während einer Validierungssitzung verwenden Analytiker bzw. Tester den SCORESTOOL-Szenario-Browser zur Konstruktion und Protokollierung der Test-Szenarien. In Abb. 20.5 ist ein „Screen-Shot“ des Szenario-Browsers bei der Konstruktion eines Test-Szenarios zum Use Case Anmelden gezeigt. Oben in der Mitte erkennt man das Test-Szenario, links unten den Schrittgraphen des in der „Listbox“ oben links ausgewählten Use Cases (hier Anmelden). Der Szenario-Browser ermöglicht verschiedene Sichten auf das Test-Szenario — in Abb. 20.5 ist die textuelle Sicht gewählt („Comment“). Benutzerfreundlicher als die textuelle Sicht ist die in den Walk-Throughs vornehmlich verwendete, mit Skizzen der Benutzungsoberfläche animierte Sicht des Szenario-Browsers („GUI“, vgl. [KPR+97]). Für den Schritt PIN Eingeben im Test-Szenario Erfolgreiche Online-Anmeldung zum Use Case Anmelden ist eine solche Sicht in Abb. 20.6 dargestellt.

**Validierungsmetriken.** Auf Anforderung können direkt im Szenario-Browser die erreichten Werte der Validierungsmetriken berechnet und angezeigt werden, so dass Analytiker und Tester unmittelbar den Fortschritt der Sitzung verfolgen können. Abb. 20.7 zeigt die entsprechende Sicht des SzenarioBrowsers.



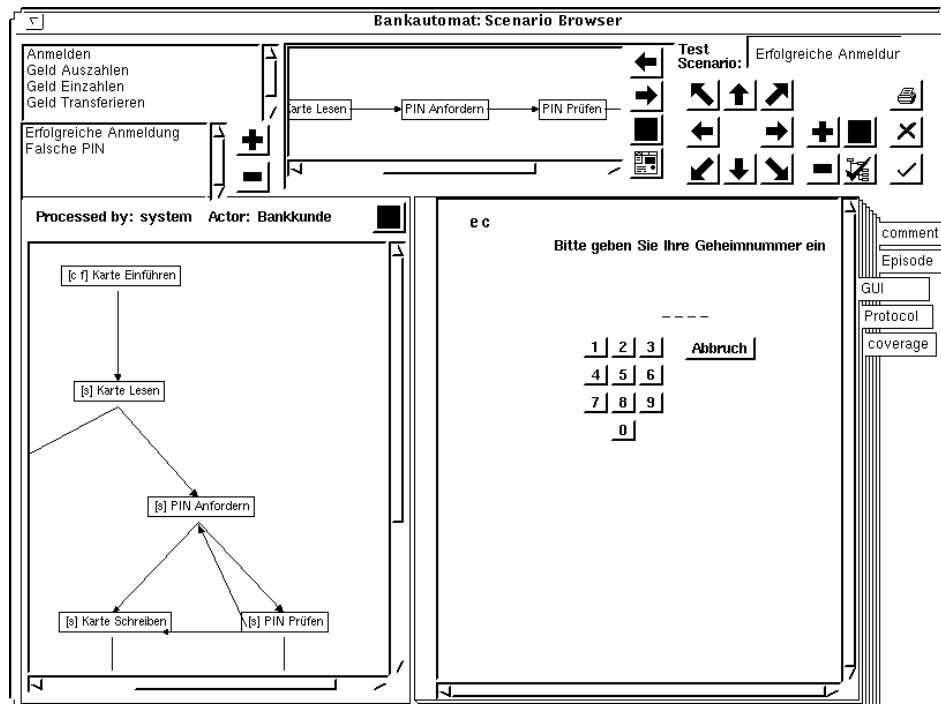


Abb. 20.6 SCORESTOOL Szenario-Browser: GUI-Sicht

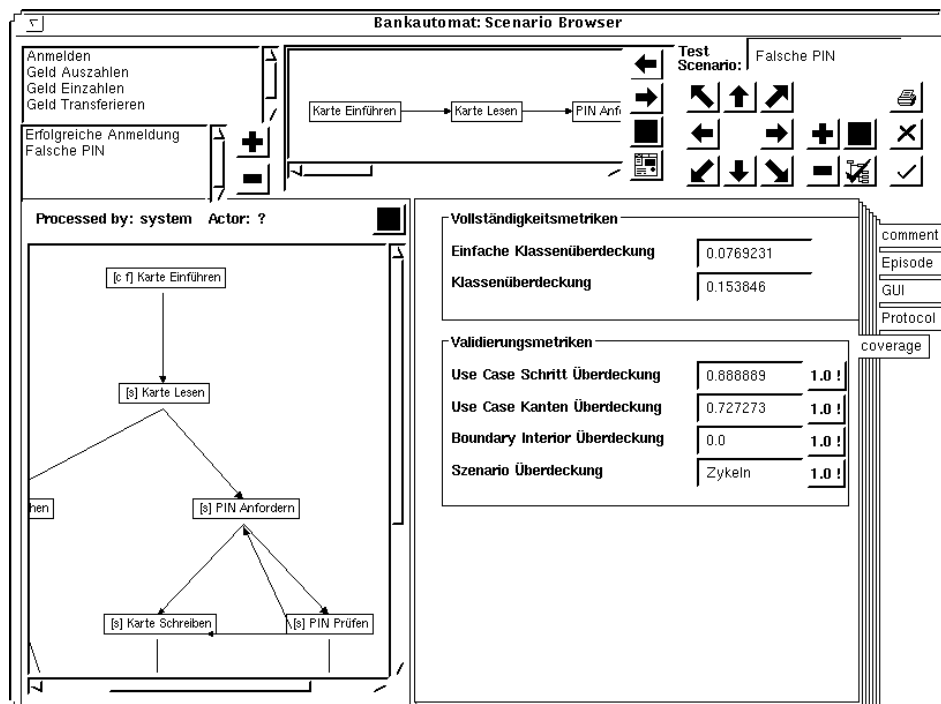


Abb. 20.7 SCORESTOOL Szenario-Browser: Metrik-Sicht

## 20.3 Verifikation

Die Werkzeugunterstützung für die Verifikation ist ähnlich aufgebaut wie die zur Validierung, arbeitet jedoch auf einer wesentlich feineren Granularität und zielt wesentlich umfassender auf das Klassenmodell ab. Wir stellen in diesem Abschnitt den Episoden-Editor und einige erweiterte Möglichkeiten der Verifikationsmetriken vor.

**Episoden.** Mit dem Klassenbereich, der Wurzelklasse und der Wurzeloperation für die Use Case Interaktionsschritte ist festgehalten, welche Operation welcher Klasse im Klassendiagramm für die Aufgabe eines Use Case Schrittes verantwortlich ist. Hierauf basierend können Analytiker und Tester für ein Test-Szenario in Episoden die Ausführung der dem Interaktionsschritt zugeordneten Wurzeloperation in der angenommenen Objektkonstellation durchspielen bzw. „simulieren“. Die unterstützende Komponente des SCORESTOOLS ist der Episoden-Editor. Abb. 20.8 zeigt den Episoden-Editor bei der Simulation der Operation prüfePIN in der Klasse **Zentralrechner-Schnittstelle**. Der aktuelle Sichtbarkeitsbereich ist in der mit „Actual Scope“ betitelten Listbox oben rechts angezeigt; bei Auswahl eines ihrer Elemente werden in der Box ganz rechts die auf dem ausgewählten Element ausführbaren Operationen (inkl. der Standardoperationen) angezeigt und können dann zur Fortsetzung der Episode ausgewählt werden. Im Fenster links wird die Spezifikation der selektierten Operation angezeigt (Signatur und Beschreibung) und kann durch „anklicken“ vom Analytiker

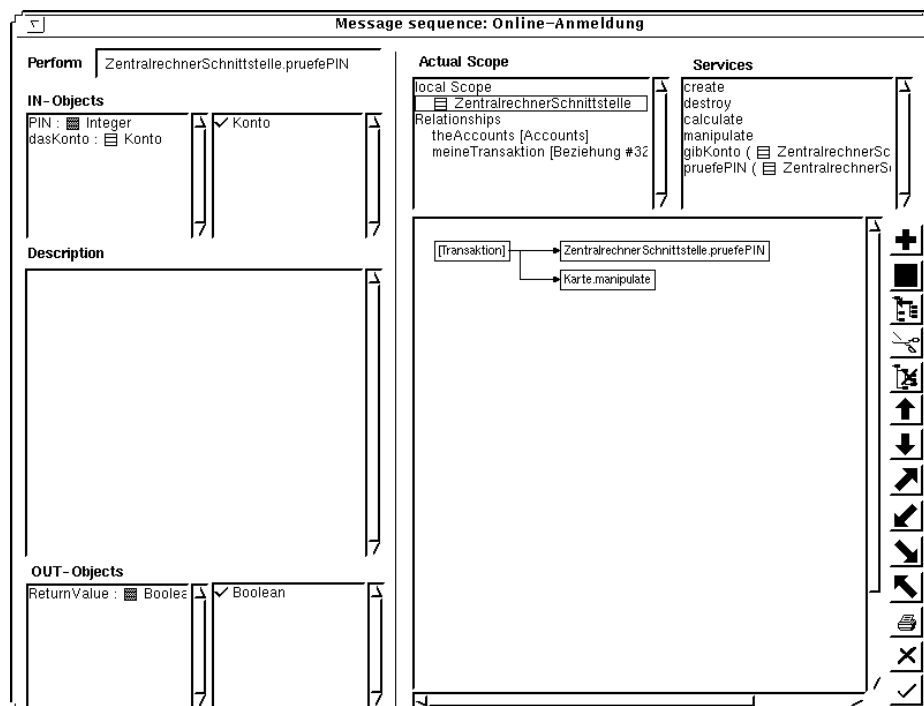


Abb. 20.8 SCORESTOOL Episoden-Editor

bzw. Tester bestätigt werden. Die Darstellung der Episode orientiert sich an der Metapher „Aufrufbaum“; optional kann das UML-Sequenzdiagramm verwendet werden.

**Verifikationsmetriken.** Ähnlich wie bei den Validierungsmetriken können auch die erreichten Werte der Verifikationsmetriken direkt im Szenario-Browser berechnet und angezeigt werden. Darüber hinaus können Analytiker und Tester einzelnen Spezifikationselementen einen Qualitätssicherungs-Status (QS-Status) wie z.B. „in Arbeit“, „Validiert“ oder „Verifiziert“ zuteilen. Der QS-Status beeinflusst optional die Berechnung der entsprechenden Metriken, so dass z.B. Elemente mit dem QS-Status „in Arbeit“ nicht im „Zähler“ der Metriken berücksichtigt werden.

## 20.4 Test

In Kapitel 15 haben wir gezeigt, wie aus den bei der Validierung und Verifikation der SCORES-Anforderungsspezifikation verwendeten und protokollierten Test-Szenarien mit minimalem Aufwand Testfälle für den black-box Test der Anwendung gegen die Anforderungsspezifikation generiert werden. Die Granularität und Semantik der Use Case Schrittgraphen ermöglicht zudem die Anwendung der für die Validierung der Anforderungsspezifikation verwendeten Testkriterien wie z.B. Use Case Knoten- und Kantenabdeckung auch beim (dynamischen) Test der Anwendung.

Wesentlich für die Generierung der Testskripte ist die auf dem Generator-Framework aufsetzende Exportschnittstelle des SCORESTOOLS. Von außen und vereinfacht betrachtet kann das

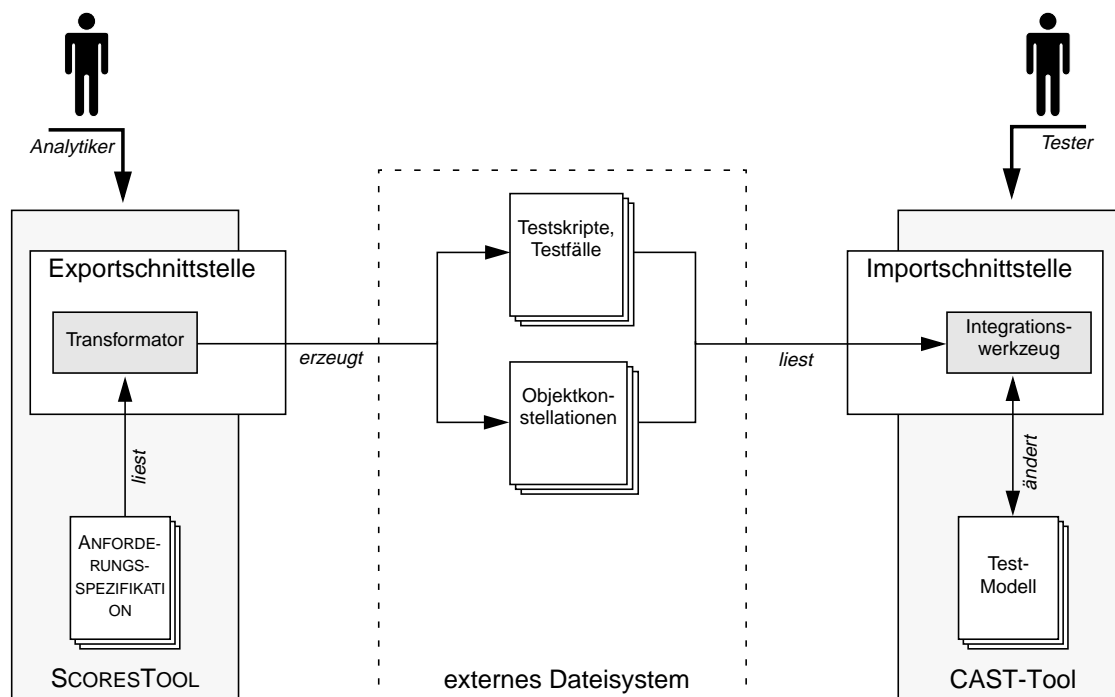


Abb. 20.9 Testfall- und Testdaten-Generierung

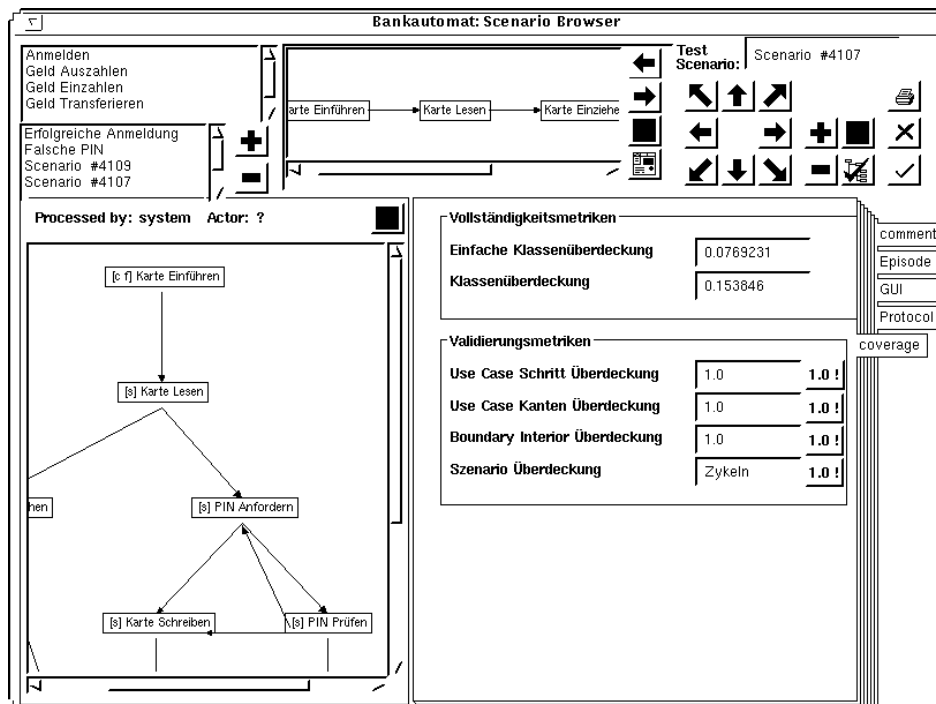


Abb. 20.10 SCORESTOOL Szenario-Browser: Szenario-Generierung

Generator-Framework als konfigurierbarer Cross-Compiler verstanden werden, der die Modelle der SCORES-Anforderungsspezifikation und die Validierungs- bzw. Verifikationsprotokolle in unterschiedliche Zielmodelle bzw. Sprachen übersetzen kann (vgl. [Kösters97] [Schulte97]). Abb. 20.9 zeigt vereinfacht das Schema der Generierung. Zunächst erstellt der Analytiker mit Hilfe der Modellierungswerkzeuge z.B. eine Objektkonstellation. Anschließend wählt er gemäß seines Generierungsziels (z.B. Transformation der Objektkonstellation in Testskripte, welche die Konstellation in der Zielumgebung aufbauen) eine Regelmengenaus und legt mit Hilfe von sogenannten globalen und lokalen Strategien eine adäquate Auswertungsstrategie fest. Nach Aktivierung des Scanners, der die Modelle der SCORES-Anforderungsspezifikation als Tokenstrom an den Parser des Frameworks übergibt, erstellt der Parser einen attribuierten Syntaxbaum in einer temporären Datenstruktur. Der Evaluator des Frameworks wertet den Syntaxbaum gemäß der gewählten Auswertungsstrategie und der zugehörigen Regelmengenaus aus. Das Ergebnis wird wahlweise in einem Browser angezeigt oder in eine Datei ausgelagert und ggf. von einem Testwerkzeug (CAST, Computer Aided Software Testing) gelesen.

**Szenariogenerierung.** Testkriterien bzw. entsprechende Metriken dienen einerseits zur „Kontrolle“ der Tests und andererseits zur gezielten Ableitung von Testfällen. In Bezug auf die Validierungsmetriken können in der Testkomponente des SCORESTOOL-Szenario-Browsers weitere Test-Szenarien zur Erreichung der 100%-igen Use Case Knoten, Kanten, Grenze-Inneres und Szenarienüberdeckung generiert werden. Abb.

20.10 zeigt noch einmal die Metrik-Sicht des Szenario-Browsers, nachdem durch „Anklicken“ des entsprechenden „1.0!“-Buttons die Generierung von Szenarien zur 100%-igen Use Case Kantenüberdeckung erfolgt ist. Die generierten Test-Szenarien sind in der Test-Szenario-Listbox direkt über dem Use Case Schrittgraphen sichtbar. Für den betrachteten Use Case Schrittgraph Anmelden erreicht die 100%-ige Use Case Kantenüberdeckung auch die 100%-ige Use Case Grenze-Inneres Überdeckung, was natürlich im Allg. nicht gilt. Da der betrachtete Use Case Schrittgraph Zyklen beinhaltet und somit die 100%-ige Szenario-Überdeckung nicht erreichbar ist, ist der entsprechende „1.0!“-Button deaktiviert.

**Testfall- und Testdatengenerierung.** Aus Test-Szenarien werden mit dem Transformations-Framework die Botschaftsfolgen sowie Orakel in der Test-Skriptsprache generiert (s. Anhang B). Testdaten in Form von Objektkonstellationen werden aus den manuell erstellten Konstellationen unter Beachtung der Klassenbereiche und des Klassenmodells generiert (vgl. [Rogotzki97]). Hierbei werden über das Transformations-Framework die entsprechenden Botschaftsfolgen zum Aufbau der Objektkonstellationen als Testfälle erzeugt und abgespeichert.

Im Hinblick auf strukturelle Testkriterien nutzen wir in den SCORES grey-box Tests die Vorteile der Prüfung des Zusammenspiels mehrerer Methoden mit Aufrufsequenzen unter Minimierung des höheren Aufwands bei der Erstellung der erwarteten Ergebnisse. Hierzu generiert eine Komponente des SCORESTOOLS das Klassen-Botschaftsdiagramm für die SCORES-Anforderungsspezifikation, welches die bei der Verifikation betrachtete systeminterne Information der Episoden geeignet zusammenfasst. Mit dem Generator-Framework werden dann entsprechende Skripte für die SCORES grey-box Testfälle generiert.

Mit diesem Querschnitt durch einige Spezifikations-, Validierungs- und Verifikations- sowie Testkomponenten des SCORESTOOLS beenden wir unsere kleine „Leistungsschau“ und kommen zum letzten Teil der Arbeit.



# **Teil VI**

## **Resümee und Ausblick**

Mit der Bewertung der erzielten Ergebnisse, einer Zusammenfassung sowie einem kurzen Ausblick erreichen wir in diesem Teil das Ende der Arbeit.

Zusätzlich zum Verzeichnis der in dieser Arbeit verwendeten Literatur und einem kleinen Index haben wir im Anhang noch einige technische Details bezüglich der internen Interaktionen in objektorientierten Anwendungen sowie der Klassen-Botschaftsdiagramme aufgeführt, welche den angemessenen Umfang einzelner Abschnitte sowie den Lesefluss beträchtlich gestört hätten.

# Kapitel 21

## Resümee

Die Qualitätssicherung ist eine der wichtigsten Aufgaben bei der Entwicklung unternehmenskritischer Software. Gerade in der Anforderungsermittlung ist die Qualitätssicherung kritisch für den Projekterfolg, denn Fehler in der Anforderungsspezifikation werden oft erst in späten Projektphasen (z.B. beim Abnahmetest) aufgedeckt und sind aufwendig zu beheben, da meist alle vorangegangenen Tätigkeiten betroffen sind. Die systematische Validierung und Verifikation der Anforderungsspezifikation sind daher von großer Wichtigkeit.

Die Anforderungen umfassen im Wesentlichen funktionale, verhaltens- und ablauforientierte sowie strukturelle Aspekte der Anwendung. In der objektorientierten Anforderungsermittlung werden funktionale Aspekte mit Hilfe von Use Cases formuliert. Neben den Use Cases bildet das (Domänen-) Klassenmodell zur Beschreibung struktureller Aspekte den Kern jeder objektorientierten Anforderungsspezifikation. Das Klassenmodell spielt eine wichtige Rolle im gesamten Entwicklungsprozess, da es als gemeinsame Basis für fast alle zentralen Entwicklungsaktivitäten dient. Jede einigermaßen vollständige Anforderungsspezifikation enthält somit (mindestens) die relevanten Use Cases inkl. ihrer Dynamikbeschreibungen und das Klassenmodell. Da diese Modelle auf unterschiedlichen Techniken und Abstraktionsniveaus basieren, ergeben sich erhebliche Konsistenzprobleme für die Gesamtspezifikation.

In dieser Arbeit haben wir ein durchgehendes Konzept für die Qualitätssicherung bei der objektorientierten Anforderungsermittlung bis hin zu entsprechenden Tests der Anwendung gegen die objektorientierte Anforderungsspezifikation vorgestellt.

### 21.1 Zusammenfassung und Ergebnisse

Thema der Arbeit ist die Qualitätssicherung bei der objektorientierten Anforderungsermittlung sowie der Test einer (objektorientierten) Anwendung gegen die objektorientierte Anforderungsspezifikation. Das Dilemma der Qualitätssicherung wird auch im Spektrum der vorliegenden Arbeit sichtbar: Der „Tester“ muss im Prinzip alles beherrschen, was die ande-



ren Stakeholder wie z.B. Benutzer, Analytiker und Entwickler können — nur ein bisschen „besser“.

Der Ausgangsfragestellung „Wie wird eine objektorientierte Anforderungsspezifikation geprüft bzw. erst prüfbar gestaltet und wie können Testfälle für den Test gegen die geprüfte Anforderungsspezifikation abgeleitet werden?“ entsprechend betreffen die zentralen Ergebnisse die drei jeweils aufeinander basierenden Themenbereiche

- ❑ Präzisierung und Verfeinerung des Use Case Konzepts,
- ❑ Qualitätssicherung bei der Anforderungsermittlung und
- ❑ Test der Anwendung gegen die Anforderungsspezifikation.

Zur Präzisierung der Anforderungsspezifikation mit Use Cases haben wir in dieser Arbeit SCORES (Systematic Coupling of Requirements Specifications) eingeführt. SCORES präzisiert und verfeinert das Use Case Konzept durch sog. *Use Case Schrittgraphen*, welche die Abstraktionslücke zwischen Use Cases und dem Klassenmodell überbrücken und darüber hinaus die Modellierung von Abläufen bzw. „Kontrollfluss“ im Use Case Konzept gestatten (Abb. 21.1). Das Konzept des *Makroschritts* setzt die uses-, extends- und Generalisierungsbeziehungen zwischen Use Cases auf die entsprechenden Use Case Schrittgraphen fort und ermöglicht somit erstmals deren präzise Semantikdefinition. Die Anforderungsermittlung mit SCORES erfasst die kontextuelle-, Interaktions- und systeminterne Information (Abb. 21.2) und berücksichtigt die Qualitätssicherung bereits in den frühen Phasen der Entwicklung objektorientierter Software, da sie den nahtlosen, verfolgbaren *Übergang von den umgebenden Geschäftsprozessen über Use Cases zum (Domänen-) Klassenmodell* ermöglicht.

Aufbauend auf der Verfeinerung und Präzisierung des Use Case Konzepts mit Use Case Schrittgraphen werden sowohl die Validierung der Use Cases und wichtiger Teile des Klassenmodells als auch die Verifikation des Klassenmodells gegen das Use Case Modell und — teilweise — umgekehrt unterstützt. Hierbei erlaubt die Scores-Anforderungsspezifikation die

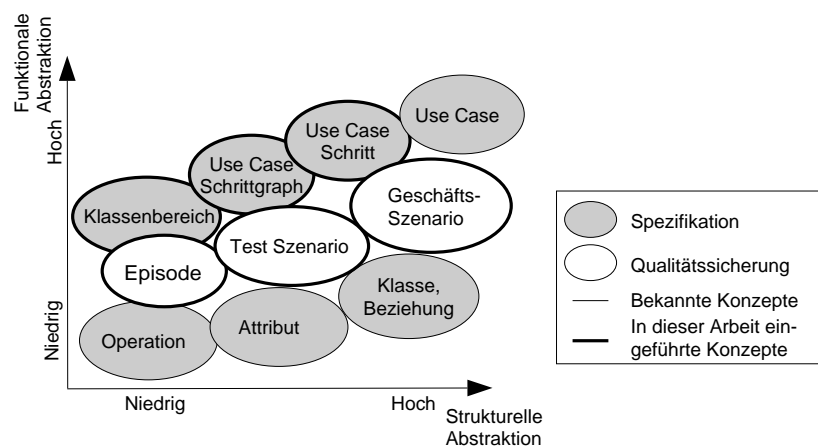


Abb. 21.1 SCORES: Übergang vom Use Case Modell zum Klassenmodell

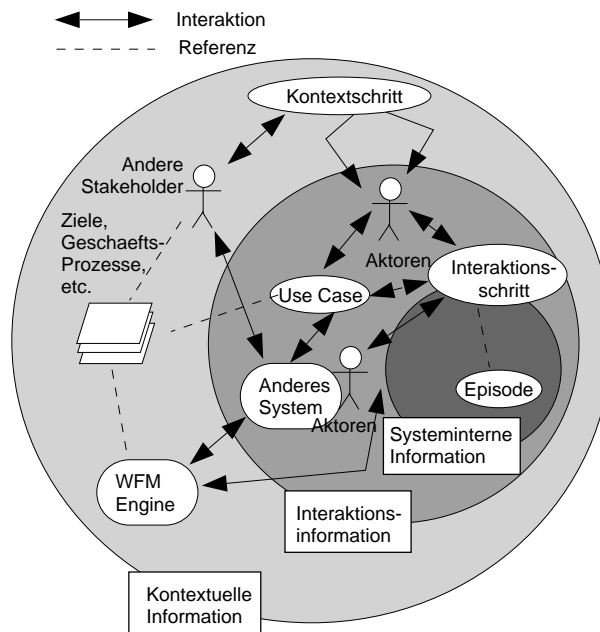


Abb. 21.2 SCORES: Abdeckung der drei Informationsarten (nach [PohHau97])

Angabe von Metriken bzw. Testkriterien zur Formulierung operationaler Testendekriterien, wodurch man erstmalig *quantitative Maße für die Steuerung und Kontrolle des Validierungs- und Verifikationsprozesses* erhält.

Aus der SCORES-Anforderungsspezifikation und den bei ihrer Validierung und Verifikation protokollierten Test-Szenarien lassen sich Testfälle für den *black-box Test* der Anwendung gegen die Anforderungsspezifikation generieren. Zusätzlich können im sogenannten SCORES *grey-box Test* ohne größeren Mehraufwand vertraglich geforderte Werte für strukturelle Testkriterien erreicht werden. Wir nutzen dazu die Vorteile der Prüfung des Zusammenspiels mehrerer Methoden mit Aufrufsequenzen unter Minimierung des Aufwands bei der Spezifikation der erwarteten Abläufe. Hierzu verwenden wir das *Klassen-Botschaftsdiagramm* (KBD) für die SCORES-Anforderungsspezifikation, welches die bei der Verifikation betrachtete systeminterne Information der *Episoden* geeignet zusammenfasst. Zusätzlich generieren wir auch für die zwischen Implementationsklassen möglichen Interaktionen ein Klassen-Botschaftsdiagramm. Dann bilden wir die anforderungsspezifische systeminterne Information auf die zwischen Implementationsklassen möglichen Interaktionen ab, welche daraufhin mit entsprechenden SCORES grey-box Testfällen gezielt ausgeführt werden.

Die Generierung entsprechender „Testdaten“ bzw. der Test-Umgebungs Aufbau führen zum Problem der *Erzeugung von Objektkonstellationen*, für welches wir in Abschnitt 17.2 gezeigt haben, dass es zur Klasse der NP-harten Probleme gehört. Erste Schritte zur heuristischen Erzeugung von Objektkonstellationen sind gegangen worden. Abschließend haben wir die im Rahmen der Arbeit (prototypisch) implementierte Werkzeugunterstützung skizziert.

Das methodische Vorgehen bei der Anforderungsermittlung mit SCORES sowie bei den entsprechenden qualitätssichernden Tätigkeiten ist an die meisten Entwicklungsprozesse adaptierbar und deckt folgende Anforderungen ab (Abb. 21.3):

- ❑ Die Testfalldefinition erfolgt ausgehend von der Anforderungsermittlung. Use Cases, Test-Szenarien und die Elemente des Klassenmodells der Anforderungsspezifikation werden dafür über den Entwurf hinweg bis in den Quellcode der Anwendung nachverfolgt.
- ❑ Die Qualitätssicherung erfolgt risikogesteuert über die Profile der Use Cases, indem die in besonders hoch gewichteten bzw. riskanten Use Cases bzw. Test-Szenarien benötigten Klassen mit entsprechend hoher Priorität (implementiert und) getestet werden.
- ❑ Die in dieser Arbeit entwickelten Klassen-Botschaftsdiagramme für die Anforderungsspezifikation ( $KBD_A$ ) bzw. die Implementation ( $KBD_I$ ) fassen die systeminterne Information aller Episoden bzw. die in der Implementation möglichen internen Interaktionen geeignet zusammen und ermöglicht den SCORES grey-box Test.
- ❑ Zusätzlich unterstützt das  $KBD_I$  neben dem SCORES grey-box Test auch Integrations- und Regressionstests für objektorientierte Anwendungen (vgl. [Winter98]).
- ❑ Die erzeugten Testfälle sind aufgrund der gewählten Einschränkung auf (öffentliche) Methoden und den Verzicht auf besondere „Testmethoden“ auch in Bezug auf das Testorakel weitestgehend sprachunabhängig.

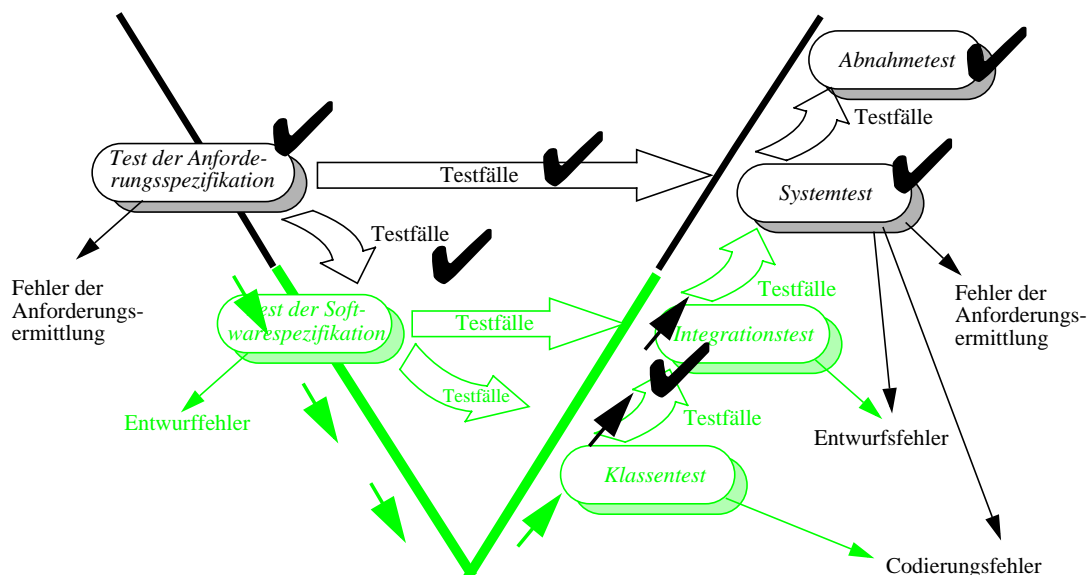


Abb. 21.3 Qualitätssicherung mit SCORES

Die „Negativ-Ergebnisse“ bezüglich struktureller Klassentests und der Generierung von Objekt-Konstellationen stellen den propagierten Vorteilen der Objektorientierung wie z.B. Wiederverwendbarkeit und höhere Produktivität einen erhöhten Aufwand beim Test entgegen. Dieser kann durch die SCORES grey-box Tests — zumindest teilweise — wieder aufgefangen werden.

## 21.2 Erfahrungen

Bis jetzt haben wir die Methode und das Werkzeug hauptsächlich in eigenen Re-Engineering Projekten erprobt, in denen wir die Anforderungen an einige kleinere bis mittelgroße z.T. kommerzielle Anwendungen mit SCORES spezifiziert und geprüft haben. Die Ergebnisse dieser Aktivitäten führten zu einer wiederholten Überarbeitung bzw. Präzisierung des methodischen Vorgehens, des SCORES-Metamodells und des SCORESTOOLS.

Der Grund dafür, dass wir derartige Re-Engineering-Versuche einer konstruktiven Entwicklung von Anwendungen vorgezogen haben, ist die Tatsache, dass SCORESTOOL zur Zeit nur als Prototyp vorliegt, der fortwährend vervollständigt und überarbeitet wird. Außerdem sind in einem kommerziellen Projekt schon allein wegen des meist doch sehr engen Zeitrahmens parallellaufende Vergleichsentwicklungen kaum möglich.

Soweit Auftraggeber und Benutzer von SCORES betroffen sind, erwarten wir weiterhin das gleiche hohe Maß an Zustimmung wie wir es bereits bei den Re-Engineeringprojekten und unseren Studierenden erhalten haben. Ausschlaggebend sind die benutzerfreundlichen Sichten und die Betonung des Ablaufaspektes: In der Regel waren die Anwender offen für den Ansatz und empfanden die erstellten SCORES-Anforderungsspezifikationen als zweckmäßiger und aussagekräftiger als die bisherigen textuellen Pflichtenhefte. Kombinierte Walk-Throughs der Use Cases (vgl. [KPR+97]) mit konkreten (benutzerfreundlich dargestellten) Objektkonstellationen haben insbesondere bei Personen ohne Modellierungserfahrung sehr zum Verständnis der strukturellen Aspekte der Anforderungsspezifikation beigetragen und bei der präzisen Formulierung von Kommentaren und Änderungswünschen geholfen.

Auch auf Seiten der Analytiker und Tester hatten wir bisher keine Probleme, diese für SCORES zu begeistern. Analytiker begrüßten am meisten die hierarchische, redundanzvermeidende Modellierung von Use Cases sowie die erstmals anhand der funktionalen Anforderungen überprüfbare „Funktionalität“ des Klassenmodells ([KPW98]). Tester sind erstmals in der Lage, auch bei der Qualitätssicherung in der Anforderungsermittlung systematische, mit entsprechenden Endekriterien untermauerte Prüfverfahren anwenden zu können ([Winter97]). Alles in allem wurden Werkzeuge wie der SCORESTOOL-Szenario-Browser als adäquate Unterstützung der entsprechenden Tätigkeiten angesehen.

# Kapitel 22

## Ausblick

### 22.1 Laufende Arbeiten

Zur Zeit portieren wir wesentliche Komponenten des SCORESTOOLS in entsprechende Stereotype und Skripts für das kommerzielle CASE-Tool Rational Rose<sup>®</sup> ([Rose98]). Die Erweiterungen werden in Diplomarbeiten implementiert und getestet (vgl. [Ritter2000] [Wulf2000]). Wenn der interne Entwicklungs- und Erprobungsprozess abgeschlossen ist, haben wir eine stabile Basis, um Methode und Werkzeug in „normalen“ Entwicklungen zu erproben. Es haben sich bereits kommerzielle Softwareentwickler interessiert gezeigt, SCORES in ihren Projekten einzusetzen. Allerdings sind die Konzepte für die Aufteilung der Anforderungsspezifikation in überschaubare Teile in SCORES weitgehend auf das UML-Paketkonzept abgestützt, so dass — angeregt durch Erfahrungen der Analytiker und Tester beim praktischen Einsatz der Methode — noch mit Ergänzungen oder Präzisierungen hinsichtlich der Verwaltung „großer“ Anforderungsspezifikationen zu rechnen ist.

Weitere laufende Arbeiten betreffen die Wiederverwendung und die weitergehende Auswertung von Use Case Schrittgraphen.

**Wiederverwendung.** Im Hinblick auf die Wiederverwendung von Use Case Schrittgraphen untersuchen wir zur Zeit sogenannte „Analyse-Muster“, die als wiederverwendbare Teile von Anforderungsspezifikationen bisher hauptsächlich für die strukturellen Aspekte angegeben wurden (vgl. [Fowler97a]). Wir wollen solche Muster mit den entsprechenden Use Case Schrittgraphen erweitern und als generische „Minispezifikationen“ katalogisieren bzw. im SCORESTOOL verfügbar machen (bzgl. der Werkzeugunterstützung für Entwurfsmuster s. auch [PagWin96]).

**Objektkonstellationen.** Die Vor-, Nach- und Kantenbedingungen der Use Case Schrittgraphen werden in Heuristiken zur Generierung von Objektkonstellationen verwendet (s. Abschnitt 17.2). Hier wird zur Zeit der Einsatz verschieden ausdrucksstarker Teilmengen von Programmiersprachen zur Formulierung und automatischen Auswertung der Constraints untersucht.

**UML-Aktivitätsdiagramm.** Die in Abschnitt 7.6 aufgeworfenen Probleme von Aktivitätsdiagrammen werden zur Zeit von der „UML Task Force“ der Object Modeling Group (OMG) bearbeitet (vgl. [OMG-AT98]). Wir untersuchen die notwendigen Änderungen des UML-Metamodells bzw. der UML-Semantik, um das von uns vorgeschlagene Konzept des Use Case Schrittgraphen mit Aktivitätsdiagrammen ausdrücken zu können und somit SCORES ganz auf der UML basieren zu können. Dies käme insbesondere der Forderung nach einer einheitlichen Modellierungssprache entgegen.

**Testbarkeit.** Die Komplexität der internen Interaktionen in objektorientierten Programmen sowie der Generierung von Objektkonstellationen als Testdaten gibt Anlass zu weitergehenden Forschungen im Bereich der Testbarkeit objektorientierter Anwendungen. Insbesondere die Relation der in dieser Arbeit angegebenen, auf dem Klassen-Botschaftsdiagramm basierenden, strukturellen Überdeckungskriterien in Bezug auf zustandsbezogene Test-Endekriterien (vgl. [Binder95], [HKR+97]) sowie die produkt- und fehlererwartungsbezogene „Generierung“ effektiver Teststrategien (vgl. [Liggesmeyer96]) wird untersucht.

## 22.2 Zukünftige Arbeiten und offene Fragen

In diesem Abschnitt skizzieren wir einige geplante Arbeiten bzw. offene Fragen. Diese richten sich einerseits auf die weitergehende formale Unterstützung insbesondere der Verifikation und andererseits auf den objektorientierten Entwurf auf der Basis von Scores Anforderungsspezifikationen. Daneben kommen wir auch auf einige eher Modellierungstechnische Fragen zu sprechen.

**Nebenläufigkeit.** Dieser Arbeit liegt ein synchrones, sequentielles Ablaufmodell zugrunde. Zu betrachten ist noch, wie sich SCORES zur Spezifikation nebenläufiger Ausführungen gleicher oder verschiedener Use Cases erweitern lässt. Bei mehreren Instanzen eines Use Cases ist sicher dann eine Synchronisierung notwendig, wenn identische Objekte von der Bearbeitung betroffen sind. Bei verschiedenen Use Cases können wir über die Klassenbereiche, die Vor- und Nachbedingungen und insbesondere die bei der Verifikation protokollierten Episoden mögliche Synchronisationsprobleme erkennen. Es könnten Transaktionsmechanismen und Sperrverfahren bei Datenbanken zur Anwendung kommen. Andererseits lässt sich diese Information auch zur gezielten Ableitung von Last- und Stresstests verwenden. Dazu muss das geforderte Antwortverhalten in der Anforderungsspezifikation enthalten sein.

**Formalisierung.** Eine weitergehende (werkzeugunterstützte) Verifikation der Anforderungsspezifikation setzt eine stärkere Formalisierung voraus, die dann auch für den Übergang in den Entwurf hilfreich wäre. Aufbauend auf (objektorientierten) Spezifikationssprachen (vgl. [Poetzsch97]) ist ein Verifikationskalkül für Use Case

Schrittgraphen denkbar. Allerdings ist der Übergang zu eher informellen, benutzerorientierten Sichten der Anforderungsspezifikation zu wahren.

**Simulation.** Werden Use Case Schrittgraphen mit Aktivitätsdiagrammen modelliert, erlaubt die Ausdruckstärke der zugrundeliegenden Zustandsmaschinen (Statecharts, vgl. [HarNaa96]) eine automatische Simulation. Solche Simulationen könnten — in Verbindung mit „funktionsfähigen“ (generierten) Domänenklassen — zur Validierung und Verifikation der Anforderungsspezifikation verwendet werden.

**Entwurf.** Eine der Stärken der objektorientierten Softwareentwicklung ist die Durchgängigkeit der Methoden von der Anforderungsermittlung (OOA) über den Entwurf (OOD) bishin zur Implementation (OOP). In bezug auf die Anforderungsspezifikation mit SCORES stellt sich beim Entwurf natürlich die Frage, ob die in den Episoden protokollierte systeminterne Information im Entwurf weiter verwendet bzw. adaptiert werden oder ob nur neue, entwurfsspezifische dynamische Sichten konstruiert und verwendet werden. Jacobson z.B. schlägt vor, die einem Use Case zugeordneten Teile eines Modells zu isolieren und dann auf die diesem Use Case zugeordneten Entitäten anderer Modelle abzubilden (s. Kap. 12 in [Carroll95]). Wir glauben, dass in objektorientierten Entwürfen solche in „Kontroll-Klassen“ implementierte und isolierte Funktionalität gerade in Bezug auf die Testbarkeit mit Vorsicht behandelt werden muss.

**GUI-Unterstützung.** In den abgeleiteten black-box und SCORES grey-box Testfällen ist der Bezug zur Benutzungsoberfläche zur Zeit nur über die Verfolgbarkeit der entsprechenden Interaktionsschritte bzw. Wurzeloperationen möglich. Zur GUI-gestützten Ausführung der Tests muss schon bei der Anforderungsermittlung mehr Information

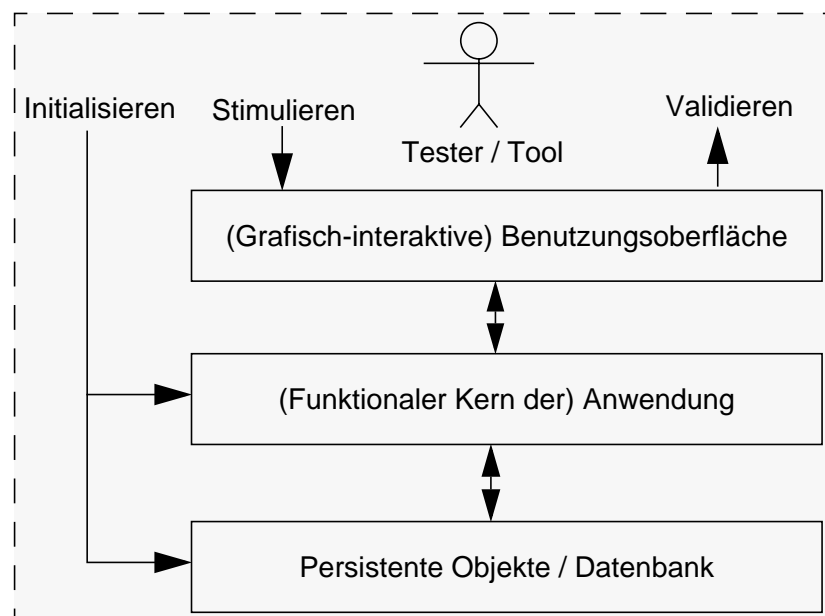


Abb. 22.1 Vollständiger GUI-basierter Systemtest

über die Benutzungsoberfläche eingebracht werden. Eine Vorgehensweise zur kombinierten Ermittlung von Domänen- und Benutzungsschnittstellen-Anforderungen ([KSV96]) wird zur Zeit in den Bereichen Software-Entwurf und -Spezifikation erweitert ([HomVos97]). Mit FLUID könnten die sich an der Benutzungsoberfläche ergebenden Interaktionen für ein Test-Szenario abgebildet und die (generierten) Oberflächenobjekte vom GUI-Testwerkzeug stimuliert werden. Wie in Abb. 22.1 gezeigt würde so ein vollständig GUI-basierter Test möglich.

## Epilog

Es verbleibt die Frage, ob der Jungfernflug der Ariane 5 Rakete („Flug 501“) bei der Verwendung z.B. von SCORES zu einem rühmlicheren Ende gekommen wäre? Natürlich werden wir diese Frage niemals definitiv beantworten können, denn wie bei allen Katastrophen war das eigentliche Unglück das letzte Glied in einer Kette von Ereignissen beginnend mit der Wiederverwendung des „alten“ Codes bis hin zur aus Kostengründen getroffenen Entscheidung gegen entsprechende Tests. Solche Ereignisketten sind schwierig zu durchbrechen und noch schwieriger vorherzusehen.

Wir glauben allerdings, dass präzise Anforderungsspezifikationen in Verbindung mit adäquaten Methoden zur Qualitätssicherung und durchgängiger Verfolgbarkeit der Anforderungen zu den sie implementierenden Teilen der Software die Wahrscheinlichkeit eines solchen Desasters drastisch verkleinern.

Dem steht jedoch entgegen, dass Entwicklungsprozesse in ihrem Umfeld oder — objektorientiert ausgedrückt — mit ihren geerbten Faktoren oft sehr schwer zu durchschauende Netzwerke mit sehr großer „träger Masse“ bilden. Die in der industriellen Praxis der Softwareentwicklung notwendigen (und nach dem Stand der Forschung möglichen) Innovationen bei den verwendeten Methoden erfolgen zur Zeit und auch in absehbarer Zukunft aufgrund dringlicher „Reparaturarbeiten“ wie z.B. der Euro-Umstellung und der Lösung des Jahr-2000-Problems nicht. Solange Software von Menschen unter Zeitdruck in unvollständig definierten Prozessen entwickelt wird, muss mit Softwarefehlern gerechnet werden.



# Literatur

- [ABR+94] R.D. Acosta, C.L. Burns, W.E. Rzepka, J.L.Sidoran *A Case Study of Applying Rapid Prototyping Techniques in the Requirements Engineering Environment* Proc. ICRE 94, IEEE 1<sup>st</sup> Int. Conf. on Requirements Engineering, Colorado Springs, April 1994, S. 66-73
- [Balzert98] Helmut Balzert *Lehrbuch der Software-Technik* Bd. II, Spektrum Akademischer Verlag, Heidelberg, 1998
- [Barbey97] Stéphane Barbey *Test Selection for Specification-Based Unit Testing of Object-Oriented Software based on Formal Specifications* Dissertation, Univ. Lausanne, 1997
- [Behringer97] Dorothee Behringer *Modelling Global Behavior with Scenarios in Object-Oriented Analysis* Dissertation, Dept. of Informatics, Ecole Polytechnique Federale de Lausanne, Swiss, 1997
- [Beizer90] Boris Beizer, *Software Testing Techniques* 2. Aufl, Van Nostrand Reinhold, New York, 1990
- [Beizer94] Boris Beizer *Testing Technology — The Growing Gap* American Programmer, Jahrg. 7, Nr. 4, 1994
- [Beizer95] Boris Beizer *Black Box Testing* J. Wiley & Sons, New York, 1995
- [Berard93] Edward V. Berard *Essays on Object-Oriented Software Engineering* Prentice Hall, Englewood Cliffs, New Jersey, 1993
- [BerMar93] Antonia Bertolino, Martina Marré *Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs* IEEE Transactions on Software Engineering, Jahrg. 20, Nr. 12, 1993, S. 885-899
- [BHJ+98] Rainer Burkhardt, Peter Hruschka, Nicolai Josuttis, Bernd Kahlbrand, Arnulf Mester, Horst Neumann, Bernd Oestereich, Markus Reinhold *UML auf gut Deutsch* OBJEKTSpektrum, Nr. 5, Sep./Okt. 1998, S. 48-49  
Aktuelle Version verfügbar unter <http://www.system-bauhaus.de/uml>

- [Binder94] Robert V. Binder *Design for Testability in Object-Oriented Systems* Communications of the ACM, Jahrg. 37, Nr. 12, Sep. 1994, S. 87-101
- [Binder95] Robert V. Binder *The FREE Approach to Testing Object-Oriented Software* Technischer Report, RBSC Corporation, 1995
- [Binder96a] Robert V. Binder *Testing Object-Oriented Software - A Survey* Software Testing, Verification and Reliability, Jahrg. 6, 1996, S. 125-252
- [Binder96b] Robert V. Binder *The FREE Approach For System Testing* Object Magazine, Jahrg. 5, Nr. 9, Feb. 1996, S. 72-79+81
- [BlaFrö98] Günther Blaschek, Joachim Hans Fröhlich *Recursion in Object-Oriented Programs* Journal of Object-Oriented Programming, Nov./Dez. 1998, S. 28-35
- [Boehm84] Barry B. Boehm *Verifying and Validating Software Requirements and Design Specifications* IEEE Software, Jahrg. 1, Nr. 1, Jan. 1984, S. 75-88
- [Booch94] Grady Booch *Object-Oriented Analysis and Design* Addison-Wesley, Reading, Mass., 1994
- [Brooks87] Frederick P. Brooks Jr. *No Silver Bullet — Essence and Accidents of Software Engineering* IEEE Computer, Jahrg. 20, Nr. 10, April 1987, S. 10-20
- [BMR+97] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal *Pattern-Oriented Software Architecture — A System of Patterns* J. Wiley & Sons, Chichester, 1996
- [Carroll95] John M. Carroll (Ed.) *Scenario-Based Design — Envisioning Work and Technology in System Development* J. Wiley & Sons, New York, 1995
- [CarSto94] David Carrington, Phil Stocks *A Tale of Two Paradigms: Formal Methods and Software Testing* TR Nr. 94-4, Software Verification Research Centre, Univ. Queensland, Australia, 1994
- [CGL81] Augusto Celentano, Carlo Ghezzi, Frederica Liguori *A Systematic Approach to System and Acceptance Testing* in: B. Chandrasekaran, S. Radicchi (Hrsg.) *Computer Program Testing* zugl. *Proc. SOGESTA '81*, Urbino, Italien, Elsevier, North Holland, 1981, S. 279-287
- [CLS+98] K. H. Chang, S.-S. Liao, S.B. Seidman, R. Chapman *Testing object-oriented programs: from formal specification to test scenario generation* Journal of Systems and Software, Jahrg. 42, Nr. 2, Aug. 1998, S. 141-151
- [CoaYou90] Peter Coad, Edward Yourdon *Object-Oriented Analysis* Prentice Hall, Englewood Cliffs, New Jersey, 1990

- 
- [CoaYou91] Peter Coad, Edward Yourdon *Object-Oriented Design* Prentice Hall, Englewood Cliffs, New Jersey, 1991
- [Cockburn97] Alistair Cockburn *Goals and Use Cases* Journal of Object-Oriented Programming, Jahrg. 8, Nr. 6/7 Sep./Nov. 1997
- [CooDan94] Steve Cook, John Daniels *Designing Object Systems — Software isn't the Real World* Journal of Object-Oriented Programming, Mai 1994, S. 22-28
- [Corriveau96] Jean-Pierre Corriveau *Traceability Process for Large OO Projects* IEEE Computer, Sep. 1996, S. 63-68
- [CusSel96] Michael A. Cusumano, Richard W. Selby *Die Microsoft Methode* Heyne, München, 1996
- [DBB97] Benedicte Dano, Henri Briand, Frank Barbier *An approach based on the concept of use case to produce dynamic object-oriented specifications* Proc. 3rd IEEE International Symposium on Requirements Engineering, Annapolis, Maryland, USA, 1997, S. 54-64
- [Davis90] Alan M. Davis *Software Requirements — Analysis and Specification* Prentice Hall, Englewood Cliffs, New Jersey, 1990
- [DHP+99] Bernhard Deifel, Ursula Hinkel, Barbara Paech, Peter Scholz, Veronika Thurner *Die Praxis der Softwareentwicklung: Eine Erhebung* Informatik-Spektrum, 22. Jahrg., Nr. 1, 1999, S. 24-36
- [Deutsch82] Michael S. Deutsch *Software Verification and Validation - Realistic Project Approaches* Prentice Hall, Englewood Cliffs, New Jersey, 1982
- [DesObe96] Jörg Desel, Andreas Oberweis *Petri-Netze in der Angewandten Informatik — Einführung, Grundlagen und Perspektiven* Themenheft: Wirtschaftsinformatik, 38. Jahrg., Nr. 4, Juli 1996
- [Dijkstra76] Edsger W. Dijkstra *A Discipline of Programming* Prentice Hall, Englewood Cliffs, New Jersey, 1976
- [DKR+91] Roger Duke, Paul King, Gordon Rose, Graeme Smith *The Object-Z Specification Language: Version 1* Technischer Report, TR 91-1, Universität Queensland, Australien, 1991
- [Fagan76] Michael E. Fagan *Design and Code Inspections to Reduce Errors in Program Development* IBM Systems Journal, Jahrg. 15, Nr. 3, 1976, S. 182-211

- [Faulk97] S.R. Faulk *Software Requirements: A Tutorial* in: Software Engineering, Hrsg.: M. Dorfman u. R. Thayer IEEE Computer Society Press, Los Alamitos, 1997, S. 82-103
- [FerSin95] Otto K. Ferstl, Elmar J. Sinz *Re-engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes* Bamberger Beiträge zur Wirtschaftsinformatik Nr. 26, Universität Bamberg, 1995
- [FMD97] Bob Fields, Nick Merriam, Andy Dearden *DMVIS: Design, Modelling and Verification of Interactive Systems* Proc. 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems, Granada, Spanien, Juni 1997
- [Firesmith98] Donald G. Firesmith *Use Cases: The Pros and Cons* Technical Journal, Knowledge Systems Corp., Jahrg. 1, Nr. 8, 1998
- [Fowler97a] Martin Fowler *Analysis Patterns — Reusable Object Models* Addison-Wesley, Reading, Mass., 1997
- [Fowler97b] Martin Fowler *UML Distilled* Addison-Wesley, Reading, Mass., 1997
- [FreWei90] Daniel P. Freedman, Gerald M. Weinberg *Handbook of Walkthroughs, Inspections, and Technical Reviews* 3<sup>rd</sup>. Ed., Dorset House Publishing, New York, 1990
- [FNZ96] Arne Frick, Rainer Neumann, Wolf Zimmermann *Eine Methode zur Konstruktion robuster Klassenhierarchien* Proc. Softwaretechnik 96, Koblenz, Sep. 1996, S. 16-23
- [Fuggetta97] A. Fuggetta, 1997 *A Classification of CASE Technology* in: Software Engineering, Hrsg.: M. Dorfman u. R. Thayer, IEEE Computer Society Press, Los Alamitos, 1997, S. 469-482
- [GarJoh79] Michael R. Garey, David S. Johnson *Computers and Intractability* W.H. Freeman and Company, San Francisco, 1979
- [GHJ+94] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides (the Gang of Four) *Design Patterns — Elements of Object-Oriented Reusable Software* Addison-Wesley, Reading, Mass., 1994
- [GelHet88] David Gelperin, Bill Hetzel *The Growth of Software Testing* Communications of the ACM, Jahrg. 31, Nr. 6, Juni 1988, S. 687-695
- [Gerrard94] Paul Gerrard *Testing Requirements* Proc. 2<sup>nd</sup> euroSTAR, Brüssel, Belgien, Okt. 1994
- [GJS96] J. Gosling, B. Joy, G. Steele *The Java Language Specification* Addison Wesley, Reading, Mass., 1996

- 
- [Glass98] Robert L. Glass *How Not to Prepare for a Consulting Assignment, and Other Ugly Consultancy Truths* Communications of the ACM, Jahrg. 41, Nr. 12, Dez. 1998, S. 11-13
- [Glinz95] Martin Glinz *An Integrated Formal Model of Scenarios Based on Statecharts* Proc. ESEC'95, Sitges, Spain (LNCS 989, Springer), 1995
- [GolRob89] Adele Goldberg, David Robson *Smalltalk-80 — The Language* Addison-Wesley, Reading, Mass., 1989
- [GotFin94] O.C.Z. Gotel, A.C.W. Finkelstein *An Analysis of the Requirements Traceability Problem* Proc. 1<sup>st</sup> Int. Symp. on Requirements Engineering, Colorado Springs, IEEE Press, 1994, S. 94-101
- [Griffel98] Frank Griffel *Componentware — Konzepte und Techniken eines Softwareparadigmas* dpunkt-Verlag, Heidelberg, 1998
- [Grimm95] Klaus Grimm *Systematisches Testen von Software — Eine neue Methode und eine effektive Teststrategie* GMD-Bericht Nr. 251, R. Oldenbourg Verlag, München, zugl. Dissertation, Fachbereich Informatik, TU Berlin, 1995
- [GutHor78] J.V. Guttag, J.J. Horning *The Algebraic Specification of Abstract Data Types* Acta Informatica, Jahrg. 10, 1978, S. 27-52
- [HamCha94] Michael Hammer, James Champy *Business Reengineering — Die Radikalkur für das Unternehmen* Campus Verlag, Frankfurt, 1994
- [HMGF92] Mary Jean Harrold, John D. McGregor, Kevin J. Fitzpatrick *Incremental Testing of Object-Oriented Class Structures* Proc. Int. Conf. on Software Engineering, ACM, Mai 1992, S. 68-80
- [HarNaa96] David Harel, Amnon Naamad *The STATEMATE Semantics of Statecharts* ACM Transactions on Software Engineering and Methodology Jahrg. 5, Nr. 4, Okt. 1996
- [Hasselbring98] Wilhelm Hasselbring *Erfahrungen mit dem Einsatz von anwendungsspezifischen Piktogrammen zur partizipativen Anforderungsanalyse* Informatik Forschung und Entwicklung, 13. Jahrg., Nr. 4, 1998, S. 217-226
- [Hatton98] Les Hatton *Does OO Sync with How We Think?* IEEE Software, Mai/Juni 1998, S. 45-54
- [HDK93] P. Hsia, A. Davis und D. Kung *Status Report: Requirements Engineering* IEEE Software, Jahrg. 10, Nr. 6, 1993, S. 75-79

- [Henderson96] Brian Henderson-Sellers *Object-Oriented Metrics — Measures of Complexity* Prentice Hall, Upper Saddle River, New Jersey, 1996
- [Hetzel88] Bill Hetzel *The Complete Guide to Software Testing* 2. Aufl., QED Information Sciences, Wellesley, Mass., 1988
- [HitKap98] Martin Hitz, Gerti Kappel *Developing with UML — Some Pitfalls and Workarounds* in: Proc. UML'98: Beyond the Notation, J. Bezivin and P.-A. Muller (Hrsg.), 1<sup>st</sup> Int. Workshop on UML, Mulhouse, France, Springer LNCS, 1998
- [HKR+97] Brigid Haworth, Colin Kirsopp, Marc Roper, Martin Shepperd, Steve Webster *Towards the Development of Adequacy Criteria for Object-Oriented Software* Proc. 5<sup>th</sup> euroSTAR 97, Edinburgh, Scotland, Nov. 1997, S. 417-427
- [HomVos97] Andreas Homrighausen, Josef Voss *Tool support for the model-based development of interactive applications — The FLUID approach* Proc. 4<sup>th</sup> Eurographics Workshop on Design, Specification and Verification of Interactive Systems, Granada, Espagne, 1997
- [HOR98] A.H.M. ter Hofstede, M.E. Orlowska, J. Rajapaske *Verification Problems in Conceptual Workflow Specifications* Data & Knowledge Engineering, Jahrg. 24, 1998, S. 239-256
- [HsiKun97] Pei Hsia, David Kung *Software Requirements and Acceptance Testing* Annals of Software Engineering, Jahrg.3, 1997, S. 291-317
- [Hürsch95] Walter L. Hürsch *Maintaining Consistency and Behavior in Object-Oriented Systems during Evolution* Dissertation, Northeastern University, 1995
- [Hurlbut98] Russel R. Hurlbut *Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models* Ph.D. Thesis, Illinois Institute of Technology, Chicago, USA, 1998
- [IEEE610.90] IEEE *Standard Glossary of Software Engineering Terminology* IEEE Standard 610, IEEE, New York, 1990
- [IEEE829.83] IEEE *Standard for Software Test Documentation* IEEE Standard 829, IEEE, New York, 1983 (bestätigt 1990)
- [IEEE830.93] IEEE *Guide to Software Requirements Specifications* IEEE Standard 830, IEEE, New York, 1993
- [IEEE1233.96] IEEE *Guide for Developing System Requirements Specifications* IEEE Standard 1233, IEEE, New York, 1996

- 
- [ISO9001-3.92] DIN/ISO 9000 Teil 3 *Qualitätsmanagement- und Qualitätssicherungsnormen — Leitfaden für die Anwendung von ISO 9001 auf die Entwicklung, Lieferung und Wartung von Software* Beuth Verlag, Berlin, 1992
- [ITU93] ITU Telecommunications Standard Sector Z.120 *Message Sequence Charts* ITU, 1993
- [Jablonski95] Stefan Jablonski *Workflow Management Systeme* TAT 9, Thomson Publishing, Bonn, 1995
- [JBR99] Ivar Jacobson, Grady Booch, James Rumbaugh *The Unified Software Development Process* Addison-Wesley/acm Press, Reading, Mass., 1999
- [JCJ+92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard *Object-Oriented Software Engineering* Addison-Wesley/acm Press, Reading, Mass., 1992
- [Jacobson95] Ivar Jacobson *The Use Case Construct in Object-Oriented Software Engineering* in: [Carroll95] S. 309-336
- [JGJ97] Ivar Jacobson, Martin Griss, Patrik Jonsson *Software Reuse* Addison-Wesley/acm Press, Reading, Mass., 1997
- [Johnson97] Ralph E. Johnson *Frameworks = Components + Patterns* Communications of the ACM, Jahrg. 40, Nr. 10, Okt. 1997, S. 39-42
- [JorEri94] Paul C. Jorgensen, Carl Erickson *Object-Oriented Integration Testing* Communications of the ACM, Jahrg. 37, Nr. 12, Sep. 1994, S. 30-38
- [Jüttner93] Peter Jüttner *Testen Objektorientierter Software* Dissertation, Universität Innsbruck, Dez. 1993
- [Kahlbrandt98] B. Kahlbrandt *Software Engineering: Objektorientierte Softwareentwicklung mit der UML* Springer Verlag, Berlin, 1998
- [KHG98] David C. Kung, Pei Hsia, Jerry Gao *Testing Object-Oriented Software* IEEE Computer Soc. Press, Los Alamitos, Calif., 1998
- [KniMye93] John C. Knight, E. Ann Myers *An Improved Inspection Technique* Communications of the ACM, Jahrg. 36, Nr. 11, 1993, S. 51-61
- [Kösters97] Georg Kösters *Requirements Engineering für GIS-Applikationen* Dissertation, Fachbereich Informatik, FernUniversität Hagen, 1997
- [KPS95] Georg Kösters, Bernd-Uwe Pagel, Hans-Werner Six *Software Engineering II* Kurs 1794, FernUniversität Hagen, 1995

- [KPS96] Georg Kösters, Bernd-Uwe Pagel, Hans-Werner Six *GeoOOA: Object-Oriented Analysis for Geographic Information Systems* Proc. of the 2<sup>nd</sup> International Conference on Requirements Engineering, Colorado Springs, 1996
- [KPW97] Georg Kösters, Bernd-Uwe Pagel, Mario Winter *Coupling Use Cases and Class Models* in: Andy Evans, Kevin Lano (Hrsg.) Proc. of BCS FACS/EROS Workshop on making object-oriented methods more rigorous, London, GB, Juni 1997
- [KPR+97] Georg Kösters, Bernd-Uwe Pagel, Thomas de Ridder, Mario Winter *Animated Requirements Walkthroughs Based on Business Scenarios* Proc. 5<sup>th</sup> euroSTAR 97, Edinburgh, Scotland, Nov. 1997 (CD-ROM)
- [KPW98] Georg Kösters, Bernd-Uwe Pagel, Mario Winter *Kombinierte Validierung von Use Cases und Klassenmodellen* Proc. Modellierung'98, Münster, März 1998, S. 117-121
- [Kruchten99] Philippe Kruchten *Der „Rational Unified Process“* OBJEKTspektrum 1/99, Jan./Feb. 1999, S. 38-42
- [KSV96] Georg Kösters, Hans-Werner Six, Josef Voss *Combined Analysis of User Interface and Domain Requirements* Proc. 2<sup>nd</sup> Int. Conf. on Requirements Engineering, Colorado Springs, 1996, S. 307-33
- [KWR96] Anita Krabbel, Ingrid Wetzel, Sabine Ratuski *Objektorientierte Analysetechniken für übergreifende Aufgaben* Proc. Softwaretechnik 96, Koblenz, in: GI Softwaretechnik-Trends, 16. Jahrg., Heft 3, Sep. 1996
- [LamWil98] Axel van Lamsweerde, L. Willemet *Inferring Declarative Requirements Specifications from Operational Scenarios* IEEE Transactions on Software Engineering, Special Issue on Scenario Management, Jahrg. 24, Nr.12, Dez. 1998, S.1089-114
- [Lauesen98] Søren Lauesen *Real-Life Object-Oriented Systems* IEEE Software, März/April 1998, S. 76-83
- [Leveson95] Nancy G. Leveson *Safeware — System Safety and Computers* Addison-Wesley, Reading, Mass., 1995
- [Liggesmeyer96] Peter Liggesmeyer *Quantitative Bewertung von Software-Prüfverfahren durch unscharfe Logik* Proc. Softwaretechnik'96, zugl. Softwaretechnik-Trends, 16. Jahrg., Heft 3, 1996, S. 81-88
- [Lions97] J. L. Lions *ARIANE 5 Flight 501 Failure Report* Paris, 19 Juli 1996, <http://sspg1.bnsc.rl.ac.uk/Share/ISTP/ariane5rep.htm>



- 
- [LisZil75] Barbara H. Liskov, Stephen N. Zilles *Specification Techniques for Data Abstractions* IEEE Transactions on Software Engineering, Jahrg. 1, Nr. 1, 1976, S. 7-19
- [Lyu96] M. R. Lyu (Hrsg.) *Software Reliability Engineering* IEEE Computer Society Press/Mc Graw Hill, New York, 1996
- [Maring96] Blayne Maring *Object-Oriented Development of Large Applications* IEEE Software, Mai 1996, S. 33-40
- [McDMcG94] Robert McDaniel, John D. McGregor *Testing the Polymorphic Interactions between Classes* Technical Report TR-94-103 Clemson University, 1994
- [McGreKor94] John D. McGregor, Timothy D. Korson *Integrated Object-Oriented Testing and Development Processes* Communications of the ACM, Jahrg. 37, Nr. 12, Sep. 1994, S. 59-77
- [McGregor97] John D. McGregor *Planning for Testing* Journal of Object-Oriented Programming, Feb. 1997
- [McGregor99] John D. McGregor *If the Devil Is in the Details, Then What Is in the Plan?* Journal of Object-Oriented Programming, Jan. 1999, S. 16-21+74
- [MCP+98] N. Maiden, M. Cisse, H. Perez, D. Manuel *CREWS Validation Frames: Patterns for Validating System Requirements* CREWS Report 98-29, 1998
- [MGMS96] John D. McGregor, Brian A. Malloy, Rebecca L. Siegmund *A Comprehensive Program Representation of Object-Oriented Software* Annals of Software Engineering, Jahrg. 2, 1996, S. 51-91
- [Meyer85] Bertrand Meyer *On Formalism in Specifications* IEEE Software, Jahrg. 2, Nr. 1, Jan. 1985, S. 6-26
- [Meyer97] Bertrand Meyer *Object-Oriented Software Construction* 2. Auflage, Prentice Hall, Upper Saddle River, New Jersey, 1997
- [MMB+96] F. Macdonald, J. Miller, A. Brooks, M. Roper, M. Wood *Applying Inspection to Object-Oriented Code* Software Testing, Verification, and Reliability, Jahrg. 6, 1996, S. 61-82
- [MMM+97] N. Maiden, S. Minocha, K. Manning, M. Ryan *A Software Tool and Method for Scenario Generation and Use* Proc. 3rd International Workshop on RE: Foundation for SW Quality, Barcelona, Spain, 1997
- [Mohl97] Gabriele Mohl *Regressionstesten objektorientierter Software* Diplomarbeit, Praktische Informatik III, FernUniversität Hagen, 1997

- [MSL96] Monika Müllerburg, Andreas Spillner, Peter Liggesmeyer (Hrsg.) *Test, Analyse und Verifikation von Software — Aus der Arbeit der FG 2.1.7 TAV der GI R.* Oldenbourg Verlag, München/Wien, 1996
- [MülPoe98] Peter Müller, Arnd Poetzsch-Heffter *Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur* in: C. H. Cap. (Ed.): JIT '98 Java-Informations-Tage 1998, Informatik Aktuell, Springer-Verlag, 1998
- [MülWie98] Uwe Müller, Thomas Wiegmann „*State of the practice*“ *der Prüf- und Testprozesse in der Softwareentwicklung — Ergebnisse einer empirischen Untersuchung bei Softwareunternehmen in Deutschland* Technischer Bericht, Universität Köln, März 1998
- [Myers79] Glenford J. Myers *The Art of Software Testing* J. Wiley & Sons, New York, 1979
- [Nuseibeh97] Bashar Nuseibeh *Ariane 5: Who Dunnit?* IEEE Software, Mai/Juni 1997, S. 15-16
- [OMG97] OMG *Unified Modeling Language Specification V. 1.1*, OMG, Sep. 1997
- [OMG-AT98] OMG *UML Task Force Activity Diagram Issues* OMG, Juli 1998
- [OMG99] OMG *Unified Modeling Language Specification V. 1.3 alpha R2*, OMG, Jan. 1999
- [Opdyke92] William F. Opdyke *Refactoring Object-Oriented Frameworks* PhD. Dissertation, University of Illinois, Urbana-Champaign, 1992
- [Oshana98] Robert S. Oshana *Tailoring Cleanroom for Industrial Use* IEEE Software, Nov./Dez. 1998, S. 46-55
- [Overbeck94] Jan Overbeck *Integration Testing for Object-Oriented Software* Dissertation, TU Wien, 1994
- [Paech98] Barbara Paech *Plädoyer für ein einheitliches Grundgerüst bei der System- und Softwaremodellierung* Proc. Modellierung'98, Münster, März 1998, S. 9-14
- [PagSix94] Bernd-Uwe Pagel, Hans-Werner Six *Software Engineering I* Addison Wesley, Bonn, 1994
- [PagWin96] Bernd-Uwe Pagel, Mario Winter *Towards Pattern-Based Tools* EuroPLoP'96 Conf. Proc., Irsee, 1996  
<http://www.informatik.fernuni-hagen.de/import/pi3/abstracts/pi3.EuroPLoP96.html>
- [PalSch94] Jens Palsberg, Michael I. Schwarzbach *Object-Oriented Type Systems* J. Wiley & Sons, Chichester, 1994

- 
- [Parnas72] David L. Parnas *A Technique for the Specification of Software Modules with Examples* Communications of the ACM, Jahrg. 15, Nr. 1, 1972, S. 330-336
- [ParWei85] David L. Parnas and David M. Weiss *Active Design Reviews: Principles and Practices* Proc. 8. International Conference on Software Engineering, IEEE Computer Society Press, Aug. 1985, S. 132-136
- [Paulk91] Mark C. Paulk et Al. *Capability Maturity Model for Software* Technical Report, CMU/SEI-91-TR-024, Software Engineering Institute, 1991
- [Paulk95] Mark C. Paulk *How ISO 9001 Compares with the CMM* IEEE Software, Jan. 1995, S. 74-83
- [PerKai90] Dewayne E. Perry, Gail E. Kaiser *Adequate Testing and Object-Oriented Programming* Journal of Object-Oriented Programming, Jahrg. 3, Nr. 2, Jan./Feb. 1990, S. 13-19
- [PinGog96] Francisco A.C. Pinheiro, Joseph A. Goguen *An Object-Oriented Tool for Tracing Requirements* IEEE Software, März 1996, S. 52-64
- [Poetzsch91] Arnd Poetzsch-Heffter *Logic-Based Specification of Visibility Rules* Proc. 3<sup>rd</sup> Int. Symp. of Programming Language Implementation and Logic Programming, LNCS 528, Springer, München, 1991, S. 63-74
- [Poetzsch97] Arnd Poetzsch-Heffter *Specification and Verification of Object-Oriented Programs* Habilitationsschrift, TU München, Jan. 1997
- [PohHau97] Klaus Pohl, Peter Haumer *Modelling Contextual Information about Scenarios* Proc. 3rd International Workshop on RE: Foundation for SW Quality, Barcelona, Spain, 1997
- [Pohl96] Klaus Pohl *Process-Centered Requirements Engineering* J. Wiley & Sons/Research Studies Press Ltd., Somerset, England, 1996
- [Pol98] Martin Pol *The Future of Testing: What Should We Prepare For?* Keynote Presentation at 6<sup>th</sup> euroSTAR 98, München, Dez. 1998
- [Poston87] Robert Poston *Preventing the Most Probable Errors in Requirements* IEEE Software, Jahrg. 4, Nr. 9, Sep. 1987, S. 81-83
- [Poston94] Robert Poston *Automated Testing from Object Models* Communications of the ACM, Sep. 1994, S. 48-58
- [Poston96] Robert Poston *Automating Specification Based Software Testing* IEEE Computer Soc. Press, Los Alamitos, Calif., 1996

- [Poston98] Robert Poston *Making Test Cases from Use Cases Automatically* Proc. Quality Week Europe 98, Brüssel, Belgien, 1998
- [PTA94] Colin Potts, Kenji Takahashi, Annie I. Anton *Inquiry-based requirements analysis* IEEE Software, Nr. 3, März 1994, S. 21-32.
- [PVB95] Adam A. Porter, Lawrence G. Votta, Victor R. Basili *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment* IEEE Transactions on SE, Jahrg. 21, Nr. 6, Juni 1995, S. 563-575
- [RamEdw93] Balasubramaniam Ramesh, Michael Edwards *Issues in the Development of a Requirements Traceability Model* Proc. 1<sup>st</sup> IEEE Int. Conf. on Requirements Engineering (RE93), San Diego, Los Alamitos, Jan. 1993, S. 256-259
- [RAB96] Björn Regnell, Michael Andersson, Johan Bergstrand *A Hierarchical Use Case Model with Graphical Representation* Proc. ECBS'96, IEEE Int. Workshop on Engineering of Computer-Based Systems, IEEE Press, März 1996
- [RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen *Object-Oriented Modeling and Design* Prentice Hall, Englewood Cliffs, New Jersey, 1991
- [RegRun98] Björn Regnell, Per Runeson *Combining Scenario-based Requirements with Static Verification and Dynamic Testing* Proc. REFSQ'98, 4<sup>th</sup> Int. Workshop on RE: Foundation for SW Quality, Pisa, Italy, 1998
- [Riedemann97] Eike Hagen Riedemann *Testmethoden für sequentielle und nebenläufige Software-Systeme* B.G. Teubner, Stuttgart, 1997
- [Ritter2000] Thorsten Ritter *Entwurf und Implementation eines Rahmenwerks für persistente Objekte* Diplomarbeit, Praktische Informatik III, FernUniversität Hagen, erscheint vor. 2000
- [RKW95] Björn Regnell, Kristofer Kimbler, Anders Wesslén *Improving the Use Case Driven Approach to Requirements Engineering* Proc. 2<sup>nd</sup> IEEE Int. Symposium on Requirements Engineering, York, UK, 1995, S. 40-47
- [Rogotzki97] Jens Rogotzki *Erzeugung von Anwendungsdaten aus GeoOOA-Modellen* Diplomarbeit, Praktische Informatik III, FernUniversität Hagen, 1997
- [RolAch98] Colette Rolland, Camille Ben Achour *Guiding the Construction of Textual Use Case Specifications* Data & Knowledge Engineering, Jahrg. 25, Nr. 1-2, März 1998, S. 125-160
- [Rosson99] M. B. Rosson *Integrating Development of Task and Object Models* Communications of the ACM, Jahrg. 42, Nr. 1, Jan. 1999, S. 49-56

- 
- [Roy90] R. Roy *A Primer on the Taguchi Method* Van Nostrand Reinhold, New York, 1990
- [Rose98] Rational Rose 98, Rational Software Corporation, Cupertino, CA, 1998
- [Rumbaugh94] James Rumbaugh *Getting started: using use cases to capture requirements* Journal of Object-Oriented Programming Jahrg. 8, Nr. 5, Sep. 1994, S. 8-12+23
- [Rumpe96] Bernhard Rumpe *Formale Methodik des Entwurfs verteilter objektorientierter Systeme* Herbert Utz Verlag Wissenschaft, zugl. Dissertation, Technische Universität München, 1996
- [Rüppel96] Peter Rüppel *Ein generisches Testwerkzeug für den objektorientierten Softwaretest* Dissertation, Fachbereich Informatik — Softwaretechnik, TU Berlin, 1996
- [Scheer98a] August-Wilhelm Scheer *ARIS — Vom Geschäftsprozeß zum Anwendungssystem* 3. Aufl., Springer, Berlin, 1998
- [Schulte97] Egbert Schulte *Ein Transformations-Framework für GeoOOA-Modelle* Diplomarbeit, Praktische Informatik III, FernUniversität Hagen, 1997
- [Shaw94] Mary Shaw *Comparing Architectural Design Styles* IEEE Software, Nov. 1995, S. 27-41
- [ShaGar96] Mary Shaw, David Garlan *Software Architecture — Perspectives of an Emerging Discipline* Prentice Hall, Upper Saddle River, New Jersey, 1996
- [Siegel96] Shel Siegel *Object Oriented Software Testing — A Hierarchical Approach* J. Wiley & Sons, New York, 1996
- [SieNew94] Ernst Siepmann, A. Richard Newton *TOBAC: A Test Case Browser for Testing Object-Oriented Software* Proc. ISSTA 94, Washington, 1994, S. 154-168
- [Sneed95] Harry M. Sneed *Objektorientiertes Testen* Informatik Spektrum, Jahrg. 18, Nr. 1, Feb. 1995, S. 6-12
- [Sneed96] Harry M. Sneed *Ein objektorientiertes Testverfahren* in: [MSL96], S. 25-48
- [Spillner90] Andreas Spillner *Dynamischer Integrationstest modularer Softwaresysteme* Dissertation, Universität Bremen, 1990
- [StrHör98] Friedrich Strauß, Thimo Hörnke *Testfallentwicklung und -automatisierung in großen OO-Projekten* Proc. Softwaretechnik'98, Paderborn, 1998
- [TheGot98] JoachimThees, Reinhard Gotzhein *The eXperimental Estelle Compiler — Automatic Generation of Implementations from Formal Specifications* in: M. Ardis

- (Edt.), Proc. 2<sup>nd</sup> Workshop on Formal Methods in Software Practice (FMSP'98), Clearwater Beach, Florida, USA, März 1998
- [Tichy94] Walter F. Tichy (Ed.) *Trends in Software — Configuration Management* J. Wiley & Sons, Chichester, 1994
- [Uhl97] R. Uhl *Qualifizierte Suche und Konsistenzprüfungen in GEOOOA-Modellen* Diplomarbeit, Praktische Informatik III, FernUniversität Hagen, 1997
- [VosGrH93] Gottfried Vossen, Margaret Groß-Hardt *Grundlagen der Transaktionsverarbeitung* Addison-Wesley, Bonn, 1993
- [VosNen98] Josef Voss, Dietmar Nentwig *Graphische Benutzungsschnittstellen — Modelle, Techniken und Werkzeuge der User-Interface-Gestaltung* Hanser, München, 1998
- [Voss97] Josef Voss *Vorgehensweisen und Werkzeuge für ein Ineinandergreifen von Modellierung und Prototyping* Proc. PB'97, Prototypen für Benutzungsschnittstellen, Fachgespräch der GI-Fachgruppe 2.1.2, Paderborn, 1997
- [VW98] VISUALWORKS, Release 3.0 ParkPlace Systems, Inc., Sunnyvale, CA, 1998
- [WalNer95] Kim Walden, Jean-Marc Nerson *Seamless Object-Oriented Software Architecture* Prentice Hall, Englewood Cliffs, New Jersey, 1995
- [WBW+90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener *Designing Object-Oriented Software* Prentice Hall, Englewood Cliffs, New Jersey, 1990
- [WBM96] David A. Wheeler, Bill Brykczynski, Reginald N. Meeson, Jr. *Software Inspektion — An Industrial Best Practice* IEEE Press, Los Alamitos, 1996
- [Wegner97] Peter Wegner *Why Interaction is More Powerfull than Algorithms* Communications of the ACM, Jahrg. 40, Nr. 5, May 1997
- [WegGol99] Peter Wegner, Dina Goldin *Interaction as a Framework for Modeling* Technical Report, Brown University, erscheint in LNCS #1565, Springer, Berlin, April 1999
- [Weihrauch87] Klaus Weihrauch *Computability* EATCS Monographs on Theoretical Computer Science Nr. 9, Springer, Berlin, 1987
- [Weyucker86] Elaine J. Weyuker *Axiomatizing Software Test Data Adequacy* IEEE Transactions on Software Engineering, Nr. 12, Dez. 1986, S. 1128-1138
- [WFMC97] Workflow Management Coalition *The Workflow Reference Model* Dok. Nr. TC00-1003 Issue 1.1, WFMC, Brüssel, 1997

- 
- [Wiegers98] Karl E. Wiegers *Read my Lips: No New Models!* IEEE Software, Sep./Okt. 1998, S. 10-13
- [Winter97] Mario Winter *Reviews in der objekt-orientierten Softwareentwicklung* TAV 10 6/ 7.03.1997 Böblingen in: GI Softwaretechnik-Trends, 17. Jahrg., Heft 2, Mai 1997, S. 6-11
- [Winter98] Mario Winter *Managing Object-Oriented Integration and Regression Testing* Proc. 6<sup>th</sup> euroSTAR 98, München, Dez. 1998, S. 189-200
- [Winter99] Mario Winter *Zur Komplexität der Generierung von Daten für objektorientierte Anwendungen* Technischer Report, FernUniversität Hagen, 1999
- [WPJ+98] Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, Peter Haumer *Scenarios in System Development: Current Practice* IEEE Software, März/April 1998, S. 34-45
- [Wulf2000] Cornelia Wulf *Validierung einer Methode zur Integration statischer und dynamischer Sichten in der objektorientierten Anforderungsermittlung* Diplomarbeit, Praktische Informatik III, FernUniversität Hagen, erscheint vor. Jan. 2000
- [Yourdon89] Edward Yourdon *Modern Structured Analysis* Prentice Hall, Englewood Cliffs, New Jersey, 1989
- [ZRL96] Sean Zhang, Barbara G. Ryder, William Landi *Program Decomposition for Pointer Aliasing: A Step towards Practical Analyses* Proc. 4<sup>th</sup> Symposium on the Foundations of Software Engineering, Okt. 1996
- [Ziegler96] Jürgen Ziegler *Eine Vorgehensweise zum objektorientierten Entwurf graphisch-interaktiver Informationssysteme* IPA-IAO-Forschung und Praxis, Bd. 240, Springer, Berlin, 1997, zugl. Dissertation, Univ. Stuttgart, 1996
- [Züllighoven98] Heinz Züllighoven *Das objektorientierte Konstruktionshandbuch* D-Punkt Verlag, 1998





# Anhang

Im Anhang präzisieren wir zunächst das im Text nur skizzierte Klassen-Botschaftsdiagramm für die Anforderungsspezifikation ( $KBD_A$ ).

Danach detaillieren wir den Begriff der internen Interaktion in objektorientierten Programmen und geben darauf aufbauend die Definition und einen Algorithmus zur Generierung des Klassen-Botschaftsdiagramms für die Implementation ( $KBD_I$ ) an. Hierbei beziehen wir uns auf die objektorientierte Programmiersprache Java.

Zusätzlich haben wir noch die Syntax der im Rahmen der Arbeit entwickelten Test-Beschreibungssprache angefügt.

# Anhang A

## Klassen-Botschaftsdiagramme

In diesem Anhang geben wir Algorithmen zur Generierung der in Teil IV der Arbeit informell eingeführten Klassen-Botschaftsdiagramme an. Wir beginnen mit dem Klassen-Botschaftsdiagramm für die Anforderungsspezifikation ( $\text{KBD}_A$ , A.1). Zur Motivation des Klassen-Botschaftsdiagramms für die Implementation ( $\text{KBD}_I$ ) diskutieren wir in Abschnitt A.2 die möglichen internen Interaktionen in objektorientierten Anwendungen und betrachten dann in Abschnitt A.3 die Generierung des  $\text{KBD}_I$  für eine Implementation in der Programmiersprache Java. Wir beenden diesen Anhang in Abschnitt A.4 mit einer kurzen Betrachtung zur Ausdruckstärke des  $\text{KBD}_I$ .

### A.1 Klassen-Botschaftsdiagramme für die Anforderungsspezifikation

Im Fokus des Klassen-Botschaftsdiagramms für die Anforderungsspezifikation liegen die Operationen der Domänenklassen sowie die Aufrufe bzw. „Benutzt“-Beziehungen zwischen ihnen. Wir gehen davon aus, dass insbesondere die komplexen Operationen der Domänenklassen umfassend mit Episoden geprüft wurden (Kapitel 12). Wir treffen zunächst folgende Vereinbarungen:

- Eine „Benutzungsbeziehung“ manifestiert sich als Element der Nachfolgerschritt-Relation auf den Episodenschritten (s. Definition 11.2); wir nennen die dem Vorgängerschritt zugeordnete Operation die *benutzende Operation*.
- Die im Kontext einer Instanz der ausführenden Klasse des Folgeschritts ausgeführte Operation nennen wir die *benutzte Operation*.

Im Hinblick auf eine effiziente Automatisierung beschränken wir uns auf ein graphentheoretisch definiertes Modell. Knoten repräsentieren die in den Klassen des Klassenmodells defi-

nierten Operationen, Kanten stellen Benutzungsbeziehungen zwischen Operationen dar. Zusätzlich merken wir uns, ob zwei oder mehrere Kanten einer aufgrund der Vererbung und des Polymorphismus „dynamisch gebundenen“ Benutzung entsprechen. Wir präzisieren dies in der folgenden Definition.

**Definition 22.1** Sei  $\mathbf{K}$  die Menge aller Domänen-Klassen im Klassenmodell einer Anforderungsspezifikation. Das *Klassen-Botschaftsdiagramm für die Anforderungsspezifikation* ( $\text{KBD}_A$ ) ist ein gerichteter, kantenmarkierter Graph  $(V, E, P)$  mit der Markenmenge  $L_A$ . Hierbei treffen wir folgende Vereinbarungen:

- $V \subseteq \text{OPS}$  repräsentiert die Menge aller expliziten *Methodendefinitionen* in  $\mathbf{K}$ ,
- $L_A = \{\text{message, inheritance}\}$  ist die Menge der *Kantenmarkierungen* mit der Bedeutung:
  - message: Benutzung einer Operation („Botschaft“),
  - inheritance: es handelt sich um eine Vererbungskante.
- $E \subseteq V \times L_A \times V$  repräsentiert die Menge aller in der Anwendung möglichen Operationsbenutzungen vereinigt mit den aufgrund von Redefinitionen zwischen Operationen bestehenden „Vererbungsbeziehungen“. Wir bezeichnen mit  $E_B \subseteq E$ ,  $x \in E_B \Leftrightarrow x = (m, j, n)$  mit  $j = \text{message}$  die Menge der *Botschaftskanten* des Klassen-Botschaftsdiagramms. Die Menge  $E_V \subseteq E$ ,  $x \in E_V \Leftrightarrow x = (m, \text{inheritance}, n)$  nennen wir die *Vererbungskanten* des Klassen-Botschaftsdiagramms; es ist  $(m, \text{inheritance}, n) \in E_V$ , wenn Operation  $m$  die Operation  $n$  redefiniert und zusätzlich gilt
 
$$\forall (b::m, \text{inheritance}, a::n) \in E_V: \forall k \in \mathbf{K}: b < k < a \Rightarrow \neg(m \in k.\text{operations}).$$
 Es gelten  $E_B \cup E_V = E$  und  $E_B \cap E_V = \emptyset$ , d.h.  $E_B$  und  $E_V$  partitionieren  $E$ .  $E_V$  ist irreflexiv, asymmetrisch und injektiv, d.h. eine Operation redefiniert im Rahmen einer Vererbungshierarchie höchstens eine andere Operation.
- Die Relation  $P \subseteq E_B \times E_B$  berücksichtigt die aufgrund der Vererbung und Redefinition von Methoden möglichen dynamisch gebundenen Methoden-„Aufrufe“. Stehen Botschaftskanten zueinander in der Relation  $P$ , so wird bei jeder Ausführung des Aufrufs genau eine der Zieloperationen benutzt.  $P$  ist reflexiv und transitiv.

□

Zur Generierung des Klassen-Botschaftsdiagramms aus Episoden erinnern wir daran, dass die „Benutzungsbeziehungen“ zwischen Operationen in der Nachfolgerschritt-Relation der Episoden protokolliert sind. Diese Tatsache liegt den im Folgenden angegebenen Generierungsregeln für das  $\text{KBD}_A$  zugrunde. Bei den einzelnen Punkten sind in Klammern die entsprechenden Zeilen des Algorithmus *Generiere KBDA* (Abb. A.1) angegeben.

1. Einer in Klasse  $k$  im Klassenmodell definierten Operation  $o$  entspricht ein Knoten mit dem qualifizierten Namen<sup>1</sup> ' $k::o$ ' des KBD. (Zeilen 2 - 4)

### Algorithmus Generiere KBDA

```

Eingabe: Domänen-Klassen  $K$ , Episoden  $EPS$ ;
Ausgabe: Klassen-Botschaftsdiagramm  $KBD_A = (V, E, P)$ ;
/* Initialisierung */
1   $V := \emptyset; E := \emptyset; P := \emptyset;$ 
   /* Operationen-Knoten erzeugen */
2  FORALL  $k \in K$  DO                                /* Alle Klassen */
3      FORALL  $k::o \in k.operations$  DO
4           $V := V \cup \langle k::o \rangle;$ 
   /* Kanten für die direkt in den Episoden protokollierten Benutzungen erzeugen */
5  FORALL  $e \in EPS$  DO                                /* Alle Episoden */
6      FORALL  $s_1, s_2 \in e.ES$  DO                    /* Alle Episodenschritte */
7          IF  $s_2 \in \sigma(s_1)$  THEN {                /*  $s_2$  ist Nachfolgeschritt von  $s_1$  */
8               $k_1 := AK(s_1); k_2 := AK(s_2);$           /* Klassen der ausführenden Objekte */
9               $o_1 := WO(s_1); o_2 := WO(s_2);$           /* Ausgeführte Operationen */
10              $E := E \cup (\langle k_1::o_1 \rangle, \langle k_2::o_2 \rangle, 'message')$ 
           }; /* IF */
   /* Vererbung und Polymorphismus berücksichtigen */
11  CALL PolyKBDA;
12  RETURN ( $V, E, P$ )
END Generiere KBDA.

```

Abb. A.1 Algorithmus Generiere KBD<sub>A</sub>

2. Ist für zwei Episodenschritte  $s_1$  und  $s_2$  mit  $AK(s_1) = \mathbf{k}_1$  sowie  $WO(s_1) = o_1$  und  $AK(s_2) = \mathbf{k}_2$  sowie  $WO(s_2) = o_2$  der Episodenschritt  $s_2$  Nachfolgerschritt von  $s_1$ , gilt also  $s_2 \in \sigma(s_1)$ , so fügen wir im KBD eine *Botschaftskante* (Marke message) von  $\mathbf{k}_1::o_1$  nach  $\mathbf{k}_2::o_2$  ein. (Zeilen 5 - 10)
3. Ist eine Klasse  $\mathbf{k}_2$  eigentliche Unterklasse einer Klasse  $\mathbf{k}_1$  und überschreibt Operation  $\mathbf{k}_2::a$  die Operation  $\mathbf{k}_1::a$ , so fügen wir eine *Vererbungskante* (Marke inheritance) vom Knoten  $\mathbf{k}_2::a$  zu Knoten  $\mathbf{k}_1::a$  in das KBD ein (Zeilen 1 - 5 in Funktion PolyKBD<sub>A</sub>, Abb. A.2). Zusätzlich duplizieren wir in diesem Fall alle Kanten, die in Knoten  $\mathbf{k}_1::a$  des KBD enden, für Knoten  $\mathbf{k}_2::a$ . Diese duplizierten und „umgeleiteten“ Kanten repräsentieren explizit die Tatsache, dass Benutzungen der Operation  $a$  dynamisch an Objekte sowohl der Klasse  $\mathbf{k}_1$  als auch der Klasse  $\mathbf{k}_2$  gebunden werden können. (Zeilen 6 - 9 in Funktion PolyKBD<sub>A</sub>).

Abb. A.1 zeigt den resultierenden Algorithmus Generiere KBDA, Abb. A.2 die verwendete Funktion PolyKBD<sub>A</sub>.

---

1 Wir qualifizieren den Namen einer Methode bzw. Variablen, indem wir ihm (mit „::“ abgesetzt) den Namen der sie deklarierenden Klasse voranstellen.

### Funktion PolyKBDA

Eingabe: Klassen  $K$ , Klassen-Botschaftsdiagramm  $KBD_A = (V, E, P)$ ;  
 Ausgabe: Klassen-Botschaftsdiagramm  $KBD_A$  mit polymorphen Benutzungen;

```

1  FORALL  $k_1, k_2 \in K$  DO                                /* Alle Paare von Klassen */
2      IF  $k_2 < k_1$  THEN                                    /* Alle Vererbungsbeziehungen */
3          FORALL  $o \in k_2.\text{redefinedOperations}$  DO
4              IF  $\neg(\exists k | k_2 < k < k_1 \wedge o \in k.\text{operations})$  THEN {
5                   $E := E \cup (\langle k_1::o \rangle, \langle k_2::o \rangle, \text{'inheritance'})$  ;
6                  /* Kanten für die polymorphen Benutzungen erzeugen */
7                  FORALL  $e := (\langle k::p \rangle, \langle k_1::o \rangle, \text{'message'}) \in E$  DO {
8                       $e_2 := (\langle k::p \rangle, \langle k_2::o \rangle, \text{'message'})$  ;
9                       $E := E \cup e_2$  ;
10                      $P := P \cup \{e, e_2\}$ 
11                 }; /* FORALL */
12             } /* IF */
13 END PolyKBDA.
```

Abb. A.2 Funktion PolyKBDA

Wie betrachten noch kurz die Zeit- und Raum-Komplexität der Generierung des  $KBD_A$ .  $OPS$  bezeichne die Menge aller Operationen im Domänen-Klassenmodell,  $ES$  die Menge aller Episodenschritte im Verifikationsprotokoll.

- Die Ermittlung der Knoten (Zeilen 2 - 4) benötigt  $O(|OPS|)$  Schritte, die Benutzungskanten werden in der Zeit  $O(|ES|^2)$  eingefügt (Zeilen 5 - 10). Funktion PolyKBDA läuft in der Zeit  $O(|OPS|^2)$ . Insgesamt ergibt sich eine Zeitkomplexität von  $O(|OPS|^2 + |ES|^2)$ .
- Für die  $|OPS|$  Knoten ergibt sich für die Generierung des  $KBD_A$  eine Raumkomplexität von  $O(|OPS|^2)$ , da jeder Knoten höchstens zu  $|OPS| - 1$  anderen Knoten über Botschaftskanten und zu höchstens einem anderen Knoten über eine Vererbungskante adjazent sein kann.

Eine auf den Nachfolgerschritt-Relationen der Episoden basierende partielle Ordnung auf den Operationen im Domänen-Klassenmodell kann u.a. zur Planung der Entwicklungs- und Testtätigkeiten auf der Basis des  $KBD_A$  verwendet werden (s. [Winter98]).

## A.2 Interaktion in objektorientierten Anwendungen

In diesem Abschnitt untersuchen wir, welche internen Interaktionen in objektorientierten Anwendungen prinzipiell möglich sind. Da im Folgenden keine Verwechslungen mit externen Interaktionen zu befürchten sind, reden wir kurz von Interaktionen und meinen interne Interaktionen.

Zur Einstimmung erinnern wir zunächst an die Grundkonzepte der modularen Programmierung (vgl. [Parnas72], s. auch [PagSix94]): Daten und Funktionen, die im Kontext der Gesamtaufgabe eine besondere Affinität besitzen, werden zu Moduln (und diese wiederum zu Teilsystemen) vereinigt, die als Komponenten der Systemarchitektur jeweils eine klar abgegrenzte und präzise definierte Teilaufgabe übernehmen. Innerhalb eines Moduls bilden die Interaktionen modulinterner Operationen einen Aufrufgraph<sup>1</sup> — für jeden Operationsaufruf steht fest, welche Operation ausgeführt wird. Benutzungsbeziehungen modellieren die Interaktionen zwischen Komponenten. Sie bilden die Kanten des sogenannten Komponentengraphen, dessen Zyklentreiheit gefordert wird — es ergibt sich also eine Komponentenhierarchie. Die möglichen Interaktionen zwischen Komponenten lassen sich statisch aus den Benutzungsbeziehungen und den expliziten Importen bzw. Exporten von Operationen in bzw. aus einer Komponente ermitteln.

Zum Vergleich der Komplexität modularer und objektorientierter Anwendungen betrachten wir Abb. 22.2 (a). Im Spezialfall einer binären „Benutzungshierarchie“ mit  $n$  Knoten gibt es  $n-1$  Interaktionen. Im Gegensatz dazu ergeben sich bei einem beliebigen vollständigen schlichten „Benutzungsgraph“ oder einem „Objektnetzwerk“ mit  $n$  Knoten  $n*(n-1) = n^2 - n$  Interaktionsmöglichkeiten (Abb. 22.2 (b)).

Die Komplexität objektorientierter Software hat drei Ursachen:

1. Klassen und insbesondere Methoden haben im Vergleich zu Moduln und Prozeduren meistens eine wesentlich feinere Granularität (vgl. [Binder94][Hatton98]).
2. Dazu kommt die — im Vergleich zur in der Anzahl der Moduln meistens linear wachsenden Komplexität modularer Software — quadratisch wachsenden Anzahl von Interaktionsmöglichkeiten der Klassen bzw. ihrer Methoden in objektorientierter Software.

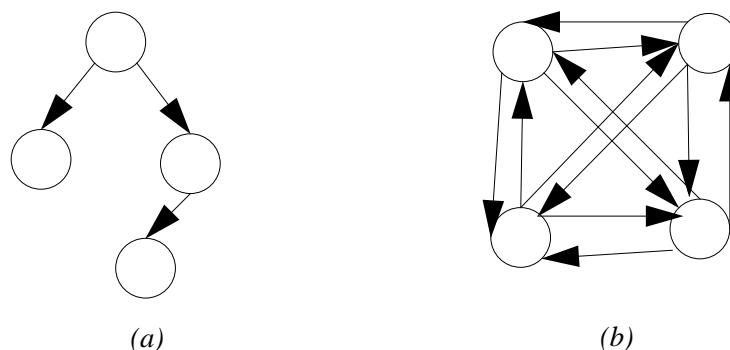


Abb. 22.2 Beziehungsgeflechte: Modular (a) und Objektorientiert (b)

<sup>1</sup> Der Aufrufgraph ist bei nicht-rekursiver Operationen azyklisch.

3. Diese führen aufgrund der Vererbung und der dynamischen Bindung sogar zu einem exponentiellen Wachstum der möglichen Interaktionen und lassen sich zudem nicht ohne größeren Aufwand statisch ermitteln (vgl. [PalSch94]).

Wir präzisieren den Begriff der (internen) *Interaktion* in objektorientierten Anwendungen folgendermaßen:

- ☐ Sendet ein Objekt bei der Ausführung einer Methode eine Botschaft an ein (nicht notwendigerweise verschiedenes) Objekt bzw. greift (lesend oder schreibend) auf eine Klassen- oder Instanz-Variable zu, so sprechen wir von einer *direkten Aufruf- bzw. Daten-Interaktion*.
- ☐ Beeinflusst eine direkte Interaktion eine nachfolgende Methodenausführung, so nennen wir dies eine *indirekte Interaktion*. Bei der hier betrachteten Klasse objektorientierter Anwendungen sind indirekte Interaktionen nur über die Manipulation von Klassen- oder Instanz-Variablen, also durch direkte Dateninteraktionen möglich, so dass wir auch von *Zustandsinteraktionen* reden.

Weiterhin unterscheiden wir, ob die Interaktion stattfindet

- ☐ innerhalb eines Objekts (Intra-Klassen/Intra-Objekt Interaktion),
- ☐ zwischen zwei Objekten der gleichen Klasse (Intra-Klassen/Inter-Objekt Interaktion),
- ☐ zwischen zwei Objekten verschiedener Klassen innerhalb einer Vererbungshierarchie, wobei die eine Unterklasse der anderen ist (und umgekehrt, Intra-Klassenhierarchie Interaktionen) oder
- ☐ zwischen zwei Objekten verschiedener Klassen, die nicht über die Vererbung verbunden sind (Inter-Klassenhierarchie Interaktion).

Wir orientieren uns bei den folgenden Betrachtungen an den sequentiellen Konstrukten der Programmiersprache Java (s. [GJS96]). Zur Vereinfachung gehen wir davon aus, dass alle Methoden sowie ggf. Klassen- bzw. Instanz-Variablen in allen Klassen einer Anwendung sichtbar sind (public). Da in Java enge Wechselwirkungen zwischen den Kapselungskonstrukten und den Regeln für die Bindung von Methoden bestehen, die im Extremfall bis hin zu Anomalien bei der dynamischen Bindung von Methodenaufrufen bzw. Botschaften an die ausgeführte Methode führen können (vgl. [MülPoe98]), betrachten wir auch den Paket-Mechanismus von Java nicht.

Die Java-Sprachspezifikation unterscheidet Klassen- und Instanz-Methoden, wobei erstere durch das Schlüsselwort **static** in der Deklaration gekennzeichnet sind. Wird von einer Unterklasse eine Klassenmethode überschrieben (*hiding*), so ist die überschriebene Methode durch Verwendung des Schlüsselworts **super** innerhalb der Methoden der Unterklasse bzw. nach einem „cast“ eines Objekts der Unterklasse in eine Instanz der Oberklasse auch „von Aussen“ (Inter-Objekt) sichtbar. Überschriebene Instanz-Methoden (*overriding*) sind im Gegensatz dazu von Aussen nicht sichtbar.

Letztendlich müssen wir noch den Unterschied zwischen Klassen- und Instanz-Variablen sowie die Effekte der gleichzeitigen Referenzierung (und Manipulation) eines Objekts über mehrere Variablen (*aliases*) berücksichtigen. Während die Modifikation einer Instanzvariablen unmittelbar nur den Zustand eines Objekts beeinflusst, wirkt sich die Modifikation einer Klassenvariablen (in Java mit dem Schlüsselwort **static** deklariert) auf alle Instanzen der Klasse (und ihrer Unterklassen) aus. Im Falle eines Alias können die aus der Transaktionstheorie bekannten Synchronisations- und Serialisierbarkeitsprobleme auftreten (vgl. z.B. [VosGrH93]).

Die Test-Problematik verschärft sich also bei der Einbeziehung von Instanzvariablen atomaren Typs hin zu Instanzvariablen vom Referenztyp ohne Aliases über Instanzvariablen mit der Möglichkeit von Aliases. Kommen darüber hinaus noch Klassenvariablen vom atomaren Typ oder sogar vom Referenztyp mit der Möglichkeit „globaler“ Aliases („Singleton“, s. [GHJ+94]) hinzu, sollten zur adäquaten Prüfung der Anwendung zusätzlich globale Kontroll- und Datenfluss- sowie Alias-Analysen durchgeführt werden (vgl. [MGMS96][ZRL96]).

### A.3 Klassen-Botschaftsdiagramme für die Implementation

Das  $KBD_A$  stellt in kompakter Art und Weise die Abhängigkeiten im Domänen-Klassenmodell zusammen. Wir streben analog dazu ein (möglichst einfaches) Test-Modell für die Implementation an, welches alle möglichen Interaktionen zwischen Objekten der Implementation unter Beachtung der Vererbungsstruktur erfasst. Hierbei betrachten wir als Interaktionen den „Botschaftsfluss“ sowie mögliche Zustandsänderungen aufgrund einer Botschaft.

Zusätzlich zur formalen Definition geben wir einen Algorithmus zur Konstruktion des  $KBD_I$  aus den Java-Quellcodes einer Anwendung sowie einige Darstellungskonventionen an und verdeutlichen die Definition an einem Beispiel.

#### Definition des $KBD_I$

Wir erweitern zunächst Definition 22.1 ( $KBD_A$ ) um die erforderlichen Elemente für die Implementation.

**Definition 22.2** Seien  $K$  die Interfaces und Klassen („Universum“) und  $OPS$  die Menge aller Operationen bzw. Methoden im Quellcode einer Anwendung. Das *Klassen-Botschaftsdiagramm* ( $KBD_I$ ) für die Implementation ist ein gerichteter, knoten- und kantenmarkierter Multigraph  $(V, E, P)$  über den Markenmengen  $L_V$  und  $L_I$  zusammen mit einer zweistelligen Relation  $P$  über  $E$ :

- $V \subseteq OPS$  ist die Menge aller expliziten *Methodendefinitionen* im Universum  $K$ .
- $L_V = \{\text{class, instance}\} \times \{\text{method, field}\}$  ist die Menge aller *Knotenmarkierungen* mit der Bedeutung:



- **class**      Element existiert einmal für alle Instanzen einer Klassen und
  - **instance**    Element existiert für jede Instanz ;
  - **method**      Knoten entspricht einer Methode,
  - **field**        Knoten entspricht einer Variablen.
- $L_I = L_A \cup \{\mathbf{this}, \mathbf{super}, \mathbf{new}, \mathbf{def}, \mathbf{uses}\} = \{\mathbf{inheritance}, \mathbf{message}, \mathbf{this}, \mathbf{super}, \mathbf{new}, \mathbf{def}, \mathbf{uses}\}$  ist die Menge aller *Kantenmarkierungen* mit der Bedeutung:
- **inheritance**, **message** wie in Definition 22.1,
  - **this**        Botschaftsaufruf ist dynamisch gebunden und selbstrekursiv,
  - **super**       Botschaftsaufruf ist statisch gebunden,
  - **new**        Botschaft führt zur Instantiierung eines neuen Objekts,
  - **def**        modifizierender Zugriff auf eine Variable und
  - **uses**       nicht-modifizierender Zugriff auf eine Variable.
- $E \subseteq V \times L_I \times V$  ist die Menge aller direkten Aufruf- und Daten-Interaktionen vereinigt mit der Menge aller zwischen Methoden bzw. Variablen (Features) bestehenden Vererbungsbeziehungen. Wir bezeichnen mit  $E_I \subseteq E$ ,  $x \in E_I \Leftrightarrow x=(m, j, n)$  mit  $j \in \{\mathbf{message}, \mathbf{this}, \mathbf{super}, \mathbf{new}\}$  die Menge der *Aufruf-Interaktionskanten* des Klassen-Botschaftsdiagramms. Für  $j \in \{\mathbf{message}, \mathbf{this}, \mathbf{super}, \mathbf{new}\}$  ist  $(m, j, n) \in E_I$ , wenn in Methode  $m$  ein Aufruf der Methode  $n$  erfolgen kann.  $E_D \subseteq E$ ,  $x \in E_D \Leftrightarrow x=(m, j, n)$  mit  $j \in \{\mathbf{def}, \mathbf{uses}\}$  bezeichnen wir als die Menge der *Daten-Interaktionskanten* des Klassen-Botschaftsdiagramms. Für  $j \in \{\mathbf{def}, \mathbf{uses}\}$  ist  $(m, j, n) \in E_D$ , wenn Variable  $m$  innerhalb der Methode  $n$  definiert wird bzw. in Methode  $m$  eine Benutzung der Variablen  $n$  erfolgen kann. Ist  $j = \mathbf{inheritance}$ , dann ist  $(m, j, n) \in E_I$ , wenn Methode  $m$  Schnittstelle und/oder Implementation von Methode  $n$  erbt bzw. die Definition der Variablen  $m$  von der Definition der Variablen  $n$  überschrieben wird.  $E_D$ ,  $E_I$  und  $E_V$  partitionieren  $E$ .  $E_V$  ist irreflexiv und asymmetrisch. Bei einfacher Vererbung ist  $E_V$  darüber hinaus auch injektiv, d.h. ein Feature redefiniert höchstens ein anderes Feature.
- Die Relation  $P \subseteq E \times E$  berücksichtigt dynamisch gebundenen Methodenaufrufe bzw. Variablenzugriffe. Stehen Kanten zueinander in der Relation  $P$ , so wird abhängig von der tatsächlichen Klasse des Zielobjekts, durch den Aufruf genau eine der den Zielknoten entsprechenden Methoden ausgeführt bzw. auf die entsprechend definierte Variable zugegriffen.  $P$  ist reflexiv und transitiv.

□

## Generierung des KBD<sub>I</sub> für Java-Klassen

Innerhalb des in Abb. 22.3 angegebenen Algorithmus Generiere KBD<sub>I</sub> beziehen wir uns auf die Syntaxregeln, insbesondere die Nicht-Terminals der Java Grammatik aus [GJS96]. Für (den Parse-Baum der) Klasse **k** ergibt z.B. der Ausdruck **k.MethodDeclarations** die Menge aller (Teilbäume der) Methodendeklarationen in **k**. Zur Vereinfachung der Darstellung gehen wir davon aus, dass Zuweisungen immer die Form „LeftHandSide = RightHandSide“ haben, wobei LeftHandSide ein Name einer (lokalen- oder Instanz-) Variablen ist und RightHandSide ein Ausdruck vom syntaktischen Typ ConditionalOrExpression (d.h. wir betrachten keine Zuweisungsketten). Wir treffen noch einige weitere Vereinbarungen und Hilfsfunktionen. Sei hierzu ml ein Methodenaufruf (MethodInvocation) innerhalb einer Klasse **k**.

- ❑ Für eine Deklaration einer Variablen *v* oder Methode *m* sei **<k::m>** die Zeichenkette mit dem qualifizierten Namen der Variablen oder Methode.
- ❑ Der Ausdruck **ml.Identifizier** liefert den Namen der aufgerufenen Methode.
- ❑ Die Funktion **PREFIX(ml)** liefert ggf. das dem Methodenaufruf vorangestellte „primary“, also eines der drei Schlüsselworte **this**, **super** und **new**. Ist **PREFIX(ml) = ε**, so handelt es sich um einen dynamisch gebundenen einfachen Methodenaufruf, bei **PREFIX(ml) = 'this'** um einen dynamisch gebundenen selbstrekursiven Methodenaufruf

### FUNCTION PolyKBD<sub>I</sub>

Eingabe: Klassen *K*, Klassen-Botschaftsdiagramm KBD<sub>I</sub> = (*V*, *E*, *P*);

Ausgabe: KBD<sub>I</sub> mit polymorphen Interaktionen;

```

1  FORALL ( k1, k2 ∈ K ) DO
2    IF ( k2 < k1 ) THEN
3      FORALL ( f ∈ k2.redefinedFields ) DO
4        IF ( ¬(∃k | k2 < k < k1 ∧ f ∈ k.Fields) ) THEN {
5          E := E ∪ (⟨k1::f⟩, ⟨k2::f⟩, 'inheritance') ;
          /* Kanten für die polymorphen Aufrufe erzeugen */
6          FORALL ( e := (⟨k::p⟩, ⟨k1::f⟩, x) ∈ E ) DO
7            IF ( x ∈ { 'message', 'this' } ) THEN {
8              e2 := (⟨k::p⟩, ⟨k2::f⟩, x) ;
9              E := E ∪ e2 ;
10             P := P ∪ { e, e2 }
            }; /* IF */
          }; /* IF */
        } /* IF */
    } /* IF */
END PolyKBDI.
```

Abb. 22.4 Funktion PolyKBD<sub>I</sub>

### Algorithmus Generiere KBD<sub>I</sub>

Eingabe: Klassen- und Interface-Deklarationen  $K$ ;  
Ausgabe: Klassen-Botschaftsdiagramm  $\text{KBD}_I = (V, E, P)$ ;

```

1   $V := \emptyset$ ;  $E := \emptyset$ ;  $P := \emptyset$ ;    /* Initialisierung */
2  FORALL  $k \in K$  DO {                    /* Operationen- und Variablen-Knoten erzeugen */
3      FORALL  $k::m \in k.\text{memberDeclarations}$  DO {
4          IF  $k::m \in k.\text{MethodDeclarations}$  THEN
5               $l := \text{'method'}$ 
6          ELSE
7               $l := \text{'field'}$ ;
8          IF "static"  $\in k::m.\text{Modifiers}$  THEN {    /* Klassen-Member */
9               $l := l, \text{'class'}$ 
10             ELSE
11                  $l := l, \text{'instance'}$ ;          /* Instanz-Member */
12              $V := V \cup (\langle k::m \rangle, l)$ 
13         } /* FORALL */
14     FORALL ( $k \in K$ ) DO                /* Kanten für die direkten Interaktionen erzeugen */
15         FORALL ( $k::m \in k.\text{MethodDeclarations}$ ) DO {
16             FORALL ( $ml \in k::m.\text{MethodInvocations}$ ) DO {
17                  $k_1 := \text{TYP}(ml)$ ;  $i_1 := ml.\text{Identifizier}$ ;  $s := \text{PREFIX}(ml)$ ;
18                 IF  $s = \varepsilon$  THEN {                /* einfacher Methodenaufruf */
19                      $E := E \cup (\langle k::m \rangle, \langle k_1::i_1 \rangle, \text{'message'})$ 
20                 ELSE                                /* 'super', 'this' oder 'new' */
21                      $E := E \cup (\langle k::m \rangle, \langle k_1::i_1 \rangle, s)$ 
22                 } /* IF */
23             FORALL ( $a \in k::m.\text{Assignments}$ ) DO { /* def/uses-Kanten erzeugen */
24                  $i := a.\text{LeftHandSide}$ ;  $e := a.\text{RightHandSide}$ ;
25                 IF  $i \in k.\text{InstanceVariables}$  THEN { /* def */
26                      $k_1 := \text{DEFCLASS}(i)$ ;
27                      $E := E \cup (\langle k_1::i \rangle, \langle k::m \rangle, \text{'def'})$ 
28                 } /* IF */;
29                 FORALL  $i \in e.\text{VARIABLES}$  DO
30                     IF  $i \in k.\text{InstanceVariables}$  THEN { /* uses */
31                          $k_1 := \text{DEFCLASS}(i)$ ;
32                          $E := E \cup (\langle k::m \rangle, \langle k_1::i \rangle, \text{'uses'})$ 
33                     } /* IF */;
34             } /* FORALL */
35         } /* FORALL */
36     CALL PolyKBD-I; /* Vererbung und Interface-Implementation berücksichtigen, Abb. 22.4 */
37     RETURN ( $V, E, P$ )
38 END Generiere KBDI.

```

Abb. 22.3 Algorithmus Generiere KBD<sub>I</sub>

und im Falle von  $\text{PREFIX}(ml) = \text{'super'}$  um einen statisch gebundenen selbstrekursiven Methodenaufruf.

- Die Funktion  $\text{TYP}(ml)$  ermittelt nach der Sprachdefinition die Wurzel der Vererbungs- bzw. Interface-Hierarchie mit den als Ziel des Aufrufs erlaubten (polymorph) substituierbaren Klassen bzw. Interface-Implementationen. In anderen Worten: Bei jeder Ausführung eines einfachen (also mit **this** oder gar nicht gekennzeichneten) Methodenaufrufs  $ml$  wird dynamisch — also zur Ausführungszeit gebunden — eine in der Klasse  $\text{TYP}(ml)$  oder einer ihrer Unterklassen (s. Funktion  $\text{PolyKBDI}$  in Abb. 22.4) definierte Methode ausgeführt. Im Gegensatz dazu sind die mit **super** bzw. **new** gekennzeichneten Methodenaufrufe statisch an eine bestimmte, syntaktisch ermittelbare Methodendefinition gebunden (vgl. [GJS96], s.a. [MülPoe98]).
- Für eine Klasse  $k$  ergibt der Ausdruck  $k.\text{InstanceVariables}$  die Menge aller innerhalb der Methoden von  $k$  sichtbaren Instanzvariablen.
- Für eine Instanzvariable  $i$  ergibt die Funktion  $\text{DEFCLASS}(i)$  die  $i$  definierende Klasse.
- Für einen Ausdruck  $e$  ergibt die Funktion  $\text{VARIABLES}(e)$  die Menge der in  $e$  vorkommenden Variablen.

Fassen wir zusammen: Das Klassen-Botschaftsdiagramm stellt mit jedem Knoten eine Methode dar. Mit den Kanten werden entweder direkte Aufruf- bzw. Dateninteraktionen oder aber Vererbungsbeziehungen dargestellt.

## Weitere Anwendungen des $\text{KBD}_I$

Die Ermittlung bzw. Verfeinerung von Integrations-Strategien auf der Basis des  $\text{KBD}_I$  ist in [Winter98] beschrieben. Darüber hinaus ist es möglich, mit dem  $\text{KBD}_I$  die Menge der für einen Regressionstest<sup>1</sup> erneut auszuführenden Testfälle einzugrenzen. Von einem exemplarisch für Anwendungen in der Programmiersprache Smalltalk-80 ([GolRob89]) implementierten Testwerkzeug wird das  $\text{KBD}_I$  generiert. Mit Hilfe des  $\text{KBD}_I$  ermittelt das Werkzeug dann alle möglicherweise von den Änderungen am Quellcode der Anwendung betroffenen, erneut zu testenden Methoden (s. [Mohl97]).

## Andere Modelle für objektorientierte Programme

**Palsberg und Schwarzbach** verwenden zur Typ-Inferenz und zur Ermittlung nicht ausführbarer Teile objektbasierter Anwendungen sogenannte *Trace-Graphen* ([PalSch94]).

---

<sup>1</sup> Als Regressionstest wird die erneute Ausführung und Auswertung von Tests nach Änderungen der Implementation bezeichnet (vgl. [Riedemann97]).

Das  $KBD_I$  stellt eine die Vererbung berücksichtigende, auf den Test zugeschnittene Erweiterung des Trace-Graphen dar.

**McGregor et al.** stellen mit dem *Object-Oriented Program Dependency Graph* (OPDG) ein feingranulares, heterogen zusammengesetztes Modell vor ([MGMS96]). Der OPDG besteht aus den vier Teilgraphen

- ❑ Class Hierarchy Subgraph (CHS, Klassendiagramm),
- ❑ Control Dependency Subgraph (CDS, Kontrollflussgraphen der Methoden),
- ❑ Data Dependency Subgraph (DDS, Datenflussgraphen der Methoden) und
- ❑ Object Dependency Subgraph (ODS, Objektkonstellation als Ergebnis einer abstrakten Interpretation des CHS, CDS und des DDS).

Der OPDG ist isomorph zum Quellcode der Anwendung und mit Hinblick auf interprozedurale Kontroll- und Datenflussanalysen sowie Alias-Untersuchungen entworfen worden. Das  $KBD_I$  bildet sozusagen einen für unsere Zwecke passenden abstrahierenden und unifizierenden Schnitt durch die drei statischen Teilgraphen CHS, CDS und DDS des OPDG.

## A.4 $KBD_I$ : Ausdrucksstärke

Das  $KBD_I$  ist rein syntaktisch, also auf Klassenebene definiert. Es abstrahiert von Aspekten der Objektebene wie z.B. Aliases und aktuellen Belegungen der Parameter eines Methodenaufrufs. Auch der Kontrollfluss bzw. die tatsächliche Reihenfolge der gesendeten Botschaften bei der Ausführung einer Methode wird vernachlässigt. Dem  $KBD_I$  liegt somit ein asynchrones, „broadcast“-basiertes Ausführungsmodell zugrunde (vgl. [CooDan94]):

- ❑ Die während der Ausführung einer Methode gesendeten Botschaften bzw. bewirkten Zustandsmodifikationen werden quasi als simultan angenommen;
- ❑ Jede Botschaft wird (gleichzeitig) an alle Instanzen der Zielklasse(n) gesendet.

Effektiv erfasst das  $KBD_I$  so eine Obermenge der bei der Ausführung einer Anwendung möglichen Interaktionen zwischen Objekten, und wir erhalten den

**Satz 22.3** Das  $KBD_I$  repräsentiert eine obere Grenze für die zwischen Objekten einer Anwendung („zur Laufzeit“) möglichen Interaktionen.

**Beweis** (Skizze) Der Beweis erfolgt zunächst durch Enumeration aller möglichen direkten Aufruf- und Daten-Interaktionen und ihre konstruktive Berücksichtigung im Algorithmus Generiere  $KBD_I$  (Abb. 22.3) bzw. der Funktion Poly $KBD_I$  (Abb. 22.4). Die Überdeckung der indirekten (Zustands-) Interaktionen sowie durch Aliasing verursachten „Nebenwirkungen“ wird mit der asynchronen, Broadcast-basierten Betrachtungsweise ([CooDan94]) des  $KBD_I$  gezeigt.

Wir betrachten hierzu die in Abschnitt A.2 aufgezählten Möglichkeiten der Interaktion nacheinander und unterscheiden jeweils im Kontext der Klasse des aufrufenden bzw. aufgerufenen Objekts

- ❑ neu definierte (N),
- ❑ redefinierte (R) sowie
- ❑ unverändert geerbte (I) Methoden (vgl. [PerKai90][HMGF92][HKR+97]) und zusätzlich auch noch
- ❑ überschriebene Methoden (overridden/hidden, O) (vgl. [BlaFrö98]).

Daneben betrachten wir den Fall des Aufrufs einer identischen Methode jeweils gesondert (Rekursion, s. auch [BlaFrö98]). Zur Illustration zeigen wir dieses Vorgehen für direkte Intra-Klassen Interaktionen. Zum Schluss skizzieren wir kurz den Fall indirekter Interaktionen.

## Direkte Intra-Klassen Interaktionen in Java

Für die Intra-Klassen-Interaktionen ergeben sich die in Tabelle 22.1 gezeigten Kombinationen von aufrufender Methode m1 in Objekt o1 („Sender“) und aufgerufener Methode m2 in Objekt o2 („Empfänger“). Unterschiede bei Klassen- und Instanz-Methoden sind durch vorangestelltes I bzw. C kenntlich gemacht. Betrachten wir zunächst die direkten Intra-Klassen/ Intra-Objekt Interaktionen in Tabelle 22.1, Spalte A, für die Abb. A.3 jeweils Beispiele zeigt. Insbesondere die Interaktionsarten 10.A und 14.A spiegeln die Kernidee der Erweiterung objektorientierter Klassen durch das Überschreiben sogenannter „Template Methoden“ (s. [GHJ+94]) wieder. Gosling, Joy und Steele schreiben hierzu in der Java Sprachspezifikation:

Overriding is sometimes called "late-bound self-reference" [... a] reference [...] invokes a method chosen "late" (at run time, based on the run-time class of the object referenced by this) rather than a method chosen "early" (at compile time, based only on the type of this). This provides the Java programmer a powerful way of extending abstractions and is a key idea in object-oriented programming. [GJS96]

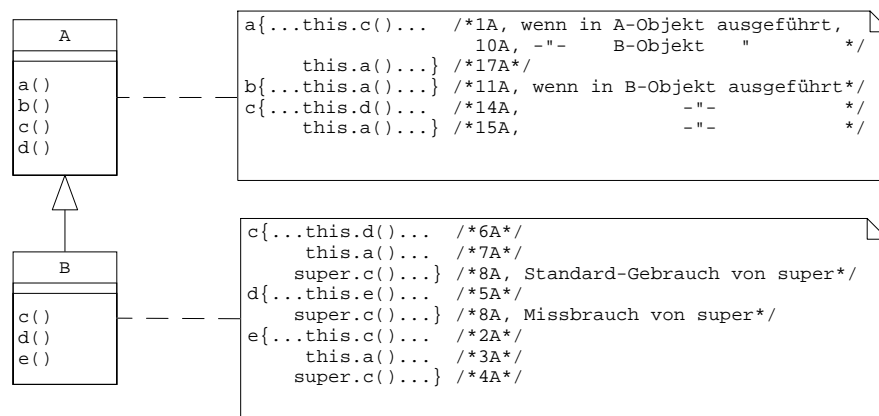


Abb. A.3 Direkte Intra-Klassen, Intra-Objekt (Aufruf-) Interaktionen

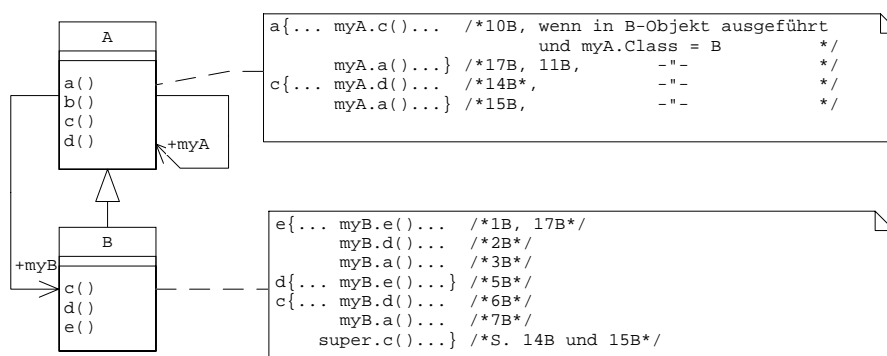


Abb. A.4 Direkte Intra-Klassen, Inter-Objekt (Aufruf-) Interaktionen

Bezüglich der direkten Intra-Klassen/Inter-Objekt Interaktionen betrachten wir Tabelle 22.1, Spalte B sowie Abb. A.4. Da überschriebene Methoden<sup>1</sup> außerhalb eines Objekts in Java (im Gegensatz zu c++, s. auch [BlaFrö98]) nicht durch eine explizite Typangabe vor dem Aufruf zugreifbar sind (*primary* bzw. *cast*, [GJS96]), sind ihre Aufrufe bei Inter-Objekt Interaktionen in Java nicht möglich, wohl aber die Tatsache, dass überschriebene Methoden (durch vorheriges **super**) selbst ausgeführt werden und somit auch Aufrufer sein können. Im Gegensatz zu überschriebenen Methoden sind überschriebene Variablen durch entsprechende Typ-Konvertierungen (*cast*) von aussen zugreifbar. Gosling et al. schreiben hierzu:

Indeed, there is no way to invoke the [overridden] `getX` method of class `Point` for an instance of class `RealPoint` from outside the body of `RealPoint`, no matter what the type of the variable we may use to hold the reference to the object. Thus, we see that fields and methods behave differently: hiding is different from overriding. [GJS96]

Wir fassen unsere bisherigen Überlegungen zusammen: Als direkte Aufrufinteraktionen haben wir die durch die Schlüsselworte **this** bzw. **super** erkenntlichen dynamisch bzw. statisch gebundenen selbstrekursiven (vgl. [BlaFrö98]) Aufrufe sozusagen als Möglichkeiten der vertikalen Wiederverwendung oder Kopplung (innerhalb einer Vererbungshierarchie) dargestellt. Alle anderen Aufrufe bedeuten eine horizontale Wiederverwendung oder Kopplung im Sinne der *Delegation* (vgl. [GHJ+94]), also dynamisch gebundene einfache Aufrufe. Überschriebene (*overridden*) Instanz-Methoden sind „von aussen“ nicht sichtbar; überschriebene (*hidden*) Klassen-Methoden sind nach einem Cast „von aussen“ sichtbar.

<sup>1</sup> Überschriebene Methoden sind in [PerKai90][HMGF92] und [HKR+97] nicht betrachtet.

Nr.	Typ m1	Typ m2	Intra-Objekt o1=o2, A	Inter-Objekt o1<>o2, B
1.	N	N	Standard-Aufruf (auch <b>this</b> )	Standard-Aufruf
2.	N	R	Wie 1.	Wie 1.
3.	N	I	Wie 1. (redundantes <b>super</b> )	Standard, „Vertikale Kopplung“
4.	N	O	Fraglicher gebrauch von <b>super</b>	I: In Java nicht möglich, C: nach Cast
5.	R	N	Wie 1.	Wie 1.
6.	R	R	Wie 1.	Wie 1.
7.	R	I	Wie 3.	Wie 3.
8.	R	O	Standardgebrauch von <b>super</b>	I: In Java nicht möglich, C: nach Cast
9.	I	N	In Java nicht möglich	I: In Java nicht möglich, C: nach Cast
10.	I	R	Polymorphismus (auch <b>this</b> )	Polymorphismus
11.	I	I	Wie 1.	Wie 1.
12.	I	O	In Java nicht möglich	I: In Java nicht möglich, C: nach Cast
13.	O	N	In Java nicht möglich	I: In Java nicht möglich, C: nach Cast
14.	O	R	Polymorphismus (nur nach <b>super</b> )	Nur nach <b>super</b> (Template/Strategy)
15.	O	I	Nur nach <b>super</b>	Nur nach <b>super</b>
16.	O	O	In Java nicht möglich	I: In Java nicht möglich, C: nach Cast
17.	x	x	m1=m2, „echte“ Rekursion	m1=m2, strukturelle Rekursion (Iterator)

Tab. 22.1 Direkte Intra-Klassen (Aufruf-) Interaktionen

## Indirekte Interaktionen in Java

Haworth et al. bezeichnen sowohl neu- als auch redefinierte Methoden bzw. Variablen als „lokal“ und unverändert geerbte sowie überschriebene Methoden bzw. Variablen als „geerbt“ ([HKR+97]). Sie notieren indirekte Intra-Klassen/IntraObjekt Interaktionen in der Form  $\langle m1, v, m2 \rangle$ , wobei m1 den Typ der schreibenden Methode, v die Art der betroffenen Variablen und m2 die Art der lesenden, also beeinflussten Methode bedeuten. Hierbei können solche zwischen je einem modifizierenden Zugriff und lesenden Zugriffen liegenden Interaktionen vernachlässigt werden, welche die modifizierte Variable nicht erneut modifizieren — man betrachtet also im Prinzip analog zum datenflussbasierten Test sogenannte „def-uses Paare“ (vgl. [Beizer90][Beizer95][Myers79][PagSix94][Riedemann97]). So bedeutet z.B. das Tripel  $\langle N, I, R \rangle$ , dass nach der Modifikation einer Variablen durch eine neu definierte Methode eine geerbte Methode die modifizierte (geerbte) Variable liest.



## Direkte Interaktionen im KBD<sub>I</sub>

Wir betrachten zunächst, wie das KBD<sub>I</sub> direkte Interaktionen, also „Methodenaufrufe“ reflektiert. Im Gegensatz zu den Modellen in [HMGF92] und [HKR+97] beschränkt sich das KBD<sub>I</sub> nicht auf Aufrufe innerhalb einer Klasse bzw. eines Objekts (Intra-Objekt/Intra-Klasse), sondern erfasst gleichzeitig auch Aufrufe der Art Inter-Objekt/Intra-Klasse und Inter-Objekt/Inter-Klasse. Die direkten (Aufruf-) Interaktionen (Tabelle 22.1, Nr. 1. bis 6. sowie Nr. 9.) werden durch die Konstruktion des KBD<sub>I</sub> berücksichtigt (Zeilen 7 bis 14). Die polymorphe direkte (Aufruf-) Interaktion der Art (I, R) (Tabelle 22.1, Nr. 8.) wird durch die Funktion PolyKBD-I berücksichtigt. Aufrufe der Art (I, N) (Tabelle 22.1, Nr. 7.) sind (in Java) nicht möglich, da aufgrund der statischen Typisierung Aufrufe „undefinierter“ bzw. noch nicht definierter Methoden zur Kompilierungszeit zurückgewiesen werden. Die direkten (Daten-) Interaktionen 1. bis 6. sowie 9. werden durch die Konstruktion des KBD<sub>I</sub> berücksichtigt (Zeilen 15 bis 24). Aus der Konstruktion des KBD<sub>I</sub> folgt so zunächst die Überdeckung der direkten Interaktionen.

## Indirekte Interaktionen im KBD<sub>I</sub>

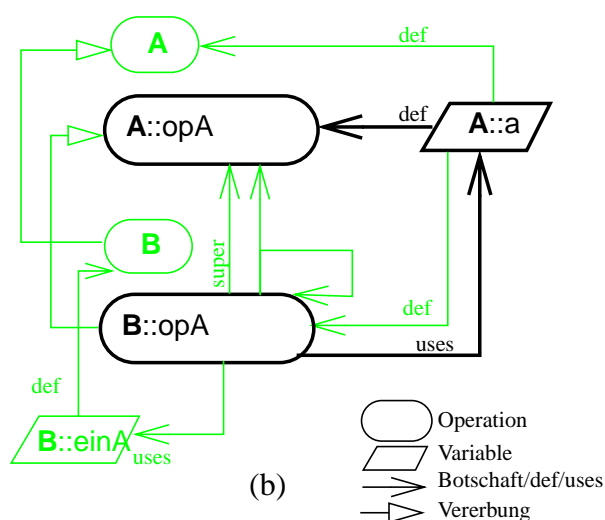
Abschließend beleuchten wir, wie sich indirekte Interaktionen (Zustandsinteraktionen), d.h. aufeinanderfolgende modifizierende und lesende Zugriffe auf Variablen, im KBDI wiederfinden. Im Prinzip sind dafür Pfade im KBDI zu verfolgen, die mit def bzw. mit uses markierte Kanten zu bzw. von verschiedenen Operationen beinhalten. Da def- bzw. uses- Kanten immer vom Variablen- zum Methodenknoten bzw. umgekehrt gerichtet sind, enthalten die indirekten Interaktionen entsprechenden Pfade dabei immer gleichviele, jeweils aufeinanderfolgende def- und uses- Kanten. Solche indirekten Interaktion spiegeln sich im KBD<sub>I</sub> durch eine Struktur wie die in Abb. A.5 fett hervorgehobene wider.

```

class A {
    public integer a;
    public void opA {
        a = 1;
    }
}
class B extends A {
    public A einA;
    public void opA {
        super.opA;
        a = a + 1;
        einA.opA
    }
}

```

(a)



(b)

Abb. A.5 Java Code und KBD<sub>I</sub> zu einer <N, I, R>-Zustands-Interaktion

## Anhang B

### Syntax der Test-Skriptsprache

Dieser Anhang beschreibt die Syntax der SCORES-Skriptsprache für Testfälle. Die Testfälle werden vom Test-Treiber gelesen und mit aktuellen Parametern (Testdaten) für eine zu testende Klasse (CUT, Class Under Test) ausgeführt. Testfälle für Klassen-, Cluster- und Systemtests werden rekursiv in Testsuiten organisiert. Zur Erzeugung von Parameterobjekten können Testfälle in Methodenaufrufen und zur Initialisierung der CUT (textuelle Substitution) in einem Testfall rekursiv eingesetzt werden.

Terminale Symbole sind **fett** gedruckt, nichtterminale Symbole in Standard. Die verwendeten Metazeichen haben folgende Bedeutung, wobei A und B beliebige syntaktische Einheiten seien:

- $A^0$     Kein oder ein A ,
- $A^*$     keines, eins oder mehrere A,  $X ::= A^*$  entspricht  $X ::= XA \mid \epsilon$ ,
- $\{AB\}$    Fasst A und B zu einer syntaktischen Einheit zusammen,
- $AB\dots$    Ein oder mehrere, durch B getrennte A's, z.B. A, ABA oder ABABA .  
Die Produktion  $X ::= AB\dots$  ist äquivalent zu  $X ::= A\{BA\}^*$  und zu  $X ::= A \mid XBA$ .

#### Produktionsregeln

```

ClassTestCase ::= TESTCASE TCId TestCase
TestCase ::= TCHdr FormParams0 LokalDekls0 Precond0 TCBdy
TCHdr ::= ROOTCLASS ClassId | USECASE UCId
FormParams ::= ( Params )
Params ::= { OList : ClassId } ; ...
OList ::= ObjectId , ...
LokalDekls ::= LOCAL Params
Precond ::= PRECONDITION ObjectQuery , ...
TCBdy ::= BEGIN Constructor0 TestMessage* END
Constructor ::= CREATE Message Outcome0 ;
TestMessage ::= Executable Outcome0 ;

```

Executable ::= MessageExp | TestCaseSubst  
 MessageExp ::= {ObjectId = }<sup>0</sup> {ObjectId . }<sup>0</sup> Message  
 Message ::= Messageld ActualParams<sup>0</sup>  
 ActualParams ::= ( ActualParam , ... )  
 ActualParam ::= BasicExpr | ObjectId | TestCaseInst  
 TestCaseSubst ::= **SUBST** TCId ActualParams<sup>0</sup>  
 TestCaseInst ::= **INST** TCId ActualParams<sup>0</sup>  
 BasicExpr ::= IntegerExp | RealExp | BoolExp | Char | String | **null**  
 Outcome ::= Exception | Oracle  
 Exception ::= **EXCEPTION** {ObjectId :}<sup>0</sup> ClassId String<sup>0</sup> ObjectQuery\*  
 ObjectQuery ::= Executable RelOp { Executable | BasicExpr }  
 Oracle ::= **ORACLE** OrPart , ...  
 OrPart ::= @pre<sup>0</sup> ObjectQuery | MessageTrace  
 RelOp ::= == | **isEqual** | != | **isEqual**  
 ObjectId ::= Identifier  
 ClassId ::= Identifier  
 TCId ::= Identifier  
 UCId ::= Identifier  
 MessageTrace ::= ( MessageTrace - ... ) | Message  
 Message ::= ClassId :: MessageId  
 MessageId ::= Identifier  
 IntegerExp ::= integer  
 String ::= string  
 Char ::= character

## Kommentare

1. Lokale Objekte dienen zur Aufnahme von Rückgabeobjekten der Botschaften und können im Orakel der Botschaft sowie im Verlauf des Testfalls (z.B. als Botschaftsparameter) verwendet werden. Botschaften an lokale Objekte dürfen nur im Orakel gesendet werden.

2. `TestCaseSubst` ist semantisch eine textuelle Einsetzung des betreffenden Testskripts mit den substituierten formalen Parametern, aber ohne den Konstruktor. Der aufgerufene Testfall muss dabei für eine Oberklasse oder für dieselbe Wurzelklasse wie der aufrufende Testfall definiert sein. Es wird kein neues Objekt erzeugt.
3. `TestCaseInst` beinhaltet dagegen die Objekterzeugung (und Initialisierung).
4. Identifizierer (`<name>`) wie `MessageId`, `ClassId` sowie Ausdrücke wie `IntegerExp`, `RealExp`, `BoolExp`, `String` und `Char` etc. müssen der Syntax der Implementationssprache entsprechen bzw. auf diese abbildbar sein.
5. Im Orakel bedeuten:
  - **@pre** der Attributwert vor Ausführung, ähnlich wie in OCL,
  - **==** die Prüfung auf Objekt-Identität (Referenz-Gleichheit),
  - **isEqual** die Prüfung auf Objekt-Gleichheit (Werte-Gleichheit, nicht rekursiv auf Instanzvariablen).

# Index

## Symbole

$\nabla$ .....	87
$*$ .....	120
$+$ .....	120
$,$ .....	120
$::$ .....	86
$<$ .....	20, 83, 87
$\leq$ .....	87
$\equiv$ .....	92
$\ll$ .....	87
$\text{IN}$ .....	66
$\text{IR}$ .....	66
$\mathbf{K}$ .....	86
$\subseteq$ .....	120
$\subseteq \text{I}$ .....	87
$\mathbb{Z}$ .....	66
$\perp$ .....	120
$\{(\dots)\}$ .....	86

## A

Abbruchkriterium .....	6
Abflachen .....	133
Abnahmetest .....	7, 161
Adäquatheitskriterium .....	6
Aktivitätsdiagramm .....	96
Aktor .....	20
Generalisierung .....	20
Alias .....	242, 247
Anforderung .....	16
extern beobachtbar .....	16
funktionale .....	16
nicht-funktionale .....	17
Anforderungsermittlung .....	16
Anforderungsspezifikation .....	17
Konsistenz .....	105
Korrektheit	
fachliche .....	105

formale .....	105
Vollständigkeit	
fachliche .....	105
formale .....	105
Aspekt	
ablauf .....	15
funktionaler .....	14
informationsbezogener .....	15
operationaler .....	14
organisatorischer .....	14
Ausfall .....	4

## B

Bedingungsüberdeckung	
einfache .....	138
Mehrfach- .....	138
Minimale-Mehrfach- .....	138
Benutzbarkeitstest .....	7
Botschaft .....	137
Botschaftskante .....	172
Business scenario .....	111

## C

CAST .....	206
Cluster .....	30, 32

## D

Defekt .....	4
Delegation .....	249
Domänenobjekt	
anwendungsinterne Darstellung .....	58, 88
reales .....	88
Domänen-Operation .....	129

<b>E</b>	Interaktionsdiagramm . . . . . 23
Entscheidungstabelle . . . . . 142	<b>K</b>
Episode . . . . . 129, 130, 169	KBD
Episodenschritt . . . . . 130	Aufruf-Interaktionskanten . . . . . 243
Evaluation . . . . . 7	Botschaftskante . . . . . 238
extends . . . . . 21, 82	Daten-Interaktionskanten . . . . . 243
<b>F</b>	Vererbungskante . . . . . 238
Fehler . . . . . 4	Klassen-Botschaftsdiagramm
Fehler-	der Anforderungsspezifikation 169, 236
korrektur . . . . . 5	der Implementation . . . . . 170, 242
ursache . . . . . 4	Klassen-Botschaftsdiagramm (KBD)
Fehlerrückmeldung . . . . . 166	für die Anforderungsspezifikation . 237
Fehlerhypothese . . . . . 5	für die Implementation . . . . . 242
Fehlerrate . . . . . 190	Klassendiagramm . . . . . 131
flattening → Abflachen	Klassenmodell
<b>G</b>	Elemente . . . . . 86
Generalisierung . . . . . 20, 21	Klassentest . . . . . 32
Geschäftsprozess . . . . . 14	Knoten
GUI-Test . . . . . 7	Methoden- . . . . . 172
<b>I</b>	Variablen- . . . . . 173
Information	Komponententest . . . . . 7
Interaktions- . . . . . 46, 58, 86	<b>L</b>
Kontextuelle . . . . . 46, 58, 63, 86	Lasttest . . . . . 7
Systeminterne 47, 58, 86, 129, 133, 169	<b>N</b>
Inspektion . . . . . 108	Nebenläufigkeitstest . . . . . 7
Schrittweise . . . . . 108	<b>O</b>
Instrumentierung . . . . . 189	OCL . . . . . 61
Integrationstest . . . . . 7	<b>P</b>
Interaktion . . . . . 46	Prototyping
direkte . . . . . 241	evolutionäres . . . . . 106
externe . . . . . 46	experimentelles . . . . . 106
indirekte . . . . . 241	exploratives . . . . . 106
interne . . . . . 28, 29, 47, 241	horizontales . . . . . 106
Aufruf- . . . . . 174	vertikales . . . . . 106
direkte Aufruf- . . . . . 172	Prüfgegenstand . . . . . 4
direkte Daten- . . . . . 173, 174	
indirekte . . . . . 174	
Zustands- . . . . . 241	

<b>Q</b>		<b>Test</b>	
Qualitätssicherung .....	5	Abnahme- .....	35
objektorientierter Anwendungen...	26	black-box .....	6
<b>R</b>		C2- → Bedingungsüberdeckung	
		einfache	
		C3- → Bedingungsüberdeckung	
		Mehrfach-	
Realzeittest .....	7	gegen die Anforderungsspezifikation .	
Referenzfunktion		33, 36, 48, 159, 161, 169	
Intra-Use-Case .....	66	Laufzeit- .....	34
Makroschritt .....	63	Massen- .....	35
Regressionstest .....	7	SCORES grey-box	11, 160, 169, 212, 213
Review .....	107	Stress- .....	34
-techniken .....	107	System- .....	34–35
Robustheit .....	34	white-box .....	6
<b>S</b>		<b>Test-</b>	
Schritt		ausführung .....	189
Test-Szenario- .....	112	auswertung .....	190
Use Case- .....	62, 112	datum .....	6
SCORES		endekriterium .....	6
Metamodell .....	113	fall .....	6
vollständiges .....	135	kriterium .....	6
Metamodell .....	68, 90	management .....	7
Methodik .....	98	orakel .....	7, 160, 163, 176, 191
Sequenzdiagramm .....	23	prozess .....	7
Sichtbarkeitsbereich .....	131	vorfallsbericht .....	190
initialer .....	131	Testmodell .....	5
von Episodenschritten .....	131	Testtheorie .....	5
Stakeholder .....	18, 47	Testverfahren .....	7
Standard-Operation .....	129	Testwerkzeug .....	7, 206
Stresstest .....	7	<b>U</b>	
Systemtest .....	7	<b>Überdeckung</b>	
anforderungsbasiert .....	161	.....	93
Szenario .....	20, 47, 65, 118	Bedingungs-	
Test- .....	112, 129, 161	minimale Mehrfach- .....	138
Szene .....	99	einfache Klassen- .....	92
<b>T</b>		vollständige Klassen- .....	93
		vollständige Operations- .....	140
Template Methode .....	248	Übergangsbedingung .....	65

Use Case .....	20, 61
Diagramm .....	21
Erweiterungspunkte .....	82
Generalisierung .....	83
Use Case Diagramm .....	42
Use Case Schritt .....	62
Blatt .....	66, 81
End- .....	82
Endschritte .....	65, 70, 80
innerer .....	66, 80
Interaktions- .....	63, 163
Kontext- .....	63, 163
Makro .....	63, 163
Startschritt .....	65, 70, 80, 82
Wurzelschritt .....	70
Use Case Schrittgraph .....	65
uses .....	21, 82

## V

Validierung .....	5, 104
Metrik .....	115
Validierungseinheit .....	200
Vererbungskante .....	172
Verfolgbarkeit .....	150
horizontale .....	148
vertikale .....	148
Verifikation .....	5, 104, 125
Metrik .....	137
Vollständigkeit	
fachliche .....	105

## W

Workflow .....	14
Wurzelklasse .....	90
Wurzeloperation .....	90

## Z

Zerlegung	
Funktionale .....	109
funktionale .....	64
Zustandsinteraktion .....	241



# Kurzbiographie des Autors

**Dipl.-Inform. Dipl.-Ing. Mario Winter**, Fachbereich Informatik, FernUniversität - Gesamthochschule in Hagen. Wissenschaftlicher Mitarbeiter am Lehrgebiet Praktische Informatik III von Prof. Dr. Hans-Werner Six. Mitglied der GI-Fachgruppe 2.1.7 „Testen, Analysieren und Verifizieren von Software“ (TAV), dort Sprecher des Arbeitskreises „Testen objektorientierter Programme“ (TOOP).

Geboren am 22. März 1959 in Berlin als zweites Kind des Dipl.-Kaufmannes Manfred R. Winter und der Helga I. Winter, geb. Macke. Seit dem 27. Mai 1987 verheiratet mit der Dipl.-Sozialarbeiterin Petra Winter, geb. Glebe. Zwei Kinder.

Abitur 1978. Studium der Elektrotechnik, Fachrichtung Informationsverarbeitung an der Universität - Gesamthochschule Siegen von Oktober 1979 bis März 1983, Abschluss mit dem Grad Diplom-Ingenieur. Studium der Informatik mit Nebenfach Elektrotechnik an der FernUniversität - Gesamthochschule in Hagen von Oktober 1986 bis August 1994. Abschluss mit dem Grad Diplom-Informatiker.

Von Juli 1978 bis September 1979 Grundwehrdienst. Von Juni 1983 bis November 1985 angestellt in einem Softwarehaus, Arbeitsgebiet Entwicklung von CAD-Systemsoftware. Von Dezember 1985 bis April 1986 Zivildienst. Danach bis Juni 1987 weiter angestellt in obigem Softwarehaus. Von Oktober 1987 bis August 1994 Laboringenieur an der Universität - Gesamthochschule Wuppertal, Arbeitsgebiet Entwicklung von KI-gestützter Software für die Regelungstechnik. Ab September 1991 dort Leiter der entsprechenden Arbeitsgruppe. Seit September 1994 wissenschaftlicher Mitarbeiter am Lehrgebiet Praktische Informatik III, Fachbereich Informatik der FernUniversität - Gesamthochschule in Hagen. Arbeitsgebiete: Software-Engineering, Software-Qualitätssicherung, räumliche Datenstrukturen.

