UML-basierter Klassen- und Integrationstest objektorientierter Programme

vorgelegt von
Diplom-Informatikerin Dehla Sokenou

von der
Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktorin der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuß:
Vorsitzender: Prof. Dr. Hans-Ulrich Heiß
Berichter: Prof. Dr. Stefan Jähnichen
Berichterin: Prof. Dr. Ina Schieferdecker

Tag der wissenschaftlichen Aussprache: 6. September 2005

Berlin 2006
D 83

Vorwort

Ein Mathematiker, ein Physiker und ein Ingenieur werden mit einer Aussage konfrontiert: "Alle ungeraden Zahlen sind Primzahlen."
Der Mathematiker denkt kurz nach und erklärt dann: "Das kann nicht stimmen, 9 ist eine ungerade Zahl, jedoch keine Primzahl."
Der Physiker meint: "Nun, mal sehen ... 1 ist eine Primzahl. 3 ist eine Primzahl. 5 ist eine Primzahl. 7 ist eine Primzahl. Es scheint zu stimmen."
Der Ingenieur antwortet: "1 ist eine Primzahl. 3 ist eine Primzahl. 5 ist eine Primzahl. 7 ist eine Primzahl. 9 ist eine Primzahl. 11 ist eine Primzahl. 13 ist eine Primzahl ..."

Als einziger hat der Mathematiker den Fehler erkannt und ein Gegenbeispiel geliefert. Der Physiker hat durch Ausprobieren die Aussage überprüft, jedoch nicht den richtigen Wert gefunden, der die Aussage widerlegt hätte. Der Ingenieur hat zwar den richtigen Wert ausprobiert, jedoch nicht erkannt, daß dieser Wert die Aussage widerlegt.

Dieses Beispiel zeigt zwei verschiedene Probleme beim Testen, einem Verfahren, das mit Stichproben arbeitet.

Das erste Problem betrifft die Auswahl der Stichprobe. Der Physiker hätte den Fehler vielleicht entdeckt, wenn er nur genug Werte ausprobiert hätte. Ein vollständiges Ausprobieren ist in der Regel nicht möglich, wie bereits dieses kleine Beispiel zeigt, da die Menge der potentiellen Werte, hier die Menge der ungeraden natürlichen Zahlen, meist unendlich ist. Die Menge muß also auf eine Stichprobe beschränkt werden. Die Beschränkung muß dabei so gewählt werden, daß der Fehler erkannt wird.

Das zweite Problem betrifft die Überprüfung des Ergebnisses mit den erwarteten Werten. Der Ingenieur probiert zwar die richtigen Werte aus, übersieht jedoch den Fehler. Nicht immer ist diese Überprüfung so einfach wie im Beispiel, so daß auch der Vergleich zwischen den erwarteten und den tatsächlichen Werten ein Problem darstellt.

Am besten wäre es natürlich, den Mathematiker nachzuahmen, es ist jedoch aufgrund der Komplexität heutiger Software praktisch unmöglich, die Korrektheit zu beweisen oder durch ein Gegenbeispiel zu widerlegen. Die beste Näherung ist die geeignete Auswahl der Stichprobe basierend auf einer durch Heuristiken gestützten Testhypothese und die möglichst korrekte und genaue Auswertung der Testergebnisse.

Frei erzählt nach [77].

Danksagung

Ich danke meinen Gutachtern Prof. Stefan Jähnichen und Prof. Ina Schieferdecker für ihre Bereitschaft, diese Arbeit zu betreuen, und für wertvolle Hinweise.

Ein besonderer Dank gilt dem Menschen, der mich zum Testen gebracht hat, Peter Rüppel, und den Kollegen, die mir den Mut gegeben haben, diese Arbeit zu vollenden, besonders Steffen Helke, der mir auch mit kompetentem fachlichem Rat zur Seite stand, Gabi Ambach und Margot Bittner. Stephan Herrmann danke ich für seine Anregungen zu aspektorientierten Programmiertechniken. Steffen Helke und Mirijam Joka Sokenou gilt mein besonderer Dank für das Korrekturlesen in letzter Minute.

Den Mitgliedern des Arbeitskreises UML-basiertes Testen (AKUT), Zhen Ru Dai, Justyna Zander-Nowicka, Dirk Seifert und Mario Friske, danke ich für die Anregungen und konstruktiven Diskussionen, die gerade in der Endphase der Dissertation hilfreich waren. Nicht unerwähnt bleiben sollen auch meine Diplomanden, die mich durch ihre eigene Arbeit unterstützt haben.

Sehr herzlich möchte ich mich bei meinem Mann Johannes von Wrochem bedanken für die große Geduld, wertvolle Hilfe und seinen unerschütterlichen Glauben an die Vollendung dieser Arbeit, sowie bei meiner Familie ebenso wie bei meinen Freunden für ihre Unterstützung während der Promotionszeit.

Inhaltsübersicht

1 Motivation **1**

I Grundlagen des UML-basierten Tests **7**

2 Der Testprozess **9**

3 Testen objektorientierter Software **19**

4 Die Unified Modeling Language **25**

5 Testbarkeit von UML-Modellen **39**

6 Zusammenfassung **51**

II Testerzeugung aus UML-Modellen **55**

7 Das Testsystem UT^3 **57**

8 Basisformalismen **65**

9 Testfälle und Testdaten aus UML-Modellen **77**

10 UML-basierte Testorakel **85**

11 Zusammenfassung **99**

III Testen als Aspekt **101**

12 Aspekte und ObjectTeams **103**

13 Aspektorientierte Testcodeintegration **113**

14 Integration zustandsbasierter Testorakel **117**

15 Integration von Zusicherungen **137**

16 Zusammenfassung **143**

IV Zusammenfassung und Ausblick **145**

17 Fazit **147**

18 Ausblick **151**

Verzeichnisse 153

Literaturverzeichnis **155**

Liste der Definitionen **163**

Abbildungsverzeichnis **165**

Anhang 167

A Struktur des Testsystems UT^3 **169**

B Grammatik zum Einlesen von OCL-Constraints **173**

C Syntax der Algorithmenbeschreibung **177**

Inhaltsverzeichnis

1 Motivation **1**
1.1 Problembereich der Stichprobenauswahl . . . 2
1.2 Problembereich der erwarteten Testergebnisse . . . 3
1.3 Problembereich der Testcodeintegration . . . 4
1.4 Zusammenfassung des Lösungsansatzes . . . 5
1.5 Aufbau der Arbeit . . . 5

I Grundlagen des UML-basierten Tests **7**

2 Der Testprozess **9**
2.1 Fehler und Fehlerursachen . . . 10
2.2 Testbarkeit von Software . . . 10
2.3 Testaktivitäten . . . 11
2.4 Testfallerzeugung und Testdatenermittlung . . . 13
2.5 Testorakel . . . 14

3 Testen objektorientierter Software **19**
3.1 Eigenschaften objektorientierter Systeme . . . 19
3.2 Objektorientierung und Test . . . 21
3.3 Populäre Testansätze für objektorientierte Systeme . . . 22
3.4 Ein einfaches Testmodell für objektorientierte Systeme . . . 23

4 Die Unified Modeling Language **25**
4.1 Statische Diagrammtypen . . . 26
4.1.1 Klassendiagramm . . . 27
4.1.2 Objektdiagramm . . . 27
4.1.3 Komponentendiagramm . . . 27
4.1.4 Kompositionsstrukturdiagramm . . . 28
4.1.5 Deploymentdiagramm . . . 28
4.1.6 Paketdiagramm . . . 28
4.2 Dynamische Diagrammtypen . . . 28
4.2.1 Use-Case-Diagramm . . . 28
4.2.2 Aktivitätsdiagramm . . . 28
4.2.3 Zustandsdiagramm . . . 29
4.2.4 Sequenzdiagramm . . . 31
4.2.5 Kommunikationsdiagramm . . . 34
4.2.6 Timingdiagramm . . . 34
4.2.7 Interaktionsübersichtsdiagramm . . . 34
4.3 Die Object Constraint Language . . . 34
4.3.1 Zustandsinvarianten in OCL . . . 36
4.3.2 Design by Contract in OCL . . . 36

5 Testbarkeit von UML-Modellen 39
5.1 Statische Diagrammtypen . . . 40
5.1.1 Klassendiagramm . . . 40
5.1.2 Objektdiagramm . . . 41
5.1.3 Andere statische Diagrammtypen . . . 42
5.2 Dynamische Diagrammtypen . . . 42
5.2.1 Use-Case-Diagramm . . . 42
5.2.2 Aktivitätsdiagramm . . . 43
5.2.3 Zustandsdiagramm . . . 44
5.2.4 Sequenzdiagramm . . . 45
5.2.5 Kommunikationsdiagramm . . . 46
5.2.6 Timingdiagramm . . . 46
5.2.7 Interaktionsübersichtsdiagramm . . . 46
5.3 Die Object Constraint Language . . . 47
5.4 Kombinierte Ansätze zum UML-basierten Test . . . 47

6 Zusammenfassung 51

II Testerzeugung aus UML-Modellen 55

7 Das Testsystem UT^3 57
7.1 Die Pakete des Testsystems . . . 59
7.2 Fallbeispiel: Banksystem . . . 59
7.2.1 Sequenzen . . . 62
7.2.2 Zustandsautomaten . . . 62
7.2.3 OCL-Constraints . . . 63

8 Basisformalismen 65
8.1 Notationen . . . 65
8.2 Protocol State Machines . . . 66
8.2.1 Formale Beschreibung von Protocol State Machines . . . 67
8.2.2 Ausführungssemantik von Protocol State Machines . . . 69
8.2.3 Grundlegende Algorithmen auf Protocol State Machines . . . 70
8.3 Sequenzdiagramme . . . 73
8.3.1 Formale Beschreibung von Sequenzdiagrammen . . . 73
8.3.2 Grundlegende Algorithmen auf Sequenzdiagrammen . . . 74
8.4 Klassen und Verträge . . . 74
8.4.1 Formale Beschreibung von Klassen . . . 74
8.4.2 Grundlegende Algorithmen auf Klassen . . . 75

9 Testfälle und Testdaten aus UML-Modellen 77
9.1 Testfallgenerierung . . . 77
9.1.1 Reguläre Testfälle . . . 79
9.1.2 Komplementäre Testfälle . . . 82
9.2 Testdatenerzeugung . . . 83

10 UML-basierte Testorakel 85
10.1 Angereicherte Zustandsdiagramme . . . 88
10.2 Erweiterte OCL-Constraints . . . 89
10.3 Vergleich der Techniken . . . 96

11 Zusammenfassung 99

III Testen als Aspekt **101**

12 Aspekte und ObjectTeams **103**
12.1 Aspektorientierte Programmierung 104
12.2 Rollenbasierte Programmierung 106
12.3 Das ObjectTeams-Programmiermodell 107
12.4 ObjectTeams-Schablone 110

13 Aspektorientierte Testcodeintegration **113**

14 Integration zustandsbasierter Testorakel **117**
14.1 Unterstützung von Zustandsdiagrammeigenschaften 117
14.2 Transformation der Zustandsdiagramme 133

15 Integration von Zusicherungen **137**
15.1 Transformation von OCL-Constraints 137
15.2 Generierung der ObjectTeams 139

16 Zusammenfassung **143**

IV Zusammenfassung und Ausblick **145**

17 Fazit **147**

18 Ausblick **151**
18.1 UML-basierte Testverfahren 151
18.2 Aspektorientierte Programmiertechniken 151
18.3 Testsystem UT^3 152

Verzeichnisse **153**

Literaturverzeichnis **155**

Liste der Definitionen **163**

Abbildungsverzeichnis **165**

Anhang **167**

A Struktur des Testsystems UT^3 **169**
A.1 Das Paket `ut.ut_base` 169
A.1.1 Das Paket `ut.ut_base.basics` 169
A.1.2 Das Paket `ut.ut_base.statecharts` 170
A.1.3 Das Paket `ut.ut_base.sequences` 170
A.1.4 Das Paket `ut.ut_base.ocl` 171
A.1.5 Das Paket `ut.ut_base.ot` 171
A.2 Das Paket `ut.ut_tfgen` 171
A.3 Das Paket `ut.ut_ora` 172
A.4 Das Paket `ut.ut_main` 172

B Grammatik zum Einlesen von OCL-Constraints **173**

C Syntax der Algorithmenbeschreibung **177**

Kapitel 1

Motivation

> *Program testing can be used to show the presence of bugs, but never to show their absence!* Edsger W. Dijkstra [24]

Der Softwaretest[1] ist ein in der Praxis weit verbreitetes Verfahren zur Qualitätssicherung von Software. Das Softwareprodukt wird beim Softwaretest mit einer Menge von Eingabedaten stimuliert und die Ausgaben auf ihre Korrektheit im Hinblick auf die Spezifikation überprüft. Da es sich nur um eine Auswahl der Eingabedaten handelt, ist der Softwaretest ein Stichprobenverfahren. Ein erfolgreicher Test läßt nicht auf die Korrektheit des untersuchten Softwareprodukts schließen, sondern nur auf die Korrektheit in Bezug auf die gewählte Stichprobe. Trotz dieser Tatsache ist der Softwaretest ein beliebtes Mittel zum Messen der erreichten Qualität.

Diese Beliebtheit verdankt er der Tatsache, daß es in den meisten Fällen keine Alternative zum Softwaretest gibt. So ist zum Beispiel ein Korrektheitsbeweis von Software aufgrund der Komplexität realer Softwaresysteme nicht möglich.

Er ist das am einfachsten anwendbare Mittel der Messung von Qualität, da die Stimulation des Softwareprodukts mit Eingaben und die Bewertung der berechneten Ausgaben eine Tätigkeit ist, die den Entwicklern der Software vertraut ist. Das Testen verlangt im Prinzip keine zusätzlichen Hilfsmittel, es wird nur die ausführbare Implementierung benötigt[2].

Der Softwaretest ist das einzige Verfahren, das ein Softwareprodukt in seiner realen Umgebung überprüfen kann, da Verfahren wie Korrektheitsbeweis, Simulation oder symbolische Ausführung keine dynamische Ausführung des Softwareprodukts vornehmen.

Der Softwaretest ist folglich im Softwareentwicklungszyklus unerläßlich. Gleichzeitig wirft sein Stichprobencharakter eine Reihe von Problemen auf:

- Wie muß die Stichprobe gewählt werden, um möglichst viele Fehler im Softwareprodukt aufzudecken?
- Läßt sich aufgrund der gewählten Stichprobe auf andere, nicht getestete Eingaben schließen?
- Woraus werden die zu den gewählten Eingaben passenden erwarteten Ausgaben gewonnen?
- Werden auftretende Fehler beim Test erkannt?

[1] Es gibt einige Arbeiten, die zwischen dynamischem und statischem Softwaretest (aus dem Englischen: *dynamic* und *static test*) unterscheiden. In der vorliegenden Arbeit wird als Test der dynamische Softwaretest bezeichnet, der statische Softwaretest wird mit dem Begriff Prüfen gleichgesetzt (vgl. [71]).

[2] Meist werden jedoch Hilfsmittel wie eine Testumgebung verwendet, da ein hoher Automatisierungsgrad den Test effizienter macht.

Diese Fragen sind bis heute Gegenstand der Forschung. Eine Antwort ist in vielen Fällen abhängig vom betrachteten Anwendungsbereich.

Betrachtet man die der vorliegenden Arbeit zugrundeliegenden objektorientierten Systeme, so ist die Frage nach adäquaten Testverfahren ein offenes Problem. Ein kurzer Ausflug in die Historie objektorientierter Systeme verdeutlicht die Problematik.

Die erste objektorientierte Programmiersprache ist Simula aus dem Jahr 1967. Zu einer großen Verbreitung objektorientierter Techniken kam es in den 1980er Jahren, hauptsächlich durch die Programmiersprachen Smalltalk und C++. Objektorientierte Analyse- und Designmethoden sind seit den 1990er Jahren verbreitet. Aus diesen Ansätzen ist auch die heute sehr populäre Modellierungssprache *Unified Modeling Language* (UML) hervorgegangen, die sich inzwischen als Quasi-Standard zur Modellierung objektorientierter Systeme etabliert hat.

Intensive Forschung auf dem Gebiet des Tests objektorientierter Software wurde erst ab Mitte der 1990er Jahre betrieben. Zuvor herrschte die Annahme vor, klassische Ansätze für den Test von imperativer Software (wie sie zum Beispiel in [81, 6, 70] beschrieben sind) ließen sich ohne weiteres auf objektorientierte Software übertragen und objektorientierte Konzepte wie Vererbung und Datenkapselung reduzierten den Testaufwand erheblich. Beide Annahmen erwiesen sich als falsch, da die besonderen Eigenschaften objektorientierter Systeme eine Anwendung klassischer imperativer Verfahren erschweren und z.T. unmöglich machen [96].

Bis heute gibt es nur wenige Testansätze, die speziell auf die Anforderungen an einen Test objektorientierter Systeme angepaßt sind. Es werden oft klassische Testverfahren eingesetzt. Deren Nutzen ist jedoch umstritten, da diese die besonderen Eigenschaften objektorientierter Systeme wie Vererbung oder die Verteilung des Systemzustands auf die Zustände einzelner Objekte nur ungenügend berücksichtigen. Eine Diskussion über die Anwendbarkeit traditioneller Testverfahren auf objektorientierte Systeme wird z.B. in [97] geführt.

Beim Test objektorientierter Systeme stehen Lösungen vor allem für drei Probleme aus, die in dieser Arbeit betrachtet werden:

1. Auf welcher Basis wird die Stichprobe bestimmt?
2. Wie werden die erwarteten Ergebnisse ermittelt?
3. In welcher Weise wird zusätzlicher Testcode in das zu testende System integriert?

Eine Analyse der drei Problembereiche zeigt, daß diese in anderen Anwendungsdomänen weitgehend gelöst sind, für objektorientierte Systeme jedoch viele Fragen offen sind.

1.1 Problembereich der Stichprobenauswahl

Für die Stichprobenauswahl existieren im Bereich imperativer Systeme viele Lösungen, die u.a. aus dem Bereich eingebetteter Systeme stammen.

So werden in [81, 70] diverse Verfahren zur Stichprobenermittlung aus Spezifikationen und Implementierungen für imperative Systeme vorgeschlagen. Die Verfahren der Stichprobenauswahl für imperative Systeme sind zwar in Teilen auf objektorientierte Systeme anwendbar, haben jedoch den Nachteil, daß sie meist schwer automatisierbar sind. Je höher aber der Automatisierungsgrad ist, desto größer ist auch der Zeitgewinn. Die Zeit ist beim Test von essentieller Bedeutung, da der Test als zumeist noch letzte Phase der Softwareentwicklung vor der Auslieferung des Softwareprodukts oft aus Zeitmangel vernachlässigt wird.

Neben dem Zeitgewinn aufgrund eines höheren Automatisierungsgrads wird zudem heute die Integration des Testens in den Softwareentwicklungsprozeß angestrebt. Der Test erfolgt in diesem Fall entwicklungsbegleitend, so daß nicht erst am Ende gravierende Fehler gefunden werden und so die Auslieferung der Software behindert wird. Je früher der Test beginnt, desto geringer sind die Risiken im Softwareentwicklungsprozeß.

Im Bereich der eingebetteten Systeme setzt sich zunehmend der modellbasierte Test durch, der Stichproben nebst den erwarteten Ausgaben aus Modellen ableitet (Arbeiten dazu finden sich u.a. in [98, 108]). Die Stichprobenermittlung erfolgt hier automatisch. Modellbasierte Testverfahren besitzen den Vorteil, Differenzen zwischen Modell und Implementierung aufzudecken.

Verfahren des modellbasierten Tests lassen sich auf objektorientierte Systeme anwenden, wenn sie entsprechend angepaßt werden. Im Gegensatz zum Bereich der eingebetteten Systeme sind die Modelle in der objektorientierten Anwendungsdomäne meist noch von einem viel höheren Abstraktionsniveau[3]. Durch die Verwendung der UML als Modellierungssprache unterscheiden sie sich auch stark von den im Bereich der eingebetteten Systeme verwendeten Modellen (z.B. Zustandsgraphen von hohem Detaillierungsgrad oder MatLab-Simulink-Modelle).

Modelle, die mit Hilfe der UML erstellt werden, haben einen semiformalen Charakter und dienen in erster Linie als Dokumentationsmittel. UML-Modelle sind meist graphische Beschreibungen des Systems, die einen großen Interpretationsspielraum zulassen. Der UML-Standard [110] gibt einen semantischen Rahmen vor, der eine große Bandbreite möglicher Auslegungen (semantische Varianz) zuläßt. Um UML-Modelle für den Test zu nutzen, müssen sie speziell auf diese Funktion hin vorbereitet werden, so muß z.B. eine mathematisch präzise semantische Interpretation erfolgen.

Die UML definiert einen Satz verschiedener Diagrammtypen. Jeder der Diagrammtypen definiert eine Sicht auf das System. Da Informationen in UML-Modellen verstreut über mehrere Sichten vorliegen, muß ein Test möglichst viele Sichten berücksichtigen. Testverfahren, die nur eine Sicht unterstützen, bieten nur einen eingeschränkten Test des untersuchten Systems. Andererseits eignen sich nicht alle Diagrammtypen in gleicher Weise als Grundlage für die Stichprobenauswahl.

In der vorliegenden Arbeit wird als Lösung eine modellbasierte Technik zur automatischen Gewinnung der Stichprobe vorgeschlagen. Als Modelle dienen dabei trotz der vorgestellten Probleme UML-Modelle, da ihre weite Verbreitung im Bereich der objektorientierten Systeme eine Nutzung für den Test aus pragmatischen Gründen unumgänglich macht.

Um die Eignung verschiedener Diagrammtypen für die Stichprobengewinnung zu ermitteln, werden zunächst alle im UML-Standard definierten Diagrammtypen untersucht. Die ausgearbeitete Technik zur automatischen Gewinnung einer Stichprobe basiert auf den Diagrammtypen Sequenzdiagramm und Zustandsdiagramm. Sie ist jedoch auf einfache Weise um den Diagrammtyp Kommunikationsdiagramm erweiterbar. Damit werden in der vorliegenden Arbeit exemplarisch zwei Sichten zur Gewinnung einer Stichprobe herangezogen.

Den Diagrammtypen wird eine formale Semantik zugrunde gelegt, die die Menge der verwendbaren Modelle und deren Interpretation einschränkt. Trotz der Einschränkungen wird, wo es möglich ist, semantische Varianz zugelassen.

1.2 Problembereich der erwarteten Testergebnisse

Im Problembereich der erwarteten Testergebnisse sind zwei Fragen entscheidend:

1. Woher kommen die erwarteten Ergebnisse?

2. Wie werden die erwarteten Ergebnisse mit den vom System berechneten Ergebnissen verglichen?

Die erste Frage wird in der vorliegenden Arbeit analog zum Problembereich der Stichprobengewinnung beantwortet. Die erwarteten Ergebnisse werden aus den UML-Modellen gewonnen. Auch hier wird eine Kombination verschiedener Sichten als Basis verwendet. Die Untersuchung der unterschiedlichen UML-Diagrammtypen dient zur Auswahl der geeignetsten Diagrammtypen

[3] Auch wenn die verwendeten Modellierungssprachen einen deutlich höheren Detaillierungsgrad erlauben, ist für den Einsatzzweck meist die Definition von Modellen auf abstrakter Ebene ausreichend. Detaillierte Modelle werden sich nur durchsetzen, wenn ihre Definition einen Vorteil bei der Softwareentwicklung verspricht.

für die Stichprobengewinnung ebenso wie zur Auswahl der geeignetsten Diagrammtypen für die automatische Gewinnung der erwarteten Ergebnisse.

Zuvor muß jedoch die zweite Frage geklärt werden: In welcher Weise werden die erwarteten Ergebnisse mit den vom System produzierten Ergebnissen verglichen? Dazu ist zu berücksichtigen, daß objektorientierte Systeme einen auf die Objekte verteilten Zustand aufweisen. Der Zustand von Objekten wird durch Attributbelegungen bestimmt. Ein Test von Eingaben und eine Beobachtung von Ausgaben ist allein nicht ausreichend, da Methoden abhängig vom internen Zustand des betrachteten Objekts unterschiedlich reagieren. Viele Methoden besitzen überhaupt keinen Rückgabewert, sondern ändern nur den internen Zustand. Um einen Test möglichst effektiv zu machen, sollte der Test sich nicht auf das Ausgabeverhalten konzentrieren, sondern in erster Linie Zustände und Zustandsänderungen als Basis für den Vergleich zwischen erwarteten und berechneten Ergebnissen verwenden.

In der vorliegenden Arbeit werden OCL-Constraints in Form von Invarianten, Vor- und Nachbedingungen in Kombination mit Zustandsdiagrammen zur Gewinnung erwarteter Zustände verwendet.

1.3 Problembereich der Testcodeintegration

Der letzte Problembereich beim Test objektorientierter Systeme ist die Integration von Testcode in das zu testende System. Diese Integration wird auch als Instrumentierung bezeichnet. Testcode muß in Form von Testtreibern oder Testorakeln in das zu testende System eingefügt werden, da ein Test ohne Zugriff auf interne Strukturen des zu testenden Systems nur erschwert durchzuführen ist.

Testtreiber müssen das System gezielt steuern, indem sie es in einen bestimmten Zustand bringen und anschließend die Testfälle ausführen. Das Testorakel vergleicht Ausgaben und Zustände des zu testenden Systems mit den erwarteten Ausgaben und Zuständen und leitet daraus ein Testurteil ab. Die Hauptaufgaben sind also das Steuern von Eingaben, das Beobachten von Ausgaben und das Überwachen von Systemzuständen. Dazu müssen sowohl Testtreiber als auch Testorakel Zugriff auf das zu testende System besitzen und nach dem Test unter Umständen deaktiviert oder gar wieder entfernt werden.

Betrachtet man das Problem aus Sicht der aspektorientierten Programmiertechniken [57], so lassen sich Gemeinsamkeiten mit dem integrierten Testcode ausmachen. Die aspektorientierte Programmierung ist eine Erweiterung der objektorientierten Programmierung. Sie definiert den Begriff des *Cross-Cutting-Concerns*. Ein Cross-Cutting-Concern besitzt zwei Eigenschaften, er ist eng mit dem Rest des Systems verwoben (*Tangling*) und gleichzeitig über das gesamte System verstreut (*Scattering*). Ein Cross-Cutting-Concern läßt sich mit objektorientierten Zerlegungstechniken nicht modularisieren. Die aspektorientierte Programmierung unterstützt die Kapselung von Cross-Cutting-Concerns in eigene Module, die Aspekte genannt werden.

Mit Blick auf den Testcode in Form von Testtreibern und Testorakeln stellt man fest, daß diese einerseits mit dem zu testenden Code eng verwoben als auch über das gesamte zu testende System verstreut sind. Es handelt sich also um klassische Cross-Cutting-Concerns im Sinne der aspektorientierten Programmierung.

Deshalb wird in der vorliegenden Arbeit das Problem der Integration von zusätzlichem Testcode mit Hilfe aspektorientierter Programmiertechniken gelöst. Da es sich bei dem eingefügten Testcode um einen Crosscutting-Concern handelt, bringt der Einsatz aspektorientierter Techniken einen Vorteil gegenüber klassischen Instrumentierungstechniken.

1.4 Zusammenfassung des Lösungsansatzes

Der entwickelte Ansatz bietet Lösungen für alle zu Beginn identifizierten Problembereiche. Dabei werden Besonderheiten objektorientierter Systeme berücksichtigt. Durch die Kombination verschiedener Sichten ist ein effektiver Test objektorientierter Systeme möglich. Zudem wurde bei der Entwicklung auf eine Erweiterbarkeit um andere UML-Diagrammtypen und damit um weitere Sichten geachtet.

Der Fokus der entwickelten Technik ist, bedingt durch die Auswahl der Diagrammtypen, der Klassen- und Integrationstest. Die Arbeit unterstützt damit vorrangig den Test von feingranularen Artefakten, wie den Test einzelner Klassen oder den Test des Zusammenspiels einiger weniger Klassen. Ergänzt werden kann die vorgestellte Technik durch Arbeiten zum UML-basierten Systemtest, wie sie z.B. in [15] vorgestellt wurde. Die leichte Erweiterbarkeit läßt eine Integration von Diagrammtypen, die sich zum Systemtest eignen, ohne weiteres zu.

1.5 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in drei Hauptteile. Jeder Teil beginnt mit einer Einführung in das Thema und endet mit einer Zusammenfassung der Ergebnisse.

Im ersten Teil werden die Grundlagen des UML-basierten Tests vorgestellt. Zunächst werden in Kapitel 2 die für die vorliegende Arbeit relevanten Begriffe des Testens eingeführt. Kapitel 3 stellt die Probleme beim Test objektorientierter Softwaresysteme im Einzelnen vor. Kapitel 4 gibt einen Überblick über die im UML-Standard definierten Diagrammtypen und die mit ihnen modellierbaren Sichten. Schließlich werden in Kapitel 5 alle Standarddiagrammtypen auf ihre Eignung für den Test untersucht.

Teil zwei stellt die UML-basierten Lösungen für die Stichprobenauswahl und die Berechnung der erwarteten Ergebnisse vor. Dabei wird zunächst in Kapitel 7 die grundlegende Lösungsidee vorgestellt und ein Überblick über das entwickelte Testsystem UT^3 gegeben. Kapitel 8 stellt die Basisformalismen und eine Reihe von grundlegenden Algorithmen auf den Basisformalismen vor. Die nächsten beiden Kapitel widmen sich den technischen Details der entwickelten UML-basierten Lösungen, Kapitel 9 der Ermittlung der Stichprobe und Kapitel 10 den erwarteten Ergebnissen.

Im dritten Teil wird der Problembereich der Integration von Testcode in das zu testende System genauer untersucht. Kapitel 12 gibt zunächst einen Einblick in aspektorientierte Programmiertechniken und das in der vorliegenden Arbeit verwendete aspektorientierte ObjectTeams-Programmiermodell [82]. Eine Einführung in die Testcodeintegration unter Verwendung aspektorientierter Programmiertechniken bietet Kapitel 13. Die Umsetzung der Integration von Testcode mit Hilfe der aspektorientierten Sprache ObjectTeams/Java beschreiben die Kapitel 14 und 15.

Ein vierter und abschließender Teil faßt die Ergebnisse in Kapitel 17 zusammen und zeigt mögliche Weiterentwicklungen der vorgestellten Technik in Kapitel 18 auf.

Teil I

Grundlagen des UML-basierten Tests

Probleme beim Test objektorientierter Programme entstehen durch die Grundkonzepte der Objektorientierung: Datenkapselung, Vererbung und Polymorphie. Peter Rüppel [97]

Kapitel 2

Der Testprozess

Trotz aller sonstigen Qualitätssicherungsmaßnahmen ist der Softwaretest (kurz als Test bezeichnet) bei komplexen Softwaresystemen immer noch das einzige Mittel, um eine Aussage über die Qualität des Endprodukts, der Software, zu treffen.

Da es sich beim Test um ein Stichprobenverfahren handelt, stellt der Test in der Regel nicht die Korrektheit eines Softwaresystems sicher, sondern kann im strengen Sinn nur Aussagen über die getesteten Vertreter treffen[1].

Obwohl es Verfahren gibt, die die Korrektheit von Modellen und Software beweisen können, sind diese in der Regel nur auf triviale Implementierungen oder Modelle anwendbar.

Einige wenige Verfahren, wie beispielsweise das Modelchecking, können auch auf komplexere Produkte angewendet werden. Doch selbst, wenn sich die Korrektheit des implementierten Systems beweisen ließe, ist ein Test nötig, um das Verhalten in der Umgebung zu berücksichtigen, von der bei einem Beweis abstrahiert wurde.

Die Wichtigkeit des Tests zeigt sich auch in üblichen Vorgehensmodellen für den Softwareentwicklungsprozeß, wie z.B. dem V-Modell, in denen das Testen eine oder mehrere Phasen einnimmt.

Definition: *Softwaretest*
Der Softwaretest ist eine Aktivität, bei der das zu testende System oder Teile des zu testenden Systems mit Eingaben stimuliert und seine Reaktion beobachtet und bewertet wird.
Testen findet auf verschiedenen Granularitätsebenen statt:

- Der *Modul- oder Klassentest* testet die kleinsten testbaren Einheiten eines Systems, Module bzw. Klassen.

- Der *Integrationstest* testet die Interaktionen verschiedener Module bzw. Klassen durch sukzessives Aufbauen und Testen immer größerer Teilsysteme.

- Der *Systemtest* testet das System als Ganzes. Dabei werden auch nichtfunktionale Eigenschaften wie Performanz überprüft.

- Der *Abnahmetest* ist ein Systemtest in der realen Umgebung des Kunden.

Einige der wichtigsten Begriffe des Testens und ein Überblick über die einzelnen Aktivitäten beim Test werden in den folgenden Abschnitten erläutert.

[1] Die einzige Ausnahme bilden sehr kleine Systeme mit einer kleinen Eingabedomain, bei der alle möglichen Eingaben in endlicher und annehmbarer Zeit getestet werden können. Doch auch hier muß die Annahme getroffen werden, daß sich das zu testende System bei gleicher Eingabe immer gleich verhält.

2.1 Fehler und Fehlerursachen

Aufgabe des Tests ist das Finden von Fehlern im Softwareprodukt. Er ist abzugrenzen vom *Debugging*, das zur Lokalisierung von Fehlerursachen dient. Die Begriffe Fehler und Fehlerursache[2] sind streng voneinander abzugrenzen:

Definition: *Fehler*
Ein Fehler ist eine Differenz zwischen berechneten oder beobachteten oder gemessenen Werten und den spezifizierten Werten oder eine Abweichung des Verhaltens oder der Funktionalität von der Spezifikation.

Definition: *Fehlerursache*
Die Fehlerursache ist ein inkorrekter Schritt, Prozeß oder eine inkorrekte Datendefintion in der Implementierung oder eine Abweichung der Spezifikation vom gewünschten Verhalten oder der gewünschten Funktionalität. Das gewünschte Verhalten oder die gewünschte Funktionalität spiegelt die Erwartung des Kunden wider.

Fehler sind damit auf die Implementierung beschränkt, die Fehlerursache kann sowohl in der Spezifikation als auch in der Implementierung liegen. Einem Fehler liegen eine oder mehrere Fehlerursachen zugrunde, eine Fehlerursache verursacht keinen, einen oder mehrere Fehler. Das bedeutet, trotz einer Fehlerursache in der Implementierung wird eventuell kein Fehler verursacht, z.B. weil Fehlerursachen sich gegenseitig maskieren oder der fehlerhafte Teil der Implementierung nicht ausgeführt wird[3].

Zu beachten ist, daß in der vorliegenden Arbeit eine Übereinstimmung von Spezifikation und Implementierung, die jedoch nicht dem gewünschten Verhalten oder der gewünschten Funktionalität entspricht, nicht unter den Begriff Fehler fällt[4]. Diese Einschränkung ist der klaren Abgrenzung der Fehler von den Fehlerursachen geschuldet.

Mit der Definition der Begriffe Fehler und Fehlerursache ist nun eine präzise Definition der Aufgabe des Softwaretests möglich:
Aufgabe des Softwaretest ist das Finden von Fehlern. Der Softwaretest dient nicht des Findens von Fehlerursachen[5].

2.2 Testbarkeit von Software

Bevor näher auf die einzelnen Testaktivitäten eingegangen wird, soll hier eine kurze Einführung in die Testbarkeit von Software gegeben werden. Im Allgemeinen wird angenommen, daß objektorientierte Software schwerer testbar ist als imperative Software. Um jedoch ein Maß für die Testbarkeit zu finden und damit einen Vergleich überhaupt möglich zu machen, ist zunächst eine Definition der Testbarkeit nötig.

Der Begriff Testbarkeit beschreibt die relative Einfachheit und den relativen Aufwand zur Entdeckung von Fehlern [8]. Die relative Einfachheit beschreibt dabei, wie leicht sich Fehler als solche erkennen lassen, der relative Aufwand macht eine Aussage über die Ressourcen (z.B. Zeit),

[2] In der englischen Literatur finden sich eine Reihe von Termini für Fehler bzw. Fehlerursachen: *Fault*, *Failure*, *Bug*, *Mistake* und *Error*. Oft werden alle diese Begriffe mit dem Begriff *Fehler* übersetzt. Leider gibt es keine einheitliche Definition der Begriffe (vgl. dazu z.B. [52, 10]). Somit ist keine eindeutige Zuordnung der englischen Termini zu den Begriffen Fehler und Fehlerursache möglich.

[3] Ein besserer Terminus wäre *potentielle Fehlerursache* statt Fehlerursache.

[4] Die Abweichung der Spezifikation vom gewünschten Verhalten oder der gewünschten Funktionalität vom Verhalten oder der Funktionalität des Softwaresystems ist nicht mit dem Klassen-, Integrations- oder Systemtest zu entdecken. Die Abweichung wird entweder durch Verfahren der Prüfung der Spezifikation (z.B. Modelchecking oder Reviewtechniken) vor dem Test oder erst beim Abnahmetest oder im realen Einsatz beim Kunden aufgedeckt.

[5] Testen dient auch der Überprüfung der Softwarequalität. Gerade bei testgetriebenen Ansätzen steht der Test als entscheidendes Mittel für die Gewährleistung der Softwarequalität im Vordergrund.

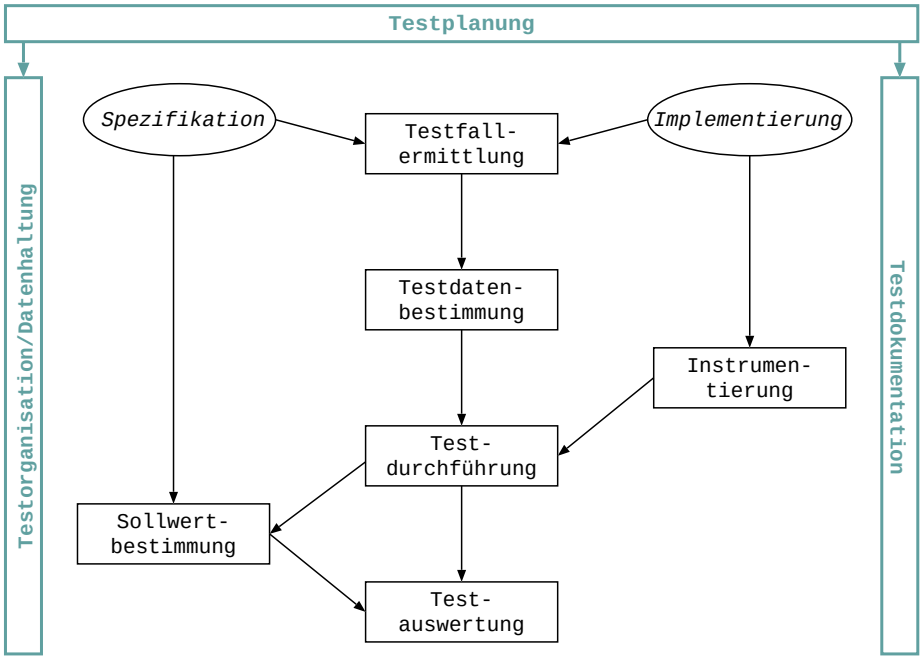


Abbildung 2.1: Testaktivitäten im Testprozeß

die investiert werden müssen, um Fehler zu finden. Ein System ist leichter testbar als ein anderes, wenn sich Fehler leichter und billiger entdecken lassen.

Da sich Systeme jedoch nicht so einfach miteinander vergleichen lassen - schließlich ist die Anzahl der Fehler in einem System nicht bekannt - müssen andere Kriterien oder Schlüsselaspekte dazu dienen, Testbarkeit faßbar zu machen.

Es lassen sich zwei Schlüsselaspekte finden, die Steuerbarkeit und die Beobachtbarkeit [8]. Das zu testende System muß die Kontrolle über Eingaben und interne Zustände von außen zulassen, also von außen steuerbar sein. Ausgaben und interne Zustände des zu testenden Systems müssen sich von außen beobachten lassen.

2.3 Testaktivitäten

Die einzelnen Schritte beim Testen werden Testaktivitäten genannt. Abbildung 2.1 gibt einen Überblick über die einzelnen Testaktivitäten[6].
Für jede Phase im Test (Modultest, Integrationstest, Systemtest) sind die folgenden Aktivitäten erneut zu durchlaufen:

Testfallermittlung. Man unterscheidet beim Testen die (abstrakten) Testfälle von den (konkreten) Testdaten. Ein Testfall ist eine Beschreibung einer bestimmten Eingabekonstellation und eines Zustands des zu testenden Systems inklusive des Testziels, also eine abstrakte Beschreibung der Stichprobe.

Wie in der Grafik dargestellt, können Testfälle sowohl aus der Spezifikation als auch aus der Implementierung abgeleitet werden. Die vorliegende Arbeit konzentriert sich auf die Ableitung spezifikationsbasierter Testfälle.

[6]Die Grafik ist entlehnt aus [39].

Testdatenbestimmung. Die Testdatenermittlung legt für jeden Testfall konkrete Werte fest. Dabei kann ein Testfall durch mehrere Testdaten instanziiert werden. Die Testdaten stellen die konkrete Stichprobe dar.

Instrumentierung. Als Instrumentierung bezeichnet man das Einfügen von zusätzlichem Code in das zu testende Softwaresystem. Dies ist nötig, um das zu testende System zu überwachen (*Monitoring*). Testcode dient darüber hinaus zur Auswertung, Initialisierung oder Ausführung der Tests (*Build-In-Tests*[7]).

Testdurchführung. Bei der Testdurchführung wird das zu testende System mit den vorher bestimmten Testdaten stimuliert und die vom zu testenden System erzeugten Ausgaben sowie Zustände in geeigneter Form protokolliert. In der Regel wird hierzu ein Softwaresystem eingesetzt, das *Testtreiber* genannt wird.

Ist das zu testende System nur unvollständig implementiert, so müssen für fehlende Teile Rahmen in Form von *Stubs* oder *Mock-Objekten* zur Verfügung gestellt werden. Stubs sind unvollständige Implementierungen der fehlenden Teile, die in der Regel Standardwerte zurückgeben. Mock-Objekte implementieren darüber hinaus bereits einen Zustand und geben zum Test passende Werte zurück [75]. Sie eignen sich daher besser für den Test objektorientierter Systeme als reine Stubs.

Sollwertbestimmung. Unter Sollwerten werden im Allgemeinen erwartete Ausgaben des zu testenden Systems verstanden. In der vorliegenden Arbeit wird dieser Begriff etwas weiter gefaßt, da im Gegensatz zu [39] neben Ein- und Ausgaben zusätzlich der Systemzustand betrachtet wird, der bei objektorientierten Systemen eine große Bedeutung besitzt (siehe auch Kapitel 3). Zudem wird das Verhalten des Systems betrachtet, also die Abfolge von Methodenaufrufen. Der Begriff Sollwert schließt somit hier die Begriffe Sollverhalten und Sollzustand ein, um die besondere Bedeutung des Systemzustands und der Zustände einzelner Objekte zu berücksichtigen. Die Erzeugung von Sollwerten wird auch als *Testorakel* bezeichnet.

Testauswertung. Um zu einem Testergebnis zu kommen, ist ein Vergleich zwischen erwarteten und erzielten Testergebnissen nötig. Es wird bewertet, ob der Test als bestanden zu werten (*pass*) oder als falsch zurückzuweisen (*fail*) oder nicht entscheidbar (*inconclusive*) ist. Darüber hinaus wird bestimmt, ob das gewählte Testkriterium (zum Beispiel im Form einer Überdeckungsmessung [70]) im gewünschten Maße erreicht worden ist.

Die einzelnen Testaktivitäten lassen sich mehr oder weniger gut automatisieren. Viele Werkzeuge leisten heute bereits gute Arbeit bei den Testaktivitäten Instrumentierung, Testdurchführung und Testauswertung.

Erfahrungen zeigen, daß sich die Aktivitäten Testfallermittlung und Sollwertbestimmung nur schwer automatisieren lassen. In jedem Fall erfordert die Automatisierung dieser Aktivitäten eine Spezifikation in formaler Form, da informelle Beschreibungen, zum Beispiel in textueller Form, nicht automatisch verarbeitet werden können.

Eine Zwischenstufe stellt die Automatisierung der Erzeugung von Testdaten dar. Liegen die Testfälle in geeigneter Form vor und ist die Domain der Testfälle durch ein Programm zu erfassen und aus dieser einzelne Werte zu erzeugen (z.B. bei Zahlen), so können für einzelne Testfälle passende Testdaten generiert werden.

Die vorliegende Arbeit konzentriert sich auf die schwer automatisierbaren Testaktivitäten Testfallerzeugung und Sollwertbestimmung (Testorakel), da es für die anderen Testaktivitäten bereits viele Lösungen zur Automatisierung gibt. Die Aktivität Testdatengewinnung wird in der vorliegenden Arbeit nur am Rande betrachtet, da sie einen eigenen Problembereich darstellt.

[7] Build-In Tests werden außerdem z.B. im Umfeld komponentenbasierter Software eingesetzt, um die Funktionalität der Software in unterschiedlichen Kontexten zu überprüfen.

2.4 Testfallerzeugung und Testdatenermittlung

Eines der Hauptprobleme des Tests ist die Auswahl der Stichprobe. Davon hängt entscheidend der Erfolg eines Tests ab, da durch die Stimulation mit der Stichprobe ein Fehler provoziert werden soll.

Die Güte der Stichprobe läßt sich leider nicht exakt messen, sondern nur abschätzen, weil die Anzahl der Fehler im zu testenden System unbekannt ist. Zwei verschiedene Testverfahren können selbst im Bezug auf das gleiche System nur dann miteinander in Beziehung gesetzt werden, wenn die Anzahl der Fehler bekannt ist. Dies ist oft nur bei speziell für den Test konstruierten Systemen der Fall. Die gewonnenen Aussagen lassen sich nicht auf andere Systeme übertragen, da die Art der Fehler hier vollkommen anders sein kann. Ein Verfahren kann viele, aber nur leichte Fehler finden, während ein zweites wenige, jedoch schwerwiegende Fehler aufdeckt.

Es gibt zwar einige Verfahren, die die Güte von Testfällen bestimmen, so z.B. Verfahren zur Messung von Codeüberdeckung (kontrollflußorientierte Verfahren, siehe [70]) oder Mutationstestverfahren, bei denen absichtlich Fehler in das zu testende System injiziert werden, um zu überprüfen, ob die gewählten Testfälle diese Fehler aufdecken können (siehe z.B. [58]). Kontrollflußorientierte Verfahren lassen sich jedoch nur schwer auf objektorientierte Systeme anwenden, da für diese, anders als für imperative Systeme, kein Kontrollflußgraph aufgestellt werden kann, der den Kontrollfluß des Gesamtsystems beschreibt [97]. Dafür ist hauptsächlich das dynamische Binden verantwortlich, das eine Aussage über den konkreten Kontrollfluß zur Laufzeit unmöglich macht. Mutationstestverfahren werden in der Regel für zu aufwendig erachtet.

Ein Versuch, sich dieser Problematik zu nähern, findet sich in [7]. Zunächst wird davon ausgegangen, daß die Menge aller (potentiell möglichen) Testfälle erschöpfend (*exhaustive*) ist. Jede (formale) Spezifikation assoziiert eine ideale erschöpfende Testfallmenge. Diese ist jedoch in der Regel unendlich, so daß ein Test auf dieser Basis unrealistisch erscheint.

Deshalb werden *Testhypothesen* eingeführt, die als Grundlage einer Auswahl der Testfälle aus der erschöpfenden Testfallmenge dienen. Die Testhypothesen reduzieren die Anzahl der Testfälle, ohne daß die Validität des Tests abnimmt. Das bedeutet, daß auch mit reduzierter Testfallanzahl dieselbe Menge von Fehlern gefunden wird. Die stärkste Hypothese geht dabei davon aus, daß das zu testende System korrekt ist und deshalb keine Testfälle nötig sind, die schwächste Hypothese nimmt an, daß keine Aussage über das zu testende System gemacht werden kann und deshalb in der Regel unendlich viele Testfälle nötig sind. Reale Testhypothesen bewegen sich im Rahmen dazwischen.

Ein *Testkontext* umfaßt die Testhypothesen, die mit Hilfe der Testhypothesen gewonnene Testfallmenge und das Testorakel, mit dem das Testergebnis überprüft wird, so daß Testfälle und Testorakel immer in Bezug zu den zugrundeliegenden Testhypothesen gesetzt werden.

Im Wesentlichen werden in [7] zwei Arten von Testhypothesen unterschieden, *Regularitätshypothesen* und *Uniformitätshypothesen*. Regularitätshypothesen sagen aus, daß alle Testsequenzen einer Länge k repräsentativ für alle Testsequenzen größer k sind. Uniformitätshypothesen sagen aus, daß eine Stichprobe aus der Menge aller Testfälle (oder einer Subdomain der Testfälle) repräsentativ für die anderen Testfälle (der Subdomain) ist.

Verallgemeinert man den Ansatz aus [7], so läßt sich folgende Aussage treffen:
Die Menge aller potentiellen Testfälle ist unendlich. Sie wird beschränkt durch die Anwendung einer oder mehrerer Testhypothesen.
Dabei ist die systematische Auswahl der Testfälle, also die Art der Testhypothese, auf dreierlei Art möglich[8]:

Funktionale Auswahl. Bei der funktionalen Auswahl erfolgt die Eingrenzung der Testfälle aufgrund der Funktion. Diese wird meist aus der Spezifikation oder den Anforderungen gewon-

[8] Ausgenommen sind Verfahren, die keine Testfälle, sondern direkt Testdaten erzeugen, wie Zufallstests oder evolutionäre Testverfahren.

nen. In diese Gruppe fallen zum Beispiel Testverfahren wie die funktionale Äquivalenzklassenbildung.

Strukturelle Auswahl. Die Auswahl von Testfällen wird anhand der Struktur der Implementierung entschieden. Beispiele hierfür sind alle Arten von Codeüberdeckungskriterien.

Probabilistische Auswahl. Bei der probabilistischen Auswahl werden Eigenschaften, Funktionen oder Sequenzen mit einer Wahrscheinlichkeit belegt, in der das System diese annimmt bzw. ausführt.

Viele Verfahren zur Erzeugung von Testdaten lassen sich nur schwer automatisieren, da in der Regel, besonders bei der funktionalen Auswahl, menschliches Wissen nötig ist. Einigkeit besteht darüber, daß sich Testfälle nur dann automatisch ableiten lassen, wenn die Spezifikation in formaler Form vorliegt, da nur diese von einem Algorithmus verarbeitet werden kann[9].

Bei der strukturellen und der probabilistischen Auswahl ist die Situation anders. Hier existieren diverse Verfahren zur automatischen Erzeugung von Testfällen. Ein Test auf der Basis von strukturell ausgewählten Testfällen ist jedoch nicht in der Lage, funktionale Fehler wie fehlende Funktionen zu finden. Die probabilistische Auswahl dient in der Regel zur Begrenzung einer unendlichen, aber bekannten oder generierbaren Menge von Testfällen. Die probabilistische Auswahl beschränkt sich deswegen auf wenige Anwendungsgebiete, z.B. zur Beschränkung unendlicher Testsequenzen[10].

Nach der Auswahl der Testfälle müssen diesen Testdaten zugeordnet werden. Auch hier ist ein systematisches Vorgehen angezeigt. Dabei wird jedem Testfall mindestens ein Testdatum zugeordnet. Da es sich bei den Testfällen bereits um abstrakte Beschreibungen der Testdaten handelt, ist eine manuelle und unter den oben genannten Einschränkungen auch eine automatische Zuordnung meist problemlos möglich.

Testdatenerzeugung ist auch ohne Testfälle möglich, z.B. durch zufällige Auswahl von Testdaten oder durch die Erzeugung von Testdaten durch evolutionäre Algorithmen wie in [105].

2.5 Testorakel

Das Testorakel, wie es u.a. von [10] verstanden wird, vergleicht die erwarteten Werte (Sollwerte) mit den tatsächlichen Werten (Istwerte) und leitet daraus ein Testurteil (*Verdict*) ab. Sollwerte sind klassischerweise Ausgaben, die das System auf eine Eingabe berechnet.

Eine Erweiterung des klassischen Sollwertbegriffs um Zustände und Verhalten ist beim Testen objektorientierter Systeme nötig. Zunächst sind objektorientierte Systeme zustandsbehaftet, machen jedoch ihren Zustand nach außen nur bedingt sichtbar. Zudem läßt sich aus den Ausgaben nicht immer auf den internen Zustand schließen. Folglich ist ein Test nur aufgrund eines Vergleich der Ausgaben unbefriedigend. Auch das Sollverhalten muß in bestimmten Fällen berücksichtigt werden, da sonst interne Abläufe dem Tester verborgen bleiben. Unter einem Ablauf wird hier die Reihenfolge von Methodenaufrufen und anderen Nachrichten im System verstanden. Dies ist besonders wichtig, um auch Ausnahmeverhalten beim Test zu berücksichtigen, da dieses oft keine Ausgaben produziert.

Obwohl viele Ansätze davon ausgehen, daß als Testurteil nur bestanden (*pass*) und fehlgeschlagen (*fail*) zu treffen ist[11], sollte pragmatischerweise auch *nicht entscheidbar* (*inconclusive*)

[9] Ein Versuch, auch aus natürlichsprachlichen Anforderungen Testfälle zu gewinnen, macht [35]. Dort werden die Anforderungen zwar in natürlicher Sprache, jedoch in sehr eingeschränkter Weise unter Beachtung von Regeln formuliert. Die Weiterverarbeitung der erzeugten Testfälle zu Testdaten erfordert jedoch manuelle Hilfe.

[10] Solche unendlichen Sequenzen werden z.B. von den in [98, 108, 60] beschriebenen Testverfahren erzeugt.

[11] So charakterisiert z.B. [67] das Testorakel als eine entscheidbare Relation, die für jedes Input-Output-Wertepaar (i, o) entscheiden kann, ob das Ergebnis der Berechnung des zu testenden Objekts $o = s(i)$ wahr oder falsch ist.

	System	Testorakel	gleiche Ursache	gleiches Ergebnis
1.	korrekt	korrekt	—	ja
2.	fehlerhaft	korrekt	—	nein
3.	fehlerhaft	korrekt	—	ja
4.	korrekt	fehlerhaft	—	ja
5.	korrekt	fehlerhaft	—	nein
6.	fehlerhaft	fehlerhaft	ja	ja
7.	fehlerhaft	fehlerhaft	ja	nein
8.	fehlerhaft	fehlerhaft	nein	ja
9.	fehlerhaft	fehlerhaft	nein	nein

Abbildung 2.2: Fehler in Testorakel und zu testendem System

als Testurteil zugelassen sein, wie es z.B auch in der Testnotationssprache TTCN-3 [29] vorgesehen ist[12]. Dabei gilt: *pass* < *inconclusive* < *fail*, d.h. das Testurteil inconclusive ist stärker als das Testurteil fail, aber schwächer als das Testurteil pass.

Wenn man von Testorakeln spricht, darf eine Schwäche nicht verschwiegen werden: Ein perfektes Testorakel kann es nicht geben.

Man stelle sich vor, ein Testorakel wüßte immer die korrekte Ausgabe und den korrekten Zustand des Systems. Handelt es sich in diesem Fall nicht bereits um eine korrekte Implementierung des geforderten Systems? Was also hindert uns daran, diese perfekte Implementierung statt des zu testenden Systems zu nehmen? Das Problem ist einerseits die Genauigkeit des Testorakels, andererseits muß auch die eventuelle Fehlerhaftigkeit des Testorakels berücksichtigt werden.

Das erste Problem ist die Genauigkeit des Testorakels. Ein Testorakel kann nicht immer den exakten Wert eines Testlaufs vorhersagen und diesen auch nicht in allen Fällen exakt mit dem erwarteten Wert vergleichen. Der exakte Wert ist oft nicht vorhersagbar, wenn sich das Ergebnis in einem bestimmten Intervall bewegen soll (z.B. bei reellen Zahlen), in manchen Fällen ist die exakte Vorhersagbarkeit eines Wertes auch nicht erwünscht (man denke nur an Zufallszahlengeneratoren). Der Vergleich ist oft schwierig, wenn das Ergebnis sich nicht exakt berechnen läßt. Auch komplexe Datentypen, wie sie z.B. überwiegend in der Objektorientierung vorkommen, lassen sich eventuell nur schwer miteinander vergleichen. So könnte z.B. ein rekursiver Vergleich einer Datenstruktur nötig sein, dieser ist jedoch aufgrund von Datenkapselung eventuell nicht durchführbar.

Das zweite Problem betrifft die Fehlerhaftigkeit des Testorakels. Man muß beim Auftreten eines Fehlers immer beide Seiten untersuchen, die Basis des Vergleichs waren, also in diesem Fall sowohl das zu testende System als auch das Testorakel. Normalerweise geht man davon aus, daß das zu testende System fehlerhaft ist. Es ist jedoch durchaus möglich, daß das Testorakel fehlerhaft ist. Dies trifft sowohl zu auf die (fehleranfällige) manuelle Überprüfung und Auswertung der Testergebnisse als auch auf automatische Testorakel, bei denen es sich wiederum um Software handelt, die potentiell Fehler enthält.
In der Tabelle in Abbildung 2.2[13] sind verschiedene Fälle aufgelistet, die bei einem Testlauf auftreten. Die Spalten für das zu testende System und das Testorakel geben an, ob diese korrekt oder fehlerhaft implementiert sind. Sind beide fehlerhaft, so wird in der nächsten Spalte aufgeführt, ob die Fehler die gleiche Ursachen besitzen. In der letzten Spalte schließlich wird angegeben, ob das zu testende System und die Implementierung das gleiche Ergebnis berechnen. Nicht angegeben ist, welches Testergebnis der Test liefert, da selbst bei fehlerhaftem System oder fehlerhaftem Testorakel das Testergebnis bestanden lauten kann, da es zudem von der gewählten Stichprobe abhängt.

[12] Dort gibt es noch eine weitere Möglichkeit der Beurteilung eines Tests, das Testurteil *error*, das einen Fehler nicht in der zu testenden Implementierung, sondern in der Testumgebung repräsentiert.
[13] Die Tabelle ist angelehnt an [10].

- Sind sowohl Testorakel als auch System korrekt, so liefert der Test ein valides Bestanden (Zeile 1).

- Ist das System fehlerhaft und das Orakel korrekt, so sind zwei Szenarien denkbar:

  Im ersten Fall liefern beide unterschiedliche Ergebnisse. Dann handelt es sich um ein valides Fehlschlagen (Zeile 2).

  Im zweiten Fall werden gleiche Ergebnisse geliefert. In diesem Fall ist das Testorakel zu schwach, um den Fehler zu erkennen (Zeile 3).

- Ist das System korrekt, das Testorakel jedoch fehlerhaft, dann kann der Test ungültig fehlschlagen (Zeile 5), oder zufällig bestanden werden, weil das Testorakel zufällig den richtigen Wert liefert, was jedoch unwahrscheinlicher ist (Zeile 4).

- Sind sowohl Testorakel als auch zu testendes System fehlerhaft, so ist zu unterscheiden, ob beide den gleichen Fehler enthalten (zum Beispiel aufgrund mißverstandener Spezifikation, Zeilen 6 und 7) oder ob beide aufgrund verschiedener Fehler voneinander abweichen (Zeile 8 und 9).

  Im ersten Fall ist die Wahrscheinlichkeit höher, daß beide das gleiche Ergebnis berechnen und der Test ungültigerweise bestanden wird (Zeile 6), im zweiten Fall ist es wahrscheinlicher, daß der Test verschiedene Ergebnisse liefert und damit ungültig fehlschlägt (Zeile 9).

Beide Probleme, die Genauigkeit des Tests als auch die Fehlerhaftigkeit des Testorakels, sind potentielle Fehlerquellen beim Test. Der Test führt dann entweder zu einem falsch-positiven oder zu einem falsch-negativen Testurteil.

Definition: *Falsch-positives Testurteil, falsch-negatives Testurteil*
Ist das Testorakel ungenau oder fehlerhaft und die zu testende Implementierung fehlerhaft, werden die Fehler in der zu testenden Implementierung eventuell nicht erkannt. Der Test liefert in diesem Fall ein falsch-positives Urteil. Ist das Testorakel fehlerhaft, die zu testende Implementierung jedoch korrekt, so werden eventuell Fehler gemeldet, die keine Fehler sind. Der Test liefert in diesem Fall ein falsch-negatives Urteil.
Beide Arten der Fehlerhaftigkeit des Testurteils treten beim Testen auf und lassen sich nicht verhindern.

Bei Testorakeln unterscheidet man oft nach dem Zeitpunkt der Berechnung [10]. Es gibt Testorakel, die Sollwerte vor, während oder nach Ausführung des Tests berechnen. Alle diese Ansätze haben Vor- und Nachteile und unterstützen unterschiedliche Arten der Gewinnung der Sollwerte.

- Bei der Berechnung vor der Ausführung des Tests ist es möglich, den Vergleich bereits zur Laufzeit auszuführen. Es müssen jedoch bereits vor der Testausführung alle Details der Testläufe vorliegen, wie die genauen Daten und Aufrufe, die das System stimulieren.

- Eine Berechnung der Sollwerte zur Laufzeit unterstützt vor allem Tests gegen andere Systeme mit der gleichen Spezifikation, zum Beispiel Vorgängerversionen des zu testenden Systems, ausführbare Spezifikationen oder Systeme mit gleicher Funktionalität. Auch Zusicherungen können als *Built-In-Test*[14] des Systems zur Laufzeit fungieren.

  Eine detaillierte Kenntnis der Testläufe ist nicht erforderlich. Somit ist auch ein einfacher Einsatz von Capture-Replay-Tools möglich, die exemplarische Testläufe aufnehmen und diese zu Regressionstestzwecken wieder abspielen können. Die aufgenommenen Testläufe können sofort bewertet werden.

[14] Als Built-In-Test bezeichnet man zusätzlich in das System eingefügten Code, der nur zu Testzwecken dient.

- Eine Berechnung nach dem Testlauf ist immer dann nötig, wenn es eine bekannte Umkehrfunktion der Funktionalität des zu testenden Systems gibt (Beispiel: Wurzelfunktion). In diese werden die Ergebnisse des Tests nach dem Testlauf eingesetzt. Ein Vergleich der Ergebnisse der Umkehrfunktion mit den Eingaben an das zu testende System dient zur Bewertung des Tests.

Kapitel 3

Testen objektorientierter Software

Das objektorientierte Programmierparadigma orientiert sich an einer objektzentrierten Sicht der Welt. Im Gegensatz dazu implementieren imperative Sprachen Folgen von Anweisungen (Kontrollflußorientierung) und orientieren sich funktionale Sprachen an mathematischen Funktionen. Neben Konzepten, die auch in anderen Programmierparadigmen eingesetzt werden, wie der Datenkapselung, führt die Objektorientierung neue Konzepte ein, wie Vererbung und dynamisches Binden. Dieser Abschnitt soll einen grundlegenden Einblick in die Objektorientierung bieten und beschreibt insbesondere Eigenschaften objektorientierter Systeme, die beim Test relevant werden.

3.1 Eigenschaften objektorientierter Systeme

Das zentrale Konzept in der objektorientierten Softwareentwicklung ist das Objekt, das Eigenschaften in Form von *Attributen*[1] und Verhalten in Form von *Methoden* kapselt. Methoden sind immer einem Objekt zugeordnet (Ausnahme: Klassenmethoden). Dieses kann innerhalb der Methode als das aktuelle Objekt referenziert werden[2].

Jedes Objekt kapselt seinen Zustand und verbirgt ihn nach außen (Geheimnisprinzip). Der Zugriff auf das Objekt erfolgt nur über die Methoden. Methoden können klassifiziert werden in zustandsneutrale Methoden (*Query-* oder *Observer-Methoden*), die keinen Einfluß auf den Zustand eines Objekts haben, und zustandsändernde Methoden (*Update-Methoden*), die den Zustand eines Objekts ändern.

Spezielle Methoden sind *Getter-* und *Setter-Methoden*. Dabei liefern Getter-Methoden den aktuellen Zustand (aber nicht unbedingt die aktuelle Wertebelegung der Attribute), ohne ihn zu ändern und gehören damit zu den Observer-Methoden. Mit Hilfe von Setter-Methoden ist in bedingtem Maße ein Setzen des Zustands möglich, sie gehören zu den Update-Methoden.

Objektorientierte Sprachen unterstützen eine Referenzsemantik, das heißt, es werden im Gegensatz zur Wertesemantik die Werte nicht direkt manipuliert oder kopiert, sondern jeder Zugriff auf ein Objekt geschieht über eine der Referenzen, die auf das Objekt zeigen.

Als *Aliasing* wird die mehrfache Referenzierung desselben Objekts in einem Programm bezeichnet. Oft wird der Begriff *Rolle* (auch als *Sicht* bezeichnet) synonym zu einer dieser Referenzen gebraucht, daß heißt, eine Rolle ist die Benutzung eines Objekts innerhalb eines Programmkontextes.

Mit dieser Eigenschaft objektorientierter Systeme ist es möglich, Objekte innerhalb eines Kontextes zu ändern und die Änderungen in allen anderen Kontexten zu propagieren. Auch wenn

[1] Attribute werden auch als *Instanzvariablen* oder *Membervariablen* bezeichnet. Diese Begriffe werden in der vorliegenden Arbeit synonym verwendet.

[2] Die Referenz auf das aktuelle Objekt wird ausgedrückt durch ein Schlüsselwort, das je nach Sprache unterschiedlich benannt wird, z.B. `self` in OCL, `this` in Java und C++ oder `Current` in Eiffel.

es Situationen gibt, in denen dies zu Problemen führen kann, ermöglicht diese Art der Programmierung eine vereinfachte Implementierung. In objektorientierten Sprachen muß zum Beispiel zur Manipulation eines Objekts in einer Liste nur eine Referenz dieses Objekts nach außen gegeben werden. Jede Änderung des Objekts außerhalb wirkt sich auch auf das Objekt in der Liste aus, da es sich um dasselbe Objekt handelt. In Sprachen ohne Referenzsemantik ist ein Kopieren des Objekts zur Änderung und anschließend ein erneutes Einfügen des Objekts in die Liste nötig.

Zur Strukturierung des Programms dienen Klassen, die Schablonen für Objekte bilden. Ein Objekt ist immer genau einer Klasse zugeordnet. Diese Zuordnung wird bei der Objekterzeugung (*Instanziierung*) festgelegt und kann bei den meisten objektorientierten Sprachen zur Laufzeit nicht mehr geändert werden. Man spricht auch davon, daß ein Objekt den *Typ* der Klasse besitzt, von der es erzeugt wurde. Dies ist der *dynamische Typ* eines Objekts.

Objektorientierte Sprachen erweitern das Typkonzept dahingehend, daß Objekte zusätzlich noch einen oder mehrere *statische Typen* annehmen können. Die Grundlage ist das Konzept der *Vererbung*, mit dem eine Reihe weiterer Konzepte wie das *dynamische Binden*, *Polymorphie* und *Casting* in Verbindung stehen.

Vererbung ist zunächst ein Konzept der Wiederverwendung von Software. Erbt eine Klasse von einer anderen, so übernimmt sie alle Eigenschaften der Klasse, von der sie erbt. Es können in der erbenden Klasse neue Eigenschaften und neues Verhalten eingeführt oder das geerbte Verhalten modifiziert werden. Die erbende Klasse wird als *Subklasse* oder *Unterklasse* bezeichnet, die Klasse, von der geerbt wird, als *Superklasse*, *Oberklasse* oder *Basisklasse*.

Polymorph sind in einer typisierten objektorientierten Sprache die Entitäten. Der Typ der Entität ist dabei der statische Typ des Objekts, der in der Regel zur Compilezeit überprüft wird[3]. Der dynamische Typ des Objekts, von dem es erzeugt wurde, ist dagegen in der Regel nur zur Laufzeit überprüfbar, so daß es auch in typisierten objektorientierten Sprachen trotz einer Compilezeitprüfung zu Typfehlern zur Laufzeit kommen kann. So kann zum Beispiel das Casting, das Zuweisen eines neuen statischen Typs zur Laufzeit, zu einem Fehler führen, wenn der dynamische Typ des Objekts nicht konform zum neuen statischen Typ ist.

Es gibt verschiedene Definitionen der Konformität, die auch von den Möglichkeiten der benutzten Sprache abhängen. Die einfachste Definition sagt aus, daß ein Typ `Y` konform zu einem Typ `X` ist, wenn der Typ `Y` direkt oder indirekt vom Typ `X` erbt.

[73] gibt eine darüber hinaus gehende Definition der Konformität, die als Liskov-Konformität bezeichnet wird. Ein Objekt `x` ist genau dann konform zu einem Objekt `y`, wenn in jedem Kontext, in dem `x` eingesetzt wird, auch der Einsatz von `y` möglich ist.

[80] verfeinert die Liskov-Konformität in einer Weise, die insbesondere für die Sprache Eiffel relevant ist, da sie die Redefinition von Methodensignaturen voraussetzt. Eine Redefinition von Methodensignaturen wird in den meisten anderen objektorientierten Sprachen nicht unterstützt. Die Redefinition von Methodensignaturen darf nur in folgender Weise vorgenommen werden: Sei `Y` eine Unterklasse von `X` und die Methode `m'` aus `Y` die Redefinition der Methode `m` aus `X`. Dann gilt für jeden Typ `I'` eines Parameters der Methode `m'` und den Typ `I` des entsprechenden Parameters der Methode `m`, daß `I'` gleich `I` ist oder `I'` ist Oberklasse von `I`. Dann gilt für jeden Typ `O'` eines Rückgabewerts der Methode `m'` und den Typ `O` des entsprechenden Rückgabewerts der Methode `m`, daß `O'` gleich `O` ist oder `O'` ist Unterklasse von `O`.

Gleichzeitig findet sich in [80] eine Definition von Konformität in Bezug auf Verträge zwischen Client und Server (*Design by Contract*, siehe dazu auch Abschnitt 4.3.2).

Objektorientierte Systeme unterscheiden sich folglich grundlegend von imperativen Systemen, denen eine funktionsorientierte Sicht zugrunde liegt. Die Eigenschaften objektorientierter Systeme sind im Hinblick auf den Test besonders zu berücksichtigen.

[3]Polymorphe Entitäten gibt es auch in objektorientierten Sprachen, in denen Entitäten keinen statischen Typ besitzen (z.B. Smalltalk). Die Gefahr von Laufzeitfehlern ist in diesem Sprachen größer als in Sprachen mit typisierten Entitäten, da der Typ zur Compilezeit noch nicht bekannt ist und somit nicht überprüft werden kann. Diese Sprachen liegen jedoch außerhalb des Fokus der vorliegenden Arbeit.

3.2 Objektorientierung und Test

Die Besonderheiten objektorientierter Software, die zu einer Erleichterung für den Programmierer führen, erschweren den Test, weshalb traditionelle Testverfahren für imperative Sprachen nicht ohne weiteres übertragbar sind [97]. Neben den in diesem Zusammenhang meist genannten Eigenschaften Vererbung, dynamisches Binden und Datenkapselung in Verbindung mit dem Geheimnisprinzip betrifft das insbesondere den Zustand des Systems und der einzelnen Objekte sowie zyklische Abhängigkeiten zwischen Klassen.

Vererbung. Durch Vererbung wird der Testaufwand enorm gesteigert. Da bereits getestete Methoden im Kontext der Unterklasse ein anderes Verhalten aufweisen können, ist ein erneutes Testen aller Methoden, auch der unverändert geerbten, im Kontext der Unterklasse nötig. Wird konform vererbt, sind zwar die Testfälle des Tests der Oberklasse wiederverwendbar, nicht jedoch die Testergebnisse [97]. Es muß derselbe Programmcode in allen Kontexten auf seine Funktionsweise geprüft werden, also in allen Unterklassen erneut getestet werden. Beim Integrationstest empfiehlt [90] das Testen von Client-Server-Verbindungen mit allen beteiligten Klassen, also auch allen Unterklassen des Clients sowie des Servers, in allen möglichen Kombinationen.

Dynamisches Binden. Entitäten im Programm können zur Laufzeit einen anderen Typ annehmen als den zur Compilezeit bekannten statischen Typ. Das führt zu versteckten und unerwarteten Kontrollflüssen, wenn zur Laufzeit die jeweils passendste Methode durch dynamisches Binden ausgewählt und ausgeführt wird. Traditionelle kontrollflußorientierte Testverfahren zur Überdeckungsmessung und Testfallermittlung lassen sich nicht ohne weiteres auf objektorientierte Systeme übertragen. Als Lösung werden z.B. analog zu kontrollflußorientierten Testverfahren Überdeckungskriterien für Zustandsautomaten vorgeschlagen [101].

Datenkapselung. Zum Testen ist neben einer Kontrolle der Eingaben eine Beobachtbarkeit der Ergebnisse essentiell. Datenkapselung schränkt die Beobachtbarkeit des Systemverhaltens ein. Ein Test kann zwar die Ausgaben beobachten, nicht jedoch den internen Zustand des Objekts[4]. Da objektorientierte Systeme zustandsbehaftet sind, viele für den Test relevante Vorgänge sich also im Innern der Objekte abspielen, ist eine Beobachtung interner Zustände nötig. Verschiedene Lösungen werden für dieses Problem vorgeschlagen, von der Verwendung interner Testorakel [10] bis zur Verwendung von aspektorientierter Programmiertechniken, die privilegierten Zugriff auf die adaptierten Objekte bieten (siehe Kapitel 13).

Systemzustand. Der Zustand objektorientierter Systeme ist über seine Objekte verteilt. Das impliziert zwangsläufig die Idee, daß ein Test objektorientierter Systeme sich auf den Test seiner Klassen beschränken könnte. Diese Annahme ist falsch. Obwohl der Systemzustand auf die Objekte verteilt ist, kann von der Korrektheit der Einzelteile nicht auf das Zusammenspiel der Komponenten geschlossen werden. Aus denselben Komponenten, hier Klassen, lassen sich beliebig viele verschiedene Systeme konfigurieren.

Testergebnisse der einzelnen Objekte lassen nicht auf Testergebnisse des gesamten Systems schließen, ebensowenig macht ein Test des Systems Aussagen über den Test seiner Komponenten. Der Test der Teile berücksichtigt keine Schnittstellen und kein Zusammenspiel der einzelnen Komponenten. Der Test des Systems führt u.U. Funktionalität seiner Teile nicht aus, so daß sich über diese keine Aussage fällen läßt.

Objektzustand. In zustandslosen Systemen genügt es, zum Test nur die Ein- und Ausgaben zu betrachten. Auch beim Test zustandsbehafteter Systeme wird im allgemeinen eine Black-Box-Sicht zugrunde gelegt, so daß nur Ein- und Ausgaben betrachtet werden und aus diesen

[4] Auch wenn einige Autoren die Ansicht vertreten, beim Test nur den von außen sichtbaren Zustand eines Objekts heranzuziehen, halte ich die Beobachtung von internen Zuständen für essentiell, z.B. für den Test von Setter- und Getter-Methoden, das Setzen von Zuständen oder den Ausschluß von Fehlermaskierungen.

auf den internen Zustand des Systems geschlossen wird. Diese Betrachtung scheint für objektorientierte Systeme unbefriedigend. Viele objektorientierte Systeme produzieren nur wenige Ausgaben, die Reaktionen auf Eingaben in Form von Methodenaufrufen werden in der Regel versteckt in Instanzvariablen als Objektzustände gespeichert. Wird zum Beispiel Geld auf ein Konto eingezahlt, reagiert das System nicht sofort mit einer Rückmeldung über den Kontostand. Erst eine Abfrage des aktuellen Kontostands, der Teil des Objektzustands ist, kann zu einer Entscheidung über Bestehen oder Fehlschlagen eines Tests führen. Deshalb ist es essentiell für die Aussagekraft des Tests, Objektzustände zu berücksichtigen.

Abhängigkeiten. Objektorientierte Systeme enthalten oft Zyklen in der Klassenstruktur. Klassische Ansätze zum Integrationstest, wie Bottom-Up- oder Top-Down-Integration, lassen sich nicht ohne weiteres auf objektorientierte Systeme übertragen (siehe dazu auch [10]).

Hinzu kommt, daß in der Regel nur eine mangelhafte Spezifikation für das entwickelte objektorientierte System erstellt wurde. Neben textueller Darstellung ist inzwischen die Modellierung mit der UML üblich. Die erstellten Modelle beinhalten dabei gleichzeitig Modellierung und Spezifikation des Systems, liegen meist in graphischer Form vor und besitzen einen semiformalen Charakter. Eine genaue Übersicht über die Modelle der UML und ihre Nutzung für den Test beschreibt das Kapitel 5.

Testverfahren für objektorientierte Systeme müssen diese Besonderheiten berücksichtigen. Insbesondere sind Lösungen zur Verbesserung der Beobachtbarkeit gefragt.

Im Folgenden werden einige populäre Testansätze für objektorientierte Systeme vorgestellt und ihre Leistungsfähigkeit in Bezug auf Anwendbarkeit und Unterstützung der besonderen objektorientierten Eigenschaften untersucht.

3.3 Populäre Testansätze für objektorientierte Systeme

In den letzten Jahren haben viele Forschungsarbeiten sich mit dem Thema des objektorientierten Tests auseinandergesetzt. Eine Auswahl der Testansätze stellt dieser Abschnitt vor.

Zunächst ist ein Großteil der Testansätze für objektorientierte Systeme dem Black-Box- oder spezifikationsbasierten Test zuzuordnen, da sich klassische White-Box-Testverfahren - wie Überdeckungsmessungen auf dem Sourcecode - nur bedingt auf objektorientierte Systeme übertragen lassen (siehe Ausführungen zum dynamischen Binden in Abschnitt 3.2). Eine Ausnahme bilden Testverfahren, die die Überdeckungsmessung auf Methodenebene beschränken. In der Regel werden die Kriterien Anweisungsüberdeckung (C0) und Zweigüberdeckung (C1) angeboten. Die Aussagekraft der Überdeckungsmessung ist begrenzt, da Methoden in der Regel klein sind und sich keine Pfade von Programmbeginn bis zum Programmende wie bei imperativen Programmen definieren lassen. In diese Kategorie fallen Werkzeuge wie *clover* [18].

Zu den funktionalen Testverfahren gehören die Äquivalenzklassenbildung, die oft mit der Grenzwertanalyse kombiniert wird (beide beschrieben in [81]), und zustandsbasierte Testverfahren.

Die Äquivalenzklassenbildung gehört zu den klassischen Testverfahren, läßt sich jedoch im objektorientierten Test einsetzen. Dabei wird der Eingabedatenraum in Äquivalenzklassen partitioniert unter der Annahme, daß sich das zu testende System für jeden beliebigen Vertreter aus einer Äquivalenzklasse gleich verhält. Jede Kombination von Äquivalenzklassen aller Eingabewerte ist ein Testfall. Aus jeder Äquivalenzklasse wird ein Vertreter als Testdatum gewählt.

Da die Erfahrung gezeigt hat, daß Fehler gehäuft an den Grenzen der Äquivalenzklassen auftreten, wird die Äquivalenzklassenbildung durch die Grenzwertanalyse ergänzt. Bei der Grenzwertanalyse liegt ein Schwerpunkt des Tests auf den Grenzen der Äquivalenzklassen. Es wird für jede Grenze je ein Wert auf der Grenze und mindestens ein Wert möglichst nah an der Grenze als Testdatum gewählt.

[10] erweitert die Grenzwertanalyse durch den Domaintest, der zwei Vorteile gegenüber dieser besitzt. Der Domaintest gibt einerseits Richtlinien vor, wie sich Grenzwerte auf objektorientierte

Datentypen übertragen lassen, andererseits reduziert er durch geschickte Kombination von Testdaten die Anzahl der Testfälle. Für Grenzen objektorientierter Datentypen werden entweder Objekte auf einfache Datentypen abgebildet[5] oder die Zustände eines Objekts als Grundlage genommen. Die Anzahl der Testdaten wird reduziert, da beim Test der Grenze eines Eingabewerts für alle anderen Eingabewerte typische Vertreter im gültigen Bereich gewählt werden[6] und zwischen offenen und geschlossenen Grenzen unterschieden wird. Dadurch werden bei Relationen genau zwei Werte an der Grenze getestet, ein Wert innerhalb und ein Wert außerhalb der Grenze. Bei einer geschlossenen Grenze handelt es sich bei dem gewählten Vertreter für den gültigen Bereich um den Grenzwert, bei einer offenen Grenze wird der Grenzwert als Vertreter des ungültigen Bereichs gewählt.

Zustandsbasierte Testverfahren schlagen u.a. [10, 79, 77] zum Test objektorientierter Systeme vor. Aus diesen werden durch (vollständige) Traversierung Testfälle abgeleitet. Zudem lassen sich Überdeckungskriterien aus den kontrollflußbasierten Testverfahren einfach auf Zustandsautomaten übertragen, da es sich bei den Zustandsautomaten um Graphen handelt (siehe z.B. [100]).

Eine Technik, Zustandsautomaten und Kontrollflußgraphen miteinander zu kombinieren, stellt [9] vor. Bei diesem Verfahren für den Klassentest werden Events in Form von Methodenaufrufen im Zustandautomaten durch den entsprechenden Kontrollflußgraphen der Methode ersetzt. Anschließend dient dieser erweiterte Zustandsautomat als Basis des Tests.

Zu den populärsten Verfahren der letzten Zeit gehören wohl die aus dem Extreme-Programming-Umfeld stammenden Unit-Tests. Als bekannteste Werkzeug ist hier *JUnit* zu nennen. Bei JUnit handelt es sich um ein reines Werkzeug zur Testausführung und Testauswertung, das jedoch um viele Erweiterungen ergänzt werden kann, so z.B. um die Integration von Mock-Objekten oder mit Hilfe von clover um Überdeckungsmessungen. Die Testfallerzeugung bleibt jedoch unklar. Das auch bei Einsatz dieses Werkzeugs zunächst sinnvolle Testfälle erstellt werden müssen, z.B. mit Hilfe der Äquivalenzklassenbildung, zeigt das Beispiel in [32].

Immer größere Bedeutung nehmen inzwischen Testverfahren ein, die auf Zusicherungen und dem *Design by Contract* [80] basieren. U.a. in [79] und [26] werden Ansätze zur Generierung von Testfällen / Testdaten und Testorakeln aus Vor- und Nachbedingungen von Methoden vorgeschlagen. Dabei werden sowohl Testfälle gebildet, die die Vorbedingung erfüllen, als auch Testfälle, die die Vorbedingung nicht erfüllen.

Die Techniken aus [12] und [3] erzeugen effizient Testdaten für komplexe Datentypen, z.B. Listen oder Bäume. Vor- und Nachbedingungen dienen hier als Testorakel.

3.4 Ein einfaches Testmodell für objektorientierte Systeme

Im Rahmen der vorliegenden Arbeit wurde ein einfaches Testmodell entwickelt. Es soll dazu dienen, den objektorientierten Test zu beschreiben.

Sei I die Input- und O die Outputdomain des zu testenden Systems x[7]. Dabei gilt $I = N \times W$ und $O = N \times W$, daß heißt, Eingaben und Ausgaben bestehen jeweils aus einer Nachricht N und einer Menge von Werten W. Sei S die Menge aller Zustände, die das zu testende System annehmen kann. Ein Zustand ist dabei charakterisiert durch eine Attributbelegung der Objekte im System[8]. Das zu testende System kann betrachtet werden als eine Funktion

$$x : S \times I \rightarrow S \times O.$$

Das zu testende System besitzt einen Vorzustand $s \in S$ und wird mit Hilfe einer Eingabe oder einer Folge von Eingaben $i \in I$ stimuliert. Das zu testende System nimmt nach der Berechnung

[5] Dies ist einfach möglich bei numerischen Datentypen wie Datums-, Zeit- und Währungstypen.
[6] Es werden also nie zwei Grenzen gleichzeitig getestet.
[7] Synonym hier auch verwendet für zu testendes Objekt oder zu testende Komponente.
[8] Die Attributbelegung kann unterschiedlich interpretiert werden (siehe Kapitel 4.2.3).

den Nachzustand $s' \in S$ an und liefert eine Folge von Ausgaben $o \in O$. Ein Test ist immer insofern total, daß es nie unendlich lange Testläufe geben kann. Eine potentiell unendliche Berechnung wird im Test durch den Einsatz von Timeouts totalisiert. Das Testorakel liefert für jedes Quintupel (x, s, i, s', o) genau einen Wert aus der Menge $\{pass, fail, inconclusive\}$. Damit ist das Testorakel total und deterministisch, da immer entschieden werden kann, ob der Test bestanden, fehlgeschlagen oder unentscheidbar ist. Das Testorakel *to* kann somit als eine Funktion

$$to : X \times S \times I \times S \times O \rightarrow \{pass, fail, inconclusive\}$$

verstanden werden, wobei X die Menge aller vom Testorakel *to* überprüfbaren Systeme ist.

Orientiert man den Test an diesem Testmodell, so sind Eingaben an das System (Testdaten) Folgen von Nachrichten. Das System befindet sich zu Beginn des Tests in einem definierten Zustand. Auf die Nachrichtenfolge reagiert das System wiederum mit einer Folge von Ausgaben und nimmt einen definierten Zustand an. Durch das Testmodell nicht beschrieben werden Ausgaben ohne vorherigen Stimulus (spontane Outputs) und gleichzeitige Eingaben an das System, die im Kontext der vorliegenden Arbeit nur eine untergeordnete Rolle spielen.

Die in der vorliegenden Arbeit entwickelten Verfahren werden anhand dieses Testmodells auf ihre Eignung für den objektorientierten Test überprüft.

Kapitel 4

Die Unified Modeling Language

Die Unified Modeling Language (UML) wurde von der Object Management Group [88] entwickelt und hat sich in den letzten Jahren weitgehend als ein Standard zur Modellierung objektorientierter Systeme durchgesetzt. Aktuell ist die Version 2.0 [110], die im Gegensatz zur Vorgängerversion UML 1.5 [109] viele neue Notationen integriert und die Semantik weiter präzisiert. Die Modellierung mit der UML erfolgt weitgehend auf der Basis graphischer Notationen durch eine Reihe von Diagrammtypen, ergänzt durch eine formale, textuelle Sprache, die Object-Constraint Language (OCL).

Ein Grund für die weitgehende Akzeptanz der UML ist die Integration unterschiedlicher populärer objektorientierter Modellierungssprachen und -techniken in die UML[1], wie z.B. die *Object Modeling Technique* (OMT) [95], die Booch-Methode [11] und *Object-Oriented Software Engineering* (OOSE) [53]. Hinzu kam eine objektorientierte Variante der Statecharts von Harel [43]. Zusammen definieren die verschiedenen integrierten Notationen eine Reihe von Standarddiagrammtypen.

Ein anderer Grund für die Popularität der UML ist sicherlich in ihrer Flexibilität zu suchen. Die UML ist bewußt offen spezifiziert und bietet diverse Erweiterungsmöglichkeiten. So kann die UML ohne weiteres auf andere Applikationsdomänen übertragen werden, was UML-Erweiterungen wie UML-RT [40] für Realzeitsysteme, UML^h [34] für hybride Systeme oder UFA [46] für aspektorientierte Programmierung zeigen.

Die Möglichkeit der Erweiterung ist Teil der Sprachdefinition der UML. Neben den von der UML vordefinierten Standarddiagrammtypen ist die UML um weitere Diagrammtypen ergänzbar und die Semantik der Standarddiagrammtypen veränderbar. Dazu stellt die UML verschiedene Mechanismen zur Verfügung, *Tagged Values*, *Constraints*, *Stereotypen* und *Profile*.

Constraints sind Bedingungen, deren Formulierung für jedes beliebige Modellelement möglich ist. Eine Notation von Constraints wird entweder in natürlicher Sprache oder in OCL vorgenommen.

Tagged Values sind Attribut-Wert-Paare. Dabei wird ein Attribut mit einem Wert belegt. Sie sind notationell schwer von Constraints zu unterscheiden, wenn diese eine Gleichheit beschreiben.

Stereotypen erweitern das Vokabular der UML um neue Begriffe und neue Semantik. Sie werden auf bestehende Modellierungselemente angewendet.

Profile sammeln Stereotypen, die eine Anwendung der UML in einem bestimmten Problembereich ermöglichen. Neben dem UML Testing Profile [111] sind für die UML 2.0 u.a. Profile für CORBA, Quality of Service und Fehlertoleranz sowie für Performanz und Zeitverhalten von der OMG als Standard spezifiziert worden.

[1] Die zuerst verfolgte Absicht der Integration zu einer einheitlichen Methode (*Unified Method*) wurde sehr bald zugunsten einer einheitlichen Notation aufgegeben.

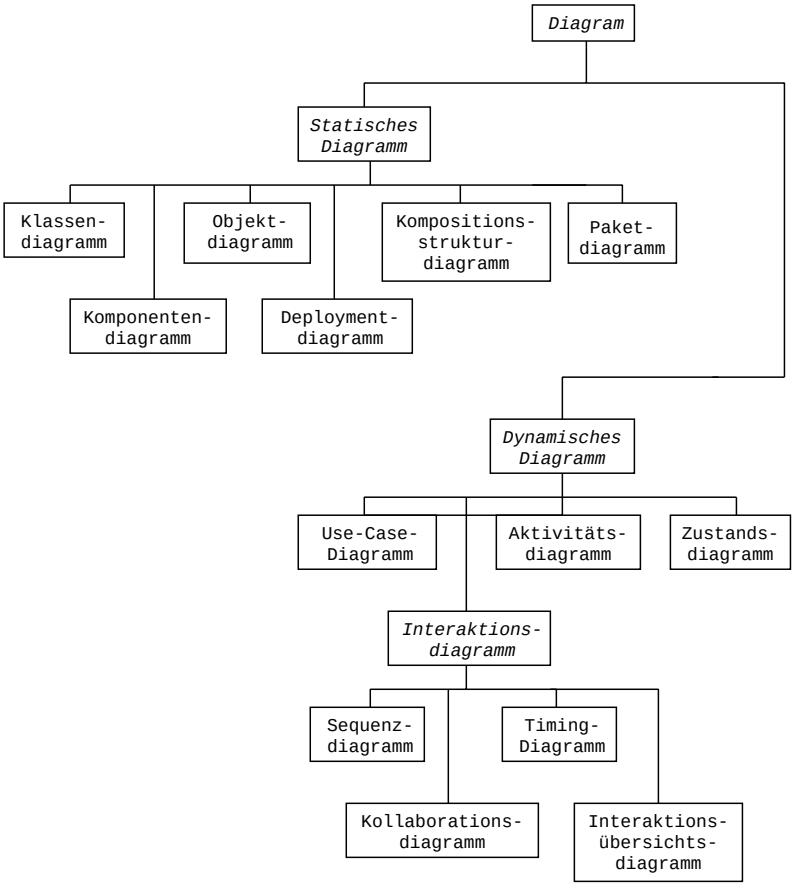


Abbildung 4.1: UML-Standarddiagrammtypen

Die Offenheit der UML wird zudem durch semantische Variationspunkte (*Semantic Variation Points*) weiter unterstützt. So liegt es in der Entscheidung des Anwenders der UML, welche der möglichen semantischen Varianten er seinen Modellen zugrunde legt. Einige Punkte sind bewußt semantisch offen gehalten, in diesen Fällen gibt es keinerlei Vorgaben als Hilfe für den Anwender. An anderen Stellen nennt die UML beispielhaft mögliche Ausprägungen der semantischen Variationspunkte.

In der UML sind 13 Standarddiagrammtypen enthalten, die jedoch mit Hilfe der Erweiterungsmechanismen um eigene Diagrammtypen ergänzbar sind. Es gibt statische (Struktur-) und dynamische (Verhaltens-) Diagrammtypen (siehe Abbildung 4.1). Diese werden im Einzelnen in den folgenden Abschnitten beschrieben, da eine Übersicht für eine Analyse der Testbarkeit der einzelnen Diagrammtypen wichtig ist.

4.1 Statische Diagrammtypen

Statische oder Strukturdiagramme beschreiben die Programmstruktur oder die Organisation eines Systems. Zu den statischen Diagrammtypen gehören das Klassen-, das Objekt-, das Komponenten-, das Deployment-, das Kompositionsstruktur- und das Paketdiagramm.

4.1.1 Klassendiagramm

Das Klassendiagramm ist ein zentraler Diagrammtyp in der UML. Es modelliert Beziehungen (Assoziationen) zwischen Klassen. Viele Werkzeuge sind in der Lage, aus dem Klassendiagramm direkt Codeschablonen für die Klassen im implementierten System zu erzeugen, so daß der Modellierung mit Hilfe des Klassendiagramms auch in der Praxis eine große Bedeutung zukommt. Das Klassendiagramm dient in erster Linie der Übersicht des zu entwickelnden Systems. Aus diesem Grund sollte es alle relevanten Strukturzusammenhänge und Klassen enthalten. Meist wird jedoch aufgrund der Anzahl der Klassen in einem realen System keine Vollständigkeit des Klassendiagramms angestrebt.

Klassen sind dabei Modelle der im realen System vorkommenden Klassen und besitzen Eigenschaften in Form von Attributen und Verhalten in Form von Methoden. Jede Klasse besitzt darüber hinaus einen eindeutigen Namen. Neben instanziierbaren Klassen erlaubt die UML auch die Modellierung abstrakter Klassen und (mit Hilfe eines Stereotyps) die Modellierung von Interfaces oder Datentypen.

Verschiedene Arten der Assoziation haben Einfluß auf den Charakter der Kopplung zwischen Klassen. Assoziationsbeziehungen werden mit Multiplizitäten versehen, die die Anzahl der beteiligten Objekte auf Instanzebene auf beiden Seiten einer Assoziation einschränken.

Neben einfachen Assoziationen, gerichteten als auch ungerichteten, gibt es die Assoziationsarten Aggregation und Komposition. Eine Aggregation bezeichnet eine Teil-Ganzes-Beziehung, bei der das Aggregat die aggregierten Objekte (Komponenten) enthält. Eine Komposition ist eine strenge Form der Aggregation, bei der jede Komponente genau einem Aggregat zugeordnet ist und die Komponenten vom zugeordneten Aggregat erzeugt und zerstört werden.

Eine Sonderstellung nimmt die Generalisierung-/Spezialisierungsbeziehung[2] ein, die eine Supertyp-/Subtyp-Beziehung darstellt. Alle anderen Arten der Beziehung zwischen Klassen betreffen auf Instanzebene mehrere (mindestens zwei) Objekte verschiedenen oder gleichen Typs. Eine Generalisierung-/Spezialisierungsbeziehung aber resultiert auf Instanzebene nur in einem Objekt.

4.1.2 Objektdiagramm

Das Objektdiagramm beschreibt eine konkrete Ausprägung (Instanz) der Klassensicht. Es gibt eine bestimmte Objektkonstellation zu einem bestimmten Zeitpunkt zur Laufzeit des Systems wieder. Im Gegensatz zum Klassendiagramm, das Übersichtscharakter besitzt, ist ein Objektdiagramm nur eine von vielen möglichen Konstellationen, besitzt also Beispielcharakter.

Im Objektdiagramm werden den Objekten konkrete Attributbelegungen zugeordnet. Mengenbeziehungen zwischen Objekten werden besser verdeutlicht. Zur Darstellung der Objekte und ihrer Beziehungen zueinander wird eine Untermenge der im Klassendiagramm verwendeten Symbole benutzt. Im Objektdiagramm sind diese jedoch auf der Instanzebene zu verstehen. So ist eine Beziehung zwischen zwei Objekten im Objektdiagramm eine Instanz einer Assoziation aus dem Klassendiagramm.

4.1.3 Komponentendiagramm

Das Komponentendiagramm zeigt Organisation und Abhängigkeiten der Komponenten des Systems. Die Modellierung mit dem Komponentendiagramm berücksichtigt haupsächlich technische Aspekte. Modelliert werden unter anderem angebotene und benötigte Schittstellen einer Komponente. Das Komponentendiagramm unterstützt insbesondere die Entwicklung komponentenbasierter Systeme.

[2] Manchmal eigentlich semantisch nicht korrekt als Vererbungsbeziehung bezeichnet.

4.1.4 Kompositionsstrukturdiagramm

Das Kompositionsstrukturdiagramm verfeinert die abstrakte Sicht des Komponentendiagramms. Modelliert wird das Innenleben einzelner Komponenten, wobei eine Komponente nicht nur im technischen Sinne verstanden wird, es kann sich auch um eine Klasse oder ein Teilsystem handeln. Das Kompositionsstrukturdiagramm unterstützt vorangig die Top-Down-Modellierung des Systems.

4.1.5 Deploymentdiagramm

Das Deploymentdiagramm ist noch stärker technisch orientiert als das Komponentendiagramm. Modelliert wird die Laufzeitumgebung des Systems in Form von Servern, Datenbanken und zugrundeliegender Hardware. Die Komponenten des Systems werden auf diese Laufzeitumgebung verteilt. Modelliert wird unter anderem, auf welchem Knoten welche Komponente des Systems laufen soll. Das Abstraktionsniveau ist sehr hoch, was sich auch in der begrenzten Darstellungsform mit wenigen Notationselementen widerspiegelt.

4.1.6 Paketdiagramm

Das Paketdiagramm faßt Modellelemente logisch zusammen. Dies dient einerseits der besseren Übersicht über das System, andererseits erfolgt hier bereits auf Modellebene eine Aufteilung des Systems in Pakete, die sich auch auf Programmiersprachenebene wiederfinden kann.

4.2 Dynamische Diagrammtypen

Dynamische Diagramme, oft auch als Verhaltensdiagramme bezeichnet, werden genutzt, um das Programmverhalten zu beschreiben. Zu ihnen gehören das Use-Case-, das Aktivitäts-, das Zustandsdiagramm und vier verschiedene Typen von Interaktionsdiagrammen. Interaktionsdiagramme umfassen die Diagrammtypen Sequenz-, Kommunikations-[3], Timing- und Interaktionsübersichtsdiagramm.

4.2.1 Use-Case-Diagramm

Das Use-Case-Diagramm beschreibt die Außensicht auf ein System. In der Regel wird es bereits sehr früh im Softwareentwicklungszyklus als Beschreibungsmittel für die vom System angebotene Funktionalität genutzt, deshalb ist das Abstraktionsniveau hoch und der Informationsgehalt gering. Die Notationsmittel sind sehr einfach gehalten.

Es werden Anwendungsfälle des Systems definiert, die in der Regel durch weitere Diagrammtypen verfeinert werden, z.B. durch Sequenzdiagramme. Jedem Anwendungsfall sind ein oder mehrere beteiligte Akteure zugeordnet. Ein Akteur kann dabei sowohl ein Endbenutzer als auch ein anderes (Software-) System sein. Anwendungsfälle werden unter Verwendung verschiedener Stereotypen zueinander in Beziehung gesetzt oder näher spezifiziert.

4.2.2 Aktivitätsdiagramm

Das Aktivitätsdiagramm eignet sich gut zur Modellierung von Datenflüssen im System. Abläufe werden durch Bedingungen, Schleifen, Verzweigungen, Parallelität und Synchonisation sehr detailliert dargestellt. Ein Aktivitätsdiagramm ist zur Modellierung auf verschiedenen Abstraktionsebenen geeignet, z.B. zur Modellierung eines einzelnen Algorithmus oder eines ganzen Prozesses.

[3] In der UML 2.0 von Kollaborations- in Kommunikationsdiagramm umbenannt.

4.2.3 Zustandsdiagramm

Beim Zustandsdiagramm, auch UML-Statechart genannt, handelt es sich um eine objektorientierte Variante der Statecharts von Harel [43], die wiederum aus einer Kombination von Moore- und Mealy-Automaten entstanden. Wichtigste Neuerung im Unterschied zu den Vorgängern ist bei den Harel-Statecharts die Einführung der Hierarchie und damit verbunden die Definition von Historie und Parallelität. UML-Statecharts unterscheiden sich von den Harel-Statecharts durch einige Änderungen in der Syntax der verwendeten Symbole und in der Ausführungssemantik. Auf die Unterschiede zwischen UML- und Harel-Statecharts wird hier jedoch nicht im Einzelnen eingegangen.

In der UML werden Zustandsdiagramme für unterschiedliche Sichten verwendet. Man kann mit ihnen Objektlebenszyklen ebenso modellieren wie Protokolle oder Systemverhalten. Sie sind geeignet, sowohl interne Zustände eines Objekts als auch eine Außensicht zu spezifizieren.

Ein Statechart besteht aus hierarchisch gegliederten Zuständen, die eine Baumstruktur bilden. Dabei werden die Zustände, die wiederum Zustände enthalten, komponierte Zustände genannt (*Composite States*), Zustände ohne Unterzustände Basiszustände (*Simple States*). Unterzustände werden zu Regionen zusammengefaßt. Komponierte Zustände enthalten eine oder mehrere Regionen, wobei jede Region wiederum ein Statechart bildet.

Eine Zuordnung von Zustandsinvarianten zu Zuständen ist möglich. Dabei wird jedem Zustand ein Prädikat zugeordnet, das die Bedingung angibt, die gelten muß, wenn sich das zugeordnete Objekt in dem entsprechenden Zustand befindet. Zustände sind disjunkt und vollständig, Zustandsinvarianten nicht notwendigerweise.

Es gibt einen ausgezeichneten Initial- oder Startzustand für jede Hierarchiestufe. Es kann einen oder mehrere Endzustände geben. Zuständen werden bei Bedarf Aktionen zugeordnet, die bei Eintritt, während der Zeit des Verbleibens im Zustand oder bei Austritt ausgeführt werden.

Zwischen den Zuständen gibt es Zustandsübergänge (Transitionen), entweder auf einer Hierarchieebene oder zwischen verschiedenen Hierarchieebenen. Eine Transition wird durch ein Ereignis (*Event*) getriggert. Die Ausführung einer Transition ist durch Bedingungen einschränkbar, sowohl durch Vorbedingungen als auch durch Nachbedingungen. Bei Ausführung einer Transition ist die gleichzeitige Ausführung einer Aktion möglich.

Der Zustand, von dem eine Transition ausgeht, wird Quellzustand der Transition genannt, der Zustand, in dem die Transition endet, Zielzustand. Transitionen zwischen Zuständen, die im Zustandsdiagramm auf verschiedenen Hierarchieebenen liegen, bezeichnet man als Interlevel-Transitionen.

Die UML stellt durch *Fork* und *Join* Sprachmittel zur Verfügung, mit denen die Verzweigung und Vereinigung von Transitionen modelliert werden kann.

Ein Zustandsdiagramm befindet sich zu jedem Zeitpunkt in genau einer Zustandskonfiguration, auch aktiver Zustand oder aktive Zustandskonfiguration genannt. Welche Transition oder welche Menge konfliktfreier Transitionen in einer aktuellen Zustandskonfiguration ausgeführt wird (schaltet), um zu einem Nachzustand zu kommen, hängt bei mehreren möglichen Mengen konfliktfreier Transitionen von der Semantik ab. In der UML wird in diesem Fall immer die innerste Transition gewählt. Stehen mehrere Transitionen auf derselben Ebene in Konflikt zueinander, wird eine nichtdeterministische Auswahl getroffen.

Eine Transition wird immer aufgrund des zugehörigen Ereignisses ausgewählt, das aktuell anliegt. Treten mehrere Ereignisse gleichzeitig auf, werden diese in eine Menge aller anliegenden Ereignisse einsortiert. Anschließend wird ein Ereignis aus dieser Menge nichtdeterministisch ausgewählt und alle zugehörigen Transitionen berechnet, die potentiell schalten können. Gibt es keine Transition, deren Ausführung möglich ist, wird das Ereignis verworfen und das nächste Ereignis aus der Menge ausgewählt. Schaltet eine Transition, werden durch die dazugehörende Aktion neue Ereignisse erzeugt und der Menge hinzugefügt.

Eine Besonderheit sind die aufgeschobenen Ereignisse (*deferred Events*). Ein Ereignis, das im

Zustandsdiagramm anliegt, aber zum aktuellen Zeitpunkt nicht verarbeitet werden kann, wird wieder in die Menge der aktuellen Ereignisse einsortiert, wenn es als aufgeschoben gekennzeichnet ist. In jedem weiteren Schritt ist zu prüfen, ob ein Verarbeiten des aufgeschobenen Ereignisses inzwischen möglich ist. Sollte dies der Fall sein, ist es anderen Ereignissen vorzuziehen.

Ein Zustandsdiagramm akzeptiert zu jedem Zeitpunkt Ereignisse und blockiert nicht, auch wenn zu einem Zeitpunkt kein Ereignis anliegt oder eine Verarbeitung des aktuell anliegenden Ereignisses nicht möglich ist.

Die Semantik eines Zustands variiert je nach Einsetzzweck, folgende Zustandsdefinitionen sind möglich:

Interner konkreter Zustand. Definiert durch reale Variablen- oder (in der Objektorientierung) Attributbelegung.
Beispiel: Jeder potentielle Wert einer Integer-Variablen bildet genau einen Zustand.

Interner abstrakter Zustand. Definiert durch Partionierung der Domains der Attribute, die den internen konkreten Zustand definieren.
Beispiel: Die Domain einer Integer-Variable ist z.B. in positive und negative oder alternativ in gerade und ungerade Werte partitionierbar.

Äußerer konkreter Zustand. Nach außen sichtbarer Teil des internen konkreten Zustands, da in der Objektorientierung meist ein Zugriff auf alle internen Variablen nicht möglich ist.
Beispiel: Das Objekt besitzt zwei Attribute vom Typ Integer, von denen nur eines von außen sichtbar ist.

Äußerer abstrakter Zustand. Nach außen sichtbarer Teil des internen abstrakten Zustands.
Beispiel: Das Objekt, in dem die Integervariable spezifiziert ist, gibt nicht deren Wert nach außen, sondern nur den Betrag des Wertes. Es ist von außen nicht unterscheidbar, ob der Wert der entsprechenden Variablen negativ oder positiv ist.

Äußerer hypothetischer Zustand. Vermuteter Zustand des Objekts. Da der interne Zustand nicht bekannt ist, wird der Zustand über das Input-/Output-Verhalten des Systems abgeleitet.

Die UML kennt zwei Arten von Zustandsdiagrammen, die *Behavioral State Machines* und die *Protocol State Machines* [110].

Die *Behavioral State Machines* ähneln stärker den Statecharts von Harel. Sie können zur Spezifikation des Verhaltens von Objekten oder Systemen sowohl in Bezug auf *Call Events* (Methodenaufrufe) als auch *Signal Events* (asynchrone Nachrichten) dienen.

Protocol State Machines dienen zur Spezifikation von Protokollen[4], zum Beispiel der Spezifikation der Aufrufreihenfolge von Methoden oder der Abfolge erlaubter Zustandsübergänge. Dabei ist eine mögliche Anwendung die Spezifikation von Objektlebenszyklen.

Protocol State Machines sind eine spezielle Variante der *Behavioral State Machines* mit folgenden Einschränkungen:

- Der Kontext einer *Protocol State Machine* ist immer eine Klassifizierung, z.B. eine Klasse in einem UML-Klassendiagramm. Als Kontext einer *Behavioral State Machine* ist auch eine Verhaltenseigenschaft erlaubt.
- Ereignisse sind in der Regel Methodenaufrufe (*Call Events*). Signale (*Signal Events*) sind nur eingeschränkt zugelassen.
- Es werden keine Aktionen an den Transitionen und keine Aktionen oder Aktivitäten in den Zuständen modelliert. Semantischer Hintergrund ist, daß als Aktion des Aufrufs einer Methode die Ausführung dieser Methode gilt.

[4] In diesem Fall ist es möglich, für jeden Port einer Komponente ein anderes Protokoll zu spezifizieren.

- Statt einer Aktion ist die Modellierung von Nachbedingungen an den Transitionen möglich.
- Methoden, die vom Zustandsdiagramm nicht referenziert werden, lösen keine Zustandsänderungen aus.

Bei Methoden, die im Zustandsdiagramm referenziert werden, jedoch nur in einigen Zuständen eine Transition auslösen, ist ein Aufruf in allen andern Zuständen undefiniert. Dies entspricht der Verletzung einer (impliziten) Vorbedingung[5]. Der Aufruf der Methode wird in diesem Fall entweder ignoriert, zurückgewiesen oder zurückgestellt (siehe *deferred Events*), es kann eine Ausnahme geworfen oder das System mit einem Fehler terminiert werden.

Der Zielzustand einer Transition entspricht einer (impliziten) Nachbedingung eines Aufrufs der die Transition auslösenden Methode[6]. Zu beachten ist die Abhängigkeit der impliziten Nachbedingung vom Quellzustand der Transition.

4.2.4 Sequenzdiagramm

Das Sequenzdiagramm hat in der UML 2.0 durch die Integration der aus dem Telekommunikationsbereich bekannten *Message Sequence Charts* eine vollständige Überarbeitung erfahren. Dadurch hat die Komplexität des einzelnen Sequenzdiagramms stark zugenommen. Gleichzeitig nimmt durch die kompaktere Modellierung die Übersichtlichkeit des Gesamtmodells zu, weil deutlich weniger Sequenzdiagramme zur Modellierung desselben Sachverhalts benötigt werden.

Sequenzdiagramme dienen zur Modellierung des Informationsaustauchs zwischen verschiedenen Objekten. Zeitliche Abläufe lassen sich sehr präzise darstellen.

Ein Sequenzdiagramm stellt die an einem Szenario beteiligten Objekte dar. Jedem Objekt ist eine Lebenslinie (*Lifeline*) zugeordnet. Von einem Objekt können Nachrichten an ein anderes Objekt versendet werden, das Auftreten dieser Nachrichten wird auch als Event bezeichnet. Eine Folge von Nachrichten wird Sequenz genannt.

In Sequenzdiagrammen wird explizit synchrone oder asynchrone Kommunikation modelliert. Die Zeit, die zwischen zwei Nachrichten vergehen darf bzw. die Zeit, die eine Nachricht vom Sender zum Empfänger maximal verbrauchen darf, ist modellierbar.

Durch die Einführung der aus den Message Sequence Charts bekannten *Interaktionsoperatoren* wird u.a. die Schachtelung von Interaktionsdiagrammen möglich. Es lassen sich z.B. Alternativen und Nebenläufigkeit, aber auch negative Sequenzen definieren.

Interaktionsoperationen werden sogenannten *Combined Fragments*, Regionen im Sequenzdiagramm, zugeordnet. Die Operanden eines Interaktionsoperators sind Sequenzen, die bei mehrstelligen Interaktionsoperatoren durch einen Separator, getrennt sind[7]. Es gibt folgende Interaktionsoperatoren:

alt. Bei der Alternative wird einer der möglichen Operanden ausgeführt. Die Auswahl des Operanden kann durch die explizite Angabe einer Bedingung unterstützt werden. In diesem Fall ist auch die Modellierung einer alternativen Sequenz möglich, die ausgeführt wird, wenn keine der Bedingungen der anderen Operanden wahr ist. Ist keine Bedingung angegeben, so wird eine implizite Bedingung angenommen, die nicht näher spezifiziert ist, die Auswahl erfolgt in diesem Fall nichtdeterministisch.

opt. Bei der Option wird der Operand entweder ganz ausgeführt oder gar nicht. Die Option entspricht einer Alternative mit zwei Operanden, von denen einer leer ist und der andere dem Operanden der Option entspricht.

break. Mit der Unterbrechung wird eine Alternative zwischen der nach der Break-Region modellierten Nachrichtenfolge der umgebenden Region und der in der Break-Region modellierten Sequenz definiert.

[5] Gleichzeitig muß jedoch auch die explizit modellierte Bedingung an der betrachteten Transition erfüllt sein.

[6] Gleichzeitig muß jedoch auch die explizit modellierte Nachbedingung an der betrachteten Transition erfüllt sein.

[7] Im folgenden werden Interaktionsoperator und Interaktionsoperand kurz als Operator und Operand bezeichnet.

par. Der Paralleloperator erlaubt eine beliebiges Merging (Interleaving) der beteiligten Sequenzen. Es kann sowohl zuerst die erste vor der zweiten Sequenz ausgeführt werden als auch umgekehrt. Eine andere mögliche Ausführungsreihenfolge ist jeweils eine Nachricht der ersten Sequenz im Wechsel mit einer Nachricht der zweiten Sequenz. Andere Reihenfolgen sind ebenso möglich, wenn dabei die Abfolge der Nachrichten innerhalb eines Operanden erhalten bleibt.

strict. Das sogenannte *Strict Sequencing*, mit dem Operator `strict` deklariert, beschreibt eine strenge Abfolge der Nachrichten der Operanden. Nachrichten des ersten Operanden werden vor den Nachrichten des zweiten Operanden versendet.

seq. Das sogenannte *Weak Sequencing*, ausgedrückt durch den Operator `seq`, erzwingt im Gegensatz zum Parallelitätsoperator eine Reihenfolge, die jedoch Freiheitsgrade besitzt. Innerhalb eines Operanden muß die Reihenfolge der Nachrichten erhalten bleiben. Events, die vom selben Objekt ausgehen, sich also auf einer Lebenslinie befinden, aber sich in verschiedenen Operanden befinden, müssen ebenfalls in der angegebenen Reihenfolge auftreten. Events in verschiedenen Operanden und auf verschiedenen Lebenslinien können beliebig gemergt werden.

Weak Sequencing entspricht der Parallelität, wenn die Operanden disjunkt in Bezug auf die beteiligten Objekte sind, und Strict Sequencing, wenn nur ein Objekt Nachrichten versendet.

loop. Mit dem Operator `loop` kann ein wiederholtes Auftreten der gleichen Sequenz markiert werden. Es ist möglich, eine obere und untere Schranke für die Anzahl der Wiederholungen anzugeben.

neg. Der Operator `neg` kennzeichnet invalides Verhalten, d.h. Sequenzen von Nachrichten, die in der angegebenen Reihenfolge nicht auftreten dürfen. Alle anderen möglichen Sequenzen mit Ausnahme der negativen sind laut UML-Semantik valide und damit ausführbar, auch wenn sie nicht explizit modelliert sind.

Leider läßt die Verwendung des Negationsoperators viel Interpretationsspielraum (siehe zum Beispiel [106]). Es ist, neben den verschiedenen Interpretationen, zu diskutieren, ob der Negationsoperator ausschließlich als äußerster Operator eines Sequenzdiagramm verwendbar ist oder besser ein Attribut eines Sequenzdiagramms ist als ein Operator einer Region [107].

critical. Nachrichtensequenzen einer kritischen Region werden nicht durch Events anderer Regionen unterbrochen, selbst wenn diese mit dem Parallelitätoperator oder ähnlichem zu einem Combined Fragment verbunden sind.

ignore. Der Operator `ignore` zeigt an, daß Teile einer Sequenz ignoriert werden. Dies ist vor allem sinnvoll, wenn Sequenzen referenziert werden, aber nur Teile dieser referenzierten Sequenzen im referenzierenden Kontext relevant sind.

consider. Der Operator `consider` kennzeichnet Nachrichten als besonders wichtig. Alle nicht mit `consider` gekennzeichneten Nachrichten werden in diesem Fall ignoriert. Damit trifft der Operator `consider` eine positive Auswahl, während der Operator `ignore` eine Auswahl durch Ausschluß trifft.

assert. Die modellierte Nachrichtenfolge ist eine Zusicherung in dem Sinn, daß alle anderen Nachrichtenfolgen invalide sind. Oft wird der Zusicherungsoperator in Kombination mit den Operatoren `ignore` und `consider` verwendet. Auch hier läßt sich diskutieren, ob der Operator `assert` nicht eher ein Attribut eines Sequenzdiagramms als ein Operator ist [107].

Einfache Sequenzdiagramme ohne Operatoren lassen sich in Kommunikationsdiagramme überführen. Mit der Einführung der geschachtelten Sequenzdiagramme ist eine einfache Abbildung von Sequenz- auf Kommunikationsdiagramm nur noch begrenzt möglich, da die Ausdrucksmächtigkeit des Sequenzdiagramms die des Kommunikationsdiagramms übersteigt. Der umgekehrte Weg ist

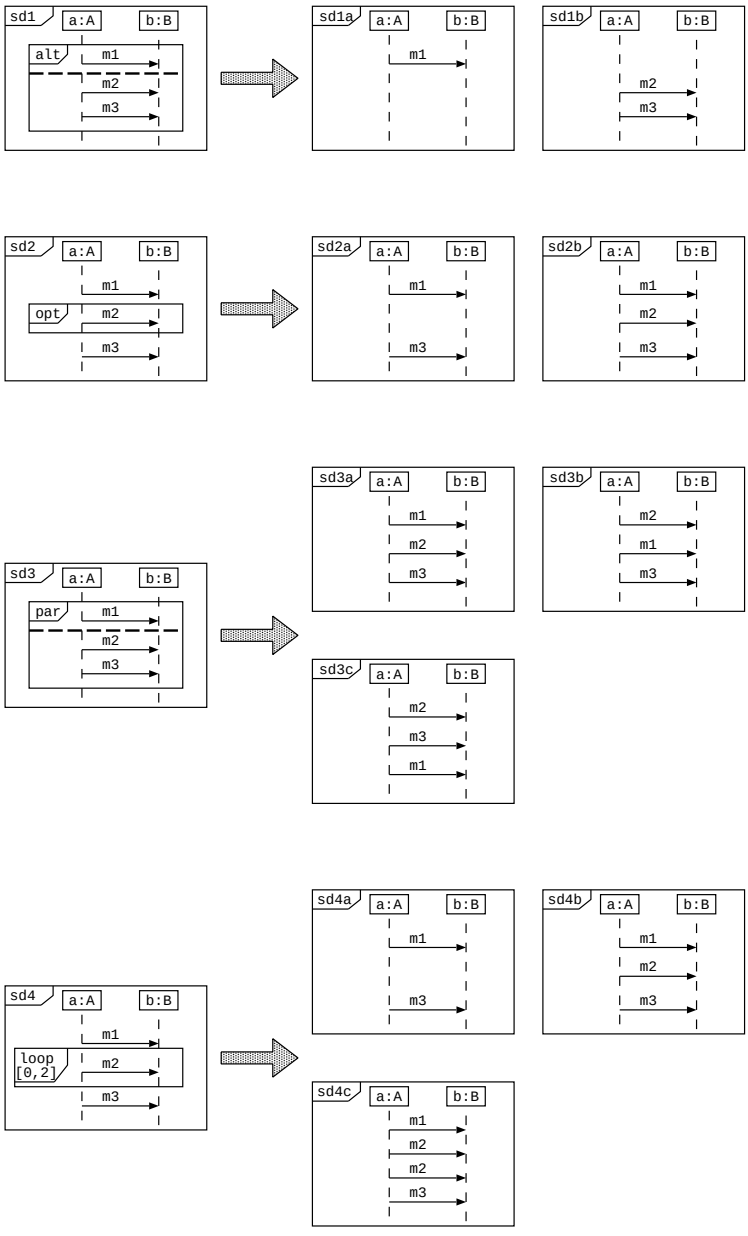


Abbildung 4.2: Transformation komplexer Sequenzdiagramme

jedoch weiterhin möglich. Da die meisten geschachtelten Sequenzdiagramme sich jedoch auf einfachere Sequenzdiagramme abbilden lassen, ist durch einen Zwischenschritt die Abbildung auf ein Kommunikationsdiagramm in vielen Fällen möglich.

Beispiele für die Transformation komplexer in einfache Sequenzdiagramme zeigt Abbildung 4.2. So läßt sich z.B. ein Sequenzdiagramm mit zwei Alternativen in zwei Sequenzdiagramme mit je einer der Alternativen umwandeln (im Beispiel `sd1`). Eine Abbildung unendlicher Iterationen ist auf diese Weise nicht möglich. Da zum Test immer eine Reduzierung auf eine endliche Anzahl von Iterationen nötig ist, werden unendliche Iterationen nur nach Reduzierung auf eine endliche Anzahl betrachtet.

4.2.5 Kommunikationsdiagramm

Das Kommunikationsdiagramm hat, bis auf einen Namenswechsel, in der UML 2.0 kaum Änderungen erfahren. Obwohl die dargestellte Information auch als Sequenzdiagramm modelliert werden kann, ist die Sicht eine andere. Im Sequenzdiagramm liegt der Schwerpunkt auf der Darstellung der zeitlichen Abfolge, beim Kommunikationsdiagramm auf der Darstellung der beteiligten Objekte an einem Szenario (Kollaboration).

4.2.6 Timingdiagramm

Das Timingdiagramm wurde in die UML 2.0 aufgenommen, um auch zeitliche Abläufe zu unterstützen, wie sie z.B. zur Modellierung zeitkritischer Systeme gebraucht werden. Die Art der Darstellung wird schon länger z.B. im Bereich der Elektrotechnik genutzt. Visualisiert wird das exakte zeitliche Verhalten von Klassen, Schnittstellen oder Komponenten. Es besteht eine Verwandtschaft zum Sequenzdiagramm, wenn dieses mit zeitlicher Information versehen wird.

4.2.7 Interaktionsübersichtsdiagramm

Das mit der UML 2.0 eingeführte Interaktionsübersichtsdiagramm zählt zu den Interaktionsdiagrammen und dient der Verbindung der anderen Interaktionsdiagramme auf einer sehr abstrakten Ebene. Es soll verdeutlicht werden, in welcher zeitlichen Reihenfolge und in welcher Abhängigkeit verschiedene Interaktionen ablaufen. Dabei wird entweder auf andere Interaktionsdiagramme verwiesen oder diese werden innerhalb des Interaktionsübersichtsdiagramms definiert. Die Notationsmittel sind dem Aktivitätsdiagramms entliehen, so gibt es wie dort Verzweigungen, Start- und Endpunkte. Die Aktivitäten im Aktivitätsdiagramm entsprechen hier den Interaktionen.

4.3 Die Object Constraint Language

Die Angabe von Bedingungen kann in der UML in natürlicher Sprache oder unter Benutzung der Object Constraint Language (OCL) erfolgen. Die Object Constraint Language ist eine textuelle mathematische Notation und als Teil der UML spezifiziert. Sie dient dazu, UML-Modelle in allen Ebenen (vom Meta-Meta-Modell bis zu den Modellinstanzen) zu präzisieren, da eine rein graphische Notation diesen Zweck allein nicht erfüllt.

Die Einsatzmöglichkeiten der OCL sind vielfältig. Hier soll nur ein Einblick gegeben werden. Eine umfassende Übersicht findet sich in der OCL-Spezifikation [84].

Am wichtigsten für die UML ist der Einsatz der OCL bei der Erweiterung des UML-Sprachumfangs durch Constraints. Auch Stereotypen können durch OCL-Bedingungen semantisch definiert werden. Wichtig für die vorliegende Arbeit ist vor allem die Möglichkeit, Vor- und Nachbedingungen sowie Invarianten mit Hilfe der OCL zu formulieren. Auch Zustandsinvarianten lassen sich in OCL spezifizieren. Weitere Einsatzgebiete sind z.B. die Definition von Einschränkungen an Assoziationen oder Regeln zum Ableiten von Attributen in UML-Modellen.

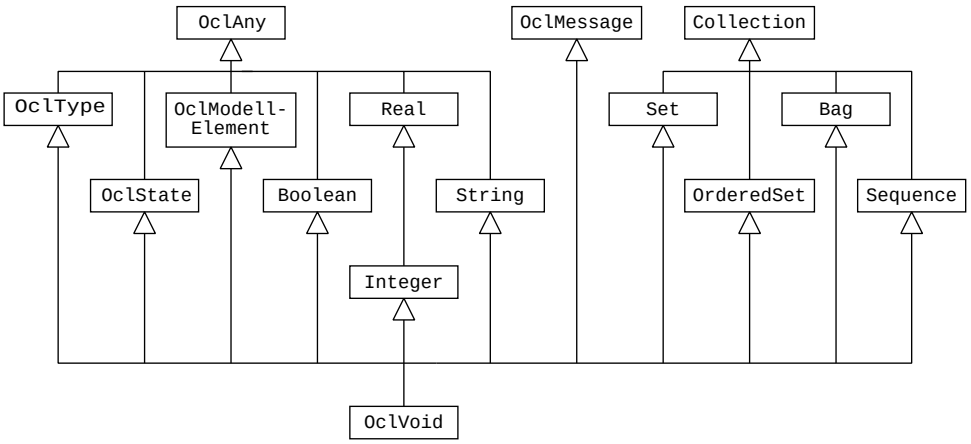


Abbildung 4.3: Typen der OCL-Standardbibliothek

Die OCL ist eine formale Sprache, die an andere Spezifikationssprachen wie z.B. `Z` [104] erinnert, aber weniger mächtig ist[8]. Die OCL definiert in der Regel Schlüsselworte statt Symbole, was ihre Akzeptanz unter Softwareentwicklern steigern soll. Dadurch erinnert die Sprache eher an eine Programmiersprache als an eine mathematische Notation.

OCL-Constraints werden immer in einem Kontext formuliert. Der Kontext wird entweder implizit durch das Hinzufügen einer Notiz an ein UML-Modellelement definiert (dann stellt das Modellelement den Kontext dar) oder durch die explizite Angabe des Kontexts durch das Schlüsselwort `context`. Aussagen in OCL ohne Kontektdeklaration haben keinen Bezug zu einem UML-Modell und somit keinen Bezug zum modellierten System. Die aktuelle Instanz des Modellelements, das den Kontext bildet, wird durch das Schlüsselwort `self` referenziert[9].

Aussagen in OCL sind immer seiteneffektfrei. Innerhalb einer Aussage in OCL werden deshalb nur seiteneffektfreie Methoden des aktuellen Objekts verwendet, also ausschließlich Observermethoden. Dies sicherzustellen, ist Aufgabe des Entwicklers eines Systems, die OCL selbst bietet dafür kaum Hilfsmittel[10].

Die OCL kennt verschiedene Arten von Typen. Abbildung 4.3 zeigt die Vererbungsstruktur aller OCL-Typen der Standardbibliothek. Fast alle Typen in OCL besitzen einen gemeinsamen Obertyp, den Typen `OclAny`. Es wird unterschieden zwischen Basistypen (Zahlentypen, Strings), mehreren Typen für Kollektionen und einigen speziellen Typen wie z.B. dem Typ `OCLState`, der OCL-Constraints mit den Zuständen eines Zustandsdiagramms in Verbindung setzt. Wichtig ist zudem, daß jede Klasse, die in einem der Diagramme vom Entwickler modelliert wird, vom Typ `OCLType` ist.

In der vorliegenden Arbeit werden OCL-Constraints in zwei Schwerpunkten eingesetzt, den Zustandsdiagrammen und dem *Design by Contract* [80].

Im ersten Schwerpunkt dienen OCL-Constraints zur Spezifikation von Zustandsinvarianten und von Bedingungen (Guards) an Transitionen.

Im zweiten Schwerpunkt wird die OCL zur Spezifikation von Vor- und Nachbedingungen von Methoden sowie von Klasseninvarianten benutzt. Dabei werden in OCL Zusicherungen im Sinne des Design by Contract spezifiziert.

[8] So fehlen OCL zum Beispiel die Ausdrucksmöglichkeiten des Schemakalküls oder die komplexen Operatoren über Mengen, die `Z` besitzt.

[9] Handelt es sich z.B. beim Kontext um eine Methodenspezifikation, so referenziert `self` die Instanz der Klasse, der die Methode zugeordnet ist.

[10] Es lassen sich jedoch Methoden mit Stereotypen, z.B. `isQuery`, versehen und auf diese Weise in Query- und Observer-Methoden unterscheiden.

In den zwei folgenden Abschnitten soll ein kurzer Einblick gegeben werden, wie sich Zustandsinvarianten sowie Zusicherungen im Sinne des Design by Contract in OCL spezifizieren lassen.

4.3.1 Zustandsinvarianten in OCL

Zustandsinvarianten sind immer einem Zustand in einem Zustandsdiagramm zugeordnet. Dabei spezifiziert die Menge aller Zustandsinvarianten im Zustandsdiagramm den gültigen Zustandsraum der Komponente, z.B. einer Klasse, der das Zustandsdiagramm zugeordnet ist. Im Falle einer Klasse ergibt die Menge aller Zustandsinvarianten die Klasseninvariante. Dabei herrscht keine Einigkeit, ob die Zustandsinvarianten den gültigen Zustandsraum exakt beschreiben und so gleich der Klasseninvariante sind.

Die Zustandsinvariante eines Zustands gilt für alle seine Unterzustände. Aus der Sicht eines Unterzustands betrachtet ist zunächst nur die für diesen Zustand spezifizierte Zustandsinvariante sichtbar. Gleichzeitig müssen jedoch alle spezifizierten Zustandsinvarianten aller Oberzustände des betrachteten Zustands berücksichtigt werden. Sei x ein Zustand und s_x die spezifizierte Zustandsinvariante des Zustandes x. Seien die Zustände y_i mit $i : 1..n$ alle Oberzustände von x und s_{y_i} die Zustandsinvariante des Zustandes yi, so ist die Zustandsinvariante des Zustandes x:

$\boxed{(\bigwedge_1^n s_{y_i}) \wedge S_x}$.

Folglich ist die Zustandsinvariante eines Zustands die Konjunktion der für den betrachteten Zustand spezifizierten Zustandsinvarianten und der Zustandsinvarianten aller seiner Oberzustände.

Zustandsinvarianten im Zustandsdiagramm werden mit Hilfe der OCL notiert, in der Regel in Form einer Notiz an den Zuständen eines Zustandsdiagramms. Der Kontext der spezifizierten Zustandsinvarianten in OCL ist der jeweilige Zustand.

4.3.2 Design by Contract in OCL

Das Design by Contract [80] ist eine Möglichkeit, Schnittstellen zwischen Softwarekomponenten[11] zu spezifizieren. Grundidee ist dabei die Betrachtung von Softwarekomponenten als Vertragspartner mit Rechten und Pflichten, die in Form von Zusicherungen angegeben werden. Ein Vertrag gliedert sich aus der Sicht der Serverkomponente in drei Teile: die Vorbedingung, die Nachbedingung und die Klasseninvariante.

Die Klasseninvariante beschreibt die Menge aller gültigen Zustände, die ein Objekt der Serverklasse in seinem Lebenszyklus annimmt. Die Klasseninvariante kann durch Zustandsinvarianten weiter verfeinert werden (siehe auch Abschnitt 4.3.1).

Jede Methode der Serverklasse muß bei einem korrekten Aufruf von außen das aktuelle Objekt von einem gültigen Zustand in einen gültigen Zustand überführen. Ein Aufruf ist genau dann korrekt, wenn sich das Objekt der Serverklasse im Augenblick des Aufrufs in einem gültigen Zustand befindet (Verantwortung des Serverobjekts) und die Vorbedingung der aufgerufenen Methode eingehalten wird (Verantwortung des Aufrufers, also des Clientobjekts). Ist ein Aufruf korrekt, so verpflichtet sich das Serverobjekt, die Nachbedingung der aufgerufenen Methode nach der Ausführung sicherzustellen und die Klasseninvariante nicht zu verletzen.

Ist der Vertragspartner zur Laufzeit kein Objekt vom Typ des Servers, sondern ein Objekt vom Typ einer Unterklasse des Servers, so spricht man vom *Subcontracting*. Das Clientobjekt erwartet ein Objekt der Serverklasse und damit, daß es selbst die in der Serverklasse spezifizierten Vorbedingungen einhalten muß und daß die in der Serverklasse spezifizierten Nachbedingungen vom Serverobjekt sichergestellt werden. Daß zur Laufzeit ein Objekt eines anderen Typs Vertragspartner auf der Serverseite sein kann, ist nur dann ohne Laufzeitfehler möglich, wenn dieses die gleichen oder geringere Anforderungen an den Aufruf von Methoden stellt und mindestens die Nachbedingung der Methoden der Oberklasse sicherstellt.

[11] Dabei sind Softwarekomponenten im weitesten Sinn gemeint, z.B. Klassen, Subsysteme oder auch Komponenten im Sinne der komponentenbasierten Softwaresysteme.

```
context SavingsAccount inv :
        self.balance >= 0

context SavingsAccount::withdraw(amount:int)
 pre  : self.balance >= amount and amount > 0
 post : self.balance = self.balance@pre - amount
```

Abbildung 4.4: Softwareverträge in OCL

Sei *service* eine Methode der Klasse Server. Sei *service′* dieselbe Methode in einer Unterklasse von Server, entweder geerbt oder überschrieben. Sei *pre* die Vorbedingung und *post* die Nachbedingung der Methode *service*, *pre′* die Vorbedingung und *post′* die Nachbedingung der Methode *service′*, so stehen diese in folgendem Verhältnis zueinander:

$\boxed{\{pre\} \Rightarrow \{pre'\} \quad service'() \quad \{post'\} \Rightarrow \{post\}}$.

Aus der Vorbedingung der Methode des Serverobjekts muß die Vorbedingung derselben Methode des Unterklassenserverobjekts folgen. Aus der Nachbedingung der Methode des Unterklassenserverobjekts muß die Nachbedingung derselben Methode des Serverobjekts folgen[12].

OCL bietet für die Definition von Softwareverträgen im Sinne des Design by Contract spezielle Schlüsselworte an, `pre` für Vorbedingungen, `post` für Nachbedingungen und `inv` für Klasseninvarianten. Dabei ist der Kontext einer Klasseninvarianten eine Klasse, der Kontext einer Vor- oder Nachbedingung eine Methode. Vor- und Nachbedingungen von Methoden können einer Kontextdeklaration gemeinsam zugeordnet werden. Als ein wichtiges Schlüsselwort verweist `@pre` in der Nachbedingung auf den Zustand des aktuellen Objekts oder der gegebenen Parameter bei Aufruf der Methode[13].

Abbildung 4.4 zeigt einen Ausschnitt der Zusicherungen für eine Klasse, hier die Klasse `SavingsAccount`. Die Invariante dieser Klasse sagt aus, daß ein Objekt dieser Klasse immer einen Kontostand aufweisen muß, der größer oder gleich Null ist (Klasseninvariante `self.balance >= 0`). Die Methode `withdraw` muß den gültigen Zustand nach ihrer Ausführung wieder sicherstellen. Dazu wurde als Vorbedingung für den Aufruf spezifiziert, daß der gegebene Betrag kleiner oder gleich dem Kontostand ist und der gegebene Betrag einen positiven Wert hat (`self.balance >= amount and amount > 0`). Als Nachbedingung wurde spezifizert, daß der Kontostand um den gegebenen Betrag reduziert ist (Nachbedingung `self.balance = self.balance@pre - amount`). Dabei verweist der Term `self.balance@pre` auf den Zustand des Kontostands unmittelbar vor der Ausführung der Methode `withdraw`.

[12] Diese Beziehung wird in Eiffel [80] durch die Disjunktion der Vorbedingungen und die Konjunktion der Nachbedingungen erzwungen. Diese Art der Vererbung wird auch konforme Vererbung genannt.

[13] Analog zu dem Schlüsselwort `old` in Eiffel.

Kapitel 5

Testbarkeit von UML-Modellen

Die Frage nach der Testbarkeit von UML-Modellen läßt sich nicht pauschal beantworten. Verschiedene Diagrammtypen beinhalten unterschiedlich gut für den Test nutzbare Informationen. Die Einteilung in statische und dynamische Modelle kann grob als Einteilung in Modelle mit höherem Modellierungsanteil und Modelle mit höherem Spezifikationsanteil gesehen werden. In der vorliegenden Arbeit werden die Begriffe Modellierung und Spezifikation wie folgt voneinander abgegrenzt:

- Eine Modellierung beschreibt die statische Struktur eines Systems.

- Eine Spezifikation beschreibt das Verhalten eines Systems.

Ein höherer Spezifikationsanteil bedeutet auch eine bessere Eignung für den Test. Außerdem läßt sich sagen, daß je formaler und präziser ein Modell ist, desto mehr Informationen für einen Test lassen sich gewinnen. Zudem stellt sich die Frage, ob ein Testverfahren sich auf einen bestimmten Diagrammtyp direkt anwenden läßt oder ob das Modell erst um weitere Informationen angereichert werden muß.

Im Folgenden werden alle Diagrammtypen der UML im Hinblick auf ihre Testbarkeit untersucht. Analog zu Kapitel 4 werden zunächst die statischen Diagramme betrachtet. Anschließend erfolgt eine Analyse der dynamischen Diagramme. Schließlich wird die OCL betrachtet.

Es werden jeweils Testansätze für die einzelnen Diagramme beschrieben. Dabei wurde im Rahmen der vorliegenden Abeit sowohl untersucht, welche Informationen das jeweilige Diagramm als Basis für den Test anbietet und ob sich nicht speziell auf die UML fokussierte Testverfahren auf den Diagrammtyp übertragen und anwenden lassen, als auch, welche Testansätze auf Basis des jeweiligen Diagrammtyps bereits existieren.

Die Untersuchung betrachtet die einzelnen Diagrammtypen nach folgenden Kategorien:

Integrationstest. Hier wird vorrangig die Frage beantwortet, in welcher Weise Informationen über Objekte unterschiedlicher Klassen und ihr Zusammenspiel im System vorhanden sind, welche Nachrichten verschickt werden und welche Reaktionen auf diese erfolgen.

Testreihenfolge. Die Testreihenfolge definiert, welche Klassen vor welchen anderen getestet werden. Dafür gibt es verschiedene Strategien, z.B. Top-Down, bei dem der Test mit den Klassen mit den meisten Abhängigkeiten beginnt, oder Bottom-Up, bei dem der Test mit den am wenigsten abhängigen Klassen beginnt [10]. Es wird die Frage geklärt, ob sich diese Abhängigkeiten aus UML-Diagrammtypen gewinnen lassen.

Testfälle. Es wird untersucht, ob und in welcher Form die Diagrammtypen Informationen zur Testfallermittlung enthalten.

Testdaten. Analog zur Testfallermittlung wird hier die Testdatengewinnung betrachtet.

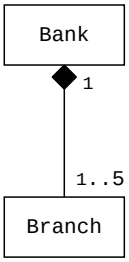


Abbildung 5.1: Aggregation mit Multiplizitäten als obere und untere Grenze

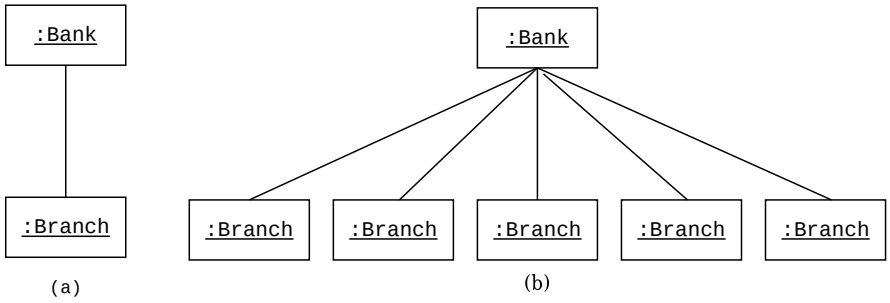


Abbildung 5.2: Objektkonstellation für (a) die untere Grenze und (b) die obere Grenze

Testorakel. Es wird betrachtet, ob Diagrammtypen Informationen enthalten, die zur Ermittlung von Sollwerten nutzbar sind.

Testfälle, Testdaten und Testorakel ermöglichen erst die Testdurchführung und bilden den Schwerpunkt der vorliegenden Arbeit. Testreihenfolge und Unterstützung für den Integrationstest dienen der Testvorbereitung. Der Integrationstest als Test der Schnittstellen wurde hier hervorgehoben, da er eine andere Qualität besitzt als der Klassentest als Test der kleinsten Einheit und der Systemtest als Test des Ganzen (siehe auch Diskussion zur Integration beim Testen in [115]).

5.1 Statische Diagrammtypen

Statische Diagramme bieten meist wenig Informationen, die als Grundlage für einen Test nutzbar sind. Eine Ausnahme ist das Klassendiagramm, das als zentraler Diagrammtyp eine Reihe von Testaktivitäten unterstützt.

5.1.1 Klassendiagramm

Information aus dem Klassendiagramm lassen sich sowohl zur Testvorbereitung als auch zur Testdurchführung nutzen.

Testreihenfolge. Da das Klassendiagramm Informationen über Klassen und ihre Beziehungen enthält (auch wenn in der Regel nicht vollständig) und diese bereits zu einem sehr frühen Zeitpunkt der Softwareentwicklung vorliegen, ist auf Basis des Klassendiagramms eine Bestimmung der Testreihenfolge möglich. Es gibt verschiedene Algorithmen, deren Anwendung möglich ist. Eine Übersicht der Verfahren, die sich besonders für objektorientierte Systeme eignen, bietet [10]. Auch das in [63] vorgestellte Verfahren benutzt ein Klassendiagramm als Basis für die Testreihenfolgebestimmung. Dieses wird dort jedoch aus dem Quellcode gewonnen und ist deshalb vollständig.

Auf Basis des Klassendiagramms ist insbesondere eine Analyse möglich, für welche Klassen im Test Stubs zur Verfügung gestellt werden müssen und welche Klassen durch Treiber angesprochen werden müssen (je nach gewählter Integrationsstrategie).

Mit der großen Verbreitung von Softwareentwicklungsumgebungen, die aus Klassendiagrammen Codeschablonen generieren, wird auch die Vollständigkeit des Klassendiagramms zunehmen, was eines der größten Hindernisse für die Ermittlung der Testreihenfolge beseitigt. Die erzeugten Codeschablonen lassen sich, z.B. ergänzt durch mit Hilfe von Zustandsautomaten spezifizierter Objektlebenszyklen, als Stubs nutzen.

Testfälle. Testfälle werden aus dem Klassendiagramm z.B. durch besondere Auszeichnungen der Methoden gewonnen. In die UML ist inzwischen das Attribut `isQuery` für Observer-Methoden eingegangen, so daß sich rudimentär Testfälle in Form von Methodensequenzen durch Anwendung des Alpha-Omega-Zyklus aus [10] gewinnen lassen. Der Alpha-Omega-Zyklus erzeugt Testfälle, indem sukzessiv längere Methodensequenzen aufgebaut werden, je nach Typ einer Methode. Die kürzesten Sequenzen enthalten nur den Konstruktor, werden im darauffolgenden Schritt um Getter-Methoden ergänzt, im nächsten um Query-Methoden, dann um die Setter-Methoden usw. bis hin zu den Destruktoren. Eine vollständige Anwendung des Alpha-Omega-Zyklusses ist nur bei Einführung von zusätzlichen Attributen zur Auszeichnung von Methoden möglich (siehe auch [115]).

Testdaten. Mit Hilfe von Assoziationen ist entscheidbar, welche Objekte mit welchen anderen Objekten Beziehungen unterhalten. Für einen Testfall werden auf Basis des Klassendiagramms entsprechende Testdaten erzeugt, indem Objekte über ihre Assoziationen miteinander verbunden werden. So ist z.B. eine Anwendung des in [90] vorgestellten Verfahrens auf die Generalisierungs/Spezialisierungs-Beziehungen im Klassendiagramm vorstellbar. Hierbei werden zum Testen Objekte von allen Typen auf der Serverseite mit Objekten von allen Typen auf der Clientseite zum Testen erzeugt und gegeneinander getestet. Der Test wird zunächst mit Objekten der Oberklassen begonnen, anschließend erfolgt ein Test je eines Objekts einer Oberklasse gegen ein Objekt der Subklasse und schließlich werden die Subklassenobjekte gegeneinander getestet.

Multiplizitäten leisten zusätzlich Unterstützung zur Testdatenerzeugung. Multiplizitäten lassen sich durch Grenzwertanalyse [81] zur Erzeugung von Testdaten in Form von Objektkonstellationen verwenden. So kann z.B. eine Aggregationsbeziehung zwischen `Bank` und `Branch`, wie in Abbildung 5.1 gezeigt, genutzt werden, um die Objektkonstellationen in Abbildung 5.2 (dargestellt als Objektdiagramme) zu erzeugen. Abbildung 5.2(a) repräsentiert dabei die untere, Abbildung 5.2(b) die obere Grenze, spezifiziert durch die Multiplizitäten aus Abbildung 5.1. Sind keine Grenzen angegeben (*-Notation oder keine Definition), so ist dieses Verfahren nicht anwendbar.

Testorakel. Eine Nutzung von Multiplizitäten ist darüber hinaus als Testorakel vorstellbar, da sie eine obere und untere Grenze spezifizieren, deren Einhaltung sich zur Laufzeit überprüfen läßt.

Auch aus den Assoziationen läßt sich ein Testorakel ableiten. So wird eine Einhaltung der modellierten Beziehungen und eine Versendung von Nachrichten nur anhand von modellierten Links[1] zur Laufzeit überprüft.

5.1.2 Objektdiagramm

Es kann die Annahme gemacht werden, daß das Objektdiagramm typische Objektkonstellationen modelliert. Diese bieten zwar keine Basis für die Generierung von Testfällen oder die Erzeugung von Sollwerten, sind jedoch hilfreich für die Erzeugung von Testdaten aus den Testfällen.

[1] Ein Link ist die Instanz einer Assoziation.

Testdaten. Ein Objektdiagramm gibt bereits Objekte mit Attributbelegungen und Beziehungen zu anderen Objekten an, so daß auf dieser Basis reale Objekte im implementierten System erzeugt werden, die als Initialisierung typischer Szenarien dienen.

5.1.3 Andere statische Diagrammtypen

Zu den statischen Diagrammtypen zählen außerdem das Komponentendiagramm, das Kompositionsstrukturdiagramm, das Deploymentdiagramm und das Paketdiagramm. Bei allen diesen Diagrammtypen handelt es sich um Diagramme zur Modellierung. Sie enthalten wenig bis keine Spezifikationsanteile. Man könnte darüber diskutieren, ob z.B. die Information aus dem Deploymentdiagramm ein Modell des verteilten Systems ist oder eine Anforderung, die überprüfbar ist, z.B. bei sich selbstorganisierenden verteilten Systemen.

Durch den geringen Gehalt an Spezifikation ist ein Test auf Basis dieser Diagrammtypen nur bedingt möglich. Vorstellbar wäre z.B. die Nutzung der Informationen aus dem Deploymentdiagramm für die Testkonfiguration, ähnlich wie beim Objektdiagramm. Eine Generierung von Testfällen oder gar Sollwerten auf Basis der genannten Diagrammtypen wird jedoch ausgeschlossen.

Dementsprechend gibt es auch nur wenige Testansätze, die einen der vier Diagrammtypen zugrunde legen. Eine Ausnahme ist [99], wo in sehr kurzer Form auf die Verwendung von Komponenten- und Verteilungsdiagramm im Integrationstest eingegangen wird.

Integrationtest. Nach [99] bietet das Deploymentdiagramm nur eine Hilfe für den Integrationstest verteilter Systeme. Außerdem wird die Nutzung des Komponentendiagramms für den Integrationstest komponentenbasierter Systeme vorgeschlagen. Gleichzeitig wird betont, daß das Komponentendiagramm ohne geeignete Spezifikation der Schnittstellen, z.B. mit Hilfe einer IDL-Sprache, unbrauchbar ist.

Das Paketdiagramm ist von der Betrachtung in [99] explizit ausgeschlossen worden, das Kompositionsstrukturdiagramm, das erst mit der UML 2.0 eingeführt wurde, ist zum Erscheinungszeitpunkt des Buches noch nicht Teil der UML gewesen. Das Kompositionsstrukturdiagramm ist jedoch in der Lage, die nötige Information zu liefern, die nach [99] dem Komponentendiagramm für den Test noch fehlt.

5.2 Dynamische Diagrammtypen

Dynamische Diagramme modellieren Verhalten auf unterschiedlichen Abstraktionsniveaus. Sie sind deshalb prädestiniert zur Ableitung von Testfällen. Aus einigen dynamischen Diagrammen lassen sich darüber hinaus Informationen gewinnen, die die Sollwertbestimmung unterstützen.

5.2.1 Use-Case-Diagramm

Obwohl es Testansätze auf der Basis von Use-Case-Diagrammen gibt, ist die Information in einem Use-Case-Modell allein meist zum Test nicht ausreichend. Wie die nachfolgend beschriebenen Ansätze verdeutlichen, ist eine Anreicherung oder eine Verfeinerung des Use-Case-Modells nötig, um genug Informationen zum Testen zu gewinnen.

Das läßt sich leicht erklären. Durch das hohe Abstraktionsniveau ist ein Bezug zum implementierten System nur schwer zu erkennen. Eine Reihe von Fragen verdeutlicht diese Situation:

- Welche Klassen oder Kollaborationen von Klassen implementieren welchen Use-Case?
- Welche Nachrichtenfolgen realisieren welchen Use-Case?
- Beschreibt ein Use-Case nur Normal- oder auch Ausnahmeverhalten?

Alle diese Fragen sind auf der Basis eines Use-Case-Modells in der Regel nicht zu beantworten.

Obwohl das Use-Case-Diagramm zu den dynamischen Diagrammtypen zählt, enthält es doch nur wenig Spezifikation. Es ist in Verbindung mit anderen Diagrammtypen zum Testen einsetzbar, wenn die Zusammenhänge einer klaren Semantik unterliegen. Use-Case-getriebene Softwareentwicklungsprozesse wie z.B. der *Unified Process* [62] definieren meist eine mehr oder weniger klare Semantik, was den Test erleichtert. Werden Use-Case-Modelle durch andere dynamische Modelle verfeinert, wie z.B. Aktivitäts- oder Sequenzdiagramme, basiert der Test meist auf diesen Modellen und muß nicht zwangsläufig die mit Hilfe des Use-Case-Diagramms erstellten Modelle berücksichtigen.

Testfälle. [10] beschreibt drei Use-Case-basierte Ansätze zum Test objektorientierter Systeme: Den *Extended Use-Case Test*, den *Covered in CRUD Test* und den *Allocated by Profile Test*. Bei allen drei Ansätzen handelt es sich um die Erzeugung von Testfällen im Hinblick auf ein bestimmtes Testendekriterium. Alle drei lassen sich kombinieren und verlangen eine Anreicherung des Use-Case-Diagramms um weitere Informationen.

Beim *Extended Use-Case Test* werden den Use-Cases Use-Case-Beschreibungen zugeordnet. Aus diesen werden Entscheidungstabellen erzeugt und mit einer minimalen Anzahl an Testfällen wird eine vollständige Überdeckung aller Varianten angestrebt.

Der *CRUD* Test dient zur Messung der Überdeckung aller Domainklassen[2]. Zum Test müssen alle Use-Cases überdeckt werden und mindestens je ein Objekt jeder Domainklasse erzeugt, gelesen, geändert und gelöscht werden. Dazu muß die entsprechende Information (welcher Use-Case erzeugt oder ändert welche Objekte) bekannt sein, z.B. durch Annotationen an den Use-Cases.

Der *Allocated by Profile Test* schließlich ordnet jedem Use-Case eine relative Häufigkeit zu, mit der er im System ausgeführt wird. Testfälle werden unter Berücksichtigung der relativen Häufigkeit und einer vorgegebenen Anzahl der Gesamttestfälle erstellt.

Auf der Basis der Häufigkeit der Nutzung von Use-Cases basiert auch das in [94] vorgestellte Verfahren. Es benutzt ein sogenanntes *Usage-Modell* in Form einer Markov-Kette. Use-Cases werden erst zu Zustandsdiagrammen und anschließend zu *Usage-Graphen* verfeinert. Aus den Usage-Graphen werden die Usage-Modelle abgeleitet und als Basis für die Testfallgenerierung benutzt. Neben der Generierung von Testfällen auf Basis der minimalen Kantenüberdeckung wird die Generierung von zufälligen Testfällen, basierend auf den Wahrscheinlichkeiten an den Kanten des Graphen, vorgeschlagen.

In [35] werden die Use-Case-Beschreibungen benutzt, um in einer Reihe von automatischen und manuellen Schritten Testfälle für ein Capture-Replay-Tool[3] zu entwerfen und damit einen automatischen Systemtest mit Hilfe der Benutzeroberfläche zu ermöglichen.

In [116] werden Use-Cases durch sogenannte Use-Case-Schrittgraphen verfeinert, aus denen sich anschließend Testfälle ableiten lassen.

Testorakel. [116] beschreibt darüber hinaus einen Ansatz, bei dem Use-Cases um Vor- und Nachbedingungen angereichert werden, um eine Auswertung des Tests zu ermöglichen.

5.2.2 Aktivitätsdiagramm

Das Aktivitätsdiagramm wird u.a. in [15] zur Testfallgenerierung eingesetzt, dort in Kombination mit anderen Diagrammtypen (siehe Abschnitt 5.4). Andere Arbeiten zur Testfallgenerierung auf Basis des Aktivitätsdiagramms sind:

[2]Eine Domainklasse ist eine Klasse, die applikationsabhängige Objekte repräsentiert. Daneben gibt es auch applikationsunabhängige Klassen, ein typisches Beispiel hierfür sind Collection-Klassen.

[3]Capture-Replay-Tools sind Werkzeuge zum automatischen Test von Benutzeroberflächen. Sie zeichnen Aktionen des Benutzers in Form eines Scripts auf und spielen diese für Regressionstestzwecke wieder ab. Unter Regressionstest versteht man die wiederholte Ausführung der gleichen Implementierung mit den selben Testdaten, nachdem die Implementierung verändert wurde, z.B. durch Fehlerbehebung oder durch Erstellung einer neuen Version.

Testfälle. In [91] werden Aktivitätsdiagramme durch die Einführung neuer Sprachmittel um Varianten angereichert. Testszenarien werden durch die Überdeckung aller Varianten anhand von Überdeckungskriterien gebildet.

[72] generiert aus Aktivitätsdiagrammen zunächst Testszenarien, indem das Aktivitätsdiagramm traversiert wird. Jede Schleife wird dabei genau einmal durchlaufen. Testfälle werden aus den Testszenarien unter Berücksichtigung von Bedingungen an den Kanten im Aktivitätsdiagramm mit Hilfe des Category Partionings [89], einer Variante der Äquivalenzklassenbildung, erzeugt.

5.2.3 Zustandsdiagramm

Zustandsdiagramme sind neben der OCL der formalste Teil der UML mit einer präzisen Semantik. Deshalb sind sie als Basis für einen Test gut geeignet. Statecharts bzw. Zustandsdiagramme unterstützen vorrangig zwei Testaktivitäten, die Testfallerzeugung und das Testorakel, was sich auch in der Anzahl der Arbeiten zu diesen Testaktivitäten zeigt. In der vorliegenden Arbeit kann aufgrund der umfangreichen Literatur zum Thema Zustandsdiagramme bzw. Statecharts und Test nur eine Auswahl verwandter Arbeiten vorgestellt werden.

Testfälle. Eine der grundlegenden Arbeiten auf dem Gebiet Testfallgenerierung auf der Basis von Zustandsautomaten findet sich in [17]. Viele andere Arbeiten bauen auf dem dort vorgestellten Algorithmus zur Traversierung von Zustandsgraphen auf, der die vollständige Überdeckung aller Zustände und Transitionen bei einmaliger Wiederholung jedes Zyklus erreicht. So benutzt z.B. [63] den in [17] vorgestellten Algorithmus für die Generierung von Testfällen aus objektorientierten Statecharts und [10] eine Variante des Algorithmus für die *N+-Strategie* zur Testfallgenerierung aus FREE-Graphen. Dabei sind FREE-Graphen objektorientierte Zustandsgraphen, die das Verhalten der Oberklasse berücksichtigen und im Gegensatz zu den UML-Zustandsdiagrammen mehrere Initialzustände erlauben.

In [85, 87] findet sich eine Reihe von Überdeckungskriterien für Zustandsdiagramme, so z.B. die Überdeckung von Transitionen oder Transitionspaaren. Auf Basis der Überdeckungskriterien erzeugt das Werkzeug *UMLTest* Testfälle, die dem jeweiligen Kriterium entsprechen.

Ebenfalls orientiert an Überdeckungskriterien, hier Zustands- und Transitionsüberdeckung, erfolgt die Auswahl der Testfälle in [56], wobei die Zustandsdiagramme zunächst in ein Zwischenformat transformiert werden, den sogenannten *Testing Flow Graph*.

In [59, 49] schließlich werden Zustandsautomaten nicht nur zur Überdeckung von Struktureigenschaften benutzt, sondern es werden zusätzlich datenflußorientierte Überdeckungskriterien definiert.

[36] beschreibt ein Verfahren zur Testfallgenerierung aus Zustandsautomaten, die eine Verfeinerung von Use-Cases bilden. Testfälle werden durch zufällige Auswahl von Transitionen erzeugt, die in den vorherigen Testfällen nicht abgedeckt wurden. Als Ergebnis wird eine vollständige Überdeckung der Transitionen erreicht.

Ein anderer Ansatz findet sich in [98]. Dort wird der in [108] vorgestellte Algorithmus zur Erzeugung einer potentiell unendlichen Menge von Testfällen aus Labeled-Transitionssystemen auf UML-Statecharts übertragen. Die Erzeugung der Testfälle basiert auf einer nichtdeterministischen Auswahl zwischen der Eingabe des nächsten Stimulus an das System, der Beobachtung der Ausgabe und dem Ende des Testfalls. Der Algorithmus ist in der Lage, alle potientiell möglichen Testfälle zu erzeugen, führt jedoch durch die beim Test nötige Einschränkung nicht zwangsläufig in der erzeugten Testsuite zu einer vollständigen Überdeckung des Zustandsdiagramms.

[9] erweitert objektorientierte Zustandsautomaten, die Objektlebenszyklen beschreiben, um die Kontrollflußgraphen der referenzierten Methoden und leitet aus den so erzeugten Zustandsautomaten Testfälle durch Traversierung ab.

Ein weiteres Verfahren zur Generierung von Testfällen aus Zustandsdiagrammen stellt das Model-Checking dar. Eine solche Technik nutzen z.B. [27, 50]. Dabei wird das Testziel in [27] negiert und das vom Model-Checker erzeugte Gegenbeispiel als Testfall genutzt. Eine spezielle Variante dieses Verfahrens findet sich in [50]. Dort werden Überdeckungskriterien als Testziel formuliert und negiert.

Testorakel. [98] bietet neben der Erzeugung von Testfällen auch die Auswertung der Ergebnisse des Tests anhand der Systemreaktion in Form einer Ausgabe an. Interne Zustände des Systems werden nicht berücksichtigt.

In [101] werden Zustandsautomaten für jedes Objekt erzeugt und ausführbar gemacht. Die Auswertung des Tests erfolgt auf der Basis der ausführbaren Zustandsautomaten, die an die zu testende Implementierung gebunden werden. Die Testfälle werden in der Testumgebung ausgeführt und sowohl an das zu testende System als auch an die Zustandsautomaten gesendet. Das Testorakel wertet nur den nach außen sichtbaren Zustand der zu testenden Objekte aus.

5.2.4 Sequenzdiagramm

Testansätze auf der Basis von Sequenzdiagrammen lassen sich zwei verschiedenen Bereichen zuordnen. Auf der einen Seite gibt es Testansätze, die auf der Grundlage der klassischen UML-Sequenzdiagramme Testfälle oder Testdaten erstellen. Meist berücksichtigen diese noch nicht die Neuerungen, die die Sequenzdiagramme mit der Version 2.0 der UML erfahren haben. Die zweite Gruppe umfaßt Testansätze, die auf den Message Sequence Charts basieren. Die enge Verwandtschaft der Message Sequence Charts mit den Sequenzdiagrammen der UML 2.0 macht diese Techniken auf die UML übertragbar.

Das Sequenzdiagramm eignet sich primär für die Testfallgenerierung und als Basis für das Testorakel, aber auch für den Integrationstest.

Integrationstest. Die originäre Aufgabe aller Interaktionsdiagramme ist die Modellierung von Interaktionen zwischen Objekten. Informationen über verschickte Nachrichten und erfolgte Reaktionen lassen sich aus ihnen gewinnen.

Testfälle. Jedes Sequenzdiagramm ist als ein Testfall zu betrachten, der Nachrichten spezifiziert, die während des Tests zwischen Objekten ausgetauscht werden. Wird einer der Operatoren verwendet (z.B. `alt`), so definiert ein Sequenzdiagramm sogar eine Menge von Testfällen. Leider ist eine Stimulation des Systems nicht immer so gezielt möglich, daß das System genau mit der modellierten Sequenz reagiert.

Ein weiteres Problem ergibt sich aus der Verteilung von Information über eine große Anzahl von Teilmodellen in Form von Sequenzdiagrammen. Auf einen Stimulus ist eine Reaktion des Systems in verschiedener Weise möglich. Die Reaktion des Systems ist u.U. in unterschiedlichen Sequenzdiagrammen modelliert, so daß immer die Gesamtheit aller Sequenzdiagramme (und auch der anderen Interaktionsdiagramme) betrachtet werden muß.

Erschwerend kommt hinzu, daß das Sequenzdiagramm keinerlei Aussage über den Vorzustand des Systems trifft. Es gibt weder einen zeitlichen Bezug (wann tritt das modellierte Verhalten auf?) noch sind die Voraussetzungen für ein Szenario gegeben (in welchem Zustand müssen sich die beteiligten Objekte befinden?).

Durch die Einführung der Operatoren für Sequenzdiagramme in die UML 2.0 ist die Betrachtung von Sequenzen als Methodensequenzspezifikationen im Sinne von [60] möglich. In [60] wird eine Technik vorgestellt, die aus (textuellen) Beschreibungen von Methodensequenzen mit Operatoren wie *Iteration* oder *Alternative* Testsequenzen erzeugt.

[33] beschreibt eine Technik zur Generierung von Testfällen aus Sequenzdiagrammen. Sequenzdiagramme sind dort miteinander kombinierbar und bilden so beliebig lange Sequenzen. Zur Testdatenerzeugung werden die Sequenzdiagramme annotiert. Als Testorakel dienen Vor- und Nachbedingungen für jedes Szenario.

Aus dem Bereich der Message Sequence Charts gibt es eine Reihe von Arbeiten, die Testfälle gewinnen. Zu nennen sind hier z.B. [2, 38, 51].

Testorakel. Sequenzdiagramme enthalten in der Regel zu wenig Information, um als Basis für ein Testorakel zu dienen. Es gibt zwar den Operator *neg*, der Nachrichtenfolgen kennzeichnet, die nicht erlaubt sind, so daß das Auftreten einer entsprechenden Sequenz als fehlgeschlagener Test zu werten ist. Bei positiven Sequenzen[4] ist das Auftreten einer entsprechenden Sequenz als bestandener Test zu interpretieren. Reagiert das System aber auf eine Nachricht mit einer Sequenz, die nicht modelliert wurde, ist eine Aussage nicht möglich (*nicht entscheidbar*), da es sich sowohl um eine vergessene negative als auch um eine vergessene positive Sequenz handeln kann. Einige Testansätze, wie der in [33] beschriebene, werten eine Reaktion mit einer nicht modellierten Sequenz als fehlgeschlagenen Test. Eine grundsätzliche Bewertung des Auftretens einer nicht modellierten Sequenz, ob als bestandener oder fehlgeschlagener Test, scheint jedoch ein nicht praktikables Vorgehen zu sein, wenn man berücksichtigt, daß in der Regel ein Großteil aller Sequenzen nicht modelliert ist. Die Anwendung einer Modellierungstechnik wie STAIRS [45] verringert zwar die Anzahl unendscheidbarer Sequenzen, eliminiert sie jedoch nicht völlig. Durch den Einsatz einer solchen Technik lassen sich Sequenzdiagramme jedoch sehr viel besser als Testorakel nutzen.

5.2.5 Kommunikationsdiagramm

Durch die Umwandlung von Kommunikationsdiagrammen in Sequenzdiagramme sind alle Testansätze für Sequenzdiagramme auf das Kommunikationsdiagramm übertragbar.

Testorakel. In [86] wird eine Technik zur Nutzung des Kommunikationsdiagramms als Testorakel vorgeschlagen. Dabei ist jeder Operation auf Systemlevel ein Kommunikationsdiagramm zugeordnet, das diese Operation vollständig beschreibt. Diese Einschränkung vermeidet unspezifiziertes Verhalten, wie es in realen Modellen häufig vorkommt. Anhand der Kommunikationsdiagramme wird die Reaktion des Systems auf die Operation ausgewertet.

5.2.6 Timingdiagramm

Für das Timingdiagramm, das in der UML 2.0 neu hinzugekommen ist, gibt es bisher keine Testverfahren, die explizit auf dem Timingdiagramm aufsetzen. Allerdings lassen sich aus dem Timingdiagramm Zeitinformationen gewinnen, die z.B. Testverfahren wie die in [114] vorgestellte Technik unterstützen können.

5.2.7 Interaktionsübersichtsdiagramm

Da das Interaktionsübersichtsdiagramm eine Mischung aus anderen Interaktionsdiagrammen und dem Aktivitätsdiagramm ist, sind alle Aussagen über die Testbarkeit jener Diagramme auf das Interaktionsübersichtsdiagramm übertragbar, sofern berücksichtigt wird, daß es sich bei den Aktivitäten um modellierte Abläufe in Form von Interaktionsdiagrammen handelt. Eine Kombination der Ansätze aus Aktivitätsdiagramm und Interaktionsdiagrammen wäre wünschenswert.

Gleichzeitig erinnert das Interaktionsübersichtsdiagramm an *High Level Message Sequence Charts*[5], so daß die Übertragbarkeit von Ansätzen für Message Sequence Charts, wie z.B. in [2] beschrieben, auf das Interaktionsübersichtsdiagramm geprüft werden muß.

[4] Alle Sequenzen, die nicht mit dem Operator *neg* gekennzeichnet sind, werden hier als positive Sequenzen bezeichnet.

[5] High Level Message Sequence Charts [78] dienen der Übersicht über die Beziehungen und den Kontrollfluß zwischen Message Sequence Charts, die eng verwandt mit den UML-Sequenzdiagrammen sind.

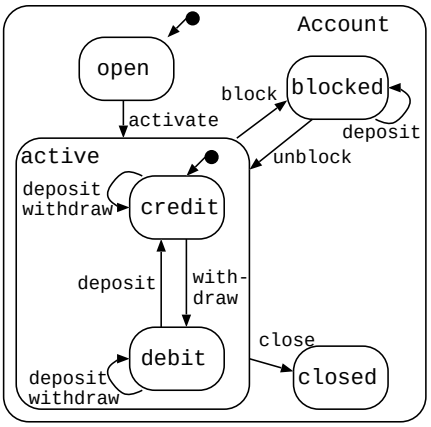


Abbildung 5.3: Zustandsdiagramm der Klasse `Account`

```
context Account::withdraw (amount:int)
   pre : true
   post: self.balance = self.balance@pre - amount
```

Abbildung 5.4: Vor- und Nachbedingung der Methode `withdraw` in OCL

5.3 Die Object Constraint Language

Die Object Constraint Language wird häufig genutzt, um Vor- und Nachbedingung zu formulieren und automatisch zu prüfen. Deshalb eignet sie sich insbesondere zur Ableitung eines Testorakels.

Testorakel. Aus OCL-Constraints lassen sich Testorakel ableiten, da sie Bedingungen spezifizieren, die nicht verletzt werden dürfen. Um diese Bedingungen prüfen zu können, ist jedoch meist eine Transformation von OCL-Constraints in ein Zwischenformat oder eine Programmiersprache notwendig. Meist werden die transformierten Constraints direkt oder indirekt in den Quellcode integriert.

In [41, 42] werden OCL-Constraints zunächst in die Sprache JML (*Java Modeling Language* [55, 66]) transformiert, die anschließend durch die JML-Laufzeitumgebung ausgewertet werden. DresdenOCL [25] dagegen wandelt OCL-Constraints direkt in Java-Code um, der in den Quellcode integriert wird. Mit Hilfe aspektorientierter Programmiertechniken werden schließlich OCL-Constraints in [13] ausgewertet.

Testdaten. Testdaten auf der Basis von Invarianten werden von [12, 3] durch die Anwendung eines einfachen Suchalgorithmus über dem Datenraum erzeugt. Bei den erzeugten Testdaten handelt es sich um Datenstrukturen wie Bäume und Sequenzen. Die Invariante dient dazu, valide von invaliden Daten abzugrenzen. Dieses Verfahren nutzt zwar keine Invarianten in OCL, sondern als direkt implementierte Methoden im Quellcode, aus den Invarianten in OCL lassen sich aber die entsprechenden Methoden generieren.

5.4 Kombinierte Ansätze zum UML-basierten Test

Alle bisher vorgestellten Ansätze zum Test basieren im Wesentlichen jeweils nur auf einem UML-Diagrammtyp. Daß ein Test möglichst alle Diagrammtypen einbeziehen sollte, wird anhand eines kleinen Beispiels motiviert. In den Abbildungen 5.3 und 5.4 werden zwei Sichten auf die Klasse `Account` modelliert.

Zustand	active	credit	debit
Invariante	self.isActive	self.balance >= 0	self.balance < 0

Abbildung 5.5: Zustandsinvarianten der Zustände `active`, `credit` und `debit`

In der OCL-Bedingung ist die Vorbedingung der Methode `withdraw` wahr (siehe Abbildung 5.3). Im Zustandsdiagramm ist die Methode `withdraw` jedoch nur im Zustand `active` definiert (siehe Abbildung 5.4). Also führt die Spezifikation des Zustandsdiagramms zu einer Einschränkung der Vorbedingung in OCL, die bei einem Test, der nur auf OCL-Constraints basiert, nicht berücksichtigt wird.

Im Zustandsdiagramm in Abbildung 5.3 führt ein Aufruf der Methode `withdraw` im Zustand `credit` zu einer nichtdeterministischen Auswahl der Transition: Entweder ist das `Account`-Objekt anschließend im Zustand `debit` oder verbleibt im Zustand `credit`.

Dieser Nichdeterminismus wird jedoch durch die Nachbedingung in OCL aufgehoben. Betrachtet man die Zustandsinvarianten der Zustände `debit` und `credit` (Abbildung 5.5), so läßt sich dies durch eine einfache Umrechnung zeigen.

Die Nachbedingung der Methode `withdraw` enthält sowohl Informationen über den Zustand vor als auch über den Zustand nach Ausführung von `withdraw`. Der Zustand vorher wird referenziert durch den Zusatz `@pre`, im Beispiel `self.balance@pre`. Der Zustand danach wird repräsentiert durch `self.balance`. Nun setzt die Nachbedingung diese beiden Zustände in Beziehung durch `self.balance = self.balance@pre - amount`.

Wir wissen, daß der Nachzustand entweder die Zustandsinvariante `self.balance < 0` (debit) oder `self.balance >= 0` (credit) besitzt. Wenn man dieses in die Nachbedingung einsetzt, läßt sich leicht eine Beziehung zwischen Vorzustand und dem Wert des Parameters `amount` der Methode feststellen:

Nachzustand `debit`	Nachzustand `credit`
`self.balance < 0`	`self.balance >= 0`
`self.balance@pre - amount < 0`	`self.balance@pre - amount >= 0`
`self.balance@pre < amount`	`self.balance@pre >= amount`

Ist folglich der Wert von `amount` größer als der Kontostand, so wird die Transition ausgewählt, die in den Zustand `debit` führt, im anderen Fall ändert sich der Zustand nicht.

Diese Berechnung läßt sich nicht verallgemeinern und ist immer abhängig von den speziellen Zustandsinvarianten, Vor- und Nachbedingungen und Klasseninvarianten, so daß in der vorliegenden Arbeit kein Algorithmus zur expliziten Auflösung von derartigen Nichtdeterminismen entwickelt wurde.

Doch dieses Beispiel zeigt, daß wichtige Informationen über mehrere Modelle verstreut sein können und OCL-Constraints dazu dienen, Modelle zu präzisieren und zu verfeinern. Da die UML diese Art der Modellierung explizit unterstützt, sollte ein UML-basiertes Testverfahren möglichst viel Information aus verschiedenen Sichten zum Test heranziehen. Im Beispiel ist ein Test, der nur auf einer der beiden Sichten beruht, zu weich in dem Sinne, daß fehlerhafte Implementierungen nicht erkannt werden, also zu einem falsch-positiven Testurteil führen.

Im Folgenden werden Testverfahren vorgestellt, die entweder die verschiedenen Sichten zur Spezifikation des Tests benutzen oder Testfälle und Testorakel aus verschiedenen Sichten der UML-Modelle ableiten.

Einige Autoren haben sich bereits mit dem Thema der Kombination von verschiedenen UML-Modellen zum Testen auseinandergesetzt. Als Beispiele sind hier der Ansatz aus [14, 15], der Ansatz aus [5, 4] und der Ansatz aus [44] zu nennen.

In [44] wird eine Kombination von Sequenzdiagrammen und UML-Statecharts beim Testen zugrunde gelegt. Beide Modelle dienen zur Generierung von Testfällen, werden jedoch unabhängig voneinander betrachtet. Es werden Testfälle entweder aus den Statecharts oder den Sequenzdia-

grammen gewonnen, das System wird anschließend mit beiden Testsuiten getestet. Zur Testfallgenerierung ist eine Transformation der Statecharts notwendig, die eine strikte Semantik zugrunde legt, die nicht in allen Punkten UML-konform ist.

Der in [14, 15] vorgestellte Ansatz beschreibt das Testsystem *TOTEM*, das auf Grundlage von verfeinerten Use-Cases Testfälle ableitet. Aktivitätsdiagramme werden benutzt, um Use-Case-Lebenszyklen zu definieren. Sequenzdiagramme dienen zur Beschreibung interner Abläufe in den Use-Cases. Beide Diagrammarten dienen anschließend zur Erzeugung von Testsequenzen. Für das Testorakel werden Vor- und Nachbedingungen spezifiziert, dabei wird die Verwendung von OCL als Notationssprache diskutiert. Eine Umsetzung der Integration der OCL-Constraints in das zu testende System mit Hilfe aspektorientierter Programmiertechniken wird in [13] beschrieben. Der Ansatz fokussiert durch die Verwendung von Use-Cases auf den Systemtest, zeigt jedoch sehr gut, wie die verteilte Information zum Testen kombinierbar ist.

Die in [5, 4] vorgestellte Technik nutzt das Use-Case-Diagramm, das jeweils einem Use-Case zugeordnete Sequenzdiagramm und das Klassendiagramm zum Integrationstest. Zunächst werden anhand der Sequenzdiagramme die einzelnen *Test-Units*, also die beteiligten Objekte, identifiziert. Für jedes der Objekte werden anhand des Klassendiagramms und der dort definierten Methodenparameter und Attribute Partitionen mit Hilfe des Category Partitioning [89] gebildet. Sequenzdiagramme dienen zur Identifikation von Methodensequenzen. Aus jeder identifizierten Methodensequenz entsteht nun durch Anwendung der Partitionierung eine Menge von Testfällen.

Als letztes soll hier das UML-Testing-Profile [111] betrachtet werden. Das Profil dient vorrangig zur Spezifikation von Testfällen und anderen testrelevanten Informationen in Form von UML-Modellen. Dabei werden verschiedene Diagrammypen mit Stereotypen angereichert. Vorrangig wird hier das Klassendiagramm benutzt, aber auch andere Diagrammtypen wie das Sequenzdiagramm werden mit Stereotypen versehen.

Obwohl das UML-Testing-Profile hauptsächlich zur expliziten Spezifikation von Testfällen dient und die UML nur die Spezifikationssprache für diese darstellt, gibt es einen Ansatz, der UML-Modelle aus den Softwareentwicklungphasen in Testing-Profile-Modellen transformiert [22]. Auch dort werden Testinformationen aus UML-Modellen gewonnen, jedoch fokussiert auf die benötigte Information für das Testing-Profile.

Kapitel 6

Zusammenfassung

Dieses Kapitel faßt die Ergebnisse des Teils zum UML-basierten Test zusammen und gibt einen Einblick in die Idee zum UML-basierten Testansatz, der im Rahmen der vorliegenden Arbeit entwickelt wurde.

Im Fokus der vorliegenden Arbeit wurden insbesondere die schwer automatisierbaren Testaktivitäten Testfallgenerierung und Testorakelerzeugung betrachtet. Wie die Untersuchungen zur Testbarkeit der einzelnen UML-Diagrammtypen zeigen, gibt es einige Diagrammtypen, die sich besser für die fokussierten Testaktivitäten eignen. Dabei stellt man fest, daß dynamische Diagrammtypen eine bessere Grundlage für die Testfallgenerierung und das Testorakel als statische Diagrammtypen bieten. Die OCL ist aufgrund ihres formalen Charakters gut für die automatische Verarbeitung beim Test geeignet.

Im Rahmen der vorliegenden Arbeit mußte eine Auswahl getroffen werden. Bei der Auswahl wurde insbesondere der Bezug des Diagrammtyps zur zu testenden Implementierung betrachtet, daß heißt, ob sich Elemente aus den UML-Modellen in der zu testenden Implementierung wiederfinden lassen. So modelliert z.B. das Aktivitätsdiagramm eher abstrakte Aktivitäten, die u.U. durch mehrere Methoden in der Implementierung umgesetzt wurden, während sich Nachrichten aus dem Sequenzdiagramm oft als einzelne Methoden in der Implementierung wiederfinden lassen. Zustände im Zustandsdiagramm lassen sich durch Zustandsinvarianten mit Attributen von Objekten in der zu testenden Implementierung in Verbindung setzen.

Eine weitere Einschränkung wurde bei der Unterstützung der verschiedenen Testphasen vorgenommen. Nicht alle Testphasen werden von allen Diagrammtypen unterstützt. Für den Systemtest sind vorrangig Use-Cases geeignet, die durch andere Diagrammtypen weiter verfeinert wurden, wie z.B. der Use-Case-getriebene Testansatz in [15] zeigt. Für den Integrationstest eignen sich Diagrammtypen, die Nachrichten zwischen verschiedenen Objekten modellieren, vorrangig Sequenzdiagramme, Kommunikationsdiagramme und Aktivitätsdiagramme, aber auch Schnittstellenspezifikationen durch Klasseninvarianten, Vor- und Nachbedingungen in OCL. Für den Klassentest vorrangig zu nutzen ist das Zustandsdiagramm. Das zu entwickelnde Testverfahren sollte zunächst nicht alle Testphasen unterstützen, sondern sich je nach gewählten Diagrammtypen auf die von diesem am besten unterstützten Testphasen konzentrieren.

Vorrangig aufgrund der enthaltenen Informationen und des guten Bezugs zur zu testenden Implementierung wurden folgende UML-Bestandteile ausgewählt:

Sequenzdiagramm. Für Informationen über Methodensequenzen und das Kommunikationsverhalten wurde das Sequenzdiagramm den anderen potentiell geeigneten Diagrammtypen vorgezogen. Insbesondere die Fähigkeit, nicht nur positive Sequenzen, sondern auch negative Sequenzen zu modellieren, macht den expliziten Test von Fehlerverhalten möglich.

Das Kommunikationsdiagramm liefert für den Test keine Informationen, die nicht auch aus einem entsprechenden Sequenzdiagramm gewonnen werden. Eine Reihe von Werkzeugen ist in der Lage, Kommunikationsdiagramme in Sequenzdiagramme zu transformieren, so daß der Ausschluß des Kommunikationsdiagramms vom Test keinen Nachteil darstellt. Das

Aktivitätsdiagramm ist viel schlechter mit der zu testenden Implementierung in Beziehung zu setzen und wird deshalb nicht berücksichtigt.

Zustandsdiagramm. Zustandsdiagramme bieten im Gegensatz zu anderen Diagrammtypen den Vorteil, daß sie sowohl für die Testfallgenerierung als auch als Basis für das Testorakel geeignet sind. Sie besitzen zudem im Gegensatz zu vielen anderen UML-Bestandteilen eine klare Semantik, so daß hier ein hoher Grad an Automatisierung des Tests zu erwarten ist.

Zwar werden z.B. in [33] auch Sequenzdiagramme als Testorakel verwendet, dieses Vorgehen ist jedoch aus den bereits in Kapitel 5 genannten Gründen nicht praktikabel.

OCL-Constraints. OCL-Constraints bieten sich vorrangig als Testorakel an, da sie Bedingungen definieren, die gelten müssen. Diese Bedingungen sind während des Tests leicht überprüfbar.

Potentiell sind OCL-Constraints in vielen Bereichen einsetzbar, so daß eine Beschränkung vorgenommen werden mußte. In der vorliegenden Arbeit werden OCL-Constraints in Form von Klasseninvarianten, Vor- und Nachbedingungen verwendet. Diese Constraints definieren Schnittstellen zwischen Objekten und sind deshalb im Integrationstest einsetzbar.

Sie sind auch eine gute Ergänzung der Zustandsdiagramme, da sie diese um explizit spezifizierte Bedingungen ergänzen, während Zustandsdiagramme implizite Bedingungen spezifizieren.

Aufgrund der Auswahl beschränkt sich der Test auf die Testphasen Klassentest und Integrationstest.

Im Gegensatz zu den meisten anderen Testansätzen auf Basis der UML werden in der vorliegenden Arbeit Testfälle und Testorakel aus verschiedenen UML-Bestandteilen abgeleitet. Diese Vorgehensweise ist aus zwei Gründen von Vorteil: Einerseits, weil sich so verschiedene Sichten zum Test heranziehen lassen. Andererseits eignen sich die unterschiedlichen UML-Diagrammtypen mehr oder weniger gut für den Einsatzzweck und ergänzen sich somit in ihrer Kombination.

Ein Testansatz, der ebenfalls mehrere Diagrammtypen zum Test betrachtet, findet sich in [14, 15]. Auch dort werden für Testfälle und Testorakel unterschiedliche UML-Bestandteile herangezogen. Der Fokus des Tests ist dort jedoch der Systemtest, folglich werden Testfälle vornehmlich aus angereicherten Use-Case-Diagrammen und Aktivitätsdiagrammen abgeleitet. Für das Testorakel werden ebenfalls OCL-Constraints in Form von Klasseninvarianten, Vor- und Nachbedingungen genutzt. Für die Integration der Testorakel werden in einer anderen Arbeit aus der gleichen Forschungsgruppe ebenfalls aspektorientierte Programmiertechniken vorgeschlagen [13][1]. Dieser Testansatz läßt sich sehr gut mit der in der vorliegenden Arbeit vorgestellten Technik kombinieren. Durch diese Kombination werden alle Testphasen vom Klassen- bis zum Systemtest unterstützt.

Sequenzdiagramme als Basis für die Testfallgenerierung zu verwenden besitzt den Nachteil, daß diese in der Regel keine Informationen über den Zustand der beteiligten Objekte und vorausgegangene Ereignisse im System beinhalten. In [33] werden drei verschiedene Lösungen für dieses Problem vorgeschlagen:

- Start aller Sequenzen im Initialzustand. Diese Lösung scheint wenig praktikabel, da ein Teil jedes Sequenzdiagramms nur zur Initialisierung dient und dadurch die eigentlich relevante Sequenz in den Hintergrund tritt.
- Explizite Spezifikation der Initialzustände. Diese Lösung erfordert einen enormen Aufwand, da für alle Sequenzdiagramme die Zustände der beteiligten Objekte zu spezifizieren sind.
- Erzeugung von Objekten von außen und die Benutzung dieser Objekte durch den Testfall. Diese Lösung kann z.B. eine Datenbank von Objekten nutzen. Aber auch hier ist die Frage nach den Zuständen der Objekte, d.h. ob ein Szenario in einem bestimmten Zustand

[1] Weitere Arbeiten zur Nutzung aspektorientierter Programmiertechniken im Test werden in Kapitel 13 vorgestellt.

ausführbar ist, nicht gelöst. Zudem stellt sich die Frage nach dem Bezug zu den Zuständen der Objekte im realen Einsatz des Systems, d.h. entspricht der Test überhaupt dem normalen Systemverhalten.

Die in der vorliegenden Arbeit vorgeschlagene Lösung leitet die Initialisierungssequenzen stattdessen aus einem weiteren Diagrammtyp ab - dem Zustandsdiagramm - und bleibt deshalb innerhalb der Spezifikation des Systems, ohne die Sequenzdiagramme zusätzlich anzureichern.

Um aus Sequenzdiagrammen nicht nur Testfälle abzuleiten, sondern auch Testdaten, ist eine Ergänzung der in der vorliegenden Arbeit vorgestellten Technik mit dem in [33] vorgestellten Ansatz denkbar. Testdaten werden dort durch Attributierung der Sequenzdiagramme automatisch erzeugt, während Testdaten in der vorliegenden Arbeit nur in eingeschränkter Weise durch Äquivalenzklassenbildung aus OCL-Constraints gewonnen werden.

Im Gegensatz zu [5, 4] dient nicht das Klassendiagramm und die daraus abgeleitete Partitionierung des Eingabedatenraums primär zur Bestimmung der verschiedenen Testfälle, die sich aus einem Sequenzdiagramm ableiten lassen, sondern das Zustandsdiagramm und die verschiedenen Zustände der beteiligten Objekte.

Anders als in den meisten Arbeiten (z.B. in [63]), die Zustandsdiagramme oder verwandte Zustandsgraphen als Basis für den Test verwenden, werden in der vorliegenden Arbeit Testfälle nicht durch Traversierung des Zustandsdiagramms oder anhand eines Überdeckungskriterium gewonnen. Die Zustandsdiagramme dienen ausschließlich zur Initialisierung der aus den Sequenzdiagrammen abgeleiteten Testsequenzen.

Auch [98] beschreibt eine Technik, die sich nicht an vollständiger Überdeckung eines Zustandsdiagramms orientiert. Dort wird durch nichtdeterministische Auswahl der Transitionen die Menge der Testfälle beliebig groß und Zyklen werden u.U. mehrfach durchlaufen. Die in der vorliegenden Arbeit vorgestellte Technik dagegen trifft die Auswahl der Transitionen, die im Statechart durch den Test ausgeführt werden sollen, auf Basis von Sequenzdiagrammen.

Im folgenden Teil wird das entwickelte UML-basierte Testverfahren im Einzelnen vorgestellt.

Teil II

Testerzeugung aus UML-Modellen

If a model is produced at all during object-oriented development, it is likely to be a collection of UML diagrams. UML models are an important source of information for test design that should not be ignored. Robert V. Binder [10]

Kapitel 7

Das Testsystem UT^3

In diesem Kapitel wird ein Überblick des im Rahmen der vorliegenden Arbeit entworfenen und implementierten Testsytems UT^3 (*UML-Based Test Tool*) gegeben, bevor in weiteren Kapiteln die einzelnen Techniken genauer beleuchtet werden.

Üblicherweise bestehen Testsysteme aus einem Testtreiber, der die Testdaten an das zu testende System sendet, und einem Testorakel, das unter Berücksichtigung der Testdaten, der Antworten des zu testenden Systems und der Zustände des zu testenden Systems ein Testurteil ableitet (siehe Abbildung 7.1). Im Falle des Testsytems UT^3 lautet das Testurteil analog zu dem in Kapitel 3 definierten Testmodell *pass*, *fail* oder *inconclusive*. In der schematischen Abbildung sind die Bestandteile des Testsystems als Rechtecke dargestellt, das zu testende System in einem Parallelogramm und die Testausführung in einem Rechteck mit abgerundeten Ecken. Die Testausführung wird durch das Laufzeitssytem der verwendeten Programmiersprache übernommen, auf dem der Testtreiber aufbaut.

Abbildung 7.2 gibt einen Überblick über das Testsystem UT^3 und die Beziehung zu den UML-Modellen. UML-Modelle dienen als Basis sowohl für die Testdaten als auch für das Testorakel.

Zur Ermittlung der Testdaten sind zwei Schritte nötig. Aus den UML-Modellen werden zunächst durch Kombination von Sequenzdiagrammen und UML Protocol State Machines abstrakte Testfälle generiert. Ein Testfall ist eine Sequenz von Methodenaufrufen mit Bedingungen für die Parameter der aufgerufenen Methoden und die Attribute der beteiligten Objekte.

In einem zweiten Schritt, der zur Zeit noch überwiegend manuell erfolgen muß, werden Testdaten zu den Testfällen erzeugt. Testdaten belegen die Parameter und Attribute unter Berücksichtigung der im Testfall spezifizierten Bedingungen mit Werten.

Das Testorakel

- berechnet die erwartete Antwort und den erwarteten Zustand des zu testenden Systems,
- vergleicht die realen Anworten und den realen Zustand des zu testenden Systems mit den Sollwerten,
- leitet das Testurteil zur Laufzeit des Tests ab.

Auch zur Erzeugung des Testorakels sind zwei Schritte notwendig. Im ersten Schritt werden UML Protocol State Machines und OCL Constraints in Form von Invarianten, Vor- und Nachbedingungen zu neuen Modellen kombiniert. Als Sprache zur Beschreibung dieser neuen Modelle dient erneut die UML. Aus diesen neuen UML-Modellen werden in einem zweiten Schritt Testorakel erzeugt, die in Code vorliegen und direkt in das zu testende System integriert werden.

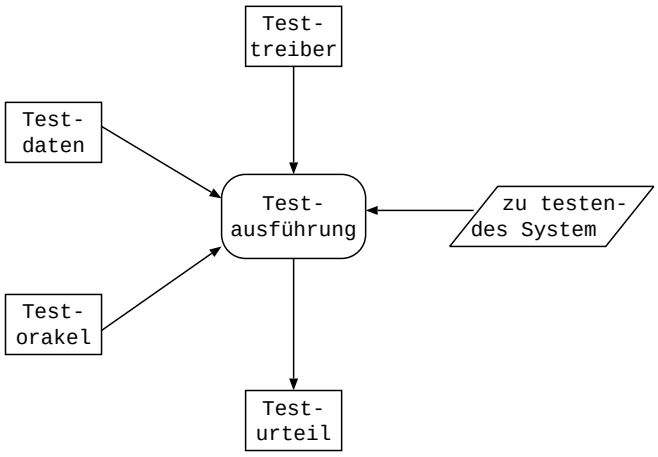


Abbildung 7.1: Schematische Darstellung eines Testsystems

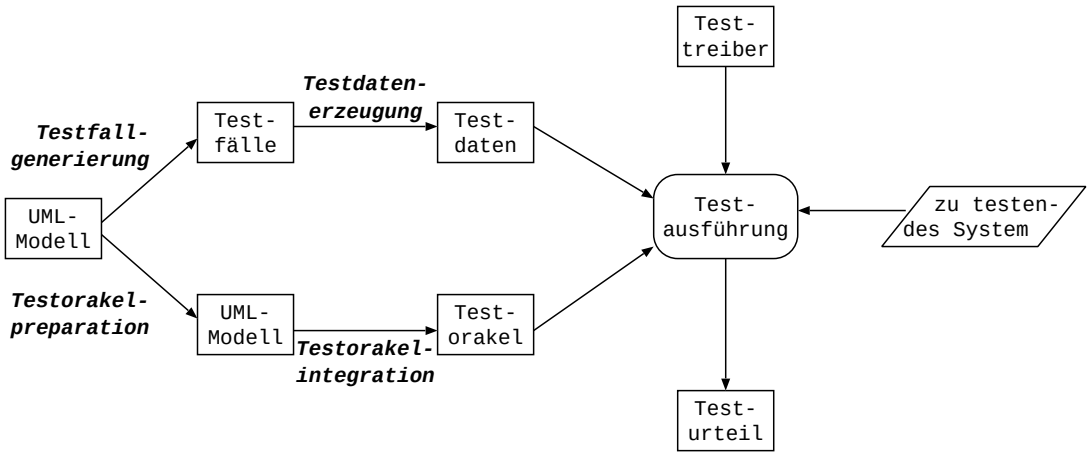


Abbildung 7.2: Schematische Darstellung des Testsystems UT^3

Den Schritten zur Erzeugung der abstrakten Testfälle und zur Kombination von UML-Modellen als Vorbereitung des Testorakels widmet diese Arbeit jeweils ein Kapitel.

Kapitel 9 beschreibt die Generierung der abstrakten Testfälle und gibt einen Ausblick auf die Erzeugung von konkreten Testdaten zu den Testfällen.

In Kapitel 10 wird die Kombination von Protocol State Machines und OCL-Constraints als Testorakel vorgestellt.

Da zur Integration der Testorakel in das zu testende System aspektorientierte Programmiertechniken verwendet werden, werden aspektorientierte Programmiertechniken und die aspektorientierte Integration der Testorakel in Teil III gesondert behandelt.

7.1 Die Pakete des Testsystems

Das Testsystem UT^3 ist in Java implementiert und gliedert sich in vier Pakete. Im Basispaket `ut.ut_base` sind alle Datenstrukturen und die Basisalgorithmen implementiert, die in Kapitel 8 präsentiert werden. Daneben gibt es das Paket `ut.ut_main`, das Hauptprogramme für die Testfallgenerierung, die Testorakelerzeugung und die Testausführung zur Verfügung stellt.

Die anderen zwei Pakete entsprechen den einzelnen Techniken, die in der vorliegenden Arbeit vorgestellt werden.

Für die Testfallgenerierung ist das Paket `ut.ut_tfgen` zuständig. Die zugehörigen Techniken sind in Kapitel 9 beschrieben.

Das Testorakel wird durch das Paket `ut.ut_ora` erzeugt. Dieses Paket ist sowohl für die Ableitung des Testorakels in Kapitel 10 als auch die Generierung der Object Teams, die in den Kapiteln 14 und 15 beschrieben sind, zuständig.

Eine umfassende Übersicht über alle Pakete und Subpakete des Testsystems UT^3 bietet der Anhang A.

7.2 Fallbeispiel: Banksystem

Als Fallbeispiel zur Vorstellung der einzelnen Techniken dient ein Banksystem. Dieses System wurde implementiert und alle Testverfahren wurden darauf angewendet. Hier soll ein Überblick über das Fallbeispiel gegeben werden, da alle Techniken anhand des Fallbeispiels erläutert werden.

Zentrale Klasse im Banksystem ist die Klasse `Bank`. Eine Bank besteht aus einer Menge von Konten und einer Menge von Kunde. Konten sind Kunden zugeordnet. Die Bank kann mehrere Filialen haben, wobei Konten immer genau einer Filiale zugeordnet sind.

Es gibt verschiedene Arten von Konten: Sparkonten, Girokonten und Anlagekonten. Die verschiedenen Kontoarten weisen unterschiedliches Verhalten auf. Geld kann auf Konten eingezahlt und von Konten abgehoben werden. Bei Sparkonten und Anlagekonten ist ein Überziehen nicht erlaubt, von Girokonten läßt sich Geld bis zum Dispositionskredit abheben. Beträge auf Sparkonten und Anlagekonten werden im Gegensatz zum Girokonto verzinst. Bei Sparkonten ist der täglich abzuhebende Betrag beschränkt, bei den anderen beiden Kontoarten nicht.

Die Kunden sind entweder Privat- oder Firmenkunden. Privatkunden erhalten einen einkommensabhängigen Dispokredit, der bei Verdienständerung angepaßt wird. Firmenkunden besitzen in dem modellierten Beispiel keine Sparkonten. Der Dispositionskredit ist bei Firmenkonten Verhandlungssache bei der Kontoeröffnung und über die Lebenszeit des Kontos fix.

Abbildung 7.3 zeigt einen Überblick über die Klassen des Banksystems und ihre Beziehungen zueinander[1]. Alle Banken sind einem Bankennetzwerk zugeordnet, damit Transaktionen zwischen Konten verschiedener Banken möglich sind.

[1] Alle Begriffe des Fallbeispiels sind in englischer Sprache gehalten.

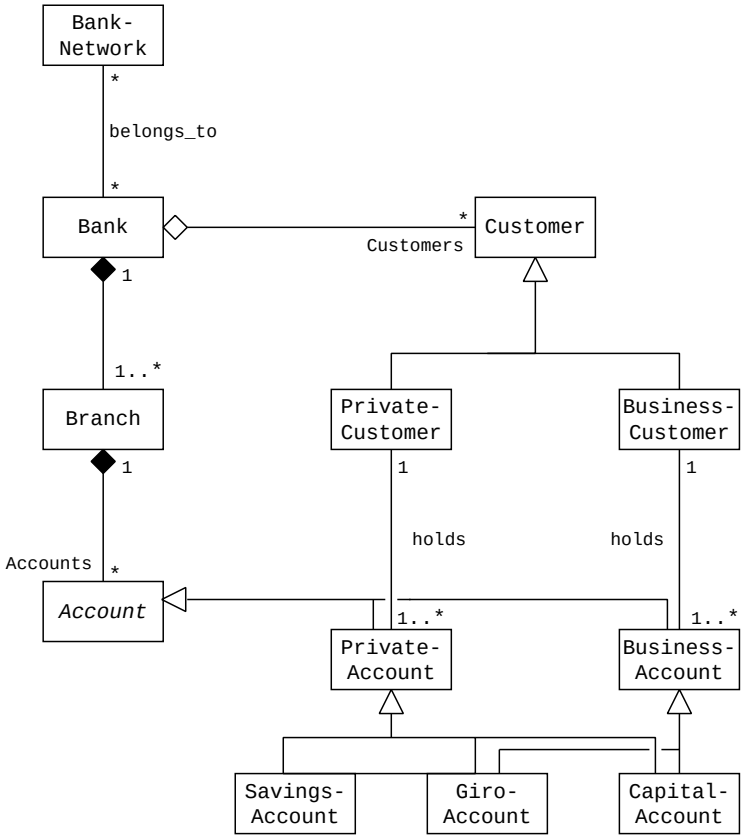
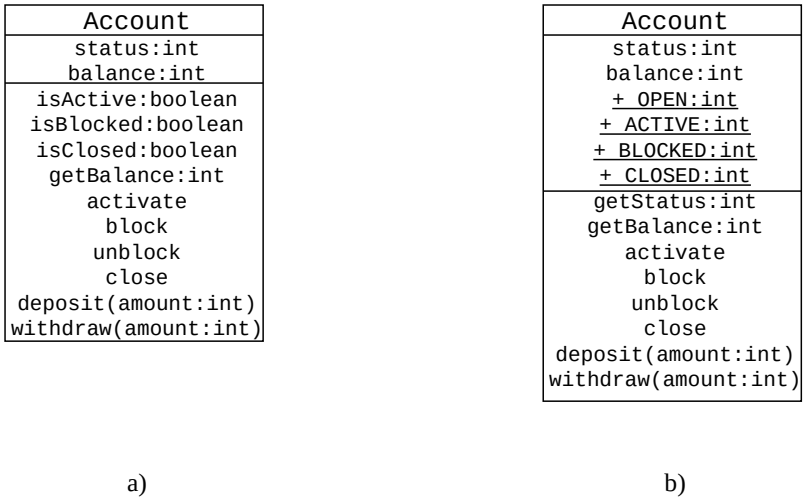


Abbildung 7.3: Klassen des Fallbeispiels *Banksystem*



a) b)

Abbildung 7.4: Klasse `Account`: a) mit 4 Observer-Methoden, b) mit 2 Observer-Methoden

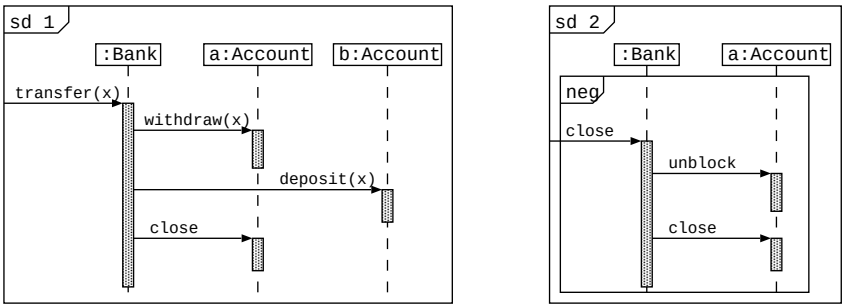


Abbildung 7.5: Positives und negatives Sequenzdiagramm

Jede Bank aggregiert eine Menge von Kunden und eine Menge von Konten. Kunden und Konten werden durch Spezialisierungen verfeinert in Privat- und Firmenkunden respektive Privat- und Firmenkonten. Von Privat- und Firmenkonten werden die Kontoarten Sparkonto, Girokonto und Anlagekonto abgeleitet[2].

In Abbildung 7.4 a) wird die Klasse `Account` mit all ihren Instanzvariablen und Methoden präsentiert, da diese Klasse in den meisten Beispielen verwendet wird. Die Klasse `Account` besitzt zwei Instanzvariablen:

`status` zeigt den Status des aktuellen Kontos an. Es gibt vier verschiedene Stati. Das Konto ist entweder geöffnet, aktiv, gesperrt oder geschlossen.

`balance` enthält den Kontostand des aktuellen Kontos.

Vier Observer-Methoden der Klasse `Account` geben Werte der Instanzvariablen zurück:

`isActive` liefert wahr, wenn das aktuelle Konto aktiv ist.

`isBlocked` liefert wahr, wenn das aktuelle Konto gesperrt ist.

`isClosed` liefert wahr, wenn das aktuelle Konto geschlossen ist.

`getBalance` liefert den Kontostand des aktuellen Kontos.

Sechs Update-Methoden der Klasse `Account` ändern den Zustand eines Kontos:

`activate` aktiviert das aktuelle Konto.

`block` sperrt das aktuelle Konto.

`unblock` entsperrt das aktuelle Konto.

`close` schließt das aktuelle Konto.

`deposit` zahlt einen Betrag auf das aktuelle Konto ein.

`withdraw` hebt einen Betrag vom aktuellen Konto ab.

Eine vereinfachte Definition der Klasse `Account` wird in Abbildung 7.4 b) dargestellt. Sie unterscheidet sich von Abbildung 7.4 a) nur insofern, daß der Zustand der Instanzvariable `status` nicht mehr durch vier, sondern nur durch eine Methode zurückgegeben wird. Die Werte, die die Methode `getStatus` zurückliefert, werden durch die statischen Klassenvariablen `OPEN`, `ACTIVE`, `BLOCKED` und `CLOSED` definiert.

Für die Anwendung der Update-Methoden gelten besondere Regeln, die durch Zustandsdiagramme und OCL-Constraints spezifiziert sind (siehe Abschnitte 7.2.2 und 7.2.3).

[2]Zu beachten ist, daß die gewählte Implementierungssprache Java keine Mehrfachvererbung unterstützt (im Beispiel erben die Klassen `GiroAccount` und `CapitalAccount` jeweils von den Klassen `PrivateAccount` und `BusinessAccount`). Deshalb weicht die reale Implementierung ein wenig vom gegebenen Klassendiagramm ab, dies ist für die vorliegende Arbeit jedoch nicht von Relevanz.

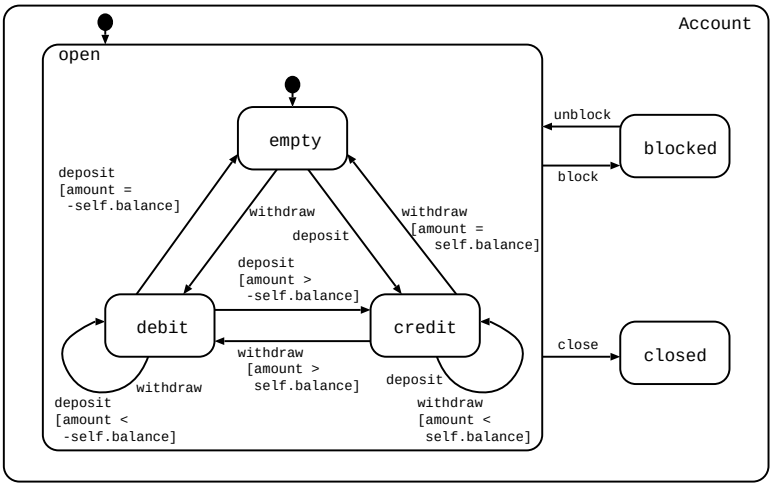


Abbildung 7.6: Zustandsdiagramm für die Klasse `Account`

```
context Account inv:
      true
```

Abbildung 7.7: Klasseninvariante für `Account`

Es wurden mehrere Szenarien modelliert und implementiert, wie zum Beispiel das Eröffnen eines Kontos mit gleichzeitiger Deponierung eines Betrags oder die Überweisung eines Betrags von einem zu einem anderen Konto. Einzelne Szenarien werden in Abschnitt 7.2.1 vorgestellt.

Für einzelne Klassen, im Speziellen die Klasse `Account`, die das Konto repräsentiert, wurden Zustandsdiagramme entworfen. Diese werden in Abschnitt 7.2.2 vorgestellt.

Schließlich wurden für einzelne Klassen Invarianten und für einzelne Methoden Vor- und Nachbedingungen in OCL spezifiziert, auch hier insbesondere für die Klasse `Account`, die in Abschnitt 7.2.3 präsentiert werden.

7.2.1 Sequenzen

Als Sequenzdiagramm dient eine positive und eine negative Sequenz als Beispiel.

Die positive Sequenz in Abbildung 7.5 (links) beschreibt den Transfer von einem Konto auf ein anderes mit abschließenden Kontoabschluß des ersten Kontos.

Das negative Sequenzdiagramm in Abbildung 7.5 (rechts) beschreibt ein Szenario, daß so nicht auftreten darf. In diesem Fall darf ein Konto nicht zunächst entsperrt und anschließend geschlossen werden.

7.2.2 Zustandsautomaten

Es werden eine Reihe von Zustandsautomaten für die Klasse Konto als Beispiel verwendet. Jeder dieser Zustandsautomaten dient dazu, einen anderen Aspekt des Testverfahrens zu illustrieren.

```
context Account::withdraw (amount:int)
   pre : amount > 0
   post: self.balance = self.balance@pre - amount
```

Abbildung 7.8: Vor- und Nachbedingung für `withdraw` der Klasse `Account`

```
context SavingsAccount inv:
      self.balance >= 0
```

Abbildung 7.9: Klasseninvariante für `SavingsAccount`

```
context SavingsAccount::withdraw (amount:int)
   pre : amount > 0 and self.balance >= amount
   post: self.balance = self.balance@pre - amount
```

Abbildung 7.10: Vor- und Nachbedingung für `withdraw` der Klasse `SavingsAccount`

Exemplarisch sei hier nur ein Zustandsautomat beschrieben (siehe Abbildung 7.6). Die anderen ähneln dem hier vorgestellten Zustandsautomaten, da sich alle Zustände auch in den anderen Zustandsdiagrammen wiederfinden.
Ein Konto besitzt im Wesentlichen drei Zustände:

`open.` Im geöffneten Zustand sind Einzahlungen und Auszahlungen möglich. In der Regel ist dieser Zustand weiter verfeinert in die Zustände `debit`, `credit` und `empty`.

`blocked.` Im gesperrten Zustand ist ein Auszahlen nicht möglich.

`closed.` Der geschlossene Zustand repräsentiert das Ende des Lebenszyklus eines Kontoobjekts. In diesem kann weder eine Einzahlung noch eine Auszahlung vorgenommen werden. Nach dem Betreten wird dieser Zustand nicht mehr verlassen.

7.2.3 OCL-Constraints

Als Beispiel dienen in der vorliegenden Arbeit die Klasseninvarianten der Klasse `Account` (Abbildung 7.7) und ihrer Unterklasse `SavingsAccount` (Abbildung 7.9). Ein Sparkonto muß immer einen positiven Kontostand aufweisen, für ein Konto gilt keine Einschränkung.

Zudem sind Vor- und Nachbedingung der Methode `withdraw` der genannten Klassen spezifiziert (in Abbildung 7.8 respektive Abbildung 7.10). Ein Abheben ist beim Sparkonto nur erlaubt, wenn der abzuhebende Betrag den Kontostand nicht übersteigt, beim Konto ist ein Abheben immer erlaubt, vorausgesetzt, der abzuhebende Betrag ist positiv. Die Nachbedingung besagt bei beiden Kontoarten, daß der Kontostand um den abgehobenen Betrag reduziert wird.

Kapitel 8

Basisformalismen

Dieses Kapitel stellt die verwendeten Basisformalismen vor. Für alle verwendeten UML-Diagrammtypen wird eine Formalisierung vorgenommen. Alle Einschränkungen an die verwendeten UML-Diagrammtypen werden hier beschrieben. Die Formalisierung ist die Basis aller im weiteren Verlauf der Arbeit angegebenen Algorithmen. Aufgrund der Komplexität des Testsystems UT^3 ist eine vollständige Beschreibung aller Algorithmen nicht möglich. Es werden jedoch die wichtigsten Algorithmen vorgestellt.

Zunächst wird die verwendete mathematische Notation vorgestellt. Anschließend wird die Formalisierung von Protocol State Machines und Sequenzdiagrammen präsentiert. Dabei werden die einzelnen Sichten, die durch die Diagrammtypen definiert sind, einzeln betrachtet, ohne eine Kombination mit anderen Sichten zu berücksichtigen. Den Abschluß bilden die von der vorliegenden Arbeit unterstützten OCL-Constraints. Auch diese werden ohne Berücksichtigung anderer Sichten betrachtet.

8.1 Notationen

Zur Formalisierung der Basisformalismen wird eine übliche mathematische Notation aus der Mengentheorie und der Prädikatenlogik verwendet.
Zur Beschreibung der Algorithmen kommen folgende Konstrukte zum Einsatz:

- Eine Funktion besteht aus einem Funktionskopf und einem Funktionsrumpf, die durch das Symbol $\boxed{==}$ voneinander getrennt werden. Ein abschließendes Symbol oder Schlüsselwort wird nicht benötigt.

- Der Funktionskopf besteht aus einem Funktionsnamen und in Klammern eingeschlossenen Parameterobjekten mit Name und Typ. Bei parameterlosen Funktionen werden die Klammern weggelassen. Rückgabewerte, sofern vorhanden, werden hinter der Klammer mit ihrem Typ angegeben. Rückgabewerte entfallen bei Funktionen, die auf dem Zustand der übergebenen Objekte arbeiten.
  Beispiel: $\boxed{alg\ (s : S, t : T) : R}$ definiert die Funktion alg mit den Parametern s vom Typ S und t vom Typ T und einem Rückgabewert vom Typ R.

- Anweisungen werden mit $\boxed{;}$ verknüpft oder in verschiedene Zeilen untereinander geschrieben. Wenn im Folgenden von einer Anweisung die Rede ist, so ist damit entweder eine einzelne Anweisung oder eine Anweisungsfolge gemeint.

  Bei längeren zusammengehörigen Anweisungsfolgen, z.B. nach einer Schleifendefinition, die nicht auf eine Zeile passen, wird die Zusammengehörigkeit durch entsprechende Einrückungen verdeutlicht. Es handelt sich in diesem Fall eigentlich um eine einzige Zeile. Die letzte Anweisung eines Algorithmus berechnet den Rückgabewert.

- Eine Zuweisung erfolgt mit dem Operator $\boxed{:=}$.

- Das Konstrukt $\boxed{\textbf{if}\ P\ \textbf{then}\ E_1\ \textbf{else}\ E_2}$ steht für eine von P abhängige Alternative. Die Anweisungen E_1 und E_2 bezeichnen die beiden Alternativen. Der **else**-Teil ist optional.

- Schleifen werden mit dem Schlüsselwort $\boxed{\textbf{loop}}$ definiert.
  Beispiel: $\boxed{\textbf{loop}\ P : E}$ führt die Anweisung E aus, solange die Bedingung P gilt. $\boxed{\textbf{loop}\ x : E}$, wobei x eine natürliche Zahl ist, führt die Anweisung E genau x-mal aus.

- Eine Schleife über alle Elemente eines Typs wird definiert durch das Schlüsselwort $\boxed{\textbf{all}}$.
  Beispiel: $\boxed{\textbf{all}\ s \in S : E}$ beschreibt die Anwendung der Anweisung E auf alle Elemente vom Typ S.
  Zusätzlich kann eine Bedingung die Menge der Elemente, auf denen die Schleife arbeitet, einschränken. Diese Bedingung wird als Prädikat hinter die Mengendefinition geschrieben, getrennt durch das Symbol |.
  Beispiel: $\boxed{\textbf{all}\ s \in S \mid P : E}$ beschreibt die Anwendung der Anweisung E auf alle Elemente vom Typ S, für die das Prädikat P gilt.

- Die Auswahl genau eines Elements aus einer Menge von Elementen, wobei auf das ausgewählte Element eine bestimmte Bedingung zutrifft, wird notiert durch $\boxed{\textbf{sel}\ s \in S \mid P : E}$, wobei P die Bedingung angibt, die ein Element s vom Typ S erfüllen muß, und E die Anweisung, die auf dieses Element angewendet wird.

- Mit dem Konstrukt $\boxed{\textbf{let}\ a := E_1\ : E_2}$ werden Teilberechnungen in die Anweisung E_1 ausgelagert und unter Verwendung der Variablen a in der Anweisung E_2 benutzt. Eine **let**-Anweisung kann auch mehrere Variablen definieren, die durch Kommata voneinander getrennt werden.
  Beispiel: $\boxed{\textbf{let}\ a := E_a, b := E_b : E}$.

- Neben den Basistypen und den Klassentypen werden die Typen *Set* und *Seq* verwendet. *Set* repräsentiert eine Menge von Objekten, *Seq* eine Sequenz. Der Typ der Elemente wird hinter dem Typ *Set* oder *Seq* notiert.
  Beispiel: $\boxed{s : Set\ \Sigma}$ definiert eine Menge s, wobei Σ der Typ der Elemente ist.
  $\oplus$ fügt ein Element hinzu, im Falle einer Menge nur, wenn es nicht bereits vorhanden ist. Im Falle einer Sequenz wird das Element am Ende angehängt. $\ominus$ entfernt ein Element. $\oplus$ und $\ominus$ sind auch auf zwei Mengen anwendbar, wobei die Elemente der zweiten Menge der ersten hinzugefügt oder aus dieser entfernt werden.
  Über Sequenzen und Mengen läßt sich mit Hilfe des **all**-Konstrukts iterieren. Bei Sequenzen erfolgt die Iteration von vorn nach hinten. Ob ein Element in einer Menge oder einer Sequenz enthalten ist, wird mit $\in$ geprüft. **sel** liefert das erste Element einer Sequenz oder ein beliebiges aus einer Menge. Ist eine Bedingung gegeben, so wird analog das Element ausgewählt, auf das die Bedingung zutrifft.
  Beispiel: $\boxed{\textbf{if}\ \neg s \in S\ \textbf{then}\ S \oplus s}$ fügt ein Element s einer Menge oder einer Sequenz S hinzu.

Eine Definition der Syntax zur Bescheibung der Algorithmen findet sich in Anhang C.

8.2 Protocol State Machines

Formale Semantiken für UML-Statecharts werden u.a. in [28, 64, 98] definiert. Für die vorliegende Arbeit erfolgt die Formalisierung der Protocol State Machines in Anlehnung an die Definition von Statecharts in [98][1].

[1] Obwohl es sich bei den Statecharts in [98] um Statecharts mit asynchronen Events und einem Aktionsteil in den Transitionen handelt, läßt sich die dort beschriebene Formalisierung ohne weiteres auf Statecharts mit synchronen Events und Nachbedingungen von Transitionen übertragen.

Im Folgenden wird zunächst die formale Beschreibung von Protocol State Machines angegeben. Anschließend erfolgt die Definition einer formalen Ausführungssemantik für Protocol State Machines, gefolgt von der Beschreibung einiger grundlegender Algorithmen auf Protocol State Machines, die als Basis für die in den folgenden Kapiteln beschriebenen Techniken dienen. In den folgenden Kapiteln werden Protocol State Machines oft vereinfacht Zustandsdiagramme genannt, gemeint sind immer die hier beschrieben Protocol State Machines mit allen hier definierten Einschränkungen.

8.2.1 Formale Beschreibung von Protocol State Machines

Definition: *Protocol State Machine*
Protocol State Machines werden wie folgt als 7-Tupel formalisiert:

$$SC = (\Sigma, \Sigma_{is}, \tau, \chi, \Lambda, \Delta, E).$$

Die Menge der Zustände Σ ist nicht leer und endlich. Zustände sind hierarchisch organisiert. Die Menge der Zustände $\Sigma_{is} \subseteq \Sigma$ ist die Menge der Initialzustände. Jeder Zustand besitzt eine Rolle in der Hierarchie, definiert durch die Funktion $\tau : \Sigma \rightarrow \{simple, and, xor\}$. Diese Funktion partitioniert die Menge der Zustände Σ in die Menge der Basiszustände (*Simple States*) und die Menge der komponierten Zustände (*Composite States*). Die Menge der komponierten Zustände enthält die Menge der Zustände mit orthogonalen Unterzuständen Σ_{and} und die Menge der Zustände mit sich gegenseitig ausschließenden Unterzuständen Σ_{xor}. Es gelten:

- $\Sigma_{simple} \cup \Sigma_{xor} \cup \Sigma_{and} = \Sigma$
- $\Sigma_{simple} \cap \Sigma_{xor} \cap \Sigma_{and} = \emptyset$

In der UML-Spezifikation entsprechen die Zustände Σ_{xor} den Zuständen mit nur einer Region und die Zustände Σ_{and} den Zuständen mit mehr als einer Region. Bei den Zuständen Σ_{and} ist jede Region durch einen weiteren Zustand aus der Menge Σ verfeinert, was der klassischen Formalisierung der Statecharts entspricht, jedoch keine direkte Abbildung der UML-Statecharts widerspiegelt[2].

Jedem Zustand s ist eine Zustandsinvariante s_{inv} vom Typ Γ zugeordnet, wobei Γ der Typ aller OCL-Constraints ist. Die Zustandsinvariante beschreibt dabei den abstrakten Zustand, wobei dieser sowohl ein interner abstrakter Zustand als auch ein externer abstrakter Zustand sein kann (siehe Kapitel 4.2.3). Die einem Zustand zugeordnete oder spezifizierte Zustandsinvariante s_{inv} entspricht jedoch nicht der allgemein gültigen Definition einer Zustandsinvarianten, sondern nur dem Teil der Zustandsinvarianten, der den betrachteten Zustand von anderen Zuständen auf der gleichen Hierarchieebene mit dem gleichen Oberzustand unterscheidet. Um die allgemeingültige Zustandsinvariante s^+_{inv} eines Zustands zu erhalten, müssen alle Zustandsinvarianten seiner Oberzustände mit diesem konjugiert werden. Beispiel: Seien zwei Zustände $s1$ und $s2$ Unterzustände des Zustand s und $s1_{inv}$ die Zustandsinvarianten von $s1$ und $s2_{inv}$ die Zustandsinvariante des Zustands $s2$. Dann ist $s1^+_{inv} = s1_{inv} \wedge s^+_{inv}$ und $s2^+_{inv} = s2_{inv} \wedge s^+_{inv}$. Für den Wurzelzustand gilt: $s^+_{inv} = s_{inv}$.

Die Hierarchie wird durch die Relation χ definiert, die die Beziehung zwischen zwei Zuständen in der Hierarchie beschreibt. Seien a und b zwei Zustände $a \in \Sigma$ und $b \in \Sigma$, so beschreibt der Ausdruck $a\chi b$, daß b ein (direkter) Unterzustand von a ist und a der Vorgängerzustand von b ist.

Die Zustände eines Zustandsdiagramms bilden durch die Eigenschaften der Relation χ eine Baumstruktur. Dabei sind die komponierten Zustände die Knoten und die Basiszustände die Blätter. Die Relation χ definiert die Kanten. Es gelten folglich alle Eigenschaften von Bäumen, so ist es z.B. nicht erlaubt, daß ein Zustand sich selbst enthält, oder daß ein Basiszustand Unterzustände besitzt. Die Wurzel des Baumes der Zustände wird als Wurzelzustand r bezeichnet, für diesen gilt: $\nexists\, a \mid a\chi r$. Basiszustände heißen auch innerste Zustände, der Wurzelzustand wird auch als äußerster Zustand bezeichnet.

[2] In der UML sind die Regionen eines Zustands aus der Menge Σ_{and} keine eigenständigen Zustände.

Die Relation χ^+ bildet den transitiven Abschluß der Relation χ, d.h. für zwei Zustände s_1 und s_2 gilt $s_1 \chi^+ s_2$, wenn s_2 direkter oder indirekter Unterzustand von s_1 ist.
Außerdem gelten folgende Eigenschaften:

- Alle Unterzustände eines Zustands aus der Menge Σ_{and} müssen wieder komponierte Zustände sein:
  $\forall a \in \Sigma_{and}, b \in \Sigma \mid a \chi b \Rightarrow \neg\, b \in \Sigma_{simple}$

- Für jeden komponierten Zustand der Menge Σ_{xor} existiert ein eindeutiger Unterzustand aus der Menge der Initialzustände Σ_{is}:
  $\forall a \in \Sigma_{xor}\, \exists_1\, b \in \Sigma_{is} \mid a \chi b$

Definition: *Transition, Quellzustand, Zielzustand*
Zustände sind verbunden durch Transitionen. Jede Transition besitzt ein Label. Labels haben die Form $e[c_{pre}]/[c_{post}]$. Die Menge aller zulässigen Label ist definiert durch Λ. Eine Zuordung der Label erfolgt durch die Transitionsrelation $\Delta \subseteq \Sigma \times \Lambda \times \Sigma$, die den Quellzustand, das Label und den Zielzustand aller Transitionen definiert. Bei c_{pre} handelt es sich um die Vorbedingung einer Transition, bei c_{post} um die Nachbedingung.

Die Menge aller Events (Ereignisse) E enthält alle Methodenaufrufe, die zu einer Zustandsänderung führen. Betrachtet werden folglich nur zustandändernde synchrone Call-Events. Für die Labels der Form $e[c_{pre}]/[c_{post}]$ in Λ gilt $e \in E$. Da es keinen Aktionsteil in den Transitionen gibt, ist eine Unterscheidung zwischen externen und internen (vom Zustandsdiagramm erzeugten) Events nicht nötig.

Definition: *Eingehende Transition, ausgehende Transition*
Betrachtet man den Zustand $a \in \Sigma$, nennt man die Menge aller Transitionen $t \in \Delta\, \exists\, l \in \Lambda, \exists\, b \in \Sigma \mid t = (a, l, b)$ ausgehende Transitionen von a und die Menge $t \in \Delta\, \exists\, l \in \Lambda, \exists\, b \in \Sigma \mid t = (b, l, a)$ eingehende Transitionen von a. Dabei kann $a = b$ sein, in diesem Fall wird t als Self-Transition bezeichnet.

Definition: *Auslösendes Ereignis, ausgelöste Transition*
Sei m ein Methodenaufruf und t eine Transition mit dem Label $m[c_{pre}]/[c_{post}]$, wobei c_{pre} und c_{post} beliebige OCL-Constraints seien, dann heißt m auslösendes Ereignis oder auslösende Methode von t. Alle Transitionen t, die durch ein Ereignis in Form eines Methodenaufrufs einer Methode m ausgelöst werden, werden als durch m ausgelöste Transitionen bezeichnet. Alle Zustände, in denen m eine Transition t auslöst, also alle Quellzustände der betrachteten Transitionen t, werden Vorzustände von m genannt, alle Zielzustände der betrachteten Transitionen t Nachzustände von m.

In der vorliegenden Arbeit werden folgende Zustandsdiagrammeigenschaften nicht betrachtet:

- Alle Eigenschaften von Behavioral State Machines, die nicht gleichzeitig auch Eigenschaften von Protocol State Machines sind.

- Aufgeschobene Ereignisse. Diese werden nicht betrachtet, da durch die Implementierung der Zustandsdiagramme das anliegende Ereignis immer bekannt ist und verarbeitet werden kann.

- Asynchrone Ereignisse. In der vorliegenden Arbeit werden nur synchrone Ereignisse betrachtet, bei denen jedes auftretende Ereignis zunächst vollständig verarbeitet werden muss, bevor das nächste Ereignis eintritt.

- Ports. Diese werden für Objektlebenszyklen nicht gebraucht.

8.2.2 Ausführungssemantik von Protocol State Machines

Zustandsdiagramme modellieren dynamisches Verhalten eines Systems oder eines Objekts. Neben der statischen Beschreibung ist die Definition einer Ausführungssemantik notwendig, die das dynamische Verhalten des Zustandsdiagramms und damit des modellierten Systems oder Objekts beschreibt.

Zur Beschreibung des dynamischen Verhaltens dient ein Labeled-Transition-System SC^{act} als semantische Repräsentation.
Dabei ist SC^{act} definiert als 4-Tupel wie folgt:

$$SC^{act} = (\Sigma^{act}, \Sigma^{act}_{is}, \Lambda^{act}, \Delta^{act}).$$

Zu jedem Zeitpunkt befindet sich das Zustandsdiagramm in einem Status Σ^{act}. Der Status Σ^{act} enthält zwei Komponenten $\langle C, e\rangle$, eine aktuelle Zustandskonfiguration $C \subseteq \Sigma$ und ein anliegendes Ereignis e. Dabei gilt $e \in E \cup \{\bot\}$, wobei das Symbol $\bot$ repräsentiert, daß kein Ereignis anliegt. Der initiale Status $\Sigma^{act}_{is} = \langle C_{is}, \bot\rangle$ enthält die intiale Zustandskonfiguration C_{is} und kein Ereignis. Zu Beginn des Lebenszyklus eines Objekts befindet sich das zugeordnete Zustandsdiagramm in der initialen Zustandskonfiguration C_{is}. Für diese gilt, daß der Wurzelzustand und alle Initialzustände aller aktiven Sublevelzustände aktiv sind, geschrieben $active(s)$, wobei s den aktiven Zustand repräsentiert. Die Menge der Zustände C enthält alle Zustände $s \in \Sigma$, für die gilt $active(s)$.
Für alle Zustandskonfigurationen gilt:

- Der Wurzelzustand r ist aktiv:
  $active(r)$.

- Ist ein komponierter Zustand aus der Menge Σ_{and} aktiv, so sind alle seine Unterzustände ebenfalls aktiv:
  $active(s) \wedge s \in \Sigma_{and} \Rightarrow \forall s' \mid s\chi s' : active(s')$.

- Ist ein komponierter Zustand aus der Menge Σ_{xor} aktiv, so ist genau einer seiner Unterzustände aktiv:
  $active(s) \wedge s \in \Sigma_{xor} \Rightarrow \exists_1 s' \mid s\chi s' : active(s')$.

In jeder aktiven Zustandskonfiguration liegt ein Ereignis e an, das entweder in der aktiven Zustandskonfiguration verarbeitet werden kann oder verworfen wird. Aufgeschobene Ereignisse, also Ereignisse, die vermerkt werden, bis sie verarbeitet werden können, werden in der vorliegenden Arbeit nicht betrachtet.

Es wird angenommen, daß immer genau ein Ereignis, das bekannt ist, oder kein Ereignis anliegt. Andere Arbeiten, wie z.B. [98], definieren eine Eventqueue, die alle noch nicht verarbeiteten Ereignisse speichert. In der vorliegenden Arbeit kann darauf verzichtet werden, da die technische Umsetzung der Protocol State Machines so vorgenommen wurde, daß nur das gerade aktuell anliegende Ereignis bekannt ist.

Die Transitionsrelation $\Delta^{act} \subseteq \Sigma^{act} \times \Lambda^{act} \times \Sigma^{act}$ setzt Stati aus Σ^{act} durch Labeled-Transitionen in Beziehung zueinander. Dabei gilt $\Lambda^{act} \subseteq E$.

Die Transitionen in SC korrespondieren zu den Transitionen in SC^{act}. Das anliegende Ereignis e eines Status SC^{act} bestimmt durch Anwendung eines Transitionsselektionsalgorithmus die Menge t^{act} der konfliktfreien Transitionen, die schalten können. Die Transitionen der Menge t^{act} müssen einen Zustand aus der aktuellen Zustandskonfiguration als Quellzustand besitzen und durch das anliegende Ereignis ausgelöst werden. In der Regel gibt es mehr als eine dieser Mengen t^{act}. Durch nichtdeterministische Auswahl wird anschließend eine dieser Mengen bestimmt und anhand dieser der neue Status $s \in \Sigma^{act}$ berechnet.

Ein Zustandsdiagramm ist als komplett spezifiziert zu betrachten, da es eine definierte Anfangszustandskonfiguration und zu jeder Zustandskonfiguration einen Nachzustand für ein anliegendes

Event aus der Menge E gibt. Ein Zustandsdiagramm ist nichtdeterministisch, da es mehrere Nachfolger für eine aktuelle Zustandskonfiguration und ein anliegendes Ereignis, sprich den aktuellen Status, geben kann.

8.2.3 Grundlegende Algorithmen auf Protocol State Machines

Hier werden einige grundlegende Algorithmen auf Protocol State Machines beschrieben.

Zustandsinvarianten. Die Zustandsinvariante s^{+}_{inv} eines Zustands s berechnet sich wie folgt:

$calc\_inv^{+}\ (s : \Sigma\ ) : \Gamma ==$
  if $\exists\ s' \in \Sigma \wedge s'\chi s$ **then** s_{inv} `"and"` $calc\_inv^{+}(s')$ **else** s_{inv}

Initiale Zustandskonfiguration. Für die initiale Zustandskonfiguration gilt, daß der Wurzelzustand aktiv ist. Auf dieser Basis können die restlichen Zustände ermittelt werden, die zur initalen Zustandskonfiguration gehören. Jeder aktive Zustand wird dabei mit einem Label versehen, das ihn als zur initialen Zustandskonfiguration zugehörend markiert. Ist ein Zustand s aktiv, geschrieben als $active(s)$, berechnet sich die initiale Zustandskonfiguration, beginnend beim Wurzelzustand, wie folgt:

$init\_conf\ (s : \Sigma\ ) ==$
  if $s \in \Sigma_{xor} \wedge active(s)$ **then sel** $s' \in \Sigma_{is} \mid s\chi s' : active(s'); init\_conf(s');$
  if $s \in \Sigma_{and} \wedge active(s)$ **then all** $s' \in \Sigma \mid s\chi s' : active(s'); init\_conf(s')$

Setzen einer Zustandskonfiguration. Für einige Algorithmen ist es notwendig, eine Menge von Zuständen als aktive Zustandskonfiguration zu setzen. Dabei werden alle in der gegebenen Menge enthaltenen Zustände als aktiv markiert, alle anderen Zustände aus Σ werden als nicht aktiv deklariert.

$set\_conf\ (st : Set\ \Sigma\ ) ==$
  all $s \in \Sigma$: **if** $s \in st$ **then** $active(s)$ **else** $\neg active(s)$

Oberzustand eines Zustands. Es wird der direkte Vorgängerzustand in der Hierarchie zurückgegeben.

$calc\_upperstate\ (s : \Sigma\ ) : \Sigma ==$
  sel $sup \in \Sigma \mid sup\ \chi\ s : sup$

Alle Oberzustände eines Zustands. Es werden alle Oberzustände eines Zustands in der Hierarchie und der Zustand selbst zurückgegeben.

$calc\_allupperstates\ (s : \Sigma\ ) : Set\ \Sigma ==$
  let $allsup = \emptyset$:
    all $sup \in \Sigma \mid sup\ \chi^{+}\ s : allsup \oplus sup;$
    $allsup \oplus s$

Alle Basiszustände eines Zustand. Es werden zu einem Zustand alle Basiszustände als direkte oder indirekte Unterzustände berechnet.

$$
\begin{array}{l}
\mathit{calc\_subbasicstates}\ (s : \Sigma\ ) : \mathit{Set}\ \Sigma == \\
\quad \textbf{if}\ s \in \Sigma_{simple}\ \textbf{then let}\ \mathit{states} := \emptyset : \mathit{states} \oplus s; \\
\quad \textbf{if}\ \neg s \in \Sigma_{simple}\ \textbf{then let}\ \mathit{states} := \emptyset : \\
\qquad \textbf{all}\ s' \in \Sigma \mid s \chi s' : \mathit{states} \oplus \mathit{calc\_subbasicstates}(s')
\end{array}
$$

Menge aller ausgehenden Transitionen. Dieser Algorithmus berechnet die Menge aller ausgehenden Transitionen zu einem Zustand. Dabei werden sowohl die Transitionen berücksichtigt, die direkt von diesem Zustand ausgehen, als auch die Transitionen der Oberzustände des betrachteten Zustands. Mit r wird der Wurzelzustand bezeichnet.

$$
\begin{array}{l}
\mathit{calc\_outtrans}\ (s : \Sigma\ ) : \mathit{Set}\ \Lambda == \\
\quad \textbf{let}\ \mathit{outs} := \textbf{if}\ s = r\ \textbf{then}\ \emptyset\ \textbf{else}\ \mathit{calc\_outtrans}(\mathit{calc\_upperstate}(s))\ : \\
\qquad \textbf{all}\ l \in \Lambda \mid (\exists\ \mathit{spost} \in \Sigma \mid (s, l, \mathit{spost}) \in \Delta) : \\
\qquad\quad \mathit{outs} \oplus l
\end{array}
$$

Menge aller eingehenden Transitionen. Dieser Algorithmus berechnet die Menge aller eingehenden Transitionen zu einem Zustand. Dabei werden sowohl die Transitionen berücksichtigt, die direkt in diesen Zustand enden, als auch die Transitionen der Oberzustände des betrachteten Zustands. Mit r wird der Wurzelzustand bezeichnet.

$$
\begin{array}{l}
\mathit{calc\_intrans}\ (s : \Sigma\ ) : \mathit{Set}\ \Lambda == \\
\quad \textbf{let}\ \mathit{ins} := \textbf{if}\ s = r\ \textbf{then}\ \emptyset\ \textbf{else}\ \mathit{calc\_intrans}(\mathit{calc\_upper}(s))\ : \\
\qquad \textbf{all}\ l \in \Lambda \mid (\exists\ \mathit{spre} \in \Sigma \mid (\mathit{spre}, l, s) \in \Delta) : \\
\qquad\quad \mathit{ins} \oplus l
\end{array}
$$

Transitionsselektion. Zur Transitionsselektion, also der Auswahl der potentiell schaltenden Transitionen, wird ein Suchalgorithmus im UML-Standard vorgeschlagen. Der Algorithmus nimmt die aktuelle Zustandskonfiguration und berechnet, startend von den Basiszuständen, auf Basis des aktuell anliegenden Ereignis die Mengen der potentiell schaltenden konfliktfreien Transitionen. Die Berechnung erfolgt von den innersten Basiszuständen nach außen, da innere Transitionen eine höhere Priorität besitzen als äußere. Dieser Algorithmus ist im Rahmen der vorliegenden Arbeit nicht implementiert worden, da er nicht benötigt wird. Die Transitionsselektion ist jedoch einfach durch einen Suchalgorithmus wie in [110] vorgeschlagen zu realisieren, wenn sie für künftige Erweiterungen der Testverfahren gebraucht wird.

Statt dessen ist ein Algorithmus implementiert, der auf Basis der aktuellen Zustandskonfiguration und eines anliegenden Ereignis in Form eines Methodenaufrufs m *alle* ausgehenden Transitionen berechnet. Aus dieser Menge wird nicht eine Transition selektiert, sondern alle potentiellen Nachfolgekonfigurationen berechnet und beim Test mit dem Zustand der zu testenden Implementierung verglichen.

$$
\begin{array}{l}
\mathit{calc\_allouttrans}(m : M\ )\ : \mathit{Set}\ \Lambda == \\
\quad \textbf{let}\ \mathit{trans} := \emptyset\ : \\
\qquad \textbf{all}\ s \in \Sigma \mid \mathit{active}(s) : \\
\qquad\quad \textbf{let}\ \mathit{ls} := \mathit{calc\_outtrans}(s) : \\
\qquad\qquad \textbf{all}\ l \in \mathit{ls} \mid (\exists\ c_{pre}, c_{post} \in \Gamma \mid l = m[c_{pre}]/[c_{post}]) : \\
\qquad\qquad\quad \mathit{trans} \oplus l
\end{array}
$$

Zustandsmenge zu einem Methodenaufruf. Zu jeder Methode m wird mit Hilfe dieses Algorithmus die Zustandsmenge berechnet, die die Vorbedingung für den Aufruf von m bildet. Dabei werden nur die direkten Quellzustände von m berücksichtigt, nicht deren Ober- und Unterzustände.

$$
\begin{array}{l}
calc\_prestates(m : M) : Set\ \Sigma == \\
\quad \textbf{let}\ states := \emptyset\ : \\
\qquad \textbf{all}\ l \in \Lambda \mid (\exists\, c_{pre}, c_{post} \in \Gamma \mid l = m[c_{pre}]/[c_{post}]) : \\
\qquad\quad \textbf{all}\ s \in \Sigma \mid (\exists\, s' \in \Sigma \mid (s, l, s') \in \Delta) : \\
\qquad\qquad states \oplus s
\end{array}
$$

Nachfolgende Zustandskonfiguration. Zu einer Zustandskonfiguration und einer gegebenen Transitionsmenge von schaltenden Transitionen wird die nachfolgende Zustandskonfiguration berechnet. Dabei werden alle Zustände mit ihren Oberzuständen mit Ausnahme des Wurzelzustands aus der aktiven Zustandskonfiguration entfernt, wenn sie Quellzustand einer der schaltenden Transitionen sind und alle Zustände mit ihren Oberzuständen mit Ausnahme des Wurzelzustands in die aktive Zustandskonfiguration aufgenommen, wenn sie Zielzustand einer der schaltenden Transitionen sind.

Anschließend wird für die Zustandsmenge überprüft, ob für jeden aktiven Zustand vom Typ Σ_{xor} genau ein Unterzustand aktiv ist und ob für jeden aktiven Zustand vom Typ Σ_{and} alle Unterzustände aktiv sind. Wenn nicht, wird entweder der Initialzustand oder der durch Historie gemerkte Zustand s, geschrieben $last\_conf(s)$, in die aktive Zustandskonfiguration aufgenommen.

$$
\begin{array}{l}
calc\_nextconf(t : Set\ \Lambda) == \\
\quad \textbf{all}\ s \in \Sigma \mid (\exists\, l \in t, s' \in \Sigma \mid (s, l, s') \in \Delta) : remove\_activestate(s); \\
\quad \textbf{all}\ s \in \Sigma \mid (\exists\, l \in t, s' \in \Sigma \mid (s', l, s) \in \Delta) : include\_activestate(s); \\
\quad proof\_conf(r) \\
\\
remove\_activestate(s : \Sigma) == \\
\quad \neg active(s); \\
\quad \textbf{sel}\ s' \in \Sigma \mid s' \chi s : (\textbf{if}\ s' \neq r\ \textbf{then}\ remove\_activestate(s')) \\
\\
include\_activestate(s : \Sigma) == \\
\quad active(s); \\
\quad \textbf{sel}\ s' \in \Sigma \mid s' \chi s : (\textbf{if}\ s' \neq r\ \textbf{then}\ include\_activestate(s')) \\
\\
proof\_conf\ (s : \Sigma) == \\
\quad \textbf{if}\ s \in \Sigma_{xor} \wedge active(s)\ \textbf{then} \\
\qquad (\textbf{if}\ \exists_1\ s' \in \Sigma \mid s \chi s' \wedge active(s') \\
\qquad\quad \textbf{then}\ proof\_conf(s') \\
\qquad\quad \textbf{else} \\
\qquad\qquad \textbf{all}\ s' \in \Sigma \mid s \chi s' : \neg active(s'); \\
\qquad\qquad \textbf{let}\ s' := get\_activestate(s) : proof\_conf(s')); \\
\quad \textbf{if}\ s \in \Sigma_{and} \wedge active(s)\ \textbf{then} \\
\qquad \textbf{all}\ s' \in \Sigma \mid s \chi s' : active(s'); proof\_conf(s') \\
\\
get\_activestate\ (s : \Sigma) : \Sigma == \\
\quad \textbf{if}\ \exists\ s' \in \Sigma \mid s \chi s' \wedge last\_conf(s') \\
\quad \textbf{then}\ s' \\
\quad \textbf{else sel}\ s' \in \Sigma_{is} \mid s \chi s' : s'
\end{array}
$$

Transitionssequenzen zu allen Zuständen. Dieser Algorithmus berechnet eine kürzeste Transitionssequenz zu allen Basiszuständen im Zustandsdiagramm[3]. Dabei wird das Zustandsdiagramm vom Initialzustand zu allen anderen Zuständen traversiert, bis alle Zustände mindestens einmal besucht wurden. Voraussetzung ist, daß alle Zustände erreichbar sind. Die Transitionssequenz zu einem Zustand s ist nach der Berechnung in der Sequenz $transseq(s)$ verfügbar. Die Transitionssequenz zu einer Zustandskonfiguration $sconf$ ist anschließend in der Sequenz $transseqconf(sconf)$ verfügbar.

$calc\_transseq ==$
let $nextStates := \emptyset \oplus init\_conf; unvisitStates := \Sigma$:
$transseqconf(init\_conf) = \emptyset \oplus$ `new`
loop $unvisitStates \neq \emptyset$:
all $sconf \in nextStates$:
$set\_conf(sconf)$;
all $s \in \Sigma_{simple} \mid s \in sconf$:
if $transeq(s) = \emptyset$ **then** $transeq(s) := transseqconf(sconf)$;
$unvisitStates \ominus s$;
all $t \in \Lambda \mid (\exists\, s \in actState, s' \in \Sigma \mid (s, t, s') \in \Delta)$:
$nextStates \oplus calc\_nextconf(t)$

8.3 Sequenzdiagramme

Verschiedene Ansätze definieren eine formale Semantik für Sequenzdiagramme, so z.B. [69, 107, 117]. In der vorliegenden Arbeit wird eine eigene Semantik für Sequenzdiagramme vorgeschlagen, die Ideen der anderen Arbeiten aufgreift. Die hier vorgeschlagene Semantik berücksichtigt die in der vorliegenden Arbeit verwendeten Sequenzdiagramme und läßt z.B. im Gegensatz zu anderen Arbeiten nur synchrone Events zu, negative Sequenzen werden nicht anhand eines Operators identifiziert, sondern durch Zuordnung zu einer Menge.

Zunächst erfolgt eine formale Beschreibung von Sequenzdiagrammen und anschließend werden grundlegende Algorithmen auf Sequenzdiagrammen definiert.

8.3.1 Formale Beschreibung von Sequenzdiagrammen

Definition: *Sequenzdiagramm*
Ein Sequenzdiagramm läßt sich formal wie folgt als 4-Tupel beschreiben:
$SD = (L, N, O, E)$.

Die Menge der Lebenslinien L repräsentiert die Lebenslinien aller an einem Szenario beteiligten Objekte. Die Menge N enthält alle Nachrichten, die zwischen den Objekten im Sequenzdiagramm ausgetauscht werden. Dabei ist $N = L \times E \times L$ eine Relation zwischen den Lebenslinien, von denen eine Nachricht gesendet bzw. von denen sie empfangen wird, und dem Nachrichtenlabel. Ein Nachrichtenlabel $m \in E$ hat die Form $m = e(para)$, wobei e ein Ereignis in Form eines Methodenaufrufs ist (analog zu der Definition der Protocol State Machine) und $para$ die aktuelle Belegung der Methodenparameter.

$O = N \times N$ beschreibt die Ordnung auf den Nachrichten. Eine Nachricht n_1 tritt vor einer Nachricht n_2 in einer Sequenz auf, genau dann, wenn $(n_1, n_2) \in O$. Es gilt:

$(n_1, n_2) \in O \wedge (n_2, n_1) \in O \Rightarrow n_1 = n_2$, d.h. eine Nachricht ist nicht gleichzeitig vor einer anderen Nachricht und nach der anderen Nachricht erlaubt.

Eine Nachricht n_2 ist genau dann die direkte Folgenachricht einer Nachricht n_1, wenn gilt:
$\nexists\, n_3 : N \mid (n_1, n_3) \in O \wedge (n_3, n_2) \in O$.

[3] Wenn es mehrere kürzeste Transitionssequenzen gibt, wird nur eine berechnet.

Es werden nur synchrone Events betrachtet, daß heißt Events, die in strenger Reihenfolge nacheinander abgearbeitet werden. Der Fall, daß ein zweites Event verschickt wird, bevor das erste empfangen wurde, tritt nicht ein. Der Interaktionsoperator `seq` ist folglich nicht zulässig.

Definition: *Negative und positive Sequenzen*
Um negative und positive Sequenzdiagramme zu unterscheiden, wird jedes Sequenzdiagramm einer der zwei Mengen SD_{pos} und SD_{neg} zugeordnet, wobei $SD_{pos} \cap SD_{neg} = \emptyset$. Damit wird der Negationsoperator ausschließlich auf komplette Sequenzen angewendet, d.h. eine gesamte Sequenz und nicht nur Teile als ungültig deklariert. Die Anwendung von Negationen auf komplette Sequenzen wird auch in [107] vorgeschlagen, dort in Form eines Attributs für Sequenzdiagramme, das den Operator `neg` in der UML 2.0 ersetzen soll.

Alle anderen Operatoren der Sequenzdiagramme werden in der vorliegenden Arbeit nicht betrachtet. Viele komplexe Sequenzdiagramme lassen sich jedoch in einfache Sequenzdiagramme umformen (siehe Kapitel 4.2.4).

8.3.2 Grundlegende Algorithmen auf Sequenzdiagrammen

Auf Sequenzdiagrammen wird in der vorliegenden Arbeit nur ein grundlegender Algorithmus benötigt, die Berechnung der Sequenz der eingehenden Nachrichten in Form eines Methodenaufrufs an einer Lebenslinie. Der Vollständigkeit halber ist zudem der Algorithmus zur Berechnung aller ausgehenden Nachrichten von einer Lebenslinie gegeben.

$$
\begin{array}{l}
\mathit{calc\_inmessages}\,(o : L\,) : \mathit{Seq}\,N == \\
\quad \textbf{let}\; \mathit{mess} := \emptyset\; : \\
\qquad \textbf{all}\; m \in E \mid (\exists\, \mathit{oout} \in L \mid (\mathit{oout}, m, o) \in N) : \\
\qquad\quad \mathit{mess} \oplus m
\end{array}
$$

$$
\begin{array}{l}
\mathit{calc\_outmessages}\,(o : L\,) : \mathit{Seq}\,N == \\
\quad \textbf{let}\; \mathit{mess} := \emptyset\; : \\
\qquad \textbf{all}\; m \in E \mid (\exists\, \mathit{oin} \in L \mid (o, m, \mathit{oin}) \in N) : \\
\qquad\quad \mathit{mess} \oplus m
\end{array}
$$

8.4 Klassen und Verträge

Abschließend sollen Klassen und die ihnen zugeordneten Spezifikationen betrachtet werden. Dazu gehören in erster Linie Invarianten, Vor- und Nachbedingungen. Jeder Klasse ist zudem maximal ein Zustandsdiagramm zugeordnet.

8.4.1 Formale Beschreibung von Klassen

Definition: *Klasse*
Eine Klasse C wird in der vorliegenden Arbeit wie folgt formal als Binärtupel definiert:
$c = (A, M)$

Die Menge A enthält alle Attribute a der Klasse c. Referenziert wird auf den Typ eines Attributs $a \in A$ mittels $\mathit{type}(a)$ und auf den Namen mittels $\mathit{name}(a)$.

Die Menge M enthält alle Methoden m der Klasse c. Analog zu den Attributen erhält man durch $\mathit{name}(m)$ den Namen der Methode, durch $\mathit{type}(m)$ den Typ des Rückgabewerts und durch $\mathit{para}(m)$ eine Sequenz der formalen Parameter mit Name und Typ.

Den Namen der Klasse C erhält man durch $name(c)$. Die Klasseninvariante einer Klasse c wird notiert als $INV(c)$. Die Vorbedingung einer Methode m ist $PRE(m)$, die Nachbedingung $POST(m)$. Invarianten, Vor- und Nachbedingung liegen in OCL vor und sind folglich vom Typ Γ.

8.4.2 Grundlegende Algorithmen auf Klassen

Auf Klassen wird in der vorliegenden Arbeit nur ein grundlegender Algorithmus benötigt, die Berechnung der Query- und Update-Methoden aus dem Zustandsdiagramm. Zur Erinnerung: Alle im Zustandsdiagramm referenzierten Methoden sind Update-Methoden, alle nicht im Zustandsdiagramm referenzierten Methoden sind Query-Methoden.

Jede Methode m wird mit einem Label versehen, geschrieben $isQuery(m)$, wenn es sich um eine Query-Methode handelt, $isUpdate(m)$ sonst.

$init\_methods ==$
$\quad$ **all** $m \in M$: **if** $name(m) \in E$ **then** $isUpdate(m)$ **else** $isQuery(m)$

Kapitel 9

Testfälle und Testdaten aus UML-Modellen

Die Analyse in Kapitel 6 zeigt, daß sich insbesondere Sequenzdiagramme gut zur Generierung von Testfällen eignen, da sie typische Normal- und Fehlerfälle darstellen. Jedes Sequenzdiagramm kann somit als ein Testfall betrachtet werden. Gleichzeitig modellieren sie in der Regel nur einen Teilausschnitt des Systemverhaltens und erlauben somit keine Aussage über das Verhalten des Gesamtsystems.

Es müssen zur Ergänzung Diagrammtypen gefunden werden, aus denen die fehlende Information gewonnen wird. In der vorliegenden Arbeit werden zur Ergänzung der Sequenzdiagramme Zustandsdiagramme herangezogen, die Informationen zur Initialisierung der beteiligten Objekte liefern.

Ein Sequenzdiagramm definiert folglich nicht nur einen Testfall, sondern eine Menge von Testfällen, die sich dadurch unterscheiden, daß die beteiligten Objekte unterschiedlich initialisiert wurden.

Zur Testdatengewinnung stellt die vorliegende Arbeit keine umfassende Lösung zur Verfügung. Testdaten müssen im Testsystem UT^3 in den meisten Fällen manuell zur Verfügung gestellt werden. Es gibt jedoch für einfache Datentypen bereits eine Lösung, sofern es sich um Methodenparameter handelt, über die in der Vorbedingung der Methode eine Aussage über den betrachteten Definitionsbereich getroffen wurde.

In diesem Kapitel wird zunächst in Abschnitt 9.1 die Testfallgenerierung auf der Basis von Sequenzdiagrammen und Zustandsdiagrammen vorgestellt. Anschließend folgt in Abschnitt 9.2 ein Einblick in die Erzeugung von Testdaten.

9.1 Testfallgenerierung

Das in der vorliegenden Arbeit vorgestellte Verfahren zur Generierung von Testfällen erzeugt Testfälle in Form von Methodensequenzen, die an ein Objekt oder eine Kollaboration von Objekten geschickt werden. Deshalb wird in der vorliegenden Arbeit unter dem Begriff Testfall immer eine Methodensequenz verstanden (ähnlich wie in [97]).

Definition: *Testfall*
Ein Testfall ist eine Methodensequenz, die an ein oder mehrere Objekte geschickt wird. Ein Testfall beschreibt nicht, wie diese Objekte zueinander in Beziehung stehen. Aktuelle Parameter von Methoden der Methodensequenz werden ausschließlich abstrakt beschrieben.

Wird ein Testfall auf ein einzelnes Objekt angewendet, so spricht man von einem Klassentest. Beim Klassentest werden meist alle anderen beteiligten Objekte durch Stubs oder Mock-Objekte

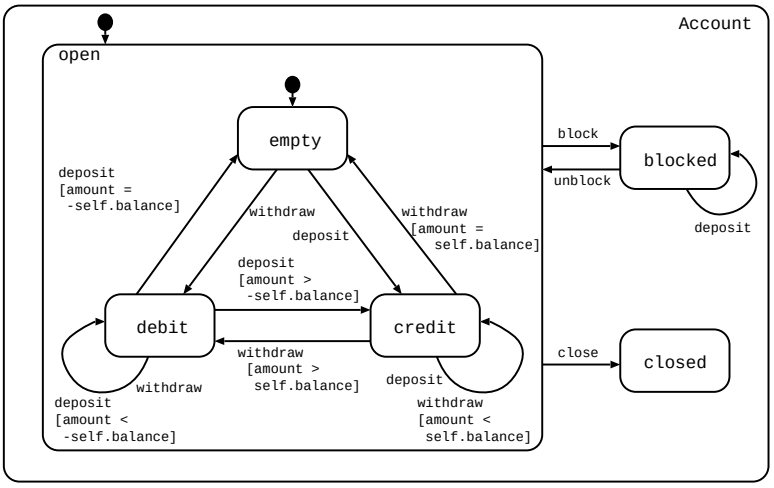


Abbildung 9.1: Beispiel: Zustandsdiagramm

ersetzt. Wenn ein Testfall auf eine Kollaboration von Objekten angewendet wird, handelt es sich um einen Integrationstest, da das Zusammenspiel der Objekte überprüft wird. Sowohl Integrationstest als auch Klassentest werden durch das hier vorgestellte Verfahren unterstützt. Voraussetzung der Anwendung der vorgestellten Verfahren ist die Konsistenz der verwendeten Modelle.

Die erzeugten Testfälle lassen sich in reguläre und komplementäre Testfälle unterscheiden. Dadurch wird sowohl der Test des funktionalen Verhaltens als auch der Test auf Robustheit und Fehlertoleranz betrachtet.

Definition: *Regulärer Testfall*
Ein regulärer Testfall ist ein Testfall, deren Ausführung laut Spezifikation erlaubt ist. Bei regulären Testfällen befinden sich alle beteiligten Objekte zu jedem Zeitpunkt der Ausführung der Sequenz in Zuständen, in denen ein Aufruf der nächsten Methode der Sequenz erlaubt ist.

Reguläre Testfälle dienen der Überprüfung des funktionalen Verhaltens eines Objekts oder einer Kollaboration von Objekten bzw. des Fehlerverhaltens und damit der Robustheit im Falle einer negativen Sequenz.

Definition: *Komplementärer Testfall*
Ein komplementärer Testfall ist ein Testfall, bei dem sich zu mindestens zu einem Zeitpunkt der Ausführung mindestens ein beteiligtes Objekt in einem Zustand befindet, in dem der Aufruf der nächsten Methode der Sequenz zu einer Verletzung der Vorbedingung führt.

Komplementäre Testfälle dienen zur Überprüfung der Robustheit eines Objekts oder einer Kollaboration von Objekten.

Es scheint zunächst ein Widerspruch in dem Begriff des regulären Testfalls auf Basis einer negativen Sequenz zu liegen. Dem ist jedoch nicht so. Wenn ein Szenario negativ modelliert ist, so darf das System oder die Kollaboration von Objekten auf einen Stimulus nicht mit dieser Sequenz reagieren. Dies ist unabhängig davon, in welchem Zustand sich die beteiligten Objekte befinden. Um die zwei Fälle zu unterscheiden, werden die Begriffe *regulärer positiver Testfall* und *regulärer negativer Testfall* bzw. *komplementärer positiver Testfall* und *komplementärer negativer Testfall* verwendet.

Da bei negativen Sequenzen immer das gesamte Szenario betrachtet werden muß, werden für den Klassentest ausschließlich positive Sequenzen verwendet und nur für den Integrationstest so-

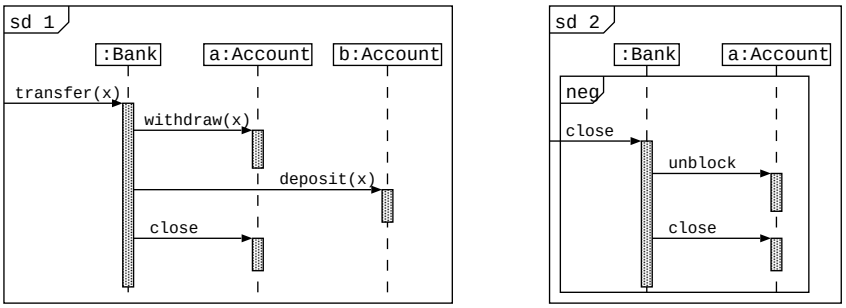


Abbildung 9.2: Positives und negatives Sequenzdiagramm

wohl positive als auch negative Sequenzen[1].

In den folgenden beiden Abschnitten wird die Generierung der regulären und der komplementären Testfälle erläutert. Die Technik wird jeweils an einem Beispiel veranschaulicht. In Abbildung 9.1 ist ein Zustandsdiagramm für die Klasse `Account` gezeigt. In Abbildung 9.2 sind zwei Sequenzdiagramme gegeben. Das Sequenzdiagramm `sd 1` beschreibt ein positives Szenario, das Sequenzdiagramm `sd 2` ein negatives Szenario. Beide Szenarien wurden bereits in Kapitel 7.2 erläutert.

9.1.1 Reguläre Testfälle

Reguläre Testfälle beschreiben eine Methodensequenz, die keine Vorbedingung verletzt. Die Informationen aus einem Sequenzdiagramm, das ja bereits Methodensequenzen beschreibt, ist noch nicht ausreichend, da es weder Aussagen macht über die Zustände der beteiligten Objekte noch zu welchem Zeitpunkt das beschriebene Szenario ausgeführt wird. Zusätzliche Informationen liefern bei der in der vorliegenden Arbeit verwendeten Technik die Zustandsdiagramme der beteiligten Objekte. Als Basis ist die Verwendung des ursprünglich spezifizierten oder des um OCL-Constraints angereicherten Zustandsdiagramms aus Kapitel 10.1 möglich.

Für den Klassentest wird jede der beteiligten Klassen einzeln betrachtet, zum Beispiel die Klasse `Account`. Da zum Klassentest ausschließlich positive Sequenzen verwendet werden, wird hier nur das Sequenzdiagramm `sd 1` aus Abbildung 9.2 betrachtet. Es lassen sich zwei Objekte der Klasse `Account` identifizieren, die am Szenario `sd 1` beteiligt sind, das Objekt `a` und das Objekt `b`.

Als Beispiel soll hier das Objekt `a` dienen. Da auf dem Objekt `b` nur die Methode `deposit` aufgerufen wird, lohnt eine Betrachtung nicht[2].

Nun werden alle Sequenzen von Methoden an das Objekt `a` aus dem Sequenzdiagramm extrahiert. Im Beispiel ist dies die Sequenz `a.withdraw(x); a.close`[3]. Ein Aufruf dieser Sequenz auf dem Objekt `a` ist nur in bestimmten Zuständen von `a` möglich. Zur Identifizierung der Zustände dient das Zustandsdiagramm der Klasse `Account`, das den Objektlebenszyklus eines Objekts des Typs `Account` spezifiziert.

Beim Integrationstest wird versucht, die modellierten Szenarien zu provozieren. Dabei wird die die Sequenz auslösende Nachricht vom Testtreiber an das erste zu testende Objekt der Sequenz

[1] Beim Klassentest wird nur ein Objekt betrachtet. Da es sich bei der Interaktion im Sequenzdiagramm üblicherweise um Kommunikation zwischen mehreren Objekten handelt, fällt dies unter den Begriff Integrationstest. Unerheblich ist hierbei, daß alle beteiligten Objekte denselben Typ besitzen können.

[2] Testfälle für den Klassentest können in UT^3 auf Objekte beschränkt werden, die in einem Szenario mehr als eine Nachricht empfangen.

[3] Der Wert des Parameters `x` ist nicht weiter spezifiziert, außer, daß es sich um den gleichen Wert wie im Aufruf `transfer` auf der Bank und `deposit` auf dem Objekt `b` handelt. Parameter in Sequenzen werden im Testsystem UT^3 vorerst nicht betrachtet.

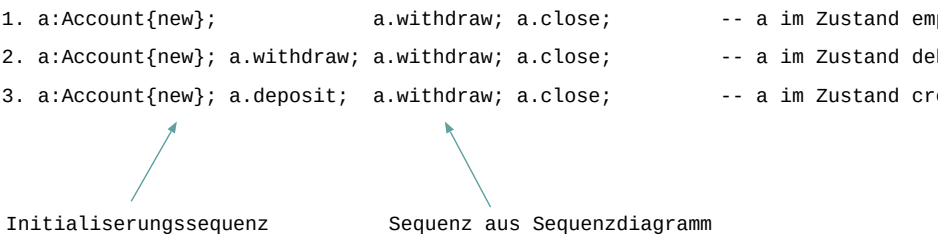


Abbildung 9.3: Reguläre (positive) Testfälle für den Klassentest von `Account`

geschickt. Im Beispiel in Abbildung 9.2 ist es im positiven Sequenzdiagramm `sd 1` die Nachricht `transfer(x)` an das unbenannte Objekt vom Typ `Bank` und im negativen Sequenzdiagramm `sd 2` die Nachricht `close` an das unbenannte Objekt vom Typ `Bank`. Das unbenannte Objekt vom Typ Bank muß zunächst benannt werden, hier bekommt es den Namen `z`.

Sowohl für den Klassen- als auch den Integrationstest erfolgt die Initialisierung der beteiligten Objekte in gleicher Weise. Es werden durch Analyse des Zustandsdiagramms Zustände gesucht, in denen ein bestimmtes Szenario ausführbar ist.
Dabei gelten folgende Regeln in Anlehnung an die Definition der Protocol State Machines in Kapitel 8.2:

- Handelt es sich um eine Query-Methode, so ist diese in jedem Zustand aufrufbar und damit ausführbar.
- Update-Methoden sind in einem Zustand nur dann ausführbar, wenn sie von einer Transition referenziert werden, die diesen Zustand als Ausgangszustand besitzt.
- Bei einer Folge von Nachrichten an ein Objekt muß die erste Nachricht in einem Zustand ausführbar sein und in einen Zustand führen, in dem die zweite Nachricht ausführbar ist. Dies ist rekursiv für alle Nachrichten der Nachrichtenfolge anzuwenden.

Für das Szenario `sd 1` aus Abbildung 9.2 ist es nötig, daß das `Account`-Objekt `a` sich in einem Zustand befindet, in dem die Methodenfolge `withdraw` gefolgt von `close` ausgeführt werden kann. Dies gilt analog für das `Account`-Objekt `b` und die Methode `deposit`.

Aus dem Zustandsdiagramm für die Klasse `Account` in Abbildung 9.1 läßt sich ermitteln, daß sich das Objekt `a` im Zustand `open` und damit in einem der Unterzustände von `open` befinden muß, das Objekt `b` in einem der Zustände `open` oder `blocked`.

Damit gibt es zwölf mögliche reguläre Testfälle für das Szenario `sd 1` im Falle des Integrationstests: jeweils einen für jeden möglichen gültigen Zustand des Objekts `a` (3 Zustände) kombiniert mit jedem möglichen gültigen Zustand des Objekts `b` (4 Zustände)[4].

Im Falle des Klassentests gibt es drei reguläre Testfälle, da sich das Objekt `a` in einem der drei Unterzustände von `open` befinden muß.

Um nun die Testsequenzen der Testfälle zu bestimmen, ist es nötig, aus dem Zustandsdiagramm eine Sequenz abzuleiten, die die an einem Szenario beteiligten Objekte in die im Testfall geforderten Zustände versetzt. Dazu wird der in Kapitel 8.2 beschriebene Algorithmus zur Berechnung der Transitionssequenzen zu einem Zustand genutzt. Bedingungen an den Transitionen im Zustandsdiagramm werden im Testfall vermerkt (Beispiel: `a.withdraw[amount>0]`) und dienen der leichteren Bestimmung von Testdaten.

Die regulären Testfälle mit den zugehörigen Testsequenzen für den Klassentest auf der Basis des positiven Sequenzdiagramms zeigt Abbildung 9.3, die regulären Testfälle mit ihrem Testsequenzen für den Integrationstest zeigt Abbildung 9.4, wobei die möglichen Kombinationen der Initialisierung der Objekte `a` und `b` tabellarisch dargestellt sind.

[4]Der Übersichtlichkeit halber wurde von den Zuständen der Bank abstrahiert.

	a:Account{new};	a:Account{new}; a.withdraw;	a:Account{new}; a.deposit;
b:Account{new};	***Init1*** a im Zustand empty b im Zustand empty	***Init2*** a im Zustand debit b im Zustand empty	***Init3*** a im Zustand credit b im Zustand empty
b:Account{new}; b.withdraw;	***Init4*** a im Zustand empty b im Zustand debit	***Init5*** a im Zustand debit b im Zustand debit	***Init6*** a im Zustand credit b im Zustand debit
b:Account{new}; b.deposit;	***Init7*** a im Zustand empty b im Zustand credit	***Init8*** a im Zustand debit b im Zustand credit	***Init9*** a im Zustand credit b im Zustand credit
b:Account{new}; b.block	***Init10*** a im Zustand empty b im Zustand blocked	***Init11*** a im Zustand debit b im Zustand blocked	***Init12*** a im Zustand credit b im Zustand blocked

```
Testsequenz TFn.        z:Bank{new}; Initn; z.transfer;
```

Abbildung 9.4: Reguläre positive Testfälle für den Integrationstest

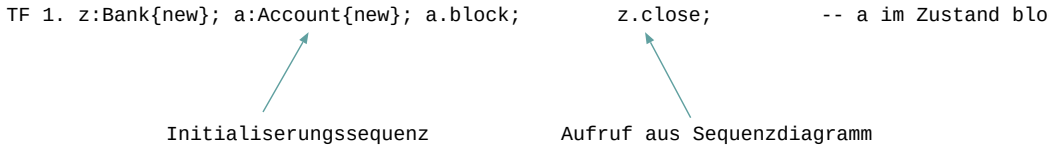


Abbildung 9.5: Regulärer negativer Testfall für den Integrationstest

Um auch negative Sequenzen zu berücksichtigen, werden weitere Testfälle generiert, jedoch ausschließlich für den Integrationtest. Reagiert das System auf die Stimulation mit einer negativen Sequenz, so ist der Test fehlgeschlagen.

Dazu sind analog zu der beschriebenen Technik für positive Sequenzen alle beteiligten Objekte in einen Zustand zu bringen, in dem die negative Sequenz (potientiell) ausgeführt werden kann. Im Beispiel ist dieses für das Objekt `a` im Zustand `blocked` der Fall. Den regulären negativen Testfall mit der dazugehörigen Testsequenz zeigt Abbildung 9.5.

Zu berücksichtigen ist, daß eine korrekte Initialisierung der beteiligten Objekte keine Garantie darstellt, daß das System sich verhält, wie die Modellierung der Sequenzdiagramme es vorsieht. Dies ist nur dann der Fall, wenn die Sequenzdiagramme das System vollständig modellieren. Das Testorakel, vorgestellt in Kapitel 10, ist bisher unabhängig von der tatsächlichen Sequenz, mit der das System auf die Stimulation reagiert. Eine Erweiterung des Testorakels ist geplant, so daß auch die Sequenz einbezogen wird, mit der das System reagiert[5].

Der Algorithmus berechnet zunächst auf der Basis der eingehenden Nachrichten an einer Lebenslinie alle (Basis-) Zustände, in denen sich das System befinden kann, ohne die Vorbedingung zu verletzen. Anschließend wird die Transitionssequenz zu diesen Zuständen berechnet.

[5]Betrachtet man das Testmodell aus Kapitel 3, so deckt der Test bisher nur Sollzustand und Sollausgaben ab, nicht jedoch Sollverhalten.

$calc\_initmessages\ (o : L\ ) : Set\ Seq\ \Lambda ==$
  let $messages := (\mathbf{all}\ mes \in calc\_inmessages(o) \mid isUpdate(mes) : mes)$:
    sel $m \in messages$:
      let $states := calc\_prestates(m); initSeq := \emptyset, initStates := \emptyset$:
        all $s \in states$: **if** $\neg is\_pre(m, calc\_allupperstates(s))$ **then** $states \ominus s$;
        all $s \in states$: $initStates \oplus calc\_subbasicstates(s)$;
        all $s \in initStates$: $initSeq \oplus transseq(s)$;

$is\_pre\ (m : Set\ M, s : Set\ \Sigma\ ) : bool ==$
  $set\_conf(s)$
  if $m = \emptyset$
    then `true`
    else sel $actm \in m$:
      if $\neg actm \in calc\_outtrans(s)$
        then `false`
        else
          let $states := calc\_nextconf(actm); erg :=$ `false` :
            all $s \in states$: $erg \vee is\_pre(m \ominus actm, s)$

9.1.2 Komplementäre Testfälle

Neben den regulären Testfällen ist die Erzeugung weiterer Testfälle, komplementärer Testfälle, mit dem Testsystem möglich, um das System auf Robustheit zu testen.

Komplementäre Testfälle berücksichtigen alle Zustände, in denen eine Sequenz die implizite Vorbedingung aus dem Zustandsdiagramm verletzt. Das sind alle Zustände, die die beteiligten Objekte nicht in den regulären Testfällen annehmen. So sind dies z.B. im Sequenzdiagramm `sd 1` für das Objekt `a` die Zustände `blocked` und `closed`, für das Objekt `b` der Zustand `closed`.

Die Erzeugung der komplementären Testfälle führt schnell zu einer Zustandsexplosion, da in der Regel viele Zustände zu berücksichtigen sind. Deshalb werden komplementäre Testfälle nur optional generiert.

Zwei Arten der Kombination der Zustände sind im Integrationstest möglich:

1. Ein beteiligtes Objekt befindet sich in einem Zustand, in dem die Sequenz die Vorbedingung der aktuell aufgerufenen Methode verletzt, also ein in Bezug auf die Sequenz ungültiger Zustand. Alle anderen beteiligten Objekte nehmen in Bezug auf die Sequenz gültige Zustände an.

2. Alle Objekte befinden sich, soweit möglich, in Zuständen, in denen die Sequenz bei mindestens einem Aufruf einer Methode die Vorbedingung verletzt.

Beide Kombinationen führen zu einer großen Anzahl von Zuständen. Komplementäre Testfälle eignen sich deshalb besser für den Klassentest, bei dem nur die Zustände eines einzelnen Objekts betrachtet werden. Beim Integrationstest wird empfohlen, sich auf einige wenige in Bezug auf die Sequenz ungültige Zustände eines Objekts zu konzentrieren und alle anderen Objekte gleichzeitig in nur einem in Bezug auf die Sequenz gültigen Zustand zu testen. Es handelt sich also um eine reduzierte Variante der Kombinationsmöglichkeit 1.

Diese Vorgehensweise - die Betrachtung nur einiger weniger Fehlerfälle und die Kombination mit gültigen Werten der anderen beteiligten Objekte bzw. Variablen - ist eine übliche Vorgehensweise beim Test zur Reduzierung des Testaufwands und wird u.a. beim Verfahren des Domaintests [10] angewendet (siehe auch Kapitel 3.3).

Typ	Ausgewertete Operationen	Erzeugte Werte
Boolean		true, false
Numerisch (Integer, Real)	<, >, <=, >=, =, <>	Grenzwert (x), Wert > x, Wert < x

Abbildung 9.6: Ausgewertete Operationen über Basistypen

9.2 Testdatenerzeugung

Für die Erzeugung von Testdaten sind verschiedene Ansätze denkbar, z.B. die Erzeugung durch einen Zufallsalgorithmus, durch Äquivalenzklassenbildung oder durch Grenzwertanalyse. Die vorliegende Arbeit kombiniert die oben genannten Varianten. Zur Erzeugung von Testdaten werden Äquivalenzklassen aus den spezifizierten Bedingungen in OCL abgeleitet. Zu jeder Bedingung wird der Grenzwert als Testdatum gewählt und weitere Werte mit Hilfe eines Zufallszahlengenerators für jede Äquivalenzklasse erzeugt.

Im Rahmen des Testsystems UT^3 werden bisher nur OCL-Constraints für die numerischen und boolschen Basistypen ausgewertet. Dabei werden nur einfache Bedingungen ausgewertet. Eine Übersicht zeigt die Tabelle in Abbildung 9.6.

Der Algorithmus erzeugt für einfache unabhängige Bedingungen auf numerischen Werten vom Typ `int` oder `real`, z.B. $amount > 0$, drei Werte:

1. Wert direkt auf der Grenze
   Beispiel: $amount = 0$
2. Wert unterhalb der Grenze durch Verwendung eines Zufallsalgorithmus
   Beispiel: $amount = -150$
3. Wert oberhalb der Grenze durch Verwendung eines Zufallsalgorithmus
   Beispiel: $amount = 2060$

Eine Bedingung ist *einfach*, wenn einer der in Abbildung 9.6 gegebenen Operatoren verwendet wird, und ist *unabhängig*, wenn es keinen anderen Teilterm gibt, indem die numerische Variable verwendet wird. So lassen sich z.B. aus der Bedingung $amount > 0\ and\ self.balance > amount$ bisher keine Testdaten für die Variablen $amount$ und $self.balance$ generieren.

Für boolsche Variablen werden beide möglichen Werte, $true$ und $false$, erzeugt.

Sei p ein einfaches Prädikat in OCL, dann werden Testdaten wie folgt erzeugt, wobei die Funktion `ocl_parse_praed` das Prädikat (numerisch oder boolsch) in Variable und Wert zerlegt, die Funktion *get_random* Zufallszahlen erzeugt und *MIN_NUM* und *MAX_NUM* den kleinst- bzw. größtmöglichen Wert einer numerischen Variablen darstellen:

$calc\_tdata\ (p : \Gamma) : Set\ ==$
  let $(x, val) := ocl\_parse\_praed(p), s := \emptyset$:
    if $type(x) =$ `bool`
    then $s \oplus \{$`true`, `false`$\}$
    else $s \oplus \{val, get\_random(MIN\_NUM, val), get\_random(val, MAX\_NUM)\}$

Das beschriebene Verfahren wurde zuerst im Rahmen der Arbeit in [26] implementiert und anschließend in das Testsytem UT^3 integriert. In [26] werden zusätzlich Objekte mit Hilfe von Standardwerten erzeugt, dieses Verfahren wurde jedoch nicht in UT^3 übernommen, da hier der Einsatz von Objektdiagrammen und einer Variante des in [12] vorgestellten Verfahrens auf der Basis von Invarianten zur Erzeugung von Testdaten geplant ist.

Kapitel 10

UML-basierte Testorakel

Aus den Überlegungen in Kapitel 6 ergibt sich, daß Zustandsdiagramme und OCL-Constraints eine gute Basis für das Testorakel bilden.

Zustandsdiagramme, speziell Protocol State Machines, spezifizieren Objektlebenszyklen. Sie definieren erlaubte Zustände und Zustandsübergänge. Der Aufruf einer Update-Methode verletzt die implizite Vorbedingung, wenn er im aktuellen Zustand keine Transition auslöst. Der Folgezustand einer Transition definiert eine implizite Nachbedingung. Zusätzliche Vor- und Nachbedingungen werden explizit in OCL spezifiziert. Implizite Vor- und Nachbedingungen aus dem Zustandsdiagramm und explizite Vor- und Nachbedingungen in OCL müssen laut UML-Spezifikation gleichzeitig gelten.

Vor- und Nachbedingungen aus den Zustandsdiagrammen und den OCL-Constraints werden im vorgestellten Ansatz miteinander kombiniert. In der vorliegenden Arbeit werden zwei unterschiedliche Arten der Kombination betrachtet.

1. Klasseninvarianten, Vor-, und Nachbedingungen in OCL werden in Zustandsdiagramme integriert. Die dadurch entstandenen Zustandsdiagramme werden als angereicherte Zustandsdiagramme bezeichnet.

2. Aus den Zustandsdiagrammen werden Vor- und Nachbedingungen extrahiert. Die extrahierten Vor- und Nachbedingungen liegen ebenfalls in OCL vor. Die ursprünglichen Vor- und Nachbedingungen in OCL werden durch die aus den Zustandsdiagrammen extrahierten Vor- und Nachbedingungen ergänzt. Die so erzeugten OCL-Constraints werden in der vorliegenden Arbeit als erweiterte OCL-Constraints bezeichnet.

Beide Varianten werden in den folgenden beiden Abschnitten vorgestellt. Dabei wird zunächst anhand eines Beispiels die Grundidee präsentiert.

Als Beispiel werden die Spezifikationen für die Klasse `SavingsAccount` in den Abbildungen 10.1, 10.2 und 10.3 und die Spezifikationen für die Klasse `Account` in den Abbildungen 10.4 und 10.5 dienen.

Abbildung 10.1 zeigt ein Zustandsdiagramm für die Klasse `SavingsAccount`. Es gibt die Zustände `empty` und `credit` als Unterzustände des Zustands `payments` und die Zustände `blocked` und `unblocked` als Unterzustände des Zustands `blocking`. Die Zustände `payments` und `blocking` sind parallel zueinander im Zustand `open` definiert. Dieser wird verlassen durch einen Aufruf der Methode `closed`, der in den Zustand `closed` führt. Transitionen werden durch eine Reihe von Methoden ausgelöst. Für das Beispiel wichtig ist die Methode `withdraw`, die im Zustand `credit` eine Transition auslöst unter der Bedingung, daß das Konto nicht gesperrt ist.

Abbildung 10.4 zeigt ein Zustandsdiagramm für die Klasse `Account`. Dieses ist ähnlich dem Zustandsdiagramm für die Klasse `SavingsAccount`, besitzt jedoch einen zusätzlichen Unterzustand `debit` im Zustand `payments`, da normale Konten überzogen werden können. Die Methode `withdraw` ist deshalb in allen Unterzuständen des Zustands `payments` aufrufbar.

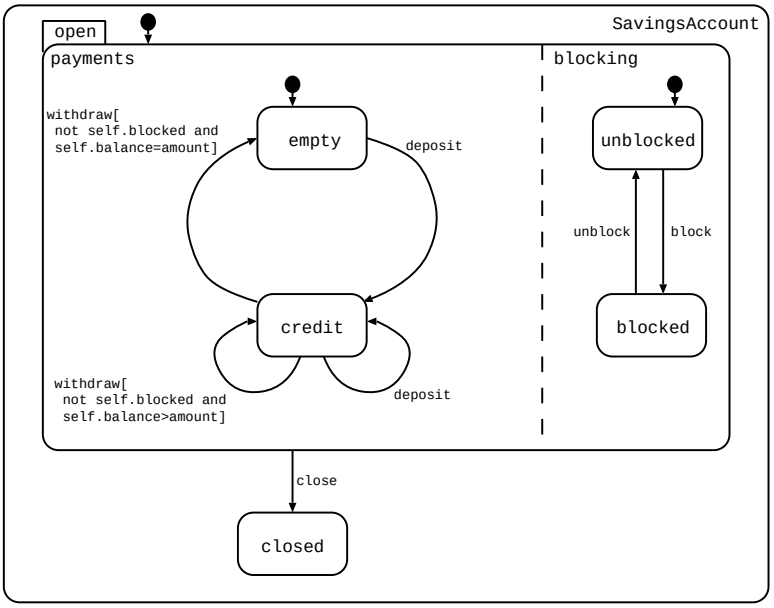


Abbildung 10.1: Zustandsdiagramm der Klasse `SavingsAccount`

```
context SavingsAccount inv:
      self.balance >= 0
```

Abbildung 10.2: Klasseninvariante für `SavingsAccount`

```
context SavingsAccount::withdraw (amount:int)
   pre : amount > 0 and self.balance >= amount
   post: self.balance = self.balance@pre - amount


context SavingsAccount::deposit (amount:int)
   pre : amount > 0
   post: self.balance = self.balance@pre + amount
```

Abbildung 10.3: Vor- und Nachbedingung für `withdraw` (oben) und `deposit` (unten) der Klasse `SavingsAccount`

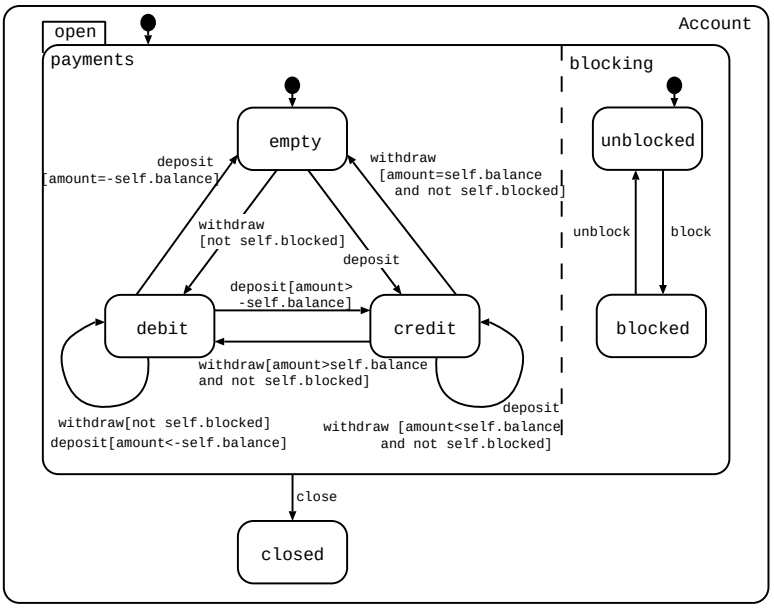


Abbildung 10.4: Zustandsdiagramm der Klasse `Account`

```
context Account::withdraw (amount:int)
   pre : amount > 0
   post: self.balance = self.balance@pre - amount
```

Abbildung 10.5: Vor- und Nachbedingung für `withdraw` der Klasse `Account`

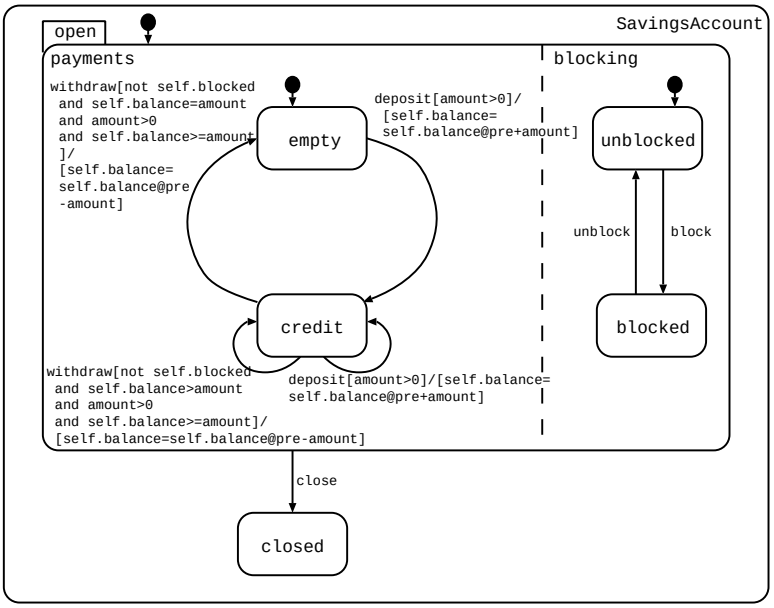


Abbildung 10.6: Um Vor- und Nachbedingungen angereichertes Zustandsdiagramm der Klasse `SavingsAccount`

In Abbildung 10.2 findet sich die Spezifikation der Klasseninvarianten der Klasse `SavingsAccount`. Diese besagt, daß der Kontostand eines Sparkontos immer positiv oder gleich Null sein muß. Die Invariante der Klasse `Account` ist hier nicht aufgeführt, da sie `true` ist.

Als Beispiel für die Spezifikation von Methodenverträgen dienen die Vor- und Nachbedingungen der Methoden `withdraw` und `deposit` der Klasse `SavingsAccount`, gegeben in Abbildung 10.3 und die Vor- und Nachbedingung der Methode `withdraw` der Klasse `Account` in Abbildung 10.5. Im Wesentlichen spezifizieren diese, daß der Kontostand nach Ausführung der Methode `withdraw` um den gegebenen Betrag reduziert ist und nach Ausführung der Methode `deposit` analog dazu entsprechend erhöht ist.

Nach der Erläuterung der Techniken an den Beispielen werden die jeweiligen Algorithmen beschrieben.

10.1 Angereicherte Zustandsdiagramme

Die erste Variante der Kombination reichert Zustandsdiagramme um Vor- und Nachbedingungen sowie Klasseninvarianten an.

- Da beide Vorbedingungen respektive beide Nachbedingungen für die Methode `withdraw` gelten müssen, werden Vor- und Nachbedingungen mit einem logischen `und` verknüpft.

  Als Beispiel wählen wir eine beliebige Transition aus dem Zustandsdiagramm in Abbildung 10.1, die von `withdraw` ausgelöst wird. In unserem Fall nehmen wir die von `withdraw` ausgelöste Transition von `credit` nach `credit`.

  Die im Zustandsdiagramm spezifizierte Transition von `credit` nach `credit` ist:

  ```
  withdraw [not blocked and self.balance>amount]
  ```

  In diese Transition werden nun Vor- und Nachbedingung von `withdraw` aus den OCL-Constraints eingefügt. Dabei wird die ursprüngliche Vorbedingung der Transition mit der Vorbedingung in OCL und die ursprüngliche Nachbedingung der Transition mit der Nachbedingung in OCL jeweils konjugiert.

Resultierende Transition in diesem Beispiel ist:

```
withdraw [not self.blocked and self.balance>amount and amount>0 and
self.balance>=amount] / [self.balance=self.balance@pre-amount]
```

Analoges gilt für die zweite von `withdraw` ausgelöste Transition. So werden alle Transitionen, die von Methoden ausgelöst werden, für die Vor- und Nachbedingungen in OCL spezifiziert sind, durch diese erweitert.

- Die Konjunktion der Vor- und Nachbedingungen wird anschließend um die Klasseninvariante erweitert, indem diese mit der neuen Vor- bzw. Nachbedingung ebenfalls mit einem logischen `und` verknüpft werden.

  Resultierende Transition ist dann:

```
withdraw [not self.blocked and self.balance>amount and amount>0 and
self.balance>=amount and self.balance>=0] / [self.balance=self.balance@pre
-amount and self.balance>=0]
```

Die so erzeugten neuen Transitionen ersetzen nun die alten Transitionen im Zustandsdiagramm, die von `withdraw` ausgelöst werden. Analoges gilt für die Transitionen, die von der Methode `deposit` ausgelöst werden. Das resultierende Zustandsdiagramm nach der Integration der Vor- und Nachbedingungen zeigt Abbildung 10.6[1].

Wie man am Beispiel sieht, wird das resultierende Zustandsdiagramm schnell unübersichtlich. Es müssen viele Bedingungen an den Transitionen überprüft werden. Das Beispiel zeigt auch, daß eine Überschneidung oder Inklusion von Bedingungen an den Transition möglich ist (Beispiel `self.balance = amount` und `self.balance >= amount`). Da die Bedingungen bisher durch das Verfahren nicht vereinfacht werden, ist eine Überprüfung aller Bedingungen, also auch von Bedingungen, die einander einschließen, nicht zu vermeiden.

Über die Integration der Klasseninvarianten läßt sich diskutieren, da die Zustandsinvarianten die Klasseninvariante nicht verletzen dürfen. Das bedeutet, daß Zustandsinvarianten Teilmengen der von der Klasseninvarianten beschriebenen Menge beschreiben. Das Testsystem stellt dem Benutzer deshalb frei, nur Vor- und Nachbedingungen oder zusätzlich auch die Klasseninvariante zu integrieren.

Der Algorithmus ersetzt jedes Label einer Transition, in der eine bestimmte Methode m eine Transition auslöst, durch die neue Transition, indem die Vorbedingung der Transition um die Vorbedingung der Methode ergänzt wird. Analoges gilt für die Nachbedingung.

replace_trans ==

all $t \in \Lambda$:

let $m[c_{pre}]/[c_{post}] := t : m[c_{pre}$ `"and"` $PRE(m)]/[c_{post}$ `"and"` $POST(m)]$

10.2 Erweiterte OCL-Constraints

Die zweite Variante, bei der die bestehenden OCL-Constraints erweitert werden, benötigt zwei Schritte.

Zunächst werden Vor- und Nachbedingungen aus einem Zustandsdiagramm extrahiert. Im zweiten Schritt erfolgt die Kombination mit den ursprünglichen Vor- und Nachbedingungen. Bei dieser Variante bleibt die Klasseninvariante unverändert, weshalb sie hier nicht weiter betrachtet wird.

Die UML-Semantik [110] beschreibt den Zusammenhang zwischen Zuständen und Vor- und Nachbedingungen einer Methode m wie folgt:

Seien S_1 und S_2 Zustände, in denen m eine Transtion auslöst. Seien C_1 die Vorbedingung, S_{1ff} der Folgezustand und C_{1ff} die Nachbedingung der Transition, die m in S_1 auslöst. Seien C_2 die

[1] Auf die Integration der Klasseninvarianten wurde in der Abbildung aus Übersichtsgründen verzichtet.

Zustand	open	credit	debit	empty
Invariante	not self.closed	self.balance > 0	self.balance < 0	self.balance = 0

Abbildung 10.7: Zustandsinvarianten der Zustände `open`, `credit`, `debit` und `empty`

Vorbedingung, S_{2ff} der Folgezustand und C_{2ff} die Nachbedingung der Transition, die m in S_2 auslöst. Dann gilt für die Vorbedingung pre und die Nachbedingung post von m:

pre:	S_1 ist in der aktuellen Zustandskonfiguration $\wedge$ C_1 $\vee$ S_2 ist in der aktuellen Zustandskonfiguration $\wedge$ C_2
post:	if Vorzustand S_1 ist in der aktuellen Zustandskonfiguration $\wedge$ C_1 $then$ S_{1ff} ist in der aktuellen Zustandskonfiguration $\wedge$ C_{1ff} $else$ if Vorzustand S_2 ist in der aktuellen Zustandskonfiguration $\wedge$ C_2 $then$ S_{2ff} ist in der aktuellen Zustandskonfiguration $\wedge$ C_{2ff}

Zustände sind jedoch nicht gleich Zustandsinvarianten, so daß bei der Extraktion von Vor- und Nachbedingungen eine Strategie gefunden werden muß, Zustandsinvarianten - und nicht Zustände - miteinander in Beziehung zu setzen.

Zur Überprüfung, ob ein Objekt sich in einem Zustand befindet, ist eine Abbildung von Zuständen auf Zustandsinvarianten nötig, da die realen Zustände der zu testenden Objekte nur anhand von Zustandsinvarianten ermittelt werden. Befindet sich ein Zustand in der aktuellen Zustandskonfiguration, so gilt die Zustandsinvariante. Die einfachste Lösung wäre folglich, die Aussage, daß ein Zustand sich in der aktuellen Zustandskonfiguration befindet, durch seine Zustandsinvariante zu ersetzen.

Eine Schwierigkeit der direkten Übersetzung von Zuständen nach Zustandsinvarianten ergibt sich jedoch aus der semantischen Interpretation des Zustandsdiagramms. Zustände im Statechart sind disjunkt, für Zustandsinvarianten gibt es jedoch je nach Semantik zwei verschiedene Möglichkeiten:

1. Die Zustandsinvarianten sind ebenso wie die Zustände disjunkt, d.h. die Zustandsinvariante gilt genau dann, wenn sich das zu testende Objekt in einem der Zustandsinvariante entprechenden Zustand befindet:
   $\forall s_1, s_2 \mid \neg(s_1 \chi^+ s_2) \wedge \neg(s_2 \chi^+ s_1) \Rightarrow (s^+_{inv}(s_1) \wedge s^+_{inv}(s_2) = false)$.
2. Die Zustandsinvarianten sind nicht disjunkt, d.h. wenn sich das zu testende Objekt in einem bestimmten Zustand befindet, gilt die dem Zustand entsprechende Zustandsinvariante. Die Umkehrung der Aussage gilt jedoch nicht.
   $\exists s_1, s_2 \mid \neg(s_1 \chi^+ s_2) \wedge \neg(s_2 \chi^+ s_1) \wedge (s^+_{inv}(s_1) \wedge s^+_{inv}(s_2) = true)$.

Beide semantische Varianten werden in der vorliegenden Arbeit betrachtet und unterliegen der Wahl des Benutzers des Testsystems UT^3. Für beide Varianten wurde eine jeweils unterschiedliche Art der Verknüpfung der Zustandsinvarianten des Vorzustands und des Nachzustands in der Nachbedingung (statt der Verknüpfung $if - then - else$) gewählt. Dies wird zunächst am Beispiel erläutert.

Als Zustandsdiagramm wird in diesem Fall das etwas komplexere Diagramm aus Abbildung 10.4 betrachtet. Dieses Zustandsdiagramm besitzt den Vorteil, daß die betrachtete Methode `withdraw` in drei Zuständen Transitionen auslöst. In einem der drei Zustände werden von der Methode `withdraw` drei Transitionen ausgelöst. Damit werden alle Möglichkeiten der jetzt vorgestellten Technik abgedeckt. Vor- und Nachbedingung der Methode `withdraw` der Klasse `Account` sind in Abbildung 10.5 gegeben. Es werden nun aus dem Zustandsdiagramm Vor- und Nachbedingungen

für die Methode `withdraw` extrahiert.

Für die Berechnung der Vorbedingung werden zunächst im Zustandsdiagramm alle Zustände identifiziert, in denen die Methode `withdraw` eine Transition auslöst. Im Beispiel sind es die Zustände `credit`, `debit` und `empty`. Anschließend werden die Zustandsinvarianten ermittelt. Eine Übersicht über die relevanten Zustandsinvarianten zeigt Abbildung 10.7[2].

Für die Vorbedingung werden nun die Zustände durch ihre Zustandsinvarianten ersetzt, wie es die einfache Lösung oben vorschlägt. Dazu wird die Disjunktion aller allgemein gültigen Zustandsinvarianten gebildet[3].

Im aktuellen Beispiel ist dies:

```
(not self.closed and self.balance > 0)      -- Zustandsinvariante credit
 or
(not self.closed and self.balance < 0)      -- Zustandsinvariante debit
 or
(not self.closed and self.balance = 0)      -- Zustandsinvariante empty
```

oder vereinfacht[4]:

```
not self.closed and (self.balance > 0 or self.balance < 0 or self.balance = 0)
```

Die Vorbedingung ist für beide Varianten gleich. Jedoch ist zu berücksichtigen, daß die Vorbedingung nur in dem Fall exakt ist, daß die Zustandsinvarianten disjunkt sind (Variante 1). Sind die Zustandsinvarianten nicht disjunkt (Variante 2), ist es möglich, daß es zwei Zustände mit der gleichen Zustandsinvarianten gibt, jedoch nur in einem der beiden Zustände löst die betrachtete Methode - hier: `withdraw` - eine Transition aus. Die Auswertung der Vorbedingung führt zu einem falsch-postiven Testurteil, wenn sich das zu testende Objekt in dem Zustand befindet, der zwar die Zustandsinvariante erfüllt, in dem die betrachtete Methode jedoch keine Transition auslöst.

Wie an der Vereinfachung zu sehen, ist der zweite Teil der Vorbedingung - der Teil nach dem *and* - äquivalent zu *true*, was damit zu erklären ist, daß die Unterzustände des Zustands `payments` den Datenraum der Variablen *self.balance* vollständig beschreiben. Es ist also möglich, nur die Invariante des Zustand `payments` als Vorbedingung anzunehmen, wenn eine Methode in allen Unterzuständen eine Transition auslöst. Dies ist jedoch nicht korrekt, wenn die Invarianten der Unterzustände den Datenraum des Oberzustands nicht vollständig abdecken.

Bei der Berechnung der Nachbedingung führen die beiden Varianten zu zwei verschiedenen Ergebnissen.

Bei der ersten Variante werden die Zustandsinvarianten der Vorzustände mit den Zustandsinvarianten der Nachzustände einer Transition mit einer Implikation in Beziehung gesetzt. Dabei werden alle Werte der Attribute und Parameter im Vorzustand mit dem OCL-Schlüsselwort @*pre* referenziert.

Beispiel für die Transition von `empty` nach `debit`, die von `withdraw` ausgelöst wird:

```
not self.closed@pre and self.balance@pre = 0   -- Vorzustand empty
and not self.blocked@pre                       -- Bedingung
   implies
not self.closed and self.balance@pre < 0       -- Nachzustand debit
```

[2]Die Zustandsinvarianten der Zustände `Account` und `payments` sind jeweils `true`. Alle anderen Zustandsinvarianten sind für die Methode `withdraw` nicht relevant, da sie in allen anderen Zuständen keine Transition auslöst.

[3]Man beachte den Unterschied zwischen allgemein gültiger Zustandsinvariante und spezifizierter Zustandsinvariante, wie er in Kapitel 8 erläutert ist.

[4]Die Vereinfachung wurde hier nur der Übersichtlichkeit halber vorgenommen. Eine weitere Vereinfachung ist in diesem Fall möglich. Vom Testsystem UT^3 wird die komplexe Form erzeugt.

```
context Account::withdraw (amount:int)

   pre : not self.closed and
         ((self.balance > 0)
           or
          (self.balance < 0)
           or
          (self.balance = 0))

   post:   (not self.closed@pre and not self.blocked@pre and self.closed)
         and
           ((self.balance@pre > 0 and amount@pre = self.balance@pre)      } Transition
                 implies                                                  } credit -> empty
             self.balance = 0)
         and
           ((self.balance@pre > 0 and amount@pre > self.balance@pre)      } Transition
                 implies                                                  } credit -> debit
             self.balance < 0)
         and
           ((self.balance@pre > 0 and amount@pre < self.balance@pre)      } Transition
                 implies                                                  } credit -> credit
             self.balance > 0)
         and
            (self.balance@pre = 0                                         } Transition
                 implies                                                  } empty -> debit
             self.balance < 0)
         and
            (self.balance@pre < 0                                         } Transition
                 implies                                                  } debit -> debit
             self.balance < 0))
```

Abbildung 10.8: Aus dem Zustandsdiagramm extrahierte Vor- und Nachbedingung für `withdraw` (Variante 1)

Beispiel für die Transition von `credit` nach `empty`, die von `withdraw` ausgelöst wird:

$not\ self.closed@pre\ and\ self.balance@pre > 0$	-- *Vorzustand* `credit`
$and\ amount@pre = self.balance@pre\ and\ not\ self.blocked@pre$ $implies$	-- *Bedingung*
$not\ self.closed\ and\ self.balance@pre = 0$	-- *Nachzustand* `empty`

Alle Implikationen werden anschließend durch Konjunktionen miteinander verknüpft. Die resultierende Vor- und Nachbedingung (vereinfacht) für `withdraw` zeigt Abbildung 10.8.

Bei der zweiten Variante sind die Implikationen durch Konjunktionen ersetzt, die Konjunktionen durch Disjunktionen. Die resultierende Vor- und Nachbedingung (vereinfacht) für die zweite Variante ist in Abbildung 10.9 gezeigt.

Vergleicht man beide Varianten, so stellt man fest, daß die erste Variante, bei der die Zustandsinvarianten disjunkt sind, sehr viel präziser und damit besser zum Test geeignet ist (siehe Abbildung 10.10). Die zweite Variante erspart dem Entwickler des Systems zwar die Überprüfung der Zustandsinvarianten im Zustandsdiagramm auf Disjunktheit, ist aber weniger zum Testen geeignet, da falsch-positive Testurteile erzeugt werden, wenn die zu testenden Objekte beim Test Zustände mit Werten im Schnittpunkt der Zustandsinvarianten annehmen. Bei der Benutzung des Testsystems UT^3 wird deshalb empfohlen, Zustandsinvarianten disjunkt zu spezifizieren, um falsch-positive Ergebnisse zu vermeiden[5].

Die aus dem Zustandsdiagramm gewonnenen Vor- bzw. Nachbedingungen werden durch Konjunktion mit den ursprünglichen Vor- und Nachbedingungen in OCL verknüpft.

[5]Die Spezifikation disjunkter Zustandsinvarianten besitzt auch Vorteile in Bezug auf die Integration des Testorakels in das zu testende System. Siehe dazu Kapitel 14.

```
context Account::withdraw (amount:int)

   pre : not self.closed and
         ((self.balance > 0)
           or
          (self.balance < 0)
           or
          (self.balance = 0))

   post:   (not self.closed@pre and not self.blocked@pre and self.closed)
         and
           ((self.balance@pre > 0 and amount@pre = self.balance@pre)      } Transition
                 and                                                      } credit -> empty
             self.balance = 0)
         or
           ((self.balance@pre > 0 and amount@pre > self.balance@pre)      } Transition
                 and                                                      } credit -> debit
             self.balance < 0)
         or
           ((self.balance@pre > 0 and amount@pre < self.balance@pre)      } Transition
                 and                                                      } credit -> credit
             self.balance > 0)
         or
            (self.balance@pre = 0                                         } Transition
                 and                                                      } empty -> debit
             self.balance < 0)
         or
            (self.balance@pre < 0                                         } Transition
                 and                                                      } debit -> debit
             self.balance < 0))
```

Abbildung 10.9: Aus dem Zustandsdiagramm extrahierte Vor- und Nachbedingung für `withdraw` (Variante 2)

	Variante 1	**Variante 2**
Zustandsinvarianten	disjunkt	nicht disjunkt
Vorbedingung	exakt	falsch-positiv
Nachbedingung	exakt	falsch-positiv

Abbildung 10.10: Bewertung der Varianten im Hinblick auf das Testurteil

```
context Account::withdraw (amount:int)

   pre : amount > 0                                  } Ursprüngliche Vorbedingung
         and
         (not self.closed and
         ((self.balance > 0)
           or                                        } Extrahierte Vorbedingung
          (self.balance < 0)
           or
          (self.balance = 0)))

   post: self.balance = self.balance@pre - amount    } Ursprüngliche Nachbedingung
         and
         ((not self.closed@pre and not self.blocked@pre and self.closed)
         and
           ((self.balance@pre > 0 and amount@pre = self.balance@pre)
                 implies
              self.balance = 0)
         and
           ((self.balance@pre > 0 and amount@pre > self.balance@pre)
                 implies
              self.balance < 0)
         and
           ((self.balance@pre > 0 and amount@pre < self.balance@pre)
                 implies
              self.balance > 0)
         and
            (self.balance@pre = 0
                 implies
              self.balance < 0)
         and
            (self.balance@pre < 0
                 implies                             Extrahierte Nachbedingung
              self.balance < 0)))
```

Abbildung 10.11: Erweiterte Vor- und Nachbedingung für `withdraw` (Variante 1)

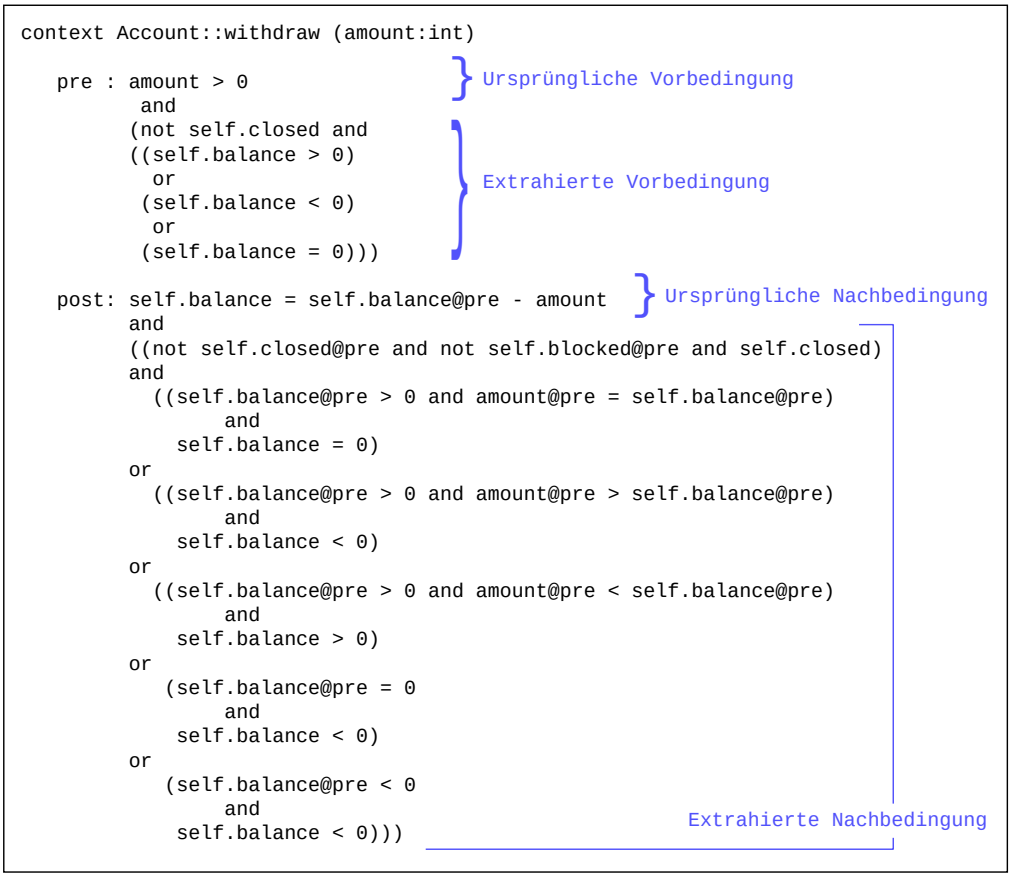


Abbildung 10.12: Erweiterte Vor- und Nachbedingung für `withdraw` (Variante 2)

Die Abbildung 10.11 zeigt die Kombination der ursprünglichen und der aus dem Zustandsdiagramm extrahierten Vor- und Nachbedingung als die erweiterte Vor- und Nachbedingung für die Methode `withdraw`. Abbildung 10.12 zeigt die gleiche Vor- und Nachbedingung für die Variante 2.

Die folgenden Algorithmen extrahieren Vor- und Nachbedingungen aus dem Zustandsdiagramm. Die Variable *variant*1 zeigt dabei an, ob es sich um die erste Variante handelt, die Variable *variant*2, ob es sich um die zweite Variante handelt.

$$
\begin{array}{l}
\mathit{calc\_precond}(m : M) : \Gamma == \\
\quad \textbf{let}\ \mathit{prem} := \texttt{"true"} : \\
\qquad \textbf{all}\ l \in \Lambda \mid (\exists\, c_{pre}, c_{post} \mid l = m[c_{pre}]/[c_{post}]) : \\
\qquad\quad (\textbf{sel}\ \mathit{sout}, \mathit{sin} \in \mathit{states} \mid (\mathit{sout}, l, \mathit{sin}) \in \Delta : \\
\qquad\qquad \mathit{prem} := \mathit{prem}\ \texttt{"or"}\ \texttt{"("}\ \mathit{calc\_inv}^{+}(\mathit{sout})\ \texttt{"and"}\ \mathit{c\_pre}\ \texttt{")"}\ ; \\
\qquad \mathit{prem}
\end{array}
$$

$$
\begin{array}{l}
\mathit{calc\_postcond}(m : M) : \Gamma == \\
\quad \textbf{let}\ \mathit{postm} := \texttt{"true"}\ ; \\
\qquad \mathit{innersym}, \mathit{outersym} := (\textbf{if}\ \mathit{variant}1 \\
\qquad\qquad \textbf{then}\ (\ \texttt{"implies"}\ ,\ \texttt{"and"}\ ) \\
\qquad\qquad \textbf{else}\ (\ \texttt{"and"}\ ,\ \texttt{"or"}\ )) : \\
\qquad \textbf{all}\ l \in \Lambda \mid (\exists\, c_{pre}, c_{post} \mid l = m[c_{pre}]/[c_{post}]) : \\
\qquad\quad (\textbf{sel}\ \mathit{sout}, \mathit{sin} \in \mathit{states} \mid (\mathit{sout}, l, \mathit{sin}) \in \Delta : \\
\qquad\qquad \mathit{postm} := \mathit{postm}\ \mathit{outersym}\ \texttt{"("}\ \mathit{calc\_inv}^{+}(\mathit{sout})\ \texttt{"and"}\ \mathit{c\_pre} \\
\qquad\qquad\qquad\qquad \mathit{innersym}\ \mathit{calc\_inv}^{+}(\mathit{sin})\ \texttt{"and"}\ \mathit{c\_post}\ \texttt{")"}\ ; \\
\qquad \mathit{postm}
\end{array}
$$

10.3 Vergleich der Techniken

Beide Techniken, sowohl die angereicherten Zustandsdiagramme als auch die erweiterten OCL-Constraints, besitzen Vor- und Nachteile.

Die angereichten Zustandsdiagramme sind einfach zu erzeugen, da nur jede Transition modifiziert werden muß. Bei der Integration der OCL-Constraints in die Zustandsdiagramme bleibt die Zustandsdiagramm-spezifische Information erhalten und wird nur durch zusätzliche Bedingungen ergänzt. Zudem kann das angereicherte Zustandsdiagramm als Basis der Testfallgenerierung dienen.

Ein Nachteil ist, daß das Zustandsdiagramm aufgebläht wird, da an jedem Ereignis, also jedem Methodenaufruf, Information zu ergänzen ist. Die Implementierung des Testorakels erhält, soweit möglich, alle Zustandsdiagramm-Eigenschaften (siehe auch Kapitel 14), so daß ein komplexes Testorakel bereits aus den ursprünglichen Zustandsdiagrammen erzeugt wird. Die Integration der OCL-Constraints erhöht die Komplexität weiter. Zudem liegt Information jetzt mehrfach vor.

Die erweiterten OCL-Constraints sind durch die Integration der gesamten Information aus dem Zustandsdiagramm entsprechend komplex. Zudem verliert man einen Teil der Zustandsdiagramm-spezifischen Information, da Zustände durch ihre Zustandsinvarianten in den OCL-Constraints repräsentiert werden. Ein direkter Rückschluß auf den Zustand im ursprünglichen Zustandsdiagramm ist nur noch bedingt möglich. Die Zustandsdiagramm-spezifische Information ist aber für die Auswertung der Testläufe und die Behebung der Fehler hilfreich.

Der Vorteil dieser Technik ist die Generierung eines kompakten Testorakels, das sich leicht in die zu testende Implementierung integrieren läßt (siehe auch Kapitel 15). Es muß nur vor jedem Methodenaufruf und nach jeder Methodenausführung die Einhaltung von Invariante, Vor- und Nachbedingung überprüft werden.

Eine Alternative ist die unabhängige Prüfung beider Diagrammtypen durch jeweils ein Testorakel. Diese Lösung wurde in der vorliegenden Arbeit nicht verfolgt, da die Machbarkeit der Integration in einen Diagrammtyp im Vordergrund der Arbeit stand.

Es ist jedoch zu prüfen, ob zwei unabhängige Testorakel aus Effizienzgründen oder der besseren Fehleraufdeckung für den Test geeigneter sind. Da sowohl Zustandsdiagramme als auch OCL-Constraints in die zu testende Implementierung integriert werden können, ist ein Vergleich der verschiedenen Techniken innerhalb des Testsystems UT^3 problemlos möglich.

Beide Techniken, die angereicherten Zustandsdiagramme sowie die erweiterten OCL-Constraints müssen ihre Effizienz in Bezug auf den Test noch zeigen. Dazu ist eine größere Fallstudie bereits in Planung. Erst nach Auswertung dieser Fallstudie kann entschieden werden, welcher der beiden Varianten der Vorzug zu geben ist.

Eines ist jedoch bereits sicher. Die Entscheidung für eines der beiden Verfahren ist immer auch von der ursprünglichen Spezifikation abhängig. So macht es wenig Sinn, bei einem einfachen Zustandsdiagramm - z.B. mit zwei Zuständen und zwei Transitionen - die Zustandsdiagrammspezifische Information zu erhalten, wenn gleichzeitig die Vor- und Nachbedingungen der referenzierten Methoden sehr komplex sind.

Kapitel 11

Zusammenfassung

Dieses Kapitel faßt die Ergebnisse der entwickelten UML-basierten Testverfahren zusammen.

Aufgrund der Analyse in Kapitel 5 wurde eine Auswahl unter den UML-Diagrammtypen getroffen, um ein UML-basiertes Testverfahren im Testsystem UT^3 zu realisieren. Die getroffene Auswahl beschränkt den Test auf die Testphasen Klassen- und Integrationstest.

Auch in Bezug auf die verwendeten Modelle mußten Einschränkungen gemacht werden. Hier sollen diese Einschränkungen noch einmal im Einzelnen für die einzelnen Diagrammtypen beschrieben werden.

Zustandsdiagramme. Zustandsdiagramme werden im entwickelten Ansatz sowohl für das Testorakel als auch für die Testfallgenerierung verwendet. Der im Rahmen der vorliegenden Arbeit entwickelte Ansatz konzentriert sich auf eine der zwei Varianten der UML-Zustandsdiagramme, die Protocol State Machines. Diese werden benutzt, um Objektzyklen zu beschreiben.

Für die Testfallgenerierung werden bisher parallele Zustandsdiagramme nur ungenügend berücksichtigt. Um diese besser zur Initialisierung von Testsequenzen zu benutzen, müssen die Zustandsdiagramme zunächst einem sogenannten *Flattening* unterzogen werden, wie es auch in dem Ansatz in [100] vorgenommen wurde. Ein Flattening ist bisher in UT^3 nicht implementiert, bedeutet aber keine größere Schwierigkeit.

Einige Abhängigkeiten werden bisher nicht berücksichtigt. Gibt es Abhängigkeiten zwischen Bedingungen an Transitionen und Zustandsinvarianten in parallelen Zuständen, so kann eine erzeugte Initialisierungssequenz eventuell nicht ausgeführt werden. Um solche Probleme zu lösen, ist die Anbindung eines Constraintsolvers nötig.

Die Initialisierungssequenzen erzeugen immer einen der kürzesten Wege zu einem Zustand. Bisher ist die Auswahl einer längeren Initialisierungssequenz, z.B. der wiederholte Aufruf einer Self-Transition, nicht vorgesehen.

Nichtdeterminismen im Zustandsdiagramm schließlich führen in einigen Fällen zu einer Transitionssequenz zur Initialisierung, die entweder im angestrebten Zielzustand oder durch nichtdeterministische Auswahl einer Transition in einem anderen, eventuell nicht gewünschten Zustand endet.

Die Ableitung des Testorakels aus Zustandsautomaten unterliegt weiteren Einschränkungen, so werden z.B. aufgeschobene Ereignisse und Transitionen vom Typ *Fork* und *Join* nicht unterstützt. Eine genaue Übersicht findet sich in Kapitel 8.2.

Sequenzdiagramme. In Sequenzdiagrammen werden bisher keine Parameter an den Nachrichten berücksichtigt. Lediglich Bedingungen an den Transitionen sind Teil des Testfalls. So geht in einzelnen Fällen Information verloren.

Die Sequenz, mit der eine Implementierung auf einen Stimulus antwortet, ist bisher nicht Teil des Testorakels.

Das Testsystem UT^3 soll jedoch so erweitert werden, daß auch Parameter an Nachrichten in die Testfallgenerierung und die Antwortsequenz der zu testenden Implementierung in das Testorakel einbezogen werden.

OCL-Constraints. Mit der vorgestellten Technik sind prinzipiell alle Invarianten, Vor- und Nachbedingungen in OCL unterstützt, da diese durch die vorgestellten Techniken nicht verändert, sondern nur erweitert werden. Eine Ausnahme ist die Generierung von Informationen über den Vorzustand der zu testenden Implementierung aus dem Zustandsdiagramm. Diese Erweiterung von Attributen, Observer-Methoden und Methodenparametern wird durch den im folgenden Teil vorgestellten OCL-Parser durchgeführt.

Fazit. Trotz der Einschränkungen an die verwendeten Diagrammtypen wird in der vorliegenden Arbeit ein umfassendes Konzept zur UML-basierten Unterstützung des Tests vorgestellt. Ziel der Arbeit war es, sowohl die Testfallgenerierung als auch die Ableitung des Testorakels zu unterstützen. Die Beschränkung auf einige Diagrammtypen war notwendig. Gleichzeitig ist das Testsytem so entworfen, daß eine Erweiterung um andere Diagrammtypen möglich ist.

Gerade in Bezug auf das Testorakel sind die gewählten UML-Bestandteile als vollständig anzusehen, da sowohl Zustandsdiagramme als auch OCL-Constraints Softwareverträge zwischen Objekten, den Zustandsraum und den Lebenszyklus eines Objekts vollständig beschreiben. Sie sind deshalb adäquat für den Klassentest und geeignet für den Integrationstest.

In Bezug auf das in Kapitel 3 definierte Testmodell wird allerdings auch klar, daß ein wesentlicher Teil beim Test bisher nicht vom Testorakel abgedeckt wird, die Überprüfung des Sollverhaltens. Dieses Manko soll jedoch mit der Nutzung von Sequenzdiagrammen als Testorakel behoben werden.

Des weiteren beschäftigt sich die vorliegende Arbeit mit der Integration von Testcode in das zu testende System. Dabei beschränkt sich die Arbeit zunächst auf die Integration des Testorakels, die schwieriger ist als die Integration der Testtreiber, da interne Zustände der zu testenden Implementierung zu überwachen sind. Die Integration des Testorakels in aspektorientierter Weise wird im folgenden Teil beschrieben.

Teil III

Testen als Aspekt

Two concerns are considered crosscutting, if given a modular structure for capturing one concern, the other cannot be encapsulated in a module, but rather cuts across several modules as introduced by the first concern. Stephan Herrmann [47]

Kapitel 12

Aspekte und ObjectTeams

Eines der Probleme beim Testen von Software betrifft das Monitoring des zu testenden Systems sowie die Integration von Testorakeln und Testtreibern. Klassische Ansätze fügen dazu zusätzlichen Code in den Quellcode, Bytecode oder Maschinencode des zu testenden Systems ein, man spricht von Instrumentierung. In der Regel ist dazu eine Neukompilierung oder zumindest ein Verändern des Maschinencodes des zu testenden System nötig.

Der instrumentierte Quellcode kann nicht ohne weiteres auf dem Zielsystem eingesetzt werden, da die Instrumentierung nur zu Testzwecken eingefügt wurde. Eine Ausnahme ist der Bereich der eingebetteten Systeme, in dem jede Art der Instrumentierung auch im ausgelieferten Code vorhanden bleibt[1].

Der Fokus dieser Arbeit sind objektorientierte Systeme. Hier ist meist eine Instrumentierung zu Testzwecken im Endprodukt nicht vorhanden, da sie für den Endbenutzer zu unerwünschtem Verhalten des Systems führen kann, sei es geringere Performanz durch den zusätzlichen Code oder die Protokollierung, seien es Fehlermeldungen, die zum Testen in das zu testende System integriert wurden. Auch sind die Anforderungen an Sicherheit und Vorhersagbarkeit des Laufzeitverhaltens in der Regel nicht so hoch wie an eingebettete Systeme.

Der in dieser Arbeit vorgestellte Ansatz verfolgt einen anderen als den klassischen Weg der Instrumentierung. Die Vorteile aspektorientierter Programmierung [57] werden genutzt, um den Code des zu testenden Systems vom Testcode zu trennen. Der Mechanismus des Einfügens von zusätzlichem Code wird wiederverwendet, da aspektorientierte Sprachen diesen Mechanismus bereits implementieren. Der Testcode wird als Aspekt implementiert (bzw. generiert). Durch die Trennung von Codeeinfügen und Codegenerierung wurde eine flexible Lösung geschaffen, die sich sehr leicht auf ähnliche Probleme übertragen läßt.

Der Einsatz der objektorientierten Spracherweiterung ObjectTeams [47, 82] bietet darüber hinaus den Vorteil, daß das zu testende System zur Ladezeit durch den zusätzlichen Code erweitert wird. Weder Sourcecode noch Bytecode werden verändert, so daß das getestete System im Anschluß an den Test unverändert beim Endbenutzer installiert werden kann. Das vermindert den Aufwand des Testens, da nicht mehrere Versionen desselben Systems gepflegt werden müssen.

Dieser Teil der Arbeit beschäftigt sich mit der aspektorientierten Instrumentierung des zu testenden Systems. Zunächst werden im aktuellen Kapitel die Grundlagen aspektorientierter Programmiertechniken vorgestellt und die Konzepte der verwendeten Sprache ObjectTeams/Java eingeführt.

Das Kapitel 13 klärt die Frage, ob es sich beim Testen um einen Cross-Cutting-Concern im Sinne der aspektorientierten Programmierung handelt und gibt einen Überblick über andere Arbeiten auf dem Gebiet. Die Kapitel 14 und 15 stellen die entwickelten Techniken vor, mit denen

[1] Da es sich bei eingebetteten Systemen meist um sicherheitskritische oder Realzeitsysteme handelt, kann ein Entfernen des Instrumentierungscodes fatale Folgen auf das System haben, z.B. kann das System ohne den Instrumentierungscode ein anderes Laufzeitverhalten aufweisen. Deshalb gilt bei diesen Systemen, daß der getestete Code (inklusive Instrumentierungscode) auch der Code ist, der ausgeliefert wird.

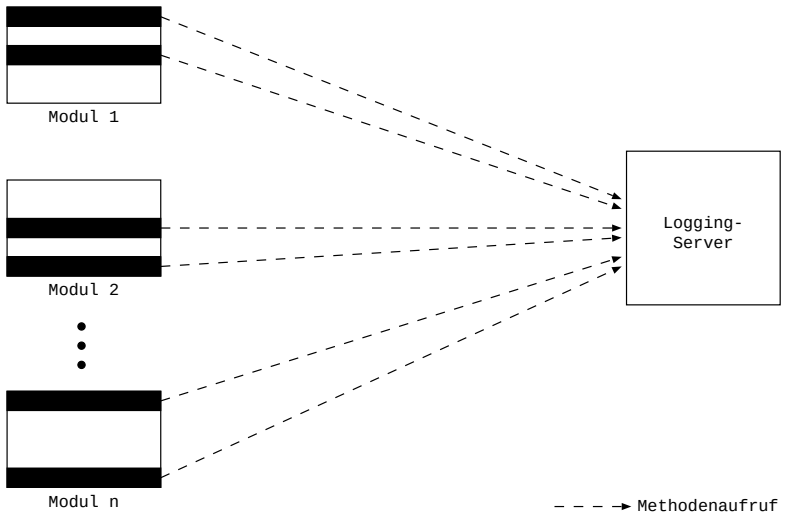


Abbildung 12.1: Beispiel Logging ohne aspektorientierte Programmierung

die in Kapitel 10 erzeugten Testorakel in ObjectTeams-Code umgesetzt und in das zu testende System integriert werden.

Kapitel 16 faßt die Ergebnisse dieses Teils zusammen.

Die vorgestellte Implementierung nutzt ObjectTeams/Java, eine aspektorientierte Erweiterung der Programmiersprache Java. Die vorgestellte Technik läßt sich jedoch problemlos auf andere objektorientierte Sprachen übertragen, sofern sie durch das ObjectTeams-Programmiermodell[2] unterstützt werden.

Das ObjectTeams-Programmiermodell stützt sich auf unterschiedliche Techniken, die neben der aspektorientierten Programmierung auch rollenbasierte Programmierung umfassen. Beide Konzepte werden hier kurz eingeführt, um das Verständnis für die Sprache ObjectTeams/Java zu erleichtern.

12.1 Aspektorientierte Programmierung

Eines der zentralen Konzepte zur Verbesserung der Qualität von Software ist die Modularisierung. Sie hilft, komplexe Probleme auf einfache Teilprobleme (in Form von Modulen) abzubilden und zur Lösung des Problems die Einzelteile zu kombinieren (modulare Dekomposition bzw. Komposition [80]). Modularisierung bietet Vorteile in allen Phasen der Softwareentwicklung. In der Analyse werden Probleme zerlegt, im Entwurf einzeln betrachtet und modelliert, in der Implementierung erleichtern Module Teamarbeit und Verantwortlichkeiten und in der Wartung können Änderungen und Erweiterungen bei guter Modularisierung lokal gehalten werden. Zudem können Module durch Rekombination zu neuen Systemen zusammengesetzt und so wiederverwendet werden.

Einige Eigenschaften und Anforderungen (Concerns) des System lassen sich jedoch mit objektorientierten Zerlegungstechniken nicht modularisieren. Oft muß bei Paaren von Concerns entschieden werden, nach welcher der Concerns das System zu modularisieren ist. Der andere Concern, der nicht Grundlage des Systementwurfs war, verteilt sich dann über das gesamte System und bildet einen *Cross-Cutting-Concern*.

Typische Beispiele für Cross-Cutting-Concerns sind Logging, Sicherheit, Synchronisation oder Verteiltheit. Zum Beispiel läßt sich der Server für das Logging sehr gut in einer Logger-Klasse

[2]Es gibt neben der Sprache *ObjectTeams/Java* die Sprache *Ruby Object Teams* als Erweiterung der Sprache Ruby sowie eine prototypische Implementierung der Sprache *Object Teams for C++*.

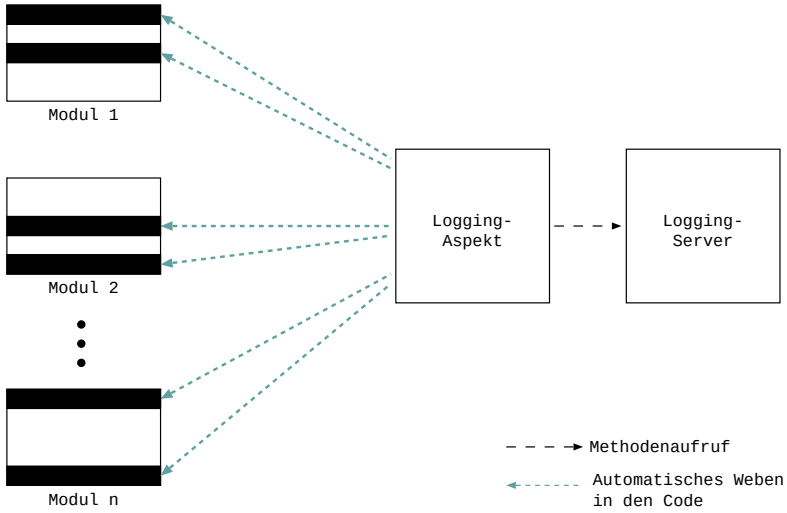


Abbildung 12.2: Beispiel Logging mit aspektorientierter Programmierung

kapseln, die Aufrufe in den Clients sind jedoch über das gesamte System verteilt (siehe Abbildung 12.1). Das führt zu unübersichtlichem Code, der schwer zu warten ist, da der Logging-Concern über viele Klassen des Systems verteilt ist (auch als *Scattering* bezeichnet) und gleichzeitig mit der Business-Logik des Systems eng verwebt ist (auch als *Tangling* bezeichnet).

Aspektorientierte Programmierung [57] löst diesen Konflikt, indem das Scattering und Tangling nicht mehr im Quellcode sichtbar ist und der Cross-Cutting-Concern in einem eigenem Modul als *Aspekt* gekapselt wird. Das Einweben des Aspekts in die Business-Logik des Systems wird nachträglich von einem *Aspektweber* - je nach verwendeter aspektorientierter Sprache zur Compile- oder zur Ladezeit - durchgeführt. Man spricht auch von der *Adaption* des ursprünglichen Systems durch einen Aspekt.

Das Einweben von zusätzlichem Code kann nur an bestimmten Punkten im Kontrollfluß, den *Joinpoints*, erfolgen. Ein Joinpoint ist zum Beispiel der Aufruf, das Eintreten oder das Verlassen einer Methode, der Zugriff auf eine Variable oder das Werfen einer Ausnahme. Joinpoints können durch die Quantifizierungsmöglichkeiten der aspektorientierten Sprache zu *Pointcuts* zusammengefaßt werden. Ein Pointcut kann zum Beispiel der Aufruf aller Methoden sein, die schreibenden Zugriff auf ein Attribut bieten (Setter-Methoden). Je nach verwendeter Sprache ist eine Quantifizierung mehr oder weniger präzise möglich (siehe auch [30]).

Der zusätzliche Code wird in einen *Advice* ausgelagert. Advices werden in der Regel in der gleichen Programmiersprache geschrieben, in der auch die Businesslogik des zu adaptierenden Systems implementiert ist. Advices werden an Pointcuts gebunden. Dabei gibt es verschiedene Möglichkeiten, wann der Advice-Code ausgeführt werden soll, sowohl in zeitlicher Hinsicht (z. B. vor oder nach dem ursprünglichen Code) als auch kontrollflußabhängig (z. B. nur wenn die zu adaptierende Methode aus einer bestimmten anderen Methode aufgerufen wurde oder der Aspekt aktiv ist).

Abbildung 12.2 zeigt die Implementierung des Logging-Beispiels mit Hilfe aspektorientierter Programmierung. Das eigentliche Programm (Module 1 bis n) enthält ausschließlich die Business-Logik. Der Logging-Code ist vollständig in zwei weitere Module ausgelagert. Das Logging-Modul enthält wie in Abbildung 12.1 den serverseitigen Logging-Code. Die zuvor verteilten Aufrufe der Clients sind nun gekapselt im Logging-Aspekt. Das Weben in den Code geschieht automatisch durch den Aspektweber und ist für den Entwickler transparent. Änderungen am Logging-Code betreffen jetzt nur noch wenige Module.

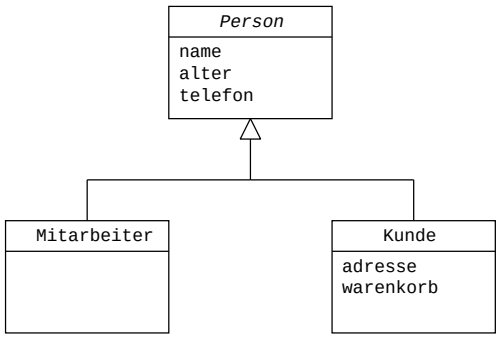


Abbildung 12.3: Beispiel Personenhierarchie ohne rollenbasierte Programmierung

12.2 Rollenbasierte Programmierung

Zum Verständnis des ObjectTeams-Programmiermodells ist es hilfreich, neben der aspektorientierten Programmierung auch einen Eindruck von rollenbasierter Programmierung zu besitzen. Dieser Abschnitt gibt einen kurzen Überblick.

Rollenbasierte Programmierung [92, 61] adressiert das Problem der Verwendung desselben Objekts in verschiedenen Kontexten. Dieses Problem wird durch Objektorientierung nur teilweise gelöst. Wie bereits in Kapitel 3 ausgeführt, ist in objektorientierten Programmen die mehrfache Referenzierung desselben Objekts in verschiedenen Kontexten möglich, wobei die einzelnen Referenzen auch als Rollen oder Sichten bezeichnet werden. Objektorientierte Systeme differenzieren diese Rollen jedoch nicht. So ist oft in der einen Sicht mehr Information oder ein anderes Verhalten eines Objekts nötig. Auch können Objekte in den meisten objektorientierten Programmiersprachen zur Laufzeit ihren Typ nicht ändern oder zu zwei verschiedenen Klassen gehören.

Ein typisches Beispiel sind Personen. Eine klassische Modellierung einer Personenhierarchie findet sich in Abbildung 12.3. Eine Person kann entweder ein Mitarbeiter sein oder ein Kunde, daß heißt, Objekte sind einer der beiden Unterklassen zugeordnet.
Diese einfache Implementierung führt zu folgenden Problemen:

Sichtbarkeit. Alle Attribute und Methoden sind in allen Kontexten sichtbar. Oft ist es nötig, die Sichtbarkeit in verschiedenen Kontexten einzuschränken.

Ausprägungen von Eigenschaften. Es gibt keine Möglichkeit, Verhalten oder Ausprägungen von Eigenschaften in verschiedenen Rollen unterschiedlich zu implementieren.

Rollenbindung. Ein Objekt ist an seine Klasse gebunden. Eine Zuordnung von Objekten zu mehreren Klassen ist nur durch Mehrfachvererbung zu lösen. In großen Klassenhierachien entsteht durch die so erzeugten *Intersection Classes* eine kombinatorische Explosion, da im schlimmsten Fall Intersection Classes zwischen allen vorhandenen Klassen in allen möglichen Kombinationen gebildet werden müssen.

Objektmigration. Objekte bleiben auch bei der Einführung von Intersection Classes an ihre Klassen gebunden. Eine Neuzuordnung eines Objekts während seines Lebenzyklusses ist nicht möglich.

Rollenbasierte Programmierung führt zur Lösung dieser Probleme einen neuen Typ von Objekten ein, sogenannte *Rollenobjekte*. Zunächst wurden Rollen[3] im Umfeld der objektorientierten

[3] In der rollenbasierten Programmierung wird der Begriff Rolle (anders als in der Objektorientierung) oft synonym gebraucht mit dem Begriff Rollenobjekt.

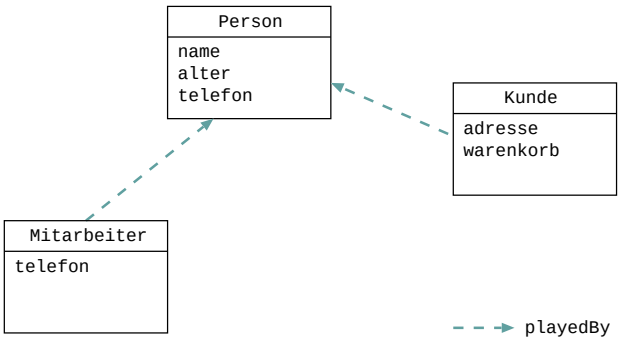


Abbildung 12.4: Beispiel Personen mit rollenbasierter Programmierung

Datenbanken (OODBS) benutzt [92]. Inzwischen gibt es auch Spracherweiterungen für objektorientierte Sprachen, die Rollenobjekte unterstützen, z.B. die Sprache *LAVA* [19, 65], die auf Basis der Sprache Java entwickelt wurde.

Abbildung 12.4 zeigt eine Modellierung der Personenhierarchie mit Hilfe rollenbasierter Programmierung. Die Klassen `Kunde` und `Mitarbeiter` werden als Rollenklassen modelliert. Statt einer Vererbungsbeziehung besteht eine *playedBy*-Beziehung zur Klasse Person. Das bedeutet, daß Objekte der Klasse `Kunde` (Rollenobjekte) zur Laufzeit auf ein *Basisobjekt*, hier vom Typ `Person`, verweisen, dessen Rolle sie spielen.

In der vorliegenden Arbeit wird die Beziehung zwischen Rollenobjekt und Basisobjekt als *Korrespondenz* bezeichet, daß heißt, ein Rollenobjekt besitzt ein korrespondierendes Basisobjekt[4].

Alle Anfragen an das Rollenobjekt werden an das Basisobjekt weitergeleitet (als *Delegation* bezeichnet[5]), wenn sie nicht direkt von Rollenobjekt bearbeitet werden können.

Ein Rollenobjekt kann zusätzliche Eigenschaften besitzen oder Eigenschaften des Basisobjekts überschreiben. Auch kann die Rolle nur einen Ausschnitt der Schnittstelle des Basisobjekts anbieten. Das Problem der Sichtbarkeit und der unterschiedlichen Ausprägung einer Eigenschaft ist damit gelöst. Objektmigration wird unterstützt, da Rollenobjekte zu einem Basisobjekt erzeugt und gelöscht werden können, daß heißt, Rollen werden eingenommen oder abgelegt.

Damit bietet rollenbasierte Programmierung eine Lösung für die Probleme unterschiedliche Sichtbarkeit, verschiedene Ausprägung von Eigenschaften, gleichzeitige Zuordnung zu mehreren Rollen und Objektmigration.

12.3 Das ObjectTeams-Programmiermodell

Das ObjectTeams-Programmiermodell ist hervorgegangen aus einer Reihe von verschiedenen Programmierparadigmen, insbesondere den zuvor vorgestellten Paradigmen aspektorientierte und rollenbasierte Programmierung sowie der Programmierung mit Objektkollaborationen[6].

Es wird ein neues Modulkonzept eingeführt, das *Team*. Ein Team ist sowohl eine statische Einheit im Programm, ähnlich einer Klasse oder eines Packages, als auch eine dynamische Einheit. Dabei dient ein Team einerseits zur Gruppierung von Klassen (statische Sicht), gleichzeitig fungiert es als Container für Objekte, die von diesen Klassen erzeugt wurden (dynamische Sicht). Ein Team enthält Objektkollaborationen, also Rollen, die zur Erfüllung einer Aufgabe in einem Team zusammengefaßt werden und dabei ihre Basisobjekte repräsentieren.

[4] In umgekehrter Richtung gilt diese Beziehung nicht, da einem Basisobjekt mehrere Rollenobjekte zugeordnet sein können. Ein Rollenobjekt besitzt dagegen immer genau ein Basisobjekt.

[5] Damit gleicht die Beziehung zwischen Rolle und Basis objektbasierter Vererbung.

[6] Unter einer Kollaboration versteht man eine Menge von Objekten, die zur Implementierung einer Aufgabe zusammenarbeiten.

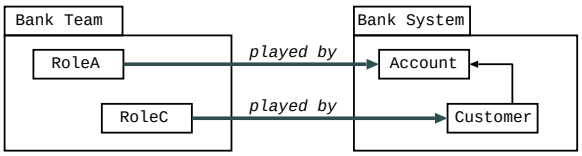


Abbildung 12.5: Beispiel: Bank und Bank-Team

Teams können wie normale Klassen behandelt werden, das schließt Instanziierung und Vererbung ein. Instanzen eines Teams sind Objektcontainer. Die Objekte innerhalb des Containers sind Rollenobjekte im Sinne der rollenbasierten Programmierung.

Basisobjekte für diese Rollen können wiederum Rollen innerhalb eines anderen Teams sein als auch Objekte, die von einer normalen Klasse erzeugt wurden. Ein Team kann gleichzeitig mehrere Klassen eines Basissystems adaptieren, indem im Team mehrere Rollenklassen enthalten sind, die jeweils eine der Basisklassen adaptieren. Eine Basisklasse kann innerhalb eines Teams verschiedene Rollen annehmen. Als Basissystem kann jede Applikation fungieren, die in der Zielsprache der ObjectTeams implementiert ist. Für ObjectTeams/Java bedeutet dies, daß das Basissystem auch als Bytecode gepackt in einem JAR-Archiv vorliegen kann.

Im Beispiel in Abbildung 12.5[7] wird an ein existierendes Banksystem (auf der rechten Seite dargestellt) ein Team (auf der linken Seite dargestellt) gebunden. Das Team enthält die Rollen `RoleA`, die von einem Objekt vom Typ `Account` gespielt wird, und die Rolle `RoleC`, die von einem Objekt vom Typ `Customer` gespielt wird.

Das ObjectTeams-Programmiermodell erlaubt das Einweben von zusätzlichem Code in das Basissystem in aspektorientierter Weise. Die Rollen stellen die Implementierung des Aspektes dar, das Team übernimmt das Aspektmanagement.

Das Einweben der Aspekte basiert auf der Call-In-Bindung von Methoden. Anders als aspektorientierte Sprachen wie AspectJ besitzt das ObjectTeams-Programmiermodell bisher keine ausdrucksstarke Joinpointsprache. Insbesondere fehlt es an Quantifizierungsmöglichkeiten, die von anderen Sprachen wie AspectJ z.B. durch Wildcardnotationen oder die Möglichkeit, Jointpoints mit Hilfe von Pointcuts zusammenzufassen, angeboten wird.
Es gibt drei Arten von Call-In-Bindungen:

- Der zusätzliche Code kann durch eine *Before-Methode* vor Ausführung einer Methode ausgeführt werden.

- Der zusätzliche Code kann durch eine *After-Methode* nach der Ausführung einer Methode ausgeführt werden.

- Der zusätzliche Code kann den Code der adaptierten Methode durch eine *Replace-Methode* komplett ersetzen.

Durch Call-Out-Bindung werden die Methoden des Basisobjekts im Rollenobjekt sichtbar gemacht, daß heißt, die Schnittstelle zum Basisobjekt muß durch Call-Out-Bindung explizit gemacht werden. Dabei werden Methoden der Rolle delegiert an Methoden des korrespondierenden Basisobjekts.

Die Rolle hat priviligierten Zugriff auf ihr korrespondierendes Basisobjekt, so daß durch Call-Out-Bindung alle Methoden des Basisobjekts aufgerufen werden können, auch private.

Call-In- und Call-Out-Bindung wird durch entsprechende Methoden realisiert, die an Methoden des korrespondierenden Basisobjekts gebunden werden. Die Call-In- und Call-Out-Methoden

[7] Zur Darstellung von ObjectTeams-Strukturen wird die UML-Erweiterung UML for Aspects (UFA) [46] verwendet. Das UML-Paketsymbol wird dabei für das Team verwendet, das UML-Klassensymbol für die Rollen. Es gibt die zusätzliche Relation `played by`, die die Zuordnung einer Rolle zu ihrer Basis beschreibt.

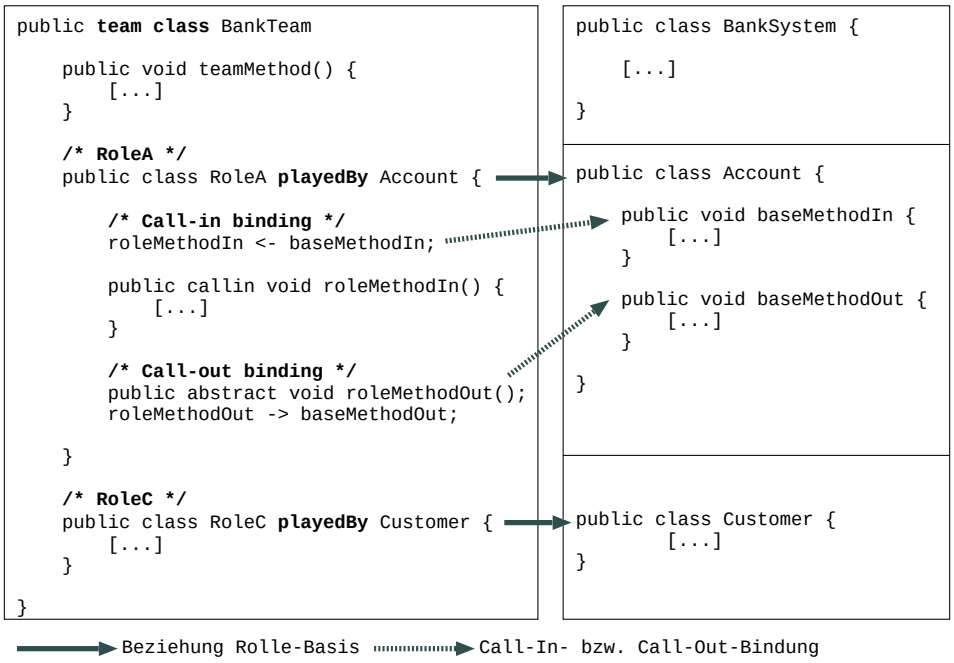


Abbildung 12.6: Beispiel: Code des Bank-Teams und der Klassen des Bank-Systems

können andere Parameter definieren als die korrespondierenden Methoden, dabei kann optional ein Parametermapping vorgenommen werden.

Das Einweben des Codes wird zur Ladezeit der Klassen vorgenommen. Das Basissystem muß deshalb nicht noch einmal für die Adaption kompiliert werden.

Zur Laufzeit werden Basisobjekte automatisch in ihre passenden Rollen gehoben (*Lifting*) und, wenn nötig, aus der Rolle zurückgewonnen (*Lowering*). Die Mechanismen Lifting und Lowering werden nie direkt vom Clientcode aufgerufen, sondern ausschließlich vom Laufzeitsystem zur Verfügung gestellt.

Das Bank-Team des Beispiels aus Abbildung 12.5 ist in Abbildung 12.6 als ObjectTeams/Java-Code implementiert. Teams sind Java-Klassen, die durch das Schlüsselwort `team` gekennzeichnet sind. Rollenklassen sind Klassen, die innerhalb des Teams deklariert sind, und werden mit dem Schlüsselwort `playedBy` an die Basisklassen gebunden. Call-In-Bindungen werden durch das Symbol `<-` notiert, Call-Out-Bindungen durch das Symbol `->`. Sowohl das Team als auch die Rollen besitzen im Beispiel Attribute und Methoden.

Teams können zur Laufzeit aktiviert und deaktiviert werden. Ist ein Team aktiviert, sind alle Call-In-Bindungen aktiv, wenn es deaktiviert ist, haben Call-In-Bindungen keinen Effekt auf das korrespondierende Basissytem. Im deaktivierten Zustand eines Teams reagiert das korrespondierende Basissytem, als wäre der zusätzliche Code nicht eingewebt worden.

Inaktive Teams halten ihren Zustand und ebenso den Zustand aller im Team enthaltenen Rollenobjekte. Nach der Aktivierung des Teams werden alle Basisobjekte wieder in ihre Rollen gehoben. Das Laufzeitsystem garantiert, daß ein Basisobjekt innerhalb eines Team immer in dieselbe Rolle gehoben wird. Der Zustand aller Rollen und der Zustand des Teams wird bei der Aktivierung wiederhergestellt.

Die Aktivierungsreihenfolge legt die Ausführungsreihenfolge der Call-In-Methoden von mehreren an ein System gebundenen Teams fest. Call-In-Methoden des zuletzt aktivierten Teams werden als erstes ausgeführt, wenn sie Before- oder Replace-Methoden sind, bei After-Methoden ist die Reihenfolge umgekehrt.

```
1  [public | ___] team class @TEAMNAME {
2
3     REPEAT TeamAttributes
4       [public | ___ | private] @TEAMATTRIBUTETYP @TEAMATTRIBUTE [ ___ | = @TEAMATTRIBUTEVALUE] ;
5     END_REPEAT TeamAttributes
6
7     REPEAT TeamMethods
8       [public | ___ | private] [@RETURN | ___] @TEAMMETHOD (@TEAMMETHODPARAMS) {
9           @TEAMMETHODSKEL
10      }
11    END_REPEAT TeamMethods
12
13
14    REPEAT Roles
15      [public | ___ ] class @ROLENAME playedBy @BASENAME{
16
17          REPEAT RoleAttributes
18            [public | ___ | private] @ROLEATTRIBUTETYP @ROLEATTRIBUTE [ ___ | = @ROLEATTRIBUTEVALUE] ;
19          END_REPEAT RoleAttributes
20
21          REPEAT RoleMethods
22            [public | ___ | private] [callin] [@RETURN | ___] @ROLEMETHOD (@ROLEMETHODPARAMS) {
23              @ROLEMETHODSKEL
24            }
25          END_REPEAT RoleMethods
26
27          REPEAT CallOutBinding
28            [public | ___ | private] abstract @RETURN @ROLEMETHOD (@ROLEMETHODPARAMS);
29            @ROLEMETHOD -> @BASEMETHOD;
30          END_REPEAT CallOutBinding
31
32          REPEAT CallInBinding
33            @ROLEMETHOD <- [before | after | replace] @BASEMETHOD;
34            [public | ___ | private] @RETURN @ROLEMETHOD (@ROLEMETHODPARAMS) {
35              @ROLEMETHODSKEL
36            }
37          END_REPEAT CallInBinding
38
39      }
40    END_REPEAT Roles
41
42 }
```

Abbildung 12.7: ObjectTeams/Java-Schablone

Neuere Erweiterungen des ObjectTeams-Programmiermodells erlauben eine implizite Aktivierung von Teams oder von einzelnen Rollen in einem Team durch ein Guard-Konzept. Dabei wird anhand einer Bedingung entschieden, ob ein Team oder eine Rolle in einem Team zu einem Zeitpunkt aktiv ist. Die Bedingung kann dabei vor allem auch den aktuellen Zustand des Basisobjekts oder des Teams berücksichtigen.

Eine weitergehende Einführung in die ObjectTeams findet sich in [47] und in der Sprachdefinition der Programmiersprache ObjectTeams/Java [83], in der die Beispiele in der vorliegenden Arbeit implementiert sind. Das Guard-Konzept des ObjectTeams-Programmiermodells wird in [48] beschrieben.

12.4 ObjectTeams-Schablone

UML-Modelle werden unter Verwendung eines Generators in ObjectTeams-Programmcode transformiert. Um aus Modellen ObjectTeams-Code zu erzeugen, wurde eine ObjectTeams/Java-Code-Schablone entworfen. In der Schablone sind feste Anteile bereits implementiert. Variable Anteile sind durch Platzhalter offengelassen und werden vom Generator durch die konkrete Implementie-

rung ersetzt.

In Abbildung 12.7 ist die Schablone schematisch dargestellt. Zwischen den Schlüsselwörtern `REPEAT` und `END_REPEAT` befinden sich Blöcke, die wiederholt werden. Dabei ist entweder keine, eine einmalige oder eine beliebige Wiederholung möglich. Optionen sind zwischen eckigen Klammern eingeschlossen und mit senkrechten Linien von einander abgegrenzt. Ein waagerechter Strich als Option bedeutet, daß in diesem Fall nichts in das Team eingefügt wird. Alle Variablen, wie Methodennamen oder Parametertypen, sind durch ein vorangestelltes `@` gekennzeichnet und kursiv geschrieben. Bei allen anderen Bestandteilen der Schablone handelt es sich um ObjectTeams/Java-Code.

Ein Team besitzt einen Kopf mit dem Namen des Teams (Zeile 1). Weitere Bestandteile eines Teams sind:

- Teamattribute und Teammethoden (Zeilen 3-11). Teammethoden ohne Rückgabewert sind Konstruktoren des Teams, wobei in diesem Fall `@TEAMMETHOD` = `@TEAMNAME` gilt.
- Rollen in beliebiger Anzahl (Zeilen 14-40).

Eine Rolle besitzt einen Kopf mit dem Namen der Rolle und dem Typ des Objekts, an das die Rolle gebunden ist (Zeile 15). Weitere Bestandteile einer Rolle innerhalb eines Teams sind:

- Rollenattribute (Zeilen 17-19).
- Normale Rollenmethoden, d.h. Methoden, die weder durch Call-In-Bindung noch durch Call-Out-Bindung gebunden sind (Zeilen 21-25). Rollenmethoden ohne Rückgabewert sind Konstruktoren der Rolle, wobei in diesem Fall `@ROLEMETHOD` = `@ROLENAME` gilt. Konstruktoren der Rolle werden in der Regel nicht gebraucht, da Rollen automatisch zu jedem Basisobjekt erzeugt werden.
- Call-Out-Bindungen (Zeilen 27-30). Diese gliedern sich in zwei Teile, die abstrakte Deklaration der Methode, die per Call-Out gebunden wird (Zeile 28), und die eigentliche Call-Out-Bindung (Zeile 29).
- Call-In-Bindungen (Zeile 32-37). Auch diese gliedern sich in zwei Teile, die eigentliche Call-In-Bindung mit Angabe der Bindungsart (Zeile 33), und die Implementierung der gebundenen Methode (Zeilen 34-36).

Kapitel 13

Aspektorientierte Testcodeintegration

In der Regel ist beim Testen zusätzlicher Testcode in das zu testende System zu integrieren. Im klassischen Testprozeß werden dazu Instrumentierungstechniken verwendet, die invasiv in den Quellcode eingreifen, d.h. direkt den Quellcode verändern.
Zusätzlicher Testcode hat verschiedene Aufgaben:

1. Initialisierung des Tests. Die zu testende Implementierung muß sich vor dem Testlauf in einem bestimmten Ausgangszustand befinden, der gesetzt werden muß.
2. Ausführen der Testfälle, eigentlicher Testlauf. Testdaten werden an die zu testende Implementierung gesendet.
3. Überwachung der zu testenden Implementierung. Das Antwortverhalten und die (Zwischen-) Zustände der zu testenden Implementierung werden eventuell bereits während des Testlaufs ausgewertet.
4. Protokollierung. Alle Ausgaben, (Zwischen-) Zustände und Testergebnisse werden gespeichert. Eventuell erfolgt die Testauswertung erst auf Basis des gespeicherten Protokolls nach dem Testlauf.

Die ersten beiden Aufgaben sind in der Regel Aufgaben des Testtreibers, die letzten beiden sind meist Aufgaben des Testorakels.

Unabhängig von der Testcodeintegration stellt man beim Testen fest, dass

- sich die zu testende Implementierung in einem Kontext befindet, von dem beim Test abstrahiert werden muß.
- in der Testphase (und nur dann) die Kapselung durchbrochen werden muß.
- Testcode nach der Testphase aus der zu testenden Implementierung entfernt werden muß.

Betrachtet man die oben gegebenen Anforderungen, bietet sich der Einsatz aspektorientierter Techniken zur Testcodeintegration an.

Aspektorientierte Programmiersprachen ermöglichen die Erweiterung von bestehendem Code (im Fall des Testens die zu testende Implementierung) um zusätzliche Funktionalität (im Falle des Testens der Testcode). Testcode ist so einfach und flexibel in die zu testende Implementierung einzufügen und zu entfernen.

Das Durchbrechen der Kapselung ist einfach zu realisieren, da Aspekte privilegierten Zugriff auf die zu testende Implementierung besitzen.

Die Unterscheidung zwischen Testkontext und anderen Kontexten der zu testenden Implementierung ist in einigen aspektorientierten Sprachen, z.B. AspectJ oder ObjectTeams, möglich, indem zusätzlicher Code entweder kontrollflußorientiert oder durch explizite Aktivierung zur Laufzeit ausgeführt wird.

Aspektorientierte Sprachen sind somit als ein mächtiges und flexibles Instrumentierungswerkzeug zu betrachten [113].

Viele Arbeiten gehen inzwischen davon aus, daß Testen ein Cross-Cutting-Concern bezüglich des zu testenden Systems ist. Der Einsatz aspektorientierter Techniken und Programmiersprachen wird deshalb im Testprozeß immer beliebter.
Dabei werden die unterschiedlichsten Wege bestritten:

- In den meisten Fällen werden aspektorientierte Techniken eingesetzt, um Testorakel in das zu testende System zu integrieren. Es lassen sich zwei verschiedene Ansätze erkennen: Die Integration zustandsbasierter Testorakel und die Integration von Zusicherungen. Beide Ansätze werden auch in der vorliegenden Arbeit verfolgt.

  Zustandsbasierte Testorakel als Aspekte zu realisieren werden von [16, 102] vorgeschlagen. Im Gegensatz zu der Arbeit in [102], auf der die vorliegende Arbeit aufbaut, wird in [16] die Hierarchie des Statecharts nicht durch mehrere Aspekte implementiert, sondern nur durch einen Aspekt. Zudem wird keine Aussage über Statechart-Eigenschaften wie Historie, Parallelität oder Nichtdeterminismus gemacht.

  Die Integration von Zusicherungen in das zu testende System mit Hilfe von Aspekten schlagen die Arbeiten in [13, 93, 113, 118] vor. Dabei reicht das Spektrum der Sprachen, in denen die Zusicherungen formuliert werden, von OCL [13, 93] über JML[1] [118] bis hin zu Java [113]. In den ersten beiden Fällen müssen die Zusicherungen zunächst durch einen Generator in Programmiersprachencode umgesetzt werden.

- Aspekte als Testtreiber zu benutzen, wird in [113] vorgeschlagen. Dabei fangen Aspekte jede Erzeugung eines Objekts ab, prüfen, ob ein Objekt dieses Typs bereits getestet wurde, und unterziehen es andernfalls einer Reihe von Testläufen.

- Mock-Objekte dienen in erster Linie der Testunterstützung. Wie in [68, 113] festgestellt wurde, sind sie sehr einfach als Aspekte zu realisieren.

- Das Monitoring von Laufzeitverhalten, speziell von Realzeit- und nebenläufigen Systemen, realisieren die Arbeiten in [23, 31, 74, 76] mit Hilfe aspektorientierter Programmiertechniken.

Einige Arbeiten konzentrieren sich vorrangig auf den Systemtest [23, 31, 74, 76], andere fokussieren auf den Unit-Test und bieten dabei meist Erweiterungen oder Reimplementierungen des populären Unit-Test-Framework JUnit [68, 113, 118]. Allen Arbeiten gemein ist, daß sie zusätzlichen Testcode in das zu testende System integrieren.

Die vorliegende Arbeit konzentriert sich auf den Einsatz aspektorientierter Techniken zur Integration des Testorakels in die zu testende Implementierung. Im Gegensatz zu den anderen Arbeiten nutzt die vorliegende Arbeit die aspektorientierte Programmiersprache ObjectTeams/Java. Die verwendete Sprache in anderen Arbeiten ist in der Regel AspectJ.
Vorteile hat ObjectTeams/Java insbesondere in Bezug auf die einfache Aktivierung und Deaktivierung von Aspekten, die hauptsächlich für die Implementierung der Zustandsdiagramme genutzt wird, als auch im Hinblick auf das Einweben der Aspekte zur Ladezeit.
Ansätze, die AspectJ als Programmiersprache benutzen, erfordern eine Neukompilierung der zu testenden Implementierung, wenn die Testaspekte wirken sollen. Es müssen zwei Versionen der zu testenden Implementierung verwaltet werden, eine mit Testcode und eine ohne Testcode. Dieses ist bei Verwendung von ObjectTeams-Java nicht erforderlich.

Da ObjectTeams im Gegensatz zu AspectJ neben der aspektorientierten auch rollenbasierte Programmierung unterstützt, werden zu testende Objekte und Testorakel als Basis und Rolle in einem Team zueinander in Beziehung gesetzt (siehe Abbildung 13.1).

[1] Java Modeling Language [66, 55].

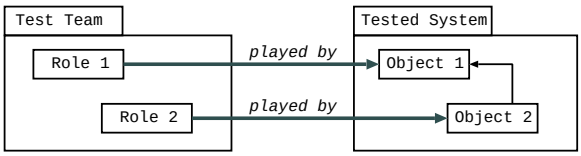


Abbildung 13.1: Korrespondenz zwischen Rollen und zu testenden Objekten

Definition: *Korrespondenz*
Jedem zu testenden Objekt ist eine ausführbare Spezifikation in Form eines ausführbaren Zustandsdiagramms oder in Form auswertbarer Zusicherungen zugeordnet. Diese Zuordnung wird in der vorliegenden Arbeit als Korrespondenz bezeichnet.

Das zu testende Objekt stellt die Basis für ein Team dar, die ausführbare Spezifikation die Rolle. Ein zu testendes Objekt ist einer oder mehreren Rollen zugeordnet, während eine Rolle immer genau zu einem zu testenden Objekt korrespondiert.

Es ergeben sich eine Reihe von Vorteilen, einerseits aus der Ausführbarkeit der Spezifikation, andererseits aus der Nutzung aspektorientierter Programmiertechniken:

- Eine ausführbare Spezifikation wird zur Laufzeit des Tests ausgeführt und wertet den Test bereits während des Testlaufs aus. Erwartete Werte werden nicht vor dem Test generiert, sondern abhängig von den aktuellen Eingaben und Zuständen. Das Beenden von negativen Testläufen ist schon beim Auftreten des Fehlers möglich, so daß nicht mit falschen Werten weitergerechnet wird. Eine Auswertung nach dem Testlauf entfällt.

- Aspektorientierte Programmiertechniken sind vorteilhaft insbesondere bei der Durchbrechung der Kapselung, einem der größten Probleme beim Test objektorientierter Software. Aspekte besitzen privilegierten Zugriff auf das zu testende Objekt und ermöglichen so die Überwachung interner Zustände. Unter Verwendung der ObjectTeams beschränkt sich der privilegierte Zugriff der Rolle auf seine korrespondierende Basis, was Vorteile in Bezug auf Zugriffschutz bietet, da nicht jedes beliebige Objekt im Testsystem auf alle Objekte im zu testenden System zugreifen darf.

  Darüber hinaus implementieren aspektorientierte Programmiersprachen bereits das Observer-Pattern [37] in ihrer Laufzeitumgebung. Eine Bindung von zu testendem Objekt und der Rolle ist leicht herzustellen. Die Rolle wird automatisch über alle Änderungen des zu testenden Objekts benachrichtigt (siehe dazu auch [112]).

Zusammenfassend läßt sich sagen, daß eine aspektorientierte Integration der in Kapitel 10 erzeugten Testorakel in Form einer ausführbaren Spezifikation überwiegend Vorteile bringt. Das Kapitel 14 beschreibt die Integration der Zustandsdiagramme in das zu testende System, das Kapitel 15 die Integration der OCL-Constraints. Eine Zusammenfassung der Ergebnisse der aspektorientierten Testcodeintegration findet sich in Kapitel 16.

Kapitel 14

Integration zustandsbasierter Testorakel

Die Integration der angereicherten Zustandsdiagramme aus Kapitel 10.1 in das zu testende System erfolgt durch die Implementierung in ObjectTeams und die Einbindung über ObjectTeams-Mechanismen.

Um Zustandsdiagramme zur Laufzeit auswerten zu können, ist die Implementierung als ausführbare Spezifikation notwendig. Ein ausführbares Zustandsdiagramm besteht in der vorliegenden Arbeit aus einem oder mehreren Team-Instanzen.

Je ein ausführbares Zustandsdiagramm wird an ein zu testendes Objekt gebunden. Es wertet zur Laufzeit sowohl die Korrektheit der Methodenaufrufe - d.h., ob der Aufruf der entsprechenden Methode im aktuellen Zustand erlaubt ist - als auch die Korrektheit des Folgezustands aus (siehe Abbildung 14.1). Dazu benachrichtigt das zu testende Objekt sein zugeordnetes Zustandsdiagramm über alle Methodenaufrufe unter Ausnutzung der ObjectTeams-Laufzeitumgebung.

Betrachtet man das zu testende System, so wird an jedes zu testende Objekt im System durch Ausnutzung der ObjectTeams-Laufzeitumgebung ein ausführbares Zustandsdiagramm gebunden, das sein korrespondierendes Objekt überwacht (Abbildung 14.2). Das bedeutet, daß bei einem Systemtest die Zustände einzelner Objekte im System nur unabhängig voneinander überwacht werden, da die Objekte im zu testenden System zwar miteinander kommunizieren, die Zustandsdiagramme im Testsystem jedoch nicht.

14.1 Unterstützung von Zustandsdiagrammeigenschaften

Zustandsdiagramme besitzen eine Reihe von Eigenschaften wie Hierarchie, Historie, Parallellität und Nichdeterminismen. Alle diese Eigenschaften müssen von einer Implementierung der Zustandsdiagramme möglichst umfassend unterstützt werden. Die folgenden Abschnitte geben auf der Basis eines Beispiels einen Einblick, wie Zustandsdiagramme unter Beachtung der oben genannten Eigenschaften in ObjectTeams transformiert werden.

Die Erläuterung der Technik erfolgt auf der Basis des generierten ObjectTeams/Java-Codes. Da es sich bei der aspektorientierten Programmierung um ein neues Programmierparadigma handelt, gibt es bisher wenig Beschreibungsmittel, um die vorgestellte Technik abstrakt zu beschreiben. Selbst die an ObjectTeams angepaßte UML-Erweiterung UFA [46] ist dazu nur bedingt geeignet, weshalb die Beispiele direkt auf der Basis des Codes erläutert werden. Wichtige Zeilen im Code sind in fetter Schrift hervorgehoben.

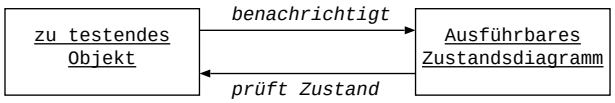


Abbildung 14.1: Ausführbares Zustandsdiagramm und zu testendes Objekt.

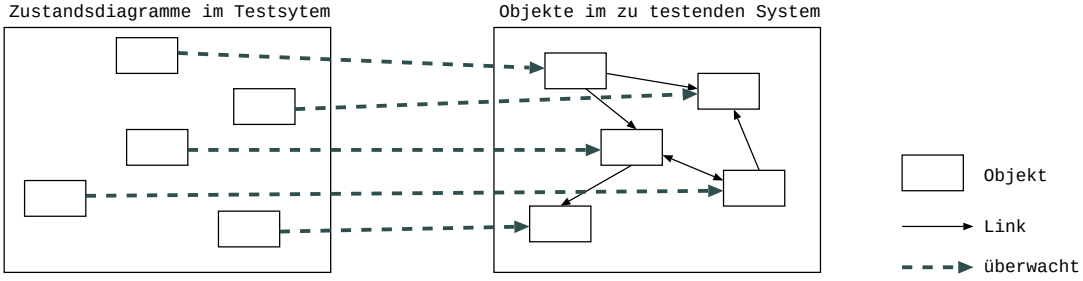


Abbildung 14.2: Ausführbare Zustandsdiagramme und zu testendes System.

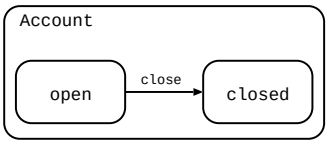


Abbildung 14.3: Beispiel: Einfaches Zustandsdiagramm

Einfache Zustandsdiagramme

Zunächst soll am Beispiel eines einfachen Zustandsdiagramms, wie es in Abbildung 14.3 gegeben ist, die grundlegende Idee der Implementierung eines Zustandsdiagramms als Team gegeben werden.

Ein einfaches Zustandsdiagramm besteht aus einem Toplevelzustand[1], im Beispiel der Zustand `Account`, der eine Menge von Basiszuständen enthält, im Beispiel die Zustände `open` und `closed`. Zwischen den Basiszuständen gibt es eine Reihe von Zustandsübergängen. Das Beispiel ist der Einfachheit halber auf einen Zustandsübergang beschränkt, der von der Methode `close` ausgelöst wird.

Die Implementierung als Team erfolgt nun in folgender Weise (der Code des Teams ist in Abbildung 14.4 auf Seite 119 dargestellt):

- Es wird ein Team mit dem Namen des Toplevelzustands und dem Zusatz "$SC" erzeugt. Im Beispiel heißt das Team `Account$SC`.

- Das Team enthält genau eine Rolle, die das Zustandsdiagramm implementiert und an das zu testende Objekt gebunden wird. Die Beschränkung auf genau eine Rolle ist notwendig, um den Teamaktivierungsmechanismus der ObjektTeams-Laufzeitumgebung auszunutzen[2].

  Die Rolle heißt immer wie der Typ des zu testenden Objekts mit dem Zusatz `$my`, also im Beispiel `$myAccount` und ist vom Typ `$Statechart` (Codezeile 3)[3]. Sie wird im Beispiel von einem zu testenden Objekt vom Typ `Account` gespielt (Codezeile 14). Das zu testende Objekt wird durch die Teammethode `$setRole` in seine Rolle geliftet (Codezeilen 9-10) und im Team bekanntgemacht.

  Die Codezeile `if (this == $myAccount)` bei allen Rollenmethoden ist notwendig, um zu verhindern, daß das Team Call-Ins von weiteren Objekten vom Typ `Account` verarbeitet (im Beispiel in den Codezeilen 20 und 29). Der Aktivierung-/Deaktivierungsmechanismus in der

[1] Der Toplevelzustand entspricht im Beispiel dem Wurzelzustand des Zustandsdiagramms. Da sich die vorgestellten Techniken rekursiv auf komponierte Unterzustände anwenden lassen, wird hier von Toplevelzustand gesprochen.

[2] Eine Aktivierung auf Rollenbasis ist erst in späteren Versionen von ObjectTeams möglich, die durch das Guard-Konzept [48] eine Aktivierung von Rollen unabhängig von einer Aktivierung des Teams zulassen. In der vorliegenden Arbeit wurde die ObjectTeams/Java-Version 0.6 benutzt, aktuell ist die Version 0.8.

[3] Mit Ausnahme von Teams, die parallele Zustände implementieren (s.u.).

```
 1 public team class Account$SC {
 2
 3   private $Statechart $myAccount;
 4
 5   public Account$SC() {
 6        this.activate();
 7   }
 8
 9   public void $setRole(Account as $Statechart asc) {
10        $myAccount = asc;
11   }
12
13 /* Role class */
14   class $Statechart playedBy Account {
15
16       private int $state = $OPEN;
17
18       /* Check state by comparing expected and current state */
19       void $checkState() {
20           if (this == $myAccount) {
21               if ($state == $CLOSED) {
22                   if ($getStatus() == Account.CLOSED) $logState ();
23                   else $logStateError ();
24               } else if ($state == $OPEN) {
22                   if ($getStatus() == Account.OPEN) $logState ();
24                   else $logStateError ();
25       } } }
26
27       /* Trigger events */
28       public void $close () {
29           if (this == $myAccount) {
30               $state = $CLOSED;
31               $checkState();
32       } }
33
34       /* Call-in bindings to update methods */
35       void $logMethodCall(int id) <- before void deposit(int amount)  with {id
36       void $logMethodCall(int id) <- before void withdraw(int amount) with {id
37       void $logMethodCall(int id) <- before void block()              with {id
38       void $logMethodCall(int id) <- before void unblock()            with {id
39       void $logMethodCall(int id) <- before void close()              with {id
40       $checkState <- after deposit, withdraw, block, unblock;
41       $close      <- after close;
42
43       /* Call-out bindings to observer methods */
44       abstract int $getBalance();
45       $getBalance  -> getBalance;
46       abstract int $getStatus();
47       $getStatus   -> getStatus;
48       abstract int $getHashCode();
49       $getHashCode -> hashCode;
50
51       /* Log Methods */
52       [...]
53   }
54 }
```

Abbildung 14.4: Implementierung eines einfachen Zustandsdiagramms

verwendeten Version der ObjectTeams ist zu schwach, um einem Team nur genau eine Rolle zuzuordnen.

- Die Basiszustände werden zu Werten, die dem Rollenattribut `$state` in der Rolle zugewiesen werden. Sie wird zu Beginn mit dem erwarteten Anfangszustand des zu testenden Objekts initialisiert, im Beispiel ist das der Zustand `open`. Im Beispiel findet sich die Deklaration und Initialisierung des Rollenattributs `$state` in Codezeile 16. Aus Gründen der Übersichtlichkeit wurde auf die Deklaration der Zustände (statische Variablen vom Typ `int`) verzichtet.

- Eine Methode `$checkstate` überprüft den Zustand des korrespondierenden Objekts anhand des in der Rolle gespeicherten Sollzustands.

- Die Transitionen werden zu Methoden, deren Namen dem Namen der auslösenden Methode mit dem Zusatz "$" entspricht. Dabei ist zu beachten, daß alle von einer Methode ausgelösten Transitionen in einer Methode zusammengefaßt werden. Im Beispiel wird die von der Methode `close` ausgelöste Transition in der Rollenmethode `$close` implementiert. Im Methodenrumpf der Methode `$close` wird der Zustand überprüft und der oder die Nachfolgezustände berechnet. Im Beispiel ist der Nachfolgezustand der Methode `$close` in jedem Fall der Zustand `closed` (Codezeilen 27-32).

- Schließlich werden die Rollenmethoden gebunden.

  Jeder Aufruf einer Update-Methode wird *vor* der Ausführung der Methode an die Rollenmethode `$logMethodCall` gebunden. Dabei wird jeder Update-Methode eine Id zugeordnet, um die aufgerufene Methode im Methodenrumpf von `$logMethodCall` zu identifizieren (Codezeilen 35-39). Die Rollenmethode `$logMethodCall` protokolliert den aktuellen Zustand und den Methodenaufruf. Der Quellcode von `$logMethodCall` ist aus Übersichlichkeitsgründen weggelassen worden.

  Nach jeder Transition wird die zur auslösenden Transition korrespondierende Methode ausgeführt. Die entsprechende Call-In-Bindung findet sich im Beispiel in Codezeile 41.

  Die Methode `$checkState` wird *nach* dem Aufruf jeder Update-Methode ausgeführt, um den Nachfolgezustand zu überprüfen. Die Bindung erfolgt entweder direkt durch Call-In-Bindung für alle Update-Methoden, die im aktuellen Zustandsdiagramm keine Zustandänderung auslösen (Codezeile 40) oder indirekt durch Aufruf aus einer transitionsauslösenden Methode (Codezeile 31). Zu beachten ist, daß alle Update-Methoden potentiell zustandsändernd sind. Deshalb werden alle diese Methoden per Call-In an die Zustandsüberprüfung gebunden.

  Aufgrund von Besonderheiten des ObjektTeams-Laufzeitsystems wird der Zustand nach Ausführung von Observer-Methoden nicht überprüft, da es sonst zu einer Endlosschleife kommt[4]. Observer-Methoden werden per Call-Out-Bindung an die korrespondierenden Methoden der Basis delegiert (Codezeilen 43-49).

Hierarchie

Abbildung 14.5 zeigt ein hierarchisches Zustandsdiagramm. Es gibt zwei Hierarchieebenen, den Toplevelzustand `Account` und den Sublevelzustand `open`.

Die Grundidee der Implementierung von hierarchischen Zustandsdiagrammen ist eine unabhängige Betrachtung der einzelnen Hierarchieebenen. Jede Hierarchieebene wird separat in einem Team implementiert (siehe Abbildung 14.7). Die Voraussetzung ist eine Abbildung von Modell auf Implementierung, d.h. Zustandsdiagramme sind jeweils einem Objekt zugeordnet und beschreiben dessen Lebenszyklus. Es gibt eine Korrespondenz zwischen aktiven Teams und der aktiven Zustandskonfiguration des zu testenden Objekts. Ein Team ist genau dann aktiv, wenn

[4]Da Observer-Methoden zum Vergleich des Ist- mit dem Sollzustand dienen, sollten sie zuvor mit Hilfe eines Teams getestet werden, das die Eigenschaft der Observermethoden überprüft, den Zustand nicht zu verändern. Die Generierung eines solchen Teams ist hier nicht im Einzelnen beschrieben, da sie sich einfach realisieren läßt, indem für jede Observer-Methode entsprechende Vor- und Nachbedingungen spezifiziert werden und anschließend das Verfahren aus Abschnitt 15 angewendet wird.

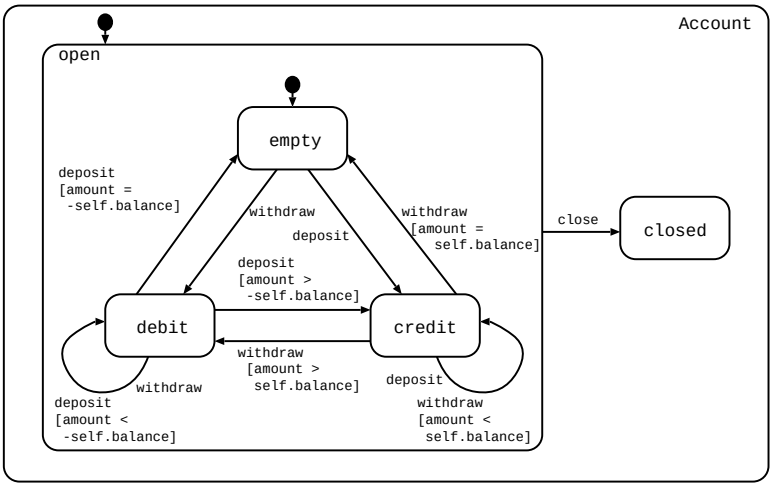


Abbildung 14.5: Beispiel: Hierarchisches Zustandsdiagramm

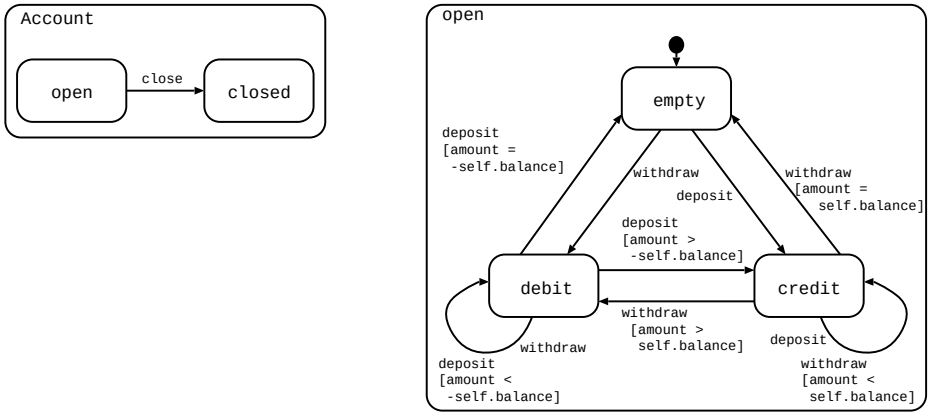


Abbildung 14.6: Beispiel: Hierarchisches Zustandsdiagramm mit Trennung der Hierarchieebenen

sich das zu testende Objekt in diesem Zustand befindet. Dabei handelt es sich um den erwarteten Zustand (Sollzustand) des zu testenden Objekts, nicht um den realen Zustand (Istzustand).

Zunächst werden die einzelnen Hierarchieebenen einzeln betrachtet, wie in Abbildung 14.6 illustriert. Im Beispiel sind das ein Team für den Toplevelzustand `Account` in der Teamklasse `Account$SC` und ein Team für den Sublevelzustand `open` in der Teamklasse `Open$SC`[5]. Das Team `Account$SC` repräsentiert den Toplevelzustand und ist damit immer aktiv. Das Team `Open$SC` ist genau dann aktiv, wenn sich das zu testende Objekt im Zustand `open` befindet.
Verantwortlichkeiten werden den einzelnen Hierarchielevels folgendermaßen zugeordnet:

- Jede Hierarchieebene ist zuständig für seine eigenen Zustände und Transitionen. Im Beispiel ist das Team `Account$SC` zuständig für die Transition, die von der Methode `close` ausgelöst wird, das Team `Open$SC` ist zuständig für die von den Methoden `deposit` und `withdraw` ausgelösten Transitionen. Die Implementierung des Teams `Account$SC` befindet sich in Abbildung 14.8 auf Seite 123 und die Implementierung des Teams `Open$SC` befindet sich in Abbildung 14.9 auf Seite 124.

[5]In der realen Implementierung wird das Team mit seinem vollständigen Namen benannt: `Account$open$SC`, da mehrere Zustände mit dem gleichen Namen, aber unterschiedlichen übergeordneten Zuständen möglich sind. Das Beispiel ist so gewählt, daß dieser Fall nicht auftritt, deshalb wird der Übersichtlichkeit wegen der kurze Name benutzt.

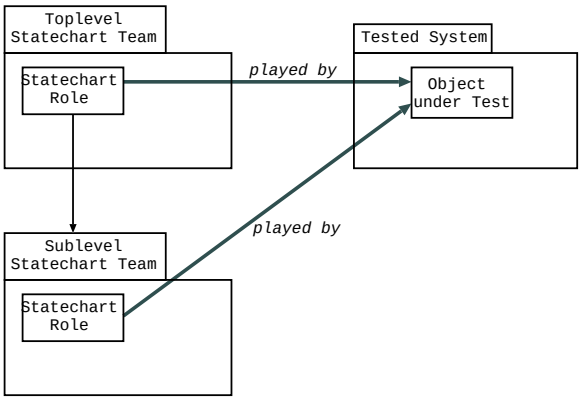


Abbildung 14.7: Korrespondenz zwischen hierarchischem Zustandsdiagramm und zu testendem Objekt

- Jede Hierarchieebene ist für die in ihr enthaltenen Sublevelzustände insofern zuständig, als daß diese von ihrem übergeordneten Zustand initialisiert, aktiviert und deaktiviert werden. Dazu hält jeder übergeordnete Zustand eine Referenz auf seine Sublevelzustände.

- Nicht immer ist eine Zuordnung zu einzelnen Hierarchieebenen einfach möglich, z.B. bei Interleveltransitionen. Bei Interleveltranstionen ist der übergeordnete Zustand zuständig für die Aktivierung. Er aktiviert bzw. deaktiviert den Unterzustand, je nachdem, ob der Unterzustand Ziel oder Quelle der Transition ist. Für die Überprüfung des korrekten Zustands ist der Quellzustand vor der Ausführung und der Zielzustand nach der Ausführung der Methode zuständig.

- Die Implementierung der einzelnen Hierarchieebenen erfolgt nun unabhängig voneinander, indem jede Hierarchieebene als ein einfaches Zustandsdiagramm betrachtet wird.

Vergleicht man die Implementierung des einfachen Zustandsdiagramms aus Abbildung 14.4 mit der des Toplevelzustands aus Abbildung 14.8, so lassen sich folglich nur wenige Unterschiede ausmachen. Der zusätzliche Code für die Verwaltung des Sublevelzustands `open` im Team des Toplevelzustands `Account` ist in Abbildung 14.8 mit den Pfeilen markiert.
Die Veränderungen zur Implementierung eines einfachen Zustandsdiagramms sind im Einzelnen (vgl. Abbildungen 14.4 und 14.8):

- Ein Rollenattribut hält eine Referenz auf das Team des Sublevelzustands (Codezeile 18). Im Beispiel ist dieses Team vom Typ `Open$SC`.

- In der Teammethode `$setRole` wird zusätzlich die Rollenmethode `$init` aufgerufen (Codezeile 11). Die Methode `$init` (Codezeilen 20-24) erzeugt die Teaminstanz für den Sublevelzustand und übergibt das eigene zu testende Objekt mit Hilfe der Methode `$setRole` des Teams `Open$SC` an dieses Team.

- Wenn der Zustand `open` durch den Aufruf der Methode `close` verlassen wird, wird das Team für den Sublevelzustand deaktiviert. So wird verhindert, daß dieses Team weiterhin Nachrichten an das zu testende Objekt verarbeitet.

Wichtig ist, daß die Rollen in den Teams, die die verschiedenen Hierarchieebenen repräsentieren, von dem selben zu testenden Objekt gespielt werden. Das ist nötig, damit beide Teams auf dasselbe zu testende Objekt verweisen und somit zum selben Objekt korrespondieren. Diese Eigenschaft wird sichergestellt durch die Methode `$setRole`.

Wird die Methode `$setRole` von der Rolle `$Statechart` im Team `$AccountSC` aufgerufen, gibt das Statechartobjekt eine Referenz auf sich selbst an das Team `$OpenSC`. Die Methode `$setRole`

```
 1 public team class Account$SC {
 2
 3   private $Statechart $myAccount;
 4
 5   public Account$SC() {
 6        this.activate();
 7   }
 8
 9   public void $setRole(Account as $Statechart asc) {
10        $myAccount = asc;
11        $myAccount.$init();
12   }
13
14 /* Role class */
15   class $Statechart playedBy Account {
16
17       private int $state = $OPEN;
18       private Open$SC $openState;
19
20       /* Initialize Role */
21       private void $init() {
22           $openState = new Open$SC();
23           $openState.$setRole(this);
24       }
25
26       /* Check state by comparing expected and current state */
27       void $checkState() {
28           if (this == $myAccount) {
29               if ($state == $CLOSED) {
30                   if ($isClosed()) $logState ();
31                   else $logStateError ();
32               } else if ($state == $OPEN) {
33                   if ($getStatus() == Account.OPEN) $logState ();
34                   else $logStateError ();
35       } } }
36
37       /* Trigger events */
38       public void $close () {
39           if (this == $myAccount) {
40               $openState.deactivate();
41               $state = $CLOSED;
42               $checkState();
43       } }
44
45       /* Call-in bindings to update methods */
46       void $logMethodCall(int id) <- before void deposit(int amount)  with {id
47       void $logMethodCall(int id) <- before void withdraw(int amount) with {id
48       void $logMethodCall(int id) <- before void block()              with {id
49       void $logMethodCall(int id) <- before void unblock()            with {id
50       void $logMethodCall(int id) <- before void close()              with {id
51       $checkState <- after deposit, withdraw, block, unblock;
52       $close      <- after close;
53
54       /* Call-out bindings to observer methods */
55       abstract int $getBalance();
56       $getBalance  -> getBalance;
57       abstract int $getStatus();
58       $getStatus   -> getStatus;
59       abstract int $getHashCode();
60       $getHashCode -> hashCode;
61
62       /* Log Methods */
63       [...]
64   }
65 }
```

Abbildung 14.8: Implementierung des Toplevelzustands `Account`

```
 1 public team class Open$SC {
 2
 3    private $Statechart $myAccount;
 4
 5    public Open$SC () {...}
 6
 7    public void $setRole(Account as $Statechart asc) {
 8        $myAccount = asc;
 9    }
10
11 /* Role class */
12    class $Statechart playedBy Account {
13
14        private int $state = $EMPTY;
15
16        void $checkState() {...}
17
18        /* Trigger events */
19        public void $deposit (int amount) {...}
20        public void $withdraw (int amount) {...}
21
22        /* Call-in bindings to update methods */
23        void $logMethodCall(int id) <- before void deposit(int amount)  with {id <
24        void $logMethodCall(int id) <- before void withdraw(int amount) with {id <
25        void $logMethodCall(int id) <- before void block()              with {id <
26        void $logMethodCall(int id) <- before void unblock()            with {id <
27        void $logMethodCall(int id) <- before void close()              with {id <
28        $checkState <- after block, unblock, close;
29        $deposit    <- after deposit;
30        $withdraw   <- after withdraw;
31
32        /* Call-out bindings to observer methods */
33        abstract int $getBalance();
34        getBalance  -> getBalance;
35        abstract int $getStatus();
36        $getStatus   -> getStatus;
37        abstract int getHashCode();
38        getHashCode -> hashCode;
39
40        /* Log Methods */
41        [...]
42    }
43 }
```

Abbildung 14.9: Implementierung des Sublevelzustands `open`

im Team `$OpenSC` erwartet allerdings ein Objekt vom Typ `Account`, nicht ein Objekt vom Typ Statechart des Teams `$AccountSC`. Also wird aus dem Objekt vom Typ Statechart des Teams `$AccountSC` automatisch durch Lowering sein korrespondierendes Basisobjekt gewonnen und an `$setRole` übergeben.

Die Methode `$setRole` liftet anschließend das Basisobjekt vom Typ `Account` in seine Rolle im Team `$OpenSC`. Dies wird ausgedrückt durch die Signatur von `$setRole` im Team `$AccountSC`:

```
Account as $Statechart asc.
```

Die Implementierung des Teams für den Sublevelzustand `open` (siehe Abbildung 14.9 auf Seite 124) entspricht der Implementierung des einfachen Zustandsdiagramms aus Abbildung 14.4 auf Seite 119 mit dem Unterschied, daß andere Update-Methoden (hier: `deposit`, `withdraw`) und andere Zustände (hier: `empty`, `debit`, `credit`) behandelt werden. Enthalten Unterzustände wiederum selbst weitere Unterzustände, werden sie analog zu Abbildung 14.8 auf Seite 123 implementiert.

Durch die Technik bisher nicht abgedeckt ist die Priorisierung bei Konflikten zwischen Transitionen auf verschiedenen Hierarchieebenen. Zwar ist durch die Aktivierungsreihenfolge sichergestellt, daß zunächst die innerste Transition ausgeführt wird, wie es die UML-Spezifikation definiert. Dann wird aber auch die Transition in einem darüberliegenden Zustand ausgewertet.

Abhilfe kann hier das Guard-Konzept in der neuen Version des ObjectTeams/Java-Systems schaffen. Hier ist es möglich, Abhängigkeiten auf einfache Weise zu definieren und auszuwerten. So ist geplant, die schaltende Transition in Form eines Flags zu vermerken und dieses Flag durch Guards in den darüberliegenden Teams auszuwerten.

Verlassen von geschachtelten Zuständen

Beim Betreten eines Zustandes werden durch die Teammethode `$setRole` und die Rollenmethode `$init`, die durch diese aufgerufen wird, die inneren Zustände ebenfalls betreten, indem die korrespondierenden Teams aktiviert werden.

Beim Verlassen eines Zustandes, der wiederum Unterzustände besitzt, muß eine Deaktivierung aller Unterzustände bzw. aller zu den Unterzuständen korrespondierenden Teams erfolgen. Die Implementierung dieser Deaktivierung nutzt eine Eigenschaft des ObjectTeams-Mechanismus zur Team-Deaktivierung. Zur Aktivierung bzw. Deaktivierung stellt die Superklasse aller Teams in ObjectTeams/Java, die Klasse `org.objectteams.Team`[6] sowie die Methoden `activate` und `deactivate` bereit. Diese Methoden lassen sich in selbst implementierten Teams redefinieren.

Um alle Teams zu deaktivieren, die einen Unterzustand eines Zustandes implementieren, wird die Methode `deactivate` in dem betreffenden Zustand so redefiniert, daß in ihr zunächst die Methode `deactivate` des gerade aktiven Teams des Unterzustandes aufgerufen wird. Besitzt dieses wiederum einen geschachtelten Unterzustand, so wird auch hier die Methode `deactivate` redefiniert. Um den normalen Deaktivierungsmechanismus von ObjectTeams/Java zu nutzen, der durch die Redefinition außer Kraft gesetzt wurde, wird nach der Deaktivierung des aktiven Teams des Unterzustandes die Methode `deactivate` der Klasse `org.objectteams.Team` durch den Aufruf `super.deactivate()` ausgeführt.

Abbildung 14.10 zeigt dieses Verhalten anhand eines Codefragments einer Implementierung des Zustandes `Account`. Die Redefinition der Teammethode `deactivate` findet sich in den Codezeilen 14-17. Die Methode `deactivate` delegiert die Deaktivierung des Teams für den Unterzustand an die Rollenmethode `$deactivate` (Codezeile 15).

Die Rollenmethode `$deactivate`, implementiert in den Codezeilen 31-35, prüft zunächst, ob das Team für den Zustand `open` aktiv ist (Codezeile 33). Ist dies der Fall, so wird an die zum Zustand `open` korrespondierende Teaminstanz `open$State` der Aufruf zur Deaktivierung weiter-

[6] Die Klasse `org.objectteams.Team` gehört zur ObjectTeams/Java-Standardbibliothek und ist implizite Superklasse aller Teamklassen. Sie hat eine ähnliche Funktion wie die Klasse `Object` als Superklasse aller anderen Klassen in Java.

```
 1 public team class Account$SC {
 2
 3   private $Statechart $myAccount;
 4
 5   public Account$SC() {
 6        this.activate();
 7   }
 8
 9   public void $setRole(Account as $Statechart as
10        $myAccount = asc;
11        $myAccount.$init();
12   }
13
14   public void deactivate() {
15        $myAccount.$deactivate();
16        super.deactivate();
17   }
18
19 /* Role class */
20   class $Statechart playedBy Account {
21
22       private int $state = $OPEN;
23       private Open$SC $openState;
24
25       /* Initialize Role */
26       private void $init() {
27           $openState = new Open$SC();
28           $openState.$setRole(this);
29       }
30
31       /* Deactivate active subchart */
32       private void $deactivate() {
33           if ($state == $OPEN) {
34               $openState.deactivate();
35       }}
36
37       [...]
38   }
39 }
```

Abbildung 14.10: Deaktivierung des aktiven Unterzustandes von `Account`

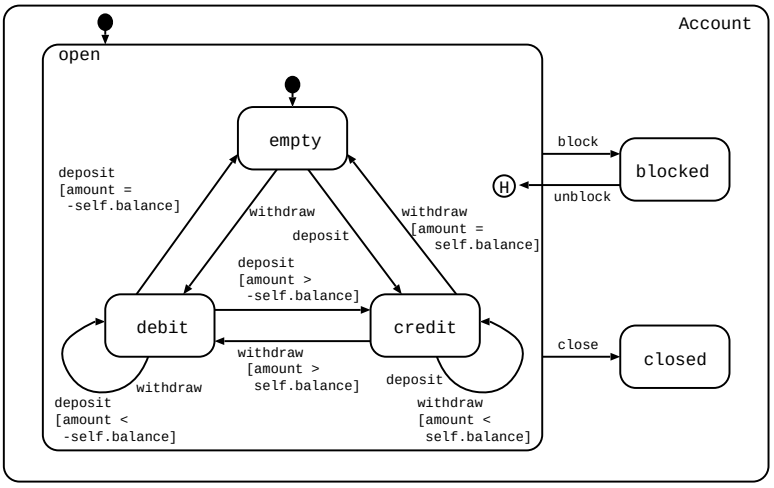


Abbildung 14.11: Beispiel: Hierarchisches Zustandsdiagramm mit Historie

geleitet (Codezeile 34). Alternativ ist es möglich, die Abfrage des aktuell aktiven Teams für den Unterzustand zu ersparen und auf allen Teaminstanzen für alle Unterzustände die Deaktivierungsmethode `deactivate` aufzurufen, da dies bei einem bereits deaktivierten Team keine Auswirkungen hat.

Historie

Viele Testansätze, die Zustandsdiagramme als Basis für den Test verwenden, bieten keine Lösung für den History-Connector an (siehe z.B. [10, 16]), so daß unklar ist, wie und ob Zustandsdiagramme mit History-Connector vom dargestellten Ansatz unterstützt werden.

Die hier vorgestellte Technik stellt einen einfachen Weg vor, wie Zustandsdiagramme mit History-Connector implementiert werden.

Da ein Team seinen Zustand hält, während es deaktiviert ist, wird auch der Subzustand gemerkt, in dem sich das System befand, bevor das Team verlassen wurde. Der History-Connector wird damit implizit vom ObjectTeams-Laufzeitsystem zur Verfügung gestellt, ohne daß zusätzliche Speichermechanismen implementiert werden müssen. Deshalb ist die Implementierung, wie sie in Abbildung 14.9 auf Seite 124 dargestellt ist, auf die Implementierung des in Abbildung 14.11 gezeigten Zustandsdiagramms übertragbar[7].

Gleichzeitig ist jetzt die Implementierung von Zustandsdiagrammen ohne Historie die schwierigere Aufgabe. Um den Zustand `open` des Zustandsdiagramms ohne Historie als Team zu implementieren, muß die Implementierung aus Abbildung 14.9 erweitert werden.

Es wurde bereits eine Rollenmethode vorgestellt, die die Aufgabe besitzt, Zustände beim ersten Betreten zu initialisieren: die Methode `$init`. Um die Historie auszuschalten, wird das Team `Open$SC` um die Rollenmethode `$init` ergänzt. Die Implementierung der Methode `$init` ist in Abbildung 14.12 auf Seite 128 in den Codezeilen 18-20 gegeben.

Parallelität

Für parallele Zustände bieten sich zwei Varianten als Implementierung an:

1. Die unabhängige Implementierung in zwei Teams.
2. Die Implementierung in einem Team als zwei verschiedene Rollen desselben zu testenden Objekts.

[7]Da der Zustand `open` im Zustandsdiagramm in Abbildung 14.9 nach dem Verlassen nicht mehr betreten wird, ist die gegebene Implementierung dennoch korrekt. In diesem speziellen Fall spielt es keine Rolle, ob der Zustand von `open` vor dem Verlassen gespeichert wird oder nicht.

```
 1 public team class Open$SC {
 2
 3    private $Statechart $myAccount;
 4
 5    public Open$SC () {...}
 6
 7    public void $setRole(Account as $Statechart asc) {
 8        $myAccount = asc;
 9        $myAccount.$init();
10    }
11
12 /* Role class */
13    class $Statechart playedBy Account {
14
15        private int $state = $EMPTY;
16
17       /* Initialize Role */
18       private void $init() {
19           $state = $EMPTY;
20       }
21
22        void $checkState() {...}
23
24        /* Trigger events */
25        public void $deposit (int amount) {...}
26        public void $withdraw (int amount) {...}
27
28        /* Call-in bindings to update methods */
29        void $logMethodCall(int id) <- before void deposit(int amount)  with {id <
30        void $logMethodCall(int id) <- before void withdraw(int amount) with {id <
31        void $logMethodCall(int id) <- before void block()              with {id <
32        void $logMethodCall(int id) <- before void unblock()            with {id <
33        void $logMethodCall(int id) <- before void close()              with {id <
34        $checkState <- after block, unblock, close;
35        $deposit    <- after deposit;
36        $withdraw   <- after withdraw;
37
38        /* Call-out bindings to observer methods */
39        abstract int $getBalance();
40        getBalance  -> getBalance;
41        abstract int $getStatus();
42        $getStatus   -> getStatus;
43        abstract int getHashCode();
44        getHashCode -> hashCode;
45
46        /* Log Methods */
47        [...]
48    }
49 }
```

Abbildung 14.12: Implementierung des Sublevelzustands `open` ohne Historie

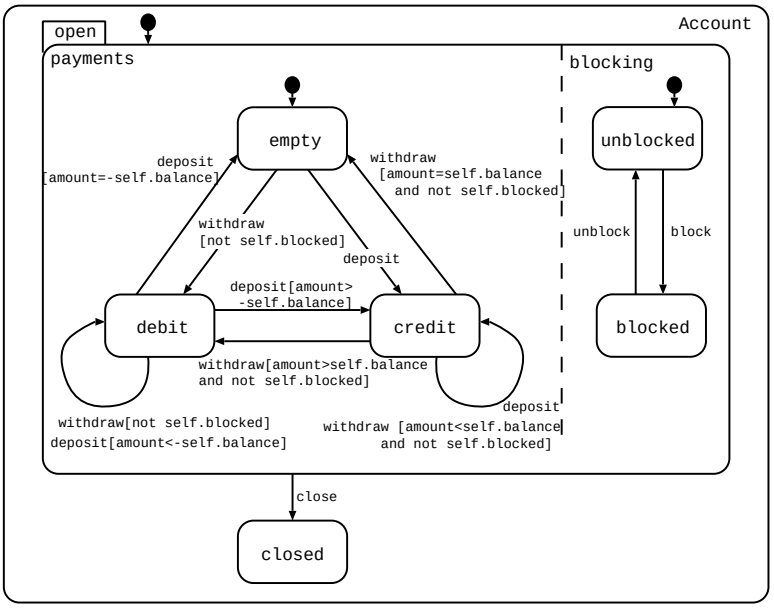


Abbildung 14.13: Beispiel: Hierarchisches Zustandsdiagramm mit Parallelität

In der vorliegenden Arbeit wurde die zweite Variante gewählt. Dies hat den Vorteil, daß das gleichzeitige Betreten und Verlassen des Zustands mit den parallelen Unterzuständen in einem Team verwaltet wird. Zudem reduziert sich die Anzahl der Teams.

Es ergibt sich allerdings ein Problem mit der aktuellen Version der ObjectTeam/Java-Implementierung. Bisher müssen die Ereignisse in den parallelen Zuständen unabhängig voneinander sein, da es bisher nicht möglich ist, dasselbe Ereignis in Form eines Methodenaufrufs in der gleichen Weise in verschiedenen Rollen im selben Team zu binden. Es fehlt insbesondere an einer Prioritätsauswahl. Dies ist jedoch nur eine vorübergehende Einschränkung, da für zukünftige Versionen von ObjectTeams/Java auch die Verarbeitung desselben Methodenaufrufs an das Basisobjekt in mehreren Rollen innerhalb eines Teams vorgesehen ist.

Die Abbildung 14.13 zeigt einen Zustand (`open`) mit zwei parallelen Unterzuständen, dem Zustand `payments` und dem Zustand `blocking`. Beide Zustände sind unabhängig voneinander in Bezug auf die transitionsauslösenden Methoden spezifiziert, jedoch durch den Wert des Attributs `blocked` voneinander abhängig.

Die Implementierung des Zustands `open` mit seinen parallelen Unterzuständen findet sich in Abbildung 14.14 auf Seite 130. Markiert durch die dicken Pfeile ist das Auftreten der Methode `$checkState`.

Der Toplevelzustand wird repräsentiert durch die Rolle `$Statechart`. Diese Rolle ist notwendig, da ObjectTeams/Java die Bindung desselben Methodenaufrufs an verschiedene Rollen im derselben Teaminstanz nicht erlaubt. Die beiden Sublevelzustände werden in den Rollen `Payments$Statechart` und `Blocking$Statechart` verwaltet. Bei Nutzung von späteren Versionen von ObjectTeams entfällt die Rolle `$Statechart` und die Unterzustände werden wie oben beschrieben als normale Zustände implementiert. Die Verantwortlichkeiten werden so zugeordnet, daß keine Konflikte entstehen:

- Die zum Toplevelzustand korrespondierende Rolle enthält die Methode `$checkState`, die die Zustandsprüfung der `$checkState`-Methoden der zu den Sublevelzuständen korrespondierenden Rollen aufruft (Codezeilen 33-36). Die eigentliche Zustandsprüfung erfolgt in den Rollen der Sublevelzustände (Codezeile 46 bzw. 58), während die Bindung an die Methodenaufrufe in der Rolle des Toplevelzustands erfolgt. Im Beispiel ist dies nur der Methodenaufruf `close`, da alle anderen Update-Methoden Transitionen in den Unterzuständen auslösen.

```
 1 public team class Open$SC {
 2
 3    private $Statechart $myAccount;
 4    private Payments$Statechart $myPaymentAccount;
 5    private Blocking$Statechart $myBlockingAccount;
 6
 7    public $OpenSC () {
 8        this.activate();
 9    }
10
11    public void $setRole(Account asc) {
12        $setToplevelRole(asc);
13        $setPaymentsRole(asc);
14        $setBlockingRole(asc);
15    }
16
17    public void $setToplevelRole(Account as $Statechart asc) {
18        $myAccount = asc;
19    }
20
21    public void $setPaymentsRole(Account as Payments$Statechart asc) {
22        $myPaymentAccount = asc;
23    }
24
25    public void $setBlockingRole(Account as Blocking$Statechart asc) {
26        $myBlockingAccount = asc;
27
28    }
29
30 /* Role class for toplevel statechart */
31    class $Statechart playedBy Account {
32
33       void $checkState() {                    <-
34           $myPaymentAccount.$checkState();
35           $myBlockingAccount.$checkState();
36       }
37
38       /* Call-in bindings of update methods to $checkState */
39       [...]                                     // Log methods
40       $checkState <- after close;             <-
41    }
42
43 /* Role class for sublevel statechart payments */
44    class Payments$Statechart playedBy Account {
45
46       void $checkState() { [...] }            <-
47       void $deposit(int amount) { [...] }
47       void $withdraw(int amount) { [...] }
48
49       /* Call-in bindings to update methods */
50       $deposit      <- after deposit;
51       $withdraw     <- after withdraw;
52       [...]
53    }
54
55 /* Role class for sublevel statechart blocking */
56    class Blocking$Statechart playedBy Account {
57
58       void $checkState() { [...] }            <-
59       void $block() { [...] }
60       void $unblock() {
61           if (this == $myBlockingAccount) {
62               if ($state == $BLOCKED) { $state = $UNBLOCKED; } else { [
63               $myAccount.$checkState();
64         }
65
66       /* Call-in bindings to update methods */
67       $unblock      <- after unblock;
68       $block        <- after block;
68       [...]
70 } }
```

Abbildung 14.14: Implementierung des Zustands `open` mit parallelen Unterzuständen

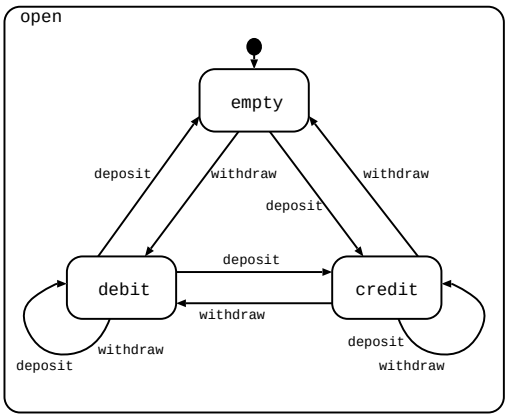


Abbildung 14.15: Beispiel: Zustandsdiagramm mit (lösbarem) Nichtdeterminismus

Daneben sind alle Bindungen, die für das Logging erstellt wurden, in die zum Toplevelzustand korrespondierende Rolle ausgelagert.

- Die zu den Sublevelzuständen korrespondierenden Rollen `Payments$Statechart` und `Blocking$Statechart` implementieren jeweils die Bindungen zu den Update-Methoden, die in diesen Zuständen Transitionen auslösen. Als Beispiel ist die (vereinfachte) Implementierung der Rollenmethode `$unblock` angegeben (Codezeilen 60-64). Nach der Ausführung der Transition wird die Zustandsprüfung aufgerufen (Codezeile 63). Da beide parallelen Zustände überprüft werden müssen, wird zunächst die Methode `$checkState` der zum Toplevelzustand korrespondierenden Rolle aus Codezeile 33 aufgerufen, die die Zustandsüberprüfung wiederum an die beiden anderen Rollen delegiert.

Nichtdeterminismus

Nicht jede Art von Nichtdeterminismus wird durch die in der vorliegenden Arbeit vorgeschlagene Implementierungsstrategie behandelt. Im Testen ist eine Unterscheidung zwischen lösbarem Nichtdeterminismus und unlösbarem Nichtdeterminismus sinnvoll[8].

Definition: *Lösbarer Nichtdeterminismus*
Wenn eine Implementierung eine nichtdetermistische Auswahl aus einer bekannten Menge von möglichen Folgeschritten trifft, dann wird dieser Nichtdeterminismus als lösbar bezeichnet, wenn nach dem Schritt eine Ermittlung des Folgezustands möglich ist.

Das bedeutet, daß zwar der Schritt unbekannt ist, der Folgezustand jedoch bekannt ist, entweder weil alle Schritte in denselben Folgezustand führen oder weil der Folgezustand auf Grund eines Vergleichs der Implementierung mit einer Referenz, z.B. der Spezifikation, zu berechnen ist.
Im Falle eines Zustandsdiagramms bedeutet dies, daß es in einem Schritt mehrere konfliktfreie

Transitionsmengen gibt, die potentiell schalten. Die ausgewählte Transitionsmenge ist unbekannt. Nach der Ausführung der Transitionen der unbekannten Transitionsmenge, die ausgewählt wurde, gibt es eine Menge potentieller Nachfolgezustände, da die Menge aller potentiell schaltenden Transitionsmengen bekannt war.

Der reale Nachfolgezustand wird ermittelt, indem alle potentiellen Nachfolgezustände mit dem realen Zustand verglichen werden, z.B. anhand der spezifizierten Zustandsinvarianten. Wenn sich die Menge aller potentiellen Nachfolgezustände auf einen Zustand reduzieren läßt, dann spricht man von lösbarem Nichtdeterminismus.

[8]Dieser Unterscheidung wurde u.a. auf einem Dagstuhl-Seminar zum Thema modellbasiertes Testen diskutiert [21].

```
 1 public team class Open$SC {
 2
 3 [...]
 4
 5 /* Role class */
 6    class $Statechart playedBy Account {
 7
 8        private int $state = $EMPTY;
 9        private Vector $pos_states = new Vector();
10
11        void $checkState() {
12            if ($state = $UNDEF) {
13                for (int i = 0; i < $pos_states.size(); i++) {
14                    int $act_state = ((Integer) $pos_states.elementAt(i)).intValue();;
15                    if ($act_state == $EMPTY && $getBalance() == 0) {
16                        $state = $EMPTY;
17                    } else if ($act_state == $CREDIT && $getBalance() > 0) {
18                        $state = $CREDIT;
19                    } else if ($act_state == $DEBIT && $getBalance() < 0) {
20                        $state = $DEBIT;
21                }   }
22                if (state == $UNDEF) {
23                    $logStateError ();
24                } else {
25                   $logState ();
26            }   }
27            else {
28               [...]            // normal state checking
29        }   }
30
31        /* Trigger events */
33        public void $deposit (int amount) {...}
34
35        public void $withdraw (int amount) {
36            if ($state == $EMPTY) {
37               $state = $DEBIT;
38            }
39            else if ($state == $CREDIT) {
40               $pos_states = new Vector();
41               $state = $UNDEF;
42               $pos_states.add(new Integer($EMPTY));
43               $pos_states.add(new Integer($CREDIT));
44               $pos_states.add(new Integer($DEBIT));
45            }
46            $checkState();
47        }
48
49        /* Call-in bindings to update methods */
50        [...]
51
52        /* Call-out bindings to observer methods */
53        [...]
54
55        /* Log Methods */
56        [...]
57    }
58 }
```

Abbildung 14.16: Implementierung des Zustands `open` mit nichtdeterministischer Auswahl der Transitionen

Das Beispiel in Abbildung 14.15 zeigt ein nichtdeterministisches Zustandsdiagramm. Ein Aufruf der Methode `deposit` oder `withdraw` führt im Zustand `debit` respektive `credit` zu nichtdeterministischem Verhalten. Das nichtdeterministische Verhalten ist jedoch lösbar, da die Zustandsinvarianten der Nachfolgezustände disjunkt sind und so der Zustand nach der Ausführung der Methoden anhand des realen Zustands des zu testenden Objekts ermittelbar ist.

Die Idee der Implementierung ist, sich alle potentiellen Zustände zu merken, die nach einem Methodenaufruf angenommen werden können. Die Implementierung in Abbildung 14.16 veranschaulicht dies anhand der Transitionen, die von der Methode `withdraw` ausgelöst werden (Codezeilen 35-47).

Wird `withdraw` im Zustand `credit` aufgerufen, so gibt es drei potentielle Nachzustände, `debit`, `empty` oder `credit`. Um alle diese Zustände zu berücksichtigen, wird ein neuer Zustand als statisches Teamattribut deklariert, das Attribut `$UNDEF`, das den undefinierten Zustand repräsentiert. Zudem wird ein weiteres Rollenattribut eingeführt, das Attribut `$pos_states`, das in einer Liste alle potentiellen Nachzustände speichert (Codezeile 9).

In der Rollenmethode `$withdraw`, die zum Aufruf von `withdraw` korrespondiert, werden alle potentiellen Nachzustände in `$pos_states` eingefügt, wenn der aktuelle Zustand `credit` ist (Codezeilen 39-45). Um zu signalisieren, daß mehrere Nachzustände möglich sind, wird das Rollenattribut `$state` auf den Zustand `$UNDEF` gesetzt (Codezeile 41).

Der anschließende Aufruf von `$checkState` ermittelt, welchen der Zustände in `$pos_states` das korrespondierende zu testende Objekt angenommen hat, wenn der Zustand undefiniert ist (Codezeilen 12-28). Entspricht keiner der Zustände in `$pos_states` dem aktuellen Zustand des korrespondierenden zu testenden Objekts, so wird eine Fehlermeldung ausgegeben (Codezeilen 22-23). Ist mindestens eine der Zustandsinvarianten der Zustände in `$pos_states` für das korrespondierende zu testende Objekt wahr, so wird der Zustand auf den ersten gefundenen dieser Zustände gesetzt.

Sind die Zustandsinvarianten der potentiellen Nachfolgezustände nicht disjunkt, so ist u.U. nicht entscheidbar, welche der Transitionen geschaltet hat. Da in der Implementierung die Zustände in `$pos_states` der Reihenfolge nach abgearbeitet werden, ist sowohl ein falsch-positives als auch ein falsch-negatives Testergebnis möglich. So wird zunächst der Zustand `empty` überprüft und das ausführbare Zustandsdiagramm nimmt diesen Zustand an, wenn der Kontostand des korrespondierenden Objekts vom Typ `Account` gleich Null ist.

Sei die Zustandsinvariante des Zustands `credit` $self.balance >= 0$ und damit nicht disjunkt zu der Zustandsinvarianten des Zustands `debit` $self.balance == 0$. Dann ist der zuerst überprüfte Zustand der Zustand `empty`, aber tatsächlich befindet sich das korrespondierende Objekt im Zustand `credit`. Die Sequenz der nächsten Aufrufe führt unter Umständen in einen Zustand, der von `credit` aus korrekt wäre, von `empty` jedoch nicht, z.B. wenn der Aufruf von `close` nur im Zustand `credit` erlaubt ist, nicht aber im Zustand `empty` (falsch-positives Testergebnis). Der umgekehrte Fall ist ebenfalls möglich (falsch-negatives Testergebnis).

Natürlich ist es möglich, sich alle potentiellen Zustände solange zu merken, bis der reale Zustand des zu testenden Objekts wieder exakt ermittelbar ist, wie es z.B. die Technik in [98] umsetzt. Davon wurde jedoch in der vorliegenden Arbeit abgesehen, da im schlimmsten Fall alle Zustände mit vielen aktiven Unterzuständen und damit vielen aktiven Teams gespeichert werden müssen. Die hier vorgestellte Implementierung ist ein Kompromiß zwischen dem Ausschluß von Nichtdeterminismen und einer Implementierung, die viele Teams gleichzeitig verwalten muß.

14.2 Transformation der Zustandsdiagramme

Für alle Teile eines Teams wird im Folgenden beschrieben, aus welchen Teilen eines Zustands s sie generiert sind. Der betrachtete Zustand $s \mid \neg s \in \Sigma_{simple}$, für den das Team generiert wird, ist ein Zustand des Zustandsdiagramms sc, das der Klasse c zugeordnet ist. Zustände s', für die gilt

$s' \in \Sigma_{simple}$, werden nur als Attribut in einer Rolle verwaltet und erhalten kein eigenes Team.

Als Grundlage der Realisierung dient die in Kapitel 12 definierte ObjectTeams-Schablone. Dabei werden die einzelnen Teile eines Teams nacheinander betrachtet.

Variable Teile sind in kursiver Schrift geschrieben und zusätzlich unterstrichen. Für Teile, die in allen Teams zur Implementierung der Zustandsdiagramme gleich sind, wird eine nichtproportionale Schrift verwendet.

Es wird in der Regel nur eine Rolle erzeugt, nur bei Zuständen mit parallelen Unterzuständen wird pro Unterzustand eine weitere Rolle erzeugt.

Teamkopf. `@TEAMNAME` ist $\underline{name(s)}$`$SC`. Das Team ist öffentlich, da es vom Testtreiber angesprochen wird, der in einem anderen Java-Paket liegt.

Teamrumpf. Es wird ein Konstruktor erzeugt:

```
public name(s)$SC() {
      this.activate();
}
```

Zudem werden einige Teamattribute und -methoden generiert, abhängig von Typ des betrachteten Zustands.

In jedem Fall wird ein Teamattribut `$Statechart` erzeugt:

```
private name(c)$Statechart $myname(c);
```

Wenn $s \in \Sigma_{and}$, werden für alle Unterzustände $s_x \mid s\chi s_x$ weitere Teamattribute generiert:

```
private name(s_x)$Statechart $myname(s_x)$name(c);
```

Für jeden Zustand $s_x \mid s\chi s_x$ wird zudem ein statisches Attribut erzeugt, wobei *TO_UPPER_CASE* eine Zeichenfolge in Großbuchstaben umwandelt und *NUM* eine fortlaufende Zahl größer oder gleich Null vom Typ `int` liefert:

```
private static int TO_UPPER_CASE(name(s_x)) = NUM;
```

Bei nichtdeterministischen Zuständen wird ein weiteres statisches Attribut generiert:

```
private static int $UNDEF = -1;
```

Es wird die Teammethode `$setRole` erzeugt, jedoch in unterschiedlicher Weise für Zustände $s \in \Sigma_{xor}$ und $s \in \Sigma_{and}$.

Für $s \in \Sigma_{xor}$ wird nur die Methode `$setRole` erzeugt. Anmerkung: Der Aufruf

`$myname(c).$init();`

in der Methode `$setRole` wird nur erzeugt, wenn $\exists s_x \in \Sigma \mid s\chi s_x \wedge \neg s_x \in \Sigma_{simple}$ oder wenn s ohne Historie definiert ist.

```
public void $setRole( name(c) as $Statechart asc ) {
      $myname(c) = asc ;
      $myname(c).$init() ;
}
```

Wenn $s \in \Sigma_{and}$ gilt, dann werden neben `$setRole` weitere Methoden für jeden Zustand $s_x \mid s\chi s_x$ erzeugt, die analog zu `$setRole` der Zustände $s \in \Sigma_{xor}$ implementiert und aus `$setRole` aufgerufen werden:

```
public void $setRole( name(c) as $Statechart asc ) {
      $setToplevelRole( name(c) as $Statechart asc );
      $setname(s_x)Role( name(c) as name(s_x)$Statechart asc );
}
```

Rollenkopf. Es wird eine Rolle mit dem Namen `$Statechart` erzeugt:

`public class $Statechart playedBy` $name(c)$

Wenn $s \in \Sigma_{and}$, werden zusätzlich weitere Rollen für jeden Zustand $s_x \mid s\chi s_x$ erzeugt:

`public class` $name(s_x)$`$Statechart playedBy` $name(c)$

Rollenattribute. Die wichtigsten Rollenattribute sind:

`private int $state =` $name(s_x)$`;`	wobei $s_x \in \Sigma_{is} \mid s\chi s_x$
`private` $name(s_x)$`SC` $name(s_x)$`State;`	wobei $s_x \in \Sigma \mid s\chi s_x \wedge s \in \Sigma_{xor} \wedge \neg s_x \in \Sigma_{simple}$
`private Vector $pos_states = new Vector();`	bei Nichtdeterminismen

Rollenmethoden. Die wichtigste Rollenmethode ist `$checkState`. Es gibt drei Varianten, eine für einfache geschachtelte Zustände, eine für parallele Unterzustände und eine für Nichtdeterminismen. Der Übersichtlichkeit halber wird hier nur die erste Variante vorgestellt, die anderen Varianten werden analog erzeugt (siehe auch die entsprechenden Codefragmente in den Abbildungen 14.14 und 14.16).

Die Variante erzeugt aus den Unterzuständen des betrachteten Zustands s eine Kaskade von `if`-Anweisungen. Für den ersten Unterzustand wird eine `if`-Anweisung generiert, für alle folgenden eine Anweisung der Form `else if`. Die Abfrage `if ( this == my$`$name(c)$` )` ist notwendig aufgrund der Beschränkung auf eine Rolle in einem Team. Hier vorgestellt ist ein Zustand mit zwei Unterzuständen s_1 und s_2. Beliebig viele weitere Zustände werden analog zu s_2 bearbeitet.

```
void $checkState() {
      if (this == my$name(c) ) {
            if ( $state == TO_UPPER_CASE(name(s1)) ) {
                  if ( OCL2Java(s_inv(s1)) ) {
                        $logState();
                  } else {
                        $logStateError();
                  }
            } else if ( $state == TO_UPPER_CASE(name(s2)) ) {
                  if ( OCL2Java(s_inv(s2)) ) {
                        $logState();
                  } else {
                        $logStateError();
                  }
            } else {
                  $logStateError();
            }
      }
}
```

Weitere erzeugte Methoden sind die Logging-Methoden `$logMethodCall`, `$logState`, `$logStateError`, `$logMethodCall` und `$logMethodCallError` sowie in einigen Fällen die Methoden `$init` (siehe oben) und `$deactivate` (wenn $\exists s_x \mid s\chi s_x \wedge \neg s_x \in \Sigma_{simple}$).

Call-Out-Bindung. Es werden Call-Out-Bindungen zu allen Methoden $m \in M \mid isQuery(m)$ und zu der von Object geerbten Methode `hashCode` hergestellt:

`abstract` $type(m)$ `$`$name(m)$ `(` $para(m)$ `);`
`$`$name(m)$ `->` $name(m)$ `;`

Call-In-Bindung. Es werden Call-In-Bindungen zu allen Methoden $m \in M \mid isUpdate(m)$ hergestellt. Dabei werden zwei Typen von Methoden unterschieden, Methoden, die in einem der direkten Unterzustände von s eine Transition auslösen und solche, die es nicht tun.

Methoden $m \in M \; \exists s_x \in \Sigma \mid s \chi s_x \wedge s_x \in calc\_prestates(m)$ werden wie folgt gebunden:

`$`$\underline{name(m)}$ `<- after` $\underline{name(m)}$ `;`

Alle anderen Update-Methoden werden nur an `$checkState` gebunden.

`$checkState <- after` $\underline{name(m)}$ `;`

Alle Update-Methoden werden zusätzlich an die Methode `$logMethodCall` gebunden:

`$logMethodCall <- before` $\underline{name(m)}$ `with {id <-` $\underline{NUM}$ `};`

Kapitel 15

Integration von Zusicherungen

Neben der Integration der angereicherten Zustandsdiagramme ist auch die Integration der erweiterten OCL-Constraints aus Kapitel 10.2 in das zu testende System nötig. Auch diese Integration wird in der vorliegenden Arbeit mit Hilfe des ObjectTeams-Programmiermodells vorgenommen.

Um OCL-Constraints zur Laufzeit auswerten zu können, sind in der vorliegenden Arbeit zwei Schritte nötig, um zu einer ausführbaren Spezifikation zu kommen. Im ersten Schritt werden OCL-Constraints durch einen Parser eingelesen und in Java-Code transformiert. Im zweiten Schritt wird aus den eingelesenen OCL-Constraints das Team generiert. Je ein Team wird an eine zu testende Klasse gebunden. Im Team wird für jedes zu testende Objekt eine Rolleninstanz erzeugt.

Beide Schritte werden in den beiden folgenden Abschnitten beschrieben.

15.1 Transformation von OCL-Constraints

Die OCL-Constraints werden zunächst mit der in Anhang B gegebenen Grammatik eingelesen und anschließend in Java-Code transformiert.

Die Transformation in Java-Code erfordert einige Betrachtungen zur Auswertungsreihenfolge in Java und OCL. Betrachtet man als Beispiel den Term:

$b > 0 \; and \; a/b > 1$,

so spielt in OCL die Reihenfolge der Auswertung der beiden Teile der Konjunktion keine Rolle. Java dagegen wertet die meisten Konstrukte bei gleicher Priorität von links nach rechts aus. Das bedeutet, daß in Java die Reihenfolge der beiden Teile der Konjunktion eine Rolle spielt. Im Fall, daß die Variable b zur Laufzeit den Wert 0 enthält, wird der oben gegebene Term korrekt ausgewertet, aber eine umkehrte Reihenfolge der Teilterme:

$a/b > 1 \; and \; b > 0$,

führt in Java zu einem Laufzeitfehler. Aus diesem Grund muß der Entwickler des Systems bei der Spezifikation der OCL-Constraints darauf achten, daß die Reihenfolge der Teilterme der OCL-Constraints der Reihenfolge der Teilterme in Java entspricht, da das Testsystem UT^3 einen solchen Konflikt nicht löst[1].

Eine Ausnahme bildet bei der Auswertungsreihenfolge der Zuweisungsoperator, der von rechts nach links ausgewertet wird. Da in OCL jedoch keine Zuweisungen formuliert werden, sondern aussschließlich Prädikate, ist der Zuweisungsoperator bei einer Transformation von OCL nach Java nicht relevant[2].

Des Weiteren müssen Vorrangregeln beachtet werden. Wie jedoch die Tabelle in Abbildung 15.1 zeigt, entsprechen sich die Vorrangregeln der beiden Sprachen in weiten Teilen. In der Tabelle sind jeweils die zu einander korrespondierenden Operatoren in OCL und Java mit ihrer Priorität einander gegenübergestellt. Wenn es keine Entsprechung in der jeweils anderen Sprache gibt, ist

[1] Zur Lösung solcher Konflikte muß ein Werkzeug wie bspw. ein Constraint-Solver an das Testsystem angebunden werden.

[2] Relevant ist der Zuweisungsoperator nur bei einer Transformation von Java nach OCL.

Priorität / Sprache	OCL	Java
Hoch	@pre	
	. ->	*Postfixoperatoren:* [] . ++ - *Methodenaufruf:* ()
	unäre Operatoren: not -	*Präfixoperatoren:* ++ – + - *unäre Operatoren*: ! ~
		new *Typcast:* ()
	binäre Operatoren: * /	*binnäre Operatoren*: * / %
	binäre Operatoren: + -	*binäre Operatoren*: + -
	if-then-else-Konstrukt	
	< > <= >=	< > <= >= instanceof
	= <>	== !=
	and or nor	&&
	implies	||
Niedrig		=

Abbildung 15.1: Vorrangregeln in OCL und Java

das Feld in der anderen Sprache leer gelassen. Wie der Tabelle zu entnehmen ist, gibt es eine Reihe von Konstrukten in Java, die keine Entsprechung in OCL besitzen, jedoch nur einige wenige in OCL, die keine Entsprechung in Java besitzen.

Aus dieser Kenntnis wurde ein einfacher OCL-Java-Transformator entwickelt. Dabei werden Operatoren auf Basistypen 1:1 von einer Sprache in die andere übersetzt.
Beispiele sind:
`"and"` in OCL wird zu `"&&"` in Java
`"not"` in OCL wird zu `"!"` in Java
`"<"` in OCL bleibt `"<"` in Java

Andere Operatoren oder das Schlüsselwort `@pre` werden gesondert behandelt.

Der Operator `implies` wird wie folgt ausgewertet:
$OCltoJava(a$ `"implies"` $b) =$ `"!"` $OCLtoJava(a)$ `"||"` $OCLtoJava(b)$.

Das Schlüsselwort `@pre` wird je nach Kontext unterschiedlich behandelt:
Bei Parametern:
$OCltoJava(parameter$`"@pre"` $) = parameter$`"$ATPRE"`.

Bei Attributen und Methoden des aktuellen Objekts:
$OCltoJava(self.field$`"@pre"` $) =$`"obj$ATPRE."`$field$.

Eingelesen werden sowohl Deklarationen mit einem Kontext, hier Invarianten, Vor- und Nachbedingungen als auch Zustandsinvarianten als Teil einer Zustandsdiagrammspezifikation.
Es gibt jedoch auch eine Reihe von Konstrukten, die nicht unterstützt werden:

- Konstrukte wie *let* und $if - then - else$,

- Der Operator ->,

- Operationen über Kollektionen.

```
context Account::withdraw (amount:int)
   pre : amount > 0
   post: self.balance = self.balance@pre - amount
```

Abbildung 15.2: Vor- und Nachbedingung für `withdraw` der Klasse `Account`

Im Rahmen der vorliegenden Arbeit wurde nur ein einfacher OCL-Java-Transformator entwickelt. Die Anbindung eines effizienteren OCL-Java-Transformators wie DresdenOCL [25] oder des in [13] vorgestellten Transformators an das Testsystem UT^3 ist jedoch geplant, um einen größeren Teil des OCL-Sprachumfangs zu unterstützen.

15.2 Generierung der ObjectTeams

Aus den eingelesenen OCL-Constraints in Form von Invarianten, Vor- und Nachbedingungen wird anschließend ObjectTeams/Java-Code generiert, der zur Überprüfung dieser Constraints eingesetzt wird.

Ein Beispiel soll dies verdeutlichen. Die Methode `withdraw` besitzt die in Abbildung 15.2 gegebene Vor- und Nachbedingung.

Das zugehörige Team zeigt Abbildung 15.3. Die Überprüfung der OCL-Constraints findet in der Rolle `$OCLCheck` im Team `Account$OCL` statt. Obwohl es möglich ist, alle Rollen für alle zu testenden Klassen in einem Team zu implementieren, wird für jede zu testende Klasse ein eigenes Team mit dem Namen der Klasse, ergänzt um den Zusatz `$OCL` erzeugt. Die Erzeugung eines Teams für jede zu testende Klasse ist einfacher als die Integration mehrerer Rollen für verschiedene zu testende Klassen in ein Team, besonders in Hinblick auf die Unabhängigkeit der Aktivierung des Tests für einzelnen Klassen, die bisher nur auf Teamebene möglich ist.

Die relevanten Methoden, hier die Methode `withdraw`, werden per Call-In gebunden. Dabei wird für jede Methode eine Überprüfung der Vorbedingung durch eine Call-In-Bindung vom Typ `before` (im Beispiel `$pre_withdraw`, Codezeile 29) und eine Überprüfung der Nachbedingung durch eine Call-In-Bindung vom Typ `after` (im Beispiel `$post_withdraw`, Codezeile 30) vorgenommen.

Die Implementierung der Methode `$pre_withdraw` (Codezeilen 32-39) überprüft zunächst die Vorbedingung und speichert anschließend sowohl das aktuelle Objekt im Rollenattribut `obj$ATPRE` als auch alle Parameter der Methode (im Beispiel im Rollenattribut `amount$ATPRE`). Dabei spielt es keine Rolle, ob die Werte in der Nachbedingung gebraucht werden. Dieser Weg der Implementierung wurde gewählt, um eine Analyse der Nachbedingung auf benötigte Objekte und Werte im Vorzustand zu vermeiden.

Zur Speicherung von Objekten wird die in der Oberklasse aller Klassen in Java, der Klasse `Object`, definierte Methode `clone` verwendet, die per Call-Out-Bindung durch die Rollenmethoden `$clone` in der Rolle verfügbar ist. Der korrekte Vergleich zwischen dem alten Zustand und dem neuen Zustand von Objekten nach der Ausführung einer Methode erfordert das Überschreiben der von `Object` geerbten Methode `clone`. Dies ist zur Zeit noch Aufgabe des Entwicklers des zu testenden Systems, wird aber im weiterentwickelten Testsystem UT^3 direkt im Aspekt, also in der Rolle, implementiert.

Anschließend folgt ein Aufruf der Methode `$checkInvariant`, um die Klasseninvariante zu überprüfen. Die Implementierung der Methode `$checkInvariant` ist im Beispiel nicht angegeben, da die Klasseninvariante im aktuellen Fall `true` ist. Diese erfolgt aber analog zur Implementierung von Vor- und Nachbedingungen.

Die Implementierung der Methode `$post_withdraw` (Codezeilen 41-46) überprüft die Nachbedingung und ebenfalls die Invariante. Dabei wird auf alte Werte mit Hilfe der Rollenattribute, die

```
 1 public team class Account$OCL {
 2
 3    Account$OCL() {
 4       this.activate();
 5    }
 6
 7
 8    /* Role class */
 9    class $OCLCheck playedBy Account {
10
11        Account obj$ATPRE;
12        int amount$ATPRE;
13
14        /* CallOutBinding */
15       abstract boolean $isActive();
16       $isActive -> isActive;
17       abstract boolean $isBlocked();
18       $isBlocked -> isBlocked;
19       abstract boolean $isClosed();
20       $isClosed -> isClosed;
21       abstract int $getBalance();
22       $getBalance -> getBalance;
23       abstract Object $clone();
24       $clone -> clone;
25       abstract int $getHashCode();
26       $getHashCode -> hashCode;25
27
28       /* CallInBinding */
29       $pre_withdraw <- before withdraw;
30       $post_withdraw <- after withdraw;
31
32       void $pre_withdraw(int amount) {
33           if (! (amount > 0)) {
34               $logPreError();
35           }
36           obj$ATPRE = (Account) $clone();
37           amount$ATPRE = amount;
38           $checkInvariant();
39       }
40
41       void $post_withdraw(int amount) {
42           if (! ($getBalance() == obj$ATPRE.getBalance() - amount)) {
44               $logPostError();
44           }
45           $checkInvariant();
46       }
47
48       void $checkInvariant() {
49           [...]
50       }
51
52       /* Log Methods */
53       [...]
54
55    }
56 }
```

Abbildung 15.3: Implementierung von Vor- und Nachbedingungen

den alten Zustand speichern, zugegriffen.

Analog zur Implementierung der Zustandsdiagramme ist der Test auf Update-Methoden beschränkt, da die Query-Methoden zur Überprüfung des Zustands des korrespondierenden Objekts benötigt werden. Abhilfe für dieses Problem wird dadurch geschaffen, daß das Testsystem sich bei der Überprüfung des Zustands direkt der Instanzvariablen bedient, da in der aktuellen Version der ObjectTeams/Java-Implementierung der Zugriff auf Attribute erlaubt ist.

Die Abbildung zeigt die Variante, die bis vor kurzem noch nötig war und in dieser Weise in UT^3 implementiert ist. Dabei müssen alle Attribute der zu testenden Klasse einer entsprechenden Query-Methode zugeordnet werden, um eine Abfrage der Werte zu ermöglichen. Eine Anpassung der Implementierung an die neue ObjectTeams/Java-Version wird gerade vorgenommen.

Auf Basis der in Kapitel 12 definierten Schablone für ObjectTeams/Java soll nun schematisch die Generierung der Teams für eine Klasse $c = (M, A)$ beschrieben werden. Dabei werden wie in Kapitel 14 die einzelnen Teile eines Teams nacheinander betrachtet und für jeden Teil erläutert, aus welchen Informationen er erzeugt wird. Variable Teile sind wie zuvor kursiv geschrieben und unterstrichen. Für Teile, die in allen Teams zur Überprüfung von OCL-Constraints gleich sind, wird eine nichtproportionale Schrift verwendet.

Teamkopf. `@TEAMNAME` ist $name(c)$`$OCL`. Das Team ist öffentlich, da es vom Testtreiber angesprochen wird, der in einem anderen Java-Paket liegt.

Teamrumpf. Es wird nur ein Konstruktor erzeugt, der das Team aktiviert:

```
public name(c)$OCL() {
      this.activate();
}
```

Es gibt keine weiteren Teamattribute oder -methoden.

Rollenkopf. Es wird pro Team nur eine Rolle mit dem Namen `$OCLCheck` erzeugt:

```
public class $OCLCheck playedBy name(c)
```

Rollenattribute. Es wird ein Rollenattribut mit dem Namen `obj$ATPRE` vom Typ $name(c)$ angelegt. Für alle Methodenparameter $p \in para(m)$ aller Methoden $m \in M$ werden Rollenattribute mit dem Namen $name(p)$`$ATPRE` und dem Typ $type(p)$ angelegt.

Rollenmethoden. Es wird die Rollenmethode `$checkInvariant` aus der Invariante $INV(c)$ generiert:

```
void $checkInvariant() {
      if ( ! OCL2Java(INV(c)) ) {
            $logInvError();
      }
}
```

Außerdem gibt es eine Reihe von Logging-Methoden, die für alle Teams gleich sind. Es handelt sich um die Methoden `$logPreError`, `$logPostError` und `$logInvError`.

Call-Out-Bindung. Es werden Call-Out-Bindungen zu allen Methoden $m \in M \mid isQuery(m)$ und zu den von Object geerbten Methoden `clone` und `hashCode` hergestellt:

```
abstract type(m) $name(m) ( para(m) );
$name(m) -> name(m) ;
```

Call-In-Bindung. Es werden Call-In-Bindungen zu allen Methoden $m \in M \mid isUpdate(m)$ hergestellt. Für jede dieser Methoden m wird eine Call-In-Bindung vom Typ `before` und eine Call-In-Bindung vom Typ `after` generiert:

```
$pre_name(m) <- before name(m) ;
$post_name(m) <- after name(m) ;
```

Die Implementierung der beiden Methoden prüft die Vor- bzw die Nachbedingung. Beide Methoden rufen `$checkInvariant` auf, um die Invariante zu überprüfen.

Die Methode zur Überprüfung der Vorbedingung speichert darüber hinaus die Parameter und das aktuelle Objekt. Es wird unterschieden zwischen Objekten, die geklont werden, und Basistypen, die zugewiesen werden. Exemplarisch ist jeweils ein solcher Parameter angegeben, wobei der Parameter vom Basistyp mit *bp* benannt ist, der Parameter vom Klassentyp mit *kp*. Bei der Generierung wird dies für alle Parameter wiederholt.

```
void $pre_name(m) ( para(m) ) {
      if ( ! OCL2Java(PRE(c)) ) {
            $logPreError();
      }
      obj$ATPRE = ( name(c) ) $clone();
      name(bp)$ATPRE = name(bp);
      name(kp)$ATPRE =( type(kp) ) name(kp).$clone();
      $checkInvariant();
}


void $post_name(m) ( para(m) ) {
      if ( ! OCL2Java(POST(c)) ) {
            $logPostError();
      }
      $checkInvariant();
}
```

Kapitel 16

Zusammenfassung

Wie auch andere Arbeiten, z.B. [113], gezeigt haben, sind Aspekte sehr gut statt klassischer Instrumentierungstechniken einsetzbar.

Ein großer Vorteil ist die nicht-invasive Integration des zusätzlichen Testcodes. Da ObjectTeams/Java auf dem Quell- oder Bytecode des zu instrumentierenden Systems arbeitet und den zusätzlichen Code zur Ladezeit einwebt, ergibt sich ein weiterer Vorteil auch gegenüber anderen apsektorientierten Sprachen wie AspectJ: Die zu testende Implementierung muß nicht noch einmal neu kompiliert werden.

Das reduziert den Testaufwand im Gegensatz zu einer Implementierung in AspectJ enorm. Auch klassischen Instrumentierungstechniken, die meist eine Neukompilierung des zu testenden Systems erfordern, ist der in der vorliegenden Arbeit verfolgte Ansatz überlegen. Mit ObjectTeams/Java ist die Pflege nur einer Version der zu testenden Implementierung nötig. Es muß keine Unterscheidung getroffen werden zwischen dem System mit Testcode und dem System ohne Testcode, das ausgeliefert wird.

Zudem ist die aspektorientierte Instrumentierung flexibler als die klassische, da eine Erweiterung um neue Funktionalität leicht möglich ist.

Einer der größten Nachteile der aspektorientierten Instrumentierung ist die Beschränkung auf methodenweise Instrumentierung. In ObjectTeams/Java ist eine Aufhebung dieser Beschränkung vorerst nicht zu erwarten, es gibt aber im AspectJ-Umfeld Bestrebungen, flexiblere Joinpoint-Definitionen zuzulassen. Der AspectJ-Compiler *abc* [1] bietet die Definition eigener Joinpoints an, die auch anweisungsweise Instrumentierung zulassen. Mit Hilfe dieser neuen Technik ist einer der größten Nachteile der aspektorientierten Instrumentierung aufgehoben.

Die in der vorliegenden Arbeit vorgestellte Technik bindet die in Kapitel 10 erzeugten Testorakel mit Hilfe von daraus generierten ObjectTeams in das zu testende System ein. Die Auswahl von ObjectTeams/Java als aspektorientierte Programmiersprache hat einige Vorteile gegenüber der populären Programmiersprache AspectJ, wie z.B. das Einweben des zusätzlichen Codes zur Ladezeit, jedoch auch einige Nachteile.

So bietet ObjectTeams/Java bisher in Ermangelung einer Joinpoint-Sprache nur bedingt Quantifizierungsmöglichkeiten. Methoden, die gebunden werden, müssen deshalb explizit aufgezählt werden. Dies ist durch die Verwendung eines Generators jedoch kein wirkliches Problem.

Eine andere Schwäche von ObjectTeams ist die fehlende Abhängigkeit der Ausführung des eingefügten Codes vom Kontrollfluß, wie es in AspectJ mit dem Schlüsselwörtern `cflow` und `cflowbelow` möglich ist. Dies ist z.B. nötig, um interne von externen Aufrufen von Methoden zu unterscheiden, da nur bei externen Aufrufen die Einhaltung von Invarianten, Vor- und Nachbedingungen gefordert ist. Die Aktivierung und Deaktivierung von Teams ist hierfür kein Ersatz. Hier ist zu untersuchen, welche Vorteile das Guard-Konzept und die geplante Joinpoint-Sprache in ObjectTeams/Java in dieser Hinsicht bieten.

Als Fazit der Untersuchung in diesem Teil läßt sich unter Einbeziehung der Aussagen in Kapitel 10 und der Zusammenfassung in Kapitel 11 die Entscheidung für eines der beiden Testorakel erleichtern. Da Zustandsdiagramme nur lösbaren Nichtdeterminismus enthalten dürfen, um adäquat in einem Team implementiert zu werden, bietet sich bei nichtdeterministischen Zustandsdiagrammen mit nichtdisjunkten Zustandsinvarianten der Einsatz der erweiterten OCL-Constraints an.

Teil IV

Zusammenfassung und Ausblick

Kapitel 17

Fazit

Im Rahmen der vorliegenden Arbeit wurde ein Ansatz zum UML-basierten Test für objektorientierte Systeme vorgestellt. Dabei wurde insbesondere darauf geachtet, eine durchgängige Technik zu entwickeln, die nicht nur die Testaktivitäten Testfallgenerierung und Testorakelerzeugung unterstützt, sondern zusätzlich eine Lösung für das Instrumentierungsproblem anbietet.

Es wurden alle UML-Diagrammtypen auf ihre Testbarkeit untersucht. Bei der Untersuchung wurden zunächst die Informationen analysiert, die die einzelnen UML-Diagrammtypen über das modellierte System zur Verfügung stellen. Anschließend wurden alle Diagrammtypen im Hinblick auf ihre Nutzung für den Test bewertet.

Eine Reihe von Arbeiten zum UML-basierten Test und dem Test auf der Basis verwandter Modelle wurden untersucht und dem in der vorliegenden Arbeit entwickelten Ansatz gegenübergestellt. Auch Arbeiten aus anderen Kontexten wurden auf ihre Übertragbarkeit auf UML-Diagrammtypen hin betrachtet.

Aus dieser Analyse wurden schließlich die für den Klassen- und Integrationstest am geeignesten scheinenden Diagrammtypen ausgewählt, gleichzeitig aber auf die Erweiterbarkeit um andere Diagrammtypen geachtet.

Im Hauptteil der Arbeit wurden drei Themengebiete bearbeitet:

1. Testfallgenerierung auf der Basis von Sequenzdiagrammen und Zustandsdiagrammen,

2. Erzeugung des Testorakels aus Zustandsdiagrammen und Invarianten, Vor- und Nachbedingungen in OCL,

3. Integration von Testcode, speziell des Testorakels, in die zu testende Implementierung unter Verwendung aspektorientierter Programmiertechniken.

Damit stellt die vorliegende Arbeit Lösungen für alle drei identifizierten Problembereiche beim Test objektorientierter Programme zur Verfügung. Alle Techniken wurden prototypisch im Testsystem UT^3 implementiert.

Testfallgenerierung Zur Testfallgenerierung wurde das Sequenzdiagramm herangezogen, da es Informationen über Nachrichten liefert, die zwischen Objekten ausgetauscht werden. Definiert man Testfälle als Nachrichtenfolgen, so sind diese direkt aus dem Sequenzdiagramm abzuleiten.

Da Testsequenzen jedoch initialisiert werden müssen und darüber in der Regel keine Information im Sequenzdiagramm enthalten ist, wird diese Information aus Zustandsdiagrammen gewonnen, die Objektlebenszyklen der an einer Sequenz beteiligten Objekte beschreiben.

Die Kombination von Sequenzdiagramm und Zustandsdiagramm löst einige der Probleme, die ein Test auf der Basis nur eines dieser Diagrammtypen besitzt.

Die Sollwertgewinnung aus Sequenzdiagrammen bringt meist große Einschränkungen mit sich, wie die Annahme, *alle* nicht modellierten Sequenzen sind als fehlgeschlagener Test zu werten. Durch die Kombination mit einem von den Testfällen unabhängigen Testorakel stellt sich die Frage nach einer Gewinnung von Sollwerten aus dem Sequenzdiagramm nicht.

Sequenzen im Sequenzdiagrammen spiegeln das Verhalten wider, das dem Entwickler des Systems besonders wichtig war. Testsequenzen, die aus dem Sequenzdiagramm gewonnen werden, testen demnach Verhalten, das als besonders wichtig angesehen wird. Doch erst die Kombination mit der Information aus dem Zustandsdiagramm ermöglicht einen effektiven Test. Die korrekte Initialisierung der Testsequenzen muß nicht wie in anderen Ansätzen von außen erfolgen.

Bezogen auf das in der vorliegenden Arbeit definierte Testmodell für objektorientierte Systeme, bestimmen die erzeugten Testfälle sowohl die Eingaben an die zu testende Implementierung als auch den Zustand der zu testenden Implementierung, da für jeden Testfall die beteiligten zu testenden Objekte instanziiert werden. Somit ist die Eignung der erzeugten Testfälle für objektorientierte Programme gezeigt.

Testorakelerzeugung Für das Testorakel wurden Zustandsdiagramme und OCL-Spezifikationen in Form von Invarianten, Vor- und Nachbedingungen in OCL herangezogen. Da Information oft verteilt über mehrere Modelle und Diagrammtypen vorliegt, ist die Kombination für einen adäquaten Test notwendig.

Die Arbeit beschränkt sich auf die Behandlung von OCL-Spezifikationen in Form von Invarianten, Vor- und Nachbedingungen. Eine Erweiterung des Ansatzes für andere OCL-Spezifikationen ist prinzipiell möglich. Trotz der vorgenommenen Einschränkung ist der Gewinn für den Test, der aus der Kombination von unterschiedlichen Modellen und Diagrammtypen gezogen werden kann, deutlich geworden.

So schränken die in OCL spezifizierten Vor- und Nachbedingungen die Verarbeitung von Transitionen im Zustandsdiagramm ein und die Transitionen im Zustandsdiagramm definieren zusätzliche Vor- und Nachbedingungen von Methoden. Wird nur eine der beiden Sichten berücksichtigt, geht wichtige Information beim Test verloren.

Gleichzeitig ist die Wahl der UML-Bestandteile für das Testorakel in der vorliegenden Arbeit auch unter einem anderen Gesichtspunkt sinnvoll. Sowohl OCL-Constraints als auch Zustandsdiagramme definieren Vor- und Nachbedingungen für Methoden als auch den Zustandsraum der zu testenden Objekte. Eine Kombination beider Sichten bietet sich geradezu an und ist in Bezug auf die zu testende Klasse als vollständig anzusehen.

Bei der Kombination von OCL-Constraints wurden zwei Techniken entwickelt, die Integration der OCL-Constraints in das Zustandsdiagramm, genannt angereicherte Zustandsdiagramme, und die Extraktion von OCL-Constraints aus dem Zustandsdiagramm und Kombination mit den OCL-Constraints, genannt erweiterte OCL-Constraints. Beide Techniken der Kombination sind erfolgsversprechend. Welche Art der Kombination in welcher Situation sinnvoll ist, hängt von der zu testenden Implementierung und vom Modell ab und wird in Zukunft im Rahmen einer größeren Fallstudie evaluiert.

Betrachtet man das im Rahmen der vorliegenden Arbeit entwickelte Testmodell, so stellt man fest, daß die bisher entwickelten Testorakel weitgehend dem Testorakel des Testmodells entsprechen. Es werden Zustände und Ausgaben der zu testenden Implementierung überprüft. Einzig die Nachrichten, die zwischen Objekten gesendet werden, sind bisher nicht berücksichtigt. Dies läßt sich aber realisieren durch die zusätzliche Integration von Sequenzdiagrammen in das Testorakel.

Instrumentierung Im Rahmen der Arbeit wurde untersucht, in wieweit sich aspektorientierte Techniken zur Instrumentierung eignen. Dies wurde realisiert am Beispiel des Testorakels. Dabei

wurden beide Arten des Testorakels, die erweiterten OCL-Constraints sowie die angereicherten Zustandsdiagramme, in der aspektorientierten Sprache ObjectTeams/Java implementiert.

Es läßt sich feststellen, daß Testen ein Cross-Cutting-Concern bezüglich des zu testenden Systems ist und der Einsatz aspektorientierter Programmiertechniken deshalb Vorteile beim Testen bringt.

Die Instrumentierung unter Verwendung aspektorientierter Programmiertechniken ist flexibler und leicht zu implementieren. Änderungen an der Instrumentierungstechnik können vom Tester selbst vorgenommen werden.

Werden aspektorientierte Programmiertechniken eingesetzt, stellt sich das Problem der Kapselung in objektorientierten Programmen nicht, da Aspekte privilegierten Zugriff auf die adaptierten Implementierung besitzen.

Ein weiterer Vorteil, der sich aus der Verwendung von ObjectTeams/Java als Programmiersprache ergibt, ist der reduzierte Testaufwand. ObjectTeams/Java webt zusätzlichen Code zur Ladezeit in die adaptierte Implementierung. Dadurch ist das Konfigurationsmanagment vereinfacht, weil nur eine Version der zu testenden Implementierung zu pflegen ist.

Das Testsystem UT^3 Um die entwickelten Techniken zu realisieren, wurde das Testsystem UT^3 entworfen. Es stellt prototypisch alle in der vorliegenden Arbeit entwickelten Techniken bereit und unterstützt die in Kapitel 8 vorgestellten Basisformalismen.

Das Testsystem besitzt eine umfangreiche Codebasis. Die modulare Struktur erlaubt die einfache Erweiterung um andere UML-Diagrammtypen und die Integration weiterer Testtechniken.

Das Testsystem wurde genutzt, um die vorgestellten Techniken auf das Fallbeispiel *Banksystem* anzuwenden und so ihre grundsätzliche Eignung für den objektorientierten Test zu zeigen. Es wird künftig im Rahmen einer größeren Fallstudie eingesetzt werden, um die Techniken abschließend zu evaluieren und zu bewerten.

Zusammenfassung Die vorliegende Arbeit stellt eine umfassende Technik zum UML-basierten Test vor, von der Ableitung von Testfällen und Testorakeln aus UML-Modellen bis hin zur Instrumentierung des zu testenden Systems.

Es hat sich gezeigt, daß unter Verwendung einer formalen Semantik ein UML-basierter Test trotz des semiformalen Charakters der UML möglich ist. Die entwickelten Techniken sind ein Fortschritt im UML-basierten Test. Bisher gibt es kaum Testansätze, die mehr als eine Sicht berücksichtigen. Techniken, die sich auf nur einen Diagrammtyp stützen, haben viele Teilprobleme gelöst, es geht jedoch ein Teil der Information, der in anderen Diagrammtypen definiert ist, beim Test verloren. Obwohl die Integration weiterer Sichten den in der vorliegenden Arbeit vorgestellten Ansatz komplettiert, zeigt sich schon jetzt, daß der Test effektiver ist, als wenn nur eine der betrachteten Sichten berücksichtigt wird.

Im folgenden Kapitel wird ein Ausblick auf mögliche Weiterentwicklungen der vorgestellten Techniken und des Testsystems UT^3 gegeben.

Kapitel 18

Ausblick

Dieses abschließende Kapitel soll einen Ausblick geben auf Weiterentwicklungen der in der vorliegenden Arbeit vorgestellten Techniken.

Neben den zwei Themengebieten, den UML-basierten Testverfahren und der aspektorientierten Testcodeintegration, wird auch das implementierte Testsystem UT^3 in den folgenden Abschnitten betrachtet.

18.1 UML-basierte Testverfahren

Die vorgestellten UML-basierten Testverfahren berücksichtigen bisher nur eine Auswahl der in der UML 2.0 enthaltenen Diagrammtypen. Primäres Ziel ist also die Berücksichtigung weiterer Diagrammtypen sowohl für die Testfallgenerierung als auch für die Erzeugung des Testorakels. Die Analyse in Kapitel 5 bietet eine gute Basis für die Entscheidung, welche Diagrammtypen neben den bereits verwendeten zukünftig für den Test genutzt werden sollen.

Erweiterung der Testfallgenerierung. Als primär zu unterstützender Diagrammtyp sei das Aktivitätsdiagramm erwähnt. Um dieses Diagramm für den Test nutzen zu können, sind jedoch entweder Verfeinerungsschritte wie in [15] oder ein semantisch präziser Bezug zu der zu testenden Implementierung nötig. Aktivitätsdiagramme können die in der vorliegenden Arbeit vorgestellten Techniken um den Systemtest erweitern.

Erweiterung der Testdatenermittlung. Die Testdatenermittlung ist bisher nur rudimentär implementiert. Um dies zu ändern, ist insbesondere der Einsatz des Objektdiagramms sinnvoll, da dieses bereits Objekte mit aktuellen Parametern und ihren Beziehungen zueinander modelliert und in der Regel typische Objektkonstellationen beschreibt. Aber auch die Multiplizitäten aus dem Klassendiagramm können für die Testdatenermittlung genutzt werden. Besonders vielversprechend sind zudem OCL-Constraints. Ein Verfahren ähnlich dem in [12] vorgestellten läßt sich auch auf Basis von OCL-Constraints realisieren.

Erweiterung des Testorakels. In das Testorakel müssen weitere OCL-Constraints integriert werden als die bisher berücksichtigten, so z.B. Abhängigkeiten an Assoziationen.

Auch die Erweiterung des Testorakels um die Sequenz, mit der das modellierte System auf einen Stimulus reagiert, ist geplant. In diesem Fall werden Sequenzdiagramme nicht nur für die Testfallgenerierung genutzt, sondern auch für das Testorakel.

18.2 Aspektorientierte Programmiertechniken

Bei den aspektorientierten Programmiertechniken sind besonders zwei aktuelle Entwicklungen von Interesse.

Einerseits wird das ObjectTeams-Programmiermodell und mit ihm die Sprache ObjectTeams/Java ständig weiterentwickelt. Neue Funktionalität sowie ein vergrößerter Sprachumfang dienen auch einem besseren Test. Viele der Nachteile, die eine Implementierung in ObjectTeams/Java z.B. im Gegensatz zu Implementierungen in AspectJ aufwies, wie z.B. die fehlende Jointpoint-Sprache, sind in der aktuellen Version bereits vorhanden oder für zukünftige Versionen geplant.

Andererseits erfährt auch die populäre aspektorientierte Sprache AspectJ einige Änderungen durch den Compiler *abc* [1], der eine flexible Instrumentierung auf der Basis von AspectJ erlaubt, dabei aber im Gegensatz zum normalen AspectJ-Compiler die Definition neuer Joinpoints erlaubt.

Weiterentwicklungen in Object Teams. Aktuelle Entwicklungen der Sprache ObjectTeams/Java sind:

- Das Guard-Konzept [48].

  Das Guard-Konzept realisiert die Abhängigkeit einer Bindung oder der Aktivierung einer Rolle oder eines Teams von einem Prädikat.

  Dadurch ist eine effizientere Implementierung der Zustandsdiagramme möglich, bei denen nicht mehr pro Hierarchieebene ein Team, sondern nur noch eine Rolle in einem Team erzeugt werden muß.

- Die Möglichkeit, Call-In- und Call-Out-Bindungen nicht nur zu Methoden, sondern auch zu Attributen vorzunehmen.

  Bisher mußten alle Zugriffe auf Attribute durch Methoden gekapselt werden. Dies ist bei Einsatz dieses Konzepts nicht mehr nötig, sondern Attribute sind direkt aus einer Rolle heraus zugreifbar.

- Joinpoint-Sprache.

  Welche Möglichkeiten die Joinpoint-Sprache von ObjectTeams/Java für den Test bietet, ist noch nicht absehbar, da sie sich gerade in der Diskussion befindet. Jedoch soll auf jeden Fall geprüft werden, welche weiteren Vorteile sich aus der Joinpoint-Sprache beim Test ergeben.

Neben den Testorakeln sollen zukünftig auch die Testtreiber mit Hilfe aspektorientierter Programmiertechniken in das zu testende System eingebunden werden. Beispiele für eine solche Einbindung finden sich in [113, 103], dort allerdings in der aspektorientierten Sprache AspectJ realisiert.

Instrumentierung mit *abc*. Bisher erlauben aspektorientierte Sprachen nur methodenweise Instrumentierung, d.h., zusätzlicher Code kann vor oder nach einer Methodenausführung, nicht jedoch im Methodenrumpf, eingewoben werden. Klassische Instrumentierungstechniken, z.B. für die Kontrollflußüberdeckungsmessung, benötigen jedoch die Möglichkeit der Instrumentierung pro Anweisung im Quellcode.

Mit dem flexiblen Compiler *abc* ist eine anweisungsweise Instrumentierung möglich, da neue Joinpoints einfach und flexibel definiert werden können. Dies eröffnet neue Perspektiven beim Einsatz aspektorientierter Techniken im Testprozeß und weist den Weg zu einem vollständigen Ersatz klassischer Instrumentierungstechniken durch aspektorientierte Instrumentierungstechniken.

18.3 Testsystem UT^3

Beim Testsystem UT^3 handelt es sich um eine prototypische Implementierung der in der vorliegenden Arbeit vorgestellten Techniken. Um das Testsystem einem größeren Nutzerkreis zu öffnen, sind primär zwei Erweiterungen vorzunehmen.

Erstens fehlt dem Testsystem eine graphische Benutzeroberfläche. Alle Funktionen werden bisher über die Kommandozeile aufgerufen.

Zweitens ist das Testsytem in eine Entwicklungsumgebung, z.B. Eclipse zu integrieren, die bereits mehrere Editoren für UML-Modelle zur Verfügung stellt und so Quellcode und Modell unter einem Dach vereint.

Verzeichnisse

Literaturverzeichnis

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, Julian Tibble. abc: An Extensible AspectJ Compiler. In *International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, Illinois, USA, 2005.

[2] Paul Baker, Paul Bristow, Clive Jervis, David King, Bill Mitchell. Automatic Generation of Conformance Tests from Message Sequence Charts. In *Telecommunications and beyond: The BroaderApplicability of SDL and MSC, 3rdInternational Workshop (SAM)*, *Lecture Notes in Computer Science*, Band 2599, Aberystwyth, Großbritanien, 2002. Springer-Verlag.

[3] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillman, Margus Veanes. Model-Based Testing with AsmL.NET. In *1st European Conference on Model-Driven Software Engineering*, Nürnberg, Deutschland, 2003.

[4] Francesca Basanieri, Antonia Bertoli, Eda Marchettia, Alberto Ribolini, Gaetano Lombardi, Giovanni Nucera. An Automated Test Strategy Based on UML Diagrams. In *Ericsson Rational User Conference*, Upplands Vasby, Sschweden, 2001.

[5] Francesca Basanieri, Antonia Bertolino. A Practical Approach to UML-Based Derivation of Integration Tests. In *Software Quality Week Europe (QWE)*, Brüssel, Belgien, 2000.

[6] Boris Beizer. *Software Testing Techniques*. International Thompson Computer Press, 1990.

[7] Gilles Bernot, Marie Claude Gaudel, Bruno Marre. Software Testing Based on Formal Specifications: A Theory and a Tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[8] Robert V. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, 37(9):81–102, 1994.

[9] Robert V. Binder. The FREE Approach to Testing Object-Oriented Software: An Overview. *http://www.rbsc.com/pages/FREE.html*, 1996.

[10] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison-Wesley, 1999.

[11] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Pearson Education, 2nd edition, 1993.

[12] Chandrasekhar Boyapati, Sarfraz Khurshid, Darko Marinov. Korat: Automated Testing Based on Java Predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, ACM SIGSOFT, Rom, Italien, 2002. Association for Computing Machinery (ACM).

[13] Lionel C. Briand, Wojciech Dzidek, Yvan Labiche. Using Aspect-Oriented Programming to Instrument OCL Contracts in Java. Technischer Report, Carlton University, Kanada, 2004.

[14] Lionel C. Briand, Yvan Labiche. A UML-Based Approach to System Testing. In *4th International Conference on the Unified Modeling Language (UML)*, Toronto, Kanada, 2001.

[15] Lionel C. Briand, Yvan Labiche. A UML-Based Approach to System Testing. *Journal of Software and Systems Modeling, Springer-Verlag*, 1(1):10–42, 2002.

[16] Jean-Michel Bruel, Joao Araújo, Ana Moreira, Albert Royer. Using Aspects to Develop Built-In Tests for Components. In *AOSD Modeling with UML Workshop, 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, Kalifornien, USA, 2003.

[17] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engeneering*, SE-4(3):178–187, 1978.

[18] clover-Homepage. http://www.cenqua.com/clover/.

[19] Pascal Constanza, Günter Kniesel, Armin B. Cremers. Lava: Spracherweiterungen für Delegation in Java. In *Java Informationstage (JIT)*, Informatik aktuell, Düsseldorf, Deutschland, 1999. Gesellschaft für Informatik (GI), Springer Verlag.

[20] CUP-Homepage. http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[21] Dagstuhl-Seminar Perspectives of Model-Based Testing. http://www.dagstuhl.de/04371/, 2004.

[22] Zhen Ru Dai, Jens Grabowski, Helmut Neukirchen, Holger Pals. From Design to Test with UML - Applied to a Roaming Algorithm for Bluetooth Devices. In *Testing of Communicating Systems. Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems (TestCom2004)*, number 2978 in LNCS, Oxford, UK, 2004. Springer.

[23] Morgan Deters, Ron K. Cytron. Introduction of Program Instrumentation using Aspects. In *Workshop of Advanced Separation of Concerns in Object-Oriented Systems, 16th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM Sigplan Notices, Tampa Bay, Florida, USA, 2001.

[24] Edsger W. Dijkstra. Structured Programming. In *Software Engineering Techniques*, Rom, Italien, 1970. NATO Science Committee.

[25] Dresden OCL-Homepage. http://dresden-ocl.sourceforge.net/.

[26] Nghia Dang Duc. Entwicklung eines generischen Testtreibers für Java-Klassen mit Hilfe von Java-Reflections. Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, 2003.

[27] Andre Engels, Loe M. G. Feijs, Sjouke Mauw. Test Generation for Intelligent Networks Using Model Checking. In *3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *Lecture Notes in Computer Science*, Band 1217, Enschede, Niederlande, 1997. Springer-Verlag.

[28] Rik Eshuis, Roel Wieringa. Requirements Level Semantics for UML Statecharts. In Scott F. Smith, Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV, 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, *IFIP Conference Proceedings*, Band 177, Stanford, California, USA, 2000. Kluwer.

[29] European Telecommunications Standards Institute, http://www.etsi.org/ptcc/ptccttcn3.htm. *Testing and Test Control Notation (TTCN-3).*

[30] Robert E. Filman, Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, Minneapolis, Minnesota, USA, 2000.

[31] Robert E. Filman, Klaus Havelund. Source-Code Instrumentation and Quantification of Events. In *Workshop on Foundations of Aspect-Oriented Languages, 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, Niederlande, 2002.

[32] Falk Fraikin, Matthias Hamburg, Stefan Jungmayr, Thomas Leonhardt, Andreas Schönknecht, Andreas Spillner, Mario Winter. Die trügerische Sicherheit des grünen Balkens. *ObjektSpektrum*, (1):25–29, 2004.

[33] Falk Fraikin, Thomas Leonhardt. SeDiTeC - Testing Based on Sequence Diagram. In *17th IEEE International Conference on Automated Software Engineering (ASE)*, Edinburgh, Großbritannien, 2002.

[34] Viktor Friesen, Andre Nordwig, Matthias Weber. Object-Oriented Specification of Hybrid Systems using UML^h and ZimOO. In *11th International Conference on the Z Formal Method (ZUM)*, *Lecture Notes in Computer Sciences*, Band 1493, Berlin, Deutschland, 1998. Springer-Verlag.

[35] Mario Friske. Testfallerzeugung aus Use-Case-Beschreibungen. In *21. Treffen des Arbeitskreises Testen, Analysieren und Verifizieren von Software (TAV)*, *Softwaretechnik-Trends*, Band 24, Berlin, Deutschland, 2004. Gesellschaft für Informatik (GI).

[36] Peter Fröhlich, Johannes Link. Automated Test Case Generation from Dynamic Models. In *14th European Conference on Object-Oriented Programming (ECOOP)*, *Lecture Notes in Computer Sciences*, Band 1850, Sophia Antipolis und Cannes, Frankreich, 2000. Springer-Verlag.

[37] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1997.

[38] Jens Grabowski. *Test Case Generation and Test Case Specification Based on Message Sequence Charts*. Dissertation, Universität Bern, Schweiz, 1994.

[39] Klaus Grimm. *Systematisches Testen von Software: Eine neue Methode und eine effektive Teststrategie*. Dissertation, Technische Universität Berlin, Deutschland, 1995.

[40] Radu Grosu, Manfred Broy, Bran Selic, Gheorghe Stefanescu. Towards a Calculus for UML-RT Specifications. In *7th OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications, 13th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vancouver, Canada, 1998.

[41] Ali Hamie. Towards Verifying Java Realizations of OCL-Constrained Design Models Using JML. In *6th IASTED International Conference on Software Engineering andApplications (SEA)*, MIT, Cambridge, Massachusetts, USA, 2002.

[42] Ali Hamie. Translating the Object Constraint Language into JML. In *19th ACM Symposium on Applied Computing (SAC)*, Nikosia, Zypern, 2004.

[43] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, (8):231–274, 1987.

[44] Jean Hartmann, Marlon Vieira, Herb Foster, Axel Ruder. UML-based Test Generation and Execution. In *21. Treffen des Arbeitskreises Testen, Analysieren und Verifizieren von Software (TAV)*, *Softwaretechnik-Trends*, Band 24, Berlin, Deutschland, 2004. Gesellschaft für Informatik (GI).

[45] Øystein Haugen, Ketil Stølen. STAIRS: Steps to Analyze Interactions with Refinement Semantics. In *6th International Conference on the Unified Modeling Language (UML)*, *Lecture Notes in Computer Science*, Band 2863. Springer-Verlag, 2003.

[46] Stephan Herrmann. Composable Design with UFA. In *Workshop on Aspect-Oriented Modeling with UML, 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, Niederlande, 2002.

[47] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World (Net.ObjectDays Conference)*, *Lecture Notes In Computer Science*, Band 2591, Erfurt, Deutschland, 2002. Springer-Verlag.

[48] Stephan Herrmann, Christine Hundt, Katharina Mehner, Jan Wloka. Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation. In *Dynamics Aspects Workshop (DAW), International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, Illinois, USA, 2005.

[49] Hyoung Seok Hong, Yong Rae Kwon, Sung Deok Cha. Testing of Object-Oriented Programs Based on Finite State Machines. In *2nd Asia-Pacific Software Engineering Conference (APSEC)*, Brisbane, Australien, 1995.

[50] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, Sung Deok Cha. Automatic Test Generation from Statecharts Using Model Checking. In *1st International Workshop on Formal Approaches to Testing of Software (FATES)*, Aalborg, Dänemark, 2001.

[51] Hans-Martin Hörcher. Testfallspezifikation mittels Message Sequence Charts. In *15. Treffen des Arbeitskreises Testen, Analysieren und Verifizieren von Software (TAV)*, *Softwaretechnik-Trends*, Band 20, Sankt-Augustin, Deutschland, 2000. Gesellschaft für Informatik (GI).

[52] Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology, Standard 610.12-1990*, 1990.

[53] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1993.

[54] JFlex-Homepage. http://jflex.de/.

[55] JML-Homepage. http://www.jmlspecs.org.

[56] Supaporn Kansomkeat, Wanchai Rivepiboon. Automated-Generating Test Case Using UML Statechart Diagrams. In *Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology (SAICSIT)*, Fourways, Südafrika, 2003.

[57] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, *Lecture Notes in Computer Science*, Band 1241, Jyväskylä, Finnland, 1997. Springer-Verlag.

[58] Sun-Woo Kim. Testing Object-Oriented Programs Using Mutation Techniques. 1st Year Qualifying Dissertation, University of York, Department of Computer Science, Großbritanien, 1998.

[59] Young Gon Kim, Hyoung Seok Hong, Doo-Hwan Bae, Sung-Deok Cha. Test Cases Generation from UML State Diagrams. *IEE Proceedings Software*, 146(4):187–192, 1999.

[60] Shekhar H. Kirani, Wei-Tek Tsai. Method Sequence Specification and Verification of Classes. *Journal of Object-Oriented Programming*, 7(6):28–38, 1994.

[61] Günter Kniesel. Objects Don't Migrate: Perspectives on Objects with Roles. Technischer Report, Universiät Bonn, Deutschland, 1996.

[62] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Object Technology Series. Addison-Wesley, 2000.

[63] David C. Kung, Nimish Suchak, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Cris Chen. On Object State Testing. In *18th International Computer Software and Applications Conference (COMPSAC)*, Taipeh, Taiwan, 1994. IEEE Computer Society Press.

[64] Diego Latella, István Majzik, Mieke Massink. Towards a Formal Operational Semantics for UML Statechart Diagrams. In Paolo Ciancarini, Alessandro Fantechi, Roberto Gorrieri, editors, *Formal Methods for Open Object-Based Distributed Systems III, 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, *IFIP Conference Proceedings*, Band 139, Florenz, Italien, 1999. Kluwer.

[65] Lava-Homepage. http://javalab.cs.uni-bonn.de/research/darwin/.

[66] Gary T. Leavens, Albert L. Baker, Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, Ian Simmonds, editors, *Behavioral Specifications of Businesses and System*, pages 175–188. Kluwer Academic Publishers, 1999.

[67] Eckard Lehmann. *Time Partition Testing: Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen.* Dissertation, Technische Universität Berlin, Deutschland, 2003.

[68] Nicholas Lesiecki. Test Flexibility with AspectJ and Mock Objects. http://www-106.ibm.com/developerworks/java/library/j-aspectj2/?loc=j.

[69] Xiaoshan Li, Zhiming Liu, Jifeng He. A Formal Semantics of UML Sequence Diagram. Technischer Report, United Nations University, International Institute for Software Technology (UNU-IIST), Macao, 2004.

[70] Peter Liggesmeyer. *Modultest und Modulverifikation - State of the Art.* BI-Wissenschaftsverlag, 1990.

[71] Peter Liggesmeyer. *Softwarequalität - Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, 2002.

[72] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, Zheng Guoliang. Generating Test Cases from UML Activity Diagram Based on Gray-Box Method. In *11th Asia-Pacific Software Engineering Conference (APSEC)*, Busan, Korea, 2004. IEEE Computer Society Press.

[73] Barbara Liskov, Jeanette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[74] Tina Low. Designing, Modelling and Implementing a Toolkit for Aspect-oriented Tracing (TAST). In *Workshop on Aspect-Oriented Modeling with UML, 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, Niederlande, 2002.

[75] Tim Mackinnon, Steve Freemann, Philip Craig. Endo-Testing: Unit Testing with Mock Objects. In *Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP)*, Cagliari, Italien, 2000.

[76] Daniel Mahrenholz, Olaf Spinczyk, Wolfgang Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC)*, Crystal City, Virginia, USA, 2002.

[77] Brian Marick. *The Craft of Software Testing (Subsystem Testing).* Prentice Hall, 1995.

[78] Sjouke Mauw, Michel A. Reniers. High-Level Message Sequence Charts. In *SDL'97: Time for Testing - SDL, MSC and Trends, Proceedings of the Eighth SDL Forum*, Evry, Frankreich, 1997.

[79] John D. McGregor, David A. Sykes. *A Practical Guide to Testing Object-Oriented Software.* Object Technology Series. Addison-Wesley, 2001.

[80] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[81] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[82] Object Teams-Homepage. http://www.objectteams.org.

[83] ObjectTeams/Java-Sprachdefinition. http://www.objectteams.org/def/0.8/index.html.

[84] *Object Constraint Language Specification, Version 2.0.* Object Management Group (OMG), *http://www.uml.org*, 2004.

[85] Jeff Offutt, Aynur Abdurazik. Generating Tests from UML Specifications. In *2nd International Conference on the Unified Modeling Language (UML)*, Fort Collins, Colorado, USA, 1999.

[86] Jeff Offutt, Aynur Abdurazik. Using UML Collaboration Diagrams for Static Checking and Test Generation. In *3rd International Conference on the Unified Modeling Language (UML)*, York, Großbritanien, 2000.

[87] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, Paul Ammann. Generating Test Data From State-based Specifications. *Journal of Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[88] Object Management Group-Homepage. http://www.omg.org.

[89] Thomas J. Ostrand, Marc J. Blacer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, 1988.

[90] Jan Overbeck. *Integration Testing for Object-Oriented Software.* Dissertation, Technische Universität Wien, Östereich, 1994.

[91] Andreas Reuys, Sacha Reis, Erik Kamsties, Klaus Pohl. Derivation of Domain Test Scenarios from Activity Diagrams. In *International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing (PLEES)*, Erfurt, Deutschland, 2003.

[92] Joel Richardson, Peter Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In *International Conference on Management of Data*, Denver, Colorado, USA, 1991. SIGMOD, Association for Computing Machinery (ACM).

[93] Mark Richters, Martin Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In *AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, Kalifornien, USA, 2003.

[94] Matthias Riebisch, Ilka Philippow, Marco Götze. UML-Based Statistical Test Case Generation. In *Objects, Components, Architectures, Services, and Applications for a Networked World (Net.ObjectDays Conference)*, *Lecture Notes in Computer Science*, Band 2591, Erfurt, Deutschland, 2002. Springer-Verlag.

[95] James R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, William Premerlani. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[96] Peter Rüppel. Besondere Aspekte beim objektorientierten Test. In *Informationstechnologien für Wirtschaft und Verwaltung (INFO)*, Potsdam, Deutschland, 1996.

[97] Peter Rüppel. *Ein generisches Werkzeug für den objektorientierten Softwaretest.* Dissertation, Technische Universität Berlin, Deutschland, 1997.

[98] Dirk Seifert, Steffen Helke, Thomas Santen. Test Case Generation for UML Statecharts. In *Perspectives of System Informatics (PSI)*, *Lecture Notes In Computer Science*, Band 2890, Novosibirsk, Russland, 2003. Springer-Verlag.

[99] Harry M. Sneed, Mario Winter. *Testen objektorientierter Software: Das Praxishandbuch für den Test objektorientierter Client/Server-Systeme.* Carl Hanser Verlag, 2002.

[100] Dehla Sokenou. Entwicklung eines Werkzeugs zur Unterstützung zustandsbasierter Testverfahren für JAVA-Klassen. Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, 1998.

[101] Dehla Sokenou. Ein Werkzeug zur Unterstützung zustandsbasierter Testverfahren für Java-Klassen. In *13. Treffen des Arbeitskreises Testen, Analysieren und Verifizieren von Software (TAV)*, *Softwaretechnik-Trends*, Band 19, München, Deutschland, 1999. Gesellschaft für Informatik (GI).

[102] Dehla Sokenou, Stephan Herrmann. Using Object Teams for State-Based Class Testing. Technischer Report, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Deutschland, 2004.

[103] Dehla Sokenou, Matthias Vösgen. FlexTest: An Aspect-Oriented Framework for Unit Testing. In *2nd International Workshop on Software Quality (SOQUA), Net.ObjectDays*, Lecture Notes in Computer Science, Erfurt, Deutschland, erscheint 2005.

[104] J. Mike Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[105] Harmen Sthamer, Joachim Wegener, Andre Baresel. Using Evolutionary Testing to Improve Efficiency and Quality in Software Testing. In *2nd Asia-Pacific Conference on Software Testing, Analysis & Review (AsiaStar)*, Melbounre, Australien, 2002.

[106] Harald Störrle. Assert, Negate and Refinement in UML 2 Interactions. In *Workshop on Critical Systems Development with UML, 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, Kalifornien, USA, 2003.

[107] Harald Störrle. Semantics of Interactions in UML 2.0. In *IEEE Symposium on Visual Languages und Formal Methods (VLFM), Human-Centric Computing Languages and Environments (HCC)*, Auckland, Neuseeland, 2003.

[108] Jan Tretmans. Test Generation with Inputs, Ouputs, and Quiescence. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *Lecture Notes in Computer Science*, Band 1055, Passau, Deutschland, 1996. Springer-Verlag.

[109] *Unified Modeling Language Specification, Version 1.5.* Object Management Group (OMG), *http://www.uml.org*, 2001.

[110] *Unified Modeling Language Specification (Infrastructure and Superstructure Specification), Version 2.0.* Object Management Group (OMG), *http://www.uml.org*, 2004.

[111] *UML 2.0 Testing Profile Specification, Version 1.0.* Object Management Group (OMG), *http://www.omg.org*, 2004.

[112] Matthias Veit, Stephan Herrmann. Model-View-Controller and Object Teams: A Perfect Match of Paradigms. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, Boston, Massachusetts, USA, 2003.

[113] Matthias Vösgen, Dehla Sokenou. Aspektorientierte Programmiertechniken im Unit-Testen. *Informatik - Forschung und Entwicklung*, erscheint 2005.

[114] Joachim Wegener, Harmen Sthamer, Hartmut Pohlheim. Testing the Temporal Behavior of Real-Time Tasks using Extended Evolutionary Algorithms. In *7th European Conference on Software Testing, Analysis and Review (EuroSTAR)*, Barcelona, Spanien, 1999.

[115] Thomas Wierczoch. Erstellung testbarer UML-Modelle zur Unterstützung der Automatisierung von Klassentests für objektorientierte Systeme. Diplomarbeit, Technische Universität Berlin, 2002.

[116] Mario Winter. *Qualitätssicherung für objektorientierte Software - Anforderungsermittlung und Test gegen die Anforderungsspezifikation.* Dissertation, Fernuniversität Hagen, Deutschland, 1999.

[117] Frank Xia, Gautam S.Kane. Defining the Semantics of UML Class and Sequence Diagrams for Ensuring the Consistency and Executablity of OO Software Specification. In *1st International Workshop on Automated Technology for Verification and Analysis (ATVA)*, Taipeh, Taiwan, 2003.

[118] Guoqiung Xu, Zongyuan Yang, Haitao Huang. A Basic Model for Aspect-Oriented Unit Testing. www.cs.ecnu.edu.cn/sel/ harryxu/research/ papers/fates04_aspect-oriented

Liste der Definitionen

Softwaretest 9

Fehler 10

Fehlerursache 10

Falsch-positives Testurteil, falsch-negatives Testurteil 16

Protocol State Machine 67

Transition, Quellzustand, Zielzustand 68

Eingehende Transition, ausgehende Transition 68

Auslösendes Ereignis, ausgelöste Transition 68

Sequenzdiagramm 73

Negative und positive Sequenzen 74

Klasse 74

Testfall 77

Regulärer Testfall 78

Komplementärer Testfall 78

Korrespondenz 114

Lösbarer Nichtdeterminismus 131

Abbildungsverzeichnis

2.1 Testaktivitäten im Testprozeß . . . 11
2.2 Fehler in Testorakel und zu testendem System . . . 15

4.1 UML-Standarddiagrammtypen . . . 26
4.2 Transformation komplexer Sequenzdiagramme . . . 33
4.3 Typen der OCL-Standardbibliothek . . . 35
4.4 Softwareverträge in OCL . . . 37

5.1 Aggregation mit Multiplizitäten als obere und untere Grenze . . . 40
5.2 Objektkonstellation für (a) die untere Grenze und (b) die obere Grenze . . . 40
5.3 Zustandsdiagramm der Klasse `Account` . . . 47
5.4 Vor- und Nachbedingung der Methode `withdraw` in OCL . . . 47
5.5 Zustandsinvarianten der Zustände `active`, `credit` und `debit` . . . 48

7.1 Schematische Darstellung eines Testsystems . . . 58
7.2 Schematische Darstellung des Testsystems UT^3 . . . 58
7.3 Klassen des Fallbeispiels *Banksystem* . . . 60
7.4 Klasse `Account`: a) mit 4 Observer-Methoden, b) mit 2 Observer-Methoden . . . 60
7.5 Positives und negatives Sequenzdiagramm . . . 61
7.6 Zustandsdiagramm für die Klasse `Account` . . . 62
7.7 Klasseninvariante für `Account` . . . 62
7.8 Vor- und Nachbedingung für `withdraw` der Klasse `Account` . . . 62
7.9 Klasseninvariante für `SavingsAccount` . . . 63
7.10 Vor- und Nachbedingung für `withdraw` der Klasse `SavingsAccount` . . . 63

9.1 Beispiel: Zustandsdiagramm . . . 78
9.2 Positives und negatives Sequenzdiagramm . . . 79
9.3 Reguläre (positive) Testfälle für den Klassentest von `Account` . . . 80
9.4 Reguläre positive Testfälle für den Integrationstest . . . 81
9.5 Regulärer negativer Testfall für den Integrationstest . . . 81
9.6 Ausgewertete Operationen über Basistypen . . . 83

10.1 Zustandsdiagramm der Klasse `SavingsAccount` . . . 86
10.2 Klasseninvariante für `SavingsAccount` . . . 86
10.3 Vor- und Nachbedingung für `withdraw` (oben) und `deposit` (unten) der Klasse `SavingsAccount` . . . 86
10.4 Zustandsdiagramm der Klasse `Account` . . . 87
10.5 Vor- und Nachbedingung für `withdraw` der Klasse `Account` . . . 87
10.6 Um Vor- und Nachbedingungen angereichertes Zustandsdiagramm der Klasse `SavingsAccount` . . . 88
10.7 Zustandsinvarianten der Zustände `open`, `credit`, `debit` und `empty` . . . 90
10.8 Aus dem Zustandsdiagramm extrahierte Vor- und Nachbedingung für `withdraw` (Variante 1) . . . 92

10.9 Aus dem Zustandsdiagramm extrahierte Vor- und Nachbedingung für `withdraw` (Variante 2) . . . 93
10.10 Bewertung der Varianten im Hinblick auf das Testurteil . . . 93
10.11 Erweiterte Vor- und Nachbedingung für `withdraw` (Variante 1) . . . 94
10.12 Erweiterte Vor- und Nachbedingung für `withdraw` (Variante 2) . . . 95

12.1 Beispiel Logging ohne aspektorientierte Programmierung . . . 104
12.2 Beispiel Logging mit aspektorientierter Programmierung . . . 105
12.3 Beispiel Personenhierarchie ohne rollenbasierte Programmierung . . . 106
12.4 Beispiel Personen mit rollenbasierter Programmierung . . . 107
12.5 Beispiel: Bank und Bank-Team . . . 108
12.6 Beispiel: Code des Bank-Teams und der Klassen des Bank-Systems . . . 109
12.7 ObjectTeams/Java-Schablone . . . 110

13.1 Korrespondenz zwischen Rollen und zu testenden Objekten . . . 115

14.1 Ausführbares Zustandsdiagramm und zu testendes Objekt. . . . 117
14.2 Ausführbare Zustandsdiagramme und zu testendes System. . . . 118
14.3 Beispiel: Einfaches Zustandsdiagramm . . . 118
14.4 Implementierung eines einfachen Zustandsdiagramms . . . 119
14.5 Beispiel: Hierarchisches Zustandsdiagramm . . . 121
14.6 Beispiel: Hierarchisches Zustandsdiagramm mit Trennung der Hierarchieebenen . 121
14.7 Korrespondenz zwischen hierarchischem Zustandsdiagramm und zu testendem Objekt . . . 122
14.8 Implementierung des Toplevelzustands `Account` . . . 123
14.9 Implementierung des Sublevelzustands `open` . . . 124
14.10 Deaktivierung des aktiven Unterzustandes von `Account` . . . 126
14.11 Beispiel: Hierarchisches Zustandsdiagramm mit Historie . . . 127
14.12 Implementierung des Sublevelzustands `open` ohne Historie . . . 128
14.13 Beispiel: Hierarchisches Zustandsdiagramm mit Parallelität . . . 129
14.14 Implementierung des Zustands `open` mit parallelen Unterzuständen . . . 130
14.15 Beispiel: Zustandsdiagramm mit (lösbarem) Nichtdeterminismus . . . 131
14.16 Implementierung des Zustands `open` mit nichtdeterministischer Auswahl der Transitionen . . . 132

15.1 Vorrangregeln in OCL und Java . . . 138
15.2 Vor- und Nachbedingung für `withdraw` der Klasse `Account` . . . 139
15.3 Implementierung von Vor- und Nachbedingungen . . . 140

A.1 Das Paket `ut.ut_base` . . . 169
A.2 Das Paket `ut.ut_base.statechart` . . . 170
A.3 Das Paket `ut.ut_base.sequences` . . . 170
A.4 Das Paket `ut.ut_base.ocl` . . . 171
A.5 Das Paket `ut.ut_ora` . . . 172

Anhang

Anhang A

Struktur des Testsystems UT^3

Das Testsystem ist in Java implementiert und besteht aus vier Java-Paketen. Das Paket `ut.ut_base` ist für die Datenstrukturen zuständig. Das Paket `ut.ut_tfgen` implementiert die Algorithmen für die Testfallgenerierung, das Paket `ut.ut_ora` die Algorithmen für die Erzeugung des Testorakels. Im Paket `ut.ut_main` befinden sich die Hauptprogramme für die Testfallgenerierung, die Testorakelerzeugung und die Testausführung. Dieses Paket wird in zukünftigen Versionen des Testsystems ersetzt durch das Paket `ut.ut_gui`, das eine graphische Oberfläche zur Verfügung stellen wird.

A.1 Das Paket `ut.ut_base`

Das Paket `ut.ut_base` ist das Basispaket, das alle Datenstrukturen definiert. Für jede Datenstruktur gibt es ein eigenes Subpaket (siehe Abbildung A.1). Zustandsdiagramme werden im Paket `ut.ut_base.statecharts` definiert, Sequenzdiagramme im Paket `ut.ut_base.sequences` und OCL-Constraints im Paket `ut.ut_base.ocl`. Ergänzt werden die Subpakete für die UML-Datenstrukturen um das Paket `ut.ut_base.basics` für Datenstrukturen, die von den anderen Paketen benötigt werden, bei denen es sich jedoch nicht um UML-Datenstrukturen handelt, und um das Paket `ut.ut_base.ot` zur Definition einer Datenstruktur zur Repräsentation von ObjectTeams/Java-Klassen.

A.1.1 Das Paket `ut.ut_base.basics`

Im Paket `ut.ut_base.basics` werden einige grundlegende Datenstrukturen implementiert, die benötigt werden, z.B. die Datenstruktur `Queue`.

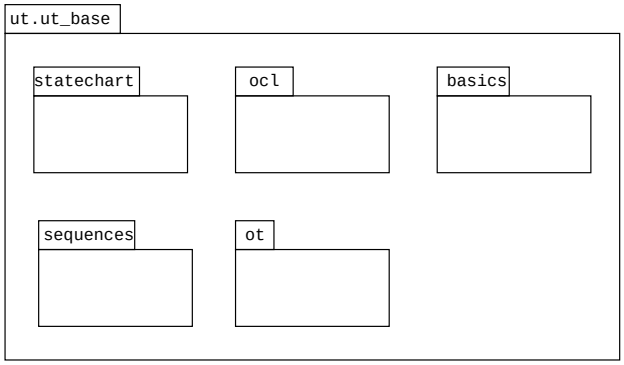


Abbildung A.1: Das Paket `ut.ut_base`

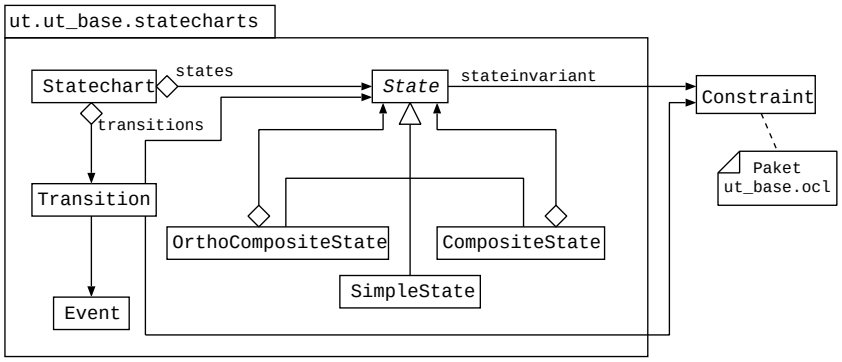


Abbildung A.2: Das Paket `ut.ut_base.statechart`

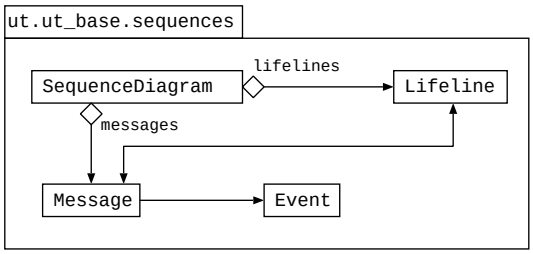


Abbildung A.3: Das Paket `ut.ut_base.sequences`

A.1.2 Das Paket `ut.ut_base.statecharts`

Das Paket `ut.ut_base.statecharts` definiert die Datenstruktur für ein Zustandsdiagramm (siehe Abbildung A.2). Es enthält die Klasse `Statechart` als Klasse, die alle Teile eines Zustandsdiagramms (Zustände und Transitionen) enthält. In dieser Klasse sind alle grundlegenden Algorithmen auf Zustandsdiagrammen implementiert (siehe Kapitel 8.2).

Zustände werden repräsentiert durch die abstrakte Klasse `State`. Von dieser Klasse abgeleitet sind die Klassen `SimpleState` als Repräsentant eines einfachen Zustands vom Typ Σ_{simple}, `CompositeState` als Repräsentant eines komponierten Zustands vom Typ Σ_{xor} und `OrthoCompositeState` als Repräsentant eines komponierten Zustands vom Typ Σ_{and}. Jeder Zustand hält eine Referenz auf die Menge der eingehenden und die Menge der ausgehenden Transitionen. Zustandsinvarianten sind als OCL-Constraint vom Typ `Constraint` des Paketes `ut.ut_base.ocl` referenziert.

Transitionen werden repräsentiert durch die Klasse `Transition`. Jede Transition hält eine Referenz auf ihren Quell- und ihren Zielzustand und auf das die Transition auslösende Call-Event. Zudem sind Vor- und Nachbedingungen einer Transition als OCL-Constraint vom Typ `Constraint` des Paketes `ut.ut_base.ocl` referenziert.

A.1.3 Das Paket `ut.ut_base.sequences`

Das Paket `ut.ut_base.statecharts` definiert die Datenstruktur für ein Sequenzdiagramm. Es besteht im Wesentlichen aus drei Klassen (siehe Abbildung A.3). Die Klasse `SequenceDiagramm` repräsentiert ein Sequenzdiagramm. Sie enthält Referenzen zu den Nachrichten, implementiert in der Klasse `Message`, und den Lebenslinien der Objekte, implementiert in der Klasse `Lifeline`. Die grundlegenden Algorithmen auf Sequenzdiagrammen sind ebenfalls in der Klasse `SequenceDiagram` implementiert.

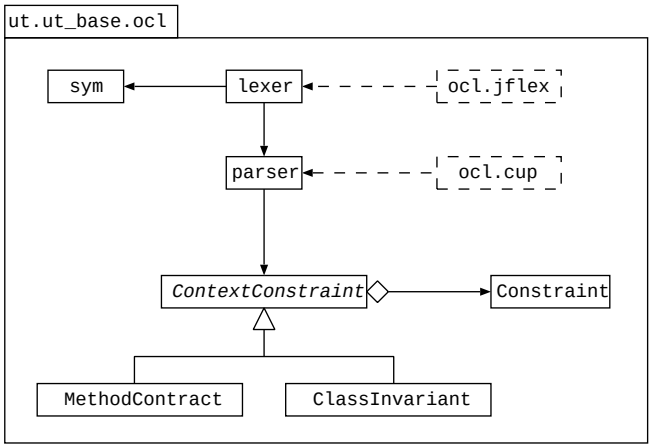


Abbildung A.4: Das Paket `ut.ut_base.ocl`

A.1.4 Das Paket `ut.ut_base.ocl`

Das Paket `ut.ut_base.ocl` stellt Klassen zum Umgang mit OCL-Constraints bereit (siehe Abbildung A.4).

Die Klasse `Constraint` repräsentiert ein einfaches OCL-Constraint in Form einer Aussage ohne Kontextdeklaration. Als einfachste Constraints sind die Aussagen `TRUE` und `FALSE` vordefiniert. Alle Aussagen sowohl in OCL als auch in Java werden in der Klasse `Constraint` abgelegt.

OCL-Constraints mit Kontextdeklaration werden durch die abstrakte Klasse `ContextConstraint` implementiert. Dabei ist die eigentliche Aussage durch die Klasse `Constraint` implementiert und die zusätzliche Kontextinformation in der Klasse `ContextConstraint`. Zwei Unterklassen der Klasse `ContextConstraint` repräsentieren zwei Arten des Kontextes. Die Klasse `MethodContract` definiert Vor- und Nachbedingung von Methoden, die Klasse `ClassInvariant` die Klasseninvariante im Kontext einer Klasse.

OCL-Constraints im Kontext einer Protocol State Machine - Zustandsinvarianten sowie Vor- und Nachbedingungen von Transitionen - werden unter Verwendung der Klasse `Constraint` im Paket `ut.ut_base.statechart` implementiert.

Das Paket enthält zudem die Klassen `sym`, `lexer` und `parser`, die automatisch generiert sind und zum Einlesen und Umwandeln von OCL-Constraints nach Java dienen.

Die Klassen `sym` und `lexer` werden von JFlex [54] unter Verwendung der Datei `ocl.jflex` erzeugt, die die Deklaration aller Terminal- und Nonterminalsymbole der verwendeten OCL-Grammatik enthält (siehe Anhang B).

Die Klasse `parser` wird von Cup [20] unter Verwendung der Datei `ocl.cup` erzeugt, die die in Anhang B vorgestellte Grammatik enthält und die Aktionen für einzelne Matches implementiert.

A.1.5 Das Paket `ut.ut_base.ot`

Das Paket `ut.ut_base.ot` enthält im Wesentlichen die Klasse `OTFrame`, die Methoden zur Generierung von ObjectTeams/Java-Code mit Hilfe einer ObjectTeam-Schablone erlaubt.

A.2 Das Paket `ut.ut_tfgen`

Im Paket `ut.ut_tfgen` gibt es bisher nur die Klasse `TCSequenceStatechart` zur Generierung der Testsequenzen aus Sequenzdiagrammen und Zustandsdiagrammen.

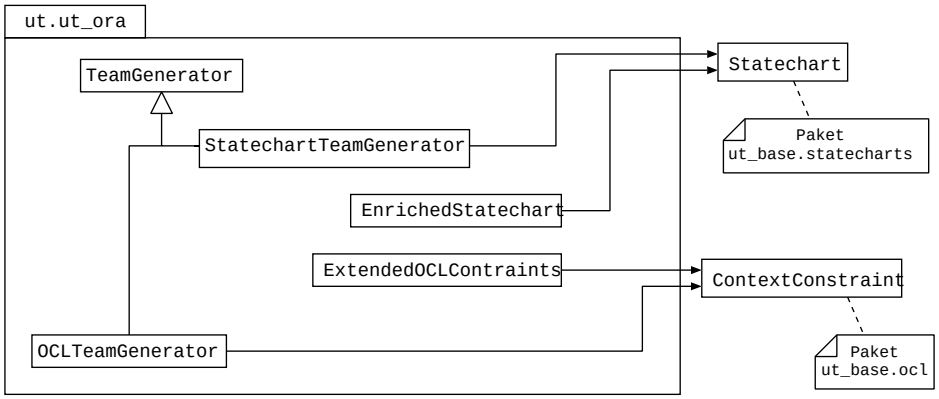


Abbildung A.5: Das Paket `ut.ut_ora`

A.3 Das Paket `ut.ut_ora`

Das Paket `ut.ut_ora` dient zur Erzeugung von Testorakeln im UML und ObjectTeams/Java (siehe Abbildung A.5).

Es enthält die Klasse `EnrichedStatecharts` zur Generierung der angereicherten Zustandsdiagramme und die Klasse `ExtendedOCLConstraints` zur Generierung der erweiterten OCL-Constraints.

Für die Generierung der ObjectTeams stehen die Klasse `TeamGenerator` zur Verfügung. Deren Unterklasse `StatechartTeamGenerator` erzeugt die Teams für die Zustandsdiagramme, deren Unterklasse `OCLTeamGenerator` die Teams für die OCL-Contraints.

A.4 Das Paket `ut.ut_main`

Das Paket `ut.ut_main` stellt eine Reihe von Klassen mit Kommandozeilenbefehlen, also `main`-Methoden zur Verfügung, um die Testfallgenerierung und die Testorakelerzeugung aufzurufen.

Anhang B

Grammatik zum Einlesen von OCL-Constraints

Zum Einlesen der OCL-Constraints wurde die im Folgenden gegebene Grammatik verwendet. Zum Einlesen wurden der Java-Lexer `jflex` [54] und der Java-Parser `cup` [20] verwendet. Die Grammatik wird für das Einlesen von Invarianten, Vor- und Nachbedingungen benutzt. Für Zustandsinvarianten wird der Teil der Grammatik verwendet, der mit `expr_part` beginnt.

```
/* Characters */

LineTerminator    = \r | \n | \r\n
InputCharacter    = [^\r\n]
WhiteSpace        = {LineTerminator} | [ \t\f]
Comment           = "--" {InputCharacter}* {LineTerminator}?
Digit             = 0 | [1-9][0-9]*
Iden              = [A-Za-z_][A-Za-z_0-9]*


/* Terminals  */

CONTEXT           = "context"
PRE               = "pre:"
POST              = "post:"
INV               = "inv:"

TRUE              = "true"
FALSE             = "false"
INTEGER           = {Digit}
REAL              = {Digit}"."{Digit}

ATPRE             = "@pre"
SELF              = "self"
DOT               = "."
ARROW             = "->"
IMPLIES           = "implies"

AND               = "and"
OR                = "or"
XOR               = "xor"
NOT               = "not"
```

```
NONEQUAL          =  "<>"
GREATEREQUAL      = ">="
LOWEREQUAL        = "<="
GREATER           = ">"
LOWER             = "<"
EQUAL             = "="

PLUS              = "+"
MINUS             = "-"
TIMES             = "*"
DIVIDE            = "/"

TWOCOLON          = "::"
COLON             = ":"
LPARAN            = "("
RPARAN            = ")"
COMMA             = ","

IDENTIFIER        = {Iden}


/* Precedences */

precedence left IMPLIES;
precedence left AND, OR, XOR;
precedence left EQUAL, NONEQUAL;
precedence left GREATER, LOWER, GREATEREQUAL, LOWEREQUAL;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left NOT, UMINUS;
precedence left DOT, ARROW;
precedence left ATPRE;


/* The grammar */
ocl_expr            ::=  ocl_expr context_part  |
                         context_part

context_part        ::=  CONTEXT context_decl_class inv_part   |
                         CONTEXT context_decl_class TWOCOLON
                                 context_decl_method method_params_paran:ps
                                 prepost_part:ocl

context_decl_class  ::=  IDENTIFIER  |
                         context_decl_class TWOCOLON IDENTIFIER

context_decl_method ::=  IDENTIFIER

method_params_paran ::=  LPARAN RPARAN  |
                         LPARAN form_method_params RPARAN

form_method_params  ::=  IDENTIFIER COLON IDENTIFIER  |
                         form_method_params COMMA IDENTIFIER COLON IDENTIFIER
```

```
prepost_part        ::=  pre_part   |
                         post_part  |
                         pre_part post_part

pre_part            ::=  PRE expr_part

post_part           ::=  POST expr_part

inv_part            ::=  INV expr_part

expr_part           ::=  base_type                  |
                         expr_part binary expr_part |
                         unary expr_part            |
                         IDENTIFIER DOT field       |
                         LPARAN expr_part RPARAN

base_type           ::=  TRUE     |
                         FALSE    |
                         INTEGER  |
                         REAL     |
                         IDENTIFIER

binary              ::=  PLUS          |
                         MINUS         |
                         TIMES         |
                         DIVIDE        |
                         AND           |
                         OR            |
                         EQUAL         |
                         NONEQUAL      |
                         GREATER       |
                         GREATEREQUAL  |
                         LOWER         |
                         LOWEREQUAL

unary               ::=  MINUS  |
                         NOT

field               ::=  IDENTIFIER  |
                         IDENTIFIER LPARAN act_params RPARAN

act_method_params   ::=  IDENTIFIER  |
                         act_method_params COMMA IDENTIFIER
```

Anhang C

Syntax der Algorithmenbeschreibung

Zur Notation der Syntax zur Beschreibung der Algorithmen wird die erweiterte Backus-Naur-Form verwendet. Zu beachten ist, daß die Nonterminalsymbole *Name* und *Typ*, in der EBNF jeweils kursiv geschrieben, wie Terminalsymbole behandelt werden. Bei ihnen handelt es sich um Zeichenketten, deren genaue Syntax die verwendete Programmiersprache festlegt. Auch die Anweisungen (*Stmnt*) und Prädikate (*Praed*), ebenfalls kursiv markiert, werden hier nicht definiert. Bei *Zahl* handelt es sich um eine natürliche Zahl.

```
Funktion    ::= FunKopf "==" FunRumpf

FunKopf     ::= Name [ "(" Parameter ")" ] ":" Rückgabe
Parameter   ::= Name ":" Typ [ "," Parameter ]
Rückgabe    ::= Typ [ "," Rückgabe ]

FunRumpf    ::= Expr [ ";" Expr ] | Expr [ "\n" Expr ]

Expr        ::= Stmnt | IfExpr | LoopExpr | AllExpr | SelExpr | LetExpr

IfExpr      ::= "if" Praed "then" Expr "else" Expr
LoopExpr    ::= ( "loop" Praed ":" Expr ) | ( "loop" Zahl ":" Expr )
AllExpr     ::= "all" Name "∈" Typ [ "|" Praed ] ":" Expr
SelExpr     ::= "sel" Name "∈" Typ "|" Praed ":" Expr
LetExpr     ::= "let" { Name ":=" Expr }+ "in" ":" Expr
```