Entwicklungsbegleitendes Testen mittels UML Sequenzdiagrammen

Vom Fachbereich Informatik der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades eines Doktor-Ingenieurs
(Dr.-Ing.) vorgelegt von

Dipl.-Wirtsch.-Inform.
Falk Fraikin

geb. am 29.9.1972 in Groß-Gerau

Referenten:

Prof. Dr. Wolfgang Henhapl

Prof. Dr. Mira Mezini

Datum der Einreichung: 10.11.2003

Datum der mündlichen Prüfung: 3.12.2003

Darmstädter Dissertation D 17

Danksagung

Diese Arbeit entstand in den Jahren 2000-2003 am Fachgebiet Praktische Informatik der Technischen Universität Darmstadt, in dem ich seit dem Abschluss meines Wirtschaftsinformatikstudiums im Februar 2000 (ebenfalls an der TU Darmstadt) tätig bin.

Mein Dank gilt allen, die mich bei der Erstellung dieser Arbeit unterstützt haben. Insbesondere gilt dies für meinen Betreuer Prof. Dr. Wolfgang Henhapl, der es in zahlreichen Diskussionen stets verstanden hat, mich mit neuen Blickwinkeln, Kritikpunkten aber auch wertvollen Denkanstößen zu konfrontieren, was wesentlich zum Gelingen der Arbeit beigetragen hat. Vielen Dank auch an Prof. Dr. Mira Mezini für die spontane Übernahme des Koreferats.

Ebenso möchte ich mich bei Prof. Dr. Andreas Spillner bedanken, der die Arbeit in der Endphase ihrer Entstehung mehrfach ausführlich in Augenschein genommen hat und mich mit seiner daraus resultierenden Kritik und seinen Verbesserungsvorschlägen meinem Ziel einen großen Schritt näher gebracht hat.

Außerdem möchte ich mich bei meinem Kollegen Thomas Leonhardt für die unzähligen fruchtbaren Gespräche bedanken sowie bei Prof. Dr. Thomas Kühne für die Hinweise gegen Ende der Arbeit.

Von sehr hohem Wert gerade auch in den frühen Phasen meiner Dissertation waren die Hinweise und Denkanstöße, die ich von den Mitgliedern des Arbeitskreises „Testen objektorientierter Programme“ der GI-Fachgruppe „Testen, Analyse und Verifikation“ bekommen habe. Neben dem bereits erwähnten Prof. Dr. Andreas Spillner, der diesem Arbeitskreis angehört, gilt mein Dank hier insbesondere auch Prof. Dr. Mario Winter.

Zusätzlich möchte ich mich bei den Studierenden der TU Darmstadt (Martin Girschick, Martin Knuth, Heiko Rossnagel, Felix Senn) bedanken, die durch ihre Diplomarbeiten und Hiwi-Tätigkeiten die Implementierung der Konzepte der Arbeit unterstützt haben.

Schließlich möchte ich noch bei meinem Vater bedanken, der diese Arbeit zum Anlass genommen hat, sich widererwartend doch mit der neuen deutschen Rechtschreibung auseinanderzusetzen und last but not least bei meiner Frau Sandra, ohne deren moralischen Unterstützung mir die Erstellung meiner Dissertation erheblich schwerer gefallen wäre.

Inhaltsverzeichnis

1 Einleitung 6

1.1 Problemstellung 8

1.2 Begriffsdefinitionen 10

1.3 Aufbau der Arbeit 11

2 Grundlagen 13

2.1 Testen objektorientierter Programme 13

2.1.1 Axiome der Adäquatheit von Testdaten 13

2.1.2 Probleme des Testens objektorientierter Programme 16

2.1.3 JUnit 18

2.2 Sequenzdiagramme 20

2.2.1 UML Sequenzdiagramme 20

2.2.2 Message Sequence Chart 21

2.3 Softwareentwicklungsprozesse 22

2.3.1 Rational Unified Process 22

2.3.2 Feature Driven Development 26

3 Testspezifikation durch UML Sequenzdiagramme 28

3.1 Sequenzdiagramme im Systementwicklungsprozess 28

3.2 Beispielanwendung Library 30

3.3 Testbare Sequenzdiagramme 31

3.4 Testfallspezifikation 33

3.4.1 Exceptions 35

3.4.2 Unscharfe Daten 35

3.5 Verfeinerung von Sequenzdiagrammen 36

3.6 Kombination von Sequenzdiagrammen 38

3.7 Plausibilitätsprüfung für Testfälle 40

3.8 Testresultat 41

3.9 Nutzen bei Unit- und Integrationstests 45

3.10 Vor- und Nachbedingungen 46

3.11 Testabdeckung 46

3.12 Probleme und Lösungsmöglichkeiten 49

4 SeDiTeC 53
4.1 Funktionalität 53
4.1.1 Testfallspezifikation 53
4.1.2 Testvorbereitung 54
4.1.3 Testausführung 54
4.1.4 Testauswertung 54
4.2 Implementierung 54
4.2.1 Kernkomponenten 55
4.2.2 Instrumentierung vs. Java Debug Interface 56
4.2.3 Anbindung an Together Control Center 57
4.2.4 SeDiTeCs Testfunktionalität innerhalb von Together 59
4.2.5 Teststubs 59
4.2.6 Instrumentierung des Sourcecodes 63
4.2.7 XML Datenexport 64
4.2.8 Testfallspezifikation mit SeDiTeC 65
4.2.9 Testausführung 67
4.2.10 Testauswertung 68
4.3 Initialisierungszustand 69
4.4 Vor- und Nachbedingungen für Methoden 70
4.5 Utility Klassen 71
4.5.1 Gleichheit von Objektstrukturen 72
4.5.2 Dateivergleich 74
4.6 Automatisierung mit Apache Ant 75
4.7 SeDiTeC und JUnit 76
4.8 Praxiserfahrung 78
4.9 Verwandte Arbeiten und existierende Testwerkzeuge 79

5 Testen von Design Pattern Implementierungen 82
5.1 Iterator 82
5.2 Observer 88
5.3 Visitor 91
5.4 Singleton 92
5.5 Bewertung 93

6 Zusammenfassung und Ausblick ... 96

Anhang ... 99

Anhang A – Methode `executeMethodCalls` ... 100

Anhang B – Instrumentierung des Sourcecodes ... 101

Literaturverzeichnis ... 103

Abbildungsverzeichnis

Abbildung 1: Kernklassen von JUnit 19
Abbildung 2: UML Sequenzdiagramm 21
Abbildung 3: Übersicht über die Bestandteile des Rational Unified Process 24
Abbildung 4: Einfacher Ereignisfluss für einen Use Case 25
Abbildung 5: Die fünf Prozesse des Feature Driven Development 26
Abbildung 6: Klassendiagramm des Library Systems 30
Abbildung 7: Einfaches testbares Sequenzdiagramm (logischer Testfall) 32
Abbildung 8: Einfaches testbares Sequenzdiagramm (konkreter Testfall) 33
Abbildung 9: Verfeinerung von Sequenzdiagrammen 37
Abbildung 10: Wiederverwendung eines Testfalls für eine Erbenklasse 50
Abbildung 11: Kernkomponenten von SeDiTeC 55
Abbildung 12: Einbindung der SeDiTeC Module in die Together GUI 58
Abbildung 13: Beispiel eines SeDiTeC Teststubs 60
Abbildung 14: Klassenhierarchie der Ausnahmebehandlung in Java 62
Abbildung 15: Rolle von Together beim Einsatz von SeDiTeC 64
Abbildung 16: Eingabe von Testfällen für einzelne Sequenzdiagramme 65
Abbildung 17: Eingabe von Testfällen für kombinierte Sequenzdiagramme 66
Abbildung 18: Grafische Aufbereitung von Testresultaten 69
Abbildung 19: Implementierung der Vergleichsfunktionalität in SeDiTeC 74
Abbildung 20: Iterator Pattern 83
Abbildung 21: Iterator Pattern – Testfall I 84
Abbildung 22: Iterator Pattern – Testfall II 85
Abbildung 23: Iterator Pattern – Testfall III 86
Abbildung 24: Iterator Pattern – Testfall IV 87
Abbildung 25: Iterator Pattern – Testfall V 88
Abbildung 26: Observer Pattern 89
Abbildung 27: Observer Pattern – Testfall I 89

Abbildung 28: Observer Pattern – Testfall II 90
Abbildung 29: Visitor Pattern..91
Abbildung 30: Visitor Pattern – Testfall I 92
Abbildung 31: Singleton Pattern..92
Abbildung 32: Singleton Pattern – Testfall I 93
Abbildung 33: Methode executeMethodCalls.......................................100
Abbildung 34: Methode Library.addBook 101
Abbildung 35: Methode Library.addBook (instrumentiert) 102

1 Einleitung

Das Thema Testen von objektorientierter Software erfreut sich seit einigen Jahren stetig wachsender Beachtung. Überkommen sind mittlerweile die anfänglichen Vorstellungen, dass sich das Testen objektorientierter Software nicht wesentlich vom Testen prozeduraler Software unterscheidet, bzw. die Objektorientierung das Testen von Software gar vereinfachen würde. Auch wenn diese Vorstellungen aus heutiger Sicht schwer nachvollziehbar erscheinen, so wurden sie doch Anfang der 90er Jahre von durchaus renommierten Autoren wie zum Beispiel Ivar Jacobson [Jacobson+92] und James Rumbaugh [Rumbaugh+91] vertreten.

Heutzutage gehört das Wissen um die speziellen Problematiken des objektorientierten Testens, die sich zum Beispiel durch Polymorphie, dynamisches Binden und Vererbung ergeben, sozusagen zum Allgemeinwissen eines Testers. Nichtsdestotrotz steht es außer Frage, dass die Testbemühungen in einem Großteil heutiger Software-entwicklungsprojekte nach wie vor Spielraum für Verbesserungen bieten.

Die Gründe hierfür sind vielfältig. Grundsätzlich lässt sich jedoch beobachten, dass die Bedeutung des Testens für den Erfolg der Softwareentwicklung von den Beteiligten häufig gar nicht wahrgenommen oder durch meist leider unbegründeten Optimismus heruntergespielt wird. Dies ist unter anderem auch ein psychologisches Problem. Softwareentwickler neigen dazu, den Fertigstellungsgrad eines Projektes anhand der prinzipiell implementierten Funktionalität zu bewerten, wobei „prinzipiell implementiert“ hier für „läuft für ein oder zwei manuelle Testeingaben ohne abzustürzen“ steht. Dieses prinzipielle Implementieren von Funktionalität und dann das Übergehen zum Implementieren der nächsten Funktionalität scheint für viele Entwickler interessanter oder befriedigender zu sein, als dafür Sorge zu tragen, dass die (scheinbar) bereits implementierte Funktionalität auch für die Spezialfälle geeignet ist, die der tägliche Einsatz später bringt. Ebenfalls scheint eine Rolle zu spielen, dass das Implementieren eher als kreative und das Testen eher als destruktive Tätigkeit empfunden wird.

Die eben erwähnten Spezialfälle, deren Existenz der durchschnittliche Entwickler für unmöglich hält, bevor er sie bzw. deren Auswirkungen nicht mit eigenen Augen gesehen hat, sind für den Endanwender jedoch tägliches Geschäft. Folglich bewerten Endanwender den Fertigstellungs-grad eines Projektes anhand der stabil lauffähigen Funktionalität. Es ist die Aufgabe des Testens, die durch die Diskrepanz dieser beiden Wahrnehmungen auftretenden Probleme zu lösen.

Interessant ist hierbei die Frage, wie es eigentlich dazu kommt, dass zumindest unerfahrene Entwickler dieses manchmal fast naiv anmutende Vertrauen in den von ihnen produzierten Code an den Tag legen. Die Gründe hierfür sind teilweise in der Ausbildung zu suchen. So gibt es beispielsweise an der TU Darmstadt keine einzige Vorlesung, die sich

speziell mit dem Testen von Software auseinandersetzt. Lediglich in der Vorlesung „Software Engineering“, deren Besuch aber nicht obligatorisch ist, wird das Thema in einigen Vorlesungsstunden kurz behandelt. Inzwischen gibt es immerhin Bestrebungen an der TU Darmstadt, das Testen auch in den Pflichtvorlesungen des Grundstudiums verstärkt einzubringen. Problematisch ist dabei jedoch, dass der Sinn des Testens um so schwieriger zu vermitteln ist, je kleiner und überschaubarer die zu entwickelnde Software ist. Naturgemäß sind die Projekte im Grundstudium aber alles andere als umfangreich. Etwas Abhilfe schafft die eben erwähnte Vorlesung „Software Engineering“, die ein einjähriges Praktikum beinhaltet, bei dem Studententeams reale Projekte aus der Industrie von der Pflichtenhefterstellung über Design und Implementierung bis zur Produktabnahme durchführen. Diese Projekte sind alleine vom Umfang her praktisch immer geeignet, die Einstellung der Teilnehmer zum Thema Testen nachhaltig zu ändern. Entschließt sich ein Student der TU Darmstadt jedoch gegen die Teilnahme an der Software Engineering Veranstaltung, so hat er eine gute Chance, dass er seinen Abschluss macht, ohne je von systematischem Testen und Prüfen von Software gehört zu haben.

Eine in der einschlägigen Literatur häufig anzutreffende Meinung besagt, dass Tester und Entwickler verschiedene Personen sein sollen [Myers79][1]. Dies hat mehrere Vorteile gegenüber der Möglichkeit, dass nur der Entwickler selbst seinen Code testet. Zum einen tendiert ein Entwickler verständlicherweise dazu, beim Testen seines Codes an die gleichen möglichen Eingaben zu denken, die ihm auch schon bei der Entwicklung als Grundlage dienten. Es ist daher wesentlich unwahrscheinlicher, dass der Code mit einer Eingabe konfrontiert wird, die nicht bei der Entwicklung bedacht wurde. Zum anderen erfordert das effektive Testen von Software einiges an theoretischem Wissen und auch Kenntnis von Testwerkzeugen. Eine Person, die sich nur mit Testen beschäftigt, kann sich daher wesentlich tiefer in diese Materie einarbeiten als eine Person, deren Hauptaufgabe in der Entwicklung von Software besteht.

In der Praxis ist die Trennung zwischen Entwicklern und Testern in verschiedenen Ausprägungen anzutreffen. Bei kleinen Projekten gibt es häufig gar keine reinen Tester, so dass jegliche Testbemühungen von den Entwicklern ausgehen. Je größer das Projekt wird, um so größer ist die Wahrscheinlichkeit, dass das von Entwicklern durchgeführte Testen zunehmend von dedizierten Testexperten übernommen wird (entweder innerhalb der Firma oder extern, wobei es inzwischen eine Reihe von Firmen gibt, die sich genau auf diese Dienstleistung spezialisiert haben). Dies gilt insbesondere für Integrations- und Systemtests und auch für

---

[1] Eine ganz andere Auffassung dazu vertritt Beizer, der den Hauptgrund für die Existenz von unabhängigen Testern im Schutz derselbigen sieht. In einer Organisation mit einer „erwachsenen Qualitätskultur“ sei dies im Allgemeinen unnötig [Beizer95].

Tests, die spezielle Werkzeuge erfordern, wie z.B. Performance- oder Lasttests. Unittests auf der anderen Seite bleiben eigentlich immer den Entwicklern selbst überlassen, da der Aufwand für den Tester, sich in einzelne Methoden einzuarbeiten und dann jeden offensichtlichen kleinen Fehler über ein mit meist erheblicher zeitlicher Verzögerung verbundenen Bugtrackingsystem weiterzuleiten, in keinem Verhältnis zum Nutzen stünde.

Problematisch ist hierbei jedoch die häufig mangelhafte Integration des Testprozesses in den Softwareentwicklungsprozess bzw. die unzureichende Flexibilität dieser Integration. So sieht beispielsweise der Rational Unified Process als Integrationsstrategie lediglich Bottom Up Testen vor [Jacobson+98], was nicht nur die Flexibilität beim Testen einschränkt, sondern auch die Implementierungsstrategie von vornherein festlegt.

Ebenfalls problematisch ist die unzureichende (Werkzeug-)Unterstützung gerade des Entwicklers beim dynamischen Testen. So erfordern die vom Entwickler durchgeführten Unittests meist das aufwendige manuelle Entwickeln von Testtreibern und Teststubs, was einem effektivem Unittest im Wege steht. Eine gewisse Linderung dieses Problems brachten Ende der 90er Jahre eine Reihe von Testframeworks, die so genannten XUnits, deren prominentester Vertreter JUnit sein dürfte. Diese Testframeworks unterstützen den Entwickler zwar nicht beim Erstellen von Testfällen und nehmen ihm auch nicht die Entwicklung von Testtreibern und Teststubs ab, sie stellen jedoch einen Coderahmen zur Verfügung, der das automatisierte Ausführen von beliebigen Mengen von Testfällen (in Testsuites gruppiert) inklusive sofortiger Rückmeldung über eventuell aufgetretene Fehler auf einfache Weise unterstützt. Dadurch wird der Aufwand für den testenden Entwickler ganz erheblich reduziert.

1.1 Problemstellung

Die vorliegende Arbeit zielt in eine ähnliche Richtung. Die den Anstoß gebende Fragestellung war, inwieweit UML Sequenzdiagramme, die im Rahmen von einer Reihe von verbreiteten Softwareentwicklungsprozessen wie zum Beispiel Rational Unified Process oder Feature Driven Development erstellt werden, aber auch beispielsweise bei der Beschreibung von Design Patterns und generell von Schnittstellen Verwendung finden, das dynamische Testen unterstützen können. Diese Fragestellung ist insofern nahe liegend, da UML Sequenzdiagramme sich mit ihren Objekten und Nachrichten intuitiv auf die Instanzen und Methodenaufrufe eines objektorientierten Programms abbilden lassen und damit einen logischen Testfall darstellen. Ergänzt man die Methodenaufrufe im Sequenzdiagramm um Aufrufparameter und Rückgabewerte, so erhält man einen konkreten Testfall für die Kommunikation zwischen den abgebildeten Instanzen der zu testenden Klassen.

Jedoch finden sich in der einschlägigen Literatur auch Hinweise, die darauf hindeuten, dass die Testspezifikation auf der Basis von Sequenzdiagrammen auch nicht ohne Schwierigkeiten ist. So bezeichnet

Robert Binder Sequenzdiagramme zwar als generell nützlich im Kontext von Design und Testspezifikation, nennt allerdings folgende Probleme [Binder00]:

- Die Darstellung komplexer Kontrollstrukturen ist schwierig.
- Dynamisches Binden kann nicht direkt dargestellt werden.
- Die Implementierung beispielsweise eines Use Cases kann normalerweise nicht in einem Sequenzdiagramm dargestellt werden, so dass der Designer gezwungen wird, mehrere Sequenzdiagramme zu erstellen, die u.U. redundante Informationen enthalten.

Die Untersuchung der Frage, inwieweit diese grundsätzlichen Schwierigkeiten relevant sind, welche weiteren Probleme es gibt und ob bzw. wie sie zu beseitigen sind, ist ein wesentlicher Bestandteil dieser Arbeit.

Will man Sequenzdiagramme als Grundlage für dynamische Tests einsetzen, bedarf es selbstverständlich eines Werkzeuges, das einerseits die Erstellung der Sequenzdiagramme und der zugehörigen Daten sowie andererseits die Ausführung und Auswertung der zugehörigen Tests unterstützt. In der Untersuchung, welche Funktionalität ein solches Werkzeug im Detail aufweisen muss, um sinnvoll eingesetzt werden zu können, sowie in der prototypischen Implementierung eines diese Funktionalität bietenden Werkzeugs namens SeDiTeC (SEquence DIagram TEst Center), liegt der Schwerpunkt dieser Arbeit.

Die in dieser Arbeit vorgestellten Konzepte lassen sich am besten für den Test der Funktionalität bei der Entwicklung von Geschäftsapplikationen anwenden, wie die Auswahl der betrachteten Entwicklungsprozesse bereits nahe legt. Beschäftigt man sich mit Softwareentwicklung für eingebettete Systeme, Softwareentwicklung mit Echtzeitanforderungen oder Softwareentwicklung mit hohen Sicherheitsanforderungen, sind Testmethoden, die auf Zustandsdiagrammen oder auf Message Sequence Charts aufbauen, unter Umständen geeigneter. Auch für die Verwendung im Kontext von Performanz- oder Lasttests sind die hier vorgestellten Konzepte nicht ausgelegt.

Als Programmiersprache wurde bei der Erstellung dieser Arbeit Java verwendet und zwar sowohl, was die zu prüfenden bzw. zu testenden Programme, als auch was die Implementierung des Testwerkzeugs angeht. Diese Einschränkung des Problembereichs auf eine Programmiersprache ermöglicht es, auf die konkreten Problemstellungen dieser Sprache detailliert einzugehen. Die allgemeinen Konzepte, die im Rahmen dieser Arbeit erstellt wurden, sollten jedoch für alle objektorientierten Programmiersprachen ihre Gültigkeit haben.

1.2 Begriffsdefinitionen

Im Rahmen dieser Arbeit wird unter **Testen von Software** in Anlehnung an [Binder00] ausschließlich das Ausführen von Programmcode unter Benutzung von Kombinationen von Eingabedaten verstanden.

Ein **logischer Testfall** beschreibt, welche Methoden in welcher Reihenfolge auf welchen Instanzen aufgerufen werden. Ein **konkreter Testfall** beschreibt zusätzlich, mit welchen Parametern und Rückgabewerten dies geschieht. Falls eine in einem konkreten Testfall vorkommende Methode eine Exception wirft, so wird auch dies im Testfall vermerkt. Die in einem konkreten Testfall relativ zu einem logischen Testfall zusätzlich enthaltenen Informationen (Parameter, Rückgabewerte, Exceptions) bilden einen **Testdatensatz.**

Unter einer **Testsuite** wird im Rahmen dieser Arbeit eine Menge von konkreten Testfällen verstanden, die gemeinsam zur Ausführung gebracht und ausgewertet werden (können). Die einzelnen Testfälle einer Testsuite laufen jedoch sequentiell ab. Testsuites werden dazu verwendet, Testfälle nach bestimmten Kriterien sinnvoll zu gruppieren (z.B. „alle Testfälle einer Komponente“), um die Verwaltung und Ausführung derselbigen effizienter zu gestalten. Der Begriff **Testsequenz** bezeichnet dagegen eine Aneinanderreihung mehrerer Testfälle, wobei die Nachbedingung eines Testfalls als Vorbedingung des jeweils folgenden Testfalls genutzt wird. Eine oder mehrere Testsequenzen können Teil einer Testsuite sein.

Konkrete Testfälle werden gegen eine **Implementation under Test (IUT)** zur Ausführung gebracht. Bei der IUT handelt es sich damit um den zu testenden Code (sprich: eine oder mehrere Klassen), auch **Testling** genannt.

Ein **Testtreiber** ist ein Werkzeug oder Programm, dass die einzelnen konkreten Testfälle gegen eine IUT zur Ausführung bringt. Bei einem **Teststub**[2] handelt es sich um einen Platzhalter für eine noch nicht implementierte Klasse oder Komponente. Teststubs verfügen üblicherweise über eine gegenüber der später angestrebten Implementierung eingeschränkte Funktionalität. Sie werden verwendet, um bereits implementierte Klassen, die noch nicht implementierte Klassen verwenden und damit von ihnen abhängig sind, trotzdem schon testen zu können.

Nach [Spillner+03] hat der Begriff **Fehler** zwei Bedeutungen. Zum einen bezeichnet er die Nichterfüllung einer Anforderung. Zum anderen ist er der Oberbegriff für die Begriffe Fehlerwirkung, Fehlerzustand und Fehlhandlung.

Unter **Fehlerwirkung** (engl. *failure*) versteht man das nach außen sichtbare Fehlverhalten der Software, wobei es sich beispielsweise um einen falschen Ausgabewert oder einen Programmabsturz handeln kann.

---

[2] Der deutsche Begriff für Teststub lautet „Teststumpf“, wird jedoch eher selten verwendet.

Davon zu unterscheiden ist der Begriff **Fehlerzustand** (engl. *fault*), der die Ursache für eine Fehlerwirkung darstellt. Hierfür kommen u.a. fehlende oder falsche Anweisungen im Programm in Frage.

Der Begriff **Fehlhandlung** (engl. *error*) bezeichnet nach [Spillner+03]:

1. Die menschliche Handlung (des Entwicklers), die zu einem Fehlerzustand in der Software führt.
2. Eine menschliche Handlung (des Anwenders), die ein unerwünschtes Ergebnis (im Sinne von Fehlerwirkung) zur Folge hat (Fehlbedienung).
3. Unwissentlich, versehentlich oder absichtlich ausgeführte Handlung oder Unterlassung, die unter gegebenen Umständen (Aufgabenstellung, Umfeld) dazu führt, dass eine geforderte Funktion eines Produkts beeinträchtigt ist.

Debugging bezeichnet den Vorgang, der initiiert wird, sobald ein Fehlerzustand in der IUT beobachtet wurde – sei es durch Testen oder bei der Benutzung des Systems. Dieser Vorgang umfasst das Auffinden (im Sinne von „Welche Stelle im Code ist verantwortlich“) und Beseitigen bzw. Korrigieren des Fehlerzustands. Debugging und Testen sind keinesfalls gleichzusetzen und beinhalten sich auch nicht gegenseitig[3].

In Abhängigkeit von der Basis, von der Testfälle mittels eines **Testverfahrens** abgeleitet werden, unterscheidet man zwischen **Black-Box-Tests** und **White-Box-Tests**. Während für die Ableitung der Testfälle für Black-Box-Tests die Spezifikation einer Anwendung herangezogen wird, dient hierfür bei den Testfällen für White-Box-Tests der Quellcode der Anwendung.

1.3 Aufbau der Arbeit

Kapitel 1 legt die für diese Arbeit zentrale Problemstellung sowie grundlegende Begriffsdefinitionen dar. In Kapitel 2 werden dann spezifische Probleme des Testens objektorientierter Programme sowie das Test-Framework JUnit vorgestellt. Außerdem enthält Kapitel 2 eine Übersicht über die beiden Softwareentwicklungsprozesse Rational Unified Process und Feature Driven Development.

Kapitel 3 beschäftigt sich mit der Testspezifikation auf der Basis von UML Sequenzdiagrammen. Hierbei wird der Begriff der testbaren Sequenzdiagramme eingeführt und erläutert, wie sich diese zum Testen von Software verwenden lassen. Hierzu wird auch betrachtet, wie sich testbare Sequenzdiagramme kombinieren und verfeinern lassen und wie Testresultate beim Testen auf der Basis von Sequenzdiagrammen effektiv dargestellt werden können.

---

[3] Anders beschreibt dies beispielsweise [Binder00] (S. 51), der das Testen, ob ein Debuggingvorgang den beobachteten Fehler tatsächlich beseitigt hat, dem Debugging zurechnet.

Kapitel 4 beschäftigt sich mit SeDiTeC, dem Prototypen eines Testwerkzeugs zum Testen auf der Basis von Sequenzdiagrammen, das im Rahmen dieser Dissertation entwickelt wurde. Dargestellt wird dabei, welche Anforderungen ein solches Werkzeug erfüllen muss und mit welchen Technologien sich diese (für die Programmiersprache Java) implementieren lassen. Insbesondere wird ein Teststubkonzept vorgestellt, dass die automatische Generierung voll funktionsfähiger Teststubs im Sinne der Testfallspezifikation mittels Sequenzdiagrammen ermöglicht.

Kapitel 5 befasst sich mit einer speziellen praktischen Anwendung von testbaren Sequenzdiagrammen. In diesem Kapitel werden eine Reihe von Design Patterns dahingehend untersucht, inwieweit sich für sie testbare Sequenzdiagramme angeben lassen, die als Grundlage eines effektiven Testprozesses von (beliebigen) Implementierungen dieser Design Patterns dienen können. Ebenfalls betrachtet wird dabei, welche Auswirkungen die Erstellung von testbaren Sequenzdiagrammen für die jeweiligen Design Patterns bzw. deren Spezifikation hat.

Kapitel 6 schließlich fasst die Ergebnisse dieser Dissertation zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

2 Grundlagen

In diesem Kapitel werden für die Arbeit grundlegende Konzepte erläutert. Hierzu zählen neben der speziellen Problematik des Testens objektorientierter Programme auch für die Entwicklung von Geschäftsapplikationen geeignete Softwareentwicklungsprozesse sowie Notationen für Sequenzdiagramme.

2.1 Testen objektorientierter Programme

Unabhängig von teilweise durchaus berechtigter Kritik ([Vessey+94], [Hatton98]) handelt es sich bei der Objektorientierung heute um eines der bedeutendsten Programmierparadigmen. Selbstverständlich geht dies einher mit einer intensiven Bearbeitung der theoretischen Grundlagen, die dieses Programmierparadigma betreffen wie z.B. die Softwareentwicklung aber auch die Qualitätssicherung und speziell das Testen. Und obwohl es auch gerade im Bereich Testen objektorientierter Programme[4] nicht an umfassenden Veröffentlichungen mangelt (z.B. [Binder00], [Sneed+02]), so lässt die in der Praxis anzutreffende Qualität des Testens objektorientierter Programme doch häufig zu wünschen übrig. Dies wird u.a. darauf zurückgeführt, dass viele herkömmliche, für imperative Programmiersprachen entwickelte Testverfahren und –werkzeuge sich für die Objektorientierung nicht oder nur eingeschränkt eignen und spezifische Werkzeuge für objektorientierte Programme noch rar sind [Liggesmeyer02].

Erste Erkenntnisse, dass objektorientierte Programme ihre ganz eigenen Fallstricke aufweisen, finden sich in dem viel zitierten Artikel von Perry und Kaiser [Perry+90]. Insbesondere wird in diesem Artikel darauf hingewiesen, dass geerbte Methoden in den Erbenklassen normalerweise erneut zu testen sind - unabhängig davon, ob sie in der Vaterklasse ausreichend getestet waren. Diese Erkenntnis lässt sich prinzipiell auch aus dem Anticomposition Axiom von Weyuker herleiten (s.u.) und macht deutlich, dass man sich im Rahmen des Testens objektorientierter Programme neue Gedanken über die Adäquatheit von Testdaten und Testsuites machen muss.

2.1.1 Axiome der Adäquatheit von Testdaten

Eines der größten Probleme beim Testen von Software besteht darin zu entscheiden, wann der Testprozess abgeschlossen ist. Naheliegenderweise sollten hierbei Faktoren wie Kritikalität der Software, Konsequenzen eines möglichen Fehlverhaltens sowie die Häufigkeit der Benutzung eine Rolle spielen. Im krassen Gegensatz dazu fand Myers als Testendekriterien in der Praxis [Myers79]:

---

[4] Auch wenn der Begriff „objektorientiertes Programm" häufig in diesem Kontext auf diese Art und Weise verwendet wird, ist er streng genommen hier irreführend. Treffender wäre eher „mit einer objektorientierten Programmiersprache entwickeltes Programm".

> *„The completion criteria typically used in practice are both meaningless and counterproductive. The two most common criteria are*
>
> *1) Stop when the scheduled time for testing expires.*
>
> *2) Stop when all the test cases execute without detecting errors."*

Während das erste Kriterium offensichtlich ungeeignet ist, liegt die Problematik des zweiten Kriteriums darin begründet, dass man es umso eher erfüllen kann, je *weniger* Testfälle man erstellt.

Selbstverständlich existierten bereits zu der Zeit, aus der dieses Zitat stammt (1979), in der Theorie erheblich sinnvollere Testendekriterien, die seitdem auch kontinuierlich erweitert und verfeinert wurden (eine kurze, prägnante Übersicht gibt beispielsweise [Whittaker00]). Allerdings wird auch an aktuelleren Studien deutlich, dass Theorie und Praxis bezüglich des Einsatzes von Testendekriterien weit voneinander entfernt sind. So ergab eine Studie der Universität Köln [Müller98], bei der 74 Software entwickelnde deutsche Unternehmen zu den Themen Management, Ausgestaltung, Methoden und Techniken sowie zum Werkzeugeinsatz beim Prüfen und Testen von Software befragt wurden, dass 30% der befragten Unternehmen überhaupt keine Testendekriterien verwenden. Dazu kommt noch, dass 40% der befragten Unternehmen angaben, dass das Testen beendet wird, sowie der Projektleiter dies anordnet, auch wenn die Testendekriterien zu diesem Zeitpunkt noch nicht erreicht sind.

Elaine Weyuker hat bereits 1986 acht Axiome für so genannte „Adäquatheitskriterien" (adequacy criteria) definiert [Weyuker86][5], wobei anhand dieser Kriterien entschieden werden soll, ob das Testen beendet werden kann. Diese Axiome[6] werden verwendet, um die Qualität von Adäquatheitskriterien zu beurteilen. Die ersten vier dieser Axiome sind intuitiv einsichtig und unabhängig vom verwendeten Programmier-paradigma und der verwendeten Programmiersprache und treffen sowohl für Black-Box (basierend auf der Spezifikation) Testen als auch für White-Box (basierend auf der Implementierung) Testen zu:

> **Applicability:** Für jedes Programm existiert ein adäquates Testset[7].
>
> **Nonexhaustive Applicability:** Zu einem gegebenen Programm P existiert ein Testset T, so dass P adäquat durch T getestet wird, T jedoch nicht erschöpfend ist.

---

[5] Weyuker hat diesen acht Axiomen später drei weitere hinzugefügt [Weyuker88], die in diesem Kontext jedoch keine Rolle spielen.

[6] Weyuker verwendet tatsächlich den Begriff „Axiom", obwohl es sich hier vielmehr um „Anforderungen" an Testendekriterien handelt. Weyuker wurde hierfür verschiedentlich kritisiert (s. zum Beispiel [DSEWiki03]).

[7] Ein Testset ist eine Menge von Testfällen.

Monotonicity: Wenn T adäquat für P ist, und T eine Teilemenge von T' ist, so ist T' adäquat für P.

Inadequate Empty Set: Die leere Menge stellt für kein Programm ein adäquates Testset dar.

Die weiteren Axiome sind nicht mehr ganz so offensichtlich, haben dafür aber interessantere Auswirkungen auf die Auswahl von Testfällen:

Antiextensionality: Es gibt Programme P und Q, so dass $P \equiv Q$[8] und ein Testset T, das für P adäquat ist, aber nicht für Q.

General Multiple Change: Es gibt Programme P und Q, die die gleiche Form (im syntaktischen Sinne, nicht im semantischen Sinne wie bei der Antiextensionality) haben, und ein Testset T, das für P adäquat ist, aber nicht für Q.

Anticomposition: Es gibt Programme P und Q, so dass ein Testset T für P adäquat ist, P(T) für Q adäquat ist, aber T nicht adäquat für die Komposition von P und Q ist.

Antidecomposition: Es gibt eine Komponente Q, die Teil eines Programms P ist, und ein für P adäquates Testset T, T' sei die Menge der Vektoren der Werte, die Variablen beim Eintreten in Q für ein t aus T annehmen können, und T' ist nicht adäquat für Q.

Aus dem Antiextensionality Axiom lässt sich ableiten, dass die Gewinnung von Testfällen nicht ausschließlich aus Black-Box Sicht heraus geschehen kann, da dieselbe zu implementierende Funktionalität abhängig von der tatsächlichen Implementierung unterschiedliche Testfälle für ein adäquates Testen benötigt.

Das General Multiple Change Axiom sagt dagegen aus, dass sich Testfälle auch nicht alleine aus White-Box Sicht ableiten lassen, da die Semantik einer Implementierung eine Rolle bei der Beurteilung der Adäquatheit eines Testsets spielt.

Das Anticomposition Axiom schließlich besagt, dass das adäquate Testen aller einzelnen Komponenten eines Systems nicht unbedingt ausreicht, um das Gesamtsystem adäquat zu testen, das sich aus diesen Komponenten zusammensetzt. Dies bedeutet u.a., dass ein adäquates Testen nicht nach dem Unittest beendet sein kann, sondern immer eine Form des Integrations- bzw. Systemtests erfordert.

Das Antidecomposition Axiom sagt aus, dass die Tests für eine Komponente unter Umständen angepasst und erneut durchgeführt werden müssen, wenn die Komponente in einer neuen Umgebung eingesetzt wird. Ein katastrophales Beispiel für nicht qualitätsgesicherte Wiederverwendung einer Komponente ist der Absturz der Ariane 5 Rakete 40 Sekunden nach dem Start am 4. Juni 1996. Die fehlerhafte Komponente

---

[8] P ist funktional äquivalent zu Q

(zuständig für die Lageregelung) war unverändert aus der Ariane 4 übernommen worden, gegen dessen Spezifikation sie auch getestet war und in der sie über 10 Jahre ohne Probleme ihren Dienst verrichtet hatte.

Die von der Komponente durchzuführenden Berechnungen für die Ariane 5 waren zwar selbstverständlich nicht prinzipiell anderer Natur als für die Ariane 4, jedoch konnten größere Zahlen in den Berechnungen vorkommen. Dies führte letztlich zu einem unbehandelten Überlauf einer Variablen, der das System zum Absturz brachte. Das Backupsystem (redundante Hardware), das im Falle eines Ausfalls des Primärsystems einspringen sollte, lief mit der gleichen Software und war bereits einige Sekunden vorher aus dem gleichen Grund abgestürzt, so dass die Rakete ihre geplante Flugbahn verließ und die Selbstzerstörung ausgelöst wurde.

Dieses Versagen ist dabei interessanterweise nicht unglücklich ausgewählten Testfällen zuzuschreiben, die die Möglichkeit eines unbehandelten Überlaufs einfach nicht aufgedeckt haben, sondern der Tatsache, dass das Projektmanagement die Wiederverwendung der Komponente *ohne* erneutes Testen beschloss. An diesem Beispiel wird sehr deutlich, welchen Hintergrund das Antidecomposition Axiom hat. Ein und die selbe Komponente wird üblicherweise in verschiedenen Programmen (zumindest leicht) unterschiedlich verwendet, was zu unterschiedlichen Anforderungen führt und damit konsequenterweise eigentlich zu einer neuen Spezifikation führen müsste. Würde man in solchen Fällen jedoch eine neue Spezifikation erstellen, so läge es auf der Hand, dass auch neue Tests durchgeführt werden müssten. In gewissem Sinne handelt es sich damit um ein Spezifikationsproblem (vgl. auch [Briand+01]).

Die sich durch die genannten Axiome ergebenden Implikationen für das Testen auf der Basis von Sequenzdiagrammen werden im Abschnitt 3.11 betrachtet.

2.1.2 Probleme des Testens objektorientierter Programme

Der Hauptgrund für die Unzulänglichkeit herkömmlicher (für imperative Programme verwendeter) Testmethoden bezüglich der Prüfung objektorientierter Software liegt darin, dass diese Testmethoden meist darauf abzielen, Fehlerzustände innerhalb *einer* Funktion bzw. Methode eines Programms zu finden. Insbesondere die strukturorientierten Abdeckungsmaße wie die Zweigüberdeckung [Liggesmeyer02] untersuchen üblicherweise einzelne Methoden.

In der prozeduralen Programmierung, in der Prozeduren bzw. Funktionen normalerweise[9] statisch gebunden sind, ist dies auch unproblematisch, da durch die Ausführung aller Zweige innerhalb aller Methoden auch garantiert wird, dass alle Zweige „zwischen" den Methoden ausgeführt werden. Probleme treten erst auf, wenn Methoden – wie in objekt-

[9] Eine Ausnahme sind beispielsweise die Function Pointer in C.

orientierten Sprachen üblich – dynamisch gebunden werden, da dann eine einen Methodenaufruf darstellende Anweisung nicht nur eine sondern mehrere Methoden aufrufen kann. Die herkömmliche Zweigabdeckung garantiert dann jedoch nur die Ausführung einer dieser Methoden und ist damit nicht adäquat für solche Fälle, da die einzelnen Methoden meist recht einfach sind und sich ein Großteil der Komplexität objektorientierter Software im Zusammenspiel der auf einzelnen Instanzen aufgerufenen Methoden verbirgt ([Binder00], [Winter98]).

Der Begriff Kapselung bezeichnet die für objektorientierte Sprachen typischen Zugriffskontrollmechanismen, die die Sichtbarkeit von Klassen, Methoden und Attributen festlegen. Durch Verwendung solcher Mechanismen lässt sich einerseits eine höhere Modularität erreichen und andererseits das Geheimnisprinzip („information hiding“ [Parnas71] umsetzen.

Obwohl dieses Prinzip selbstverständlich der Fehlervermeidung dient, wie sie zum Beispiel durch globale Variablen entstehen können, kann es jedoch das Testen erschweren, da es – je nach verwendeter Sprache – schwierig bis unmöglich sein kann, den Zustand von Objekten zu setzen und auszulesen. Insbesondere das Setzen von Zuständen ist aber für die meisten Testtechniken von großer Wichtigkeit, um überhaupt eine Ausgangsposition für einen Test herstellen zu können [Binder94]. Aber auch das Auslesen von Attributbelegungen ist unter Umständen essentiell für die Bewertung eines Testlaufs, wobei verschiedene Lösungen für dieses Problem existieren. So ist es zum Beispiel denkbar, hierfür spezielle Methoden in die zu testenden Klassen einzubauen oder sozusagen „gewaltsam“ die Kapselungsmechanismen der Sprache außer Kraft zu setzen (ein Beispiel hierfür ist die Verwendung der Java Reflection API). Diesen beiden Methoden vorzuziehen ist es jedoch, sich mit den tatsächlich vorhandenen öffentlichen Schnittstellen der zu testenden Klassen zu begnügen und sie somit weder mit „unnötigem“ Code zu belasten noch ihre Kapselung zu verletzen [McGregor+94].

Abstrakte und generische Klassen stellen ein besonderes Problem dar, da sie sich dadurch auszeichnen, dass sie sich nicht instanziieren lassen und üblicherweise auch nicht vollständig implementiert sind (während bei abstrakten Klassen meist eine oder mehrere Methoden lediglich deklariert aber noch nicht implementiert sind, fehlt bei generischen Klassen der Typparameter).

Im Falle einer abstrakten Klasse muss daher erst einmal eine davon abgeleitete Klasse implementiert werden, um die abstrakte Klasse testen zu können. Am schwierigsten zu testen sind hierbei die konkreten Methoden der abstrakten Klasse, die abstrakte Methoden verwenden, da die zugehörigen Testfälle nur jeweils im Kontext einer implementierenden Klasse sinnvoll erstellt werden können, dann aber wiederum spezifisch für die implementierende Klasse sind.

Bei den generischen Methoden liegt das Hauptproblem in der möglichen (semantischen, nicht syntaktischen) Unverträglichkeit zwischen dem Code der generischen Klasse und einem tatsächlich angegebenen Typparameter.[10] Firesmith geht sogar soweit zu sagen, dass eine generische Klasse aus diesem Grund niemals als vollständig ausgetestet angesehen werden kann, da man ja nicht vorhersehen kann, ob es mit bestimmten bisher nicht bedachten Typparametern nicht doch zu Problemen kommt [Firesmith93]. Allerdings ist diese Argumentation insoweit unbefriedigend (bzw. nicht sonderlich aussagekräftig), als man sie analog z.B. für eine beliebige Klasse führen könnte, von der andere Klassen erben, wobei die Erbenklassen manche Methoden der Vaterklasse redefinieren. Auch hier könnte man sagen, dass die potentielle Vaterklasse nicht als ausreichend getestet angesehen werden kann, da unvorhersehbar ist, welche Methoden wie redefiniert werden und ob diese noch korrekt mit den nicht redefinierten Methoden interagieren.

Laufzeitumgebungen von objektorientierten Programmiersprachen stellen üblicherweise spezielle Dienste zur Verfügung, die u.U. getestet werden müssen. Hierzu zählt beispielsweise die Garbage Collection. Abgesehen davon wird von Compilern von objektorientierten Programmiersprachen auch – je nach Sprache – bestimmter Code generiert, ohne dass der Programmierer hierfür aktiv werden müsste, im Gegenteil: Viele dieser generierten Programmstücke lassen sich gar nicht unterdrücken. Trotzdem kann es notwendig sein, auch diesen generierten Code zu testen. Beispiele hierfür sind Default-Konstruktoren und -Destruktoren oder die verschiedenen „equal“ Methoden von Eiffel [Meyer92].

2.1.3 JUnit

Im Bereich des Klassentests gewann in den letzten Jahren das von Erich Gamma und Kent Beck 1998 veröffentlichte Testrahmenwerk JUnit (vgl. [Gamma+98], [Gamma+99]) zunehmend an Bedeutung. Die grundlegende Motivation war, dass einer der Hauptgründe für mangelnde Testaktivitäten auf Klassenebene in typischen Softwareentwicklungen das Fehlen einer mit geringem Aufwand verbundenen Möglichkeit für solches Testen ist. Mit geringem Aufwand ist dabei sowohl eine geringe Einarbeitungszeit gemeint, als auch die Möglichkeit, einen Test in kurzer Zeit aufzusetzen (in diesem Fall zu programmieren) und auch auszuführen.

Da es sich bei JUnit um ein Testrahmenwerk für die Programmiersprache Java[11] handelt, besteht es im Wesentlichen nur aus Java Klassen (vgl. Abbildung 1), die vom Entwickler verwendet oder beerbt werden, um

---

[10] Wobei die in dieser Arbeit näher betrachtete Programmiersprache Java in der aktuellen Version generische Klassen nicht unterstützt. Dies soll sich mit der nächsten Version jedoch ändern [Sun03a].

[11] Mittlerweile gibt es eine Unmenge von so genannten „XUnits“, sprich: Implementierungen der JUnit Funktionalität für die meisten derzeit gebräuchlichen Programmiersprachen.

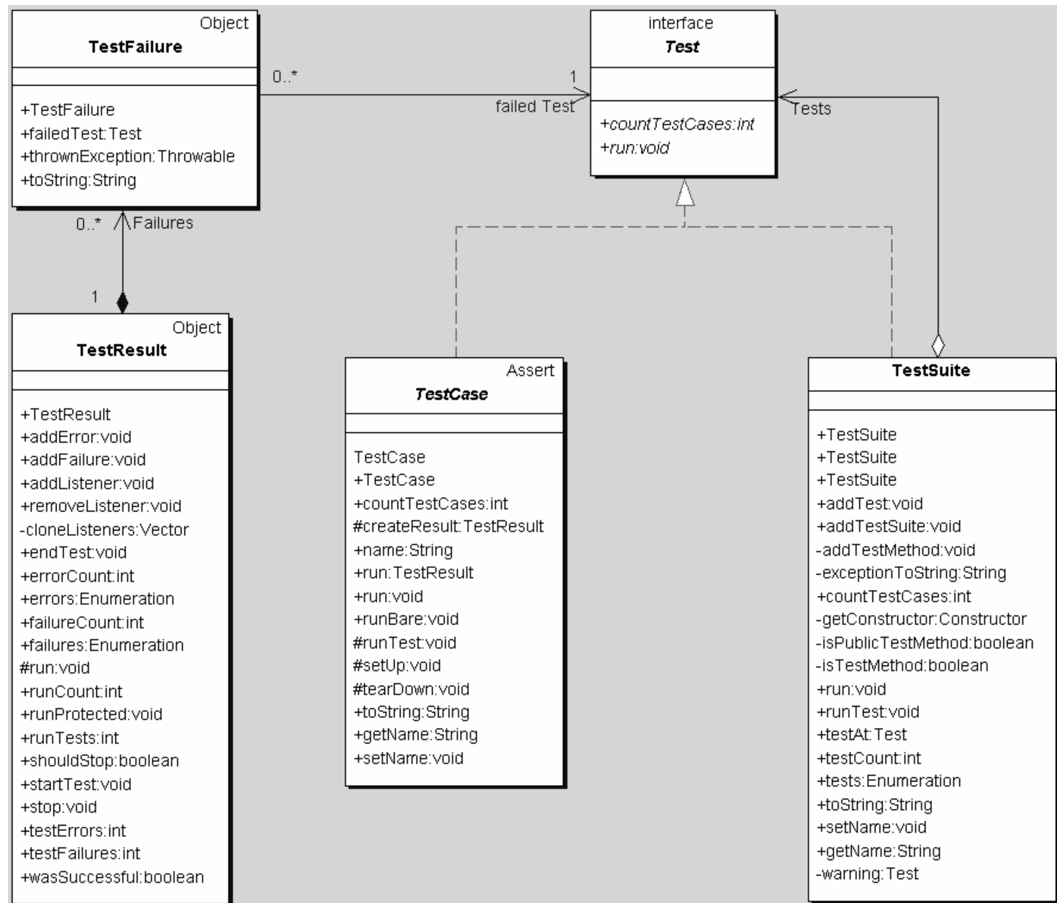

Abbildung 1: Kernklassen von JUnit

Tests zu programmieren. Die Testspezifikation erfolgt damit durch „normalen" Java Code, der die zu testende Funktionalität verwendet.

Für einen Entwickler, der mit Hilfe von JUnit Tests schreiben möchte, sind die beiden Klassen `TestCase` und `TestSuite` von zentraler Bedeutung.[12] Testfälle werden hierbei in Methoden definiert, die sich in Klassen befinden, die von `TestCase` erben. Jede dieser Testmethoden entspricht einem Testfall und dient dazu, eine bestimmte Funktionalität (meist eine Methode in einer Klasse) zu testen. Hierbei kann man auf eine Reihe von statischen Methoden der Klasse `Assert` zurückgreifen, die einem das Vergleichen von Basistypen, Objekten oder Abfragen auf `null` ermöglichen. Schlägt einer dieser Vergleiche fehl, so wird diese Information aufgezeichnet und dem Tester am Ende des Tests zur Verfügung gestellt (ein Testlauf wird nicht nach einem fehlerhaften Testfall abgebrochen).

Bevor ein Testfall ablaufen kann, müssen häufig bestimmte Initialisierungen vorgenommen werden (z.B. Erzeugen von Objekten, Herstellen von Datenbankverbindungen). Hierfür gibt es in der Klasse `TestCase` die

---

[12] Die hier angestellten Betrachtungen beziehen sich auf die Version 3.7 von JUnit.

Methode `setUp`, die direkt vor jeder Testmethode aufgerufen wird und gegebenenfalls in den Erbenklassen zu redefinieren ist. Das Gegenstück zu `setUp` ist die Methode `tearDown`, die direkt nach jeder Testmethode aufgerufen wird (z.B. um den Inhalt einer Datenbank zurückzusetzen).

Die Klasse `TestSuite` wird verwendet, um Testfälle (Testmethoden) zu gruppieren, d.h. beim Ausführen von Tests mit JUnit wird eine (oder mehrere) so genannte Testsuite zur Ausführung gebracht. Einem `TestSuite` Objekt können sowohl einzelne Testfälle als auch weitere Testsuites hinzugefügt werden, so dass es sich bei `TestCase`, `TestSuite` und dem von beiden Klassen implementierten Interface `Test` um eine Anwendung des Composite Pattern (vgl. [Gamma+95]) handelt.

Die eigentliche Testausführung kann entweder unter einer graphischen Oberfläche oder von der Kommandozeile aus vorgenommen werden. In beiden Fällen wird man über den Fortschritt der Testausführung auf dem Laufenden gehalten und kann erkennen, ob bereits einer der bisher ausgeführten Testfälle fehlgeschlagen ist.

Erich Gamma und Kent Beck betonen in [Gamma+98], dass die Implementierung der Testmethoden *vor* der Implementierung der zu testenden Methoden stattfinden sollte. Damit soll zum einen garantiert werden, dass der Test überhaupt geschrieben wird und zum anderen, dass der Entwickler sich im Voraus mehr Gedanken darüber macht, wie die Methode später verwendet werden könnte. Außerdem wird durch diese Herangehensweise ein gewisses Maß an Testbarkeit erzwungen.[13]

2.2 Sequenzdiagramme

Dieser Abschnitt dient dazu, zwei Typen von Sequenzdiagrammen vorzustellen, die im Rahmen der Softwareentwicklung Verwendung finden: UML Sequenzdiagramme und Message Sequence Charts.

2.2.1 UML Sequenzdiagramme

Das UML (Unified Modeling Language) Sequenzdiagramm ist einer der beiden Typen von Interaktionsdiagrammen, die die UML [OMG03] beinhaltet. UML Interaktionsdiagramme dienen insbesondere dazu, eine Menge von Objekten sowie die zwischen diesen Objekten ausgetauschten Nachrichten darzustellen. Im Gegensatz zum anderen Interaktionsdiagrammtyp der UML, dem Kollaborationsdiagramm, liegt beim Sequenzdiagramm die Betonung auf der zeitlichen Abfolge dieser Nachrichten. Beide Interaktionsdiagrammtypen werden hauptsächlich bei der detaillierteren Beschreibung von Use Cases verwendet (Use Case Realisierungen).

---

[13] Die prinzipielle Idee, Tests vor der eigentlichen Implementierung zu erstellen, war allerdings bereits zum Zeitpunkt der oben erwähnten Veröffentlichung von Beck und Gamma (1998) wohl bekannt, wenn auch in der Praxis wenig verbreitet (s. zum Beispiel [Gelperin+88]).

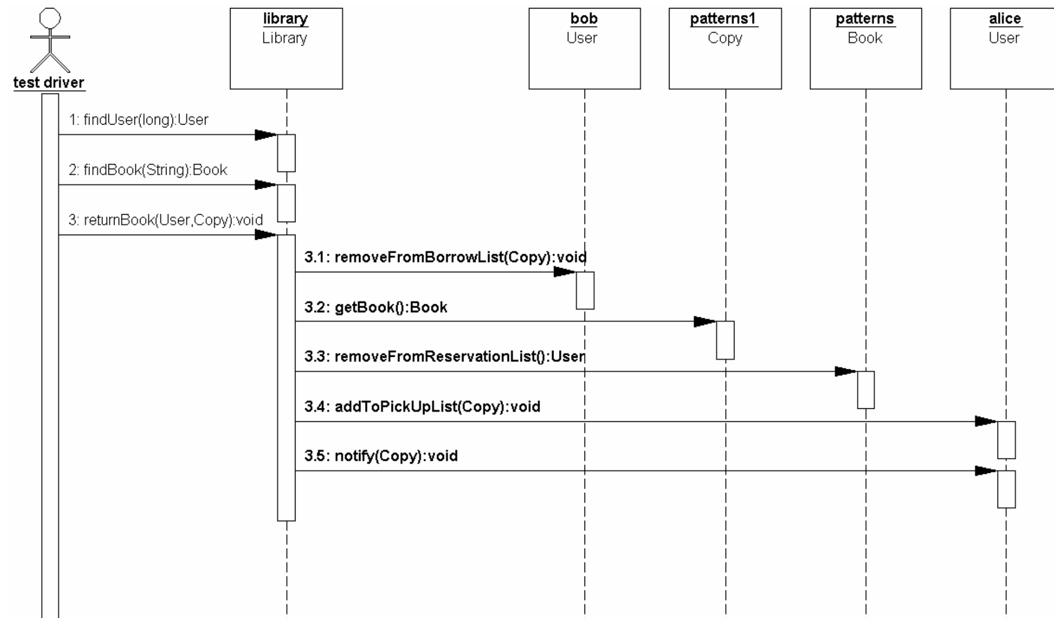

Abbildung 2: UML Sequenzdiagramm

Sequenzdiagramme sind zweidimensionale Diagramme (s. Abbildung 2), wobei sich auf der horizontalen Achse die an der modellierten Interaktion beteiligten Objekte befinden und die vertikale Achse die Nachrichten in zeitlicher Reihenfolge (von oben nach unten) darstellt. Die Objekte werden durch Rechtecke am oberen Rand des Diagramms identifiziert, welche den Namen des Objektes gefolgt von der Klasse des Objekts beinhalten. Unterhalb eines jeden Objektes befindet sich seine Lebenslinie, die während der Lebensdauer des Objektes gestrichelt ist und sich zu einer Doppellinie erweitert, wenn das Objekt aktiv ist (d.h. wenn ein Methodenaufruf auf dem Objekt abgearbeitet wird).

Nachrichten bzw. Methodenaufrufe werden durch Pfeile dargestellt, die ihren Ursprung in der Lebenslinie des Objektes haben, das den Methodenaufruf initiiert, und an der Lebenslinie des Objektes enden, auf dem der Methodenaufruf ausgeführt wird. Ursprung- und Zielobjekt können identisch sein. Die Beschriftung des Pfeils besteht aus einer Sequenznummer und der Signatur des beschriebenen Methodenaufrufs.

Bei den bisher beschriebenen Bestandteilen und Eigenschaften von UML Sequenzdiagrammen handelt es sich um die am häufigsten von CASE Werkzeugen unterstützten und in Projekten verwendeten. Darüber hinausgehend existieren beispielsweise Möglichkeiten zur Modellierung von Verzweigungen, Schleifen, asynchronen Nachrichten etc. Diese werden jedoch erheblich seltener verwendet und von CASE Werkzeugen uneinheitlich oder gar nicht unterstützt.

2.2.2 Message Sequence Chart

Bei Message Sequence Chart (MSC) handelt es sich um eine von der International Telecommunication Union definierte Empfehlung („Recommendation“) für einen Standard (s. [ITU99]). Diese Empfehlung ist

in der Z-Serie von Empfehlungen der International Telecommunication Union enthalten, die Sprachen und allgemeine Softwareaspekte von Telekommunikationssystemen behandelt. Die Abkürzung MSC bezeichnet sowohl die MSC Sprache als auch ein MSC Diagramm.

Die grundlegenden Bestandteile von MSCs entsprechen im Wesentlichen den oben genannten Bestandteilen von UML Sequenzdiagrammen, so dass Abbildung 2 mit nur unwesentlichen Änderungen auch ein MSC darstellen könnte (z.B. wären die Lebenslinien nicht gestrichelt sondern durchgezogen, die Typangaben der Objekte befänden sich nicht in sondern oberhalb der Rechtecke etc.).

Im Vergleich zur Notation der UML Sequenzdiagramme ist MSC jedoch erheblich komplexer. So unterstützt MSC z.B. die Verwendung von Timern und durch so genannte High-Level MSCs (HMSCs) eine Hierarchisierung von MSCs.

Insgesamt merkt man MSC die Herkunft aus der Telekommunikationsbranche deutlich an. Asynchrone Nachrichten, Echtzeitanforderungen, Prozesse sind Stichwörter, die im Zusammenhang mit MSC häufig fallen. Die Unterstützung für objektorientierte Konzepte dagegen ist ein erst recht junges Thema für MSC, die speziell in diesem Kontext vorhandene Werkzeugunterstützung ist entsprechend spärlich. Aus diesem Grund und wegen der erheblichen größeren Bedeutung der UML im für diese Arbeit ins Auge gefassten Anwendungsentwicklungsbereich wird im weiteren Verlauf dieser Arbeit MSC nicht weiter betrachtet.

2.3 Softwareentwicklungsprozesse

Um einen Überblick darüber zu geben, in welcher Form und zu welchem Zweck UML Sequenzdiagramme in verbreiteten Softwareentwicklungsprozessen Verwendung finden, werden in diesem Abschnitt der Rational Unified Process sowie Feature Driven Development diesbezüglich näher betrachtet.

Hierbei ist anzumerken, dass auch eine Reihe der populären leichtgewichtigen Prozesse[14] (zu denen sich auch das Feature Driven Development zählt) wie zum Beispiel Extreme Programming [Beck00] keineswegs grundsätzlich auf Sequenzdiagramme verzichten [Fowler01]. In den beiden hier betrachteten Softwareentwicklungsprozessen ist die den Sequenzdiagrammen zugedachte Rolle jedoch überdurchschnittlich klar definiert.

2.3.1 Rational Unified Process

Der Rational Unified Process (RUP) hat seine Wurzeln in dem 1987 zuerst veröffentlichten Objectory Process, der dann wiederum 1997 zum Rational Objectory Process wurde [Jacobson+98]. Der RUP selbst wurde (unter

[14] Inzwischen werden diese auch unter dem Begriff „Agile Software Development“ zusammengefasst [Fowler+01].

diesem Namen) zum ersten Mal 1998 veröffentlicht und zwar in der Version 5.0.

Der RUP unterteilt sich auf der fachlichen Ebene in so genannte Disziplinen und auf der zeitlichen Ebene in Phasen (s. Abbildung 3). Diese Darstellung ist eine von den Autoren des RUP bewusst gewählte Abstraktion nicht nur der Realität sondern auch der eigentlichen Vorgänge innerhalb des RUPs, um die Verständlichkeit des Prozesses zu erhöhen. In der Realität wird diese Ansicht dem tatsächlichen Projekt angepasst, so dass zum Beispiel Aufgaben, die eigentlich der Elaboration-Phase zugeordnet werden, durchaus parallel zu Aufgaben der Inception-Phase ablaufen können. Ebenso besteht keine grundsätzliche strenge zeitliche Abfolge zwischen den einzelnen Disziplinen, die daher größtenteils parallel ausgeführt werden.[15]

Der RUP wird als Use Case getrieben bezeichnet, d.h. Use Cases spielen eine zentrale Rolle bei der Anforderungsanalyse (Requirements Disziplin) und dienen als maßgebliche Grundlage für andere Hauptdisziplinen wie zum Beispiel Analysis & Design aber auch Test. Die in der Anforderungsanalyse entstandenen Use Cases[16] werden dann in der Analysis & Design Disziplin mittels so genannter Use Case Realisierungen verfeinert. Zentraler Bestandteil einer Use Case Realisierung sind immer ein oder mehrere Interaktionsdiagramme, d.h. Sequenz- und / oder Kollaborationsdiagramme, anhand derer analysiert wird, welche Objekte für die Implementierung der Funktionalität eines Use Cases benötigt werden, welche Methoden diese Objekte anbieten müssen und auf welche Art und Weise sie interagieren. Der Großteil dieser Arbeit liegt in der zweiten, der Elaboration-Phase. Ebenfalls in der Elaboration-Phase und damit hierzu parallel soll nach dem RUP die Testspezifikation erfolgen. Eine detaillierte Beschreibung des Entstehungsprozesses von Sequenzdiagrammen als Use Case Realisierungen gibt [Rosenberg+01]. Eine formalere Vorgehensweise, um Use Cases konsistent zu verfeinern, wird in [Winter99] vorgestellt.

[15] Für konkrete Elemente bzw. Artefakte und deren Bearbeitung in einzelnen Disziplinen kann selbstverständlich eine zeitliche Abhängigkeit bestehen – eine Klasse wird beispielsweise nicht implementiert werden, bevor sie designed wurde.

[16] Es gibt zahlreiche Veröffentlichungen zum Thema Erstellen von Use Cases, das im Rahmen dieser Arbeit nicht weiter behandelt wird. Eine ausführliche und auch recht eng an den RUP angelehnte Abhandlung findet sich beispielsweise in [Rosenberg+99].

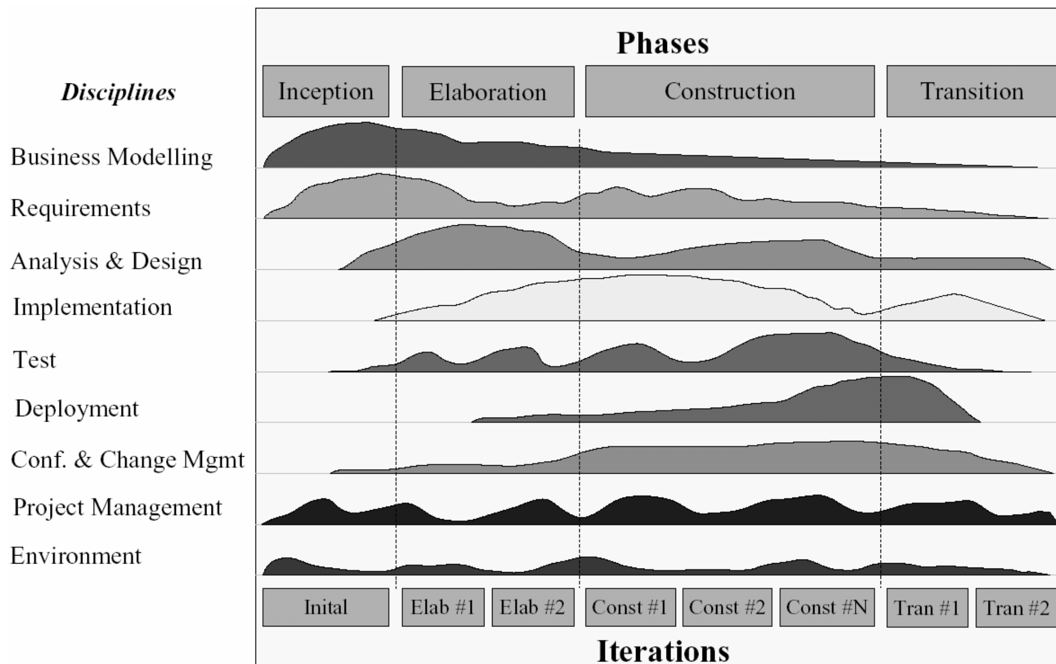

Abbildung 3: Übersicht über die Bestandteile des Rational Unified Process

Im Vergleich zur Analyse und Design Disziplin ist die Test Disziplin des Unified Process eher einfach strukturiert. Auch befinden sich die bezüglich der Test Disziplin gegebenen Richtlinien und Anweisungen auf recht abstraktem Niveau.[17] Im Kontext dieser Arbeit interessant ist die generelle Richtlinie, dass aus Use Cases Systemtests bzw. Black Box Tests und aus Use Case Realisierungen Integrationstests bzw. White Box Tests abgeleitet werden [Jacobson+98]. Wie dies jedoch genau geschehen soll bleibt unklar.

Etwas konkreter ist an dieser Stelle die aktuelle Version des RUP[18]. Dort wird vorgeschlagen, dass die Testfälle aus Use Case Szenarien abgeleitet werden sollen, die sich aus den verschiedenen Pfaden durch den Basis- und die eventuell vorhandenen Alternativflüsse des Use Cases ergeben. Dies soll an einem Beispiel verdeutlicht werden.

Abbildung 4 zeigt einen einfachen Ereignisfluss eines Use Cases. Der durchgezogene senkrechte Pfeil repräsentiert dabei den Basisfluss, die gestrichelten Pfeile repräsentieren Alternativflüsse. Während der Basisfluss für sich alleine genommenen den einfachsten Pfad durch diesen Ereignisfluss darstellt, lassen sich unter Einbeziehung der Alternativflüsse auch kompliziertere Pfade bilden. Wie aus der Abbildung ersichtlich ist, können Alternativflüsse sowohl vorwärts (Alternativfluss 1 und 3) als auch rückwärts (Alternativfluss 2) gerichtet sein. Außerdem

---

[17] Je nach Standpunkt des Lesers gilt das sicherlich für alle im RUP beschriebenen Disziplinen. Bei den Disziplinen Test und Implementation ist es jedoch besonders ausgeprägt.

[18] Derzeit ist dies die Version 2001A.04.00.13.

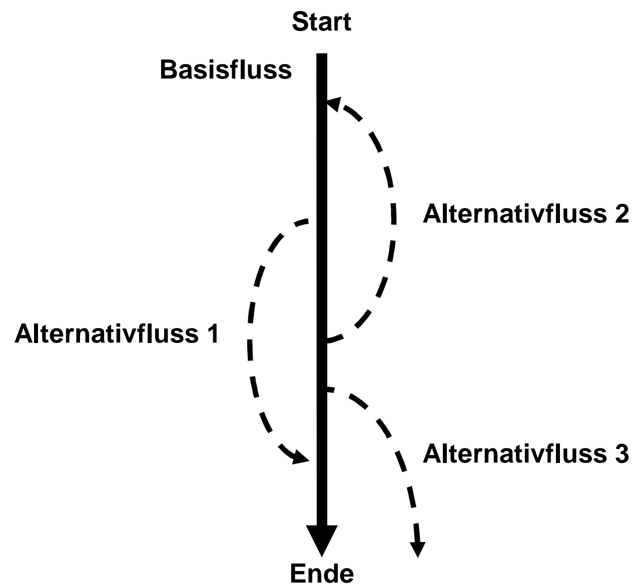

Abbildung 4: Einfacher Ereignisfluss für einen Use Case

müssen sie nicht wieder im Basisfluss enden, sondern können selbst ein alternatives Ende des Ereignisflusses darstellen (Alternativfluss 3). Ebenfalls ist denkbar (wenn auch nicht im Beispiel enthalten), dass ein Alternativfluss seinen Anfang nicht im Basisfluss nimmt, sondern in einem anderen Alternativfluss.

Nach dem RUP wird nun wie folgt für die unterschiedlichen Pfade durch den Ereignisfluss je ein Use Case Szenario identifiziert:

Szenario 1: Basisfluss

Szenario 2: Basisfluss, Alternativfluss 1, Basisfluss

Szenario 3: Basisfluss, Alternativfluss 3

Szenario 4: Basisfluss, Alternativfluss 2, Basisfluss

Szenario 5: Basisfluss, Alternativfluss 2, Basisfluss, Alternativfluss 1, Basisfluss

Szenario 6: Basisfluss, Alternativfluss 2, Basisfluss, Alternativfluss 3

Es fällt auf, dass hierbei von der Möglichkeit der mehrfachen Wiederholung des rückwärts gerichteten Alternativflusses 2 abstrahiert wird. In der RUP Dokumentation findet sich auch eine entsprechende Anmerkung (bzgl. dieser Abstraktion), ohne dass dabei jedoch in irgendeiner Weise auf die damit zusammenhängende Problematik oder mögliche Lösungen eingegangen wird.

Ausgehend von diesen Szenarien, den zugehörigen Use Case Beschreibungen und dem Design Modell werden nun Testfälle abgeleitet. Dabei wird explizit darauf hingewiesen, dass es nicht notwendig ist, alle

Use Case Szenarien zu testen.[19] Wie die eigentliche Ableitung eines Testfalls vor sich geht, wird nicht genauer beschrieben, bzw. es wird lediglich vermerkt, dass für jeden Testfall der Ausgangszustand, Testeingabedaten, erwartete Ausgabedaten sowie eventuell für den Ablauf des Testfalls zusätzlich benötigte Daten zu spezifizieren sind.

2.3.2 Feature Driven Development

Der Softwareentwicklungsprozess Feature Driven Development (FDD) wurde in [Coad+99] zum ersten Mal publiziert und ist untrennbar mit dem Namen Peter Coad verbunden, der eine gewichtige Rolle in den so genannten „Method Wars“ Mitte der 90er spielte. Bei FDD handelt es sich (zumindest noch) nicht um einen der „ganz großen“ Prozesse, was die Verbreitung angeht, er enthält jedoch eine Reihe interessanter Ansätze, insbesondere bzgl. des Designs einer Anwendung.

FDD beinhaltet fünf verschiedene Prozesse (s. Abbildung 5) und ist – wie der Name bereits beinhaltet – im Gegensatz zum RUP nicht Use Case getrieben sondern „Feature getrieben“. Ein Feature wird dabei definiert (vgl. [Palmer+02]) als eine kleine für den Anwender nützliche Funktion, die in der Form

<Aktion> <Resultat> <Objekt>

ausgedrückt wird (z.B. „*Berechne* den *Gesamtpreis* einer *Bestellung*“). Ein häufig bei der Erstellung von Use Cases anzutreffendes Problem, dass innerhalb eines Projektes unterschiedliche Auffassungen darüber existieren, auf welcher Abstraktionsebene Use Cases definiert werden, versucht FDD u.a. dadurch zu beseitigen, dass eine geplante Implementierungsdauer von zwei Wochen als streng einzuhaltende Obergrenze für Features festgelegt wird. Wird diese Grenze überschritten, so wird das betroffene Feature weiter aufgeteilt.

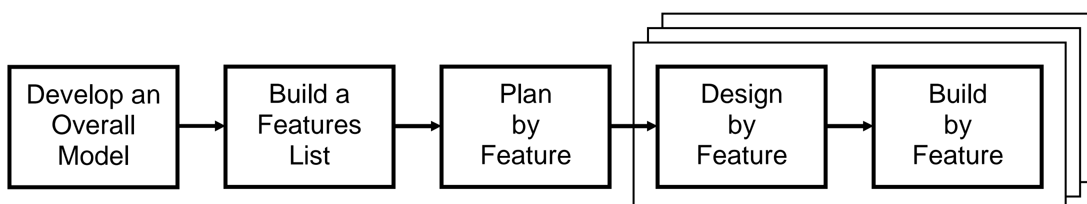

Abbildung 5: Die fünf Prozesse des Feature Driven Development

Zu Beginn eines FDD Projektes wird in Zusammenarbeit mit den Domänenexperten ein Domänenobjektmodell entworfen. Ausgehend von diesem Objektmodell und weiteren anforderungsanalytischen Tätigkeiten

[19] Leider wird diese in ihrer Absolutheit doch provokante Aussage in der RUP Dokumentation nicht weiter kommentiert.

wird dann von den Entwicklern eine Featureliste erstellt. Daraufhin wird ein grober Zeitplan erstellt, und es werden Verantwortlichkeiten für die einzelnen Features zugeteilt. Im tatsächlich iterativen Teil von FDD (den Prozessen „Design by Feature“ und „Build by Feature“) schließlich werden dann kleine Gruppen von Features in nicht länger als zwei Wochen dauernden Iterationen designed und implementiert. Dies wird so lange fortgesetzt, bis alle Features abgearbeitet wurden.

Im Rahmen des Design by Feature Prozesses ist das Erstellen eines oder mehrerer Sequenzdiagramme[20] für jedes einzelne Feature obligatorisch. In diesen Sequenzdiagrammen wird nicht nur der „Hauptablauf“ eines Features beschrieben, sondern auch alternative Pfade sowie Ausnahmebehandlungen.

Die Prozesse von FDD sind explizit auf das Design und die Implementierung einer Applikation ausgerichtet. Das heißt auch, dass das Testen an sich – insbesondere der Systemtest – keine nennenswerte Rolle innerhalb von FDD spielt. In [Palmer+02] ist dem Thema folglich auch nur ein 15-seitiges Kapitel gewidmet, in dem im Prinzip nur die generelle Wichtigkeit des Testens hervorgehoben wird und kurz festgestellt wird, dass dem Testen in FDD keine bedeutende Rolle eingeräumt wird (im Gegensatz zu z.B. XP). Begründet wird dies zu Beginn des 15-seitigen Kapitels wie folgt:

> *“One of the reasons FDD does not say much about testing is because, in many cases, the processes used for testing are not the main process issues with which a team or organization are struggling. Usually, the core development processes – the designing and constructing the software in a timely and cost-effective manner – are the main problem.”*

Der Rest des Kapitels führt einige grundlegende Begriffe des Testens ein wie zum Beispiel die Unterteilung in Unit-, Integrations-, System- und Akzeptanztests. Dem Problem des Erstellens von Testfällen widmet Palmer lediglich den folgenden Absatz und bleibt damit ähnlich vage wie Jacobson bei der Beschreibung des Unified Process:

> *“Test cases are prepared from requirements. This means that the input for test cases in an FDD project comes from the features list and domain walkthroughs in whatever form they were recorded. Input may also be taken from anything used as requirements during the development such, as existing use cases, screen mockups or prototypes, functional specifications, user manuals, business policy documents, etc.”*

---

20 Im Gegensatz zu einer Reihe anderer Autoren wie zum Beispiel [Jacobson+99] vernachlässigen [Palmer+02] die Unterschiede zwischen Sequenz- und Kollaborationsdiagrammen, so dass nach [Palmer+02] in diesem Designschritt prinzipiell auch Kollaborationsdiagramme erstellt werden können.

3 Testspezifikation durch UML Sequenzdiagramme

Ziel dieses Kapitels ist es, die Vorteile und Möglichkeiten aber auch die Probleme der Testspezifikation für dynamisches Testen auf der Basis von UML Sequenzdiagrammen zu beleuchten. Hierfür wird der Begriff der testbaren Sequenzdiagramme eingeführt. Insbesondere wird im Laufe des Kapitels gezeigt werden, dass das Testen anhand testbarer Sequenzdiagramme während der Systementwicklung eine Reihe von Vorteilen beim Unit- und Integrationstesten mit sich bringt. Naheliegenderweise ist es dafür notwendig, die Erstellung und Pflege von Sequenzdiagrammen[21] im Systementwicklungsprozess zu verankern. Zunächst soll beschrieben werden, wie dies geschehen kann.

3.1 Sequenzdiagramme im Systementwicklungsprozess

Sequenzdiagramme haben als allgemein bekannter Diagrammtyp der UML den Vorteil, dass sie häufig bereits zu Dokumentationszwecken verwendet werden. Insbesondere als Kommunikationsmittel zwischen Entwicklern kommt ihnen hierbei eine besondere Rolle zu, weil sich Programmabläufe in einem Sequenzdiagramm intuitiv darstellen lassen, da der dargestellte Ablauf auf die Sequenz der Methodenaufrufe reduziert wird.

Wie hoch der Abstraktionsgrad eines Sequenzdiagramms relativ zum tatsächlichen Code dabei ist, hängt zum einen von der Komplexität der Methoden ab. Bei im Schnitt sehr kurzen Methoden (z.B. einfache `get` und `set` Methoden) ist das Sequenzdiagramm nur eine unwesentliche Abstraktion vom eigentlichen Sourcecode. Werden die Methoden jedoch komplizierter (z.B. komplexe Berechnungsfunktionen), so ist ein Sequenzdiagramm in der Regel erheblich überschaubarer als das abgebildete Programmfragment. Zum anderen hängt der Abstraktionsgrad jedoch auch von der Intention des Erstellers des Diagramms ab, da er entscheidet, welche Objekte im Sequenzdiagramm auftauchen und welche nicht. Gibt es in einem Projekt beispielsweise eine Klasse, die dem Rest des Programms eine Schnittstelle für den Datenbankzugriff zur Verfügung stellt, so braucht in einem Sequenzdiagramm, das einen Datenbankzugriff dokumentiert, nur diese Klasse zu erscheinen. Von den restlichen eventuell vorhandenen Klassen, die den eigentlichen Datenbankzugriff realisieren, kann abstrahiert werden (siehe auch Abschnitt 3.5).

Nun ist es derzeit bei Softwareentwicklungen sicherlich fast nie der Fall, dass ein System von relevanter Größe vollständig (oder auch nur annähernd vollständig) durch Sequenzdiagramme beschrieben wird. Der Grund hierfür liegt auf der Hand: Der Nutzen ist relativ zum damit

[21] Soweit nicht ausdrücklich anders erwähnt, sind mit Sequenzdiagrammen in diesem Kapitel immer UML Sequenzdiagramme gemeint.

verbundenen Aufwand einfach zu gering. Die Situation könnte sich jedoch durch geeignete Werkzeuge stark verändern.

Geht man von einem dem Unified Process ähnlichen Systementwicklungsprozess aus, so ist die Spezifikation der Tests eine der ersten Tätigkeiten, die (meist parallel zum Systemdesign) durchgeführt werden. Auch beim aktuell recht intensiv diskutierten Extreme Programming steht die Testspezifikation am Beginn des Entwicklungsprozesses. Interessant ist hierbei, das beim Extreme Programming den bei der Testspezifikation entstehenden Testfällen eine weitaus größere Bedeutung zugesprochen wird als in vergleichbaren Prozessen. Abgesehen von der typischen Rolle eines Testfalls (Aufspüren von Fehlerzuständen bzw. Demonstrieren von Funktionalität), dienen Testfälle beim Extreme Programming ausdrücklich auch als Hauptform der Dokumentation und implizit auch als ein Teil des Designs, da durch sie Interaktionsmöglichkeiten zwischen Objekten auf Implementierungsebene festgelegt werden.[22]

Bisher finden beim Systemdesign und auch bei der Testspezifikation aus den oben angeführten Gründen Sequenzdiagramme nur punktuell Verwendung (bei FDD Projekten etwas mehr, bei RUP Projekten etwas weniger, bei XP Projekten u.U. gar nicht, s. auch Abschnitt 2.3). Wäre man jedoch in der Lage, anhand von Sequenzdiagrammen nicht nur Klassenschnittstellen und Programmabläufe zu spezifizieren, sondern dadurch gleichzeitig Testfälle festzulegen, die (mehr oder weniger) automatisch geprüft werden können, würde sich diese Phase erheblich effizienter gestalten lassen. Auch eines der Hauptargumente gegen die Verwendung von Sequenzdiagrammen, nämlich die aufwändige Konsistenzerhaltung bei eingeschränktem Nutzen, wäre damit beseitigt, da eine Änderung im Design sich unter Umständen zwar auf Sequenzdiagramme auswirkt, mit deren Aktualisierung man aber auf jeden Fall gleichzeitig die Testfälle konsistent erhält.

Ebenfalls ist die Verfeinerung des Systementwurfs anhand von Sequenzdiagrammen im Zuge der fortschreitenden Systementwicklung ausgesprochen intuitiv. So braucht man wie oben beschrieben in einer frühen Phase der Entwicklung lediglich die Klasse, die die Schnittstelle zu einer Datenbank zur Verfügung stellt, in die Sequenzdiagramme einbauen. Den restlichen diese Klasse benutzenden Code kann man dann bereits implementieren (und testen!), obwohl es die eigentliche Datenbankanbindung noch gar nicht gibt – vorausgesetzt man verfügt über ein Werkzeug, das in der Lage ist, Teststubs für noch nicht implementierte aber bereits mit Sequenzdiagrammen spezifizierte Klassen zu generieren (s. Abschnitt 4.2.5). Steht das Design der für den Datenbankzugriff verantwortlichen Klassen fest, kann man die

[22] Somit kann das Entwerfen von Tests vor Beginn der Implementierung als ein Schritt in Richtung einer formalen Spezifikation angesehen werden, da – wie im Falle der formalen Spezifikation – deutlicher wird, welche Funktionalität gefordert wird und die Prüfbarkeit dieser Funktionalität erhöht wird.

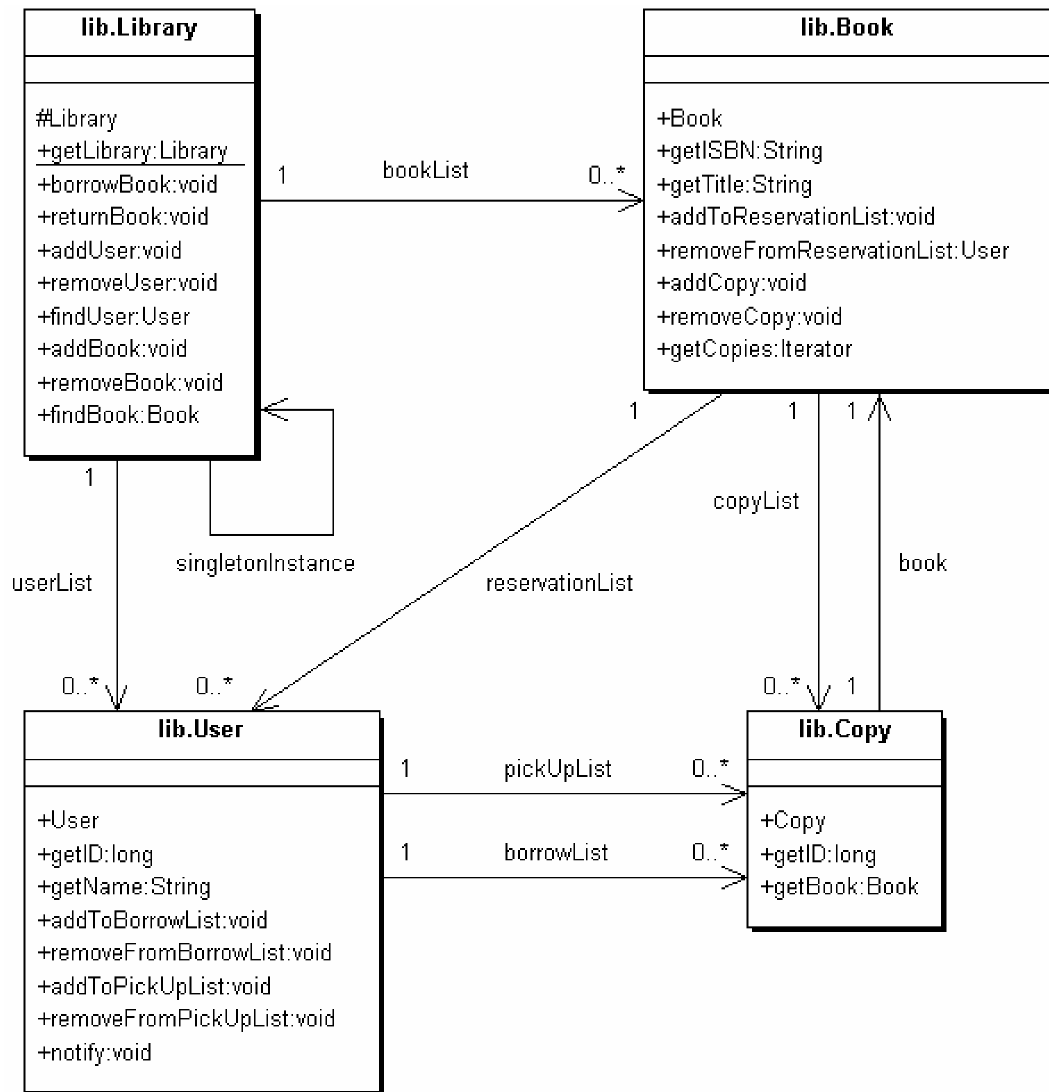

Abbildung 6: Klassendiagramm des Library Systems

entsprechenden Sequenzdiagramme um diese Klassen erweitern (s. Abschnitt 3.5).

3.2 Beispielanwendung Library

Abbildung 6 zeigt ein Klassendiagramm mit ausgewählten Klassen, Methoden und Assoziationen einer exemplarischen Bibliotheksanwendung, die im Rahmen dieser Arbeit zur Illustration der vorgestellten Konzepte verwendet wird.

Die Klasse `Library` implementiert das Singleton Pattern [Gamma+95] und verwaltet Listen von Büchern (Instanzen der Klasse `Book`) und Benutzern (Instanzen der Klasse `User`). Außerdem bietet sie Suchoperationen für Benutzer, Bücher und Kopien an.

Die Klasse `User` repräsentiert die Benutzer der Bibliothek, die Buchexemplare ausleihen. Neben Stammdaten eines Benutzers verwaltet sie daher eine Liste von Buchexemplaren, die der Benutzer ausgeliehen hat,

sowie eine Liste von Buchexemplaren, die für den Benutzer zum Abholen bereitstehen, d.h. fest für ihn reserviert sind.

Die Klasse `Book` repräsentiert die Daten, die allen Exemplaren des gleichen Buchs gemeinsam sind wie z.B. Titel und Autor. Instanzen dieser Klasse verfügen über eine Liste aller Exemplare des betreffenden Buches (Instanzen der Klasse `Copy`) sowie eine Liste von Benutzern, die darauf warten, dass eines der Exemplare dieses Buches für sie verfügbar wird (Reservierungswarteliste).

Eine Instanz der Klasse `Copy` repräsentiert ein einzelnes Exemplar eines Buches, dass man in der modellierten Bibliothek ausleihen kann. Abgesehen von einer Identifikationsnummer verfügt die Klasse `Copy` nur über eine einfache Assoziation zur Klasse `Book`, mittels der beschrieben wird, ein Exemplar welchen Buches die jeweilige Instanz der Klasse `Copy` ist.

Die Methoden `borrowBook` und `returnBook` der Klasse `Library` erwarten beide jeweils ein Objekt vom Typ `User` und vom Typ `Copy` als Parameter. Sie werden aufgerufen sowie ein Benutzer ein Buchexemplar ausleiht bzw. zurückgibt. Die Methode `notify` der Klasse `User` erwartet ein `Copy` Objekt als Parameter und dient dazu, den Benutzer darüber zu informieren (z.B. durch eine E-Mail), dass das gegebene Buchexemplar für ihn abholbereit ist.

3.3 Testbare Sequenzdiagramme

Die im Rahmen dieser Arbeit betrachteten Sequenzdiagramme entsprechen prinzipiell der UML 1.4 (s. Abschnitt 2.2.1). Allerdings müssen die hier betrachteten Sequenzdiagramme einige weitere Anforderungen erfüllen, damit sie sich zur Testspezifikation eignen.

Definition: Ein **testbares Sequenzdiagramm** ist ein der UML 1.4 entsprechendes Sequenzdiagramm, das zusätzlich den folgenden Anforderungen genügt:

1. Es enthält einen Aktor, der den Testtreiber bzw. das Testwerkzeug repräsentiert. Für diesen „Testtreiber-Aktor“ ist keine Klasse spezifiziert.
2. Es enthält mindestens ein Objekt (zusätzlich zum Testtreiber-Aktor).
3. Für jedes Objekt im Sequenzdiagramm ist die zugehörige Klasse bzw. ein zugehöriges Interface angegeben (abgesehen vom Testtreiber-Aktor).
4. Für jedes Objekt, auf dem eine nicht statische Methode aufgerufen wird, ist ein eindeutiger Name spezifiziert (dieser wird für die Testfallspezifikation verwendet, s.u.).
5. Der erste Methodenaufruf wird vom Testtreiber-Aktor initiiert.
6. Auf dem Testtreiber-Aktor werden keine Methoden aufgerufen.

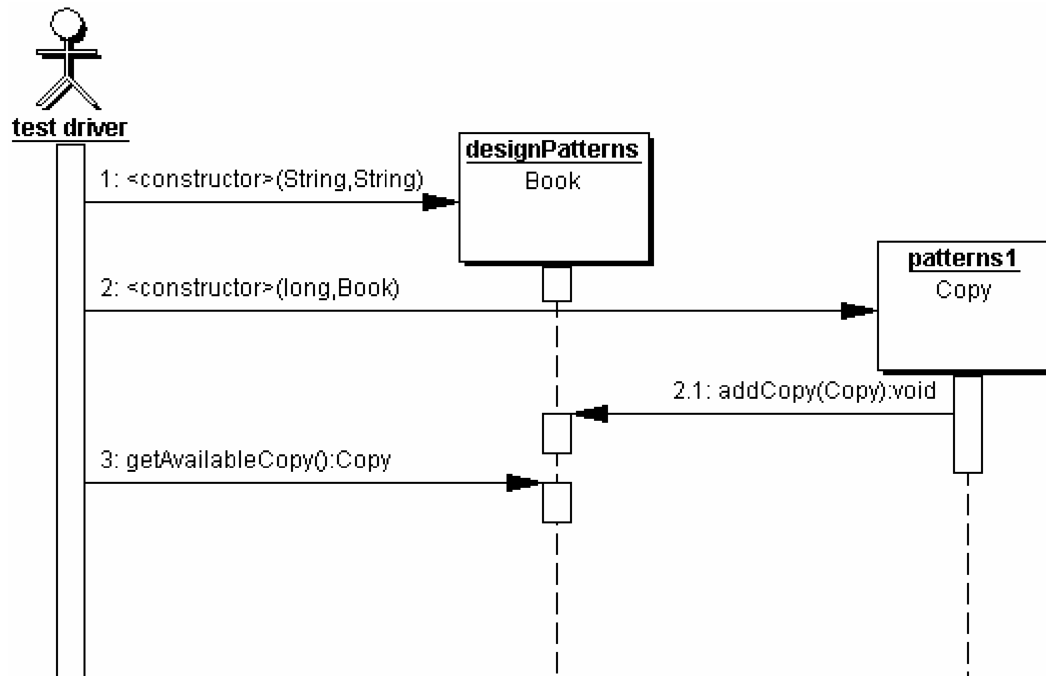

Abbildung 7: Einfaches testbares Sequenzdiagramm (logischer Testfall)

7. Alle Methodenaufrufe können genau einer deklarierten Methode in der für das entsprechende Objekt angegebenen Klasse bzw. Interface zugeordnet werden.
8. Der erste Aufruf auf jedem Objekt ist ein Konstruktoraufruf.
9. Die Methodenaufrufsequenz repräsentiert einen einzelnen Pfad durch das Programm (parallele Verarbeitung, asynchrone Aufrufe, Verzweigungen und Schleifen sind nicht erlaubt).

Testbare Sequenzdiagramme sind damit Diagramme auf Implementierungsebene, d.h. die für die darin enthaltenen Instanzen angegebenen Typen entsprechen Klassen oder Interfaces der Implementierung, und die enthaltenen Methodenaufrufe lassen sich 1:1 in der Implementierung enthaltenen Methoden zuordnen. Daraus folgt, dass ein testbares Sequenzdiagramm immer einen logischen Testfall beschreibt.

Testfälle werden üblicherweise durch einen Testtreiber bzw. ein Testwerkzeug angestoßen (siehe aber auch Stichwort „passives SeDiTeC“ im Ausblick, Kapitel 6). Diese „externe“ Rolle wird durch den Testtreiber-Aktor dargestellt. Methodenaufrufe, die vom Testtreiber-Aktor ausgehen, sollten vom Testwerkzeug automatisch aufgerufen werden. Zu beachten ist hierbei, dass die in Sequenzdiagrammen eventuell anzutreffende Verwendung von sekundären Aktoren zur Darstellung von externen Systemen (z.B. einer Datenbank oder eines weiteren Systems), die von der zu entwickelnden Anwendung benutzt werden, in testbaren Sequenzdiagrammen nur dann gestattet ist, wenn für diese Aktoren entsprechend Anforderung 3 konkrete Klassenangaben gemacht werden.

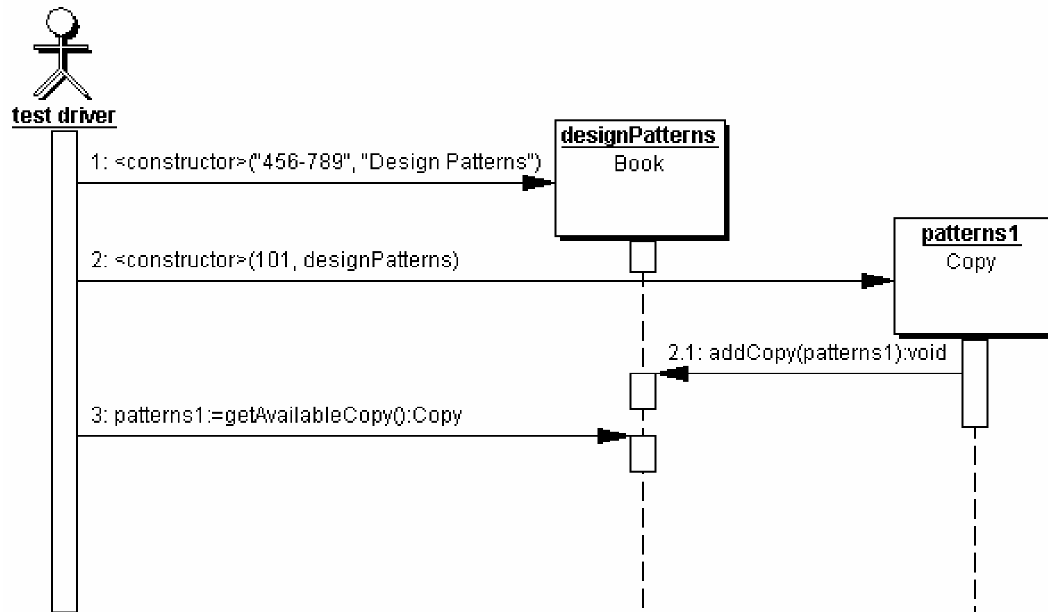

Abbildung 8: Einfaches testbares Sequenzdiagramm (konkreter Testfall)

Die Forderung, dass der erste Methodenaufruf auf jedem Objekt ein Konstruktoraufruf sein muss, ist zwar sehr restriktiv, aber (zumindest aus theoretischer Sicht) zwingend notwendig, um die Ausführbarkeit eines testbaren Sequenzdiagramms „von sich aus“ (ohne Einbeziehung von Testdatensätzen, s. Abschnitt 3.4) sicherzustellen. Schließlich kann man nicht davon ausgehen, dass bereits initialisierte Objekte (sprich: ein Ausgangszustand für den Testfall) vorhanden sind und muss daher die notwendigen Objekte erst selbst erstellen. Das im Rahmen dieser Arbeit entwickelte Testwerkzeug SeDiTeC kennt allerdings alternative Möglichkeiten, um einen Ausgangszustand herzustellen, so dass diese Anforderung an testbare Sequenzdiagramme für die bei SeDiTeC verwendeten Diagramme nicht erfüllt sein muss (s. Abschnitt 4.3). Ein Beispiel für ein einfaches testbares Sequenzdiagramm zeigt Abbildung 7.

3.4 Testfallspezifikation

Offensichtlich ist die Information, die sich dem testbaren Sequenzdiagramm entnehmen lässt, für sich allein genommen noch nicht ausreichend, um einen sinnvollen Test auszuführen. Es sind zwar die Typen der beteiligten Objekte und die aufzurufenden Methoden bekannt, es müssen jedoch noch konkrete Werte für die Parameter und Rückgabewerte der Methodenaufrufe ergänzt werden.

In den meisten professionellen UML Case Tools wie z.B. Together von Togethersoft [Togethersoft03] oder Rational Rose von Rational Software [Rational03] lassen sich Parameter und Rückgabewerte von Methodenaufrufen direkt textuell im Sequenzdiagramm spezifizieren.

Abbildung 8 enthält nun zusätzlich zu dem Sequenzdiagramm aus Abbildung 7 für alle Methodenaufrufe die jeweiligen Parameter und Rückgabewerte. Die Parameter eines Aufrufs erscheinen durch Komma

getrennt in den runden Klammern des Methodenaufrufs, ein Rückgabewert steht – wenn vorhanden – ganz links einschließlich eines „:=“ (siehe Methodenaufruf 3 in Abbildung 8). Die Parameter und Rückgabewerte (sowie eventuelle Exceptions, s. Abschnitt 3.4.1) bilden einen Testdatensatz. Ein konkreter Testfall besteht hier somit aus einem testbaren Sequenzdiagramm und einem dazu passenden Testdatensatz. Konkrete Testfälle werden im Rahmen dieser Arbeit daher auch als **Eingabesequenzdiagramme** bezeichnet.

Die Daten werden hierbei genauso angegeben, wie es auch in Java Quellcode der Fall wäre, d.h. dass Basistypen wie z.B. der `long` Parameter des zweiten Aufrufs (in diesem Fall `101`) ohne Weiteres einfach hingeschrieben werden können, während Strings in Anführungszeichen gesetzt werden müssen (z.B. der Titel des Buches „Design Patterns“).

Da es sich bei den Typen der Parameter und Rückgabewerte auch um Objekte handeln kann, braucht man eine Möglichkeit, um eindeutig auf ein bestimmtes Objekt verweisen zu können. Hierzu wird der Name verwendet, den jedes Objekt eines testbaren Sequenzdiagramms besitzen muss. Das `Copy` Objekt aus Abbildung 8 trägt beispielsweise den Namen „patterns1“. Dieser Name kann dann wie ein Variablenname verwendet werden, wie es auch in den Methodenaufrufen 2.1 und 3 geschieht (zuerst als Parameter, dann als Rückgabewert). Als Nullreferenz wird das Schlüsselwort `null` verwendet, was impliziert, dass diese Zeichenfolge nicht als Name für Objekte dienen kann.

Es wird deutlich, dass Strings im Rahmen der Testfallspezifikation wie Ausprägungen eines Basistypen und nicht wie Objekte behandelt werden, obwohl Strings in Java durch Instanzen der Klasse `String` repräsentiert werden. Dies geschieht aus zwei Gründen: Erstens werden Objekte vom Typ `String` in den meisten Fällen als Value-Objekte behandelt, d.h. man ist eher daran interessiert zu wissen, ob in einer Instanz der Klasse `String` zum Beispiel die Zeichenfolge „Design Patterns“ gespeichert ist, als dass man wissen wollte, welche konkrete Instanz (im Sinne von welche Referenz im Speicher) man nun gerade vor sich hat. Zweitens ist die Klasse `String` in Java ausgesprochen „value-orientiert“ implementiert. Dies äußert sich dadurch, dass eine Instanz der Klasse `String` nach ihrer Initialisierung nicht mehr verändert werden kann. Sämtliche manipulierende Methoden verändern nicht die Instanz, auf der sie aufgerufen werden, sondern erstellen stattdessen eine weitere Instanz.[23]

Dadurch, dass sich Objekte als Parameter und Rückgabewerte spezifizieren lassen, verringert sich auch die Bedeutung der Anforderung an testbare Sequenzdiagramme, dass der erste Aufruf auf jedem Objekt ein Konstruktoraufruf sein muss. Alternativ zum expliziten Konstruktor-

[23] Dies führt in Java Anwendungen, die in größerem Umfang Stringmanipulationen durchführen, häufig zu erheblichen Performanzproblemen ([Vermeulen+00]).

aufruf im Sequenzdiagramm kann ein Objekt auch zuerst als Parameter oder Rückgabewert erscheinen. Dann wird implizit davon ausgegangen, dass die entsprechende Methode das fragliche Objekt selbst direkt oder indirekt erstellt hat, das Sequenzdiagramm jedoch von diesem Vorgang abstrahiert.

Ganz allgemein ist es beim Testen unerlässlich, dass bereits vor der Ausführung eines Testfalls das erwartete Testresultat genau spezifiziert wurde ([Spillner+03]). Ein Eingabesequenzdiagramm beschreibt daher nicht nur die Testeingabedaten im herkömmlichen Sinn (die Daten, die nötig sind, um den Testfall ablaufen zu lassen), sondern beschreibt auch das erwartete Testresultat (das, was man beobachten möchte). Man betrachte hierzu noch mal das einfache testbare Sequenzdiagramm in Abbildung 8. In diesem Sequenzdiagramm befinden sich drei Objekte: der Aktor und jeweils eine Instanz der Klassen `Book` und `Copy`, die man offensichtlich testen möchte. Um den Testlauf durchführen zu können, sind lediglich die Methodenaufrufe notwendig, die vom Aktor initiiert werden (Aufrufe 1, 2 und 3), inklusive der zugehörigen Parameter. Der Methodenaufruf 2.1 sowie der Rückgabewert von Methodenaufruf 3 jedoch sind für die eigentliche Ausführung des Tests unnötig. Vielmehr dienen sie der Spezifikation des erwarteten Verhaltens der IUT. Konkret erwartet man in diesem Fall, dass nach der Initiierung der Methodenaufrufe 1 und 2 der Methodenaufruf 2.1 mit dem richtigen Parameter auf der richtigen Instanz von der IUT initiiert wird und dass der Methodenaufruf 3 eine Referenz auf die `Copy` Instanz zurückgibt.

3.4.1 Exceptions

Da man beim Testen nicht nur das Ziel verfolgt zu demonstrieren, dass eine bestimmte Funktionalität vorhanden ist, sondern insbesondere auch daran interessiert ist, Fehlerzustände zu finden, müssen Testfälle konstruiert werden, die die Anwendung mit speziellen bzw. extremen Situationen konfrontieren und damit auf ihre Robustheit testen. Um dazu in der Lage zu sein, ist es unumgänglich, das Werfen von Exceptions spezifizieren zu können.

Da die Spezifikation von geworfenen bzw. zu werfenden Exceptions in der UML bisher nicht vorgesehen ist, wird hierfür die Notation für die Rückgabewerte herangezogen. Soll spezifiziert werden, dass eine Methode nicht mittels einer `return` Anweisung sondern durch das Werfen einer Exception verlassen wird, so wird anstelle des Rückgabewertes im Diagramm das Schlüsselwort „Exception“ gefolgt von einem Doppelpunkt und dem Typ (inklusive Packageangabe) der entsprechenden Exception angegeben.

3.4.2 Unscharfe Daten

Um einen Testfall ausführen zu können, müssen dem Testtreiber für die von ihm aufzurufenden Methoden konkrete Daten vorliegen. Dies gilt jedoch nicht für die Teile des Testdatensatzes, die der Spezifikation des erwarteten Verhaltens der IUT dienen. Zum Beispiel ist es bei der Angabe

einer zu prüfenden Fließkommazahl häufig unpassend, diese konkret anzugeben, da sonst Abweichungen aufgrund von Rundungsfehlern zu unerwünschten Fehlermeldungen führen können. Hier bietet sich beispielsweise die Verwendung eines Intervalls an.

Es folgt eine Aufzählung der verschiedenen Typen von Daten, die in einem Testdatensatz auftreten können:

- **Konkrete Werte:** Ein konkreter Wert ist entweder eine Ausprägung eines Basisdatentyps (z.B. `true` oder `42`), ein String (z.B. „Design Patterns“) der Name eines Objekts (z.B. `patterns1`) oder das Schlüsselwort `null`.
- **Intervalle:** Intervalle bestehen aus der unteren und der oberen Grenze des Intervalls und werden durch einen Bindestrich voneinander getrennt. Die Angabe „1-5“ für einen Rückgabewert einer Methode vom Typ `int` bedeutet also, dass eine der Zahlen {1, 2, 3, 4, 5} als Rückgabewert erwartet wird. Intervalle können nur für numerische Basisdatentypen angegeben werden und werden immer als offene Intervalle interpretiert.
- **Aufzählungen:** Aufzählungen sind eine Menge von durch Komma getrennten konkreten Werten und / oder Intervallen.
- **Don’t Care:** Ein „Don’t Care“ wird durch ein einzelnes Fragezeichen kenntlich gemacht und dient dazu festzulegen, dass man keine Annahme über den an dieser Stelle beobachteten Wert trifft.

Insbesondere die letztgenannte Kategorie (Don’t Care) erleichtert die Testfallspezifikation erheblich, da man in bestimmten Situationen sonst dazu gezwungen wäre, Werte zu spezifizieren, die nur mit unvertretbarem Aufwand (oder gar nicht) zu ermitteln sind und u.U. die Aussagekraft des Testfalldatensatzes nicht erhöhen.

3.5 Verfeinerung von Sequenzdiagrammen

Die meisten Sequenzdiagramme bleiben nach ihrer anfänglichen Erstellung nicht stabil, sondern werden im Laufe der Systementwicklung verändert oder verfeinert. Dies trifft insbesondere für Sequenzdiagramme zu, die während der Analyse erstellt wurden, da sie normalerweise Instanzen und Methoden enthalten, die aus Analyseklassen stammen, die im Zuge des Designs zu Designklassen verfeinert werden müssen [Jacobson+98]. Aber auch die erste Version eines Sequenzdiagramms, die auf Klassen und Methoden des Designs beruht, wird typischerweise während der Entwicklung mehrfach verändert. Dies geschieht aus zwei Gründen:

1. Die ursprüngliche Vision des Designs könnte sich im Laufe des Projektes als undurchführbar oder zumindest verbesserungswürdig erweisen, so dass das Sequenzdiagramm mehr oder weniger stark

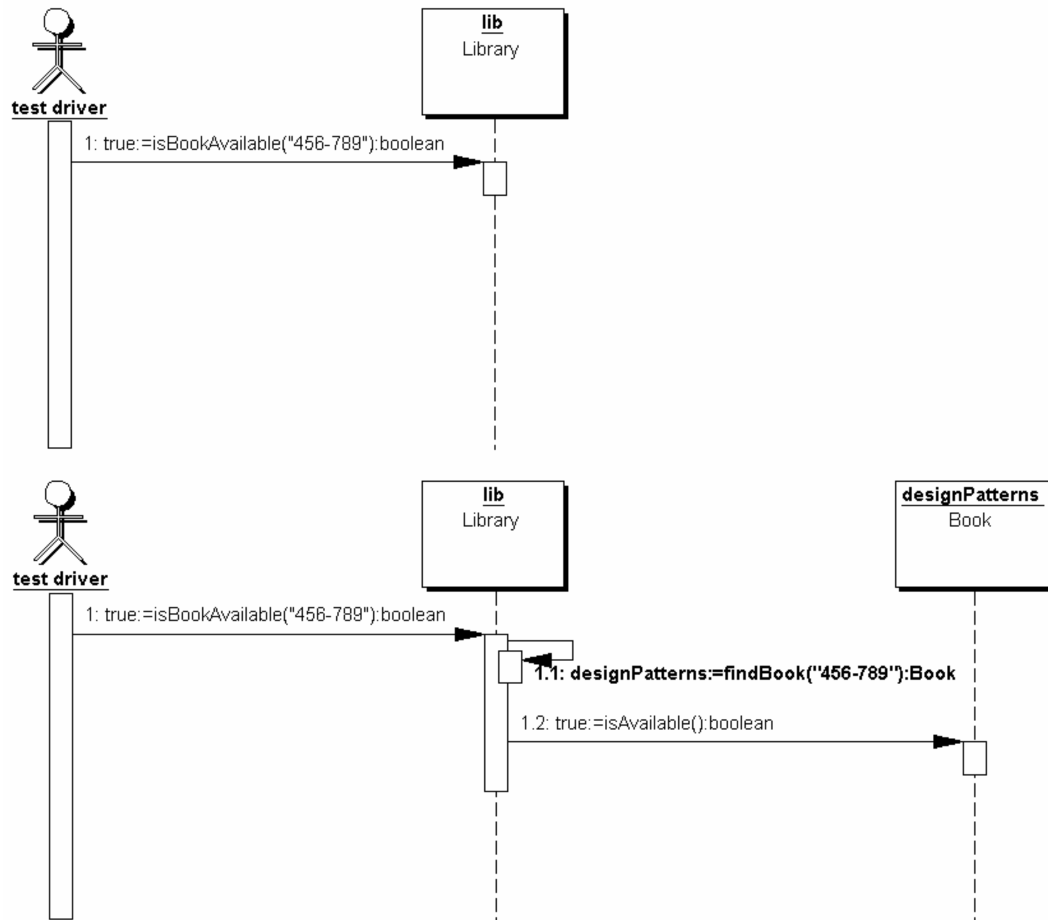

Abbildung 9: Verfeinerung von Sequenzdiagrammen

verändert wird (Methodenaufrufe werden entfernt, andere hinzugefügt, die Reihenfolge der Aufrufe ändert sich etc.).

2. Das Sequenzdiagramm könnte im Laufe des Projektes verfeinert werden, indem Informationen hinzugefügt werden, während die ursprünglichen Informationen erhalten bleiben. Zum Beispiel könnte die erste Version eines Sequenzdiagramms lediglich die Schnittstelle einer Klasse oder Komponente beschreiben, während in weiteren Versionen in zunehmenden Maße auch die interne Interaktion beschrieben wird (inkrementelles Design).

In diesem Sinne verfeinert ein Sequenzdiagramm *B* ein Sequenzdiagramm *A* genau dann, wenn die in *A* enthaltene Information eine Untermenge der in *B* enthaltenen Information ist. Für die zusätzliche Information, die die Verfeinerung ausmacht, kommen hauptsächlich zusätzliche Methodenaufrufe – häufig begleitet von zusätzlichen Objekten (s. Abbildung 9) – in Frage. Abgesehen davon kann für die in *A* bereits existierenden Objekte in *B* ein speziellerer Typ angegeben sein.

Entsteht im Zuge der Entwicklung ein neues Sequenzdiagramm *B* als Verfeinerung eines Sequenzdiagramms *A*, so können die für das Diagramm *B* zu erstellenden konkreten Testfälle von den für das

Diagramm *A* bereits existierenden konkreten Testfällen abgeleitet werden, da letztere komplett übernommen werden können und nur um die Daten für zusätzliche Methodenaufrufe erweitert werden müssen. Dabei ist es u.U. sinnvoll, das Diagramm *A* inklusive zugehöriger konkreter Testfälle auch nach der Verfeinerung durch *B* weiterhin als eigenständiges Sequenzdiagramm zu führen. Zum einen besteht dann die Möglichkeit, dass man verschiedene Verfeinerungen von einem Sequenzdiagramm ableiten kann. Zum anderen ist die Wahrscheinlichkeit hoch, dass Änderungen (nicht im Sinne einer Verfeinerung) an einem verfeinerten Sequenzdiagramm *B* lediglich die Verfeinerung betreffen, so dass das geänderte Diagramm *B´* ebenfalls als eine Verfeinerung des ursprünglichen Diagramms *A* aufgefasst werden kann. Ist das ursprüngliche Diagramm *A* (und seine konkreten Testfälle) dann noch vorhanden, erleichtert dies wiederum die Ableitung der konkreten Testfälle für *B´*.

3.6 Kombination von Sequenzdiagrammen

Beim eigentlichen Ausführen eines Testfalls, der mittels Sequenzdiagrammen spezifiziert ist, lassen sich prinzipiell drei Phasen unterscheiden:

1. Ausgangszustand der IUT herstellen
2. Zu testende Funktionalität der IUT ausführen
3. Überprüfen, ob sich die IUT im gewünschten Zustand befindet

Von zentraler Bedeutung ist hierbei die zweite Phase, da das Ausführen der zu testenden Funktionalität das ist, was das „Dynamische Testen“ ausmacht. Meist lässt sich die zu testende Funktionalität jedoch nicht ohne gewisse Vorkehrungen ausführen. Möchte man beispielsweise testen, ob die Buchrückgabe des Library Systems funktioniert, wird hierfür mindestens jeweils eine Instanz der Klassen `Library`, `Book`, `Copy` und `User` benötigt, die untereinander die notwendigen Beziehungen aufweisen (z.B. muss das Buchexemplar gerade vom entsprechenden Benutzer ausgeliehen sein, d.h. die `Copy` Instanz wird in der `borrowList` der `User` Instanz referenziert, vgl. Abbildung 6). Die Herstellung dieses Ausgangszustands ist die Aufgabe der ersten oben genannten Phase. Werden Testfälle vollständig auf der Basis von Sequenzdiagrammen spezifiziert, so muss das entsprechende Sequenzdiagramm zu Beginn die für die Herstellung des Ausgangszustands notwendigen Methodenaufrufe beinhalten.

In einem solchen Sequenzdiagramm sind daher zwei Kategorien von Methodenaufrufen enthalten. Zum einen gibt es die Methodenaufrufe, die den gewünschten Ausgangszustand ausgehend von Konstruktoraufrufen herstellen und zum anderen die Methodenaufrufe, die eigentlich getestet werden sollen. Da die den Anfangszustand herstellenden Methoden das Diagramm erheblich verlängern können, liegt es nahe zuzulassen, dass man ein Sequenzdiagramm, das zu testende Methoden enthält, mit einem oder mehreren Sequenzdiagrammen kombiniert, die zustandserzeugende Methodenaufrufe enthalten, die aber eigentlich nicht Gegenstand des

Testfalls sind. Hierbei wird unter „Kombinieren“ das Hintereinanderhängen der Sequenzdiagramme verstanden, d.h. es muss nicht nur festgelegt werden, welche Sequenzdiagramme kombiniert werden, sondern auch in welcher Reihenfolge dies geschieht.[24] Sequenzdiagramme, die lediglich zustandserzeugende Methodenaufrufe beinhalten, werden im Folgenden **zustandserzeugende Sequenzdiagramme** genannt.

Die Wiedererkennung eines Objektes, das in mehreren der kombinierten Sequenzdiagramme vorkommt, geschieht mittels des Objektnamens, der in den einzelnen Diagrammen identisch sein muss. Es ist durchaus möglich, das gleiche Sequenzdiagramm mehrfach in ein kombiniertes Sequenzdiagramm einzufügen. Abgesehen davon, dass die einzelnen Sequenzdiagramme eines kombinierten Sequenzdiagramms für sich wesentlich übersichtlicher und damit leichter handhabbar sind, als es ein großes Diagramm wäre, hat man durch die Möglichkeit, Sequenzdiagramme zu kombinieren, den Vorteil, dass sich dasselbe Sequenzdiagramm für mehrere logische Testfälle verwenden lässt, so dass die Anzahl notwendiger Diagramme insgesamt reduziert werden kann. Insbesondere zustandserzeugende Sequenzdiagramme lassen sich häufig wieder verwenden.

Die eigentliche Prüfung beim Testen auf der Basis von Sequenzdiagrammen besteht darin, das Verhalten der getesteten Methoden (die die zu testende Funktionalität implementieren) mit der Spezifikation abzugleichen (s. Abschnitt 3.4 bzw. 3.8). Gegebenenfalls ist man jedoch auch an dem Zustand interessiert, in dem sich das System nach Ausführung der zu testenden Methoden befindet. Dies kann z.B. mittels `get` Methoden geschehen, die die Attribute – und damit den Zustand – der beteiligten Objekte abfragen.[25]

Solche überprüfenden Methodenaufrufe, die nicht zu den eigentlich im Mittelpunkt des Tests stehenden Methodenaufrufen gehören, lassen sich prinzipiell auch wieder in einem eigenen Sequenzdiagramm spezifizieren. Solche Sequenzdiagramme werden im Folgenden als **zustandsprüfende**

[24] Ebenfalls wäre denkbar, eine hierarchische Kombination von Sequenzdiagrammen zuzulassen, wobei z.B. in einem Sequenzdiagramm *A* ein Methodenaufruf *F* enthalten ist, der seinerseits (in *A*) keine Methoden initiiert, jedoch auf ein Sequenzdiagramm *B* verweist, in dem die von *F* initiierten Aufrufe dann enthalten sind. Diese Art von Kombination ließe sich in vielen Fällen zur Beschreibung von Verfeinerungen verwenden (s. Abschnitt 3.5).

[25] Womit man bei einem – glücklicherweise eher theoretisch als praktisch interessanten – Problem angelangt wäre: Beobachtet man mittels der angesprochenen `get` Methoden einen unerwünschten Zustand des Systems, so kann das u.a. auch an fehlerhaft implementierten `get` Methoden liegen. Diese `get` Methoden selbst, deren Aufgabe das Abfragen des Zustands eines Objektes ist, lassen sich nur dann ausreichend testen, wenn man über Objekte verfügt, deren Zustand man zuverlässig kennt. Solche Objekte kann man jedoch nur mittels korrekt funktionierender `set` Methoden erzeugen, die sich wiederum nur mit Hilfe der `get` Methoden testen lassen. Durchbrechen kann man diesen Teufelskreis beispielsweise durch die Verifikation der meist sehr simplen `get` Methoden.

Sequenzdiagramme bezeichnet. Im Gegensatz zu den zustandserzeugenden Sequenzdiagrammen spielt bei den zustandsprüfenden Sequenzdiagrammen der Gesichtspunkt der Wiederverwendbarkeit nur eine untergeordnete Rolle, da sie im Regelfall sehr stark von der Kombination der vorhergehenden Sequenzdiagramme abhängen. Unterschiedliche Situationen, in denen man das gleiche zustandsprüfende Sequenzdiagramm verwenden kann, dürften daher zwar durchaus vorkommen, aber nur mit relativ hohem Aufwand zu erkennen sein.

Dabei ist zu beachten, dass die beiden Begriffe zustandserzeugendes und zustandsprüfendes Sequenzdiagramm lediglich Instrumente sind, die der Strukturierung der Testspezifikation dienen. Selbstverständlich braucht ein reguläres testbares Sequenzdiagramm weder aus mehreren einzelnen Sequenzdiagrammen zusammengesetzt werden, noch muss es eine Aufteilung in zustandserzeugende, zu testende und zustandsprüfende Methodenaufrufe geben. Da es sich bei allen in testbaren Sequenzdiagrammen vorkommenden Methoden um tatsächlich in den Klassen des Systems vorkommende Methoden handelt, können sie prinzipiell auch alle der Kategorie „zu testende Methoden" angehören. Die Erfahrung zeigt allerdings, dass eine Aufteilung des Diagramms häufig von Vorteil sein kann.

Des Weiteren zu beachten ist, dass die einzelnen Sequenzdiagramme, die zu einem testbaren Sequenzdiagramm kombiniert werden, selbst keine testbaren Sequenzdiagramme sein müssen, d.h. sie müssen nicht allen in Abschnitt 3.3 genannten Anforderungen genügen. Insbesondere gilt das für die Forderung, dass der erste Methodenaufruf auf jedem Objekt ein Konstruktor sein muss, die bei zu kombinierenden Sequenzdiagrammen offensichtlich nicht sinnvoll ist. Aber auch die Forderung, dass der erste Methodenaufruf in jedem Sequenzdiagramm vom Testtreiber-Aktor ausgehen muss, ist für zu kombinierende Sequenzdiagramme unnötig. Alle anderen Anforderungen bleiben jedoch bestehen.

Durch die alleinige Kombination von Sequenzdiagrammen erhält man jedoch keine ausführbaren Eingabesequenzdiagramme. Hierzu gilt es noch, die Testdatensätze sinnvoll (s. Abschnitt 3.7) zu kombinieren, die für die einzelnen Sequenzdiagramme existieren.

3.7 Plausibilitätsprüfung für Testfälle

Gerade durch die Möglichkeit, Sequenzdiagramme und zugehörige Testdatensätze zu kombinieren, wird die Erstellung eines Testfalls u.U. zu einer komplexen Aufgabe. Auch wenn im Rahmen dieser Arbeit eine automatische Prüfung der Richtigkeit eines Testfalls im semantischen Sinne nicht möglich ist, so lassen sich doch zumindest zwei strukturelle Anforderungen an Testfälle (statisch) überprüfen, deren Nichterfüllung eindeutig darauf hinweist, dass ein Testfall an sich fehlerhaft ist.

1. **Objekte müssen initialisiert worden sein:** Wie in Abschnitt 3.4 beschrieben, muss der erste Methodenaufruf auf jedem Objekt in einem Eingabesequenzdiagramm entweder ein Konstruktoraufruf

sein, oder das Objekt muss vor seinem ersten Methodenaufruf als Parameter oder Rückgabewert in Erscheinung treten.

2. **Vermeidung von Typfehlern:** Objekte müssen innerhalb eines Testdatensatzes „typgerecht“ verwendet werden. (Beispiel: Ein Objekt `user1`, das von einer Methode vom Typ `User` zurückgegeben wird, kann nicht im nächsten Methodenaufruf als Parameter vom Typ `Book` auftreten.)

Beide Anforderungen lassen sich jedoch nicht in allen Fällen statisch entscheiden. Bei Anforderung 1 können unscharfe Daten in einem Testdatensatz dafür sorgen, dass sich nicht entscheiden lässt, ob ein Objekt zu einem gegebenen Zeitpunkt bereits initialisiert ist. So könnte als Rückgabewert einer Methode beispielsweise eine Aufzählung von Objekten oder ein Don’t Care angegeben sein, so dass statisch nicht entscheidbar ist, ob ein Objekt zurückgegeben wird und wenn ja welches.

Anforderung 2 ist im Kontext von Vererbungshierarchien nicht immer statisch überprüfbar. Gibt z.B. eine Methode vom Typ `User` ein Objekt zurück und wird dieses Objekt im nächsten Aufruf als Parameter vom Typ `AdvancedUser` (wobei `AdvancedUser` von `User` erbt) verwendet, so kann dies zur Laufzeit zu einem Typfehler führen (falls das zurückgegebene Objekt vom dynamischen Typ `User` ist), muss aber nicht (falls das zurückgegebene Objekt vom dynamischen Typ `AdvancedUser` oder einem davon erbendem Typ ist).

3.8 Testresultat

Da wie in Abschnitt 3.4 beschrieben das erwartete Verhalten der IUT durch ein testbares Sequenzdiagramm mit einem zugehörigen Testdatensatz spezifiziert wird (Eingabesequenzdiagramm), bietet es sich an, den Testlauf ebenfalls mittels eines Sequenzdiagramms zu protokollieren, dem so genannten **beobachteten Sequenzdiagramm**.[26] Genauso wie das Eingabesequenzdiagramm besteht das beobachtete Sequenzdiagramm aus einem testbaren Sequenzdiagramm und einem zugehörigen Testdatensatz. Selbstverständlich enthält der Testdatensatz nun jedoch ausschließlich konkrete Werte (d.h. keine Intervalle, Aufzählungen etc.).

Wurde ein Testlauf basierend auf einem Eingabesequenzdiagramm ausgeführt und dabei ein beobachtetes Sequenzdiagramm protokolliert, so sind für die Testauswertung die beiden Diagramme zu vergleichen, um zu entscheiden, ob der Testlauf fehlerfrei[27] war. Falls der Test fehlerfrei war,

[26] Die technische Realisierung dieser Protokollierung soll hier erst einmal vernachlässigt werden. Verschiedene Möglichkeiten werden in Abschnitt 4.2.2 diskutiert.

[27] Der Begriff „fehlerfrei“ (bezogen auf einen Testlauf) wird hier streng genommen in der Bedeutung „das während des Testfalls beobachtete Istverhalten entsprach dem durch den Testfall spezifizierten Sollverhalten“ verwendet. Dabei ist anzumerken, dass dies nicht äquivalent zu der Aussage „der Testlauf beobachtete keine Fehlerwirkung“ ist,

werden keine weiteren Informationen benötigt. Ist der Test jedoch fehlgeschlagen (nicht fehlerfrei), so werden konkrete Informationen darüber benötigt, aufgrund welcher Abweichung(en) des beobachteten Sequenzdiagramms vom Eingabesequenzdiagramm diese Beurteilung des Testlaufs zustande kam. Hierzu ist festzulegen, ob grundsätzlich jede Abweichung dazu führt, dass der Testlauf als fehlgeschlagen bezeichnet wird oder ob es Abweichungen gibt, die nicht zu einer solchen Beurteilung führen, also toleriert werden.

Beim Vergleich eines Eingabesequenzdiagramms mit einem beobachteten Sequenzdiagramm sind die relevanten Eigenschaften der Diagramme:

- Identität der Methodenaufrufe
- Identität der Objekte, auf denen die Methoden aufgerufen werden (abgesehen von Umbenennungen)
- Reihenfolge der Methodenaufrufe
- Parameter und Rückgabewerte der Methodenaufrufe
- Exceptions

Falls alle diese Eigenschaften übereinstimmen, hat die IUT das erwartete Verhalten gezeigt und der Test verlief fehlerfrei. Falls mindestens ein Methodenaufruf des Eingabesequenzdiagramms im beobachteten Sequenzdiagramm nicht vorkommt oder die Reihenfolge der Methodenaufrufe nicht übereinstimmt, ist der Test fehlgeschlagen. Dies ist somit immer der Fall, wenn das beobachtete Sequenzdiagramm keine Verfeinerung des Eingabesequenzdiagramms ist (s. Abschnitt 3.5). Ebenfalls fehlgeschlagen ist der Test, wenn das beobachtete Sequenzdiagramm zwar eine Verfeinerung des Eingabesequenzdiagramms ist, aber mindestens ein Parameter oder Rückgabewert oder eine geworfene Ausnahme der zugehörigen Testdatensätze nicht übereinstimmt (bzw. nicht Teil einer eventuell angegebenen Aufzählung ist, nicht in ein angegebenes Intervall fällt etc.). Durch die Berücksichtigung der Testdatensätze weitet man den Begriff der Verfeinerung für Sequenzdiagramme auf die Verfeinerung von Eingabesequenzdiagrammen aus. Da die Verfeinerung von (Eingabe-)Sequenzdiagrammen im Rahmen dieser Arbeit ein zentrales Konzept darstellt und vom später beschriebenen Testwerkzeug SeDiTeC so weit wie möglich automatisch prüfbar sein sollte, folgt eine formale Definition. Hierzu wird zuerst der Begriff Verfeinerung für Werte und für Methodenaufrufe definiert.

Unter dem Begriff „Werte“ werden dabei Parameterwerte, Rückgabewerte und geworfene Ausnahmen verstanden. Ein Wert kann entweder ein konkreter Wert, eine Menge von Werten (Aufzählung oder Intervall) oder ein Don’t Care sein (s. Abschnitt 3.4.2).

---

da eine Abweichung des beobachteten vom spezifizierten Verhalten auch durch einen fehlerhaften Testfall zustande kommen kann.

Definition Verfeinerung von Werten: Ein Wert w_1 verfeinert einen Wert w_2 ($w_1 \leq_w w_2$) genau dann, wenn eine der folgenden Aussagen wahr ist:

- w_1 und w_2 sind konkrete, identische Werte.
- w_1 ist ein konkreter Wert, w_2 ist eine Menge von Werten und w_1 ist in w_2 enthalten.
- w_1 und w_2 sind Mengen von Werten und alle Elemente aus w_1 sind in w_2 enthalten.
- Bei w_2 handelt es sich um ein Don't Care.

Ein Methodenaufruf verfügt in diesem Kontext über folgende relevante Eigenschaften: Name, Parameterwerte, Rückgabewerte und geworfene Ausnahmen.[28]

Definition Verfeinerung von Methodenaufrufen: Ein Methodenaufruf m_1 verfeinert einen Methodenaufruf m_2 ($m_1 \leq_m m_2$) genau dann, wenn die folgenden Aussagen wahr sind:

- m_1 und m_2 bezeichnen Aufrufe derselben Methode oder m_1 bezeichnet einen Aufruf einer (in einer Erbenklasse) redefinierten Version der zu m_2 gehörigen Methode.
- Die Anzahl der Parameterwerte, Rückgabewerte und geworfenen Ausnahmen von m_1 und m_2 sind jeweils identisch.
- Jeder Wert von m_1 (d.h. Parameterwerte, Rückgabewerte und geworfenen Ausnahmen) ist eine Verfeinerung des entsprechenden Wertes von m_2.

Für die Definition der Verfeinerung von Eingabesequenzdiagrammen sind einige Annahmen zu treffen. Gegeben seien...

- zwei Sequenzdiagramme A und B mit den Objekten $o_{A1}...o_{An}$ und $o_{B1}...o_{Bm}$ und den Methodenaufrufen $m_{A1}...m_{Ap}$ und $m_{B1}...m_{Bq}$
- eine Funktion $t: O \rightarrow T$, die die Objekte der beiden Sequenzdiagramme auf ihre Typen abbildet
- eine Funktion $h_1: M \rightarrow O$, die die Methodenaufrufe der beiden Sequenzdiagramme auf das jeweilige den Methodenaufruf initiierende Objekt abbildet
- eine Funktion $h_2: M \rightarrow O$, die die Methodenaufrufe der beiden Sequenzdiagramme auf das jeweilige Objekt abbildet, auf dem der Methodenaufruf initiiert wird

---

[28] Wobei ein Methodenaufruf im konkreten Fall selbstverständlich nur über entweder einen Rückgabewert oder über eine geworfene Ausnahme oder über keines der beiden verfügt (bei jeweils beliebiger Parameterwerteanzahl).

Für zwei Typen t_1 und t_2 wird geschrieben $t_1 \leq_t t_2$, wenn t_1 und t_2 identisch sind oder t_1 ein (direkter oder indirekter) Erbe von t_2 ist oder t_1 t_2 implementiert (falls t_2 ein Interface ist).

Für zwei Methodenaufrufe m_1 und m_2 wird geschrieben $m_1 <_o m_2$, wenn m_1 vor m_2 aufgerufen wird. Da in Eingabesequenzdiagrammen alle Methodenaufrufe sequentiell initiiert werden, gilt für zwei beliebige Methodenaufrufe m_1 und m_2 innerhalb eines Eingabesequenzdiagramms immer entweder $m_1 <_o m_2$ oder $m_2 <_o m_1$.

Definition Verfeinerung von Eingabesequenzdiagrammen: Ein Eingabesequenzdiagramm A verfeinert ein Eingabesequenzdiagramm B genau dann, wenn eine injektive Abbildung f: $O_B \rightarrow O_A$ und eine injektive Abbildung g: $M_B \rightarrow M_A$ existieren, so dass gilt:

- Typkonformität: Ein Objekt o_B aus B wird durch die Funktion f immer auf ein Objekt aus A abgebildet, dessen Typ konform zum Typ o_B ist.

  $t(f(o_{Bi})) \leq_t t(o_{Bi})$ $\quad$ $i = 1...m$

- Verfeinerung von Methodenaufrufen: Ein Methodenaufruf m_B aus B wird durch die Funktion g immer auf einen Methodenaufruf aus A abgebildet, der eine Verfeinerung von m_B darstellt.

  $g(m_{Bi})) \leq_m m_{Bi}$ $\quad$ $i = 1...q$

- Erhaltung der initiierenden Objekte: Wird ein Methodenaufruf m_B aus B von einem Objekt o_B aus B initiiert, so wird der m_B durch die Funktion g zugeordnete Methodenaufruf aus A durch das o_B durch die Funktion f zugeordnete Objekt aus A initiiert.

  $f(h_1(m_{Bi})) = h_1(g(m_{Bi}))$ $\quad$ $i = 1...q$

- Erhaltung der aufgerufenen Objekte: Wird ein Methodenaufruf m_B aus B auf einem Objekt o_B aus B initiiert, so wird der m_B durch die Funktion g zugeordnete Methodenaufruf aus A auf dem o_B durch die Funktion f zugeordnete Objekt aus A initiiert.

  $f(h_2(m_{Bi})) = h_2(g(m_{Bi}))$ $\quad$ $i = 1...q$

- Erhaltung der Methodenaufrufreihenfolge: Wird der Methodenaufruf m_{Bi} aus B vor dem Methodenaufruf m_{Bj} aus B initiiert, so wird der m_{Bi} durch die Funktion g zugeordnete Methodenaufruf aus A vor dem m_{Bj} durch die Funktion g zugeordneten Methodenaufruf aus A initiiert.

  $m_{Bi} <_o m_{Bj} \Rightarrow g(m_{Bi}) <_o g(m_{Bj})$ $\quad$ $i = 1...q, j = 1...q$

Die Frage ist nun, was eine sinnvolle Beurteilung für einen Testlauf ist, bei dem das beobachtete Sequenzdiagramm zwar nicht genau dem Eingabesequenzdiagramm entspricht, aber eine Verfeinerung davon ist. Die Antwort auf diese Frage kann jedoch von Fall zu Fall unterschiedlich ausfallen. Der Grund hierfür liegt darin, dass ein beobachtetes

Sequenzdiagramm, das verglichen mit dem Eingabesequenzdiagramm zusätzliche Informationen enthält (sprich eine Verfeinerung desselbigen ist), nicht notwendigerweise auf einen gefundenen Fehlerzustand hinweist, sondern u.U. nur bedeutet, dass die Testspezifikation nicht jedes Detail der Programmausführung beinhaltete. Handelt es sich bei den zusätzlichen Informationen beispielsweise um zusätzliche Methodenaufrufe, so kann es sich bei diesen Methodenaufrufen entweder einfach um die Implementierung von im Eingabesequenzdiagramm enthaltenen Methodenaufrufen handeln oder aber um unerwünschte Seiteneffekte, die als Fehler zu behandeln wären.

3.9 Nutzen bei Unit- und Integrationstests

Ein beim Durchführen von Unit Tests häufig anzutreffendes Problem ist die Tatsache, dass eine zu testende Klasse nicht in dem Sinne alleine „lauffähig“ ist, da sie mit anderen Klassen interagiert. Daher müssen für diese anderen Klassen üblicherweise Teststubs entwickelt werden, um das Testen auf Unittestebene zu ermöglichen. Das Entwickeln von Teststubs ist jedoch meist recht aufwendig, so dass abgewogen werden muss zwischen dem zu erwartenden Nutzen von Tests und dem dafür zu erbringenden Aufwand. Häufig führt das dazu, dass Unittests nur in geringem Umfang durchgeführt werden und der Schwerpunkt des Testens eher in Richtung Integrations- bzw. Systemtests verschoben wird.

Aber auch beim Integrationstest ist das Erstellen von Teststubs und der damit verbundene Aufwand ein wichtiges Thema. So ist die Minimierung des Aufwands für die Erstellung von Teststubs eines der Hauptentscheidungskriterien beim Festlegen der Integrationsstrategie. Dies hat zur Folge, dass Varianten der Bottom-Up Integrationsstrategie in der Praxis vorherrschen. Der Rational Unified Process beispielsweise sieht bei seinen Betrachtungen grundsätzlich eine Bottom-Up Integrationsstrategie vor. Dies hat jedoch den Nachteil, dass damit implizit auch maßgeblicher Einfluss auf die Reihenfolge genommen wird, in der verschiedene Teile einer Anwendung entwickelt werden, wodurch einiges an Flexibilität in diesem Bereich verloren geht.

Das Erstellen von Teststubs ist um so aufwändiger, je „realistischer“ sich der Teststub verhalten soll. Daher mangelt es Teststubs häufig an Flexibilität – z.B. geben Methoden unter Umständen bei jedem Aufruf nur ein und denselben Standardwert zurück, oder verfügen nicht über die Funktionalität, ihrerseits andere Methoden aufzurufen.

Testet man auf der Basis von Sequenzdiagrammen, so sind in einem Testfall die relevanten Informationen über das Verhalten der beteiligten Objekte vollständig enthalten, wobei mit Verhalten hier gemeint ist, welche Methoden mit welchen Parametern in welcher Reihenfolge aufzurufen sind, welche Werte zurückzugeben sind und welche Ausnahmen evtl. geworfen werden müssen. Die passende Werkzeugunterstützung vorausgesetzt erlauben diese Informationen aber nicht nur die Überprüfung des Verhaltens dieser Instanzen, sondern auch die

Ersetzung dieser Instanzen durch Teststubs, die in ihrer Funktionalität im Idealfall lediglich durch die im Sequenzdiagramm enthaltene Information eingeschränkt sind.[29]

Die Folge ist, dass man beim Testen auf der Basis von Sequenzdiagrammen den Aufwand für das Erstellen von Teststubs vernachlässigen kann. Dies ermöglicht sowohl auf Unit- als auch auf Integrationstestebene ein effizienteres und damit gründlicheres Testen und erhöht die Flexibilität des gesamten Entwicklungsprozesses, da ein erheblicher limitierender technischer Faktor ausgeschaltet wird.

3.10 Vor- und Nachbedingungen

Eine weitere nahe liegende Thematik im Kontext des Testens auf der Basis von Sequenzdiagrammen ist die Spezifikation von Vor- und Nachbedingungen für Methoden [Meyer97]. Die verschiedenen Vorteile für die Qualitätssicherung und das Testen, die Produkt des Einsatzes von Vor- und Nachbedingungen sind (bessere Dokumentation und damit Benutzbarkeit von Schnittstellen, leichtere Fehlerlokalisierung im Falle der tatsächlichen Prüfung der Bedingungen etc.), können sicherlich als allgemein anerkannt gelten.

Auf der anderen Seite bedeutet die Spezifikation von adäquaten Vor- und Nachbedingungen natürlich auch zusätzlichen Aufwand. Dieser Effekt wird zusätzlich dadurch vergrößert, dass Java – im Gegensatz zu Sprachen wie Eiffel – praktisch keinerlei diesbezügliche Unterstützung bietet. Während der Aufwand in Eiffel sich im Prinzip darauf beschränkt, die entsprechenden Bedingungen als boolesche Ausdrücke unter der Verwendung der Schlüsselwörter `require` und `ensure` im Quelltext einzufügen und dann automatisch Exceptions im Falle einer Verletzung einer Bedingung generiert werden, muss eine entsprechende Funktionalität in Java „zu Fuß“ selbst implementiert werden. Gleiches gilt selbstverständlich für die in Eiffel vorhandene Möglichkeit, die Überprüfung der Bedingungen – z.B. aus Performanzgründen – per Compilerswitch in der Applikation zu aktivieren oder zu deaktivieren.

Daher liegt es nahe, an ein Sequenzdiagramme als Testspezifikation verwendendes Testwerkzeug für Java die Anforderung zu stellen, dass es die Prüfung von Vor- und Nachbedingungen von Methoden in geeigneter Weise unterstützt. Wie diese Unterstützung in SeDiTeC realisiert ist, wird in Abschnitt 4.4 beschrieben.

3.11 Testabdeckung

Wie in Abschnitt 2.1.1 beschrieben, besteht eines der größten Probleme beim Testen darin zu entscheiden, wann der Testprozess beendet werden

[29] Allerdings dürfen die im Testfall für eine durch einen Stub zu simulierende Instanz enthaltenen Daten selbstverständlich nicht unscharf im Sinne von Abschnitt 3.4.2 sein.

kann. Um dieses Problem zu lösen werden Testabdeckungskriterien definiert. Sind diese erfüllt, so gilt die IUT als ausreichend getestet.

Zu den häufig anzutreffenden Testabdeckungskriterien zählen die Überdeckungsmaße für Sourcecode wie z.B. Anweisungs-, Zweig-, Pfadüberdeckung etc. (s. [Liggesmeyer02]). Erfolgt das Design mittels Zustandsdiagrammen, so bieten sich Zustands- bzw. Transitionsüberdeckung als Testabdeckungskriterien an (ebenfalls in [Liggesmeyer02] beschrieben).

Alleine an diesen beiden Beispielen wird bereits deutlich, dass der Begriff Testabdeckung nicht ganz unproblematisch ist, da die durch eine bestimmte Testabdeckung getroffene Aussage relativ zur Bezugsgröße gesehen werden muss. Bei den verschiedenen Codeabdeckungsmaßen ist die Aussage, dass man hundertprozentige Testabdeckung erreicht hat, gleichbedeutend mit der Aussage, dass man gemäß dem verwendeten Kriterium alles ausreichend getestet hat, was man bereits implementiert hat. Stellt man noch auf andere Art und Weise sicher, dass man auch tatsächlich alles implementiert, so dass die Anwendung alle funktionalen Anforderungen erfüllt, ist diese Familie von Testabdeckungskriterien aus theoretischer Sicht sehr elegant. Aus praktischer Sicht spielen sie jedoch gerade im Bereich der objektorientierten Softwareentwicklung nur eine geringe Rolle (s. Abschnitt 2.1.2).

Zustandsdiagramme dagegen werden als geeignetere Basis für das Testen objektorientierter Programme angesehen. Welche Aussage kann man jedoch treffen, wenn man eine hundertprozentige Testabdeckung bzgl. Zustands- bzw. Transitionsüberdeckung von Zustandsdiagrammen erreicht hat? Die Antwort ist: Lediglich die, dass man die Diagramme getestet hat, denn zusätzlich zu dem auch bei der Codeabdeckung vorhandenen Problem, dass man keine Aussage darüber treffen kann, ob die Zustandsdiagramme alle an die Anwendung gestellten funktionalen Anforderungen beschreiben, kommt noch ein weiteres Problem. Man kann nun auch (ohne weitere Maßnahmen) keine Aussagen mehr darüber treffen, ob man die gesamte Implementierung getestet hat. Zusätzliche (u.U. unerwünschte) Funktionalität würde nicht entdeckt werden.

Die gleiche Problematik besteht für Sequenzdiagramme. Allerdings mit dem Unterschied, dass es zumindest für die im Rahmen dieser Arbeit verwendeten Eingabesequenzdiagramme selbst in dem Sinne keine Abdeckungsmaße gibt – abgesehen von dem trivialen, dass für jedes Sequenzdiagramm mindestens ein Testdatensatz existieren (und getestet sein) muss. Mehr lässt sich aus der Struktur der Eingabesequenzdiagramme nicht ableiten, da sie nur genau eine Sequenz von Methodenaufrufen modellieren.

Dem Problem, dass u.U. nicht genügend Sequenzdiagramme spezifiziert werden, um auf deren Basis ein erklärtes Testziel zu erreichen, lässt sich nur durch geeignete Maßnahmen im Softwareentwicklungsprozess begegnen (s. Abschnitt 2.3, aber auch in dieser Hinsicht interessante Publikationen wie [Kirani94] und [Spillner98]). Geht man hierbei von der

Verwendung von Use Cases aus, ist es normalerweise wenig praktikabel zu versuchen, einen Use Case vollständig innerhalb eines Sequenzdiagramms abzubilden. Vielmehr wird ein erstes Sequenzdiagramm, dass auf einer abstrakten Ebene für einen Use Case erstellt wurde, im Laufe des Design weiter verfeinert und schon aus Gründen der Übersichtlichkeit auf mehrere Diagramme verteilt werden. Je nach Komplexität des Use Cases bieten sich beispielsweise die Kanten eines Ereignisflussgraphen an, um durch ein (möglicherweise kombiniertes) Sequenzdiagramm beschrieben zu werden. Durch Kombination dieser Diagramme lässt sich dann eine Pfadabdeckung bzgl. des Ereignisflussgraphen für einen Use Case erreichen – ein Abdeckungskriterium, was die in Abschnitt 2.1.1 genannten Axiome von Weyuker erfüllt. Dabei solle man jedoch nicht aus den Augen verlieren, dass nicht alle diese Tests notwendigerweise auf der Basis von Sequenzdiagrammen spezifiziert werden müssen bzw. sollten (s. Abschnitt 3.12).

Dass die Pfadabdeckung bzgl. der Ereignisschritte für einen Use Case Weyukers Axiome erfüllt, impliziert (leider) nicht, dass man sich mit der Erfüllung dieses Testendekriteriums zufrieden geben sollte bzw. könnte. Zumindest muss man diese Pfadabdeckung noch kombinieren mit einer Äquivalenzklassen- bzw. Grenzwertanalyse, da Repräsentanten unterschiedlicher Äquivalenzklassen nicht notwendigerweise zu unterschiedlichen Sequenzdiagrammen führen, wohl aber zu zusätzlichen Erkenntnissen. Als Beispiel sei der Ausleihvorgang im Bibliothekssystem genannt: Geht man davon aus, dass es verschiedene Benutzerklassen gibt, so würde der Standardauswahlvorgang für alle Benutzerklassen wahrscheinlich zu dem (bis auf die Objekterzeugung) gleichen Sequenzdiagramm führen. Daraus sollte man aber nicht schließen, dass es ausreicht, den Ausleihvorgang nur mit *einer* Benutzerklasse zu testen.

Das Problem des Auffindens von überflüssiger bzw. unerwünschter Funktionalität mittels Testens zu lösen, ist ähnlich schwer, wie das Problem, mittels Testen zu zeigen, dass eine Anwendung eine bestimmte Funktionalität hat.[30] Nicht nur deswegen wird es in der Praxis häufig ignoriert. Ein kleiner, relativ leicht realisierbarer Schritt in Richtung einer Lösung für dieses Problem besteht in der Messung der Anweisungsüberdeckung der vorhandenen Testfälle. Für die Stellen des Codes, die durch die Tests nicht erreicht werden, gilt es zu analysieren, ob es sich um unerwünschte Funktionalität handelt oder ob weitere Tests zu spezifizieren sind (oder ob diese Zeilen zwar erwünscht aber nicht mit vertretbarem Aufwand durch Tests auszuführen sind).[31]

---

[30] In einem Fall geht es darum zu zeigen, dass eine bestimmte Funktionalität für alle möglichen Eingaben vorhanden ist. Durch Testen lässt sich das aber meist nur exemplarisch zeigen. Im anderen Fall besteht das Problem darin zu zeigen, dass alle(!) nicht in der Anforderungsspezifikation enthaltenen Funktionalitäten tatsächlich nicht vorhanden sind – eine durch Testen praktisch unlösbare Aufgabe.

[31] Nochmals verschärft wird das Problem der „überflüssigen Funktionalität“, wenn der Erweiterbarkeit der zu entwickelnden Anwendung eine hohe Bedeutung zukommt. In

3.12 Probleme und Lösungsmöglichkeiten

Es sollen nun einige tatsächliche und vermeintliche Probleme angesprochen werden, die im Zusammenhang mit dem Testen auf der Basis von Sequenzdiagrammen auftreten können. Insbesondere werden die in der Einleitung erwähnten von Robert Binder angeführten Kritikpunkte diskutiert (s. Abschnitt 1.1).

Der erste von Binder genannte Kritikpunkt betraf die mangelhafte Eignung von Sequenzdiagrammen, komplexe Kontrollstrukturen benutzerfreundlich darzustellen. In Sequenzdiagrammen werden bedingungsabhängige Verzweigungen dargestellt, indem zwei oder mehr Nachrichtenpfeile an derselben Stelle ihren Ursprung haben. Da ab dieser Verzweigung die verschiedenen Ausführungspfade parallel nebeneinander gezeichnet werden, wird das Diagramm in der Tat meist sehr schnell unübersichtlich. Aus zwei Gründen hat dieses Problem in diesem Kontext jedoch nur geringe praktische Relevanz: Zum einen bewegt man sich beim Erstellen von Diagrammen auf einer abstrakteren Ebene als beim Programmieren selbst, so dass Kontrollstrukturen häufig gar nicht in dem Sinne im Diagramm modelliert werden (müssen). Zum anderen kann man durch die Kombination von entsprechenden Teil-Sequenzdiagrammen die unübersichtliche parallele Darstellungsweise elegant vermeiden.

Die Möglichkeit, Sequenzdiagramme zu kombinieren, beseitigt auch das zweite von Robert Binder angesprochene Problem, nämlich dass Sequenzdiagramme die Darstellung von dynamischem Binden nicht direkt unterstützen. Hierfür ist lediglich der jeweils tatsächlich klassenspezifische Teil (der meist aus der Konstruktion der Objekte besteht), von dem Teil zu trennen, der die „interessanten“ dynamischen Aufrufe enthält (s. Abbildung 10).

Bleibt noch die letzte Aussage von Robert Binder, dass die Implementierung beispielsweise eines Use Cases meist nicht in einem, sondern nur in mehreren teilweise redundanten Sequenzdiagrammen modelliert werden kann. Ähnlich wie beim ersten Punkt („komplexe Kontrollstrukturen“) gilt aber auch hier, dass es einerseits eine Frage der Abstraktionsebene der Diagramme ist, inwieweit das Problem überhaupt auftritt. Andererseits lässt sich das Problem redundanter Information durch sich wiederholender Methodensequenzen wiederum durch Kombination von Sequenzdiagrammen deutlich verringern. Die Verfeinerung von Sequenzdiagrammen bietet zusätzlich noch die Möglichkeit, die Abstraktionsebene, auf der beispielsweise ein Use Case modelliert wird, an relevanten Stellen geeignet anzupassen.

diesem Fall werden Use Cases (bzw. Anforderungen ganz allgemein) häufig „übererfüllt“, um bereits Vorkehrungen für Erweiterungen zu treffen, die aber im Detail noch nicht bekannt sind.

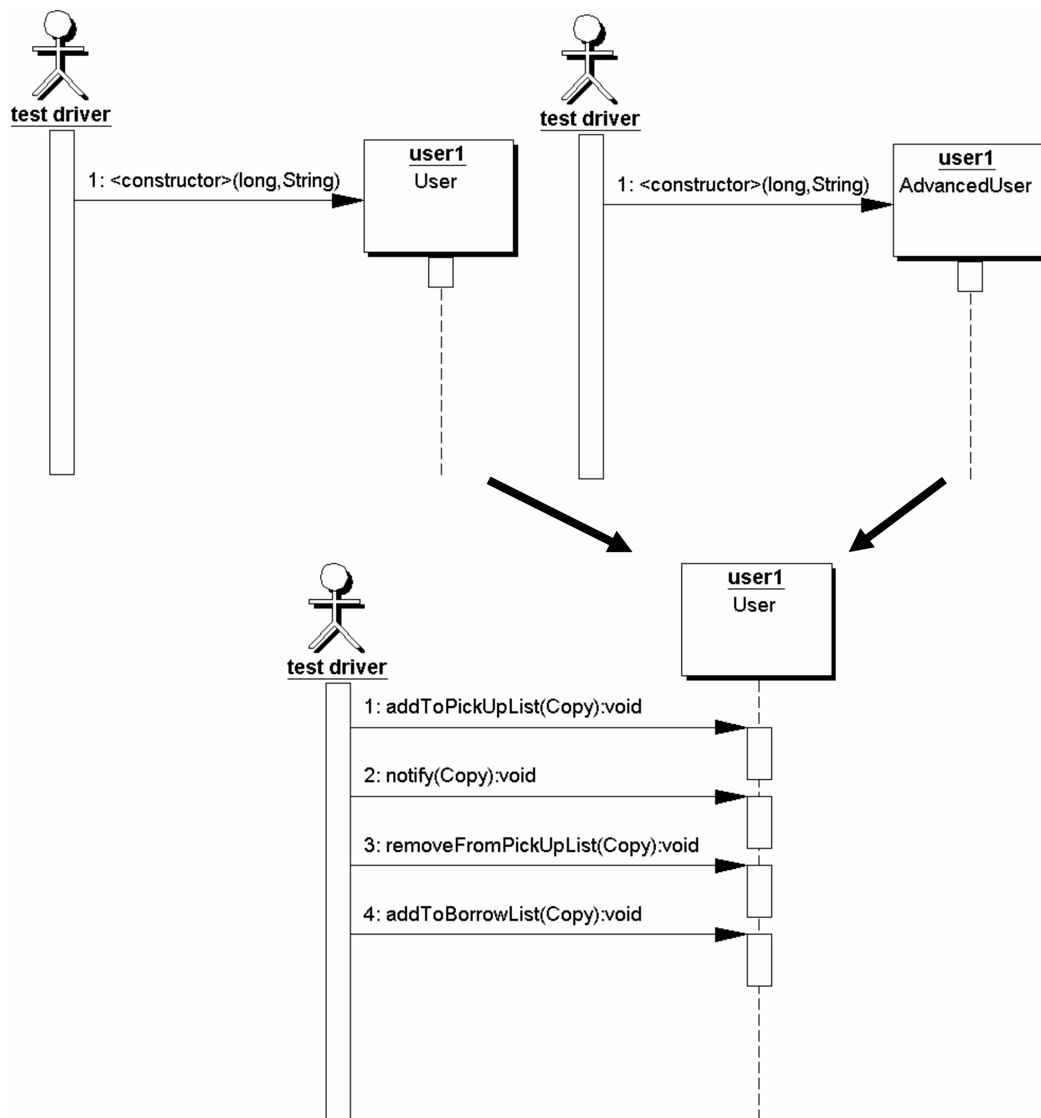

Abbildung 10: Wiederverwendung eines Testfalls für eine Erbenklasse

Eine tatsächliche Einschränkung des Testens auf der Basis von Sequenzdiagrammen besteht darin, dass dabei vorausgesetzt wird, dass verschiedene Objekte nur mittels Methodenaufrufen kommunizieren. Obwohl ein von der Theorie her guter Programmierstil diese Annahme rechtfertigt, so ist es in der Realität doch bei fast allen objektorientierten Programmiersprachen zumindest möglich, auf Attributwerte eines Objektes direkt (ohne Methodenaufruf) zuzugreifen. Dies ließe sich dann jedoch nicht in einem testbaren Sequenzdiagramm modellieren.

Abgesehen davon gilt noch zu beachten, dass – genauso wie (UML) Sequenzdiagramme nicht die ideale Notation für jeden Designaspekt in der Softwareentwicklung sind – Sequenzdiagramme prinzipiell auch nicht die ideale Notation für jeden Aspekt der Testspezifikation sind.

Im Bereich der Anwendungsentwicklung mit Java gilt dies beispielsweise für das Testen von graphischen Benutzungsoberflächen (GUIs). Auf der einen Seite bestehen GUIs üblicherweise aus Dutzenden von Objekten, die in die Eingabesequenzdiagramme aufgenommen werden müssten, was einen hohen Aufwand für das Erstellen dieser Diagramme mit sich bringen würde. Auf der anderen Seite nutzen in Java implementierte GUIs meist eine Vielzahl von anonymen oder inneren Klassen, die in einem Sequenzdiagramm nicht dargestellt werden können (bzw. nur sehr umständlich). Aber auch selbst wenn man diese Probleme befriedigend lösen könnte, wäre es doch immer noch sehr fraglich, ob nicht z.B. Capture & Replay Werkzeuge trotzdem erheblich geeigneter für das Testen von GUIs wären[32].

Ebenfalls weniger geeignet sind Sequenzdiagramme für das Darstellen von mehreren Threads, die sich überlappende Methodenaufrufe beinhalten. Allerdings ist das explizite Verwenden von mehreren Threads und der damit verbundene Wunsch, die parallele Ausführung dieser Threads zu testen, bei den hier betrachteten Geschäftsanwendungen eher selten.

Durchaus spannend sind die Änderungen, die sich im Bereich Testen durch die neue Version 2.0 der UML [OMG03] ergeben werden bzw. durch das auf diese Version aufbauende, kurz vor der Fertigstellung stehende UML Testing Profile [U2TP03]. Das UML Testing Profile definiert Konzepte für

- Testverhalten, d.h. Aktivitäten und Beobachtungen während des Testens,
- Testarchitektur, d.h. die Elemente und deren Beziehungen, die in einen Test involviert sind,
- Testdaten, d.h. die Struktur und Bedeutung von Werten, die in einem Test verarbeitet werden und
- Zeit, d.h. temporale Anforderungen und Beobachtung der Zeit während des Tests.

Diese Konzepte erhöhen die Mächtigkeit der UML für den Bereich Testen ganz erheblich. Allerdings gibt es derzeit noch keine Werkzeuge, die das UML Testing Profile implementieren, so dass über die Akzeptanz und auch die Nutzbarkeit des Profils noch keine fundierten Aussagen getroffen werden können.

In diesem Zusammenhang erwähnenswert ist die Tatsache, dass zwei Mappings vom UML Test Profile auf JUnit und auf TTCN [ETSI03] definiert wurden. Diese Mappings sind zwar eingeschränkt, da nicht alle Bestandteile des UML Testing Profiles auf JUnit bzw. TTCN abgebildet

[32] Allerdings weist auch die Verwendung von Capture & Replay Werkzeugen in solchen Szenarien eine Reihe von Fallstricken auf, so dass sie nicht in jedem Fall eine praktikable bzw. effektive Lösung darstellt [Kaner97].

werden können. Immerhin ermöglichen sie jedoch zumindest theoretisch das Zurückgreifen auf bestehende Werkzeuge, so dass der bei der Einführung des UML Testing Profiles entstehende Aufwand etwas geringer gehalten werden kann.

4 SeDiTeC

Bei SeDiTeC handelt es sich um das im Rahmen dieser Dissertation entwickelte Testwerkzeug. Ziel war es dabei, mittels einer prototypischen Entwicklung die generelle Verwendbarkeit von Sequenzdiagrammen als Testbasis besser beurteilen zu können und die Anforderungen, die an ein solches Werkzeug gestellt werden, zu überprüfen und zu verfeinern. Besonders hohe Bedeutung wurde dabei der Frage beigemessen, wie ein solches Werkzeug möglichst frühzeitiges Testen unterstützen kann.

Als Zielanwendergruppe für SeDiTeC wurden explizit die Software-entwickler ins Auge gefasst, u.a. weil dies die notwendige enge Integration des Testens in den Softwareentwicklungsprozess stark vereinfacht und diese Zielgruppe am ehesten von frühzeitiger Testunterstützung profitieren kann. Ebenfalls spricht für die Fokussierung auf diese Zielgruppe die Tatsache, dass es kaum in der Praxis verbreitete Werkzeuge gibt (abgesehen von Testframeworks wie JUnit), die die Entwickler bei den ihnen obliegenden Testaktivitäten unterstützen.

Um dieser Zielsetzung gerecht zu werden, galt es aufgrund der begrenzten Entwicklerressourcen, die für dieses Projekt zur Verfügung standen, zwei konkurrierende Faktoren gegeneinander abzuwägen. Auf der einen Seite musste möglichst die gesamte in Abschnitt 4.1 beschriebene Funktionalität implementiert werden, um nicht nur die einzelnen Funktionalitäten, sondern auch deren Zusammenspiel beim Testen von Software zu beurteilen. Auf der anderen Seite musste aber auch ein gewisses Augenmerk auf die einfache und effiziente Benutzbarkeit gelegt werden, da sonst eine objektive Beurteilung der Funktionalität von vornherein entweder unmöglich oder stark verfälscht gewesen wäre.

4.1 Funktionalität

Im Folgenden wird beschrieben, welche Funktionalität für SeDiTeC neben der offensichtlichen, nämlich Eingabesequenzdiagramme ausführen zu können, noch ins Auge gefasst wurde.

4.1.1 Testfallspezifikation

Der Benutzer muss in der Lage sein, Eingabesequenzdiagramme (konkrete Testfälle) zu spezifizieren. Hierzu muss es im Einzelnen möglich sein,

- Sequenzdiagramme zu erstellen,
- beliebig viele Testdatensätze für jedes erstellte Sequenzdiagramm zu spezifizieren (ohne dabei das Sequenzdiagramm immer wieder neu erstellen zu müssen),
- Sequenzdiagramme zu testbaren Sequenzdiagrammen zu kombinieren,

- Testdatensätze einzelner Sequenzdiagramme zu kombinieren, um Testdatensätze für (kombinierte) testbare Sequenzdiagramme zu bilden.

Da hierfür umfangreiche komplexe Daten zu verarbeiten sind, soll diese Funktionalität aus Gründen der Benutzbarkeit möglichst mittels einer durchgehend graphischen Benutzungsoberfläche zur Verfügung gestellt werden.

Außerdem muss es möglich sein, diese Informationen zur späteren (Wieder-) Verwendung abzuspeichern.

4.1.2 Testvorbereitung

Entsprechend den Ausführungen in Abschnitt 3.9 soll SeDiTeC in der Lage sein, voll funktionsfähige Teststubs für Klassen zu generieren, deren Verhalten in den Eingabesequenzdiagrammen beschrieben ist, womit automatisiertes Testen bereits von Beginn der Implementierungsaktivitäten an möglich ist, wie es eingangs des Kapitels gefordert wurde. Dies soll automatisch geschehen, so dass der Benutzer lediglich auswählen muss, welche Klassen durch entsprechende Teststubs zu ersetzen sind.

Außerdem erscheint es sinnvoll, dem Benutzer die Möglichkeit zu geben auszuwählen, welche Klassen für den Test relevant sind, um die entstehenden Testergebnisse in Form von beobachteten Sequenzdiagrammen (s. Abschnitt 4.2.10) möglichst übersichtlich und aussagekräftig zu halten.

4.1.3 Testausführung

Der Benutzer soll in der Lage sein, per Knopfdruck beliebig viele Eingabesequenzdiagramme innerhalb eines Testlaufs ausführen zu lassen.

4.1.4 Testauswertung

Für jedes Eingabesequenzdiagramm, das während eines Testlaufs ausgeführt wurde, muss ein entsprechendes beobachtetes Sequenzdiagramm erzeugt werden. Informationen über eventuelle Abweichungen zwischen dem Eingabesequenzdiagramm und dem zugehörigen beobachteten Sequenzdiagramm sollen angezeigt werden. Hierbei muss zu erkennen sein, worin genau die Abweichung besteht.

Ebenfalls ist eine Gesamtübersicht zu erstellen, die erkennen lässt, wie viele Testfälle ausgeführt wurden und wie viele davon erfolgreich verlaufen sind. Alle diese Informationen sollen gespeichert werden, um auch zu einem späteren Zeitpunkt abrufbar zu sein.

4.2 Implementierung

Dieser Abschnitt gibt einen Überblick darüber, wie und unter Verwendung welcher Techniken und Technologien SeDiTeC implementiert wurde.

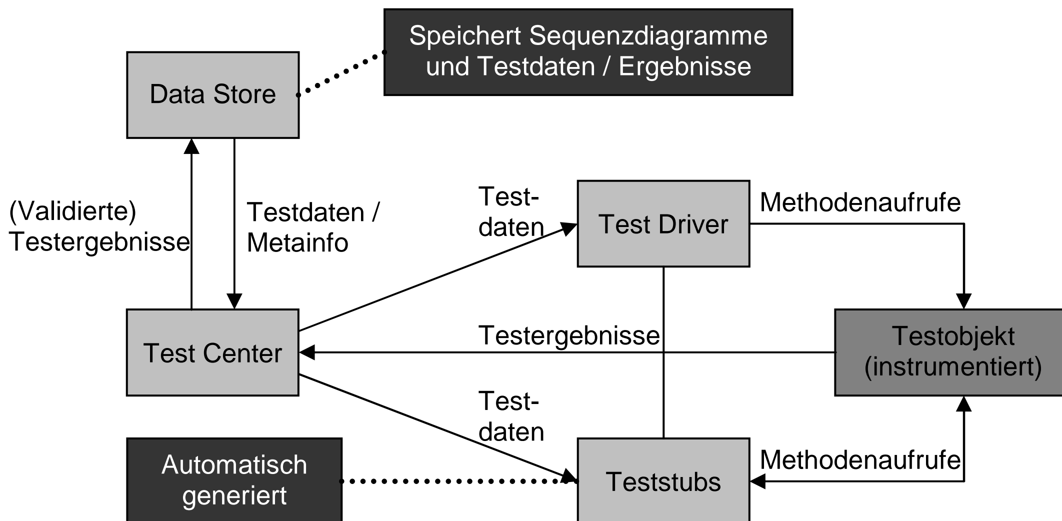

Abbildung 11: Kernkomponenten von SeDiTeC

4.2.1 Kernkomponenten

Abbildung 11 zeigt die für die eigentliche Testausführung zuständigen Kernkomponenten von SeDiTeC:

- Die **Test Driver** Komponente übernimmt die Rolle des Testtreiber-Aktors (s. Abschnitt 3.3) im Eingabesequenzdiagramm und führt die von diesem ausgehenden Methodenaufrufe aus.
- Die **Data Store** Komponente stellt die in den Eingabesequenzdiagrammen enthaltenen Informationen zur Verfügung und ist für die Speicherung der Testergebnisse verantwortlich.
- Die **Test Center** Komponente verwaltet die Testdaten des jeweils gerade aktiven Testfalls, stellt der Test Driver Komponente und den Teststubs die notwendigen Informationen über aufzurufende Methoden und Rückgabewerte zur Verfügung und leitet die Testergebnisse an die Data Store Komponente weiter. Außerdem verwaltet sie eine Liste mit allen am aktiven Testfall beteiligten Objekten.
- Die **Teststubs** werden automatisch vor einem Testlauf generiert und ersetzen noch nicht implementierte oder noch nicht voll funktionsfähige Klassen. Ein von SeDiTeC generierter Teststub verhält sich zur Laufzeit exakt so, wie es im jeweils aktiven Eingabesequenzdiagramm beschrieben ist (s. Abschnitt 4.2.5). In einigen Fällen kann es sinnvoll sein, bereits implementierte Klassen durch Stubs zu ersetzen, um unerwünschte Seiteneffekte zu vermeiden bzw. die Fehlersuche zu vereinfachen.

4.2.2 Instrumentierung vs. Java Debug Interface

Während eines Testlaufs muss SeDiTeC beobachten, welche Objekte welche Methoden auf welchen Objekten aufrufen, welche Parameter und Rückgabewerte dabei übergeben werden und welche Exceptions geworfen werden. Um an diese Informationen zu gelangen, wurden zwei Alternativen in Betracht gezogen, das Java Debug Interface (JDI) und die Instrumentierung des Sourcecodes.

Beim JDI (s. [Sun03b]) handelt es sich um eine Java Programmierschnittstelle, die von Sun als Teil der Java Platform Debugging Architecture (s. [Sun03c][33]) zur Verfügung gestellt wird, mit deren Hilfe man in der Lage ist, Applikationen zu beobachten, bzw. Debugging Funktionalität zu implementieren. Beim Verwenden des JDI muss zu Beginn eine Verbindung zu einer Java Virtual Machine (VM) aufgebaut werden. Dabei kann es sich um dieselbe VM handeln, in der das JDI gerade verwendet wird, um eine weitere VM, die auf demselben Rechner läuft oder um eine VM, die auf einem über das Netzwerk erreichbaren Rechner läuft. Ist die Verbindung zu der zu beobachtenden VM aufgebaut, so werden Ereignisse (z.B. beim Betreten und Verlassen von Methoden) von der beobachteten VM an die beobachtende VM gesendet. Ebenfalls lassen sich in der beobachteten VM Breakpoints setzen, Threads anhalten und wieder starten etc.

Der Vorteil des JDI liegt darin, dass seine Verwendung keinerlei Anforderungen an die zu testende Anwendung stellt (abgesehen davon, dass die Anwendung auf einer VM laufen muss, die JPDA kompatibel ist, s.o.). Der Sourcecode der zu testenden Anwendung ist nicht erforderlich.

Trotzdem eignet sich das JDI aus zwei Gründen nicht für die Verwendung im Rahmen von SeDiTeC. Zum einen ist es unmöglich, auf die konkreten Rückgabewerte von Methodenaufrufen zuzugreifen (die Abfrage der Parameter stellt jedoch kein Problem dar). Zum anderen gibt es massive Performanzprobleme, die dazu führen, dass sich die Laufzeit von manchen Tests um den Faktor 100 und mehr erhöht. Da es sich bei SeDiTeC um eine Anwendung handelt, deren Benutzbarkeit nicht zuletzt davon abhängt, dass der Anwender eine schnelle Rückmeldung bezüglich der Testergebnisse erhält, ist ein solches Laufzeitverhalten auch im Rahmen einer prototypischen Implementierung inakzeptabel.

Die zweite in Betracht gezogene Alternative, um Applikationen zu beobachten, nämlich die Instrumentierung des Sourcecodes, genügt allen von SeDiTeC gestellten Anforderungen bezüglich der „Beobachtungsfunktionalität“. Auch das Laufzeitverhalten der beobachteten Applikation wird bei relevanten Tests nicht merklich durch die Instrumentierung verändert, so dass diese Alternative in SeDiTeC implementiert wurde.

---

[33] Die Java Platform Debugging Architecture ist im Java Development Kit seit der Version 1.3.1 standardmäßig enthalten und ist damit auf Rechnern mit einer neueren Java Installation verfügbar. Für ältere Java Versionen lässt sie sich getrennt nachinstallieren.

Jedoch hat auch das Instrumentieren von Sourcecode zwei Nachteile. Zum einen muss der Sourcecode verfügbar sein (was kein echtes Problem ist, da es sich bei den Anwendern normalerweise um die Entwickler der zu testenden Applikation handelt) und vor einem Testlauf mindestens einmal in instrumentierter Form neu kompiliert werden. Zum anderen verhindert oder verfälscht die Instrumentierung des Sourcecodes die Ergebnisse von anderen Werkzeugen, die auf Instrumentierung beruhen, wie zum Beispiel Werkzeuge zum Messen von Codeabdeckung.

4.2.3 Anbindung an Together Control Center

Für die Testspezifikation ist es notwendig, dem Benutzer eine Möglichkeit an die Hand zu geben, Sequenzdiagramme zu erstellen. Da die Aufgabe, eine grafische Benutzungsoberfläche zu implementieren, die das Erstellen von UML konformen Sequenzdiagrammen ermöglicht, in etwa genauso mühselig wie im Kontext dieser Arbeit unspannend ist, wurden verschiedene CASE Tools auf ihre grundsätzliche Eignung für diese Aufgabe und auf ihre Erweiterbarkeit hin überprüft.

In die nähere Auswahl kamen hierbei Together Control Center (kurz: Together) von Togethersoft und Rational Rose von Rational Software. Da zum Zeitpunkt der Entscheidung (Ende 1999) Together sowohl über die deutlich konsequentere Umsetzung der UML bezüglich Sequenzdiagrammen, als auch über eine erheblich mächtigere Programmierschnittstelle verfügte, fiel die Wahl auf dieses CASE Tool. Eine ausführlichere Gegenüberstellung der beiden Werkzeuge im Problemkontext findet sich in [Fraikin+00].

Together vereinigt in sich sowohl die Funktionalität eines UML Modellierungswerkzeugs (Erstellung von UML Diagrammen, Generierung von Sourcecode auf Basis dieser Diagramme etc.) als auch einer Entwicklungsumgebung (Bearbeitung, Verwaltung und Übersetzung von Sourcecode, GUI-Builder, Debugging etc.). Die Programmierschnittstelle, die externen Entwicklern zum Zweck der Erweiterung der Funktionalität von Together zur Verfügung gestellt wird (Together OpenAPI), gliedert sich dabei in drei separate Schnittstellen (IDE, RWI und SCI), die unterschiedliche Zugriffsmöglichkeiten bieten. Während die IDE-Schnittstelle nur lesenden Zugriff auf die Oberfläche von Together ermöglicht, um beispielsweise Diagramme zu öffnen, ein Projekt zu speichern oder eine Meldung auszugeben, dient die RWI-Schnittstelle (Read Write Interface) der Programmierung der Strukturen, die die einzelnen Elemente eines Together-Modells ausmachen. Dabei kann es sich um Diagramme, Klassen und Attribute in einem Klassendiagramm, Methodenaufrufe in einem Sequenzdiagramm etc. handeln. Die SCI-Schnittstelle (Source Code Interface) schließlich ermöglicht es, komfortabel direkt auf den Sourcecode zuzugreifen und diesen gegebenenfalls zu verändern.

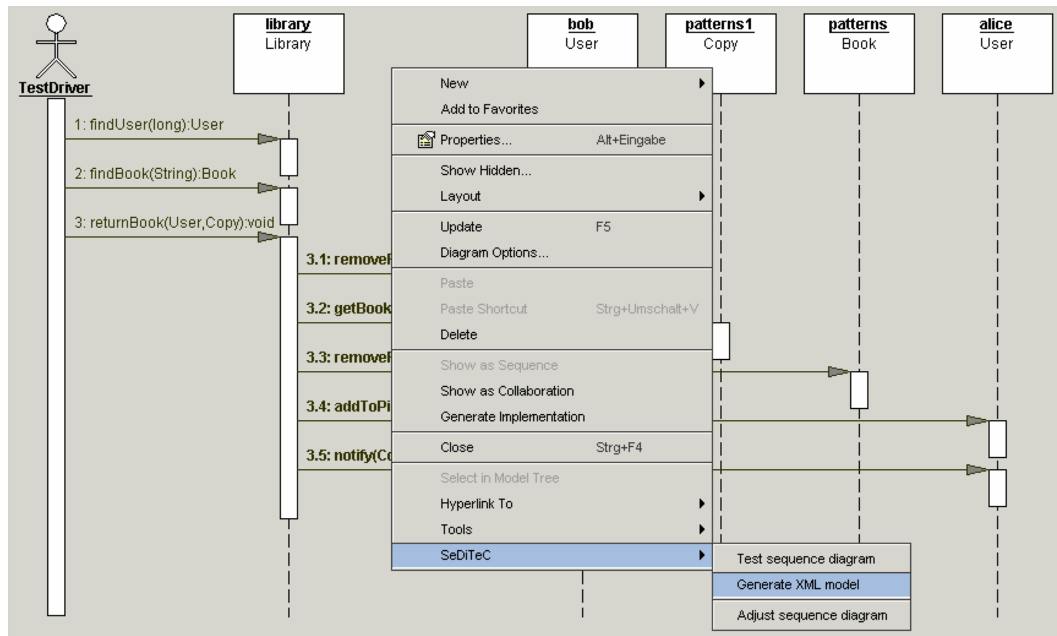

Abbildung 12: Einbindung der SeDiTeC Module in die Together GUI

Die flexibelste Möglichkeit, Together um zusätzliche Funktionalität zu erweitern, besteht darin, so genannte Module zu programmieren. Bei diesen Modulen handelt es sich um normale Java-Klassen, welche mindestens eines der beiden in der Together OpenAPI enthaltenen Interfaces `IdeScript` und `IdeStartup` implementieren müssen. Ein Modul, welches das Interface `IdeScript` implementiert, kann zu einem beliebigen Zeitpunkt gestartet werden, während man mit Together arbeitet. Implementiert ein Modul das Interface `IdeStartup`, so wird es beim Starten von Together einmal automatisch ausgeführt. Die Module lassen sich dabei flexibel in die Menüstruktur von Together integrieren und zwar sowohl in Kontextmenüs als auch in Menüleisten (s. Abbildung 12). Einmal gestartet können Module dann mittels der drei oben beschriebenen Schnittstellen der Together OpenAPI auf Modelldaten von Together-Projekten, den zugehörigen Sourcecode usw. zugreifen und ggf. Änderungen durchführen.

Die im Rahmen der Implementierung von SeDiTeC entwickelten Together Module lassen sich mittels Kontextmenüs auf Sequenzdiagramme (Modul *Test Sequence Diagram,* s. Abschnitt 4.2.4), Klassen (Modul *Generate method stubs,* s. Abschnitt 4.2.5 und Modul *Instrument methods,* s. Abschnitt 4.2.6) und das gesamte Together-Modell (Modul *Generate XML model*, s. Abschnitt 4.2.7) anwenden.

Abgesehen von diesen primäre Funktionalität implementierenden Modulen gibt es weitere Module, die nicht-essentielle Komfortfunktionen bieten, wie zum Beispiel das automatische Layouten von Sequenzdiagrammen oder das Starten der SeDiTeC Benutzungsoberfläche aus Together heraus.

4.2.4 SeDiTeCs Testfunktionalität innerhalb von Together

Die in Abschnitt 4.1.1 erwähnten Anforderungen bzgl. des Kombinierens von Sequenzdiagrammen und der Spezifikation mehrerer Testfälle für ein Sequenzdiagramm realisiert SeDiTeC innerhalb einer eigenen Benutzungsoberfläche, so dass diese Funktionalität nicht direkt in Together verfügbar ist. Togethers diesbezüglich inhärente Funktionalität beschränkt sich auf das Erstellen einzelner Sequenzdiagramme und maximal eines Testfalls pro Sequenzdiagramm (s. Abschnitt 3.4).

Um jedoch auch ein möglichst zügiges Ad-hoc-Testen zu ermöglichen, ohne die SeDiTeC Oberfläche starten und dort Testfälle zusammenstellen zu müssen, kann man mittels des Moduls *Test Sequence Diagram* direkt aus Together heraus ein einzelnes Sequenzdiagramm unter Verwendung der darin spezifizierten Testdaten ausführen. Solche einzelnen Testfälle, bei denen die Testdaten innerhalb des Together-Modells spezifiziert sind, sind dann jedoch nicht Teil der automatisch mittels SeDiTeC ausführbaren Regressionstests. Diese Vorgehensweise eignet sich daher ausschließlich für Wegwerftestfälle bzw. Testfälle, für die (noch) nicht feststeht, ob sie Bestandteil der Regressionstests werden sollen.

4.2.5 Teststubs

SeDiTeC implementiert ein flexibles Teststubkonzept, das es ermöglicht, Klassen durch Stubs zu ersetzen, die sich dann zur Laufzeit gemäß beliebiger Eingabesequenzdiagramme verhalten und sich bezüglich der in einem Eingabesequenzdiagramm enthaltenen „Verhaltensinformationen“ praktisch nicht von einer tatsächlich implementierten Klasse unterscheiden lassen. Um eine Klasse durch einen Stub ersetzen zu können, benötigt SeDiTeC lediglich die Klassendeklaration einschließlich der relevanten Methodensignaturen.

Abbildung 13 zeigt einen SeDiTeC Teststub[34], wie er für eine Methode generiert wird, die einen Parameter vom Typ `long` erhält, ein Objekt vom Typ `User` zurückgibt und eine Exception vom Typ `UserNotFoundException` werfen kann. Eine ersetzte Methode in einem solchen Teststub besteht aus bis zu sechs Sektionen, die die folgenden Aufgaben wahrnehmen:

- Initialisierung
- Logging des Methodenaufrufbeginns an sich sowie der eventuell erhaltenen Parameter
- Überprüfung der Vorbedingung (falls vorhanden)

[34] Der hier gezeigt Teststubcode weicht aus Gründen der Lesbarkeit vom tatsächlich generierten Code ab (die Funktionalität ist jedoch identisch). Insbesondere die Variablennamen sind zur Vermeidung von Namenskonflikten in der Realität erheblich kryptischer.

```
1 public User findUser(long uid) throws
                               UserNotFoundException{
2   // initialization
3   String uniqueMethodName =
"<oiref:java#Member#lib.Library#findUser#(#long#)#:oiref>";
4   TestCenter tc = TestCenter.getTestCenter();
5   // check call order and parameters
6   Object params = new Object[1];
7   params[0] = tc.convertPrimitiveType(uid);
8   MethodCallDataWrapper mcdWrapper =
      tc.acceptStubbedCall(this, uniqueMethodName, params);
9   MethodCallData mcd = mcdWrapper.getMethodCallData();
10  // precondition
11  if (!(uid != 0)))
12  {
13    throw new PreconditionError("Condition: uid != 0");
14  }
15  // execute method calls
16  executeMethodCalls(mcd);
17
18  // check for Throwable to throw
19  if (mcd.isThrowing())
20  {
21    String throwableName = mcd.getThrowableName();
22    Throwable t = createThrowable(throwableName);
23    tc.acceptThrowable(mcdWrapper, t);
24    if (t instanceof Error)
25    {
26      throw(Error)t;
27    }
28      else if (t instanceof lib.UserNotFoundException)
29      {
30        throw(lib.UserNotFoundException)t;
31      }
32    else
33    {
34      throw(RuntimeException)t;
35    }
36  }
37
38  // get return value
39  Object result = mcd.getReturnVariable().getAsObject();
40  tc.acceptStubbedReturn(this, mcdWrapper, result);
41
42  // postcondition
43  if (!(result != null))
44  {
45    throw new PostconditionError(
                       "Condition: RESULT != null");
46  }
47  return (lib.User) result;
48}
```

Abbildung 13: Beispiel eines SeDiTeC Teststubs

- Initiierung einer beliebigen Menge weiterer Methodenaufrufe
- Eventuell Erstellen und Werfen einer Exception

Logging des Methodenaufrufendes sowie eventuell Rückgabe eines Rückgabewertes und Überprüfung der Nachbedingung (falls vorhanden)

Die Initialisierungssektion sorgt dafür, dass sich die Methode mittels einer systemweit eindeutigen Zeichenkette identifizieren kann (Zeile 3) und fragt die Referenz des Singletons `TestCenter` ab, das die Schnittstelle für Anwendungsklassen zu SeDiTeC bildet (Zeile 4).

Die nächste Sektion verpackt zuerst alle erhaltenen Parameter in ein `Object` Array (wobei Werte von Basistypen in Instanzen der zugehörigen Wrapperklassen[35] verwandelt werden) und meldet diese dann an SeDiTeC (Zeilen 6-9). Falls SeDiTeC die sich meldende Stubmethode einem Methodenaufruf im zugrunde liegenden Eingabesequenzdiagramm zuordnen kann, wird ein Objekt vom Typ `MethodCallDataWrapper` bzw. in diesem ein Objekt vom Typ `MethodCallData` zurückgeliefert, welches die notwendigen Informationen über das von der Stubmethode für diesen Aufruf innerhalb dieses Testfalls erwartete Verhalten enthält.

Falls, wie in diesem Fall, im Klassenmodell für die Methode eine Vorbedingung angegeben ist, folgt nun Code zum Überprüfen dieser Bedingung (s. Abschnitt 4.4).

Die vierte Sektion besteht lediglich aus einem Aufruf der für jeden Teststub generierten Methode `executeMethodCalls` (s. Anhang A), die anhand der im übergebenen `MethodCallData` Objekt enthaltenen Informationen eine beliebige Menge von weiteren Methodenaufrufen initiiert. In dieser Fähigkeit, das komplette Verhalten von Objekten zu simulieren, besteht ein wesentlicher Vorteil der SeDiTeC Teststubs gegenüber anderen Teststub-Konzepten, die diese Funktionalität nicht bieten, da sich dadurch eine Reihe von Sachzwängen vermeiden lassen, denen das Testen sonst unterworfen ist. Hierzu gehört zum Beispiel die Notwendigkeit, Anwendungsklassen für das Testen in einer bestimmten Reihenfolge zu implementieren bzw. zu integrieren.

Die zweite von SeDiTeC generierte Methode `createThrowable`, die dafür zuständig ist, eventuell zu werfende `Exception` oder `Error` Objekte zu erstellen, wird in der fünften Sektion verwendet, falls dieser Methodenaufruf mit dem Werfen eines `Throwable` Objektes beendet werden soll.

---

[35] Java unterscheidet zwischen „echten" Objekten (Instanzen von Erbenklassen der Klasse `Object`) und Werten von Basistypen (`int`, `long`, `char` etc.). Möchte man letztere als Objekte ansprechen, so muss man sie erst in Instanzen so genannter Wrapperklassen (`Integer`, `Long`, `Character` etc.) „einwickeln", was das Programmieren mit Java in einigen speziellen Situationen verhältnismäßig umständlich macht.

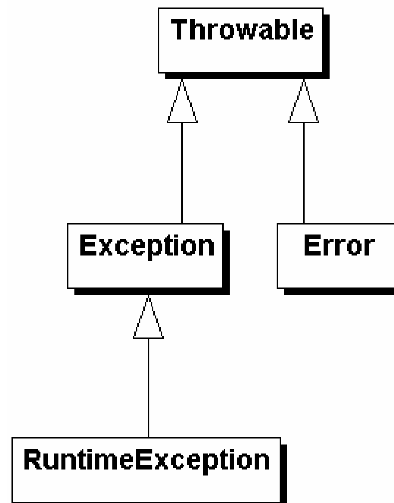

Abbildung 14: Klassenhierarchie der Ausnahmebehandlung in Java

Das Generieren von Code, der diese Aufgabe löst (Zeilen 19-36), ist aufgrund des „Checked Exception"-Konzepts von Java nicht ganz unproblematisch.

Java unterscheidet bei seinem Ausnahmebehandlungsmechanismus in so genannte checked (geprüfte) und unchecked (ungeprüfte) Exceptions (Ausnahmen), wobei hier streng genommen der Begriff „Exception" etwas irreführend ist, da in Java nicht nur Instanzen der Klasse `Exception` und deren Erben geworfen werden können (s. Abbildung 14), sondern alle Instanzen der Klasse `Throwable` und deren Erben, zu denen insbesondere auch die Klasse `Error` zählt. Besteht die Möglichkeit, dass im Körper einer Methode (das schließt von dieser Methode wiederum aufgerufene Methoden mit ein) eine geprüfte Ausnahme geworfen wird, so gibt es zwei Alternativen. Entweder muss die Methode diese Ausnahme selbst behandeln (fangen) oder sie muss die Ausnahme in der so genannten `throws` Klausel deklarieren und damit ihrerseits Klienten verdeutlichen, dass sie u.U. diese Ausnahme wirft. Die Klassen `RuntimeException` und `Error` sowie alle ihre Erbenklassen werden von Java als ungeprüfte Ausnahmen definiert. Diese Definition ist fester Bestandteil der Programmiersprache Java und lässt sich in eigenen Java Programmen weder ausschalten noch anpassen.

Eine Java Methode, die eine eventuell geworfene geprüfte Ausnahme weder selbst fängt noch in der `throws` Klausel deklariert, wird vom Java Compiler nicht übersetzt. Damit soll sichergestellt werden, dass Programmierer ein Mindestmaß an Ausnahmebehandlung in ihren Programmen implementieren müssen. Die Wirksamkeit bzw. der Nutzen dieser Maßnahme ist allerdings umstritten [Eckel03].

Für die zu generierende Teststubmethode bedeutet dies, dass man für eine zu generierende Methode anhand deren `throws` Klausel eine Fallunterscheidung inklusive entsprechender Typumwandlungen treffen

muss (vgl. Abbildung 13). Der simple Ansatz, sinngemäß einfach eine Anweisung wie

```
if (method.isThrowing()) throw (Throwable) t;
```

in den Teststubmethoden zu verwenden, wäre ungeschickt, da man damit die Methoden, die einen Teststub aufrufen, pauschal dazu zwingen würde, Ausnahmen vom Typ `Throwable` zu fangen. Eine zu testende Klasse, die einen Teststub aufruft, müsste dann speziell für den Test angepasst bzw. erweitert werden und das auch noch mit „unerwünschter“ Funktionalität, was inakzeptabel ist.

Die letzte Sektion der Teststubmethode schließlich ist für das Beenden der Methode zuständig (falls sie nicht bereits in der vorigen Sektion durch das Werfen eines `Throwable` Objektes verlassen wurde) und gibt gegebenenfalls nach einer Typkonvertierung den vorgesehenen Rückgabewert an die aufrufende Methode zurück.

Ist im Klassenmodell eine Nachbedingung angegeben, so enthält die für die Beendigung der Methode zuständige Sektion auch einen Abschnitt (Zeilen 43-46), der für die Überprüfung dieser Nachbedingung (s. Abschnitt 4.4) zuständig ist.

Um eine Klasse durch einen SeDiTeC Teststub zu ersetzen, muss man sie im entsprechenden Together-Modell auswählen; die Funktionalität ist dann z.B. über das Kontextmenü verfügbar. Alternativ kann man dafür auch ein Ant-Skript verwenden (s. Abschnitt 4.6).

4.2.6 Instrumentierung des Sourcecodes

Soll eine implementierte Klasse mit SeDiTeC getestet werden, so muss sie instrumentiert werden. Durch die Instrumentierung wird den Methoden der zu testenden Klasse Code hinzugefügt, der im Wesentlichen eine Untermenge der Aufgaben erfüllt, die der Code einer Teststubmethode bewältigt (s. Abschnitt 4.2.5). Es sind dies im Einzelnen:

- Initialisierung
- Logging des Methodenaufrufbeginns an sich sowie der eventuell erhaltenen Parameter
- Überprüfung der Vorbedingung (falls vorhanden)
- Logging des Methodenaufrufendes inklusive des zurückgegebenen Rückgabewertes (falls vorhanden) bzw. einer eventuell geworfenen Ausnahme
- Überprüfung der Nachbedingung (falls vorhanden)

Die ersten drei Aufgaben unterscheiden sich in keiner Weise von denen, die auch von einer Teststubmethode erfüllt werden müssen, und werden in exakt gleicher Weise gelöst. Im Gegensatz zu einer Teststubmethode muss sich der Instrumentierungscode einer implementierten Methode nicht um das Aufrufen anderer Methoden oder das Generieren von eventuell zu werfenden Ausnahmen kümmern, da dies durch die

implementierte Methode selbst geschehen sollte (schließlich handelt es sich dabei um das, was man testen möchte). Daher muss lediglich das Beenden der Methode überwacht werden, um diesen Vorgang inklusive eines eventuellen Rückgabewerte oder einer geworfenen Ausnahme an SeDiTeC zu melden sowie eine möglicherweise vorhandene Nachbedingung überprüft werden.

Um eine Klasse mittels SeDiTeC zu instrumentieren, muss man sie im entsprechenden Together-Modell auswählen; die Funktionalität ist dann z.B. über das Kontextmenü verfügbar. Wie auch beim Ersetzen einer Klasse durch Teststubs kann man alternativ für diese Aufgabe ein Ant-Skript verwenden (s. Abschnitt 4.6). Ein Beispiel für eine instrumentierte Methode findet sich in Anhang B.

4.2.7 XML Datenexport

Möchte man nicht nur triviale (Ad-hoc-)Tests durchführen, die jeweils nur auf einzelnen Sequenzdiagrammen und einzelnen Testfällen beruhen können (s. Abschnitt 4.2.4), so sind hierfür die relevanten Informationen des Together-Modells aus Together zu exportieren. Die hierbei entstehende XML-Datei besteht dabei im Wesentlichen aus zwei Abschnitten. Im ersten Abschnitt werden Informationen über alle im Together-Modell enthaltenen Sequenzdiagramme gespeichert. Zu diesen Informationen gehören der Name des Sequenzdiagramms, die beteiligten Instanzen (inklusive Name und Typ) sowie die im Sequenzdiagramm vorkommenden Methodenaufrufe.

Im zweiten Abschnitt sind Informationen über alle Methoden gespeichert, die in mindestens einem der exportierten Sequenzdiagramme vorkommen. Zu diesen Informationen gehören der Name der Methode, die Typen des Rückgabewertes und der Parameter, der voll qualifizierte Name der Klasse, in der diese Methode enthalten ist, sowie die Modifikatoren der Methode (z.B. `static`).

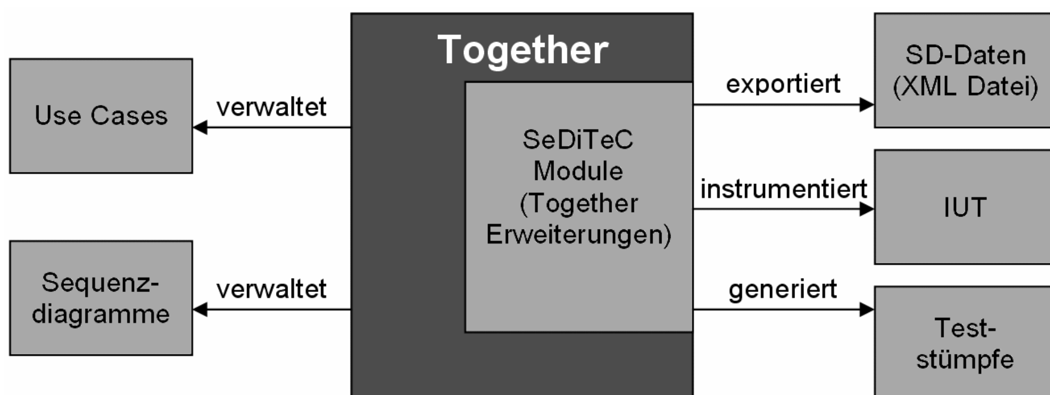

Abbildung 15: Rolle von Together beim Einsatz von SeDiTeC

Abbildung 15 zeigt noch einmal zusammengefasst, welche Rolle Together beim Einsatz von SeDiTeC spielt.

4.2.8 Testfallspezifikation mit SeDiTeC

Sind die für das Testen relevanten Modelldaten aus Together in eine XML-Datei exportiert worden (s. Abschnitt 4.2.7), so können sie vom SeDiTeC-Hauptprogramm eingelesen werden. Mittels der Oberfläche von SeDiTeC lassen sich im Wesentlichen drei Aufgaben erfüllen:

- Erstellen von Testfällen für einzelne Sequenzdiagramme
- Kombinieren von Sequenzdiagrammen
- Kombinieren von Testfällen für einzelne Sequenzdiagramme zu Testfällen von kombinierten Sequenzdiagrammen

Das Erstellen von Testfällen für einzelne Sequenzdiagramme wird auf der Karteikarte „Sequence Diagrams“ (s. Abbildung 16) vorgenommen. Diese Karteikarte umfasst zwei Fenster. Das linke Fenster enthält eine Liste mit allen verfügbaren Sequenzdiagrammen und das rechte Fenster zeigt die für das ausgewählte Sequenzdiagramm vorhandenen Testfälle in

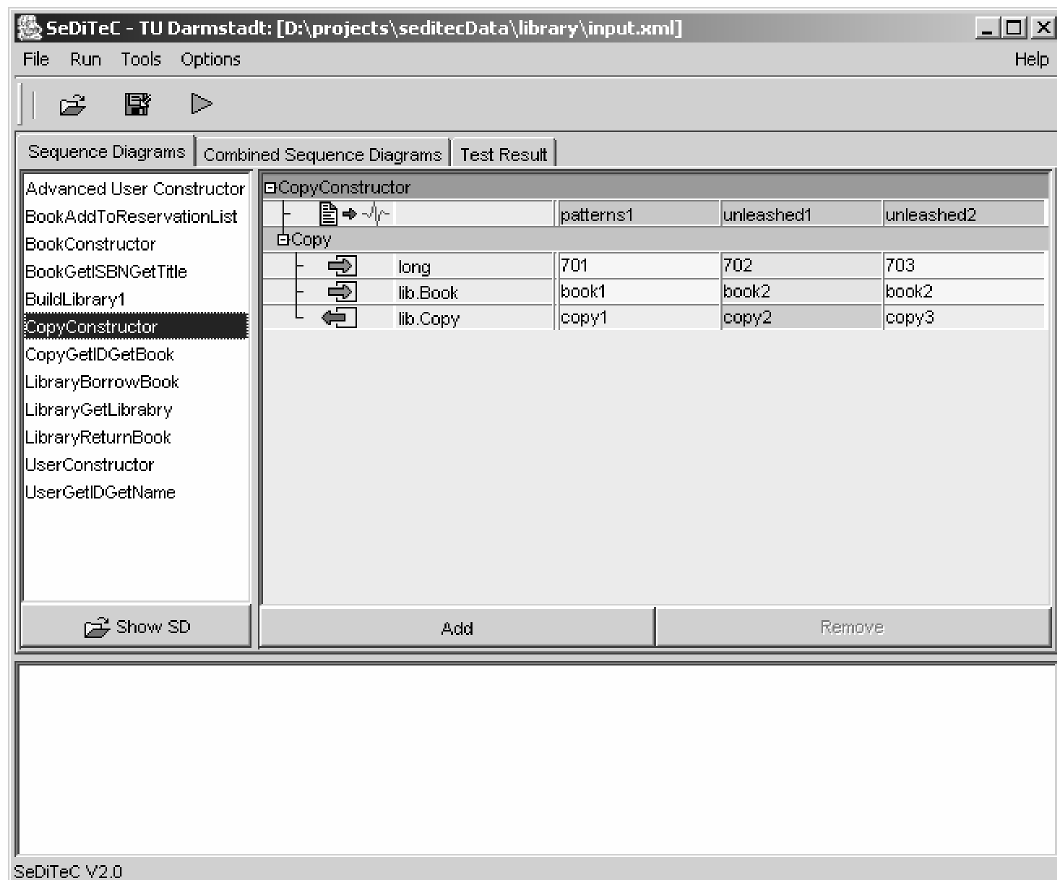

Abbildung 16: Eingabe von Testfällen für einzelne Sequenzdiagramme

tabellarischer Form. Die erste Spalte dieser Tabelle enthält Icons, die angeben, ob es sich bei der jeweiligen Zeile um einen Parameter (Pfeil von links nach rechts) oder einen Rückgabewert (Pfeil von rechts nach links) einer Methode handelt. Die zweite Spalte enthält den Typ des Parameters oder Rückgabewerts. Jede weitere Spalte beschreibt dann einen Testfall für das Sequenzdiagramm, wobei die oberste Zeile die Namen der Testfälle enthält (in diesem Fall „patterns1“, „unleashed1“ und „unleashed2“). Über den Button „Show SD“ kann man ein Bild des Sequenzdiagramms einblenden.

Das Kombinieren von Sequenzdiagrammen sowie das Kombinieren von Testfällen wird auf der Karteikarte „Combined Sequence Diagrams“ durchgeführt (s. Abbildung 17). Das linke obere Fenster der Karteikarte zeigt alle vorhandenen kombinierten Sequenzdiagramme. Das derzeit zu bearbeitende ist markiert („LibraryBorrowBook“). Das linke untere Fenster zeigt an, aus welchen einzelnen Sequenzdiagrammen das gerade ausgewählte kombinierte Sequenzdiagramm besteht („BookConstructor“, „CopyConstructor“ etc.). Der Button „Add/Remove Sequence Diagrams“ öffnet einen Dialog, über den die Zusammensetzung des kombinierten Sequenzdiagramms aus einzelnen Sequenzdiagrammen festgelegt werden kann.

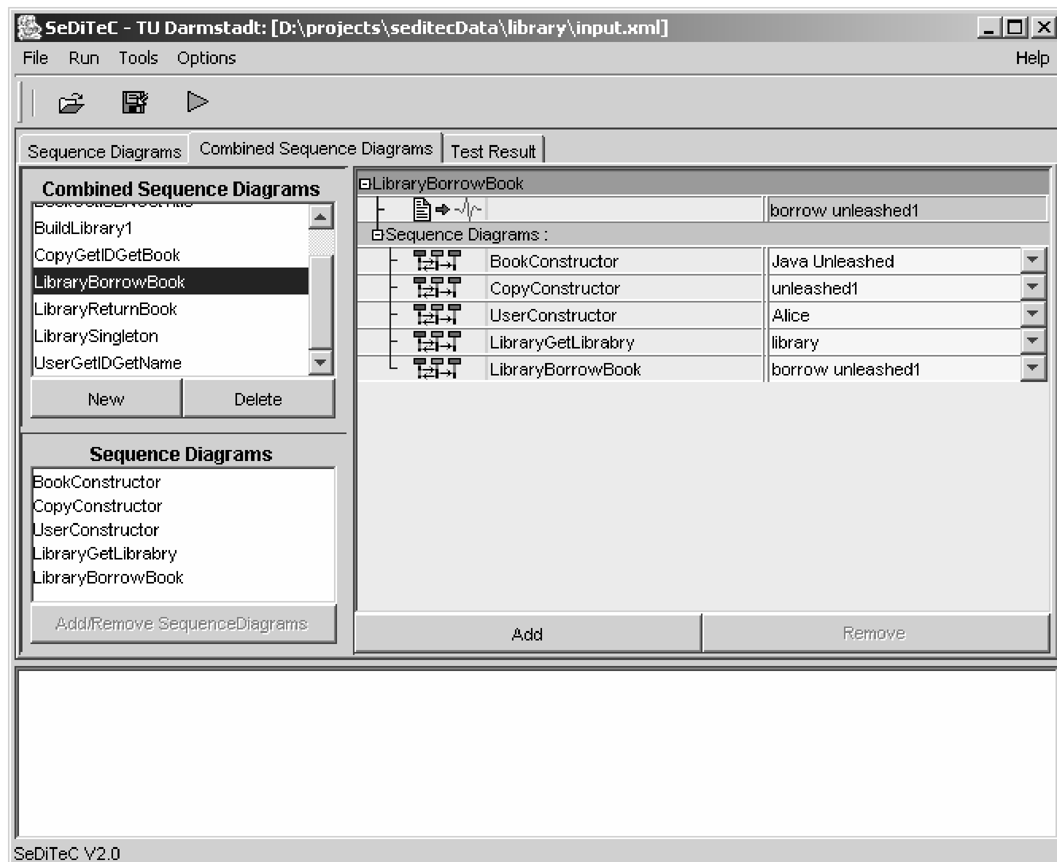

Abbildung 17: Eingabe von Testfällen für kombinierte Sequenzdiagramme

Das rechte Fenster zeigt wiederum die kombinierten Testfälle an. Die erste Zeile enthält dabei die Namen der kombinierten Testfälle. Die erste Spalte (nach den Icons) enthält die Namen der einzelnen Sequenzdiagramme, aus denen das kombinierte Sequenzdiagramm besteht. Um einen kombinierten Testfall zu erstellen, muss für jedes Sequenzdiagramm einer seiner auf der ersten Karteikarte erstellten Testfälle ausgewählt werden. Daher sind die einzelnen Zellen einer Spalte auch nicht frei beschreibbar, sondern sind durch Listboxen implementiert, die die Auswahl eines Testfalls aus den vorhandenen Testfällen des entsprechenden Sequenzdiagramms ermöglichen.

Beim Kombinieren von Sequenzdiagrammen und zugehörigen Testdatensätzen wäre theoretisch eine gewisse Vorauswahl möglich. Beim Kombinieren von Sequenzdiagrammen könnte man z.B. die Liste der wählbaren Sequenzdiagramme auf solche Sequenzdiagramme einschränken, die alle in ihnen enthaltenen Objekte selbst erstellen (durch Konstruktoraufruf), oder – für die Objekte, für die das nicht gegeben ist – diese Objekte in bereits vorher gewählten Sequenzdiagrammen erstellt werden. Da Objekte in typischen Testfällen aber auch durch Parameter und Rückgabewerte der Testdatensätze „erstellt“ werden können (wie in Abschnitt 3.7 beschrieben), wäre eine solche Vorauswahl zu restriktiv.

Dies führt direkt zu der nächsten Frage, ob sich nach erfolgter Kombination der Sequenzdiagramme zumindest die Auswahl der Testdatensätze beim Kombinieren derselbigen einschränken lässt. Zu prüfen wäre in diesem Fall, ob unter Berücksichtigung der Testdatensätze alle Objekte entweder durch Aufruf eines Konstruktors oder durch Auftreten des Objektes als Parameter oder Rückgabewert initialisiert werden, bevor die erste Methode auf ihnen aufgerufen wird. Testdatensätze, die diese Anforderung nicht erfüllen, würden dann aus der Auswahlliste gefiltert. Allerdings muss auch auf diese Art der Vorauswahl verzichtet werden, und zwar zugunsten einer flexiblen Möglichkeit zur Herstellung von Ausgangszuständen, durch die Objekte auch auf eine andere Art und Weise initialisiert werden können (s. Abschnitt 4.3).

4.2.9 Testausführung

Für die Testausführung müssen die relevanten Klassen der zu testenden Anwendung bereits instrumentiert oder durch Teststubs ersetzt worden sein (s. Abschnitte 4.2.5 und 4.2.6). Ebenso müssen andere Vorbereitungen getroffen worden sein, wie zum Beispiel das Setzen des richtigen Klassenpfades oder das Starten eines Datenbank-Managementsystems.

Auf der Oberfläche von SeDiTeC kann nun eine Tests enthaltende XML-Datei geladen werden und per Knopfdruck können dann alle darin vorhandenen kombinierten Testfälle ausgeführt werden. Während der Testausführung verhindert SeDiTeC das Beenden der Ausführung der

Java Virtual Machine durch die Methode System.exit() und fängt sämtliche von der zu testenden Anwendung geworfenen Ausnahmen ab.[36] Dies soll gewährleisten, dass auch falls Testfälle fehlschlagen alle Testfälle ausgeführt werden können und die Testausführung nicht vorzeitig abgebrochen wird.

Alternativ kann man beim Starten von SeDiTeC die XML-Datei, die die Tests enthält, auch als Kommandozeilenparameter angeben und damit die Tests ausführen, ohne die Benutzungsoberfläche zu öffnen. Somit ist SeDiTeC auch für den Batchbetrieb geeignet.

In beiden Fällen erhält der Benutzer während der Testausführung eine einfache Fortschrittsanzeige und nach Abschluss der Tests eine Rückmeldung darüber, bei wie vielen Testfällen Fehler aufgetreten sind.

4.2.10 Testauswertung

Während der Testausführung wird für alle ausgeführten Eingabesequenzdiagramme jeweils ein beobachtetes Sequenzdiagramm generiert, dass das tatsächlich beobachtete Verhalten der IUT beinhaltet (s. Abschnitt 3.8). Die beobachteten Sequenzdiagramme werden mit einem Zeitstempel versehen und wiederum in einer eigenen XML-Datei abgelegt. Zusätzlich wird eine HTML-Seite generiert, in der tabellarisch etwaige Abweichungen aufgeführt werden.

Eine alternative, grafische Aufbereitung der Ergebnisse von SeDiTeC-Testläufen wurde in [Girschick02] implementiert. Unter anderem wird dabei ein Sequenzdiagramm erzeugt, dass durch das „Über-einanderlegen“ des Eingabesequenzdiagramms und des beobachteten Sequenzdiagramms entsteht (s. Abbildung 18). Die grau hinterlegten Objekte und Methodenaufrufe stehen dabei für Übereinstimmungen zwischen den beiden Sequenzdiagrammen, während die schwarz hinterlegten Objekte und Methodenaufrufe nur im Eingabesequenz-diagramm, nicht aber im beobachteten Sequenzdiagramm vorhanden sind.

Weichen Parameter, Rückgabewerte oder Exceptions voneinander ab, so wird dies im Diagramm direkt unterhalb des Methodenaufrufs durch Gegenüberstellen der spezifizierten und beobachteten Werte kenntlich gemacht.

Der Kommentar rechts unten gibt den Grund für das Fehlschlagen des Tests an: Das Objekt vom Typ `Library` war nicht vorhanden, so dass der vom Testtreiber darauf auszuführende Methodenaufruf nicht initiiert werden konnte.

Es wird deutlich, dass man in Abhängigkeit vom Detaillierungsgrad der Eingabesequenzdiagramme sehr genaue Angaben darüber erhält, an

[36] SeDiTeC läuft derzeit vollständig in der gleichen Java Virtual Machine wie die zu testende Anwendung, was in speziellen Situationen problematisch sein kann (s. Kapitel 6).

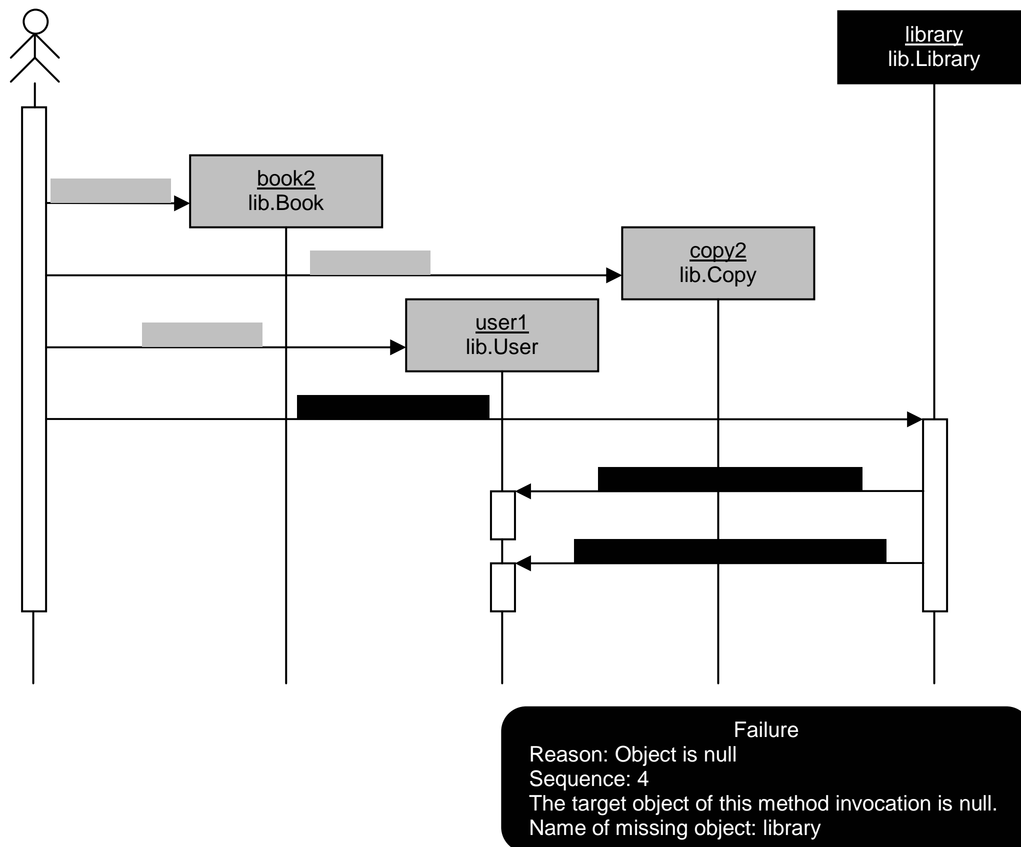

Abbildung 18: Grafische Aufbereitung von Testresultaten

welcher Stelle in der Anwendung welches Fehlverhalten aufgetreten ist, was im Allgemeinen die Fehlersuche erheblich vereinfacht. Auf der anderen Seite bedarf es jedoch auch einer gewissen Planung beim Instrumentieren der zu testenden Klassen, da sonst u.U. beobachtete Sequenzdiagramme entstehen, die Größen annehmen, die vollkommen unhandlich sind.[37]

4.3 Initialisierungszustand

Problematisch ist der Anfangszustand, in dem sich die Objekte vor dem ersten Methodenaufruf eines Testfalls befinden sollen, wobei mit Zustand hier die Belegung der Attribute eines Objekts gemeint ist.

Grundsätzlich sind zwei Lösungsansätze denkbar: Zum einen könnte man die Anfangsbelegungen der Attribute aller beteiligten Objekte manuell spezifizieren und diese Objekte dann vom Testwerkzeug mit einer geeigneten Routine entsprechend initialisieren lassen. Zum anderen kann man an das Sequenzdiagramm, dass den Testfall definiert, die Anforderung stellen, dass die erste Methode, die auf jedem enthaltenen

[37] So entsteht zum Beispiel beim Beobachten eines Java-Programms, das lediglich aus einer leeren `main` Methode besteht – wenn man das Java Development Kit instrumentiert – ein Sequenzdiagramm mit über 1700 Methodenaufrufen („Overhead" der Java Virtual Machine beim Starten und Beenden eines Programms).

Objekt aufgerufen wird, ein Konstruktor ist. In diesem Fall ist der Initialisierungszustand simpel: Es existieren einfach keine für den Test relevanten Instanzen vor dem ersten Methodenaufruf.

Die erste Alternative erfordert daher umfassendes Wissen über alle Attribute einer Klasse, da es sonst leicht passieren könnte, dass inkonsistente Attributbelegungen innerhalb eines Objekts erzeugt werden. Dieses Problem könnte man bestenfalls durch die (normalerweise alles andere als triviale) Spezifikation von Klasseninvarianten ausschließen. Da der zu erwartende Nutzen dieser Alternative als entsprechend gering eingestuft wurde, wurde sie für die Implementierung von SeDiTeC nicht weiter verfolgt.

Die zweite Alternative erfordert dagegen die komplette Methodenfolge, die ein Objekt ausgehend vom Konstruktor in den gewünschten Zustand versetzt. Diese Methodenfolgen lassen sich durch testbare Sequenzdiagramme beschreiben, welche dann die in Abschnitt 3.6 beschriebene Rolle der zustandserzeugenden Sequenzdiagramme übernehmen. Da es sich dabei um „normale" testbare Sequenzdiagramme handelt, wird diese Alternative von SeDiTeC sozusagen automatisch unterstützt.

Im Praxiseinsatz hat sich jedoch gezeigt, dass es mitunter sehr mühselig ist, solche zustandserzeugenden Sequenzdiagramme zu erstellen, zumal diese häufig ausschließlich dem Testprozess dienen und weder aus designtechnischen Gründen noch zu Dokumentationszwecken erstellt worden wären. Daher bietet SeDiTeC eine Schnittstelle an, mit der dem Werkzeug vor Beginn der Testfallausführung (aber auch währenddessen) Objekte bekannt gegeben werden können. Der folgende Codeausschnitt beispielsweise erzeugt ein Objekt vom Typ `User`, das dann unter dem Namen `user` bekannt gegeben wird und somit in Sequenzdiagrammen direkt verwendet werden kann.

```
User u = new User (42, "John Doe");
TestCenter tc = TestCenter.getTestCenter();
tc.introduceObject("user", u);
```

Dieser Mechanismus erlaubt daher auf einfache Art und Weise, alternative Initialisierungsmethoden zu verwenden, wie zum Beispiel das Laden von Objekten aus einer Datenbank oder das Initialisieren von Objekten mit Hilfe von normalem Sourcecode, ohne dabei auf das Erstellen von Sequenzdiagrammen angewiesen zu sein. Damit ähnelt dieser Mechanismus bewährten Initialisierungsmöglichkeiten wie beispielsweise der `setup` Methode der Klasse `TestCase` von JUnit.

4.4 Vor- und Nachbedingungen für Methoden

Die Spezifikation von Vor- und Nachbedingungen für Methoden ist ein wesentlicher Bestandteil des „Design by Contract" Konzepts [Meyer97]. Die Hauptaufgabe von Vor- und Nachbedingungen liegt dabei in der semantischen Spezifikation von Methoden, d.h. sie werden bereits im Zuge der Designaktivitäten erstellt. Da mit diesen Bedingungen allgemeingültige Aussagen über das Verhalten einer Methode getroffen

werden (sie sollen für jedes denkbare Szenario erfüllt sein), sind sie zumindest potentiell „mächtigere“ Werkzeuge der Spezifikation, als es einzelne Testfälle für eine Methode sind. Allerdings hängt das letztlich davon ab, wie exakt die formulierten Bedingungen sind – etwas ausführlicher als das meist wenig hilfreiche `true` sollte eine Nachbedingung schon sein. In der Praxis ist dieser Forderung jedoch häufig nicht leicht nachzukommen, da in vielen Fällen die für die Spezifikation einer Nachbedingung verwendete Logik identisch mit der für die Implementierung notwendigen Logik ist.

Ein weiteres Problem von Vor- und Nachbedingungen liegt darin, dass sie im Gegensatz zu Testfällen in diesem Kontext häufig nicht automatisiert überprüfbar sind. Selbst wenn eine ausreichend formale Spezifikation der Bedingungen vorliegt, hängt es immer noch von der verwendeten Sprache ab, wie leicht die Überprüfung der Bedingungen zur Laufzeit realisierbar ist. Während Eiffel – als eine Sprache, die sich nicht nur für die Implementierung, sondern auch für das Design als geeignet sieht – Vor- und Nachbedingungen umfangreich unterstützt, ist in Java lediglich ein rudimentäres Zusicherungskonzept enthalten (und auch das erst seit der aktuellen Version des JDK 1.4).

SeDiTeC unterstützt die Überprüfung von Vor- und Nachbedingungen sowohl in Methoden von instrumentierten Klassen (s. Anhang B) als auch in Teststubmethoden (s. Abbildung 13). Dafür müssen die Bedingungen im Klassenmodell als boolesche Ausdrücke in Java Syntax angegeben sein. Mittels des dafür definierten Schlüsselwortes `RESULT` kann man dabei auf den Wert zugreifen, den die Methode zurückgeben wird, falls die Methode über einen Rückgabewert verfügt. Daher lautete zum Beispiel die im Klassenmodell angegebene Nachbedingung, die zu der in Abbildung 13 enthaltenen Prüfung führte (Zeilen 43-46), `RESULT != null`.

4.5 Utility Klassen

Mit SeDiTeC lässt sich das Verhalten der IUT auf vielfältige Art und Weise testen. So lassen sich Objekte und die auf ihnen initiierten Methodenaufrufe, deren Reihenfolge sowie aufgetretene Ausnahmen prüfen. Ebenfalls prüfen lassen sich die Parameter und Rückgabewerte der Methoden – Ausprägungen von Basistypen werden auf Gleichheit geprüft und Instanzen von Klassen auf Identität („dasselbe Objekt an derselben Adresse im Speicher“).

Um diese Prüfungen durchführen zu können, sind Sequenzdiagramme auf einem angemessenen Abstraktionsniveau zu erstellen, d.h. einerseits müssen die Sequenzdiagramme so detailliert sein, dass die zu testende Funktionalität entsprechend enthalten ist, andererseits müssen diese Sequenzdiagramme und die zugehörigen Testdaten mit vertretbarem Aufwand erstellt werden können.

Bei manchen regelmäßig auftretenden Prüfszenarien ist es nicht sinnvoll, sie vollständig durch Sequenzdiagramme zu beschreiben, da sich das

eben angesprochene Gleichgewicht zwischen Erstellungsaufwand und Detaillierungsgrad nicht befriedigend herstellen lässt. Eine Lösung, um diese Szenarien trotzdem testen zu können, besteht darin, Prüfroutinen zu implementieren, die in diesen regelmäßig auftretenden Szenarien die Prüfung übernehmen können und dann an SeDiTeC lediglich die Rückmeldung liefern, ob die Prüfung erfolgreich verlaufen ist oder nicht. SeDiTeC bringt für zwei dieser Szenarien bereits Implementierungen mit, die in den beiden folgenden Abschnitten beschrieben werden.

4.5.1 Gleichheit von Objektstrukturen

Prinzipiell kann man die Auffassung vertreten, dass man beim Testen lediglich am von außen beobachtbaren Verhalten eines Objektes interessiert ist, während der Zustand und die Struktur des Objekts oder der Komponente – d.h. die Attributbelegungen der Instanzen – nebensächlich ist. Der Grund hierfür liegt darin, dass ein „fehlerhafter" Zustand, der sich nicht von außen beobachten lässt, keine Fehlerwirkung haben kann.

Tatsächlich ist es jedoch so, dass es in manchen Fällen erheblich effizienter ist, nach der Ausführung einer Methodensequenz auf einem Objekt zu überprüfen, ob sich dieses Objekt im erwarteten Zustand befindet, als alle möglichen Methodenaufrufe bzw. deren Kombinationen auszuführen, um zu testen, ob sich nicht ein Fehlverhalten beobachten lässt.

Die Frage ist nun, wie man den Zustand eines Objektes überprüfen kann. Eine Möglichkeit besteht darin, in die zu testende Klasse Methoden einzubauen, die explizit für diesen Zweck entworfen werden. Abgesehen davon, dass diese Methoden je nach Komplexität des Zustandsraumes eines Objektes selbst sehr komplex werden können und damit gründlich getestet werden müssen, ist diese Möglichkeit auch nicht immer umsetzbar. Zum einen handelt es sich bei diesen Methoden um Code, der für die eigentliche Funktionalität der Anwendung oft unnötig ist und zum anderen hat der Tester auf den entsprechenden Sourcecode u.U. keinen (schreibenden) Zugriff.

Eine weitere Möglichkeit besteht darin, ein Referenzobjekt zu erstellen, das dem erwarteten Sollzustand entspricht und dieses dann mit dem Objekt zu vergleichen, auf dem die zu testende Methodensequenz initiiert wurde. Hierbei gilt es zwei Probleme zu lösen. Erstens muss der Zustand des Referenzobjektes hergestellt werden. Dies lässt sich jedoch über vorhandene Konstruktoren und `set` Funktionen meist vergleichsweise einfach bewerkstelligen. Zweitens braucht man eine Methode, die in der Lage ist, zwei Objekte miteinander (angemessen) zu vergleichen. Anders als beispielsweise in Eiffel gibt es in Java hierfür keine in der Sprache eingebaute Funktionalität.

Zur Lösung dieses Vergleichsproblems könnte man nun von den Entwicklern verlangen, dass sie in jeder Klasse eine entsprechende Vergleichsoperation implementieren. Dies hätte jedoch die gleichen

Nachteile, die oben bereits im Kontext einer zustandsüberprüfenden Methode angeführt wurden. SeDiTeC verfügt daher über eine Funktionalität zur Überprüfung der Gleichheit zweier Objekte, die auf beliebige Objekte anwendbar ist. Von der Semantik her entspricht diese Funktionalität dem aus Eiffel bekannten `deep_equal` [Meyer92]. Demnach sind zwei Objekte O_1 und O_2 genau dann „deep-equal", wenn sie die folgenden Bedingungen erfüllen:

1. O_1 und O_2 sind vom exakt selben Typ.
2. Die Attribute in O_1, die vom Typ her keine Referenzen auf andere Objekte darstellen, sind mit den gleichen Werten belegt (bitweise gleich), wie die entsprechenden Attribute in O_2.
3. Jedes Referenzattribut, dass in O1 mit dem Wert `null` belegt ist, ist auch in O2 mit dem Wert `null` belegt.
4. Für jedes Referenzattribut in O_1, dass auf ein Objekt P_1 verweist, verweist das entsprechende Referenzattribut in O_2 auf ein Objekt P_2, und es ist unter der Annahme, dass O_1 und O_2 deep-equal sind, rekursiv möglich zu zeigen, dass P_1 und P_2 deep-equal sind.

Die etwas umständliche Formulierung aus Punkt 4 „... unter der Annahme, dass O_1 und O_2 deep-equal sind..." ist notwendig, um Gleichheit auch für Objekte feststellen zu können, die direkte oder indirekte Selbstverweise (Zirkel in der Referenzstruktur) beinhalten.

Dieser Ansatz, um zwei Objekte miteinander zu vergleichen, lässt sich zwar prinzipiell auf beliebige Objekte anwenden. Allerdings führt er – je nach Anwendungssituation – gegebenenfalls nicht zum erwarteten bzw. erwünschten Ergebnis. Betrachtet man beispielsweise die Klasse `java.lang.String`, so stellt man fest, dass diese im Wesentlichen drei für den Vergleich zweier `String` Objekte interessante Attribute beinhaltet: Ein Character-Array zum Speichern der Zeichenkette, einen Offset, der angibt, ab welcher Position des Arrays die gespeicherte Zeichenkette tatsächlich beginnt und die Länge der Zeichenkette. Wenn daher zwei Objekte vom Typ String beide die Zeichenkette „abc" speichern, so können sie sich intern trotzdem beispielsweise durch die Länge des Arrays und den Offset unterscheiden. Die beiden Objekte wären daher nicht deep-equal.

Dieses Verhalten mag erwünscht sein, wenn man zwei `String` Objekte direkt miteinander vergleicht. Möchte man jedoch beispielsweise zwei Objekte vom Typ `User` aus dem Bibliotheksbeispiel miteinander vergleichen, so ist es schon sehr fraglich, ob die Ungleichheit nicht nur aufgrund unterschiedlicher Ausleihlisten oder unterschiedlicher Namen etc. festgestellt werden soll, sondern auch aufgrund unterschiedlicher interner Speicherstruktur der beiden `String` Objekte, die z.B. den jeweiligen Vornamen repräsentieren.

Aus diesem Grund stellt SeDiTeC eine Möglichkeit zur Verfügung, den Vergleich zweier Objekte zu flexibilisieren. Abbildung 19 gibt einen

Überblick darüber, wie die Vergleichsfunktionalität in SeDiTeC implementiert ist. Für den eigentlichen Vergleich verantwortlich ist die Methode `deepEqual` der Klasse `Equality`. Diese Methode verwendet den oben beschriebenen Algorithmus zum Vergleichen zweier Objekte. Möchte man dieses Standardverhalten für Instanzen einer bestimmten Klasse ändern, so implementiert man hierfür eine Klasse, die von der abstrakten Klasse `EqualityCustomImplementation` erbt.[38] Diese Klasse muss zwei Methoden (re-)definieren: `isResponsibleFor` und `deepEqual`. Erstere entscheidet für ein übergebenes Objekt, ob diese „Vergleichsklasse“ für den Vergleich eines solchen Objektes mit einem anderen zuständig ist und letztere führt den Vergleich dann durch. Diese Möglichkeit zur Flexibilisierung der Vergleichsfunktionalität erhöht die praktische Verwendbarkeit derselben erheblich.

4.5.2 Dateivergleich

Ein weiteres häufiges Szenario beim Testen von Software ist die Prüfung von Dateien auf Gleichheit. Dies geschieht nicht notwendigerweise nur dann, wenn die zu testende Anwendung tatsächlich im normalen Betrieb Dateien erstellt oder verändert. Häufig werden auch aus der zu testenden Anwendung bestimmte Informationen in Dateien protokolliert, da dies das Testen und auch das sich gegebenenfalls anschließende Debugging u.U. vereinfacht. Während der Testausführung werden dann die erstellten

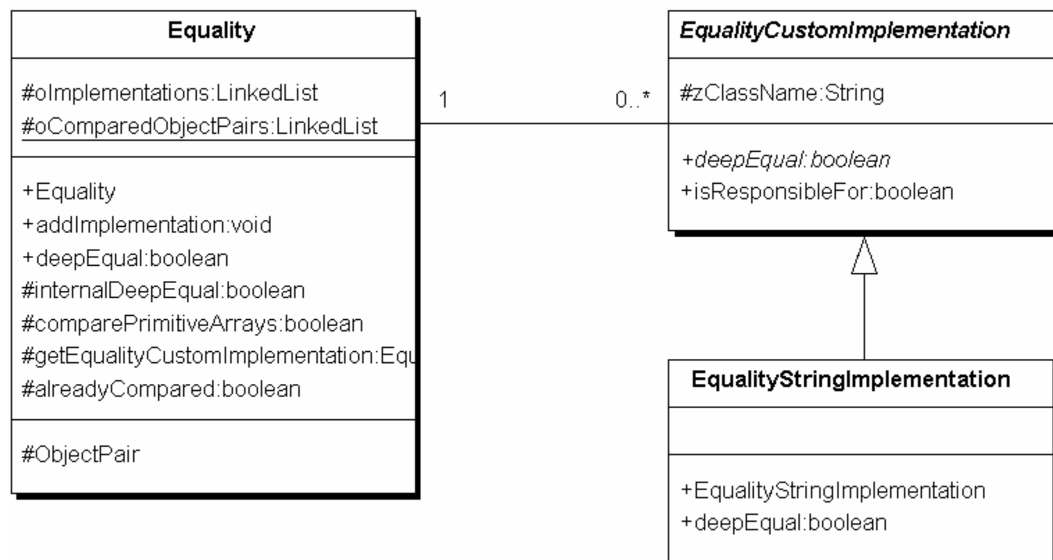

Abbildung 19: Implementierung der Vergleichsfunktionalität in SeDiTeC

[38] Die in Abbildung 19 gezeigte Klasse `EqualityStringImplementation` beispielsweise implementiert einen (alternativen) Vergleich für Objekte vom Typ `String`.

Testprotokolldateien mit bereits vor dem Test erstellten erwarteten Protokolldateien verglichen.

Den inhaltlichen Vergleich zweier Dateien bzw. das Erstellen einer Datei in einem Eingabesequenzdiagramm zu beschreiben, ist üblicherweise wenig sinnvoll, da zum einen die dabei involvierten Methodenaufrufe (für das eigentliche Herausschreiben der Datei) nur von geringem Interesse sind und zum anderen die als Parameter anzugebenden Daten sehr umfangreich und damit unhandlich sind. SeDiTeC verfügt daher über eine Klasse `FileCompare`, die byteweise überprüft ob zwei Dateien identisch sind. Hierfür ist lediglich ein Methodenaufruf notwendig, dem als Parameter die vollständigen Namen der beiden Dateien übergeben werden.

4.6 Automatisierung mit Apache Ant

Möchte man eine Anwendung mit SeDiTeC testen, so sind unter Verwendung der bisher vorgestellten Funktionalität eine Reihe von manuellen Schritten notwendig, um einen Testlauf durchzuführen (s. auch Abschnitt 4.2). Zunächst einmal sind in einem ersten Schritt im Rahmen der Testspezifikation die Sequenzdiagramme zu erstellen (bzw. die aus den Designaktivitäten hervorgegangenen zu ergänzen). Diese müssen dann exportiert werden und mit Hilfe von SeDiTeC kombiniert und mit Testdaten versehen werden. Für den eigentlichen Testlauf sind dann in einem zweiten Schritt die Anwendungsklassen in Together zu instrumentieren bzw. durch Teststubs zu ersetzen. Die auf diese Art und Weise bearbeiteten Klassen müssen daraufhin kompiliert werden. Schließlich muss noch aus SeDiTeC heraus eine Datei mit Eingabesequenzdiagrammen ausgewählt und der eigentliche Testlauf angestoßen werden.

Gerade das manuelle Instrumentieren bzw. „Stubben" der Anwendungsklassen kann dabei sehr zeitraubend sein, da man zum einen alle beteiligten Klassen (wobei es sich leicht um mehrere 100 handeln kann) auf der grafischen Benutzungsoberfläche auswählen muss, die sich meist auch in vielen unterschiedlichen Diagrammen befinden. Zum anderen arbeiten die SeDiTeC Module direkt auf den Klassen, die dem Modell zugrunde liegen, so dass man die Anwendungsklassen zunächst sichern muss, da sonst u.U. die Originalklassen des Entwicklers beispielsweise mit Teststub- oder Instrumentierungscode überschrieben werden. Da dieser Arbeitsaufwand insgesamt die Benutzbarkeit von SeDiTeC als Werkzeug zum entwicklungsbegleitenden Testen stark einschränken würde, wurden einige dieser Schritte mit Apache Ant (kurz: Ant) automatisiert.

Bei Ant [ASF03a] handelt es sich um ein Java basiertes Build-Werkzeug ähnlich Make für C/C++. Ant wird über „Build-Dateien" gesteuert, die XML-basiert sind. Innerhalb einer solchen Build-Datei können so genannte „Targets" definiert werden, die bestimmte Aufgaben („Tasks") innerhalb eines Build-Prozesses übernehmen und für die Abhängigkeiten

angegeben werden können. So kann man zum Beispiel für ein Target „Execute“ definieren, dass es abhängig ist von einem Target „Compile“, womit sichergestellt wird, dass das betreffende Projekt immer zuerst erneut kompiliert wird, bevor die Applikation ausgeführt wird.

Ant unterstützt dabei eine ganze Reihe von vordefinierten Tasks z.B. zum Kompilieren und Ausführen von Java-Anwendungen, zum Kopieren, Verschieben und Löschen von Dateien etc. Wird eine Funktionalität benötigt, die nicht durch die vordefinierten Tasks abgedeckt wird, so kann man eigene Tasks in Java programmieren, indem man auf die dafür zur Verfügung gestellte Programmierschnittstelle zurückgreift.

Für SeDiTeC wurde daher eine solche Task entwickelt, mit deren Hilfe sich im Rahmen der Benutzung von Ant die oben als „zweiter Schritt“ bezeichneten Aufgaben (Kopieren der Sourcedateien, Instrumentieren und Stubben, Kompilieren sowie Ausführen der Testfälle) komfortabel automatisieren lassen, so dass – eine entsprechende Build-Datei vorausgesetzt – für alle diese Tätigkeiten lediglich ein einzelner Befehl notwendig ist.

4.7 SeDiTeC und JUnit

Bei der Vorstellung der in dieser Arbeit enthaltenen Konzepte auf Konferenzen bzw. bei Gesprächen mit Werkzeugherstellern wie SQS und Togethersoft tauchte immer wieder die Frage auf: Wie steht SeDiTeC eigentlich zu JUnit?

Auf den ersten Blick mag die Vermutung nahe liegen, dass typische Testfälle für JUnit auch typische Testfälle für SeDiTeC sind und umgekehrt, da man in beiden Fällen Methodenaufrufsequenzen auf Implementierungsebene angibt – einmal sozusagen textuell als Sourcecode, einmal visuell als Diagramm. Bei näherer Betrachtung fällt jedoch auf, dass der Fokus jeweils etwas anders gesetzt ist.

Bei typischen JUnit-Testfällen versucht man meist eine oder mehrere Methoden *einer* Klasse zu überprüfen. Werden im Zuge des Testfalls Instanzen weiterer Klassen erstellt, so sind sie häufig nur „notwendiges Übel“ und nicht Gegenstand des Tests. Die eigentliche Prüfung erfolgt dabei anhand der Rückgabewerte von durch die Testmethode aufgerufenen Methoden und eventuell geworfenen Ausnahmen.

Bei typischen SeDiTeC-Testfällen hingegen geht es eher darum, das Zusammenspiel mehrerer Instanzen verschiedener Klassen zu testen. Überprüft werden daher nicht nur die Rückgabewerte der von außen aufgerufenen Methoden, sondern auch die von diesen Methoden angestoßenen Interaktionen, sprich: Methodensequenzen inklusiver involvierter Parameter, Rückgabewerte und Ausnahmen. Solch eine Prüfung ist mit JUnit ohne weiteres gar nicht möglich – auf der anderen Seite sind solche Testfälle natürlich auch komplexer und daher aufwendiger zu erstellen.

Auch auf Anregung von Togethersoft hin wurde in [Rossnagel02] untersucht, inwieweit ein Roundtrip-Engineering zwischen SeDiTeC- und JUnit-Testfällen sinnvoll und realisierbar sei. Dabei wurde gezeigt, dass sich JUnit-Testfälle prinzipiell immer in SeDiTeC-Testfälle umwandeln lassen und die so gewonnenen SeDiTeC-Testfälle sich wieder auf die Ausgangs-JUnit-Testfälle abbilden lassen. Die andere Richtung – ausgehend von einem SeDiTeC-Testfall zu JUnit und wieder zu SeDiTeC – funktioniert jedoch aus oben genanntem Grund (mittels JUnit kann man nicht direkt Interaktionen zwischen Testobjekten überwachen) nur für spezielle Testfälle, bei denen alle Methodenaufrufe vom Testtreiber-Aktor ausgehen.

Hier stellt sich die Frage: Warum sollte man dann nicht alle ggf. vorhandenen JUnit-Testfälle nach SeDiTeC importieren und auf JUnit verzichten? Auch wenn das ein prinzipiell gangbarer Weg wäre, gibt es doch zumindest zwei Gründe, die dagegen sprechen. Zum einen besteht eine zentrale Idee von SeDiTeC darin, dass Sequenzdiagramme, die im Zuge des Designs entstehen, auch als Testbasis verwendet werden. Sequenzdiagramme sollten nicht ausschließlich zum Testen erstellt werden müssen (zumindest ein erheblicher Anteil nicht). Die Sequenzdiagramme, die durch importierte, meist vergleichsweise simple JUnit-Testfälle entstehen, wären aber häufig gerade keine Kandidaten für Diagramme, die im Zuge des Designs entstehen würden. Aus Sicht eines SeDiTeC-Anwenders sehen solche importierten Testfälle dann häufig konstruiert und / oder unvollständig aus und ihre Pflege ist (innerhalb von SeDiTeC dann verglichen mit JUnit) mühsam.

Zum anderen gibt es auch gar keinen Grund, JUnit und SeDiTeC nicht parallel einzusetzen. Eine einfache und elegante Möglichkeit ergibt sich beispielsweise durch die Verwendung von Ant (s. Abschnitt 4.6), dass auch die Automatisierung von JUnit unterstützt. Es lässt sich daher problemlos eine Build-Datei für Ant erstellen, die sowohl die vorhandenen SeDiTeC- als auch die JUnit-Testfälle sozusagen auf einen Knopfdruck ausführt.

Anzumerken ist, dass sowohl SeDiTeC als auch JUnit einen Schwerpunkt auf eine „Test-First-Herangehensweise“ [Link02] legen. JUnit unterstützt dies insbesondere durch seine einfache Verwendbarkeit und die Möglichkeit, kleine, kompakte Tests mit wenig Aufwand zu erstellen und durchzuführen. Bei SeDiTeC hingegen ist die Testfallspezifikation aufwändiger – man erhält jedoch im Gegenzug eine erhöhte Flexibilität durch das von SeDiTeC implementierte Teststubkonzept, das nicht nur sehr frühes Testen auch von abhängigen Komponenten erlaubt, sondern auch einen gewissen Spielraum einräumt bei der Konfiguration einzelner Testfälle: Ersetzt man beispielsweise in einem Eingabesequenzdiagramm alle vorkommenden Klassen bis auf eine durch Teststubs, so erhält man einen klassischen Unittest, verzichtet man auf den Einsatz von Teststubs, kann das gleiche Eingabesequenzdiagramm in Abhängigkeit von seiner Granularität als Integrationstest oder auch als Systemtest dienen.

Gerade diese besondere Unterstützung von Integrationstests ist von hoher praktischer Bedeutung, da sich solche Tests in vielen Fällen weder mit JUnit noch mit klassischen (System-)Testwerkzeugen wie zum Beispiel Capture & Replay Werkzeugen befriedigend durchführen lassen.

4.8 Praxiserfahrung

Im Rahmen von Industriekooperationen, die im Kontext der in Kapitel 1 erwähnten Software Engineering Veranstaltung an der TU Darmstadt stattfanden, wurde SeDiTeC im Laufe seiner Entwicklung in verschiedenen Projekten in der Praxis eingesetzt. Inhalte dieser Projekte waren u.a. die Entwicklung eines Informationsmanagementsystems für Wertpapiere und die Entwicklung eines Werkzeugs zur Aufwandsschätzung von RUP basierten Softwareentwicklungsprojekten. Es handelte sich damit um typische Projekte aus der erklärten „Zielgruppe“ von SeDiTeC.

Beim ersten Praxiseinsatz von SeDiTeC traten neben den erwarteten Problemen insbesondere eine ganze Reihe von technischen Detailproblemen auf. Ein Beispiel: Wie sehr wohl bekannt ist, kann man beim Erzeugen von Objekten einer Java Klasse, in der keine Konstruktoren explizit definiert sind, den argumentlosen Standardkonstruktor verwenden. Es macht aus Sicht des Programmierers normalerweise keinen Unterschied, ob tatsächlich kein Konstruktor in dieser Klasse definiert ist, oder ob ein leerer, argumentloser Konstruktor (mit Sichtbarkeit `public`) in der Klasse enthalten ist. Für SeDiTeC war dies jedoch sehr wohl ein Unterschied, da zum einen Together einen nicht explizit vorhandenen Konstruktor bei der Sequenzdiagrammerstellung nicht zur Auswahl zulässt. Zum anderen funktionierte die damals gewählte Methode zur Initiierung von Konstruktoraufrufen durch den SeDiTeC-Testtreiber mittels der Java Reflection API nur für explizit vorhandene Konstruktoren.

Selbstverständlich beinhaltet das Feedback, das aus einem Praxiseinsatz resultiert, meist auch eine Reihe von wertvollen Anregungen. Viele der zu Anfang dieses Kapitels genannten Anforderungen an ein auf Sequenzdiagrammen basierenden Testwerkzeugs gehen auf solche Anregungen zurück, wie zum Beispiel die Möglichkeit, Initialisierungszustände nicht ausschließlich über Sequenzdiagramme spezifizieren zu müssen.

Die größte Hürde, die viele Anwender von SeDiTeC zu überwinden hatten, die erst spät im Projekt damit begannen, SeDiTeC und Sequenzdiagramme zu nutzen[39], war ein Gefühl dafür zu bekommen, auf welcher Abstraktionsebene Sequenzdiagramme angesiedelt sein müssen, um für

[39] Die Tatsache, dass SeDiTeC bei zwei Projekten erst gegen Ende derselbigen eingesetzt wurde, resultierte einerseits daraus, dass SeDiTeC zu Beginn dieser Projekte noch nicht einsatzfähig war und andererseits daraus, dass zumindest bei einem der beiden Projekte die generelle Einsicht in die Notwendigkeit des Testens erst spät erfolgte.

das Testen sinnvoll eingesetzt werden zu können. Häufig bestanden die ersten Versuche aus mehr oder weniger vollständig (inklusive *aller* Methodenaufrufe) spezifizierten Use Case Szenarien. Der Aufwand, um ein solches Sequenzdiagramm mit zugehörigen Testdaten zu erstellen, ist enorm hoch und damit inakzeptabel – zumal sich aus Testersicht normalerweise ein vergleichbarer Nutzen mit einem abstrakteren Sequenzdiagramm erreichen lässt. Ein solches Sequenzdiagramm ähnelt dann auch viel mehr den Sequenzdiagrammen, die tatsächlich im Designprozess entstehen (und die im Normalfall die Grundlage für die testbaren Sequenzdiagramme bilden).

Problematisch ist auch der hohe Aufwand, der entsteht, wenn man ein Sequenzdiagramm leicht abändert, da man derzeit auf der Oberfläche von SeDiTeC keine Testfälle kopieren und einfügen kann. Das führt dazu, dass die Testfälle für ein geändertes Diagramm meist erneut eingegeben werden müssen.

Sehr positiv wurde in den Projekten die Stubfunktionalität von SeDiTeC bewertet, die zum Beispiel in einem Projekt dazu verwendet wurde, einen Webserver zu simulieren und eine Reihe von Tests damit unabhängig von einer Netzwerkverbindung und einem (noch nicht verfügbaren) Webserver durchführen zu können.

Insgesamt gesehen waren die Erfahrungen aus den Projekten auf der einen Seite sehr ermutigend, da sich eine Reihe von Anwendungsmöglichkeiten aufgetan haben, in denen die hier vorgestellten Konzepte (insbesondere die Möglichkeit, die Kommunikation zwischen implementierten Methoden zu beobachten und die Stubfunktionalität) einen echten Mehrwert darstellen. Auf der anderen Seite war mit dem Einsatz von SeDiTeC aber auch immer ein hoher Zeitaufwand verbunden. Dieser kann zwar durch eine Verbesserung der Benutzungsoberfläche gesenkt werden – je nach Anwendungsgebiet bleibt aber die Frage, wie weit er sich senken lässt.

4.9 Verwandte Arbeiten und existierende Testwerkzeuge

Der Forschungsprototyp SCENTOR [Wittevrongel03] bedient sich JUnit, um aus Sequenzdiagrammen ausführbare Tests zu generieren. Entsprechend der Funktionalität von JUnit lässt sich jedoch immer nur ein Objekt des Sequenzdiagramms testen (s. Abschnitt 2.1.3). Es existiert weder die Möglichkeit, Sequenzdiagramme zu kombinieren, noch eine Stubfunktionalität.

Einen weitaus formaleren Ansatz verfolgen Briand und Labiche mit dem Projekt TOTEM („Testing Object-orienTed systEms with the unified Modeling language“) [Briand+01], das für (System-)Tests nicht nur auf Sequenzdiagramme, sondern auch auf eine Reihe von anderen UML Diagrammen zurückgreift. Es werden erhebliche Anforderungen an die Erstellung der UML Modelle gestellt, wie zum Beispiel die Verwendung der OCL zum Definieren von u.a. Klasseninvarianten [Warmer+99] und die Verwendung von „Extended Use Cases“ [Binder00], um das Generieren

von Testfällen zu unterstützen. Dem erhöhten Aufwand bei der Modellerstellung stehen erweiterte Automatisierungsmöglichkeiten gegenüber, die die Autoren bei Anwendungsentwicklungen mit hohen Sicherheitsanforderungen oder im Bereich eingebetteter Systeme für gerechtfertigt sehen. Eine prototypische Implementierung einiger dieser Konzepte existiert [Software03].

Ebenfalls von Briand und Labiche ist ein Ansatz, Änderungen eines UML Modells zu erkennen und aus einer Menge von (System-)Tests diejenigen auszuwählen, die als Folge dieser Änderungen erneut durchgeführt werden müssen [Briand+02]. Hierbei beschränken sie sich auf Klassen-, Sequenz- und Use Case Diagramme. Stark vereinfachend gehen die Autoren jedoch davon aus, dass ein Use Case mit allen alternativen Pfaden immer durch genau ein Sequenzdiagramm realisiert wird.

Abdurazik und Offutt schlagen vor, UML Kollaborationsdiagramme als Grundlage für das Testen zu verwenden [Abdurazik+00]. Dabei gehen sie davon aus, dass jede Methode durch ein Kollaborationsdiagramm beschrieben wird. Die Testdaten sind jeweils im Diagramm direkt anzugeben. Eine Möglichkeit, Diagramme zu kombinieren oder von der starren Zuordnung zwischen Diagramm und Methode abzuweichen, ist nicht vorgesehen. Ebenfalls bleibt offen, wie mit unterschiedlichen Testdatensätzen verfahren werden soll. Zusätzlich zum dynamischen Testen schlagen sie jedoch noch die Ableitung von statischen Tests vor, die die in Kollaborationsdiagrammen ggf. vorhandenen Informationen über Assoziationen von Objekten prüfen sollen.

Die Zahl der existierenden Testwerkzeuge, die bzgl. der Testfallspezifikation auf UML Sequenzdiagrammen beruhen, ist (zumindest noch) recht gering. Zu den im Kontext dieser Arbeit interessanteren zählt der Rhapsody TestConductor von I-Logix [I-Logix03]. Bei Rhapsody handelt es sich um ein UML-basiertes Entwicklungswerkzeug, dass auf die Entwicklung von eingebetteten Systemen spezialisiert ist. Der TestConductor (als Add-On für Rhapsody verfügbar) ist ein Sequenzdiagramme verwendendes Werkzeug zum Ausführen von szenariobasierten Tests. Dabei ist es u.a. möglich, mehrere Datensätze für ein Sequenzdiagramm anzugeben, Sequenzdiagramme zu kombinieren und Testergebnisse wiederum als Diagramm darzustellen. Der TestConductor befindet sich allerdings noch in der Entwicklung, so dass in der aktuellen Version 1.2 einiges an elementarer Funktionalität fehlt wie z.B. die Möglichkeit, Rückgabewerte von Methodenaufrufen zu prüfen oder die Fähigkeit zum Batchbetrieb. Auch eine Stubfunktionalität im Sinne von SeDiTeC, die es dem Tester erlaubt, für ein bestehendes Sequenzdiagramm Klassen dynamisch entweder zu testen oder durch einen Stub zu ersetzen, gibt es nicht.

Beim Rational QualityArchitect [Rational03] handelt es sich um ein Werkzeug, dass Sequenzdiagramme für die Testspezifikation verwendet und auf das Testen von EJB- und COM-Komponenten spezialisiert ist. Aufbauend auf den in einem Rational Rose Modell vorhandenen

Sequenzdiagrammen generiert der QualityArchitect Sourcecode, der auf die in einem Datenpool gespeicherten Daten zurückgreift. Für die Codegenerierung werden dabei Templates verwendet, die bei Bedarf angepasst werden können. Die Generierung von Stubs ist möglich, allerdings beschränkt sich die Funktionalität der Stubmethoden darauf, abhängig von der Parameterkombination bestimmte Werte zurückzugeben oder Exceptions zu werfen. Die Parameterkombinationen, bei denen es sich nicht um komplexe Datentypen handeln darf, müssen vorher in einer Tabelle hinterlegt werden. Eine Kombination von Sequenzdiagrammen ist nicht möglich.

Was JUnit für das Erstellen von Unittests leistet, leisten so genannte Mock Objects [ASF03b] für das Erstellen von Stubs [Mackinnon+01]. Mit Hilfe der Mock Objects API lassen sich Stubs einheitlich unter Verwendung eines zielgerichteten Formats implementieren, so dass der Aufwand, der für die Stuberstellung entsteht, zumindest reduziert werden kann. Mock Objects werden häufig zusammen mit JUnit verwendet.

5 Testen von Design Pattern Implementierungen

Die bekannten Gang-of-Four Design Patterns [Gamma+95] stellen Lösungen für immer wiederkehrende Problemstellungen im Rahmen der Softwareentwicklung bzw. des Softwaredesigns dar. Die Beschreibung eines Design Patterns[40] besteht dabei u.a. aus einer prinzipiellen Motivation für das Pattern, einer Beschreibung der beteiligten Rollen sowie einer Beschreibung der Implementierungsstruktur in Form eines Klassendiagramms. Außerdem wird zu einer Reihe von Design Patterns ein abstraktes Sequenzdiagramm zur Verfügung gestellt, das beispielhaft eine mögliche Interaktion zwischen den beteiligten Objekten veranschaulicht.

Eine an und für sich nahe liegende sinnvolle Erweiterung der Design Pattern Beschreibungen wären Testfälle, die geeignet sind, eine Implementierung der in einem Design Pattern enthaltenen Semantik zu testen. Selbstverständlich können ohne nähere Kenntnis des konkreten Anwendungsfalls solche Testfälle bestenfalls die Semantik testen, die tatsächlich Bestandteil des Design Patterns ist. Das heißt, es ist beispielsweise möglich, Testfälle anzugeben, die überprüfen, ob sich eine Menge von Klassen gemäß dem Visitor Pattern verhalten. Ob die verschiedenen `visit` Methoden jedoch korrekt im Sinne der Anwendung implementiert sind, lässt sich jedoch natürlich nur mit Testfällen prüfen, die auf *diese* konkrete Anwendungssituation abgestimmt sind.

Ziel dieses Kapitels ist es, für einige der Design Patterns zu überprüfen, inwieweit sich für die selbigen Testfälle in Form von testbaren Sequenzdiagrammen[41] angeben lassen. Die Auswahl der Testfälle erfolgt dabei anhand der Beschreibungen der Design Patterns bzw. der beteiligten Objekte und Methoden, da allgemeine Use Case Szenarien nicht vorliegen und konkrete Implementierungen offensichtlich ungeeignet zur Herleitung von allgemein gültigen Testfällen wären. Die Design Patterns werden jeweils kurz inhaltlich beschrieben – für detailliertere Informationen sei auf [Gamma+95] verwiesen. Im Anschluss folgt eine Bewertung.

5.1 Iterator

Container-Objekte wie Listen oder Vektoren sollten eine Möglichkeit zur Verfügung stellen, wie auf die in ihnen gespeicherten Elemente zugegriffen werden kann, ohne dass dabei die interne Struktur der Container-Objekte offen gelegt wird. Eine mögliche Lösung für diese Aufgabe stellt das Iterator Pattern dar. Dieses Pattern definiert ein

---

[40] In diesem Kapitel sind mit dem allgemeinen Begriff „Design Patterns“ immer die in [Gamma+95] vorgestellten Design Patterns gemeint.

[41] Wobei zu beachten ist, dass in den erstellten Sequenzdiagrammen auf die Initialisierung der beteiligten Objekte – soweit sie für den Testfall nicht wichtig ist – verzichtet wird (s. Abschnitt 4.3).

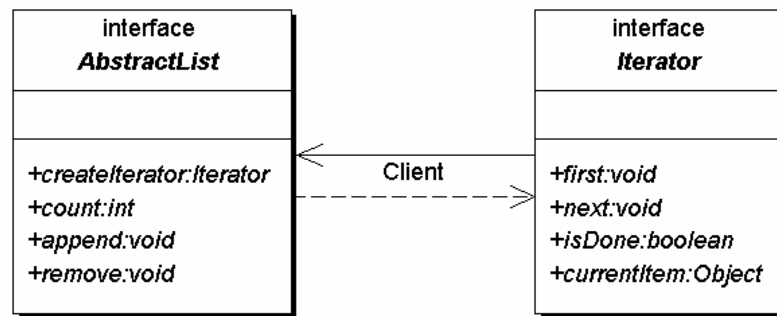

Abbildung 20: Iterator Pattern

Interface `Iterator` und beschreibt anhand des (beispielhaften) Interfaces `AbstractList` (s. Abbildung 20) mögliche Interaktionen zwischen einem Iterator- und einem Container-Objekt.

Ein Iterator ist sozusagen ein Zeiger oder ein Lesezeichen in einem Container, der zu jeder Zeit entweder auf ein bestimmtes Objekt der Liste zeigt, das mittels der Methode `currentItem` abgefragt werden kann, oder auf das Ende der Liste zeigt („hinter" das letzte Objekt des Containers). In diesem Fall soll die Methode `currentItem` eine `NoSuchElementException` werfen.

Die Methode `isDone` gibt genau dann `true` zurück, wenn sich der Iterator hinter dem letzten Element des Containers befindet oder der Container leer ist, sonst `false`. Die beiden Methoden `first` und `next` dienen der Navigation des Iterators. Die Methode `first` bewegt dabei den Zeiger auf das erste Element des Containers (oder „hinter das letzte" Element, falls der Container leer ist). Die Methode `next` bewegt den Zeiger auf das nächste Element oder hinter das letzte Element, falls er sich vor diesem Aufruf von `next` auf dem letzten Element befunden hat. Wird die Methode `next` aufgerufen, wenn sich der Zeiger bereits hinter dem letzten Element des Containers befindet, so wird eine `NoSuchElementException` geworfen.[42]

Ein Unterscheidungsmerkmal von Iteratoren ist, ob sie nach ihrer Erschaffung Veränderungen am zugrunde liegenden Container wahrnehmen (d.h. zum Beispiel auch erst später hinzugefügte Elemente zurückgeben) oder ob sie diese Veränderungen ignorieren (d.h. den zum Zeitpunkt ihrer Erschaffung aktuellen Zustand des Containers sozusagen konservieren). Letztere Iteratoren werden in [Gamma+95] als „robust" bezeichnet. Für den Rest dieses Abschnitts gehen wir jedoch von

[42] Selbstverständlich macht es nur Sinn von einem „ersten", „nächsten" und „letzten" Element zu sprechen, wenn für die Elemente des Containers eine Reihenfolge definiert ist. Dies kann sowohl eine durch den Container implizierte Reihenfolge sein (z.B. vorwärts durch eine Liste) als auch eine beliebige andere durch bzw. für den Iterator festgelegte – zumal es eine Reihe von Containern wie z.B. Mengen gibt, die keine Reihenfolge ihrer Elemente implizieren.

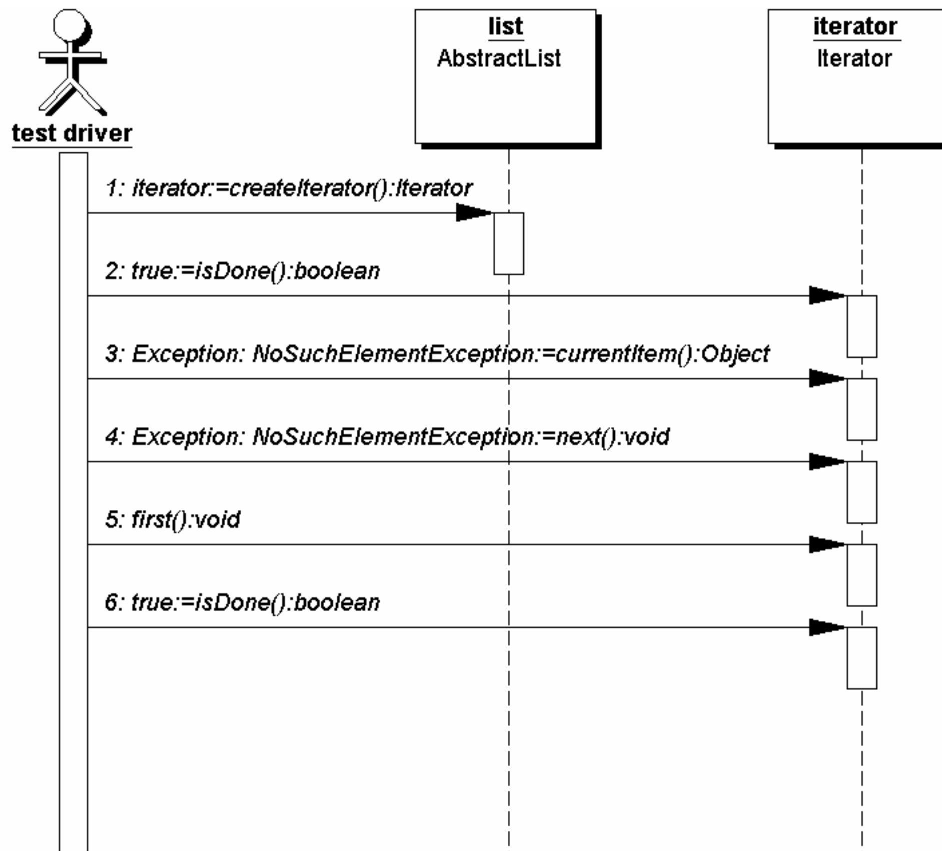

Abbildung 21: Iterator Pattern – Testfall I

Iteratoren aus, von denen erwartet wird, dass sie auch Änderungen des Containers nach ihrer Erschaffung reflektieren – die also in diesem Sinne nicht robust sind.

Das Interface `AbstractList`, das stellvertretend für beliebige Container-Klassen steht, wird benötigt, um aussagekräftigere Testfälle zu erstellen, als das mit dem `Iterator` Interface alleine möglich wäre. Die Methoden von `AbstractList` sind intuitiv verständlich und in den meisten Container-Klassen so oder zumindest sehr ähnlich enthalten. Erwähnenswert ist lediglich die Methode `createIterator`, die von Klienten eines `AbstractList`-Objekts verwendet werden kann, um einen Iterator für dieses `AbstractList`-Objekt zu erhalten.

Das in Abbildung 21 gezeigte testbare Sequenzdiagramm beschreibt ein Szenario bei dem ein Iterator von einer leeren Liste erstellt wird. Laut Spezifikation sollte die Methode `isDone` bei einer leeren Liste immer `true` zurückgeben, während von den Methoden `currentItem` und `next` in dieser Situation eine `NoSuchElementException` erwartet wird. Dies wird in den Methodenaufrufen 2-4 überprüft. Die Methodenaufrufe 5 und 6 machen deutlich, dass auf „leeren“ Iteratoren zwar die Methode `first` aufgerufen werden kann (ohne eine Exception auszulösen), dies aber nicht daran ändert, dass der Iterator „am Ende“ ist.

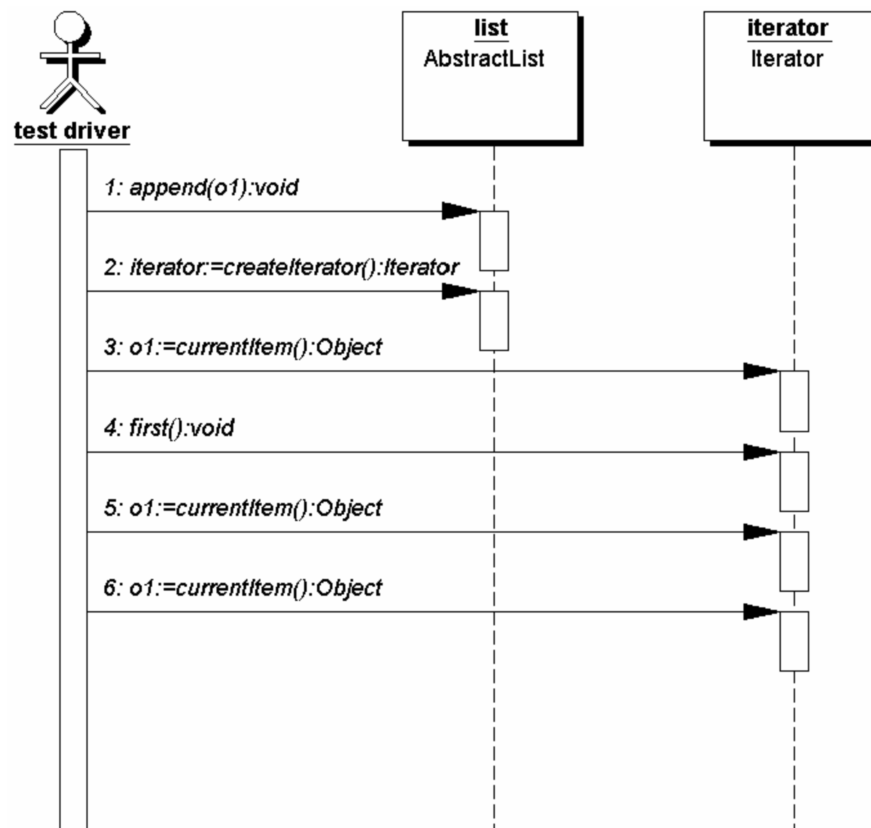

Abbildung 22: Iterator Pattern – Testfall II

Das nächste testbare Sequenzdiagramm (s. Abbildung 22) verwendet nun eine Liste, in der ein Element enthalten ist, nämlich das Objekt `o1`. Die Methodenaufrufe 3-6 testen nun im Wesentlichen zwei Dinge: Zum einen soll ein Iterator nach der Erschaffung von sich aus auf das erste Element zeigen, ohne dass dafür ein expliziter Aufruf der Methode `first` durch den Klienten notwendig wäre. Zum anderen sollen zwei direkt aufeinander folgende Aufrufe der Methode `currentItem` dasselbe Element zurückliefern, d.h. ein Aufruf von `currentItem` ändert nicht die Position des Zeigers.

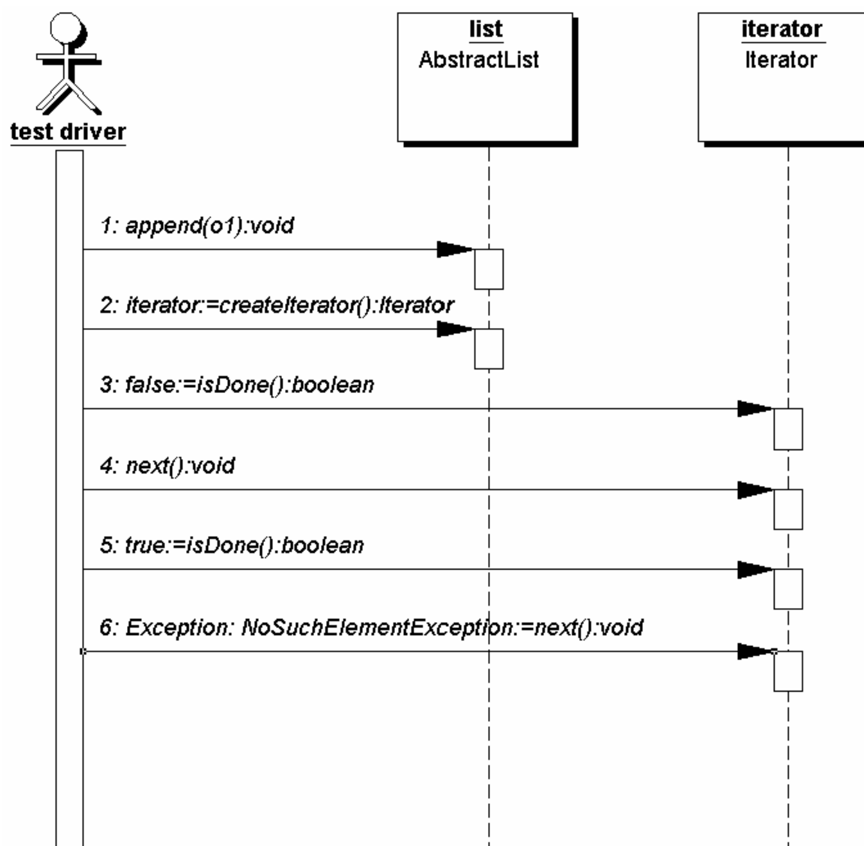

Abbildung 23: Iterator Pattern – Testfall III

Auch das folgende testbare Sequenzdiagramm (s. Abbildung 23) verwendet eine Liste, in der ein Element enthalten ist. Getestet wird hier das Iterieren durch die Elemente des Containers mittels der Methode `next` im Zusammenspiel mit der Methode `isDone`. Wichtig ist auch hier, dass am Ende des Testfalls überprüft wird, ob als Reaktion auf eine Fehlbenutzung des Iterators (der Aufruf der Methode `next`, obwohl das Ende der Liste bereits überschritten wurde) die erwartete Ausnahme geworfen wird.

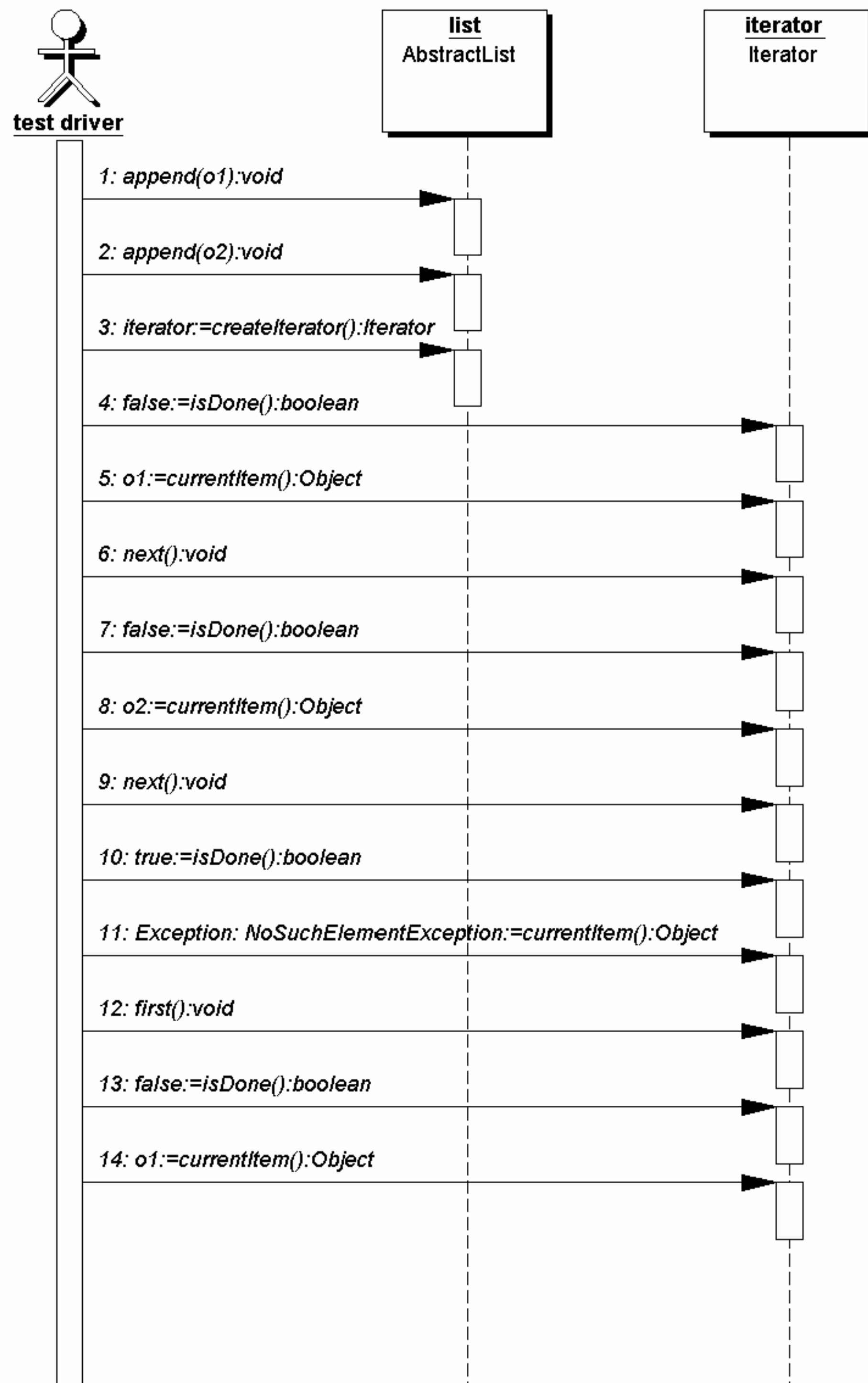

Abbildung 24: Iterator Pattern – Testfall IV

Der in Abbildung 24 gezeigte Testfall IV testet ein relativ typisches Szenario für die Verwendung eines Iterators, bei dem die einzelnen Elemente des Containers nacheinander abgefragt werden, wobei durch den jeweiligen Aufruf der Methode `isDone` abgefragt wird, ob das Ende der Liste bereits erreicht wurde. Nachdem der Durchlauf der Liste mit Methodenaufruf 10 beendet wurde, wird mit Methodenaufruf 11 wieder

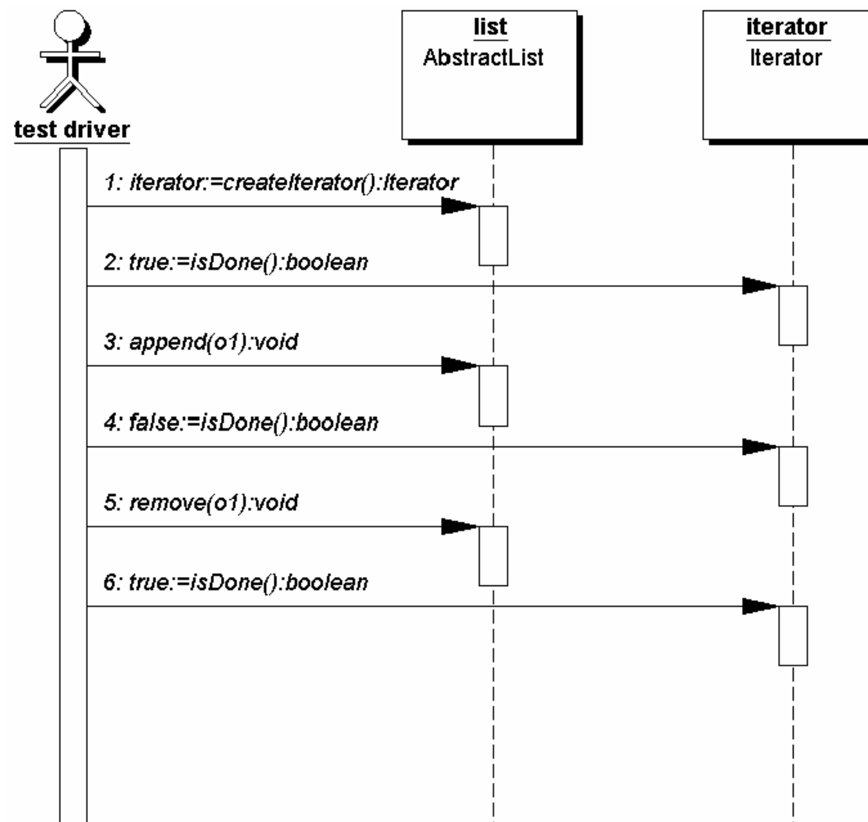

Abbildung 25: Iterator Pattern – Testfall V

das Ausnahmeverhalten des Iterators überprüft. Die übrigen drei Methodenaufrufe testen die Funktionalität der Methode `first`.

Abbildung 25 schließlich zeigt ein testbares Sequenzdiagramm, das testet, ob der Iterator robust im Sinne von [Gamma+95] ist (in diesem Fall soll er es nicht sein, d.h. er soll Änderungen des zugrunde liegenden Containers berücksichtigen).

5.2 Observer

Das Observer (oder auch Publish-Subscribe) Pattern wird in Situationen verwendet, bei denen eine Menge von Objekten, die „Observer", an Zustandsänderungen eines Objektes, dem „Subject", interessiert sind. Das Subject verwaltet dabei eine Liste von Observern, die es dann ggf. über Zustandsänderungen oder andere Dinge informiert. Der Liste von Observern werden Observer mittels der Methoden `attach` und `detach` (s. Abbildung 26[43]) im Interface `Subject` hinzugefügt bzw. wieder daraus entfernt. Das betreffende `Observer`-Objekt wird dabei jeweils als Parameter übergeben.

---

[43] Das in [Gamma+95] für das Observer Pattern angegebene Klassendiagramm enthält neben den hier gezeigten Interfaces auch konkrete Klassen. Diese sind jedoch weder hilfreich noch notwendig für die Erstellung von allgemein gültigen Testfällen für das Observer Pattern und werden daher hier nicht gezeigt.

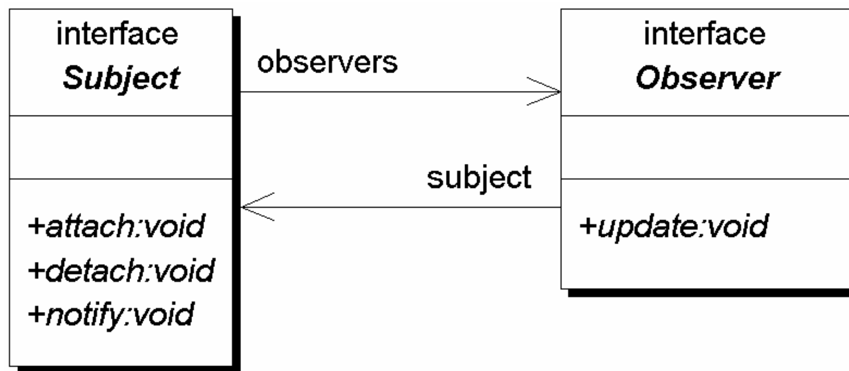

Abbildung 26: Observer Pattern

Für das Benachrichtigen der angemeldeten Observer ist die Methode `notify` im Interface `Subject` verantwortlich. Diese ruft auf allen angemeldeten Observern deren Methode `update` auf.

Das testbare Sequenzdiagramm in Abbildung 27 testet die Methoden des Interfaces `Subject` bei keinem bzw. einem angemeldeten Observer. Hierbei wird auch getestet, ob die Methode `attach` eine

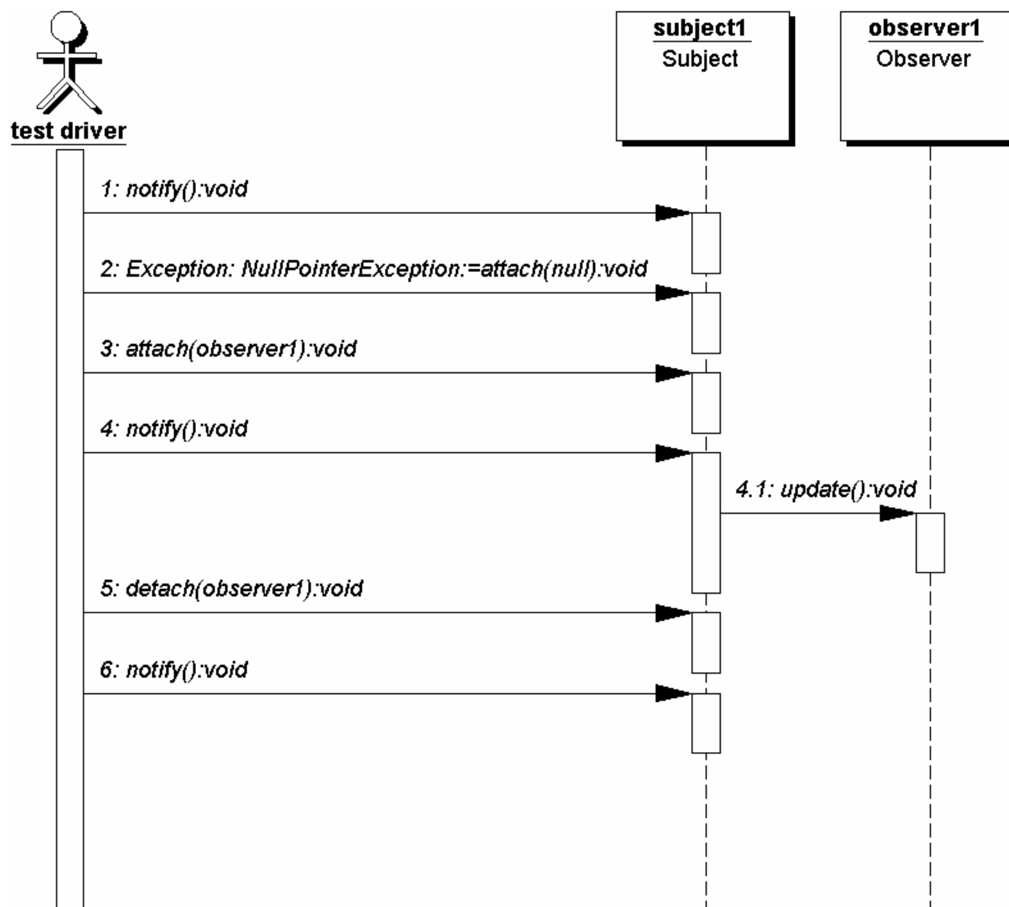

Abbildung 27: Observer Pattern – Testfall I

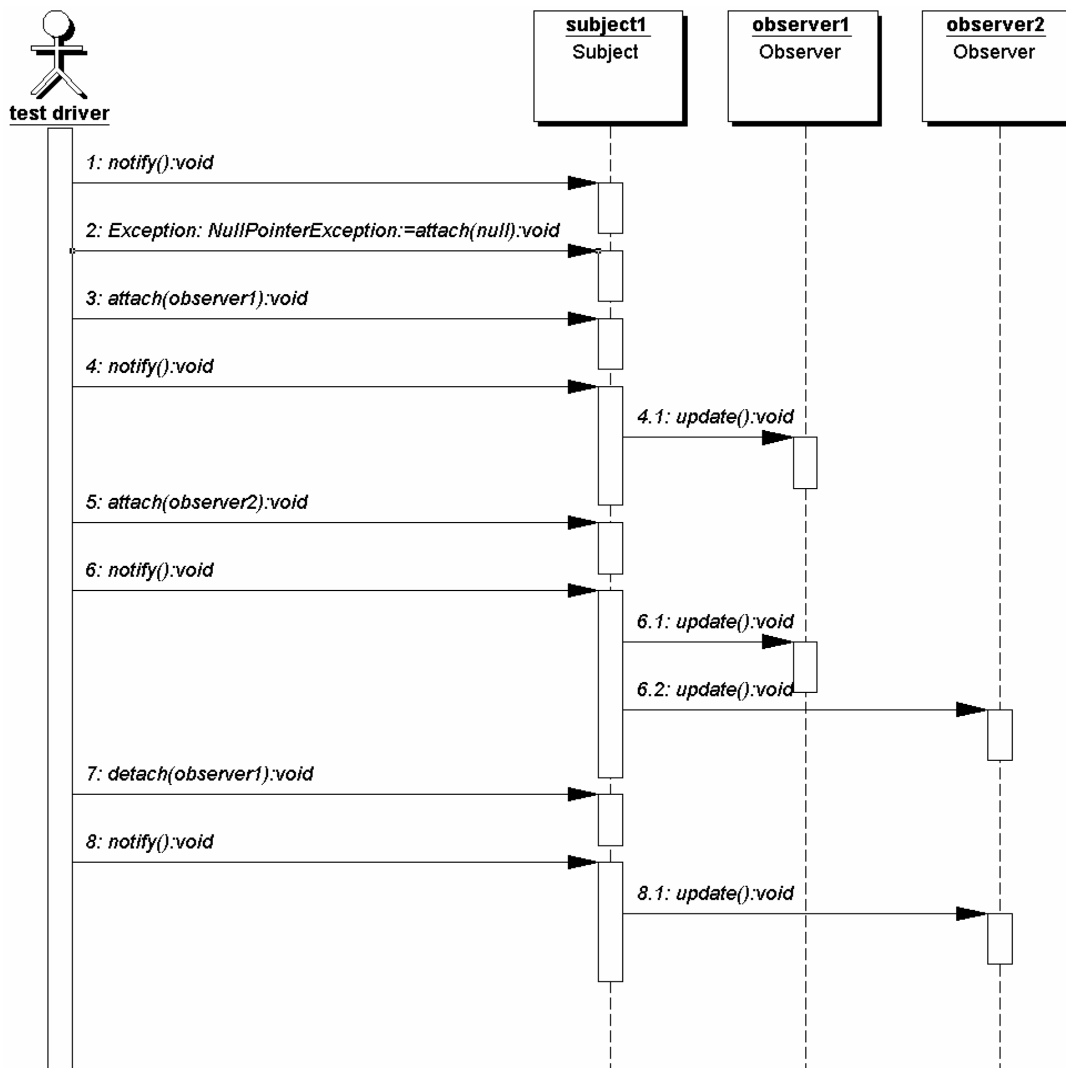

Abbildung 28: Observer Pattern – Testfall II

`NullPointerException` wirft, falls ihr als Argument `null` übergeben wird.

Das in Abbildung 28 dargestellte testbare Sequenzdiagramm ist eine Verfeinerung des Sequenzdiagramms aus Abbildung 27 und testet die Interaktion zwischen einem `Subject`-Objekt und zwei `Observer`-Objekten.

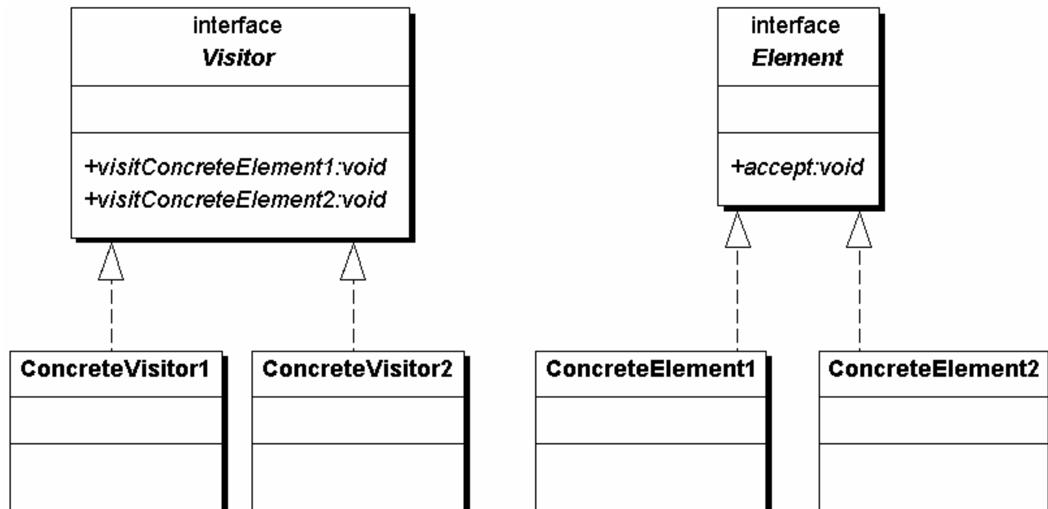

Abbildung 29: Visitor Pattern

5.3 Visitor

Das Visitor Pattern ist durch zwei Interfaces gekennzeichnet: `Visitor` und `Element` (s. Abbildung 29[44]). Mit Hilfe dieser beiden Interfaces lässt sich prinzipiell Double-Dispatch realisieren. Normalerweise ist bei Programmiersprachen wie Java oder C++ nur die Methodensignatur sowie das Objekt, auf dem die Methode aufgerufen wird von Belang, wenn zu entscheiden ist, welche Methode tatsächlich ausgeführt werden soll (Single-Dispatch). Beim Double-Dispatch hingegen hängt diese Entscheidung nicht von der Methodensignatur und einem Objekt, sondern von der Methodensignatur und zwei Objekten ab.

Erreicht wird dieser Effekt einerseits durch die Methode `accept` im Interface `Element`, der als Argument ein Objekt vom Typ `Visitor` übergeben wird. Die konkreten Elemente (die Klassen, die das Interface `Element` implementieren) rufen in ihrer Methode `accept` die für ihre Klasse zuständige Methode auf dem übergebenen Visitor auf (z.B. `visitConcreteElement1`). Dies impliziert, dass es zwar nur eine Methode `accept` im Interface `Element` gibt, jedoch üblicherweise ebenso viele `visit`-Methoden im Interface `Visitor`, wie es konkrete Elemente gibt.

[44] Anders als beim Observer Pattern (s.o.) kann beim Visitor Pattern nicht auf die konkreten Klassen verzichtet werden, da sie für die Formulierung der Testfälle bekannt sein müssen. Im Falle des Visitor Patterns führt dies daher zu zusätzlichem Anpassungsaufwand, wenn die hier angegebenen testbaren Sequenzdiagramme für das Testen einer konkreten Implementierung verwendet werden sollen.

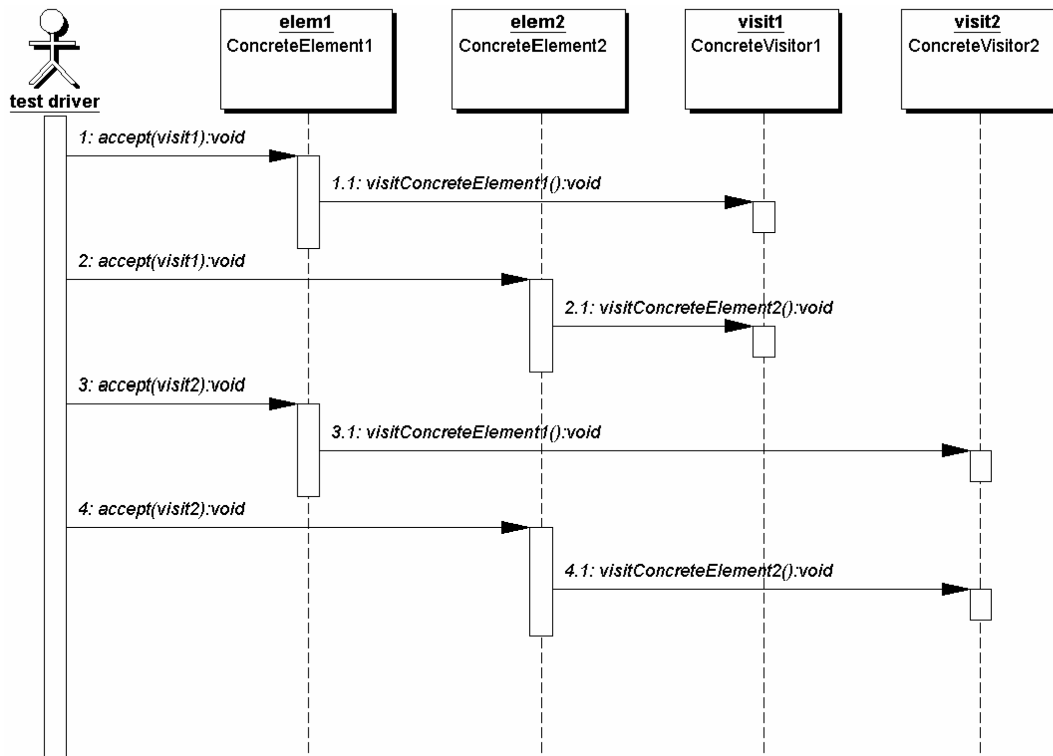

Abbildung 30: Visitor Pattern – Testfall I

Abbildung 30 zeigt ein testbares Sequenzdiagramm, das eine typische Methodenaufruffolge bei Verwendung des Visitor Patterns darstellt und testet.

5.4 Singleton

Das Singleton Pattern wird eingesetzt, wenn sichergestellt werden soll, dass von einer Klasse nur genau eine Instanz existiert, auf die jedoch global zugegriffen werden kann. Während ersteres dadurch erreicht wird, dass alle verfügbaren Konstruktoren `private` bzw. `protected` sind (d.h. nicht von außerhalb der Klasse selbst bzw. ihrer Erben aufrufbar), wird letzteres meist mittels einer statischen Methode `getInstance` im Singleton realisiert, die für das Erstellen der einen Instanz zuständig ist und diese zurückgibt (s. Abbildung 31).

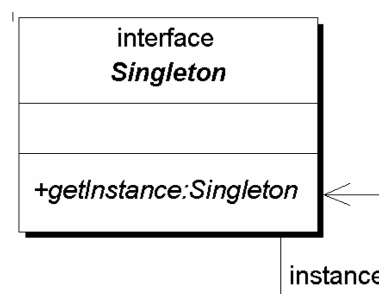

Abbildung 31: Singleton Pattern

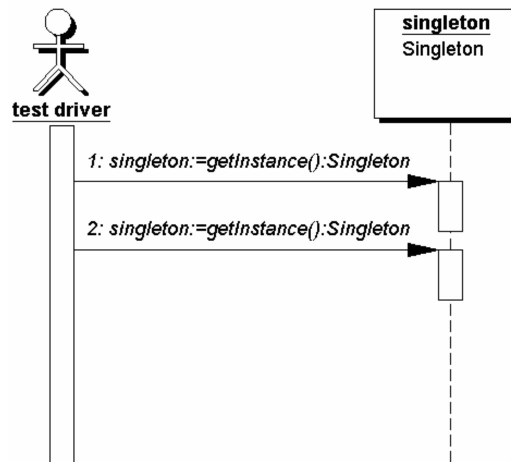

Abbildung 32: Singleton Pattern – Testfall I

In dem in Abbildung 32 dargestellten testbaren Sequenzdiagramm wird die statische Methode `getInstance` zweimal hintereinander aufgerufen und dabei überprüft, ob bei beiden Methodenaufrufen dasselbe Objekt zurückgegeben wird.

5.5 Bewertung

Betrachtet man die vier in den letzten Abschnitten untersuchten Design Patterns und die für sie erstellten testbaren Sequenzdiagramme, so fällt auf, dass für das Iterator Pattern mehr Sequenzdiagramme erstellt wurden als für die anderen drei Patterns zusammen. Der Grund hierfür liegt jedoch nicht etwa darin, dass sich die Patterns in unterschiedlichem Maße für das Testen auf der Basis von Sequenzdiagrammen eignen, sondern offensichtlich darin, dass sich die Patterns in unterschiedlichem Maße in Sourcecode niederschlagen.

So besteht der Inhalt des Iterator Patterns aus einer sehr implementierungsnahen Beschreibung von vier bzw. fünf Methoden inklusive praktisch vollständiger Beschreibung der Funktionalität und der Ausnahmebehandlung im Falle einer fehlerhaften Verwendung. Das von außen sichtbare Verhalten eines Iterators ist prinzipiell unabhängig von der Anwendung, in der er verwendet wird. Es lassen sich daher auch ohne Kenntnis der Anwendung, in der das Iterator Pattern verwendet werden soll, eine ganze Reihe von Tests spezifizieren, die für jede Implementierung des Patterns sinnvoll sind.

Anders stellt sich der Sachverhalt beim Observer und beim Visitor Pattern dar. Ihr Nutzen besteht vielmehr darin zu beschreiben, auf welche Art und Weise die in Form von Methoden zu implementierende Funktionalität in bestimmten Anwendungssituationen auf verschiedene Klassen aufgeteilt werden kann und wie diese Methoden interagieren. Die Semantik der Kern-Methoden dieser beiden Patterns (`update` beim Observer und `visit` beim Visitor Pattern) ist jedoch vollständig abhängig von der Anwendung, in der die Patterns verwendet werden, so dass in

Abhängigkeit der jeweiligen Anwendung weitere Testfälle erstellt werden müssen. Ein weiterer Unterschied zum Iterator Pattern besteht darin, dass das Observer und das Visitor Pattern von sich aus (fast) keine Ausnahmebehandlung erfordern.

Ein essentieller Bestandteil des Visitor Patterns – nämlich die Anforderung, dass ein Visitor normalerweise nicht nur einzelne Objekte, sondern eine ganze Objektstruktur in einer bestimmten Reihenfolge „besuchen" muss – lässt sich ebenfalls nur für konkrete Implementierungen testen. Dies liegt zum einen daran, dass es sich um beliebige Objektstrukturen mit unterschiedlichen Traversierungsalgorithmen handeln kann und zum anderen daran, dass das Visitor Pattern nicht festlegt, wer (d.h. welche Klasse bzw. Rolle) für das Traversieren der Struktur verantwortlich ist.

Das Singleton Pattern schließlich definiert überhaupt nur zwei Anforderungen, die sich theoretisch prüfen lassen. Zum einen muss ein Singleton sicherstellen, dass zur Laufzeit genau eine Instanz des Singletons existiert (und keine weiteren erstellt werden können[45]) und zum anderen, dass die Methode `getInstance` des Singletons auch bei mehreren Aufrufen immer genau diese eine Instanz zurückliefert. Während das in Abbildung 32 dargestellte Sequenzdiagramm darauf abzielt, die zweite der beiden Anforderungen zu überprüfen, lässt sich die erste Anforderung in dem Sinne nicht testen, da es hierbei vielmehr darum geht, die *Abwesenheit* von Funktionalität[46] zu prüfen. Zur Prüfung dieser Anforderung bietet sich daher eher ein Review an.

In [Gamma+95] werden die vorgestellten Design Patterns in drei Kategorien eingeteilt: *Creational*[47], *Structural*[48] und *Behavioral*[49] Patterns. Es liegt nahe zu vermuten, dass Behavioral Patterns am meisten Informationen enthalten, die zu konkretem testbaren Sourcecode führen, da es ja die identifizierende Eigenschaft dieser Patterns ist, dass sie Verhalten beschreiben. Es kommt jedoch auch darauf an, wie konkret das Verhalten letztlich beschrieben ist und wie viel Interpretationsspielraum

---

[45] Die Erfüllung dieser Anforderung ist im Kontext der im Rahmen dieser Arbeit betrachteten Anwendungen vergleichsweise trivial. Bei parallelen bzw. verteilten Anwendungen kann es jedoch aufgrund von gleichzeitig ausgeführten Instanzen der Methode `getInstance` oder durch mehrere beteiligte Virtual Machines erhebliche Komplikationen geben.

[46] In diesem Falle die Möglichkeit, weitere Instanzen des Singletons aus anderen Klassen heraus zu erstellen.

[47] Creational Patterns beschreiben verschiedene Möglichkeiten der Objekterzeugung.

[48] Structural Patterns beschäftigen sich damit, auf welche Art und Weise man einzelne Objekte innerhalb größerer Strukturen organisieren kann, so dass diese (effektiv) interagieren können.

[49] Behavioral Patterns beschäftigen sich mit Algorithmen, der Zuweisung von Verantwortlichkeiten zwischen Objekten und damit, wie Objekte untereinander kommunizieren können.

besteht. So enthält die Beschreibung des Visitor Patterns offensichtlich erheblich weniger konkret testbare Information als die Beschreibung des Iterator Patterns (beides sind Behavioral Patterns).

Auf der anderen Seite gibt es aber auch Design Patterns, die nicht zu den Behavioral Patterns gehören, aber dennoch sehr exakt ein bestimmtes Verhalten beschreiben, dass sich durch testbare Sequenzdiagramme ähnlich testen ließe wie das Verhalten des Iterator Patterns. Als konkrete Beispiele lassen sich hier das Prototype Pattern (Creational) und das Composite Pattern (Structural) nennen.

Insgesamt fällt auf, dass viele der erstellten testbaren Sequenzdiagramme jeweils mehrere Aspekte der Implementierung testen. Dies steht im scheinbaren Widerspruch zur Testfallerstellung für zum Beispiel JUnit, bei der man i.A. versucht (aus implementierungstechnischen Gründen häufig erfolglos), möglichst nur genau einen Aspekt zu testen, um die gegebenenfalls notwendige Fehlerfindung zu erleichtern. Dieses Problem stellt sich jedoch bei der Verwendung von SeDiTeC aufgrund der detaillierteren Testresultate (Beobachtung der relevanten einzelnen Methodenaufrufe) so nicht.

Wie eingangs des Kapitels bereits erwähnt wurde, muss noch die fehlende Objektinitialisierung vorgenommen werden, wenn die hier aufgeführten Sequenzdiagramme zum Testen konkreter Design Pattern Implementierungen verwendet werden sollen. Diese ist direkt abhängig von der jeweiligen Implementierung und kann daher nicht allgemein für die einzelnen Design Patterns erfolgen. Ist dieser vergleichsweise niedrige Aufwand jedoch erbracht, lassen sich im Falle von Patterns wie dem Iterator Pattern bereits eine Reihe von sinnvollen Tests effizient durchführen. Auch im Falle der anderen hier betrachteten Patterns stehen dann bereits Tests zur Verfügung, die jedoch teilweise je nach Anwendungskontext sinnvollerweise noch zu verfeinern sind, um die anwendungsabhängige Semantik zu ergänzen. Dieses Problem wird dadurch verschärft, dass bei manchen Design Patterns keine Interfaces zur Erstellung der testbaren Sequenzdiagramme verwendet werden können, da die Methodensequenzen bzw. die aufgerufenen Methoden direkt abhängig vom dynamischen Typ der beteiligten Objekte sind (z.B. beim Visitor Pattern). Abhilfe könnte hier die Verwendung von Typvariablen schaffen, die jedoch nicht Bestandteil der UML sind.

Ein weiterer großer Vorteil der Ergänzung der Design Pattern Beschreibungen um testbare Sequenzdiagramme besteht darin, dass der Inhalt der Patterns präziser und vor allem auch anschaulicher definiert ist, die Interpretationsspielräume werden somit kleiner. Insbesondere gilt das für die oftmals etwas vernachlässigte Ausnahmebehandlung, die aber nicht nur für die Robustheit einer Anwendung, sondern gerade auch für die Fehlerfindung wertvolle Dienste leisten kann.

6 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es zu untersuchen, welche Funktionalität ein auf Sequenzdiagrammen basierendes Testwerkzeug im Detail aufweisen muss, um sinnvoll in der Praxis eingesetzt werden zu können, sowie in der prototypischen Implementierung eines diese Funktionalität bietenden Werkzeugs.

Hierzu wurden eine Reihe von Konzepten entwickelt, die einen tatsächlichen Mehrwert im Testbereich darstellen. Grundlegend ist hier das Konzept der Eingabesequenzdiagramme und die Definition der Verfeinerung der selbigen, die eine flexible Testspezifikation und aussagekräftige Testergebnisevaluierung ermöglichen. Dadurch, dass das in dieser Arbeit vorgestellte Konzept zur Testspezifikation UML Sequenzdiagramme mit einbezieht, die während des Systemdesign entstehen, lässt sich der Testprozess leichter in den Systementwicklungsprozess integrieren.

Von großer praktischer Bedeutung ist das vorgestellte Teststubkonzept, das es erlaubt, spezifizierte aber noch nicht implementierte Klassen und deren Instanzen realitätsnah zu simulieren. Hieraus ergeben sich einige wesentliche Vorteile für den Testprozess:

- Mit dem Testen kann gleich zu Beginn der Implementierung begonnen werden, ohne dass dafür Implementierungsaufwand für Teststubs oder -treiber anfiele.
- Eine Implementierungsreihenfolge für Komponenten des Systems wird nur noch in erheblich geringerem Maße durch den Testprozess bestimmt.
- Durch praktisch aufwandsfreies Ein- und Ausschalten der Teststubgenerierung für bestimmte Klassen kann die Fehlersuche erheblich vereinfacht werden.

Sehr viel versprechend erscheint auch der Ansatz, die Beschreibungen von Design Patterns um testbare Sequenzdiagramme zu erweitern, da dies sowohl das Testen von Design Pattern Implementierungen erheblich vereinfacht als auch die Verständlichkeit und Genauigkeit der Beschreibungen erhöht.

Die prinzipielle Umsetzbarkeit der vorgestellten Konzepte wurde anhand der Implementierung des Testwerkzeugs SeDiTeC belegt. Der Einsatz von SeDiTeC in Projekten hat gezeigt, dass gerade Problemstellungen im Bereich des Integrationstests mit den vorgestellten Konzepten verglichen mit anderen Testkonzepten gut zu lösen sind.

Während der Entwicklung und Nutzung von SeDiTeC haben sich aber auch Fragestellungen und Anregungen ergeben, die in zukünftigen Arbeiten weiter zu bearbeiten sind.

Verfeinerungsbewertung: Momentan wird ein Testfall, bei dem das beobachtete Sequenzdiagramm eine (echte) Verfeinerung des

Eingabesequenzdiagramms ist, als fehlgeschlagen bewertet, was in einer Reihe von Fällen unerwünscht ist. Als Beispiel sei hier das testbare Sequenzdiagramm für das Singleton Pattern genannt (s. Abschnitt 5.4). Bei einer Testausführung wird der erste Methodenaufruf üblicherweise dazu führen, dass ein Konstruktor auf dem Singleton aufgerufen wird, was in diesem Fall aber nicht zur Folge haben sollte, dass der Testlauf als fehlerhaft bewertet wird.

Zu untersuchen wäre, ob es hilfreich ist, für einzelne Testfälle festlegen zu können, ob beobachtete Verfeinerungen als Fehler gewertet werden sollen oder nicht. Das birgt jedoch die Gefahr, dass unerwünschte Verfeinerungen (sprich: Seiteneffekte) u.U. nicht mehr als Fehler erkannt werden würden.

Nachbedingungen und Exceptions: In der aktuellen Version von SeDiTeC werden Nachbedingungen einer Methode nur dann überprüft, wenn die Methode durch eine `return`-Anweisung beendet wird und nicht durch das Werfen einer Exception. Dies ist zwar in realen Anwendungssituationen häufig die praktikablere Lösung, entspricht aber streng genommen nicht der Idee einer Nachbedingung. Es stellt sich die Frage, ob man nicht auch hier eine Wahlmöglichkeit schaffen sollte, um einzelne Exceptions entweder nur nach `return`-Anweisungen zu prüfen oder generell bei Verlassen der jeweiligen Methode. Die auf den ersten Blick nahe liegende Alternative, Nachbedingungen (ohne Wahlmöglichkeit) *immer* nach dem Verlassen einer Methode zu überprüfen, würde aber wohl eher dazu führen, dass Nachbedingungen nur sporadisch eingesetzt werden.

Passiver Betrieb von SeDiTeC: Die Ausführung eines Testfalls unter SeDiTeC beginnt immer mit einem Methodenaufruf, der vom Testtreiber-Aktor initiiert wird, d.h. SeDiTeC nimmt eine aktive Rolle von Beginn der Testausführung an ein. Bei Tests, die auf sehr komplexen Systemzuständen beruhen, könnte es sinnvoll sein, dass SeDiTeC beispielsweise erst bei Aufruf einer bestimmten Methode („sowie ein Eingabesequenzdiagramm betreten wird“) aktiv wird und weitere Methodenaufrufe initiiert.

Verteilte Anwendungen: Momentan unterstützt SeDiTeC das Testen von verteilten Anwendungen nicht in besonderer Weise. Gerade durch den eben beschriebenen passiven Betrieb von SeDiTeC würden sich hier jedoch sehr interessante Möglichkeiten ergeben, wenn man beispielsweise auf einem Client SeDiTeC aktiv und auf einem oder mehreren Servern passiv laufen lassen würde.

Einzelne Java Virtual Machine: SeDiTeC startet das zu testende Programm in der gleichen Java Virtual Machine (JVM), in der es selbst läuft. Insbesondere wird daher nicht für die Ausführung jedes Eingabesequenzdiagramms innerhalb eines Testlaufs eine neue JVM gestartet. Einerseits hilft dies zwar bei der Erfüllung der Anforderung, dass Testläufe möglichst schnell ausgeführt werden können, andererseits kann dies jedoch auch zu Problemen führen, falls Änderungen an der

Laufzeitumgebung durchgeführt werden, die dann zwischen zwei Testfällen nicht wieder rückgängig gemacht werden.

Anhang

Anhang A – Methode `executeMethodCalls`

Die Methode `executeMethodCalls` (s. Abbildung 33) ist Bestandteil eines jeden von SeDiTeC generierten Teststubs. Sie ist für das Aufrufen weiterer Methoden durch die Methode eines Teststubs zuständig und stellt damit den Kern des „intelligenten" Verhaltens der SeDiTeC Teststubs dar. Die hier gezeigte Version enthält der Übersichtlichkeit halber keine Ausnahmebehandlung (tatsächlich enthält sie Anweisungen zur Behandlung von sieben verschiedenen Ausnahmetypen).

Der Parameter vom Typ `MethodCallData` ist ein Identifikator für die aktive Methodeninstanz. Mittels dieses Parameters wird zur Laufzeit ermittelt, ob noch weitere Methoden aufzurufen sind (Zeile 4). Ist dies der Fall, so wird anschließend die Information über die aufzurufende Funktion in Form eines Objekts vom Typ `Invocation` abgefragt (Zeile 6).

In den Zeilen 7 und 8 wird der Identifikator für die aufzurufende Methode sowie der Typ des Objekts, auf dem diese Methode aufgerufen werden soll, ermittelt. Der `if`-Block dient dann letztlich der Unterscheidung, ob es sich um einen Konstruktor- oder einen „normalen" Methodenaufruf handelt. Im jeweiligen Zweig erfolgt dann der eigentliche Methodenaufruf.

```
1 private static void executeMethodCalls
                                (MethodCallData mcd)
2 {
3   TestCenter tc = TestCenter.getTestCenter();
4   while (tc.hasMoreInvocations(mcd))
5   {
6     Invocation invoc = tc.getNextInvocation(mcd);
7     MethodCallData mcdToBeCalled =
                            invoc.getMethodCallData();
8     Class typeToInvokeOn = null;
9     typeToInvokeOn = invoc.getTypeToInvokeOn();
10    if (invoc.isConstructorCall())
11    {
12      Constructor constructor =
typeToInvokeOn.getConstructor(invoc.getParameterTypes());
13      constructor.newInstance(invoc.getParameters());
14    }
15    else
16    {
17      Method method = typeToInvokeOn.getMethod
   (invoc.getMethodName(), invoc.getParameterTypes());
18      Object invokedOn = invoc.getObject();
19      method.invoke(invokedOn, invoc.getParameters());
20    }
21  }
22}
```

Abbildung 33: Methode `executeMethodCalls`

Anhang B – Instrumentierung des Sourcecodes

Anhand der Methode `addBook` aus der Klasse `Library` (s. Abbildung 34) soll verdeutlicht werden, auf welche Art und Weise SeDiTeC Sourcecode instrumentiert. Die Methode `addBook` fügt den ihr übergebenen Parameter vom Typ `Book` in die Liste von Büchern (`bookList`) ein. Sie verfügt sowohl über eine Vor- als auch über eine Nachbedingung, die aus den Javadoc-Kommentaren ersichtlich sind.

Eine von SeDiTeC instrumentierte Methode besteht aus bis zu fünf Sektionen, die die folgenden Aufgaben wahrnehmen:

- Initialisierung
- Logging des Methodenaufrufbeginns an sich sowie der eventuell erhaltenen Parameter
- Überprüfung der Vorbedingung (falls vorhanden)
- Ausführung der ursprünglichen Anweisungen
- Logging des Methodenaufrufendes sowie eventuell Rückgabe eines Rückgabewertes und Überprüfung der Nachbedingung (falls vorhanden)

Unterschiede zu den in Abschnitt 4.2.5 beschriebenen Sektionen der Methode eines Teststubs bestehen zum einen darin, dass die Sektion fehlt, die für das (künstliche) Erstellen und Werfen von Exceptions zuständig war. Zum anderen wird die Sektion, die für das Aufrufen anderer Methoden zuständig war, ersetzt durch die ursprünglichen Anweisungen der instrumentierten Methode.

Abbildung 35 zeigt die Methode `addBook` nach der Instrumentierung. Ihre ursprüngliche Anweisung befindet sich in Zeile 17 in einem `try`-Block, der auch das Logging des Methodenaufrufendes und die Prüfung der Nachbedingung enthält. Wie bereits erwähnt wurde, führt dies dazu, dass in dem Fall, dass der ursprüngliche Code der instrumentierten Methode eine Exception wirft, die Nachbedingung nicht überprüft wird.

Ein weiterer Unterschied zwischen einer instrumentierten Methode und der Methode eines Teststubs besteht darin, dass die Sektion, die für das Logging des Methodenaufrufendes und die Prüfung der Nachbedingung

```
/**
 * @preconditions book != null
 * @postconditions findBook(book.getISBN()) == book
 */
public void addBook(Book book)
{
  bookList.add(book);
}
```

Abbildung 34: Methode `Library.addBook`

zuständig ist, nicht notwendigerweise genau einmal vorkommt. Vielmehr findet sie sich an allen Stellen, an denen sich in der ursprünglichen Methode `return`-Anweisungen befanden sowie am Ende der ursprünglichen Anweisungen.

```
1 public void addBook(Book book)
2 {
3   // initialization
4   String zUniqueMethodName = "<oiref:java#Member
              #lib.Library#addBook#(#lib.Book#)#:oiref>";
5   TestCenter tc = TestCenter.getTestCenter();
6   // check call order and parameters
7   Object params = new Object[1];
8   params[0] = book;
9   MethodCallDataWrapper mcdWrapper =
      tc.acceptInstrumentedCall(this,
                          zUniqueMethodName, params);
10  // precondition
11  if (!(book != null))
12  {
13    throw new PreconditionError
                        ("Condition: book != null");
14  }
15  try
16  { // original method body
17    bookList.add(book);
18    tc.acceptInstrumentedReturn(zUniqueMethodName,
                                  mcdWrapper, null);
19    // postcondition
20    if (!(findBook(book.getISBN()) == book))
21    {
22      throw new PostconditionError
         ("Condition: findBook(book.getISBN()) == book");
23    }
24    return;
25  }
26  // exception reporting and rethrowing
27  catch (Throwable t)
28  {
29    tc.acceptThrowable(mcdWrapper, t);
30    if (t instanceof Error)
31    {
32      throw(Error)t;
33    }
34    else
35    {
36      throw(RuntimeException)t;
37    }
38  }
39}
```

Abbildung 35: Methode `Library.addBook` (instrumentiert)

Literaturverzeichnis

[Abdurazik+00] Aynur Abdurazik, Jeff Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", *Proceedings of the Third International Conference on the Unified Modeling Language*, 2000, S. 383-395

[ASF03a] Apache Software Foundation, *Apache Ant*, http://ant.apache.org/, 2003 (a)

[ASF03b] Apache Software Foundation, *Mock Objects*, http://www.mockobjects.com/, 2003 (b)

[Beck00] Kent Beck, *Extreme Programming Explained*, Addison-Wesley, Reading, 2000

[Beizer95] Boris Beizer, *Black-Box Testing - Techniques for Functional Testing of Software and Systems* , John Wiley & Sons, New York, 1995

[Binder00] Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, Reading, 2000

[Binder94] Robert V. Binder, "Design for Testability in Object-Oriented Systems", *Communications of the ACM*, Vol. 37, Nr. 9, 1994, S. 87-101

[Briand+01] Lionel Briand, Yvan Labiche, "A UML-Based Approach to System Testing", *Software and Systems Modeling (SoSyM)*, Vol. 1, Nr. 1, 2001, S. 10-42

[Briand+02] Lionel Briand, Yvan Labiche, *Automating Impact Analysis and Regression Test Selection Based on UML Designs*, Montreal, 2002

[Coad+99] Peter Coad, Eric Lefebvre, Jeff De Luca, *Java Modeling in Color with UML: Enterprise Components and Process*, Prentice Hall, Upper Saddle River, 1999

[DSEWiki03] Deutsches Software Entwickler Wiki, *Drei Test Axiome*, http://www.wikiservice.at/dse/wiki.cgi?DreiTestAxiome, 2003

[Eckel03] Bruce Eckel, *Thinking in Java*, Prentice Hall, Upper Saddle River, 2003

[ETSI03] European Telecommunications Standards Institute, *Tree and Tabular Combined Notation 3*, http://www.etsi.org/, 2003

[Firesmith93] Donald G. Firesmith, *Testing Object-Oriented Systems*, Prentice Hall, Englewood Cliffs, 1993

[Fowler+01] Martin Fowler, Jim Highsmith, "The Agile Manifesto", *Software Development Magazine*, Vol. 9, Nr. 8, 2001, S. 28-32

[Fowler01] Martin Fowler, "Is Design Dead?", *Software Development Magazine*, Vol. 9, Nr. 4, 2001, S. 42-46

[Fraikin+00] Falk Fraikin, Thomas Leonhardt, *Top-Down Testen auf der Basis von Sequenzdiagrammen*, http://www.pi.informatik.tu-darmstadt.de/studarb/FraikinLeonhardt/Welcome.html, 2000

[Fraikin+01] Falk Fraikin, „SeDiTeC - Testen auf der Basis von Sequenzdiagrammen“*, GI Softwaretechnik-Trends,* Vol. 21, Nr. 3, 2001, S. 8

[Fraikin+02a] Falk Fraikin, Thomas Leonhardt “SeDiTeC - Testing Based on Sequence Diagrams” *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, Edinburgh, 2002, S. 261-266

[Fraikin+02b] Falk Fraikin, gemeinsam mit Michael Becker, Stefan Jungmayr, Moritz Schnitzler, Andreas Schoolmann, Mario Winter “Test von Komponenten” *, GI Softwaretechnik-Trends,* Vol. 22, Nr. 3, 2002, S. 12-15

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, 1995

[Gamma+98] Erich Gamma, Kent Beck, "Test Infected: Programmers Love Writing Tests", *Java Report*, Vol. 3, Nr. 7, 1998, S. 37-50

[Gamma+99] Erich Gamma, Kent Beck, "JUnit: A Cook's Tour", *Java Report*, Vol. 4, Nr. 5, 1999, S. 27-38

[Gelperin+88] David Gelperin, Bill Hetzel, "The Growth of Software Testing", *Communications of the ACM*, Vol. 31, Nr. 6, 1988, S. 687-695

[Girschick02] Martin Girschick, *UMLDiff - Erkennung und Analyse von Unterschieden in Klassendiagrammen und Sequenzdiagrammen*, http://www.pi.informatik.tu-darmstadt.de/studarb/girschick/Welcome.html, 2002

[Hatton98] Les Hatton, "Does OO Sync with How We Think?", *IEEE Software*, Vol. 15, Nr. 3, 1998, S. 46-54

[I-Logix03] I-Logix, *Rhapsody TestConductor*, http://www.ilogix.com, 2003

[ITU99] ITU-T Recommendation Z.120, *Message Sequence Chart (MSC)*, Genf, 1999

[Jacobson+92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard, *Object-Oriented Software Engineering*, Addison-Wesley, Reading, 1992

[Jacobson+98] Ivar Jacobson, Grady Booch, James Rumbaugh, *The Unified Software Development Process*, Addsion-Wesley, Reading, 1998

[Jacobson+99] Ivar Jacobson, Grady Booch, James Rumbaugh, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, 1999

[Kaner97] Cem Kaner, "Pitfalls and Strategies in Automated Testing", *IEEE Computer*, Vol. 30, Nr. 4, 1997, S. 114-116

[Kirani94] Shekhar H., Tsai, Wei-Tek Kirani, "Method Sequence Specification and Verification of Classes", *Journal of Object-Oriented Programming*, Vol. 7, Nr. 6, 1994, S. 28-38

[Liggesmeyer02] Peter Liggesmeyer, *Software-Qualität: Testen, Analysieren und Verifizieren von Software*, Spektrum Akademischer Verlag, Berlin, 2002

[Link02] Johannes Link, *Unit Tests mit Java: Der Test-First-Ansatz*, dpunkt.verlag, Heidelberg, 2002

[Mackinnon+01] Tim Mackinnon, Steve Freeman, Philip Craig, "Endo-Testing: Unit Testing with Mock Objects," *Extreme Programming Examined*, Addison-Wesley: 2001, 287-301.

[McGregor+94] John D. McGregor, Timothy D. Korson, "Integrated Object-Oriented Testing and Development Processes", *Communications of the ACM*, Vol. 37, Nr. 9, 1994, S. 59-77

[Meyer92] Bertrand Meyer, *Eiffel - The Language*, Prentice Hall, Englewood Cliffs, 1992

[Meyer97] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, Upper Saddle River, 1997

[Myers79] Glenford J. Myers, *The Art of Software Testing*, Wiley, New York, 1979

[Müller98] Uwe Müller, "Prüfen und Testen von Software in Deutschland", *Softwaretechnik-Trends*, Vol. 18, Nr. 2, 1998, S. 12-14

[OMG03] Object Management Group, *Unified Modeling Language*, http://www.uml.org/, 2003

[Palmer+02] Stephen R. Palmer, John M. Felsing, *A Practical Guide to Feature Driven Development*, Prentice Hall, Upper Saddle River, 2002

[Parnas71] David L. Parnas, *Information Distribution Aspects of Design Methodology*, 1971

[Perry+90] Dewayne Perry, Gail Kaiser, "Adequate Testing and Object-Oriented Programming", *Journal of Object-Oriented Programming*, Vol. 3, Nr. 2, 1990, S. 13-19

[Rational03] Rational Software, *Rational Rose*, http://www.rational.com, 2003

[Rosenberg+01] Doug Rosenberg, Kendall Scott, "Sequence Diagrams: One Step at a Time", *Software Development Magazine*, Vol. 9, Nr. 4, 2001, S. 38-47

[Rosenberg+99] Doug Rosenberg, Kendall Scott, *Use Case Driven Object Modeling with UML*, Addison-Wesley, Reading, 1999

[Rossnagel02] Heiko Rossnagel, *Automatisiertes Roundtrip Engineering zwischen Sequenzdiagrammen und JUnit Testklassen*, http://www.pi.informatik.tu-darmstadt.de/studarb/rossnagel/Welcome.html, 2002

[Rumbaugh+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, 1991

[Sneed+02] Harry M. Sneed, Mario Winter, *Testen objektorientierter Software*, Hanser, Müchen, 2002

[Software03] Software Quality Engineering Laboratory, *TOTEM*, http://www.sce.carleton.ca/Squall/Totem/, 2003

[Spillner+03] Andreas Spillner, Tilo Linz, *Basiswissen Softwaretest*, dpunkt.verlag, Heidelberg, 2003

[Spillner98] Andreas Spillner, "Four Kinds of Class Modality - Four Kinds of Class Testing?", *Proceedings of the EuroSTAR 1998*, 1998, S. 107-119

[Sun03a] Sun Microsystems, *JSR-014 Adding Generics to the Java Programming Language*, http://www.jcp.org/en/jsr/detail?id=014, 2003 (a)

[Sun03b] Sun Microsystems, *Java Debug Interface*, http://java.sun.com/j2se/1.4.1/docs/guide/jpda/jdi/index.html, 2003 (b)

[Sun03c] Sun Microsystems, *Java Platform Debugger Architecture*, http://java.sun.com/products/jpda/doc/, 2003 (c)

[Togethersoft03] Togethersoft, *Together*, http://www.togethersoft.com, 2003

[U2TP03] U2TP Consortium, *UML Testing Profile*, http://www.fokus.fraunhofer.de/u2tp/, 2003

[Vermeulen+00] Allan Vermeulen, Scott W. Ambler, Greg Baumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, Patrick Thompson, *The Elements of Java Style*, Cambridge University Press, Cambridge, 2000

[Vessey+94] Iris Vessey, Sue A. Conger, "Requirements Specification: Learning Object, Process, and Data Methodologies", *Communications of the ACM*, Vol. 37, Nr. 5, 1994, S. 102-113

[Warmer+99] Jos Warmer, Kleppe Anneke, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, Reading, 1999

[Weyuker86] Elaine J. Weyuker, "Axiomatizing Software Test Data Adequacy", *IEEE Transactions on Software Engineering*, Vol. 12, Nr. 12, 1986, S. 1128-1138

[Weyuker88] Elaine J. Weyuker, "The Evaluation of Program-Based Software Test Data Adequacy Criteria", *Communications of the ACM*, Vol. 31, Nr. 6, 1988, S. 668-675

[Whittaker00] James A. Whittaker, "What Is Software Testing? And Why Is It So Hard?", *IEEE Software*, Vol. 17, Nr. 1, 2000, S. 70-79

[Winter98] Mario Winter, *Managing Object-Oriented Integration and Regression Testing*, München, 1998

[Winter99] Mario Winter, *Qualitätssicherung für objektorientierte Software: Anforderungsermittlung und Test gegen die Anforderungsspezifikation*, dissertation.de, Berlin, 1999

[Wittevrongel03] Jeremiah Wittevrongel, *SCENTOR*, http://sern.ucalgary.ca/~milos/etesting/scentor/index.html, 2003

Lebenslauf

Personaldaten

Name:	Falk Fraikin
Adresse:	Heinrichstr. 27 64283 Darmstadt
Tel.:	06151-294918
E-Mail:	fraikin@informatik.tu-darmstadt.de
Geburtsdatum:	29.9.1972 (Groß-Gerau)

Ausbildung

08/19 – 6/92	**Gymnasium Gernsheim** Abschluss: Abitur
10/94 – 2/00	**Technische Universität Darmstadt** Studium der Wirtschaftsinformatik, Abschluss: Dipl.-Wirtsch.-Inform.

Berufliche Tätigkeit

5/94 – 8/94	**MLP, Heidelberg** Werksstudententätigkeit, Entwicklung von Datenbankanwendungen
6/96 – 2/00	**BBS, Neu-Anspach** Freier Mitarbeiter für Softwareentwicklung und Beratung
11/97 – 3/98	**Mercedes-Benz China Ltd., Hong Kong** Praktikum Abteilung Finance & Controlling
3/00 – heute	**TU Darmstadt, FG Praktische Informatik** Wissenschaftlicher Mitarbeiter, Vorlesungen im Bereich Software Engineering, Software Qualitätssicherung und Programmierung, Betreuung von Diplomarbeiten und Industrie-Projekten

Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades „Dr.-Ing." mit dem Titel „Entwicklungsbegleitendes Testen mittels UML Sequenzdiagrammen" selbstständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, 10. November 2003 Falk Fraikin