Technische Universität Berlin
Fachbereich Informatik

Diplomarbeit

# Entwicklung eines aspektorientierten Frameworks für den objektorientierten Unit-Test

vorgelegt von
Matthias Vösgen
Matrikel-Nr. 206105

Berlin, 21. Juni 2004

# Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, den 22.Juni 2004

---

(Matthias Vösgen)

# Inhaltsverzeichnis

# 1. Einleitung und Überblick

Diese Arbeit untersucht die Einsatzmöglichkeiten aspektorientierter Programmierung, kurz AOP, im Unit-Testen. Im Vergleich zu anderen Konzepten in der Informatik wie der Objektorientierung ist die Aspektorientierung ein neues Konzept, mit dem nicht jeder Informatiker vertraut ist. Auch der Begriff Unit-Test kann nicht vorausgesetzt werden.

Darum wird in der Einleitung der Begriff Unit-Test, so wie er in dieser Arbeit verstanden werden soll, umrissen und eine kurze Einführung in die aspektorientierte Programmierung gegeben. Im Anschluss werden einige andere Beiträge zu diesem Thema vorgestellt. Auf diesen Hintergrundinformationen aufbauend wird zum Abschluss der Einleitung das theoretische und praktische Ziel der Diplomarbeit formuliert.

## 1.1. Unit-Test

### 1.1.1. Bedeutung des Begriffs Unit-Test

Wenn man ein Softwaresystem zufriedenstellend testen möchte, muss man dies auf verschiedenen Granularitätsstufen[1] tun. In einem objektorientierten System beispielsweise testet man auf der niedrigsten Ebene Methoden und Klassen, auf der nächsthöheren testet man die Zusammenarbeit mehrerer zusammengehöriger Klassen, auf darüber liegenden Ebenen die mehrerer Module und auf der höchsten Ebene schließlich das System in einem Systemtest als Ganzes. Sollte es einen externen Auftraggeber geben, kann er das System noch einem Abnahmetest unterziehen.

Unit-Tests befinden sich in dieser Hierarchie auf der niedrigsten Ebene. Sie haben einen sehr kleinen Wirkungsbereich (engl. Scope). Robert Binder grenzt den Scope von Unit-Tests in seinem Werk „Testing Object-Oriented Systems“ [4] folgendermaßen ein: *„The scope of a unit test typically comprises a relatively small executable. In object-oriented programming, an object of a class is the smallest executable unit, but test messages must be sent to a method, so we can speak of method scope testing.“*

In der Praxis bedeutet das Unit-Testen, für jede Methode in einem Modul voneinander unabhängige Testfälle zu schreiben. Jeder Testfall soll, sofern es möglich ist, unabhängig von anderen Testfällen getestet werden. Ein durchlaufener Unit-Test macht mit seinem Ergebnis eine Aussage[2] über eine getestete Methode, nicht über komplexere Zusammenhänge im System. Fehler in einzelnen Methoden können so leicht gefunden werden, da jeder Testfall zu einer Methode gehört, die für das Scheitern verantwortlich ist.

[1] Verfeinerungsebenen: Je niedriger, desto geringer der Abstraktionsgrad und die Anzahl der zur Ebene gehörenden Objekte.

[2] Korrektheit kann man leider nur in einem theoretischen Modell nachweisen. In praktischen Implementierungen erreicht man immer nur einen gewissen Grad der Sicherheit. Siehe dazu z.B. [13].

Auch wenn keine Systemzusammenhänge getestet werden, ist eine Methode einer Klasse immer in eine Umgebung eingebettet, in der sie funktionieren soll und von der Tests selten vollständig abstrahieren können. Ändert sich die Art und Weise des Zusammenwirkens verschiedener Komponenten im System[3], kann es zu Fehlern kommen, die auf einer fehlerhaften Zusammenarbeit der getesteten Methode mit veränderten Komponenten beruhen. Darum empfiehlt es sich, ein Unit-Testframework für ein komplettes System auch nach der Änderung an nur einer Komponente vollständig durchlaufen zu lassen, wobei jede einzelne Methode mit ihrem Unit-Test getestet wird. Das Durchlaufen einer ganzen Suite von Unit-Tests nach jeder Änderung wird als Regressionstest [4] bezeichnet.

Unit-Tests werden vom Entwickler selbst geschrieben. Für die Entwickler ist damit der Zusatznutzen einer Dokumentation verbunden. Tests, die die Funktionalität einer Methode testen, zeigen, wie sie eingesetzt werden kann und soll. Sie helfen sogar dem Entwickler, die Aufgaben seiner Methode besser zu verstehen und die Methode u.U. zu verändern, damit sie von anderen Entwicklern besser als Funktionseinheit verstanden werden kann, die ein genau spezifiziertes Verhalten implementiert. Aus psychologischen Gründen kann es natürlich nachteilig sein, ein System nur von Entwicklern zu testen, die ihren eigenen Code in Frage stellen müssen. Sollte es sich um Anwendungen handeln, deren Versagen finanzielle oder Personen schädigende Konsequenzen hat, reichen die von den Entwicklern geschriebenen Tests nicht aus. Eine separate Test-Abteilung sollte das Programm zusätzlich testen [14].

Die Begriffe Black- und White-Box-Testing, die im Zusammenhang mit dem Testen häufig fallen, werden in dieser Arbeit wie auch in [4] nicht benutzt und stattdessen, wenn es nötig sein sollte, auf die Begriffe responsibility-based- und implementation-based-testing zurückgegriffen, da sie weniger missverständlich sind. Auch z.B. in [3] wird empfohlen, die Begriffe White- und Black-Box-Testing sehr vorsichtig zu verwenden.

Die Unterscheidung ist eine Frage der Testfallerstellung. Beim responsibility-based-testing werden die Testfälle so erstellt, dass alle Aufgaben der Methode exemplarisch getestet werden, beim implementation-based-testing betrachtet man die Struktur der Implementierung und versucht eine gewisse Art der Abdeckung über den Code (Zweig, Pfad, etc.) zu erreichen.

Beim Unit-Test, auf den sich diese Arbeit beschränkt, ist beides sinnvoll. Responsibility-based-testing testet die Methoden mit Testfällen, die durch die Spezifikation bestimmt werden. Das implementation-based-testing liefert Testfälle, die in den Methoden eine definierte Code-Abdeckung erzielen.

### 1.1.2. Herkunft des Begriffs Unit-Test

Historisch gesehen steht der Begriff Unit-Testen nicht isoliert im Raum. Er wurde als Teil eines Softwareentwicklungsprozesses namens Extreme Programming- oder kurz XP [2, 43, 44] Anfang der 90er Jahre eingeführt.

Mit dem Extreme Programming wollte sein Hauptförderer Kent Beck seinerzeit das Software-Engineering revolutionieren. Seine Ideen wurden teils skeptisch ablehnend teils

[3] Z.B. die Art und Anzahl der Parameter, die eine Methode akzeptiert.

begeistert aufgenommen. Die Meinungen sind auch heute noch geteilt, wenn sich die Lager auch beruhigt haben[4]. Die wenigsten halten Extreme Programming noch für eine Wunderwaffe, die die Antwort auf alle Fragen des Software-Engineering ist oder für eine für alle Probleme vollkommen untaugliche Methode. Hinzu kommt, dass die Methode aus vielen Komponenten besteht, die sich häufig unabhängig voneinander einsetzen lassen.

Eine dieser unabhängig einsetzbaren Komponenten ist das Unit-Testen in Verbindung mit dem Regressionstest. Wenn man es von der Gesamtmethode isoliert betrachtet, handelt es sich um einen Testprozess, so wie es sich bei der Paar-Programmierung, einem anderen Teil der XP-Methode, nur um eine Programmier-Technik handelt. Sie lässt sich in jeden anderen Softwareentwicklungsprozess integrieren.

### 1.1.3. Grenzen des Unit-Tests

Die Beschränkungen des Unit-Tests werden schon aus seiner Definition klar. Ein Unit-Test testet nur einzelne Methoden. Fehler, deren Ursache aus größeren Zusammenhänge resultieren, müssen durch andere Techniken gefunden werden.

Testmethoden mit größerem Wirkungsbereich sind [4]:

**Integrationstest** Testet ein komplettes System oder Subsystem von Hard- oder Softwareeinheiten. Die Einheiten im Subsystem sind voneinander abhängig und müssen zusammenarbeiten, um den Anforderungen an das System zu genügen. In der Regel geht man beim Integrationstest schrittweise vor, indem man erst kleinere Subsysteme testet und sie dann zu größeren zusammenfügt.

Man kann damit beginnen, Subsysteme auf niedriger Granularitätsebene zu testen, wobei die Aufrufe außerhalb des Systems durch Treiber ersetzt werden (Bottom-Up-Integration). Oder man kann auf der höchsten Granularitätsebene beginnen und die noch nicht getesteten Subsysteme durch triviale Stubs oder Mock Objects[5] ersetzen (Top-Down-Integration). Als dritte Alternative kann man ohne Iterationen das gesamte System auf einmal testen (Big-Bang-Integration). Diese Alternative ist nur in den Fällen zu empfehlen, in denen das zu testende System klein ist oder die meisten Komponenten in ihrer Zusammenarbeit für sehr sicher gehalten werden, weil es sich z.B. um jahrelang eingesetzte Bibliotheken handelt.

**System- und Abnahmetest** Der Systemtest und der Abnahmetest umfassen beide eine fertig gestellte Anwendung. Getestet werden nur Eigenschaften, die das Gesamtsystem betreffen und Auswirkungen nach außen zeigen. Sie erfordern also nicht unbedingt das Wissen eines Programmierers, obwohl der Systemtest noch vom Entwicklungsteam durchgeführt wird. Der Abnahmetest richtet sich an diejenigen, die die Anwendung später einsetzen werden.

Testfälle werden als Eingabe an das gesamte System geschickt, so wie sie später in der eigentlichen Nutzung vom Benutzer ausgehen werden. Die Tests können rein funktionelle aber auch Performance-Anforderungen umfassen.

[4] Diskutiert wird u.A. unter [60].

[5] Siehe Abschnitt 2.3.

## 1.2. Aspektorientierte Programmierung

### 1.2.1. Einführung

Komplexität wird als eines der größten Qualitätshindernisse in der Softwareentwicklung angesehen. Darum konzentriert sich von jeher ein großer Teil der Forschungsbemühungen auf Kontrolle und Übersicht in Analyse, Entwurf und Implementierung. Objektorientierung ist eines der Ergebnisse der Bemühungen. Neben anderen Konzepten wie z.B. der Vererbung erlaubt die Objektorientierung, Objekte zu definieren, die bestimmte Daten in sich kapseln, sich auf klar definierte Weise manipulieren lassen und so von außen als Ganzes betrachtet werden können. Von der Komplexität aller Abläufe im Objekt wird abstrahiert. Ein Benutzer des Objektes braucht sich über seine inneren Abläufe (meistens) wenig oder keine Gedanken zu machen, da er nur über bestimmte Methoden auf das Objekt zugreift, für dessen Funktionieren der Autor des Objekts verantwortlich ist.

Leider lassen sich nicht alle Concerns[6] als Objekt einkapseln. Das klassische Beispiel für so einen Concern ist Logging. Angenommen in der Entwicklungsphase eines Software-Projekts wird die Entscheidung getroffen, dass bestimmte Ereignisse im Programmablauf (z.B. das Eintreten in alle oder bestimmte Methoden) auf eine einheitliche Art und Weise geloggt werden sollen.

Die Server-Seite dieser Aufgabe, die aus dem eigentlichen Schreiben des Log-Textes in eine Datei, auf den Bildschirm oder ein anderes Ziel besteht, kann bequem in eine Klasse gekapselt werden.

Der Aufruf des Clients in jeder Methode hingegen, muss in *jeder* Methode stehen. Und zwar nicht nur in allen bereits geschriebenen Methoden, sondern auch in denen, die noch geschrieben werden, sowie in externen Bibliotheksfunktionen, die nach jeder Veränderung neu kompiliert werden müssen, falls es überhaupt legal ist, diese zu manipulieren. Sollte sich einmal etwas am Aufruf ändern, muss der Code in allen Funktionen geändert werden. Wenn das Projekt reif für den Kunden ist, muss der gesamte Logging-Code wieder entfernt werden. Abbildung 1.1 (angelehnt an eine Grafik aus [20]) verdeutlicht die Situation.

Durch diesen sogenannten Cross-Cutting Concern wird der Code aufgebläht, unübersichtlich und schwer zu warten. Änderungen an den Cross-Cutting Concerns ziehen hohen Aufwand nach sich und werden daher lieber vermieden. Die Programmierer verbringen viel Zeit mit ermüdenden Routineaufgaben, anstatt sich auf die eigentlichen Probleme konzentrieren zu können.

Für solche Aufgaben wäre eine Objekt ideal, das bestimmte Verhaltensweisen an bestimmten definierten Stellen im Code vorschreibt. In der AOP werden diese Stellen als Pointcuts und der Code, der angibt, was dort geschehen soll, als Advice bezeichnet. Viele aspektorientierte Sprachen wie AspectJ kapseln die Pointcuts in Objekten. Diese Objekte heißen Aspekte. Die Pointcuts, denen innerhalb eines Aspektes ein Verhalten vorgeschrieben wird, liegen auf sogenannten Joinpoints.

Je nach Sprache können andere Joinpoints, potentielle Pointcuts, möglich sein. Bei-

[6] Abgrenzbare Aufgaben, die das Programm erfüllen muss, werden hier entsprechend der AOP-Terminologie als Concerns bezeichnet.

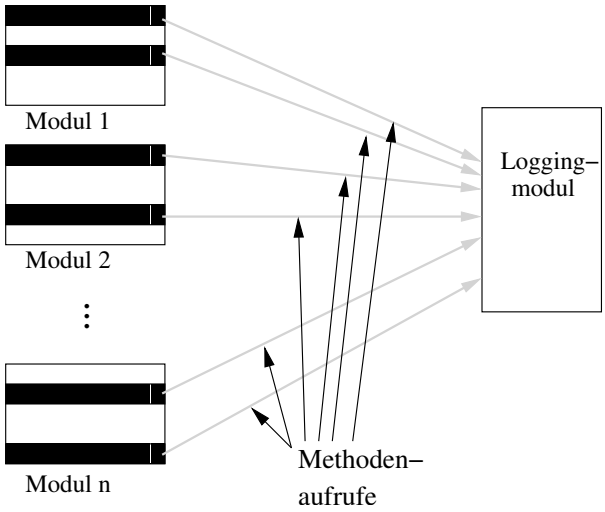


Abbildung 1.1.: Logging ohne AOP

spiele dafür sind: Das Eintreten in eine Funktion, der Zugriff auf eine Variable oder das Werfen einer Ausnahme. Die AOP erlaubt damit Quantifizierungen über gewisse Ereignisse im Code wie „nach allen Methodenaufrufen folgenden Musters tue dieses und jenes“.

Damit der Advice-Code zu den richtigen Zeitpunkten ausgeführt werden kann, muss er rechtzeitig in den Code eingesetzt oder „gewebt“ werden.

Die wichtigsten Lösungen dazu aus der Praxis sind:

**Compile-Time Weaving oder statisches Weben** Beim Kompilieren sorgt der Präprozessor des Compilers dafür, dass der Code an die richtigen Pointcuts eingefügt, „gewebt“ wird. Der Quellcode bleibt dabei für den Entwickler unverändert. Der gewebte Code existiert nur für den Compiler. AspectJ geht z.B. so vor.

**Run-Time Weaving oder dynamisches Weben** Der kompilierte Advice-Code wird erst zur Laufzeit an den richtigen Stellen in das laufende Programm eingefügt [27]. Object-Teams/Java webt z.B. dynamisch [12].

Der Concern, der sich vorher durch den gesamten Code gezogen hat, wird aus ihm herausgezogen und in ein einziges Modul gekapselt.

Zurück zu unserem Logging Beispiel: Anstatt die Logging Aufrufe über den ganzen Code zu verteilen, wird nur noch ein Aufruf in einen Aspekt geschrieben, der für die Client-Seite des Loggings im gesamten Projekt verantwortlich ist. Als Joinpoint kann man nun z.B. das Eintreten in eine Funktion definieren. Der Präprozessor webt dann den Logging-Code beim Kompilieren an den Beginn jeder Funktion ein. Ändert sich etwas am Logging-Aufruf, wird der Aspekt verändert. Möchte man kein Logging mehr, entfernt man einfach den Aspekt. Abbildung 1.2 (angelehnt an [20]) illustriert das Logging Beispiel mit AOP.

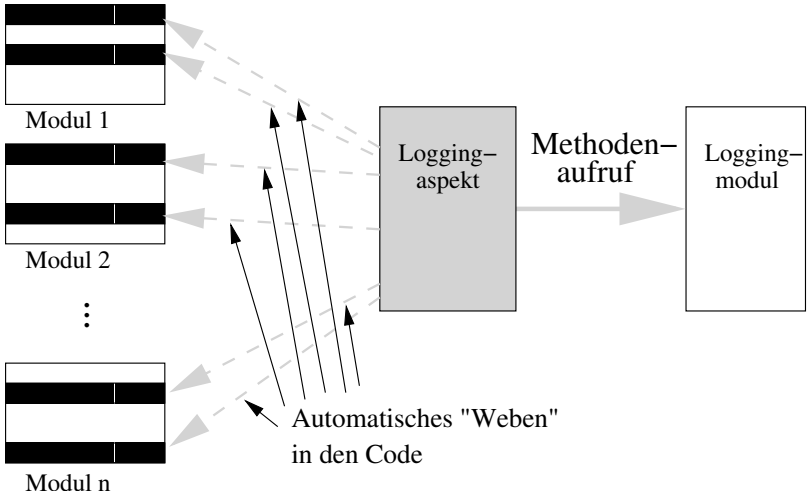


Abbildung 1.2.: Logging mit AOP

### 1.2.2. Historische Entwicklung der AOP

Die aspektorientierte Programmierung entstand aus den Anstrengungen vieler voneinander unabhängiger Institutionen, die zu Beginn der 90er Jahre mit Reflexion[7] und Variationen objektorientierter Konzepte experimentierten. Ihre Ideen flossen später zu dem zusammen, was heute unter AOP verstanden wird.

Die Gruppe um Gregor Kiczales bei Xerox PARC [16] ist von all diesen die bekannteste. Sie hat die meisten der in der AOP-Szene akzeptierten Begriffe geprägt. Ihre Forschung konzentrierte sich Anfang der 90er Jahre auf neue Ansätze zur Überwindung einiger Schwächen objektorientierter Programmierung bei großen, modularen Systemen mit hohen Flexibilitätsanforderungen. Während dieser Zeit kristallisierte sich die Kern-Eigenschaft der aspektorientierten Programmierung heraus: Sie ermöglicht es dem Programmierer, Cross-Cutting Concerns zu modularisieren.

Zusätzlich unterstützt von der DARPA, der Forschungsabteilung des US-amerikanischen Verteidigungsministeriums, entwickelte die Gruppe die praktische Implementierung ihrer Forschungsbemühungen. 1997 begannen die Arbeiten an AspectJ, einer aspektorientierten Erweiterung für den Java-Compiler. Die ersten externen User begannen 1998 AspectJ zu nutzen. Seitdem wird es stetig weiter entwickelt und gewartet.

Andere Projekte wie IBMs Hyper/J oder das Object Team Konzept stellen andere Spielarten dar, die eigene Variationen der AOP etabliert haben.

### 1.2.3. AOP Sprachen

Aus der aspektorientierten Forschung gingen zahlreiche Sprachen hervor. Zur Übersicht werden hier ohne Anspruch auf Vollständigkeit und in aller Kürze einige wichtige Spra-

[7] Fähigkeit einer Programmiersprache ihre eigenen Sprachkonstrukte in einem ihrer Programm zu analysieren. Das Programm „sieht sich selbst“ wie in einem Spiegel.

chen beschrieben.

### Hyper/J

Hyper/J [32] ist ein aspektorientierter Forschungsbeitrag des IBM-Konzerns. Hyper/J zeichnet sich vor anderen AOP-Sprachen dadurch aus, dass es innerhalb des eigentlichen Java-Quellcodes keine neuen Sprachkonstrukte verwendet, was die Einstiegsprobleme verringert.

Alle Concerns des Programms befinden sich in einem sog. Hyperspace und wurden als Java-Klassen implementiert. Mehrere Klassen, die einen Concern repräsentieren, bilden ein Hyperslice. Hyperslices in einem Bank-Terminal wären z.B. die verschiedenen Transaktionen und ein zusätzliches Hyperslice, das für die Autorisierung zuständig ist. Einige der Transaktionen erfordern Autorisierung, die sie in einem konventionellen Java-Programm selbst anstoßen müssten. Der betreffende Code wäre über das gesamte Transaktions-Slice verstreut. Von einem Autorisierungs-Slice als zusammengehörige Einheit könnte man nicht sprechen.

In Hyper/J bleibt das Autorisierungs-Slice eine Einheit. Sämtlicher Code, der sich auf die Autorisierung bezieht, bleibt im Autorisierungs-Slice. Wie die beiden Slices miteinander verzahnt sind, wird in einer zusätzlichen XML-Datei beschrieben. Der Effekt ist, dass die beiden Slices in einem Präprozessorschritt wie zwei Folien auf einem Overhead-Projektor übereinandergelegt werden und so im Bytecode fest verbunden sind. Für den Entwickler handelt es sich bei beiden Slices um getrennte Einheiten.

### Object-Teams/Java

Die Object-Teams/Java-Sprache [11, 54], die im Fachgebiet Softwaretechnik der TU Berlin entstand, baut auf der aspektorientierten Programmierung auf.

Die Beobachtung, dass an Objekte in verschieden Kontexten verschiedene Forderungen gestellt werden, führte zu der Idee, sie in verschiedenen Zusammenhängen verschiedene Rollen annehmen zu lassen. Objekte können zwar durch Vererbung auf verschiedene Anwendungen maßgeschneidert werden; diese Lösung ist aber häufig etwas zu schwerfällig für bestimmte Probleme.

Object Teams bilden einen Kontext, in dem die Objekte zu ihm passende Rollen annehmen können. Wenn der Instanz eines Objektes eine Rolle in einem Object Team zugewiesen wird, wird es fest an diese Rolle gebunden und verhält sich so, wie es im umgebenden Object Team vorgeschrieben ist. Das Objekt bildet die Basis zur Rolle im Object Team.

Die vier Hauptmechanismen der Sprache sind:

**Call-out** Eine Methode der Rolle wird an eine Methode des Basis-Objektes delegiert. Wird die Rollen-Methode aufgerufen, kümmert sich die Basis-Methode um die Ausführung.

**Call-in** Eine Call-in-Bindung leitet den Aufruf einer Basis-Methode an eine Methode der Rolle weiter.

**Lifting** Lifting ist das implizierte Wrappen der Basis in ihre Rolle mit den zugehörigen neuen Funktionalitäten. Hier wird auch sichergestellt, dass die Basis zur Rolle passt. Das Typsystem ist für das Lifting im richtigen Kontext zur richtigen Zeit verantwortlich.

**Lowering** Hierbei handelt es sich um den gegensätzlichen Vorgang zum Lifting. Wird die Basis eines Rollen-Objekts gebraucht, so wird die Rolle durch Lowering auf ihre Basis reduziert.

Anstelle mit gewissen Sprachmitteln ein besonderes Verhalten an Joinpoints zu binden, implementieren die Object Teams ein besonderes Verhalten für die Rollen.

Object Teams können laut [11] vieles schlanker ausdrücken als gewöhnliche aspektorientierte Methoden. Dafür bezahlt man mit einem Verlust an Flexibilität, da man ohne eine echte Joinpoint-Sprache auskommen muss.

### AspectJ

AspectJ [36, 20, 15], die bekannteste aspektorientierte Sprache, entstand, wie im historischen Teil schon erwähnt wurde, unter Mithilfe des AOP-Pioniers Gregor Kiczales.

Sie basiert auf Java und bietet dazu aspektorientierte Erweiterungen an. Die Sprachmittel der Sprache setzen die oben vorgestellten Ideen zur aspektorientierten Programmierung um. Bestimmte Stellen im Java-Code wie Methodenaufruf, Methodeneintritt, Erzeugung und Zerstörung von Objekten sind in AspectJ Joinpoints, die durch Pointcut-Beschreibungen erfasst werden können. Für die Pointcuts lassen sich Advices definieren, die vor, nach oder um einen Pointcut herum wirken können.

Die Pointcuts und Advices werden in Aspekten gekapselt. Die Aspekte gelten normalerweise global; ihre Wirkung kann aber auch auf bestimmte Objekte oder Kontrollflüsse eingegrenzt werden. In diesem Fall werden Instanzen der Aspekte beim Erzeugen eines bestimmten Objekts, bzw. beim Eintreten in einen bestimmten Kontrollfluss, erzeugt, und, wenn das Objekt zerstört oder der Kontrollfluss verlassen wird, zerstört.

Das Schlüsselwort `privileged` schließlich erlaubt den Advices, in den Aspekten auf durch Kapselung geschützte Member zuzugreifen.

Der praktische Teil dieser Diplomarbeit wurde in AspectJ geschrieben[8]. Der theoretische Teil bezieht seine Terminologie aus AspectJ.

### AspectC++

AspectC++ [35] ist das C++-Pendant zu AspectJ. Die Sprache wurde nach dem Vorbild von AspectJ entwickelt und ist AspectJ daher in der Syntax sehr ähnlich. In ihrer Bedeutung sind die Sprachmittel identisch.

[8] Warum die Wahl auf AspectJ fiel, wird in 3.1.2 geklärt.

## 1.3. AOP als Unit-Test-Instrument

### 1.3.1. Motivation

Die aspektorientierte Programmierung birgt wie die meisten „Paradigmen“ die Gefahr, an ihrer Intension vorbei eingesetzt zu werden. Ein der Syntax nach objektorientiert programmierter Code kann die Grundsätze der Objektorientierung vollkommen missachten. Die sprachlichen Ausdrucksmittel der Objektorientierung sind in diesem Fall nur noch Ballast, der die Verständlichkeit verschlechtert.

In der AOP empfiehlt es sich, noch strenger auf ihre sinnvolle Anwendung zu achten. Ihre vor allem durch die Ereignis-Quantifizierungen mächtigen Ausdrucksmittel üben eine große Macht über den Kontrollfluss aus. Einige wenige unnütz eingesetzte Aspekte können jedes Programm unlesbar machen.

Die AOP kann nur dann fähig sein, sinnvolle Beiträge im Aufgabengebiet Unit-Test zu leisten, wenn es sich dabei um ein Gebiet handelt, auf das die Mittel der AOP zugeschnitten sind. Aus den vorangegangen Ausführungen ging hervor, dass aspektorientierte Programmierung in erster Linie ein Werkzeug ist, das entwickelt wurde, um die Beherrschbarkeit von Cross-Cutting Concerns zu verbessern. Wenn es sich beim Testen also um einen solchen Cross-Cutting Concern handelt, sind sinnvolle Einsatzmöglichkeiten der AOP zumindest wahrscheinlich.

Zur Klärung dieser Frage benötigen wir eine exaktere theoretische Fundierung des Begriffs Cross-Cutting Concern, der bisher nur kurz angeschnitten wurde. In [9] wird eine sehr abstrakte Definition des Begriffs Concern vorgeschlagen. Ein Concern sei etwas, was man von einem Programm oder der Bedeutungsebene eines Programms entfernen kann. Genaueres solle auf der konzeptuellen Ebene erfolgen. Dieser sehr allgemeinen Definition folgen Details, die den Begriff des Cross-Cutting Concern klarer beleuchten.

Es werden zwei Funktionen definiert: Die Reduktionsfunktion $\pi_{syn}$ ist die Funktion, die das Programm auf der syntaktischen Ebene auf sich selbst ohne den Concern abbildet. Die Reduktionsfunktion $\pi_{sem}$ bildet die Bedeutung des Programms auf die Bedeutung des Programms ohne den Concern ab.[9] Die Interpretationsabbildung $\varphi$ von Syntax zu Semantik ist ein Homomorphismus bezüglich (Syntax, $\pi_{syn}$) und (Semantik, $\pi_{sem}$). Das heißt, es macht keinen Unterschied, ob man zuerst den Concern aus dem Quellcode entfernt ($\pi_{syn}$) und das Programm dann ausführt oder ob man das Programm sofort ausführt und die entsprechenden anschließend Aktionen verwirft ($\pi_{sem}$). Laxer ausgedrückt: $\pi_{syn}$ tut in der syntaktischen Welt genau dasselbe wie $\pi_{sem}$ in der semantischen Welt.

Wenn diese beiden Funktionen einfach und leicht anzugeben sind, ist der Concern auf syntaktischer und semantischer Ebene sauber abgegrenzt. Wenn $\pi_{sem}$ komplex ist, bestehen komplizierte Abhängigkeiten zwischen der Bedeutung des Concerns und den Bedeutungen der anderen Elemente des Systems. Eine komplizierte $\pi_{syn}$-Funktion bedeutet, dass der den Concern betreffende Quellcode nicht einfach vom Quellcode des Restsystems zu trennen ist. Hierbei handelt es sich um einen Cross-Cutting Concern.

Auf den ersten Blick scheint die $\pi_{syn}$ des Concerns (Unit-)Testen eine sehr einfache zu

[9] Kann man als das Weglassen bestimmter Operationen im ausgeführten Programm verstehen.

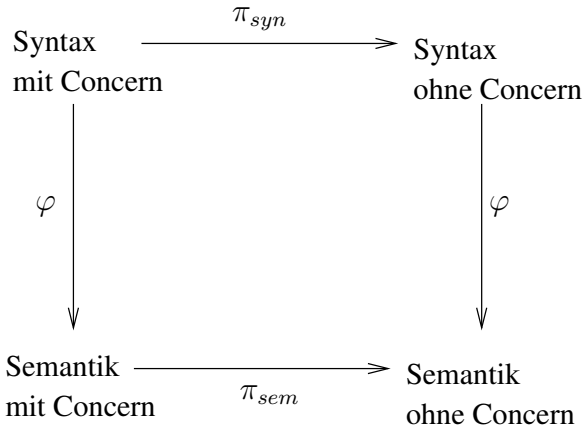


Abbildung 1.3.: Reduktionsfunktionen

sein: „Entferne das Modul, welches die Testnachrichten an die zu testenden Methoden schickt und die Ergebnisse prüft."

Das ist richtig, wenn man außer acht lässt, dass der wirksame objektorientierte Unit-Test einige Eingriffe z.B. zur Instrumentierung oder zur Simulation nötig macht, die zwangsläufig in den Quellcode des zu testenden Systems eingreifen. Diese Eingriffe sind keine Ausnahme. Sie betreffen alle zu testenden Objekte auf einer niedrigen Granularitätsebene. Alles in allem ein Concern, der sich über viele Einheiten im Code verteilt: Ein Cross-Cutting Concern.

Die Verwendung der AOP im Unit-Testen hilft nicht nur dabei, fundamentalen Eigenschaften des Cross-Cutting Concerns Unit-Testen elegant zu fassen. Sie bietet mit ihren mächtigen Quantifizierungen noch viele andere Anwendungsmöglichkeiten, die das eigentliche Unit-Testen vereinfachen, auf die in den Abschnitten des zweiten Kapitels noch näher eingegangen wird.

### 1.3.2. Publikationen und praktische Beiträge zu diesem Thema

Die Idee AOP im Kontext Testen einzusetzen ist nicht neu. In den Publikationen [20, 21, 18, 30] wird Testen und in [8, 6] Sourcecode-Instrumentierung als mögliches Anwendungsgebiet erwähnt. Der Artikel [21] beschäftigt sich intensiv mit dem Thema. Insbesondere liefert er Vorschläge zum Thema Mock Objects im Unit-Test im Extreme Programming-Umfeld.

CricketCage [38] ist ein Code-Generator für JUnit, der aus Java-Quellcode einfache JUnit-Testfälle erzeugt. Der Nutzer muss in einem Aspekt zwei Bereiche abgrenzen. Den outside-Bereich und den inside-Bereich. Aufrufe des outside-Bereichs an den inside-Bereich werden als Testfälle interpretiert, aus denen der Generator die Testfälle konstruiert. Zur Erweiterung der Mächtigkeit von JUnit wird AOP in diesem Framework nicht eingesetzt. Das Projekt befindet sich in der alpha-Phase und wurde seit Mitte 2002 nicht mehr aktualisiert.

VirtualMock [59] ist eine Erweiterung zu JUnit. Das Framework stellt Hilfsmittel bereit, die es erlauben, manipulierte, in den meisten Fällen vereinfachte Varianten von Objekten und Methodenaufrufen ohne Manipulation des Quellcodes zu erzeugen. Das Projekt befindet sich noch in einer frühen alpha-Phase, lässt sich aber schon benutzen und wird stetig weiterentwickelt.

### 1.3.3. Ziele der Arbeit

In dieser Arbeit sollen theoretisch die Einsatzmöglichkeiten aspektorientierter Programmierung im Unit-Testen untersucht werden. Auf der praktischen Seite steht ein Unit-Testframework, das im Aufbau JUnit ähnelt, aber erweiterte Merkmale aus der Sprache AspectJ besitzt. Soweit möglich wurden alle Ideen aus der Theorie im Framework umgesetzt. Die Ausnahmen werden im Ausblick besprochen.

# 2. AOP im Unit-Test in der Theorie

Dieses Kapitel setzt sich auf theoretische Weise mit den Möglichkeiten der AOP im Unit-Test auseinander und vergleicht die Ergebnisse mit konventionellen Methoden und anderen Ansätzen.

## 2.1. Instrumentierung und Durchbrechen der Kapselung

Beim Unit-Test werden Nachrichten an Methoden gesandt und nach ihrer Abarbeitung überprüft, ob das Ergebnis mit den gestellten Anforderungen übereinstimmt. Das Ergebnis einer Methode besteht nur in funktionalen Sprachen ausschließlich aus ihrem Rückgabewert. Besonders in objektorientierten Sprachen kommen Seiteneffekte wie die Änderung von gekapselten Member-Variablen hinzu. Andere Effekte wie z.B. die Ausführungsdauer einer Methode oder die Anzahl von Methodenaufrufen sind ohne geeignete Maßnahmen nicht messbar. Das Operationalisieren[1] von nicht direkt sichtbaren Ereignissen und Ergebnissen bezeichnet man in diesem Zusammenhang als Instrumentieren [6].

Sowohl die Überwindung der Kapselung als auch das Durchsetzen des Codes mit Instrumentierungsmeldungen führt zu Problemen und Konflikten.

### 2.1.1. Konflikte zwischen Test- und Einsatzphase

Die Informationsgewinnungs-Maßnahmen, die in der Testphase unabdingbar sein mögen, sollten in der Release-Version der Anwendung nicht mehr aktiv sein. Das gilt besonders für die Kapselung aber auch für die Instrumentierung.

#### Kapselung

Das einfache, funktionale Testen einer Methode erfordert keinerlei Hilfsmittel. In einem prozeduralen System würde man Nachrichten an die Methode schicken, vielleicht einige globale Variablen setzen und prüfen, ob sie richtige Ergebnisse liefert.

In objektorientierten Systemen ist die Lage komplizierter. Klassen kapseln ihre Member und verhindern den Zugriff auf ihre Werte. Von den Werten dieser Member hängen jedoch auch die Ergebnisse ihrer Methoden ab. Also müssen diese Member und, falls es sich um komplexere Objekte handelt, auch die Member der Memberobjekte in der Testphase zugänglich gemacht werden.

Das Durchbrechen der Kapselung ergibt nur in der Testphase Sinn. Würde diese Änderung an einer Klasse auch nach der Testphase bestehen bleiben, verlöre die Klasse mit der Kapselung eine ihre wichtigsten Eigenschaften: Objektorientierte Sprachmittel könnten

[1] Messbar machen, so dass die Ergebnisse ordinal darstellbar sind.

nicht mehr sicher stellen, dass sie auf die richtige, konsistente Art und Weise eingesetzt wird.

#### Instrumentierung

In der Testphase könnten einige Informationen über den laufenden Code für den Tester interessant sein, an die er ohne instrumentierende Maßnahmen nicht gelangt. Solche Informationen wären z.B.: Die Anzahl der Aufrufe einer teuren Funktionalität oder die Zeit, die eine Methode für eine bestimmte Aktion benötigt.

Oft ist es dabei nicht mit einer Ausgabe getan. Manchmal müssen Namenslisten verwaltet werden oder Timer zu den richtigen Zeitpunkten aktiviert und deaktiviert werden. Vieles davon muss der Tester selbst in den Code integrieren, wenn er manuell instrumentieren will.

Nach dem Testen muss der Instrumentierungscode aus dem gesamten instrumentierten Code wieder entfernt werden. Instrumentierungsausgaben verwirren den Nutzer oder auch andere Tester. Hinzu kommen die Performancekosten durch die Instrumentierung.

### 2.1.2. Bereits existierende Lösungen aus der Praxis

Da größere Systeme häufig einen hohen Instrumentierungsbedarf haben, wäre es für den Tester angenehm, wenn ihm Automatismen diese mühevolle Arbeit abnehmen könnten. Aus diesem Grund gibt es auch schon einige Tools für diesen Zweck. Besser als spezielle Tools einzusetzen, wäre es jedoch, wenn sich Instrumentierungen sicher und konsistent mit den Sprachmitteln der benutzten Programmiersprache vornehmen ließen.

#### Sprachmittel

**C++** Die teilweise objektorientierte Programmiersprache C++ [29] stellt dem Programmierer das Schlüsselwort `friend` zur Verfügung, mit dem er Klassen vollständigen Zugriff auf alle Member einer Klasse gewähren kann. Eine Testklasse, die den vollen Zugriff benötigt, kann ihn so exklusiv bekommen. Im folgenden Beispiel bekommt die Klasse `TestApplication` exklusiven Zugriff auf alle Member der Klasse `Application`:

```
...
#include "TestApplication.h"
...

class Application
{
        // hier bekommt die Klasse TestApplication ihre Zugriffsrechte.
        friend class TestApplication;

        private:
        ...
}
```

Einigen kann dieser exklusive Bruch mit der Kapselung über die Testphase hinaus schon zuviel sein. Dass sich die Header der Testsuite-Dateien immer im Scope des Compilers befinden müssen, ist auch nicht immer angebracht.

Abhilfe schafft der dem eigentlichen Compiler vorgeschaltete C-Makro-Präprozessor. Er ermöglicht bedingte Kompilierungen, so dass der Tester den exklusiven Testzugriff per Compiler-Option aktivieren und deaktivieren kann.

Nach ein paar kleineren Änderungen für den Präprozessor wird die Brechung der Kapselung mit der Compiler-Option `TESTPHASE` konfigurierbar:

```
...
// Ist TESTPHASE definiert?
#ifdef TESTPHASE
    // Wenn ja, importiere den folgenden Header
    #include "TestApplication.h"
#endif
...

class Application
{
    // Ist TESTPHASE definiert?
    #ifdef TESTPHASE
        // Wenn ja, gewähre TestApplication Zugriffsrechte.
        friend class TestApplication;
    #endif

        private:
        ...
}
```

Anstatt `TESTPHASE` über den Compileraufruf zu definieren, kann man auch an eine passende Stelle im Code ein `#define TESTPHASE` einfügen.

Auch Instrumentierung ist mit Makros möglich.

Im folgenden Beispiel wird bei definiertem `TESTPHASE` gezählt, wie oft die Methode `veryExpensiveMethod()` aufgerufen wird:

```
...
// Ist TESTPHASE definiert?
#ifdef TESTPHASE
    // Wenn ja, besorge einen Zeiger auf die Instanz des Statistics-Objects
    Statistics* pStatInstance = Statistics::getInstance();
#endif
...
if (veryComplicatedPredicate())
{
    // Ist TESTPHASE definiert?
```

```
        #ifdef TESTPHASE
        // Wenn ja, inkrementiere den Zähler für die teure Methode
        pStatInstance->incVeryExpensiveMethod();
    #endif
    veryExpensiveMethod();
...
```

Wie man sieht, bietet C++ dank seines Präprozessors (der den meisten C++-Compilern beigefügt ist) und des `friend`-Schlüsselwortes ausreichend Unterstützung zur konfigurierbaren Instrumentierung und Kapselungsdurchbrechung.

Nachteilig am Makro-Einsatz ist, dass

- die eigentliche Sprache C++ verlassen wird.
- das Programm unübersichtlicher wird.

**Java** Auf direktem Wege erlaubt Java in keinster Weise einen Zugriff auf `private`- und `protected`-gekapselte Member durch nichtverwandte Klassen. An die `protected`-Member käme man über Vererbung heran. Das ist nicht ausreichend. Hinzu kommt, dass von einigen Klassen gar nicht vererbt werden darf (z.B. Java-Klassen, die mit dem Schlüsselwort `final` gekennzeichnet sind).

Das Java-Reflection-Package [58, 46] erlaubt über Umwege den Zugang zu `protected`- und `private`-Membern, sofern der u.U. aktive Security Manager das gestattet.

Der folgendene Code gibt Informationen inkl. den Werten aller Member von `TestClass` aus:

```
public class TestClass
{
  private int test1 = 1;
  protected int test2 = 2;
  public int test3 = 3;
}

public class Reflector
{
  public static void main(String[] args)
  {
    try
    {
      TestClass instance = new TestClass();
      Class cls = Class.forName("TestClass");

      Field fieldlist[]  = cls.getDeclaredFields();
      for (int i= 0; i < fieldlist.length; i++)
      {
```

```
        Field fld = fieldlist[i];
        fld.setAccessible(true);
        System.out.println("Name = " + fld.getName());
        System.out.println("Dekl. Klasse = " + fld.getDeclaringClass());
        System.out.println("Typ = " + fld.getType());
        System.out.println("Wert = " + fld.getInt(instance));
        int mod = fld.getModifiers();
        System.out.println("modifiers = " + Modifier.toString(mod));
        System.out.println("-----");
      }
    }
    catch(Throwable e)
    {
      System.err.println(e);
    }
  }
}
```

Bei deaktiviertem oder tolerant konfiguriertem Security Manager sieht die Ausgabe so aus:

```
Name = test1
Dekl. Klasse = class TestClass
Typ = int
Wert = 1
modifiers = private
-----
Name = test2
Dekl. Klasse = class TestClass
Typ = int
Wert = 2
modifiers = protected
-----
Name = test3
Dekl. Klasse = class TestClass
Typ = int
Wert = 3
modifiers = public
-----
```

Man kann festhalten, dass das Durchbrechen der Kapselung, wenn auch über Umwege, in Java möglich ist.

Anders sieht es mit der bedingten Instrumentierung aus. Java-Compiler besitzen in der Regel keinen Präprozessor und können darum auch nicht bedingt kompilieren. Bedingte Instrumentierung lässt sich nur über `if... else...`- Konstrukte realisieren.

Je nach verwendetem Compiler können daraus Ineffizienzen resultieren. Verzweigungen verschlechtern die Voraussagbarkeit des Kontrollflusses. Automatische Optimierungen, in denen Anweisungs-Umgruppierung eine Rolle spielt, werden erschwert. Ist die Instrumentierungs-Bedingung eine statische Konstante, können viele Compiler den Verzweigungscode heraus optimieren. Trotzdem bleiben zumindest für den Entwickler viele `if... else...`-Konstrukte, die mit dem Concern, der an der jeweiligen Stelle implementiert werden soll, nichts zu tun hat.

#### Tools

Die Firma Glen McCluskey & Associates LLC [45] bietet einen Instrumentierungs-Toolkit für Java an. Dieser Toolkit besteht aus Klassen, die den zu instrumentierenden Code parsen und einen Syntaxbaum erstellen. Auf diesen kann der Benutzer des Toolkits in seinen Instrumentierungsklassen zugreifen und die gewünschten Instrumentierungen in den Baum einfügen. Aus dem veränderten Syntaxbaum erstellt das Toolkit neuen Klassen-Source-Code, der wieder kompiliert werden muss. Die kostenlose, binäre Version des Toolkits, die sich auf der Homepage der Firma herunterladen lässt, erlaubt keine eigenständige Instrumentierung, da man in seinem Sourcecode keinen Zugriff auf die Funktionalität des Code-Parsens und Syntax-Baum-Parsens hat. Dafür werden vorkompilierte Lösungen für die Standardaufgaben vollständiges Tracen, Tracen von Methodenaufrufen oder die Zahl der Aufrufe von Statements mitgeliefert.

Bei JIapi [47], der Java Instrumentation API, handelt es sich um ein Framework, in dem die Instrumentierung auf Bytecode-Ebene erfolgt. Es kann den Bytecode einer Klasse instrumentieren und eine neue instrumentierte Klasse ausgeben. Man kann es auch vor die Java Virtual Machine schalten. In diesem Fall werden die Klassen, die instrumentiert werden sollen, dann instrumentiert, wenn sie vom Class Loader angefordert und an die Java Virtual Machine weitergegeben werden.

Die Instrumentierungen der Java Instrumentation Engine [48] werden über XML spezifiert. Sie liest den zu testenden Code ein, parst ihn, wendet XML-kodierte Instrumentierungs-Anweisungen auf den geparsten Code an und erzeugt schließlich eine neue Quelldatei, die den instrumentierten Code enthält. Dieser in Java zur Java-Instrumentierung entwickelte Code-Generator ist freie Software und kann auf seiner Homepage heruntergeladen werden. Er wird allerdings seit 1999 nicht mehr gewartet. Die Entwickler verweisen explizit auf AspectJ als Alternative.

Es existieren noch viele weitere Lösungsansätze für das Instrumentierungsproblem, die hier nicht alle vorgestellt werden sollen. In vielen Fällen wird der Tester auch kein externes Hilfsmittel benutzen, sondern am zu testenden Quellcode selbst Instrumentierungen vornehmen. Dass das aufwändig und vielleicht sogar gefährlich sein kann, wurde im Motivationskapitel 1.3.1 schon besprochen.

### 2.1.3. AOP-Ansätze

Dieser Abschnitt handelt davon, wie das Instrumentierungsproblem und andere kleine Probleme, die mit der Informationsgewinnung im Testen zu tun haben (darunter fällt

auch das Durchbrechen der Kapselung) und das einfache Unit-Testen behindern, mit aspektorientierten Methoden gelöst werden können, ohne den Quellcode der zu testenden Klassen ändern zu müssen, was sich u.U. nur schwer wieder rückgängig machen ließe. Der Testprozess bleibt dabei derselbe. Es werden immer noch Nachrichten an Methoden geschickt und deren Ergebnis geprüft. Die Überlegungen dieses Abschnitts sind mehr Hilfsmittel und Werkzeuge als neue Ideen zum Unit-Testing.

Bevor man das Instrumentierungsproblem aus der aspektorientierten Perspektive betrachtet, sollte man sich über die verwendeten Fähigkeiten und Werkzeuge im Klaren sein. Eine aspektorientierte Sprache, die die Überlegungen dieses Abschnitts umsetzen kann, sollte über Sprachmittel verfügen, die folgendes ausdrücken können:

- Quantifizierung von Ereignissen (wie den Eintritt in Methoden).
- Bindung eines Aspektes an ein Objekt.
- Bindung eines Aspektes an eine Kontrollflusseinheit. [2]
- Privilegierte Aspekte, die auf private Klassenmember zugreifen können.

Eine Sprache, die diese Forderung erfüllt, ist z.B. AspectJ.

#### Privilegierte Aspekte

Am wichtigsten für den Unit-Test sind Instrumentierungen, die den Zugriff auf sonst unzugängliche Daten erlauben. Nehmen wir als Beispiel ein Objekt an, das einen Gegenstand in einem drei-dimensionalen Raum beschreibt. In sich kapselt es eine positionsbeschreibende 4x4-Matrix, auf die der Benutzer des Objekts keinen direkten Zugriff haben soll. Die Start-Position des Objekts wird im Konstruktor übergeben. Veränderungen geschehen ausschließlich über lineare Transformationen mit anderen Matrizen. Jeder Transformationsschritt bringt kleinere numerische Fehler mit sich, die sich nach genügend langer Zeit zwangsläufig zu größeren Fehlern aufaddieren. Die Rotationsvektoren dürfen aber niemals ihre Normalitätseigenschaft verlieren. Darum gibt es in der Klasse einen Mechanismus, der häufig genug eine Normalisierung der Rotationsvektoren durchführt. Eine Möglichkeit wäre es z.B. bei jeder Veränderung der Matrix zu normalisieren. Die Aufgabe des Testers ist es nun, diesen Mechanismus zu testen.

Der Tester kann dazu z.B. einfach die konventionellen Möglichkeiten nutzen und so lange mit möglichst numerisch ungünstigen Matrizen transformieren, bis etwas passieren muss. Das kann im Testlauf zu viel Zeit kosten. Außerdem ist es schwer, bestimmte Matrizen zu testen, da man berechnen muss, mit welchen Transformationen man auf die gewünschte Matrix kommt.

Sehr viel angenehmer wäre es, wenn man nur in der Testphase vollen Zugriff auf die Matrix hat danach aber nicht mehr. Man könnte die Matrix in der Testphase z.B. öffentlich zugänglich machen. Für einen kurzen Test mag das vertretbar sein. Alle Methoden

[2] Eine Kontrollflusseinheit ist eine Folge von Anweisungen, die zwischen zwei Punkten durchlaufen wird. Eine Kontrollflusseinheit könnte beginnen, wenn eine Methode aufgerufen wird und enden, wenn sie wieder verlassen wird.

eines Projektes werden aber wiederholt getestet (Regressionstest) und zwar mit allen anderen Klassen zusammen, so wie sie in der Endanwendung sein sollen. In diesen Tests müssen sie sich so verhalten, wie sie gedacht sind, also auch die sinnvolle Kapselung beibehalten, damit andere Entwickler sie richtig benutzen.

In [20] wird der Einsatz von privilegierten Aspekten in der Testphase vorgeschlagen. Zugriffsbeschränkungen auf andere Objekte existieren für privilegierte Aspekte nicht. In ihnen ist es also möglich, auf die gekapselten Werte zuzugreifen. Die Aufhebung der Kapselung gilt nur für diesen Aspekt, andere Objekte sind davon nicht betroffen. Darum ist auch keine Trennung zwischen Test- und Release-Sourcecode nötig. Im Release ist der Aspekt einfach nicht enthalten.

Im obigen Beispiel könnte z.B. das Setzen der Matrix mit verschiedenen Werten in einem privilegierten Aspekt geschehen. Derselbe Aspekt könnte auch direkt von der Klasse erfahren, ob die Matrix noch über die gewünschten Eigenschaften verfügt.

Der Tester muss nun noch dafür sorgen, dass der Aspekt mit den richtigen Testdaten testet und die Ergebnisse überprüft. Advices werden nicht einfach aufgerufen wie Klassenmethoden. Ihr Code wird an den Stellen eingewebt, für die ihr zugehöriger Pointcut definiert ist.

Soll ein Advice einfach nur das Ergebnis eines oder mehrerer Testfälle prüfen, reicht es, einen Pointcut für die Testfallaufrufe, deren Ergebnis mit dem Advice geprüft werden soll, zu definieren. In diesem Advice kann dann auf alle Member getesteter Klassen zugegriffen werden. Möchte man auch eine Vorbedingung setzen, bindet man an denselben Pointcut auch einen Advice, der die Vorbedingung vor dem Ausführen der Methode setzt.

Das Binden an die richtigen Punkte lässt sich vereinfachen, indem der Aspekt nur an eine bestimmte Testfall-Klasse oder an eine Kontrollflusseinheit gebunden wird. Das Binden an eine Klasse oder Kontrollflusseinheit beschränkt die Wirkung des Aspekts auf das Objekt, an das er gebunden wurde. So wird vermieden, dass sich Aspekt-Code an Stellen webt, an denen er nicht benötigt wird.

#### Instrumentierungen mit AOP

Im obigen Abschnitt über C++-Sprachmittel zur Instrumentierung wurde gezeigt, dass der C++-Präprozessor eine ausreichende Unterstützung für bedingte Instrumentierungen bietet. Das Mitkompilieren des Instrumentierungscodes wurde durch die Definition einer Konstante beim Kompilieren aktiviert und deaktiviert.

Das Mitkompilieren von Aspekten lässt sich auf eine ähnliche Weise beim Kompilieren ein- und ausschalten. Definiert man Instrumentierungen für bestimmte Punkte in einem Aspekt, lässt sich die Instrumentierung beim Kompilieren aktivieren und deaktivieren. Die Aspekte lassen sich folglich ähnlich wie C-Makros zur Instrumentierung verwenden.

Außerdem erlauben die aspektorientierten Quantifizierungen mächtigere und dabei einfachere Instrumentierungen. Falls die Joinpoints, für die eine Instrumentierung aktiviert werden soll, sich durch ein Muster beschreiben lassen, lassen sie sich alle mit einem Pointcut fassen und instrumentieren.

### Debugging und Logging in der Testphase

Die Hauptaufgabe eines Testframeworks ist es, es dem Tester leicht zu machen, eine möglichst vollständige Suite von Testfällen zu erstellen. Darüberhinaus sollte die Ursache auftretender Fehler leicht zu lokalisieren und das Abarbeiten der Testsuite von außen gut nachvollziehbar sein. Letzteres kann man unter dem Begriff Logging zusammenfassen.

Logging ist das klassische Anwendungsbeispiel für aspektorientierte Programmierung. In nahezu jedem Lehrbuch für aspektorientierte Sprachen [20] wird diese Anwendung ausführlich besprochen. Das Problem ist so einfach zu lösen, weil es sich bei den Punkten, an denen das Loggen stattfinden soll, um leicht zu beschreibende Joinpoints (Eintritt in eine Methode) handelt. Ein Pointcut mit zugehörigen Advice, der die Logging-Anweisung enthält, reicht aus, um das Problem zu lösen.

Die Forderung nach der Fehlerlokalisation[3] könnte man als eine Forderung an den Tester ansehen. Er sollte seine Tests so gestalten und ausführlich mit Textausgabe kommentieren, dass die auftretenden Fehler problemlos zu lokalisieren sind. Gegen diese Ansicht sprechen die Unmengen trivialer Tests, deren vollständige Kommentierung niemandem zumutbar ist.

Betrachtet man die Aufgabe genauer, stellt man fest, dass sich eine Kommentierung der Testfälle, die zur Ursachenlokalisation ausreicht, auf einige wenige verallgemeinbare Textausgaben beschränkt. Gescheiterte Unit-Testfälle haben meist einen so kleinen Wirkungsbereich, dass der Hinweis auf den richtigen Testfall ausreicht, um die Ursache des Fehlers zu finden.

Ein Testframework in einer aspektorientierten Sprache verfügt über die mächtigen Reflexionsfähigkeiten, die die Sprache bietet, und kann diese sowohl in den Tests, die konventionell ablaufen, als auch in den aspektunterstützten Tests einsetzen. Eine einheitliche Syntax für die Testfälle wie z.B. `check(boolean)`, wobei der boolesche Ausdruck die zu testende Eigenschaft ist, ermöglicht es, alle diese Ausdrücke in einem Aspekt zu prüfen. Der kann über die Reflexion Auskunft über die Quellcodedatei, den Klassennamen, den Methodennamen und die Zeilennummer, in der der Testfall aufgerufen wurde, erlangen. Diese Informationen kann er im Fehlerfall ausgeben oder einer Liste für die Statistik hinzufügen.

Ein Advice für einen Pointcut, der sich auf alle `TestCase.check(boolean)` bezieht, kennt den Ort, auf den sich sein Pointcut bezieht. In AspectJ weiß er nicht nur, in welcher Methode und Klasse der `check()`-Aufruf sich befand, sondern sogar die Zeilennummer. Der Advice kann dann automatisch Fehlermeldungen wie folgende erzeugen:

```
Testfall aufgerufen in TestProbe.java: Zeile 40 erzeugte ein fehlerhaftes
Ergebnis.
```

[3] Der Ort, an dem der Fehler bemerkt wurde (z.B. in einer bestimmten Methode) ist gemeint, nicht die Ursache des Fehlers. Könnte das Framework die Ursachen von Fehlern finden, wäre der menschliche Tester überflüssig. Außerdem ist es weniger die Aufgabe des Testers, die Ursachen bemerkter Fehler zu finden.

### 2.1.4. Vor- und Nachteile des AOP Einsatzes

#### Beim Durchbrechen der Kapselung

AOP ist eine völlig neues Herangehensweise, in das man sich als Tester erst einmal einarbeiten muss. Gerade in der Einführungsphase kann es zu Anwendungsfehlern aufgrund mangelnden Verständnisses kommen, und es stellt sich die Frage, ob dieser Preis nicht zu hoch ist, um an die Werte einiger gekapselter Variablen zu kommen.

Darauf lässt sich erwidern, dass die Kapselungsdurchbrechung nicht der einzige sinnvolle Einsatz der AOP im Unit-Testen ist, wie sich im weiteren Verlauf der Arbeit noch zeigen wird. Schon wenn man einen der anderen vorgeschlagenen Ansätze praktisch implementiert, reicht z.B. bei AspectJ ein Schlüsselwort in der Aspekt-Deklaration, um die Kapselung nur für den Testaspekt zu brechen.

Wenn man also sowieso den Einsatz der AOP plant, sind die privilegierten Aspekte keine Hürde mehr. Ansonsten sind die Bedenken wohl berechtigt. Es gibt schließlich in den meisten Sprachen noch andere Lösungen, auch wenn es sich dabei wie in Java um recht mühsame Wege handeln kann.

#### Bei der Instrumentierung

Der AOP-Einsatz für die Instrumentierung ist unnötig, wenn nur eine bestimmte, abgegrenzte Codeeinheit instrumentiert wird und diese Instrumentierung nicht jederzeit leicht entfernbar sein muss. Es reicht dann einfach, den Instrumentierungscode an die Codestelle zu schreiben.

Stellt man höhere Anforderungen an die Flexibilität der Instrumentierungen, kommt es darauf an, ob die Programmiersprache über eine Makro-Sprache verfügt, wieviel instrumentiert wird und wie hoch die Performance-Anforderungen sind.

- Wird sehr wenig instrumentiert, kann man den Instrumentierungscode leicht manuell entfernen.
- Sollten die Performance-Anforderungen niedrig sein, lässt es sich in einer booleschen Variable festlegen, ob instrumentiert werden soll. An den kritischen Stellen wird mit `if...else...` geprüft, ob instrumentiert wird oder nicht.
- Wenn die vorigen beiden Bedingungen nicht zutreffen sollten und die benutzte Programmier-Sprache über einen Präprozessor verfügt, lassen sich die Instrumentierungen als Compiler-Option optional kompilieren.
- Ohne Präprozessor mit einem anspruchsvollen Instrumentierungsproblem belastet, tut der Tester sehr gut daran mit der AOP zu instrumentieren. Wenn an verschiedenen Stellen auf die gleiche Weise instrumentiert wird, kommt er sogar mit sehr viel weniger Code aus als in der konventionellen Fassung.

#### Beim Debuggen und Logging

An den hier zum Testen vorgestellten Hilfsmitteln aus der AOP, die das Debuggen und Loggen betreffen, ist schwer etwas zu kritisieren. Logging bedeutet im Normalfall repetetive Kleinstarbeit. Mit der AOP kann man das Logging für ein ganzes Framework in zwanzig Zeilen sicher, modular und übersichtlich zusammenfassen.

Auch dass er nur noch Fehlerbeschreibungen für gescheiterte Testfälle schreiben muss, wenn er will, kann dem Tester nur willkommen sein. Das Framework erstellt automatisch einen Fehlerstring, der den Ort des Fehlers angibt.

Dabei darf nicht vergessen werden, dass die aspektorientierten Lösungen alleine nicht immer ausreichend sind. Es handelt sich nur um eine Erweiterung. Zeilenweises Logging ist in den meisten AOP-Sprachen z.B. nicht möglich. Möchte der Tester eine Nachricht in eine beliebige Zeile loggen, sollte er das konventionell tun.

## 2.2. Verallgemeinerte Tests

Testen kann eine stupide, repetitive Tätigkeit sein. Ein Tester muss seine Zeit oft mit der Implementierung von immer gleichen, trivialen Testfällen vergeuden. In vielen Fällen erwartet eine Menge von Testfällen dasselbe Ergebnis, das für jeden Fall einzeln abgeprüft werden muss. Dabei entstehen Unmengen trivialen Codes, der aber leider allein des Regressionstests wegen nötig ist.

Die mächtigen Quantifizierungsfähigkeiten der AOP erscheinen wie geschaffen dafür, dem Tester diese Schreibarbeit abzunehmen, damit er seine Arbeitszeit für anspruchsvollere Testaufgaben nutzen kann.

### 2.2.1. AOP-Ansätze

#### Testergebnisse zusammenfassen

Methoden, deren Werte aus einer kleinen Menge kommen, zeichnen sich dadurch aus, dass viele Überprüfungen wenig verschiedene Ergebnisse liefern. Im letzten Kapitel wurde erläutert, wie sich Ergebnisse von Testfällen mit Hilfe von Pointcuts überprüfen lassen. Die Quantifizierungen der AOP erlauben eine Verallgemeinerung dieser Überprüfungen.

Je nach Sprache lassen sich Pointcuts nach bestimmten Mustern definieren, die den Aufruf der Testmethode beschreiben. Anstatt nur einen bestimmten Testfallaufruf abzufangen, kann man auch alle Testfallaufrufe, die dasselbe Ergebnis erzeugen, in einem Pointcut zusammenfassen.

Ein Matrix-Konstruktor, der nur gültige Daten akzeptiert, kann folgendermaßen getestet werden: Man definiere einen Pointcut für alle positiven Fälle, in denen die Konstruktion gelingen soll und einen für alle negativen Fälle, in denen die Konstruktion nicht gelingen darf. Dann schreibe man in einem Kontext, den die gewählte aspektorientierte Programmiersprache von anderen unterscheiden kann, seine Testfälle für die Konstruktionen, die gelingen sollen und in einen anderen die Konstruktionen, die scheitern sollen.

Der Tester schreibt nur noch die positiven und negativen Aufrufe, die Überprüfung für alle Fälle erfolgt in jeweils einer Zeile in den betreffenden Aspekten.

### Subklassen-Tests für Liskov-konforme Subklassen

#### Liskov-Konformität

Vererbung hat in einem Softwaresystem zweifachen Nutzen [4]:

- Methoden, Subklassen und Strukturen sind leicht und einfach wiederverwendbar (Convenience Inheritance).
- Sie drückt verwandtschaftliche Beziehungen zwischen den Klassen bezüglich ihres Einsatzgebietes und ihrer Funktionsweise aus und gestaltet die Struktur übersichtlicher (Representation Inheritance).

Der erste Nutzen kommt mehr der Bequemlichkeit der Programmierer der neuen Subklasse entgegen, die sich Arbeit sparen können, wenn sie benötigte Funktionalitäten aus anderen Klassen erben. In geschlossenen Systemen, die nicht mehr viel überarbeitet werden und nur von wenigen Entwicklern betreut werden, kann man diesen Weg gehen.

Der zweite Nutzen kommt den Benutzern der Klasse entgegen, weil er ihnen hilft, die Klasse in den richtigen Kontext einzuordnen. Wenn sie sich in einer sorgfältig entworfenen Klassenhierarchie auskennen, können sie schnell entscheiden, wie und wo bestimmte Klassen eingesetzt werden können. Wenn sich viele Entwickler mit dem System auseinandersetzen müssen, sollte man die Klassen besser durch Representation Inheritance organisieren.

Zur Illustration sollen hier zwei Beispiele für Klassenhierarchien aus [4] dienen. Die Hierarchie, die in Abbildung 2.1 dargestellt wird, ist ein Beispiel für eine Convenience Hierarchie. Die Vererbung berücksichtigt in erster Linie Implementierungsdetails wie die Datenstrukturen, auf denen die jeweiligen Collection-Klassen basieren.

Das Verhalten nach außen ist in dieser Hierarchie zweitrangig. Besonders deutlich wird das bei der Klasse `Semaphore`, die von `Queue` abgeleitet wurde. Im ursprünglichen Design enthielt `Semaphore` eine `Queue` mit den Tasks, die die Semaphore anfordern. Der Nachrichtenoverhead zwischen `Semaphore` und ihrer `Queue` war nun so groß, dass die Designer sich entschieden, Semaphore direkt von `Queue` abzuleiten, damit `Semaphore` auf die privaten Member von `Queue` zugreifen kann. Auf den ersten Blick erscheint diese Vererbung nicht sehr logisch. Es ist auch zu vermuten, dass sich eine Semaphore nicht immer wie eine Queue verhalten kann.

In Abbildung 2.2. ist eine Repräsentation Inheritance abgebildet. Die Klassen werden nach ihrem Verhalten sortiert, nicht nach ihren Implementierungsdetails. Der Benutzer dieses Frameworks kann sich sicher sein, dass sich die Klassen `Ordered Collection`, `Queue` und deren Unterklassen in ihren geerbten Methoden so verhalten, wie es in `IndexableCollection` vorgeschrieben wurde. Eine Testsuite für `IndexableCollection` könnte eine Teilmenge der Testsuites für `OrderedCollection` und `Queue` sein.

Der Begriff Liskov-konform fasst die Anforderungen an Subklassen einer Representation Inheritance konkret: *„If for each object $o_1$ of type $S$ there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $T$, the behavior of $P$ is unchanged when $o_1$ is substituted for $o_2$ then $S$ is a subtype of $T$.“*

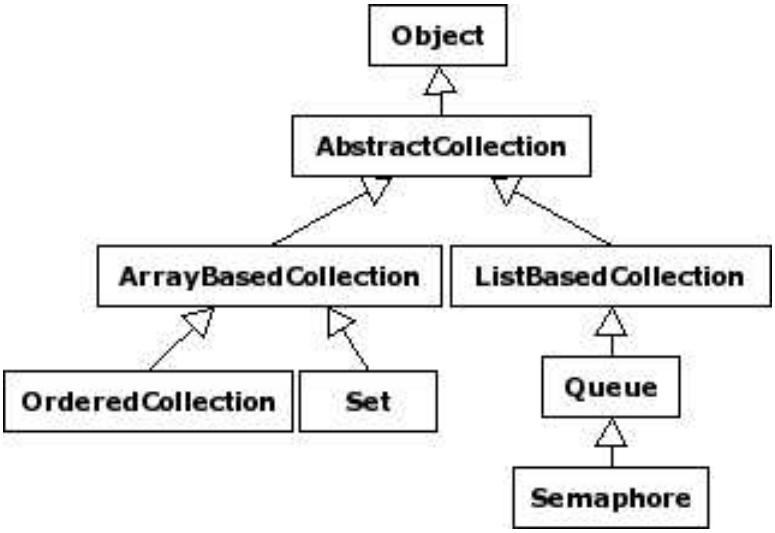


Abbildung 2.1.: Convenience Hierarchie

Weniger formal: *Eine Klasse ist eine Subklasse gemäß dem Liskov Ersetzungsprinzip, wenn sie in eine Client-Anfrage an eine ihrer Superklassen für diese eingesetzt werden kann und ihr Verhalten nach außen dem der Superklasse entspricht* [22, 23, 25].

Die Liskov-Konformität ist mit der Forderung an die Vererbung des Design by Contract Ansatzes [26] verwandt, der in der Programmiersprache Eiffel auch praktisch umgesetzt wurde [42]. Überladene Subklassenmethoden dürfen nach Design by Contract nur schwächere oder gleich starke Vorbedingungen und stärkere oder gleich starke Nachbedingungen aufweisen[4]. Weil der Begriff der Liskov-Konformität sich in diesem Zusammenhang leichter verwenden lässt, wird er hier dem Design by Contract-Begriff vorgezogen.

Der Benutzer einer Liskov-konformen Subklasse erwartet von ihr, dass sich ihre ererbten Methoden verhalten wie die ihrer Superklassen. Sie können sich natürlich in ihrer Implementierung unterscheiden; das Verhalten nach außen zu ihrem Client sollte aber dasselbe sein. Die Methode `getSmallest()` einer Superklasse `SortedContainer` sollte sowohl in einer sortierten, verketteten Liste als auch in einem Heap das kleinste Element zurückliefern.

[4] Vor- und Nachbedingungen werden in Abschnitt 2.8.1 vorgestellt.

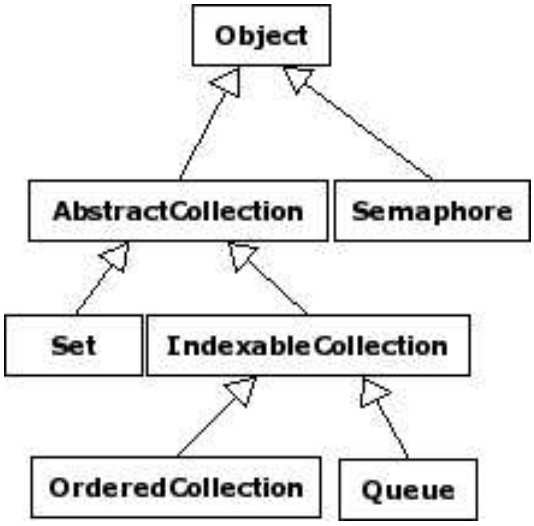


Abbildung 2.2.: Representation Hierarchie

### Testen von Liskov-Konformität

Eine Testsuite, in der Liskov-konforme Klassen getestet werden, sollte die Testfälle der Liskov-konformen Superklassen auch auf die Subklassen anwenden. Natürlich reichen diese Testfälle nicht aus, die Subklasse vollständig zu testen. Wenn sie ausreichen würden, würde die Subklasse keine Erweiterung der Superklasse darstellen und wäre damit überflüssig.

In [4] wird ein Testverfahren zum Testen Liskov-konformer Klassen vorgeschlagen. Das Verfahren schreibt vor, jede Superklassenmethode im Kontext jeder Subklasse auszuführen. Darüberhinaus wird die Vererbung von Testfällen der Superklassen-Testsuites für die Subklassen-Testsuites vorgeschlagen. Die Testfälle der Superklassen-Testsuite müssen dann in jeder Subklassen-Testsuite nochmal mit der richtigen Subklasse als Argument aufgerufen werden. Bei einfachen Hierarchien ist der Zusatzaufwand eher gering. In komplizierteren Hierarchien muss aber schon einige Zeit aufgewendet werden, in den richtigen Subklassen-Testsuites alle relevanten Superklassen-Testsuites aufzurufen. Außerdem muss man bei Änderungen in der Hierarchie auch die Testsuite-Hierarchie noch einmal durchsehen. Zumindest bei komplizierteren Hierarchien ist das ein Cross-Cutting Concern.

Aspektorientierte Quantifizierungen erlauben es, die Anwendung der Superklassen-Testsuite ohne explizite Anweisungen, Vererbungen oder sonstige Interaktionen der Subklassen-Testsuite mit der Superklassen-Testsuite, auf die Subklassen auszudehnen. Der Tester kann mit einem Aspekt dafür sorgen, dass alle Tests einer Klasse für jedes ihrer Subobjekte durchlaufen werden. Dazu genügt ein Joinpoint in der Superklassen-Testsuite, der die erste Kreation eines Subobjekts in einer Klasse, die als Testklasse identifizierbar ist, abfängt und dieses Subobjekt in einem Advice den nötigen Tests unterzieht.

Da der Aspekt über die Klassenhierarchie des Systems informiert ist, kann es niemals passieren, dass die falschen Klassen als Subklassen getestet werden. Allerdings muss jede der konventionellen Testklassen des Objekts einen Konstruktortest enthalten, der fehlerfrei durchläuft. Der Superklassen-Testaspekt braucht sein Ergebnis, damit ihm ein korrekt konstruiertes Subklassenobjekt zum Testen zur Verfügung steht.

Die Superklassen-Tests werden auf diese Weise aus dem Kontext des Subklassen-Tests im Quellcode herausgezogen. Die Entwickler der Subklassen, die für ihre eigenen Tests verantwortlich sind, müssen sich weder um die Superklassen-Tests kümmern, noch müssen sie Testhierarchien verwalten. Dass Entwickler, die sich mit einem bestimmten Concern beschäftigen, sich weniger oder gar nicht mit den Concerns anderer Entwickler beschäftigen müssen[5], wird als eine charakterisierende Eigenschaft der aspektorientierten Programmierung angesehen [7].

Kommerzielle Anbieter von Modulen und Frameworks, die über Vererbung konfiguriert bzw. ausgefüllt werden, können ihren Kunden mit einer Superklassen-Testsuite einen großen Teil der Testarbeit abnehmen. Die Suite kann zusätzlich zu dem Produkt geliefert werden und auf Wunsch auch auf Liskov-Konformität der vom Kunden selbst abgeleiteten Klassen testen.

### 2.2.2. Vor- und Nachteile des AOP Ansatzes

#### Zusammengefasste Testergebnisse

Wenn man Testfälle für Methoden zusammenfasst, sollte man sicher mit den Quantifizierungen der AOP umgehen können. Sonst können viele Testfälle der Überprüfung entgehen.

Auf den ersten Blick könnte es etwas verwirrend sein, dass der Aufruf der Testfälle in der konventionellen Testklasse ohne eine sichtbare Überprüfung des Ergebnisses vor sich geht. Die räumliche Trennung stellt ein kleines Verständnisproblem dar.

Wenn man die entsprechenden Pointcuts aber vorsichtig definiert und mit der AOP vertraut ist, wird einem durch die Zusammenfassung nicht nur Schreibarbeit abgenommen, es erleichtert einem u.U. auch die Übersicht. Hunderte von Testfällen können einer Überprüfung zugeordnet und ohne Zwischenüberprüfungen in einem Block dargestellt werden.

#### Beim Testen der Liskov-Konformität

Das automatische Mittesten der Superklassen-Eigenschaften in Subklassen durch einen Testaspekt ist eine relativ komplizierte Angelegenheit. Das Framework muss dem Superklassen-Testaspekt von irgendwoher korrekt konstruierte Subklassenobjekte liefern, die es zum Testen heranziehen kann. Dabei sollte es dieselbe Subklasse nicht mehrmals in einem Testfall testen.

Wenn das alles in einem Framework funktioniert, erhält der Tester damit nicht nur ein Werkzeug zum Unit-Testen, das ihm die Arbeit abnimmt, Methoden in Subklassen

[5] Im Englischen durch das schwer übersetzbare Wort Obliviousness ausgedrückt.

zu testen, die in Superklassen definiert wurden. Es bietet ihm zusätzlich eine Qualitätskontrolle der Vererbungshierarchie.

## 2.3. Mock Objects

### 2.3.1. Was sind Mock Objects?

Das Konzept der Mock Objects entstammt dem Unit-Test-Umfeld. Was der Begriff genau bedeutet, und wie er sich auf das praktische Unit-Testen auswirkt, wird noch kontrovers diskutiert[6].

Mock Objects ähneln Rumpf-Implementierungen, sogenannten Stubs[7]. Anders als diese beschränken sich Mock Objects nicht nur auf Trivial-Implementierungen von noch nicht fertig gestellten Funktionen. Ein Mock Object ist ein Server-Objekt, das für ausgewählte Testfälle eines Clients ein Server-Objekt ersetzt, welches vom Client aufgerufen wird. Dabei muss es folgenden Bedingungen genügen [53]:

- Es simuliert das Verhalten des Originals, auf eine Weise, die zum Testen günstiger ist als das Original-Verhalten. Meistens bedeutet das eine vereinfachte oder zeitsparende Implementierung.
- Es beobachtet, ob der Client auf die richtige Weise mit dem Mock Objekt agiert, indem es dessen Nachrichten mit vom Tester gesetzten Erwartungswerten vergleicht.

Der zweite Punkt wird häufig übersehen, ist aber sehr wichtig. Denn ohne ihn würde es sich bei dem sogenannten Mock Object nur um einen Stub handeln. Nachfolgend einige Anwendungsbeispiele für den Einsatz von Mock Objects [53, 24].

#### Unfertige Implementierungen

In manchen Situationen kommt man nicht daran vorbei, weitreichende Entscheidungen in einem Projekt, wie z.B. die Wahl der Datenbank oder einer Hardwareeinheit auf einen späteren Zeitpunkt zu verschieben. Aus Zeitgründen möchte man trotzdem schon den Rest des Projekts angehen und auch testen können. Auch wenn man ein Framework entwickelt, das unter verschiedenen Bedingungen seinen Dienst tun soll, kennt man nicht im voraus alle Situationen, die auftreten können.

Da man nicht weiß, wie sich die noch nicht genauer spezifizierten Teile des Systems verhalten, ersetzt man sie zum Testen durch Mock Objects. Die zu testende Klasse wird dahingehend getestet, ob sie auf die richtige Weise interagiert.

#### Isolierte, lokalisierte Tests

Unit-Tests sind per Definition feingranular. Sie beschränken sich darauf, Funktionalitäten einzelner Methoden zu testen. Komplexe Interaktionen der zu testenden Klasse

[6] siehe z.B. [39].
[7] Dazu siehe z.B.[4].

mit anderen können es dem Tester erschweren, geeignete Testfälle für alle Situationen zu finden. Außerdem kann es zu Performance-Problemen beim Testen kommen, wenn eines der Objekte, mit denen die Klasse kommuniziert, aufwändige Operationen ausführt.

Mock Objects erlauben es, Zustände und Situationen zu generieren, die in einem vollständig laufenden System schwer zu erreichen sind. Sie können jede Ausnahme und jede Fehlerbedingung simulieren, die man möchte. Will man automatisiert testen wie sich ein System verhält, dessen Anfrage an eine überlastete Datenbank abgelehnt wird, ist es leichter und zeitsparender, eine Mock Datenbank zu generieren, die einfach diesen Fehler für den Test zurückliefert, anstatt eine tatsächliche Datenbank zuverlässig zu überlasten und dann die Anfrage zu stellen.

Auch der Zeitfaktor macht sich im Datenbankbeispiel bemerkbar. Ein Datenbankclient, der viele Anfragen an eine externe Datenbank stellt, muss auch viele solcher Fälle testen. Da es nicht darum geht die Funktionalität der Datenbank zu testen, ersetzt man sie für den Test besser durch ein Mock Object. Dasselbe gilt für alle Tests, die aufwändige externe Aktionen involvieren, sofern die externen Aktionen nicht Gegenstand des Tests sind.

Sollten später einmal im Regressionstest Fehler auftreten, sind sie leichter zu lokalisieren, wenn das Objekt mit einfach gestalteten Mock Objects interagiert. Je mehr sich der Zuständigkeitsbereich eines Tests auf das eigentlich zu testende Objekt bezieht, desto aussagekräftiger sind die erzeugten Fehler und desto leichter ist es, ihre Ursache zu finden.

#### Nicht-deterministisches Verhalten unbeeinflussbarer Quellen

Unit-Tests haben Tests größeren Wirkungsbereich unter anderem voraus, dass sie unwahrscheinliche Spezialfälle testen können, die in einem komplexen System selten auftreten, dann aber für Probleme sorgen können [4]. Wenn diese Spezialfälle einfach nur Werte von Parametern sind, kann man sie in einem Testfall einfach testen.

Handelt es sich bei ihnen jedoch um das Verhalten von Objekten, das nicht ohne weiteres automatisiert festgelegt werden kann, z.B. Datenbanken, Netzwerkverbindungen oder Hardware, lassen sich nicht einfach nur an einer Stelle Werte einsetzen. Solche Systeme können unter Umständen auch schwer zu konfigurieren oder bereitzustellen sein.

Dem Tester bleibt nichts anderes übrig, als das Verhalten der externen Objekte für seinen Test zu simulieren. Dazu gehört natürlich auch die Überprüfung des richtigen Client-Verhaltens der zu testenden Klasse.

### 2.3.2. Konventionelle Mock Object Frameworks

Es gibt sehr viele frei verfügbare Tools zum Thema Mock Objects. Einige erzeugen neuen Code, wie MockCreator [52], andere generieren Mock Objects dynamisch, wie EasyMock [40] und DynaMock [53].

Der dynamische Ansatz hat den Vorteil, dass kein neuer Quellcode generiert werden muss, der Nachteil ist, dass sich die Möglichkeiten der Mock Object-Erzeugung nur in

dem Rahmen dessen bewegen können, was die Java-Reflection-API zulässt.

### 2.3.3. AOP Mock Objects

Die praktischen Schwierigkeiten bei der Implementierung von Mock Objects in einem Testsystem lassen sich folgendermaßen zusammenfassen:

- Das Mock Object muss zuverlässig im richtigen Kontext für das Originalobjekt eingesetzt werden.
- Das Mock Object darf nur in der Testphase eingesetzt werden. Für die Release-Version sollte es leicht zu entfernen sein.

Beide Schwierigkeiten sind in der AOP leicht zu überwinden. Aspekte erfüllen beide Voraussetzungen. Der Tester kann den Aufruf von Methoden des Objekts, das durch ein Mock Object ersetzt werden soll, mit Hilfe von Pointcuts abfangen und durch Methoden ersetzen, die nichts tun als triviale, aber passende Ergebnisse zurückzuliefern und zu prüfen, ob der Aufruf zur richtigen Zeit am richtigen Ort geschah.

Das eigentliche Objekt muss, wenn es durch ein Mock-Objekt simuliert wird, noch nicht vollständig implementiert sein. Die Methoden, die simuliert werden sollen, müssen allerdings mindestens in Rumpfform vorhanden sein, damit der Code kompiliert werden kann.

Diese Ersetzungen lassen sich so definieren, dass sie nur in einem bestimmten Kontext, z.B. in einem bestimmten Testcase, gültig sind. Wird der Testcase betreten, setzt sich der Mock-Aspekt auf einen Initialzustand, der ein Verhalten simuliert, das für diesen Testcase benötigt wird. Nach dem Verlassen des Testcases prüft der Aspekt, ob der Testcase von der Seite des aufrufenden Objekts aus fehlerfrei abgelaufen ist. Das heißt, er prüft, ob die richtigen Methoden, in der richtigen Reihenfolge, mit gültigen Parametern aufgerufen wurden. Tritt eine Abweichung vom geforderten Verhalten auf, wird der Fehler registriert.

Wer Mock Object-Tests mit Hilfe aspektorientierter Methoden einsetzen möchte, muss dafür nicht unbedingt eine aspektorientierte Sprache lernen. VirtualMock [59] ist ein Projekt in der alpha-Phase, das ein Framework bereitstellt, welches zwar intern aspektorientiert arbeitet, der Nutzer des Frameworks kann jedoch ohne Kenntnis einer aspektorientierten Sprache Mock Objects erzeugen und sie zum Testen benutzen.

### 2.3.4. Vor- und Nachteile des AOP-Einsatzes

Aspekte sind ideale Mock Objects. Da sie den Quellcode nicht verändern, sondern nur die Aufrufe an die originalen Objekte mit einem Advice umgeben, sind sie nach ihrer Entfernung aus dem Compiler-Scope nicht mehr vorhanden.

Sie verfügen zusätzlich über alle wichtigen Bestandteile einer Klasse wie Methoden und Instanzvariablen und können darum das Verhalten eines Klassenobjekts problemlos simulieren.

## 2.4. Ausnahmebehandlung im Unit-Test

### 2.4.1. Unterbrechungsfreier Testablauf

Da sich der Unit-Test in der Testhierarchie auf der niedrigsten Ebene befindet, ist die Propagierung der Ausnahmen über mehrere Ebenen hinweg von keinerlei praktischem Interesse. Die Ausnahmen werden einzeln für jede Methode mit `try...catch`-Blöcken getestet. Sollte dabei etwas nicht funktionieren, wird der Fehler in die Fehlerstatistik aufgenommen.

Häufig kommt es aber vor, dass Ausnahmen geworfen werden, mit denen auch der Tester nicht gerechnet hat. In den meisten Fällen sind das Zugriffe auf ein `null`-Objekt, dessen Initialisierung vergessen wurde. Solche Ausnahmen, die nicht gefangen werden, unterbrechen den Testlauf und werden auf die oberste Ebene weitergeleitet. Je nach Qualität des Testframeworks und der Nachvollziehbarkeit der Ausnahme, ist es mehr oder weniger schwer, den ausnahmewerfenden Code-Abschnitt zu finden. Fehler, die in der Suite nach der geworfenen Ausnahme geworfen werden, können erst in einem komplett neuen Testlauf entdeckt werden.

Der Tester kann dieses Problem vermeiden, indem er jeden seiner Testcases mit einem `try...catch`-Block umgibt, in dessen `catch`-Teil die auftretende Exception notiert und sonst nichts getan wird. Die Suite kann ungestört durchlaufen. Man sieht leicht, dass es sich bei dem Concern „Umgebe jeden Testcase mit einem `try...catch`-Block und melde unerwartete Ausnahmen“ um einen Cross-Cutting Concern handelt. Ein Aspekt, der diesen Concern realisiert, ist leicht zu programmieren, sofern die Testfälle gut quantifizierbar sind, also alle nach einem für die benutzte aspektorientierte Sprache identifizierbaren Muster quantifizierbar sind.

### 2.4.2. Vor- und Nachteile des AOP-Einsatzes

Die Maßnahme ist leicht und schnell umzusetzen. Sie kann dem Tester eine Menge Ärger beim Suchen nach Ausnahmen ersparen. Allerdings wird der Bytecode durch die vielen `try...catch`-Blöcke etwa um 10% aufgebläht.

## 2.5. Unit-Testen von aspektorientiertem Code

Thema dieser Arbeit ist das Unit-Testen von objektorientierten Code mit aspektorientierter Unterstützung; nicht das Testen von aspektorientiertem Code. Da es sich aber anbietet, wird in diesem Abschnitt ein kurzer Einblick in das Thema gegeben.

### 2.5.1. Unit-Testen aspektorientierter Programme nach Zhao

Jianjun Zhao präsentiert in [33] einen systematischen, datenflussorientierten Unit-Test-Ansatz für aspektorientierte Programme, der nähere Betrachtung verdient. In diesem Kontext interessiert der datenflussorientierte Teil weniger. Vielmehr geht es hier darum, die Schwierigkeiten zu identifizieren, die bei der Suche nach einer Abgrenzung unabhängiger Einheiten in AOP-Programmen zum Testen auftreten, und Begriffe und Verfahren zu

finden, die einen systematischen Unit-Test überhaupt erst ermöglichen. Darum werden nur die Strukturen und das allgemeine Vorgehen für den Unit-Test vorgestellt.

In einem objektorientierten System ist es einfach, zu testende Einheiten zu identifizieren. Es handelt sich um Klassen, Methoden oder Subsysteme, in denen ein mehr oder weniger gut definiertes Verhalten implementiert wird. Es mag nötig sein, für einige Testfälle kompliziertere Vorzustände herzustellen. Die Ausführung des Testfalls besteht nur aus dem Aufruf der abgegrenzten Einheit.

Aspekte können nicht so einfach isoliert getestet werden. Die Advices der Aspekte schreiben zwar ein Verhalten zu einem klar abgegrenzten Concern vor, sie stellen aber immer nur ein Teilverhalten einer Methode dar. Man kann sie auch nicht einfach aufrufen, um sie zu aktivieren. Man muss ihnen objektorientierten Code anbieten, in den sie sich hinein weben können. Ähnliches gilt für objektorientierte Methoden, auf die Aspekte wirken. Testet man sie ohne die Auswirkung ihrer Aspekte zu beachten, lässt man einen Teil des Verhaltens ungetestet. Auch wenn die Semantik des Aspekt-Codes orthogonal zur Semantik des eigentlichen Methoden-Codes ist, müssen die Ergebnisse des Aspekts als Seiteneffekte getestet werden.

Aus den vorangegangen Überlegungen folgt, dass die zu testenden Einheiten klassifiziert und gesondert behandelt werden müssen. Zhao hat dazu fünf Begriffe definiert:

**Normal Method** oder kurz n-method ist eine Methode, auf die niemals ein Aspekt einwirken wird.

**Normal Class** oder kurz n-class ist eine Klasse, auf die niemals ein Aspekt einwirken wird.

**Clustering Method** oder kurz c-method ist eine Menge bestehend aus einer Methode und allem Advice-Code, der auf sie wirkt.

**Clustering Class** oder kurz c-class ist eine Menge bestehend aus einer Klasse, sämtlichen Advice-Code, der auf Methoden der Klasse einwirkt, und Variableneinführungen durch Aspekte. Gemeint ist also eine Klasse zusammen mit allen aspektorientierten Konstrukten, die auf ihr Verhalten und ihre Struktur einwirken.

**Clustering Aspect** oder kurz c-aspect ist eine Menge bestehend aus einem Aspekt und allen Methoden und Klassen, deren Verhalten und Struktur er beeinflusst.

Das Testen von n-classes und n-methods wird hier nicht weiter betrachtet, da es sich um die konventionellen Unit-Test-Einheiten handelt.

Für den anstehenden Unit-Test der c-Einheiten definiert Zhao drei Granularitätsebenen:

**Module Testing** Ist verantwortlich für das Testen einzelner Module von c-aspects oder c-classes. Getestet werden also n-methods, c-methods oder Variableneinführungen durch Aspekte. Beim Modultesten von c-aspects müssen alle c-Methods, auf die ihre Advices wirken, separat getestet werden. Beim Testen von c-classes sind die Advice-Effekte zu berücksichtigen.

**Inter-Module Testing** Befindet sich eine Ebene über dem Module Testing. Die Module werden nicht mehr einzeln sondern im Kommunikations-Zusammenhang mit anderen Modulen desselben c-aspects oder derselben c-class getestet. Die Grenzen des c-aspects oder der c-class werden nicht überschritten.

**Aspect oder Class Testing** Wie Inter-Module Testing wird die Kommunikation verschiedener Module getestet. Dabei werden die Grenzen des c-aspects oder der c-class jedoch nicht überschritten.

Genau wie für den Unit-Test werden auch im aspektorientierten Unit-Test Treiber und Stubs benötigt.

## 2.6. Integration in eine Testmethode

Auch das beste Werkzeug ist nutzlos, wenn es nicht sinnvoll eingesetzt wird. Die vorgestellten aspektorientierten Ideen sind Hilfsmittel, die den Testprozess unterstützen sollen. Der Testprozess ist Teil eines Softwareentwicklungsprozesses und genau wie dieser systematisch. Bekanntlich gibt es viele Softwareentwicklungsprozesse von dem Entwicklungsprozess nach dem Wasserfallmodell bis zum Extreme Programming. Der Testprozess kann in den verschiedenen Modellen verschiedene Gestalten annehmen, z.B. der des monolithischen Blocks am Ende des Gesamtprozesses wie im Wasserfallmodell oder der der im eigentlichen Entwicklungsprozess immer wieder aufgegriffenen, sogar treibenden[8] Iteration wie im Extreme Programming.

Die verschiedenen Testprozesse unterscheiden sich demnach sehr darin, wie die Testsuites entstehen und sich über die Zeit verändern. Sinn und Zweck der Suite und ihre endgültige Gestalt sind sich dagegen in allen Entwicklungsprozessen sehr ähnlich. Das gilt auch für den Unit-Test, der zwar aus dem Extreme Programming kommt, aber auch in einem Wasserfallmodell Sinn ergeben würde. Das Ziel des Unit-Tests bleibt bei allen Varianten dasselbe: Alle kleinsten funktionalen Einheiten des Systems sollen in einem zufriedenstellenden (am besten objektiv festlegbaren) Maße getestet werden.

Dieser Abschnitt soll sich nun damit beschäftigen, wie eine Unit-Testsuite, die das System zufriedenstellend testet, mit aspektorientierten Methoden beim Aufbau ergänzt werden kann. Die AOP-Erweiterungen beziehen sich meistens auf die kleinstmöglichen Einheiten im System. Auch Tests, die auf Liskov-Konformität testen, lassen sich als das Testen von Methoden der Superklasse in der Subklasse, also als für die Superklasse geschrieben verstehen.

Darum sollte es für die folgenden Überlegungen gleich sein, ob man die Testsuite kleinschrittig und in steter Veränderung begriffen aufbaut, ob man am Ende einen großen Block mit allen Tests konstruiert oder die Testsuite aus einer Mischung der beiden extremen Vorgehensweisen entsteht.

Die globalen, aspektorientierten Hilfsmittel wie das Logging oder die automatische Fehlergenerierung wirken sich nicht auf den Aufbau der Testsuite im Prozess aus.

[8] Der Begriff Test-Driven-Development aus dem Extreme Programming bezeichnet diese Vorgehensweise.

### 2.6.1. Hierarchischer Aufbau

#### Aufbau konventioneller Testsuites

Eine Testsuite, die mit aspektorientierten Methoden erweitert wurde, behält ihren konventionellen, objektorientierten Kern. Nichts spricht dagegen, diesen Kern genauso aufzubauen, wie es bei rein konventionellen Unit-Testframeworks wie [51, 37] üblich ist.

Die meisten Frameworks sind darauf ausgerichtet, die Testsuites in einer abgestuften Hierarchie in Unter-Testsuites aufzuteilen. In Java könnte die erste abgestufte Hierarchieebene die Package-Ebene sein, so dass alle Testfälle einer solchen Unter-Testsuite zum selben Package gehören. Eine Ebene darunter könnten die Unter-Testsuites alle Testfälle einer Klasse fassen und so strukturiert sein, dass die Aufrufe an eine Methode auch zusammengehörig in einer Methode stehen.

Wie man sieht, ist die Struktur der Testsuite an die der Anwendung angelehnt.

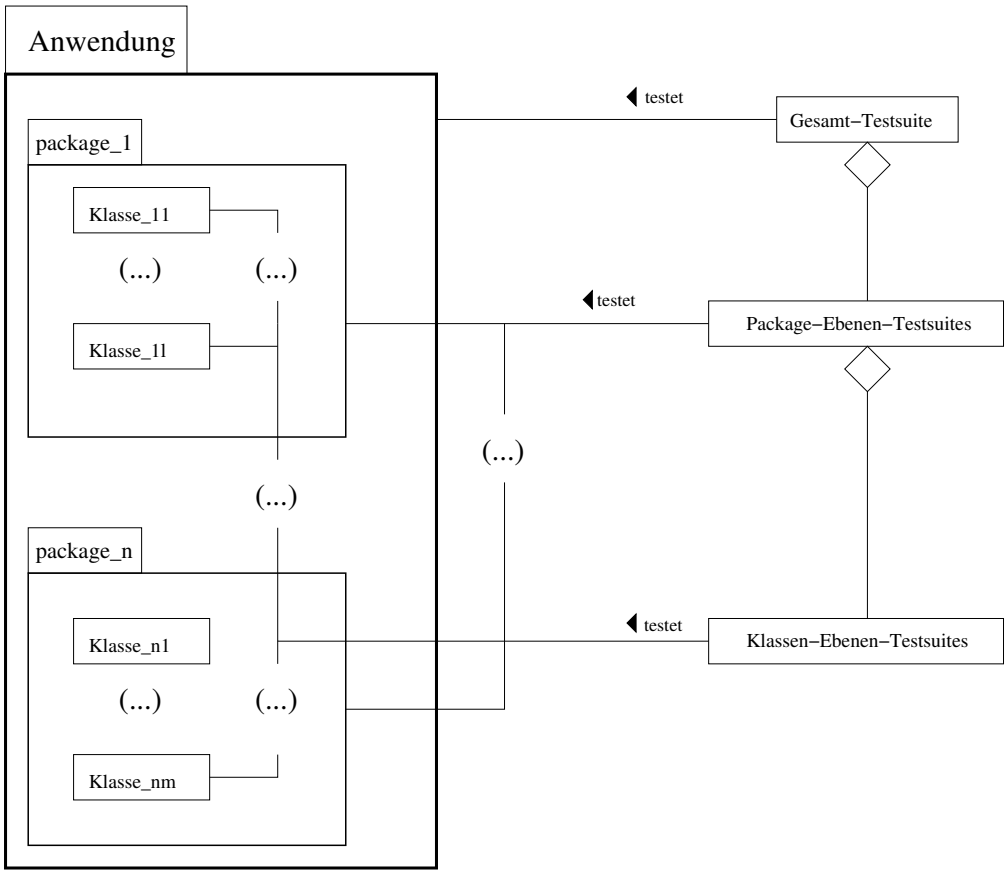


Abbildung 2.3.: Beziehung zwischen Suite und Anwendung in einer hierarchischen Testsuite

### Aufbau mit AOP-Komponenten

Damit sich die aspektorientierten Hilfsmittel reibungslos in die Hierarchie einfügen, sollten sie möglichst auf eine Weise in einer Suite organisiert sein, die zu der bestehenden konventionellen Hierarchie passt. Dafür ist es notwendig, dass die einzelnen aspektorientierten Komponenten, die als Aspekte implementiert sind, einen klar definierten Wirkungsbereich haben.

Anders als bei den Klassen der Testsuite stellen Aspekte höherer Ebene keinen Container für Aspekte niedrigerer Ebene dar. Sie haben stattdessen einen weiteren Wirkungsbereich, der jeweils zu der Hierarchie in der Testsuite passen sollte. Ausnahmen von dieser Regel sind Aspekte, die z.B. auf eine Klassenhierarchie in der Anwendung wirken und nicht auf ein ganzes Package.

Für Java lässt sich das wie oben im konventionellen Kern in drei Ebenen einteilen. Global wirkende, testende Aspekte unterstützen den gesamten Testprozess und können als der Gesamt-Testsuite zugehörig betrachtet werden. Aspekte niedrigerer Ebene unterstützen den konventionellen Kern auf Package- oder Klassenebene.

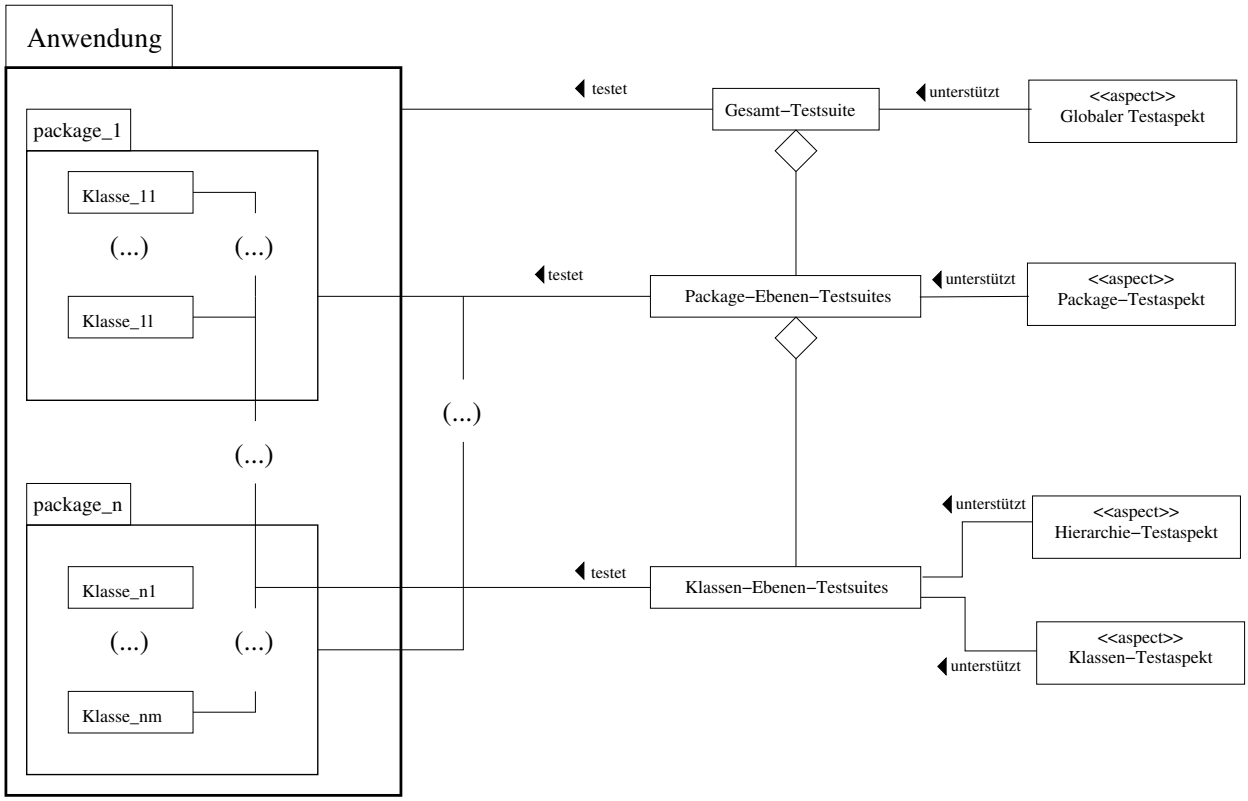


Abbildung 2.4.: Hinzufügen aspektorientierter Hilfsmittel in die Testsuite

Wenn die aspektorientierten Hilfsmittel mit wenigen Ausnahmen auf diese Weise organisiert wurden, sind die Verantwortlichkeiten klar aufgeteilt. Überschneidungen und falsche Wirkungen werden vermieden.

## 2.7. Notation in Analyse und Entwurf

Für diejenigen objektorientierten Softwareentwicklungsprozesse, die echte Analyse- und Entwurfsphasen kennen, gibt es Modellierungssprachen wie UML [57], die den Entwicklern graphische Modelle zur Verfügung stellen. Damit die aspektorientierten Ergänzungen in einen Softwareentwicklungsprozess integriert werden können, der echte Analyse- und Entwurfsphasen kennt, muss der Entwickler über geeignete Modellierungswerkzeuge verfügen, die auch aspektorientierte Konzepte darstellen können. Ohne Modelle für die aspektorientierten Konzepte entstehen in den Phasen Analyse und Entwurf Löcher, die erst in der Implementierungsphase ausgefüllt werden.

Selbst wenn man einen Softwareentwicklungsprozess einsetzt, in dem es keine explizite Analyse- und Entwurfsphase gibt (wie im Extreme Programming), sollten zumindest zu Dokumentationszwecken die wichtigeren Teile des System einmal in einem Modell dargestellt werden.

Das Ziel der aspektorientierten Modellierung ist, dass sich alle AOP-Konzepte mit den Modellierungsmitteln ausdrücken lassen, so dass die aspektorientierte Modellierung zur aspektorientierten Implementierung die Beziehung hat, die die objektorientierte Modellierung zur objektorientierten Implementierung hat. Laut [1] lassen sich AOP-Komponenten besser wiederverwenden, wenn Aspekte in einem frühen Stadium des Softwareentwicklungsprozesses identifiziert werden. Automatische Code-Generierung soll dann auch möglich sein,

Da aspektorientierte Programmierung inzwischen eine gewisse Reife erreicht hat, existieren sehr viele Arbeiten zu dem Thema. Vor allem die Arbeiten [1, 31, 28] haben das Folgende beeinflusst.

Zwei verschiedene Modellierungsansätze sollen hier vorgestellt werden; der erste nur kurz, weil er in dieser Arbeit keine Verwendung findet.

### 2.7.1. Use Case Maps

R.J.A. Buhr schlägt in „A Possible Design Notation for Aspect Oriented Programming“ [5] Use Case Maps, kurz UCMs, als Entwurfs-Modellierung vor. UCMs sind makroskopische Verhaltensbeschreibungs-Graphen. Sie stellen den Verlauf der Abwicklung eines Anwendungsfalls über mehrere Komponenten dar.

Ein UCM setzt sich aus drei Elementen zusammen:

**Responsibilities** Benannte Punkte auf einem Ausführungspfad, an denen bestimmte Aktionen im System geschehen.

**Pfade** Wege durch ein System, die den Ablauf einer größeren Aufgabe simulieren. Sie verbinden die Responsibility-Punkte.

**Komponenten** Wie der Name schon sagt, werden unter diesem Namen System-Komponenten verstanden. Damit sind weniger in der Implementierung zusammengehörige Entitäten gemeint. Es handelt sich um größere Subsysteme, die dafür verantwortlich sind, die Responsibilities umzusetzen, die sich auf ihnen befinden.

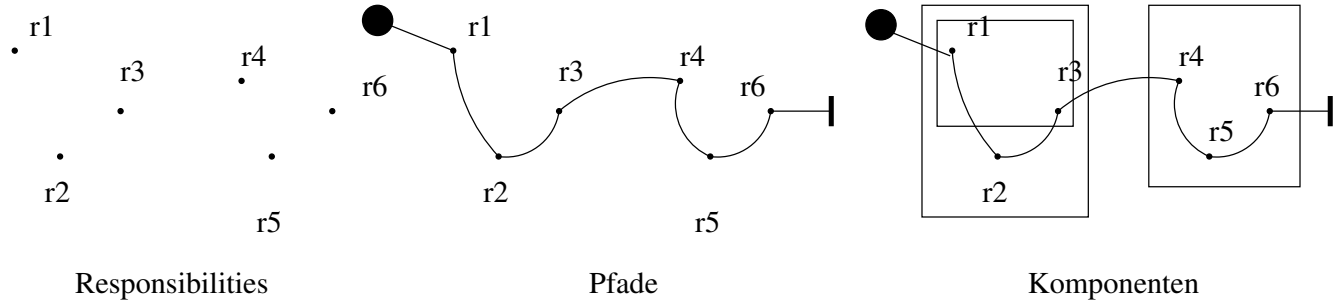


Abbildung 2.5.: Elemente von UCM-Graphen

Die Pfade können sich spalten (fork) oder miteinander verschmelzen (join), so dass auch parallel ablaufende Vorgänge beschreibbar sind:


Abbildung 2.6.: Joins und Forks in UCM-Graphen

Aspektorientierte Abläufe lassen sich jetzt beschreiben, indem die Pfade genau wie durch Klassen einfach durch die Aspekte laufen, wenn ihr gewebter Code in Aktion tritt.

Die Vorteile der UCM Graphen sind:

**Übersichtlichkeit** Der Entwickler betrachtet einen ganzen Vorgang auf sehr hoher Granularitätsebene, ohne sich mit Details beschäftigen zu müssen. Es gibt keine komplexen Beziehungen zwischen Klassen und Aspekten, sondern nur einen gut verfolgbaren Durchlauf.

**Einfachheit** UCM-Graphen sind schnell gezeichnet, da sie nur aus wenigen Elementen bestehen.

**Flexibilität** Die Granularitätsebene ist frei wählbar, solange sie hoch genug ist, um komplexe Prozesse als grobe Pfade durch ein System darzustellen.

Den Vorteilen stehen folgende Nachteile gegenüber:

**Ungenauigkeit** Wie in [5] hervorgehoben eignen sich UCMs nur für stark abstrahierte Probleme auf hoher Granularitätsebene. Lange unbeschriftete Pfade durch Klassen und Methoden, die auch noch zyklisch sein können, haben im Vergleich zu Sprachmitteln aus der UML einen niedrigen praktischen Wert.

**Unsauberkeit** Es gibt in der UCM-Modellierung keine Constraints oder sonstige Vorschriften, die eine gewisse Konsistenz vorschreiben. Es gibt auch keinerlei Vorschrif-

ten, die z.B. regeln, dass bestimmte Arten von Komponenten mit einer bestimmten Vollständigkeit modelliert werden müssen. Dabei entsteht die Gefahr, dass verschiedene Entwickler UCM-Diagramme von extrem unterschiedlicher Qualität zum selben Entwurf erstellen.

**Nicht ohne zusätzliche Werkzeuge anwendbar** UCMs können keine Vererbungshierarchien oder allgemein statische Beziehungen ausdrücken. Man braucht zusätzlich andere Modellierungssprachen wie UML.

Die aspektorientierten Hilfsmittel, mit denen sich diese Arbeit beschäftigt, greifen zumeist auf niedriger Granularitätsstufe. U.a. das macht die UCM-Notation für die Zwecke dieser Arbeit ungeeignet.

### 2.7.2. Erweiterung der UML

UML [57] ist die am weitesten verbreitete Modellierungssprache für objektorientierte Analyse und Entwurf. Sie bietet nicht nur die Möglichkeit objektorientierte Systeme zu modellieren. Sie verfügt auch über ein ausführlich spezifiziertes Meta-Modell, das auf Erweiterungen ausgelegt ist. Eine solche Erweiterung ist z.B. Real-Time-UML [34] zur Modellierung von Systemen mit Echtzeit-Anforderungen. UML-Kenntnisse, die ausreichen ein konventionelles, objektorientiertes Softwaresystem zu modellieren, werden hier vorausgesetzt. Um die Erweiterung zu verstehen, wird hier noch kurz auf die Meta-Architektur von UML eingegangen.

#### Die UML-Meta-Architektur

Die Modell-Ebene von UML, die i.A. dazu benutzt wird, objektorientierte Systeme zu modellieren, baut auf eine Meta-Ebene auf, die die Elemente definiert, die in der Modell-Ebene benutzt werden dürfen. Der Meta-Ebene liegt noch eine Meta-Meta-Ebene zugrunde, die hier aber nicht weiter interessiert[9].

Die Meta-Ebene stellt nicht nur die Standard-UML-Elemente zur Verfügung, sondern verfügt auch über Konstrukte, die Erweiterungen ermöglichen. Eine definierte, zusammengehörige Sammlung erweiterter Konstrukte wird als Profile bezeichnet.

Erweiternde Konstrukte teilen sich auf in:

**Stereotypen** Führen ein neues Modellierungselement ein, dass auf einem bereits existierenden Basiselement beruht.

**Tagged-values** Schlüssel-Wert-Paare, die eine Eigenschaft eines existierenden UML-Elements bezeichnen.

**Constraints** Regeln in der Object Constraint Language [55], kurz OCL, oder natürlicher Sprache, mit denen neue Semantiken eingeführt werden.

[9] Die Meta-Meta-Ebene definiert die Sprachmittel und deren Semantik für die Meta-Ebene. Der an dieser Ebene interessierte Leser sei auf die aktuelle UML-Spezifikation verwiesen, die er unter [57] finden kann.

### Ein Profile für AspectJ

Diese Arbeit verwendet das Profile, welches in „An UML-based Aspect-Oriented Design Notation for AspectJ“ [28] vorgestellt wird. Einerseits, weil es auf AspectJ zugeschnitten ist, andererseits weil es sehr kompakt und praktisch ist.

Die folgenden Stereotypen wurden übernommen und in dieser Arbeit verwendet:

**Pointcuts** Sie werden als spezielle Operationen mit dem Stereotyp <<pointcut>> repräsentiert. In ihrer Bedeutung sind Pointcuts Methoden zwar vollkommen unähnlich, in ihrer Struktur haben sie mit UML Operationen aber sehr viel gemein:

- Sie gehören zu einem umschließenden Block (Aspekt).
- Sie besitzen Parameter und eine Signatur.
- Etwas, das in seiner Struktur implementierungsähnlich ist, aber natürlich keine Implementierung ist, weil es sich nicht um eine Folge von Anweisungen handelt, sondern um die Beschreibung des Wirkungsbereichs des Pointcuts.

Beispiel:

$$\underbrace{pointcut\ matrix44BooleanConstructorTest\ \overbrace{(boolean\ bHom)}^{Parameter}:}_{Signatur}$$
$$\underbrace{call\ (MatrixDouble44.new(boolean))\&\&args(bHom);}_{Pointcut\ Deklaration(Implementierung)}$$

Der Pointcut-Stereotyp enthält einen tagged-value base, der seine Implementierung, also die Pointcut-Definition enthält.

Außerdem muss er einem Constraint gerecht werden. Es muss einen Advice geben, dessen tagged-value base seinen Namen als Wert enthält.

Der Beispiel-Pointcut würde innerhalb eines Aspektes in einem UML-Diagramm folgendermaßen dargestellt werden:

```
<<pointcut>> matrix44BooleanConstructorTest(bHom: boolean)
                {base=call(MatrixDouble44.new(boolean)) && args(bHom)}
```

**Advice** Wie Pointcuts werden auch Advices als Stereotyp zu Standard-Methoden repräsentiert. Der Stereotyp heißt <<advice>>. Sie sind den Standard-Methoden auch in der Semantik ähnlich, da in ihnen Verhalten als Folge von Anweisungen definiert wird. Weil sie aber keinen eindeutigen Bezeichner besitzen, können sie nicht durch Advices in abgeleiteten Aspekten überschrieben werden. Es kann auch zu Konsistenzproblemen mit der UML kommen, weil es möglich ist, mehrere Advices mit derselben Signatur in einen Aspekt zu definieren. Darum bietet es sich an Pseudo-Identifier einzusetzen.

Beispiel:

$$\underbrace{\overbrace{(adviceid099)}^{Pseudo\ Identifier}\ after\ \overbrace{(boolean\ bHom)}^{Parameter}\ returning(MatrixDouble44\ newConst)\ :}_{Signatur}$$

$$\underbrace{matrix44BooleanConstructorTest(bHom)}_{Pointcut\ Deklaration}\ \overbrace{...}^{Implementierung}$$

Die Pointcut-Deklaration wird im Diagramm als tagged-value mit dem Namen base ausgedrückt. Der Constraint des Stereotyps ist, dass es einen <<pointcut>> gibt, dessen Name der Wert von base ist.

Der Beispiel-Advice würde in einem Aspekt innerhalb eines UML-Diagramms folgendermaßen dargestellt werden:

```
<<advice>> adviceid099 after(bHom: boolean): MatrixDouble44
                {base=matrix44BooleanConstructorTest(bHom: boolean)}
```

**Aspekt** Aspekte werden als Stereotyp <<aspect>> von Standard-Klassen repräsentiert. Das wird dadurch begründet, dass sie

- als Container und Namespace für Attribute, Operationen und Pointcuts dienen.
- an Vererbungs- und allgemeinen Beziehungen beteiligt sein können.

Sie besitzen zusätzlich drei tagged-values:

- Einen Aufzählungstyp instantiation, der die Instanziierungsart never (für abstrakte Aspekte), perJVM (global), perobject[10] oder percontrolflow[11] angibt.
- Einen booleschen Wert privileged, der wahr ist, wenn es sich um einen privilegierten Aspekt handelt.
- Einen String für den Basispointcut base, der bei perobject- oder percontrolflow-Aspekten den Instanziierungspointcut angibt. Bei global definierten Aspekten enthält base den String „undefined“.

Ein Constraint legt fest, dass der Stereotyp <<aspect>> nur Standard-Operationen und abstrakte Pointcuts durch Vererbung überschreiben darf.

Der Advice in Abbildung 2.7 enthält den oben vorgestellten Pointcut und seinen Advice.

**Weaving Relation** Zur Darstellung des aspektorientierten Webens von Aspekt-Code wird eine neue Relation eingeführt. Es handelt sich um eine wirkliche Neuerung, nicht um einen Stereotyp. Die Relation wird als gestrichelter Pfeil dargestellt, der

[10] Bedeutet in AspectJ: Der Aspekt wird einem Objekt zugeordnet, mit dem er erzeugt und vernichtet wird. Seine Pointcuts beziehen sich nur auf dieses Objekt.

[11] Bedeutet in AspectJ: Der Aspekt wird einem Kontrollfluss zugeordnet, mit dem er erzeugt und vernichtet wird. Seine Pointcuts beziehen sich nur auf diesen Kontrollfluss.

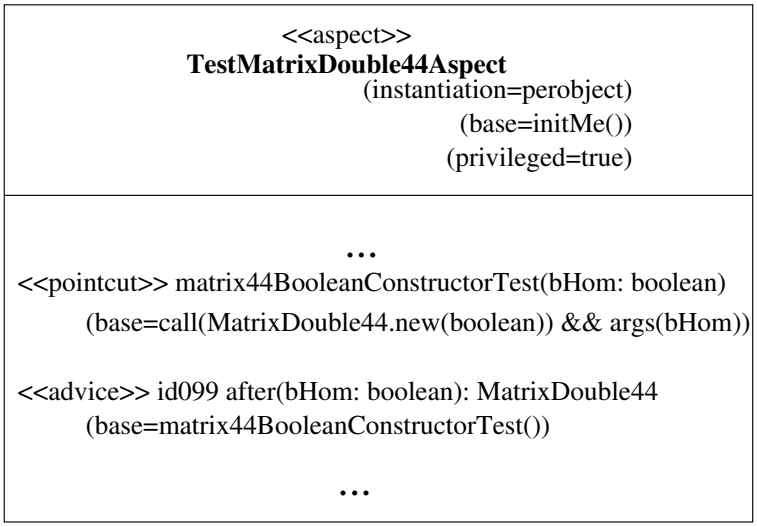


Abbildung 2.7.: Aspekt-Stereotypen in UML

vom Aspekt ausgeht[12] und auf die Basisklasse, auf die gewebt wird, zeigt. Der Pfeil ist oben mit dem Schlüsselwort <<crosscut>> gekennzeichnet.

### Sequenzdiagramme

Sequenzdiagramme beschreiben die Kommunikation zwischen Objekten auf einem Ausführungspfad. Stellt man die Aspekte in einem Sequenzdiagramm dar, als wären sie gewöhnliche Klassen, gehen keine Informationen verloren, wie das in einem Klassendiagramm der Fall wäre. In Klassendiagrammen wäre ohne das Herausstellen der Aspekteigenschaften mit speziellen Stereotypen und der Weaving Relation nicht klar, wie sich die Aspekte auf ihre Basisklassen auswirken.

In einem Sequenzdiagramm dagegen wird der reine Kommunikationsfluss dargestellt. Wo Aspekte aktiv werden, muss gezeigt werden. Die Aspekte können behandelt werden, als wären sie konventionelle Klassen, müssen aber durch <<aspect>> namentlich als Aspekte gekennzeichnet werden.

### Andere Diagramme

Auf andere Diagramme der UML wird hier nicht weiter eingegangen. Zum Verständnis des Frameworks reichen die Klassendiagramme aus, um seine statische Struktur darzustellen. Die dynamischen Abläufe können mit Sequenzdiagrammen beschrieben werden.

[12] Der Übersicht halber gehen die Pfeile bei Diagrammen in dieser Arbeit von ihren Aspekten aus, so wie konventionelle Beziehungen von ihren Klassen ausgehen und nicht von ihren Methoden, wie in [28] vorgeschlagen.

## 2.8. Über Unit-Testen hinaus

Das Hauptinteresse dieser Arbeit liegt beim Unit-Testen. Im folgenden Kapitel wird trotzdem ein Blick auf die Anwendungsmöglichkeiten der AOP auf andere Testvarianten mit größerem Wirkungsbereich geworfen, weil die AOP in einigen dieser Varianten mit geringem Aufwand vieles erleichtern kann.

### 2.8.1. Zusicherungen (Assertions)

#### Was sind Zusicherungen?

Zusicherungen (engl. Assertions) [4] sind nicht im eigentlichen Sinne Tests. Es handelt sich bei ihnen um eine Qualitätssicherungsmaßnahme, die sich im Applikations-Code befindet. Sie bezeichnet eine Bedingung, die nicht verletzt werden darf, wenn sich die Anwendung in einem konsistenten Zustand befindet. Die Evaluierung kann vornehmlich aus Performance-Gründen im Release ausgeschaltet werden. Da sich die Zusicherungen im zu testenden Programmcode selbst befinden, spricht man von built-in-tests.

Eine Zusicherung besteht aus drei Teilen:

**Boolescher Ausdruck** Der Ausdruck, der in einem konsistenten System immer wahr sein muss.

**Aktion** Die Aktion, die die Zusicherung auslösen soll, wenn der boolesche Ausdruck als falsch evaluiert wurde. In der Hauptsache wird hier festgelegt, ob die Ausführung des Programms nach Verletzung der Zusicherung fortgesetzt und eine Meldung abgesetzt wird oder ob das Programm halten soll.

**Enable/Disable-Funktion** Die Zusicherungsevaluierung muss sich zentral, z.B. über eine Compiler-Option, abschalten lassen.

Eine Zusicherung wird als stärker relativ zu einer anderen bezeichnet, wenn sie nur Werte aus einer Menge $P$ akzeptiert, für die gilt

$$P \subset Q$$

und $Q$ die Menge der Werte bezeichnet, die die relativ schwächere Zusicherung akzeptiert. Über Zusicherungen mit Akzeptanzmengen, in der eine nicht vollständig in der anderen enthalten ist, können keine Aussagen über Stärke und Schwäche getroffen werden.

Zusicherungen treten in der Literatur [4] als Vorbedingung, Nachbedingung, Schleifeninvariante, Schleifenvariante und Klasseninvariante auf.

#### Vorbedingung

Zusicherungen, die zu Beginn einer Operation ausgewertet werden, bezeichnet man als Vorbedingungen. Wenn die Vorbedingung einer Operation nicht erfüllt ist, wird eine korrekte Ausführung der Operation nicht gewährleistet. Die Vorbedingung kann Wertebereiche, die eingehalten werden müssen, oder Zustände, in denen sich Objekte befinden

müssen, enthalten. Wird die Vorbedingung verletzt, liegt der Fehler beim aufrufenden Client, der den Server so nicht hätte aufrufen dürfen.

In einer Liskov-konformen Klassenhierarchie darf die Vorbedingung einer Subklasse nicht stärker sein als die ihrer Superklasse. Die Subklasse muss mindestens dieselbe Wertemenge akzeptieren wie ihre Superklasse.

### Nachbedingung

Nachdem eine Operation eines Servers abgeschlossen worden ist, prüft eine Nachbedingung die Gültigkeit ihres Ergebnisses. Dabei handelt es sich meistens um eine Beziehung zwischen Eingabeparametern, Startzustand und dem Rückgabewert mit Folgezustand. Sollte die Nachbedingung verletzt worden sein, ohne dass die Vorbedingung verletzt wurde, liegt der Fehler beim Server, da er unter den richtigen Bedingungen aufgerufen wurde, seine Operation aber trotzdem eine unerwünschte Wirkung hervorrief.

Häufig wird z.B. die Nachbedingung gefordert, dass sich keine Membervariablen der Klasse durch die Operation ändern, wenn es in der Sprache nicht möglich ist, so etwas per Schlüsselwort zu vermeiden.

In einer Liskov-konformen Klassenhierarchie darf die Nachbedingung einer Subklasse nicht schwächer sein als die ihrer Superklasse. Die Subklasse darf sich vom Ergebnis her nicht anders verhalten als ihre Superklasse.

### Schleifeninvarianten

Schleifeninvarianten sind Forderungen an eine Schleifen, bestimmte Werte konstant zu halten, und zwar

- bevor die Schleife aufgerufen wurde.
- nach jeder Iteration.
- wenn die Schleife verlassen wird.
- wenn keine Iteration stattfindet.

### Schleifenvarianten

Schleifenvarianten enthalten in ihrem booleschen Ausdruck einen ganzzahligen Ausdruck, der jede Iteration ausgewertet wird. Nach jeder Iteration muss sich sein Wert verringern. Er darf aber nie unter null fallen. Der Zweck der Schleifenvariantenzusicherung ist es, zu prüfen, dass die Schleife in ihren definierten Grenzen bleibt und nicht irgendwo unkontrolliert ausbricht. Im Hoare-Kalkül dient sie auch dazu zu beweisen, dass eine Schleife terminiert.

### Klasseninvarianten

Klasseninvarianten sind Bedingungen, die den Zustand eines Objekts einer Klasse bezeichnen, und wahr sein müssen

- nach der Instanziierung des Objekts.
- vor dem Eintreten in eine Methode (außer bei klasseninternen Aufrufen).
- nach dem Austreten aus einer Methode (außer bei klasseninternen Aufrufen).
- unmittelbar vor der Zerstörung des Objekts.

Man kann, wenn die jeweilige Sprache es nicht anders zulässt, Klasseninvarianten realisieren, indem man die Vor- und Nachbedingung einer jeden Methode der Klasse mit dem die Klasseninvariante bezeichnenden Ausdruck mit einem logischen AND verknüpft. Man muss dann noch darauf achten, dass die persistenten Objekte der Klasse nicht anders als über ihre Methoden manipuliert werden können (Kapselung).

In einer Liskov-konformen Klassenhierarchie darf die Klasseninvariante der Subklasse nicht schwächer sein als die ihrer Superklassen.

#### Warum Zusicherungen?

Der Einsatz von Assertions wird unter anderem aus folgenden Gründen empfohlen [4]:

- Fehlern wird durch die Artikulation der Voraussetzungen und Folgen einer Operation vorgebeugt.
- Zusicherungen können die Dokumentation zwar nicht ersetzen, sie aber unterstützen.
- Unit-Tests werden nur auf der niedrigsten Ebene eingesetzt. Assertions bemerken Fehler auf niedriger Granularitätsstufe auch im Integrationstest.
- Testcases für den Unit-Test lassen sich leicht aus den Zusicherungen herleiten.
- Zusicherungen können als Verträge interpretiert werden, die die Wiederverwendung von Klassen und Methoden vereinfachen[13].
- Sie sind in den meisten Sprachen leicht zu schreiben.

#### Klasseninvarianten mit AOP

Vor- und Nachbedingungen, Schleifenvarianten und -invarianten lassen sich mit den meisten konventionellen Sprachmitteln einfach ausdrücken. Eine Überprüfung der Invariantenbedingung durch ein `if... else...`-Konstrukt reicht dafür aus. In einigen Fällen kann es zu Problemen mit der Kapselung kommen. Z.B. wenn die Nachbedingung den Zustand des Objekts vor der Ausführung der Methode zum Vergleich benötigt. Diesem Problem kann mit privilegierten Aspekten[14] abgeholfen werden.

Klasseninvarianten sind bei Sprachen, die Zusicherungen nicht als Sprachmittel unterstützen, dagegen ein größeres Problem. Die Punkte, in denen die Klasseninvariante geprüft wird, sind Erzeugung und Zerstörung eines Objekts, Eintritt in eine Methode

[13] Design by Contract [26].
[14] Siehe dazu Kapitel 2.1.3.

und Austritt aus einer Methode. In konventionellen Sprachen müsste an jedem dieser Punkte eine Klasseninvariantenüberprüfung stattfinden.

Bei diesen Punkten handelt es sich um Joinpoints und darum eignet sich die AOP ausgezeichnet für diese Aufgabe, da es sich beim Überprüfen von einer Bedingung an allen Ein- und Austrittspunkten einer Klasse um einen leicht beschreibbaren Cross-Cutting Concern handelt.

Die Vor- und Nachbedingungen können, wenn es nötig sein sollte, weil die Kapselung durchbrochen werden muss, auf dieselbe Weise auch im selben Aspekt für einzelne Methoden implementiert werden.

### 2.8.2. Mutationstests

Testen ist eine Aktivität, die niemals vollständig abgeschlossen ist. Man kann von einem größeren Softwaresystem nie sicher sagen, dass es fehlerfrei ist. Deswegen ist es aber nicht unmöglich, die Qualität der Testsuite auf bestimmte, als qualitätsbegünstigend eingeschätzte Metriken abzubilden. Code-Überdeckung ist z.B. eine dieser Metriken,

Eine andere Möglichkeit die Qualität der Tests zu beurteilen, ist der Mutationstest [19, 17]. Mutationstests testen eine Testsuite, indem sie den zu testenden Code verändern und danach prüfen, ob die Testsuite den Fehler entdeckt. So lassen sich Anhaltspunkte dafür finden, dass die Testcases die Implementierung nicht gründlich genug testen.

Das Tool Jester [50] automatisiert den Mutationstest. Es lässt eine JUnit-Testsuite einmal durchlaufen. dann ändert es unabhängig voneinander Zeilen im Sourcecode und lässt die Suite für jede Änderung noch einmal durchlaufen. Jede nicht bemerkte Änderung wird notiert und dem Tester am Ende präsentiert.

Manuelle erzeugte Mutationstests lassen sich mit Hilfe der AOP leicht in den Sourcecode weben. Ähnlich wie bei Mock Objects fängt man spezielle Ereignisse wie Methodenaufrufe in dafür spezifizierten Pointcuts ab. Voraussetzung dafür ist, dass die interessierenden Stellen Joinpoints in der jeweiligen Sprache sind.

## 2.9. Zusammenfassung

In den Abschnitten dieses Kapitels wurde gezeigt, dass aspektorientierte Konzepte neue Lösungsansätze für ausgewählte Probleme des Unit-Testens liefern können.

Die Ideen waren im Einzelnen:

- Instrumentierung und gezielte Durchbrechung der Kapselung mit privilegierten Aspekten. Dazu gehört auch die automatische Fehlergenerierung über die AOP-Reflexionsfähigkeiten.
- Das Zusammenfassen von Testfällen, die dasselbe Ergebnis liefern sollen, und Testen auf Liskov-Konformität von Subklassen.
- Der Einsatz von Aspekten als Mock Objects.
- Die Allgemeine Behandlung unerwartet auftretender Ausnahmen.

- Der Einsatz von Aspekten als Implementierung von Zusicherungen und als Mutationstest-Hilfsmittel.

Darüber hinaus wurden Vorschläge zur Integration der AOP in eine Testmethode und zur Notation in Design und Analyse angeführt, die für den praktischen Teil der Arbeit wichtig sein werden.

Unit-Testen von aspektorientiertem Code ist zwar nicht Thema dieser Arbeit. Dennoch wurde auch in dieses Thema ein Einblick gegeben, da Unit-Testen mit Aspektorientierung von objektorientierten Code natürlicherweise auch die Frage nach dem Unit-Testen von aspektorientiertem Code aufwirft.

# 3. Das Framework, die praktische Umsetzung

Im vorigen Kapitel wurden einige Einsatzgebiete für aspektorientierte Sprachen auf dem Gebiet des Unit-Testens vorgestellt. Dabei wurde häufig betont, dass ihr Einsatz den Unit-Test einfacher und übersichtlicher gestalten kann. Um diese Behauptung zu stützen, ist zu dieser Arbeit ein Unit-Testframework entstanden, welches es gestattet, die im letzten Kapitel vorgestellten Ideen praktisch auszuprobieren.

## 3.1. Übersicht über das Framework

### 3.1.1. Wozu ein Framework?

Aspektorientierte Sprachen bieten von sich aus die Sprachmittel an, die nötig sind, die Ideen des letzten Kapitels umzusetzen. Zur bloßen Umsetzung dieser Ideen ist kein Framework nötig.

Andererseits sollte das Testen eines Softwaresystems ein möglichst systematischer Vorgang sein. Um diese Bedingung zu erfüllen, muss ein Testsystem für eine Anwendung, hierarchisch und, soweit es geht, an den Aufbau der zu testenden Anwendung angelehnt sein. Zusätzlich sollte das Testsystem Funktionalitäten zum Aufnehmen von Statistiken, zum Logging und Sicherheitsmechanismen bieten, die dafür sorgen, dass bestimmte Zusatzfunktionalitäten (wie Aspekte) auch wirklich eingesetzt werden. Außerdem ist es angenehmer für den Tester, wenn er auf ein erprobtes Framework zurückgreifen kann und nur noch die Testfälle definieren und den Testlauf starten muss.

Deswegen gibt es zum Aufbau konventioneller Tests Frameworks wie z.B. JUnit für Java [51] oder das Testframework der boost-Bibliothek für C++ [37].

Das für diese Arbeit erstellte Framework hätte JUnit für den konventionellen Teil übernehmen und mit aspektorientierter Zusatzfunktionalität ausstatten können. Bei JUnit handelt es sich aber um ein größeres Projekt, dessen Gesamtfunktionalität nicht benötigt wird und teilweise sogar störend wirkt. Die Ideen, die im theoretischen Teil vorgestellt wurden, kollidieren teilweise mit den JUnit Strukturen. Besonders die, die sich auf feingranulare Funktionalitäten auswirken.

Für dieses eher zu Demonstrationszwecken geschriebene Framework wiegt das Mehr an Freiheit und die Flexibilität verschiedene Konzepte auszuprobieren, die Vorteile des schon existierenden Frameworks JUnit auf. Die Programmierung des konventionellen Teils eines einfachen Testframeworks ist auch keine allzu schwere Aufgabe. Darum wurde auf JUnit vollständig verzichtet und das gesamte Framework von Grund auf neu implementiert.

### 3.1.2. Sprache

AspectJ [36] ist unter den aspektorientierten Sprachen die bekannteste und verbreitete. Sie wurde mit der Zusammenarbeit des AOP-Pioniers Gregor Kiczales entwickelt und unterhält eine große Entwicklergemeinde, die sich mit ihrer Weiterentwicklung und Stabilisierung beschäftigt. In ihrer Entwicklung ist sie schon schon so weit, dass bereits lehrbuchartige Literatur [20] über sie existiert. Sie lässt sich auch leicht in die Entwicklungsumgebung eclipse [41] integrieren, was ihre Verwendung besonders angenehm macht.

Die Entscheidung fiel also auf die AOP-Sprache AspectJ, was bedeutet, dass mit dem Framework Java-Code getestet werden kann.

## 3.2. Die getesteten Klassen

Als Anwendungsbeispiel wurden drei Java-Packages erstellt, die mit dem Testframework getestet wurden.

Da es sich beim Quellcode dieser Klassen zum größten Teil um mathematische Standard-Implementierungen ohne komplizierte Optimierungen handelt, die mit dem Thema dieser Arbeit nichts zu tun haben, wurde er im Anhang nicht abgedruckt.

### 3.2.1. `linearAlgebra`, Abbildung A.1

linearAlgebra besteht aus zwei Klassenhierarchien: Der Vektorhierarchie und der Matrixhierarchie.

Die Superklassen Vector und Matrix deklarieren abstrakte und definieren konkrete Methoden, die in all ihren Unterklassen sinnvoll sind. Die abstrakte Methode getRow(int n) ist z.B. eine Methode, die in all ihren Subklassen implementiert sein sollte.

Die Subklassen sind Spezialfälle bezüglich enthaltenen Datentyp- und Elementanzahl der Matrizen und Vektoren. Sie unterscheiden sich damit natürlich auch in ihrer Funktionalität. Eine `normalize()`-Funktion ist bei einem Vektor bestehend aus vier `doubles` sinnvoll, bei einem aus vier `ints` nicht.

Die Methoden der Subklassen werden aus Platzgründen in der Abbildung nicht dargestellt.

### 3.2.2. `misc`

Dieses Package enthält Hilfsmittel zur Ganz- und Fließpunktzahlenarithmetik und Zufallszahlenerzeugung.

Da es nur die Klasse `ConstsAndStatics` mit statischen Werten und Funktionen enthält und diese in keiner dauerhaften Beziehung zu anderen Klassen steht, wurde auf eine grafische Darstellung im Anhang verzichtet.

`ConstsAndStatics`

Im `linearAlgebra`-Package werden häufig Fließkommazahlen verglichen oder Zufallswerte benötigt. Auch in anderen Packages einer Anwendung können diese Funktionalitäten unabhängig von der linearen Algebra benötigt werden.

ConstsAndStatics enthält einen Zufallsgenerator und verschiedene statische Methoden, die sich um den Zahlenvergleich kümmern.

### 3.2.3. `applicationUnderTest`, Abbildung A.6

Das `applicationUnderTest`-Package wurde geschrieben, um Testszenarien für verschachtelte Abhängigkeiten und Mock Objects zu liefern.

`MovingObject`

`MovingObject` ist ein bewegliches Objekt im Raum, dessen Position durch eine 4x4 Matrix beschrieben wird und seine Position durch Matrizentransformationen verändert.

`Probe`

`Probe` ist eine Subklasse von `MovingObject`. Die `Probe` bewegt sich in einem dreidimensionalen Raum und kann Nachrichten versenden.

`Radio`

`Radio` simuliert eine Hardware-Interface-Klasse. Andere Klassen z.B. `Probe` können über `Radio` Nachrichten verschicken, was immer mehr oder weniger lange dauert. Um das korrekte Verhalten von `Probe` ohne das nicht-deterministische und zeitraubende Verhalten von `Radio` zu testen, wird ein Mock Object benötigt, das das Verhalten von `Radio` simuliert und dabei auch das korrekte Verhalten von `Probe` prüft.

## 3.3. Statischer Aufbau des Frameworks

Der Aufbau des Frameworks ist auf den Abbildungen A.2 und A.3 im Anhang graphisch dargestellt. A.2 stellt den Aufbau der Klassenhierarchie dar, A.3 den Aufbau der Aspekthierarchie.

In den nächsten beiden Abschnitten werden die Aufgaben der einzelnen Klassen und Aspekte des Frameworks Klasse für Klasse und Aspekt für Aspekt besprochen.

### 3.3.1. Klassenhierarchie, Abbildung A.2

Der konventionelle Klassenteil des Frameworks kann ohne den AOP-Teil verstanden werden. Es ist aber nicht sinnvoll, wenn auch möglich, ihn ohne den Aspekt-Teil einzusetzen. Das Framework wurde für den Einsatz mit AOP entworfen und ist darum umständlicher zu benutzen, wenn der aspektorientierte Teil fehlt.

`TestEntity`

`TestEntity` ist die abstrakte Superklasse von `TestCase` und `TestAspectEntity`. In ihr werden `check()`-Funktionen definiert, die sowohl für die Klasse `TestCase` als auch für die Testaspekte der Aspekthierarchie von Nutzen sind. Die Aufgabe der `check()`-Funktionen ist das Überprüfen einer Bedingung, deren Verletzung automatisch als Fehler erkannt und in die Statistik aufgenommen wird.

Die `check()`-Funktionen alleine können ohne die aspektorientierten Zusatzfunktionalitäten keinen hilfreichen Fehlerstring generieren. Der Tester sollte daher, wenn er auf die AOP verzichten möchte, nicht die Funktion benutzen, die nur einen booleschen Ausdruck als Parameter erhält, sondern stattdessen diejenige, die zusätzlich noch einen Fehlerstring annimmt.

Jede Instanz einer Klasse, die von `TestEntity` abgeleitet wurde, enthält eine Referenz auf die Singleton[1]-Instanz von `TestStatus` (s.u.).

`TestCase`

Die Klassen, die den konventionellen Teil des Unit-Tests übernehmen, werden von der abstrakten Klasse `TestCase` abgeleitet und sollten immer genau einer Klasse, die von ihr getestet wird, zugeordnet sein. Die abgeleitete Klasse muss die abstrakte Methode `run()` implementieren, in der sämtliche Tests ausgeführt werden.

Die Klasse `TestCase` hat nicht die Bedeutung, die ihre wörtliche Übersetzung suggeriert. Es handelt sich nicht um einen einzelnen Testfall für eine Methode, sondern um eine Sammlung von Testfällen für eine Klasse.

Es empfiehlt sich zur Systematisierung, Übersicht und vor allem, um den Einsatz von AOP-Quantifizierungen zu erleichtern, die von `TestCase` abgeleiteten Klassen nach einem festen Schema zu benennen. `Test-Name-der-zu-testenden-Klasse` wäre z.B. eine Möglichkeit.

Falls zu einer `TestCase`-Subklasseninstanz ein Aspekt gehören sollte, muss er mit `initTestAspect()` an geeigneter Stelle aktiviert werden. Existiert zum Zeitpunkt der Aktivierung kein wirksamer `TestAspect` für diese Klasse, wird eine `NoCorresponding-AspectException` geworfen.

`TestSuite`

Eine `TestSuite` enthält `TestCase`s, die strukturell oder aus anderen Gründen zusammengehören. Eine `TestSuite` könnte z.B. alle `TestCases`, die ein bestimmtes Package testen enthalten. `TestCases` werden mit `addTestCase()` hinzugefügt. Die `run()`-Methode lässt alle `TestCase`s durchlaufen.

Typischerweise wird sie vom Tester aufgebaut, einem `TestSupervisor` hinzugefügt und bei dessen Durchlauf von ihm angesprochen. Der Tester kann auch eine isolierte `TestSuite` laufen lassen, die zu keinem `TestSupervisor` gehört. Um globale Einstellungen, die normalerweise `TestSupervisor` durchsetzen würde, muss sich der Tester dann

[1] Zum Begriff Singleton-Klasse siehe z.B. [10].

allerdings selbst kümmern.

Befinden sich `TestCompleteInheritance`-Aspekte im Scope werden neue Instanzen von ihnen mit jedem neuen `TestSuite`-Lauf erzeugt und nach jedem beendeten Lauf zerstört.

#### `TestSupervisor`

Die Klasse `TestSupervisor` enthält alle `TestSuite`s für einen Testlauf und allgemeine Einstellungen. Der Tester kann in einen `TestSupervisor` verschiedene `TestSuite`s oder mehrere Instanzen der gleichen `TestSuite` über die Methode `addTestSuite()` einhängen. Wird der Testlauf mit `runTests()` gestartet, laufen alle `TestSuite`s je einmal durch. Dabei wird automatisch den Einstellungen des `TestSupervisor`s entsprechend geloggt und Fehlerstatistiken erstellt.

#### `TestFailure`

`TestFailure` ist die abstrakte Superklasse für `TestError`, `TestWarning` und `TestUnexpectedException`. Alle diese drei Subklassen erben von ihr einen Beschreibungsstring über den Ort des Fehlers und einen String für weitere Kommentare. Der Kommentarstring ist optional, der String, der den Ort beschreibt, muss nicht unbedingt, sollte aber angegeben werden.

#### `TestError`

Ein `TestError` repräsentiert einen gescheiterten Testfall in einer Subklasse von `TestCase`. `TestErrors` werden meistens von einer `check()`-Methode erstellt, die sie automatisch der Statistik hinzufügt.

#### `TestWarning`

Eine `TestWarning` ist eine kleinere Fehlfunktion oder Ineffizienz, die so nicht oder nur selten auftreten sollte. Das System erfüllt im Falle ihres Auftretens zwar seine Aufgabe, der Tester ist aber mit einer Ablaufeigenschaft unzufrieden.

Wenn eine Berechnung in Ausnahmefällen länger dauert als erwartet und eine kurze Berechnungsdauer keine kritische Eigenschaft des zu testenden Systems ist, könnte man beispielsweise für diesen Fall eine `TestWarning` generieren.

#### `TestUnexpectedException`

`TestUnexpectedException`s bezeichnen Ausnahmen, die während eines Testlaufs auftreten und mit denen nicht gerechnet wurde. Sie werden automatisch vom Framework erzeugt und in die Statistik aufgenommen, falls eine nicht gefangene Ausnahme in einem `TestCase` oder `TestAspect` auftritt.

Beispielsweise könnte im Test ein `null`-Objekt, das referenziert wird auftreten und eine Ausnahme auslösen. Würde eine solche Ausnahme nicht abgefangen, würde der Testlauf an dieser Stelle abbrechen.

Das Framework fängt diese Ausnahmen über den Aspekt `TestSupervisorAspect`.

`TestStatus`

`TestStatus` ist eine Singleton-Klasse. Es existiert also nur eine Instanz von ihr in jedem Testsystem. Sie erfasst Fehler, Warnungen und unerwartet geworfene Ausgaben. Weil sie eine Singleton-Klasse ist, können alle anderen Klassen im Framework eine Instanz von ihr unter der Garantie, dass es die einzige richtige ist, benutzen. Dadurch wird vermieden, dass gemeldete Fehler in einer falschen Instanz landen.

Zusätzlich zu den entdeckten Fehlern, Ausnahmen und Warnungen wird in der Klasse noch über den Ablauf des Tests Buch geführt.

Die Klasse `TestStatus` bleibt dabei immer passiv. Sie wird von Subklassen von `TestEntity`, `TestSupervisor` und `TestSupervisorAspect` angesprochen.

Am Ende eines Testlaufs lassen sich alle relevanten Informationen einzeln abfragen oder in einem auf deutsch ausformulierten String zusammenfassen.

`TestLogger`

Die Singleton-Klasse `TestLogger` übernimmt die Ausgabe der Logging Strings. Die Nachrichten werden im `TestLoggerAspect` oder auch explizit zu den richtigen Zeitpunkten erzeugt und dann an `TestLogger` mit einem Integer, der die Priorität angibt, geschickt.

Überschreitet die Priorität die Mindestpriorität der `TestLogger`-Instanz, die sich mit `setMinPriority()` setzen lässt, wird die Nachricht an eine Instanz eines Subtyps der Java-Klasse `Writer` geschickt. Normalerweise ist das `System.out`.

`NoCorrespondingAspectException`

Diese Ausnahme wird geworfen, wenn in einem `TestCase` mit `initTestAspect()` ein `TestAspect` aktiviert werden soll, dieser aber nicht existiert.

### 3.3.2. Aspekthierarchie, Abbildung A.2

Die Aspekthierarchie, dargestellt in Abbildung A.2, ergänzt die Klassenhierarchie des Testframeworks um die Aspekte, die nötig sind, die im theoretischen Teil vorgestellten Ideen umzusetzen. Ihre Klassen erleichtern auch andere Vorgänge im Ablauf des Frameworks, so dass ihr Einsatz auch ohne explizite Nutzung aspektorientierter Testansätze von Nutzen ist.

`TestAspectEntity`

Der abstrakte Aspekt `TestAspectEntity` leitet sich von `TestEntity` ab und ergänzt seine Superklasse um Methoden zur automatischen Fehlererzeugung mit Reflexions-Informationen, die die konventionellen Java-Reflection-Klassen nicht bieten.

Genutzt werden diese Fähigkeiten von ihren beiden Subklassen `TestAspect` und `TestCompleteInheritanceAspect`.

### TestAspect

TestAspect ist eine abstrakte Klasse, von denen der Tester konkrete Subklassen ableiten kann. Ein TestAspect wird genau einem TestCase zugeordnet. Sein Wirkungsbereich ist auf diesen TestCase eingeschränkt. Er wird mit dem TestCase erzeugt und vernichtet.

Die Bindung an den TestAspect erfolgt über den Pointcut initMe(), der die Methode initTestAspect() umschließt, die im zugehörigen TestCase aufgerufen wird. Der Nutzer kann initMe() auch anders definieren, was aber nicht der Bestimmung von initMe() entspricht, da der Aspekt dann nicht aktiv wird. initMe() dient nur dazu, den richtigen TestCase für den Aspekt auszuwählen. Die Instanziierung findet nur statt, wenn initTestAspect() aufgerufen wird.

Der TestAspect umschließt auch alle check()-Methoden des zugehörigen TestCases mit seinem Pointcut testCaseCheck() und verwendet die erweiterten Möglichkeiten der AspectJ-Reflexion, die er aus TestAspectEntity geerbt hat, um Fehlermeldungen zu erzeugen.

### TestCompleteInheritanceAspect

Dieser abstrakte Aspekt erbt wie TestAspect von TestAspectEntity. Seine Aufgabe ist aber nicht die Unterstützung der TestCase-Klassen sondern das Testen von Superklassenmethoden in einer ganzen Klassenhierarchie.

Darum kann dieser Aspekt auch nicht einem TestCase zugeordnet werden. Stattdessen wird eine Instanz von ihm mit jedem neuen Lauf einer kompletten TestSuite erzeugt. Das ermöglicht es, dieselben Klassen in verschiedenen TestSuites mehrmals zu testen. Die interne Logik der Klasse vermeidet, dass eine Klasse mehrmals in derselben Testsuite getestet wird.

Der Tester muss einen Aspekt von diesem Aspekt ableiten und den Pointcut newSubclass() definieren. Der Pointcut muss die Erzeugung einer neuen Subklasse der Superklasse erfassen. Wenn eine solche Konstruktion durchgeführt wird und gelingt, wird dieses Objekt geklont und der Klon als Testobjekt herangezogen. Die Tests werden in der Methode runTests() ausgeführt, die der Tester konkretisieren muss.

Der Tester darf nicht vergessen, dass zum Testen der Subklassen für jede Subklasse in einem Subklassen-Test mindestens einmal eine Konstruktion gelungen sein muss. Sonst wird der Test übergangen.

### TestLoggerAspect

TestLoggerAspect übernimmt den Client-Teil des Loggings. Das heißt, er sorgt dafür, dass die Methoden der TestLogger-Klasse zu den richtigen Zeitpunkten, z.B. wenn eine neue Testmethode betreten wird, aufgerufen werden.

### TestSupervisorAspect

Dieser Aspekt übernimmt allgemeine Aufgaben, die den gesamten Testlauf betreffen, dabei aber, im Gegensatz zu den Einstellungen in der Klasse TestSupervisor, sehr

feingranular sind.

Eine wichtige Aufgabe ist das Umgeben aller Testfälle mit einem `around()`-Advice, der sämtliche unerwartet auftretenden Ausnahmen fängt und daraus `TestUnexpectedExceptions` konstruiert, die an `TestStatus` gemeldet werden.

## 3.4. AOP-Abläufe im Framework

Der konventionelle Teil des Testframeworks ist leicht zu verstehen. Er lehnt sich an JUnit [51] an. Wenn aspektorientierte Komponenten hinzugefügt werden, sind die Zusammenhänge nicht mehr so leicht verständlich. Einem Entwickler, der noch keine Erfahrungen mit der aspektorientierten Programmierung gesammelt hat, können einige Zusammenhänge des Frameworks verwirrend und unübersichtlich vorkommen.

In diesem Abschnitt werden die wichtigsten Abläufe im Framework, in denen Aspekte eine Rolle spielen, textuell und grafisch mit UML-Sequenz-Diagrammen dargestellt. Die Diagramme befinden sich im Anhang.

In den Diagrammen werden Aspekte dargestellt, als wären sie Objekte, die Nachrichten empfangen und senden. Das entspricht nicht der Realität. Aspekte fügen Quellcode-Bausteine während einer Compiler-Vorstufe in durch Pointcuts definierten Stellen ein. Das Verhalten lässt sich aber auf diese Weise einfacher abstrahieren und klarer darstellen. Was wirklich geschieht, wenn ein Advice eines Aspektes im Diagramm eine Nachricht erhält, ist, dass der Code ausgeführt wird, der im Advice definiert wurde und an die Stelle mit der abgesetzten Nachricht hineingewoben wurde.

### 3.4.1. Logging, Abbildung A.7

Im Framework wird zweifach geloggt. Vor der Ausführung einer Methode, deren Name ein Muster enthält, das sie als Testmethode herausstellt, wird eine Log-Nachricht mit allen relevanten Informationen an die Klasse `TestLogger` geschickt, die dieses Ereignis in eine adäquate Form bringt und es auf ihr Logging-Ziel ausgibt, sofern es ihre Prioritätseinstellung erlaubt.

Zweitens wird nach dem Verlassen der Methode ein Task-Done-String erzeugt, der die durchgeführten Test-Aktionen beschreibt. Dieser String wird an `TestStatus` geschickt, wo er für die Statistik in einer Liste aufbewahrt wird.

### 3.4.2. Wirkung eines `TestAspects`, Abbildung A.8

Zur Unterstützung der Testfälle in den von `TestCase` abgeleiteten Klassen definiert der Tester Subklassen von `TestAspect`, die jeweils einem `TestCase` zugeordnet werden. Der jeweilige konkrete Testaspekt bezieht sich nur auf seine `TestCase`-Klasse und existiert auch nur in Zusammenhang mit ihr. Dadurch wird unter anderem verhindert, dass der Testaspekt Code in Testfälle einwebt, in denen er nicht gebraucht wird.

Die Zuordnung geschieht durch die Definition des abstrakten Pointcuts `initMe()`. Der Tester kann ihn als eine `AND`-Verknüpfung zwischen `initSuper()` und der Bedingung `within(konkreter TestCase)` definieren. `initSuper()` ist ein von `TestAspect` geerbter

abstrakter Pointcut, der den Aufruf der Methode `initTestAspect()` in Subklassen von `TestAspect` abfängt. Wichtig ist, dass die richtige `TestCase`-Subklasse im `initMe()`-Pointcut z.B. in einem within Statement enthalten ist. `initMe()` lässt sich zwar beliebig definieren, der Aspekt wird aber nur erzeugt, wenn `initSuper()` auch aktiviert wurde.

Der Pointcut sieht dann z.B. so aus:

```
public pointcut initMe() :
      initSuper() && within(TestVectorDouble4);
```

Eine Instanz eines Subaspektes von `TestAspect` wird erzeugt, sobald eine Instanz der ihm zugeordneten Testklasse ihn mit der Methode `initTestAspect()` anstößt.

In den Testfällen, in denen sie den `TestCase` unterstützt, stellt sie `before()` oder `after()`-Advices zur Verfügung, die sich auf die jeweiligen Testfälle beziehen. Mit dem `before()`-Advice setzt der Aspekt vor dem Aufruf der Testfälle einen Vorzustand, wenn das in der Klasse `TestCase` nicht möglich ist. Nachdem der Testfallaufruf abgeschlossen ist, prüft der Code im `after()`-Advice, ob der Folgezustand den Erwartungen entspricht.

Ist dies nicht der Fall wird ein Fehler erzeugt.

### 3.4.3. Abläufe im Hierarchietest, Abbildung A.9

Der Hierarchietest, realisiert im abstrakten Aspekt `TestCompleteInheritanceAspect`, baut darauf auf, dass in den `TestCase`s Konstruktoren aller Subklassen der zu testenden Superklasse getestet werden und einer davon ein gültiges Ergebnis zurückliefert.

Um eine Klassenhierarchie zu testen, muss der Tester einen Subaspekt von `TestCompleteInheritanceAspect` ableiten, in welchem er den abstrakten Pointcut `newSubclass()` und die abstrakte Methode `runTests()` redefiniert.

`newSubclass()` erfasst die Kreation einer Subklasse, die erfolgreich ablaufen sollte. Das könnte z.B. so aussehen:

```
public pointcut newSubclass():
  call (Vector+.new()) && within (TestVector*);
```

`Vector+.new()` erfasst alle Kreationen von Objekten, die Subklassen von `Vector` sind. `within(TestVector*)` sorgt dafür, dass nur solche Kreationen erfasst werden, die in Klassen geschehen, deren Name mit `TestVector*` beginnt.

`runTests()` enthält die eigentlichen Testfälle, die auf jede der in `newSubclass()` definierten Klassen angewandt werden soll.

Tritt der Fall ein, dass in einer Subklasse von `TestCase` ein solches Subklassenobjekt erzeugt wird, wird geprüft, ob die Subklasse schon getestet wurde. Wenn sie schon getestet wurde, tut `TestCompleteInheritance` nichts mehr. Wurde sie noch nicht getestet, lässt `TestCompleteInheritance` seine `runTests()`-Methode für einen Klon des Objekts durchlaufen. Danach fügt er den Namen der Klasse in die Liste der schon getesteten Klassen ein, um ein erneutes Testen zu verhindern.

Jede Instanz eines `TestCompleteInheritanceAspect`s ist dem Kontrollfluss der Methode `TestSuite` zugeordnet. Wenn eine `Testsuite` durchlaufen ist, wird die Instanz vernichtet. Wenn eine neue oder dieselbe `Testsuite` nocheinmal durchlaufen wird, wird

eine neue Instanz generiert. Lässt der Tester also mehrmals dieselbe `TestSuite` durchlaufen, wird für jeden Durchlauf der Hierarchietest durchlaufen.

### 3.4.4. Abläufe bei Mock Objects

Mock Objects sind mit den Mitteln von AspectJ so einfach einzusetzen, dass man keine Hilfe von einem Testframework benötigt. Der automatischen Fehlerstringgenerierung wegen lohnt es sich Mock Objects von `TestAspectEntity` abzuleiten.

Ein Mock Object lässt sich mit einem Aspekt umsetzen, der zu bestimmten Zeiten die Aufrufe an ein Objekt abfängt, das Verhalten des Objekts simuliert und dessen Nachrichten protokolliert. Am Ende muss geprüft werden, ob das zu testende Objekt sich korrekt verhalten hat.

Ein Aspekt, der als Mock Object wirken soll, muss folgendes enthalten:

- Einen Pointcut, der das Betreten einer Testmethode abfängt, in der ein Objekt getestet wird, das mit dem Objekt kommunizieren soll, das durch den Mock Aspekt simuliert wird. Nur innerhalb des Kontrollflusses dieser Methode wirkt der Aspekt.
- Statusvariablen, die den aktuellen Zustand der Kommunikation z.B. durch das Zählen der Schritte angeben.
- Pointcuts und Advices, die die Nachrichten an das simulierte Objekt abfangen. Sie müssen nicht nur das Verhalten des Objekts simulieren. Sie prüfen auch durch den Abgleich mit den Statusvariablen, ob die Nachricht zum richtigen Zeitpunkt gesendet wurde. Ist alles in Ordnung, ändern sie die Statusvariablen und simulieren das Verhalten des Objekts. Wenn nicht, geht das Mock Objekt in einen Fehlerzustand, den es nicht mehr verlassen kann, bis die Testmethode neu gestartet wird.
- Beim Verlassen der Testmethode muss eine `verify()`-Funktion prüfen, ob sich der Mock-Advice in einem gültigen Endzustand für den Testfall befindet. Wenn nicht, wird ein Fehler generiert.

### 3.4.5. Durchsetzen von Zusicherungen, Abbildung A.10

Im theoretischen Teil wurde gezeigt, dass sich die AOP dazu eignet Klasseninvarianten durchzusetzen.

Die Invariante muss nach Erzeugung der Klasse und beim Methodenein- und austritt gelten. Diese Punkte kann man allgemein mit zwei Pointcuts erfassen, die so aussehen könnten:

```
pointcut invariantConstructor() : call (NameDerKlasse.new());
pointcut invariantMethod() :  call (* NameDerKlasse.*(..));
```

Zum Konstruktor-Pointcut gibt es einen `after()`-Advice, der nach jeder Konstruktion eines Objekts der Klasse, eine Methode zur Invariantenprüfung aufruft, zum Methoden-Pointcut einen `around()`-Advice, der vor und nach der Ausführung einer Methode der Klasse die Invariantenmethode aufruft.

## 3.5. Zusammenhänge im Framework

In den Abbildungen A.4 und A.5 wird noch einmal zusammengefasst, wie welche Klassen und Aspekte aufeinander einwirken.

Die Abbildung A.4 zeigt die Elemente des Testframeworks zur besseren Übersicht ohne seine aspektorientierten Komponenten. Abbildung A.5 zeigt, wie die Aspekte im Framework sich auf die Klassen auswirken.

## 3.6. Tutorials

In den vorigen Abschnitten dieses Kapitels wurde die statische Struktur des Testframeworks beschrieben und auch auf die dynamischen Abläufe eingegangen.

In den folgenden Tutorials sollen Schritt-für-Schritt-Anleitungen gegeben werden, die die praktische Verwendung des Frameworks an konkreten Beispielen illustrieren.

### 3.6.1. Konventionelle Testsuites

1. **Eigene Testcases von `TestCase` ableiten** Jeder Klassentest im Framework wird von der Klasse `TestCase` abgeleitet:

   ```
   public class TestVectorDouble4 extends TestCase
   ```

   In der in `TestCase` abstrakten `run()`-Methode wird definiert, was beim Ablauf des `TestCase`s geschehen soll:

   ```
   public void run()  throws Exception
   {
       initTestAspect();
       testConstructor();
       testGetRandom();
       testGetNoOfElements();
       testGetLength();
       testIsNormalPositive();
       testIsNormalNegative();
       testNormalize();
       testMult();
   }
   ```

2. **Testsuites zusammenstellen** Im Hauptprogramm werden Instanzen vom Typ `TestSuite` erzeugt:

   ```
   _miscSuite = new TestSuite("Test Consts and Statics Suite");
   _matheSuite = new TestSuite("MatheSuite");
   _appSuite = new TestSuite("Test Application Suite");
   ```

Instanzen der erstellten `TestCase`s werden erzeugt...

```
TestVectorInt3 tc1 = new TestVectorInt3();
TestVectorDouble4 tc2 = new TestVectorDouble4();
TestMatrixDouble44 tc3 = new TestMatrixDouble44();
TestMatrixDouble33 tc4 = new TestMatrixDouble33();
```

... und in die `TestSuite`s eingehängt:

```
_matheSuite.addTestCase(tc1);
_matheSuite.addTestCase(tc2);
_matheSuite.addTestCase(tc3);
_matheSuite.addTestCase(tc4);
```

**3. Test ausführen** Eine Instanz vom Typ `TestSupervisor` erzeugen:

```
TestSupervisor testSuper = new TestSupervisor();
```

Die `TestSuite`s in das `TestSupervisor` Objekt einhängen:

```
testSuper.addTestSuite(_matheSuite);
testSuper.addTestSuite(_appSuite);
testSuper.addTestSuite(_miscSuite);
```

Testlauf starten:

```
testSuper.runTests();
```

**4. Ergebnisse auswerten** Die Ergebnisse befinden sich in der Singleton-Klasse `TestStatus`. Mit der `getStatistics()` Methode gelangt man an den Ergebnisstring:

```
System.out.println(TestStatus.getInstance().getStatistics());
```

Der Ergebnisstring präsentiert die durchlaufenen Testmethoden und einen zusammenfassenden Text. Lief die Suite fehlerfrei durch, sieht der Ergebnisstring ungefähr so aus:

```
Folgende Test-Methoden wurden abgearbeitet:
boolean testLinearAlgebra.TestVectorAspect.testGeneralRequirements()
void testLinearAlgebra.TestVectorInt3.testConstructor()
void testLinearAlgebra.TestVectorInt3.testGetElement()
(...)
void testMisc.TestConstsAndStatics.testRandomLong()

Keine Fehler oder Warnungen beim Durchlaufen der Testsuite.
```

Sind Fehler, Warnungen oder unerwartete Ausnahmen aufgetreten, liest man etwas wie:

```
Folgende Test-Methoden wurden abgearbeitet:
void testLinearAlgebra.TestVectorInt3.testConstructor()
(...)
void testMisc.TestConstsAndStatics.testRandomLong()


Folgende Fehler und Warnungen sind aufgetreten:
Testfall aufgerufen in TestProbe.java: Zeile 40
        erzeugte ein fehlerhaftes Ergebnis.
Testfall aufgerufen in TestMatrixDouble44.java: Zeile 22
        erzeugte eine unerwartete Ausnahme.
Testfall aufgerufen in TestMatrixDouble44.java: Zeile 108
        erzeugte eine unerwartete Ausnahme.
Testfall aufgerufen in TestProbe.java: Zeile 18
        erzeugte eine unerwartete Ausnahme.
Testfall aufgerufen in TestProbe.java: Zeile 32
        erzeugte eine unerwartete Ausnahme.

Insgesamt:
Anzahl Warnungen: 0
Anzahl unerwartet geworfener Ausnahmen: 4
Anzahl Fehler: 1
```

### 3.6.2. Testsuites mit Aspekten

**1. Eigene `TestCase`s erzeugen und `TestSuite` aufbauen wie im letzten Tutorial**

**2. Einen Testaspekt von `TestAspect` ableiten** Jeder gewöhnliche Unit-Testaspekt wird von `TestAspect` abgeleitet. Wenn er privilegiert sein soll, darf das Schlüsselwort `privileged` nicht fehlen:

```
public privileged aspect TestVectorDouble4Aspect extends TestAspect
```

**3. `TestAspect` einem `TestCase` zuordnen** Jeder Aspekt, der von der Klasse `TestAspect` abgeleitet wurde, ist einem bestimmten `TestCase` assoziiert, mit dem er gleichzeitig erzeugt und vernichtet wird. Alle Pointcuts in diesem Aspekt beziehen sich nur auf den `TestCase`, mit dem die Assoziation besteht. Der Pointcut `initMe()` bestimmt im `within()`-Teil, mit welchem `TestCase` der Aspekt assoziiert werden soll. Er sollte immer in Verbindung mit `initSuper()` verwendet werden. `initSuper()` ermöglicht ein Einschalten des Testaspektes durch den Aufruf `initTestAspect()`.

```
public pointcut initMe() :
 initSuper() && within(TestVectorDouble4);
```

4. **Aspekt aktivieren** An geeigneter Stelle in der assoziierten Klasse dem Aspekt signalisieren, dass er aktiv werden soll. Z.B. zu Beginn der `run()`-Methode des `TestCases`:

```
public void run()  throws Exception
{
  initTestAspect();
  (...)
}
```

5. **Pointcuts definieren** Die Pointcuts bestimmen die Punkte, an denen der Aspekt aktiv werden soll. Es gibt viele Möglichkeiten sie zu definieren. Im Beispiel wird ein Pointcut definiert, der alle Konstruktoraufrufe mit vier `double`-Parametern abfängt. Der `call`-Teil der Pointcut-Definition beschreibt die Gestalt des Konstruktoraufrufs mit vier Parametern, der `args`-Teil gibt an, dass die vier out-Parameter des Pointcuts den vier Argumenten des Konstruktor-Aufrufs entsprechen.

```
pointcut vector4ConstructorTest(double nX, double nY,
                                double nZ, double nW) :
call (VectorDouble4.new(double, double, double, double))
&& args(nX, nY, nZ, nW);
```

6. **Testenden Advice definieren** Für jeden Test-Pointcut muss nun ein Advice definiert werden, der angibt wie zu testen ist. Im Beispiel wird für jeden 4-parametrigen Konstruktor überprüft, ob die Member des erzeugten Vektor-Objekts die Werte haben, die dem Konstruktor als Parameter übergeben wurden.

```
after(double fX, double fY, double fZ, double fW)
                returning(VectorDouble4 newConst) :
  vector4ConstructorTest(fX, fY, fZ, fW)
{
  check(((newConst.m_fX == fX) && (newConst.m_fY == fY) &&
  (newConst.m_fZ == fZ) && (newConst.m_fW== fW)));
}
```

Bei dem Advice handelt es sich um einen `after()`-Advice zu `vector4Constructor-Test`. Das bedeutet, er wird nach der Konstruktion eines `VectorDouble4`-Objekts eingewebt. Das neu konstruierte Objekt wird im `returning(...)`-Teil mit `newConst` benannt und dessen einzelne Werte mit den Eingabeparametern des Konstruktors verglichen.

### 3.6.3. Verallgemeinerte Tests

**1. `TestAspect` definieren und zuordnen wie im letzten Tutorial**

**2. Testfälle mit verschiedenen Ergebnissen in verschiedenen Methoden definieren** Im Beispiel wird die Methode `isNormal()` der Klasse `VectorDouble4` getestet. Die möglichen Ergebnisse sind `true` und `false`. In `testIsNormalPositive()` werden alle Testfälle absolviert, die ein positives Ergebnis liefern müssen, wenn die Methode korrekt ist. Die negativen Testfälle werden in `testIsNormalNegative()` getestet.

```
public void testIsNormalPositive()
{
  VectorDouble4 vec2 = new VectorDouble4(0, 1, 0, 0);
  vec2.isNormal();
  (...)
}


public void testIsNormalNegative()
{
  VectorDouble4 vec = new VectorDouble4(3, 2, 1, 5);
  vec.isNormal();
  (...)
}
```

**3. Pointcuts für die verschiedenen Fälle definieren** Der erste Pointcut fängt alle positiven Fälle, weil seine Wirkung mit `cflowbelow(call(* TestVectorDouble4.testIsNormalPositive()))` auf die `isNormal()`-calls im Kontrollfluss der Methode `testIsNormalPositive()` beschränkt ist. Analog werden die negativen Testfälle alle von `cflowbelow(call(* TestVectorDouble4.testIsNormalNegative()))` gefangen.

```
pointcut vectorDouble4PositiveIsNormalTest() :
  call (* VectorDouble4.isNormal())
  && cflowbelow(call(* TestVectorDouble4.testIsNormalPositive()));

pointcut vectorDouble4NegativeIsNormalTest() :
  call (* VectorDouble4.isNormal())
  && cflowbelow(call(* TestVectorDouble4.testIsNormalNegative()));
```

**4. Ergebnisse mit Advices prüfen** Hier findet die eigentliche Überprüfung statt:

```
after() returning(boolean retValue) :
  vectorDouble4PositiveIsNormalTest()
{
  check(!retValue);
}

after() returning(boolean retValue):
  vectorDouble4NegativeIsNormalTest()
{
  check(retValue);
}
```

### 3.6.4. Hierarchietestende Aspekte

**1. Neuen Aspekt von `TestCompleteInheritanceAspect` ableiten**
`public aspect TestVectorAspect extends TestCompleteInheritanceAspect`

**2. Superklasse festlegen** Ein hierarchietestender Aspekt ist dem Kontrollfluss eines `TestSuite`-Laufs zugeordnet, damit der Test auf alle Subklassen der definierten Subklasse genau einmal für jeden Suite-Durchlauf ausgeführt wird. Der Pointcut `newSubclass()` sollte so definiert werden, dass er auf die Konstruktion aller Subklassenobjekte wirkt:

```
public pointcut newSubclass():
    call (Vector+.new()) && within (TestVector*);
```

Im Beispiel werden alle Konstruktoraufrufe von `Vector` erfasst, die innerhalb einer Klasse auftreten, deren Name mit dem String `TestVector` beginnt. Die interne Logik von `TestCompleteInheritanceAspect` sorgt dafür, dass jede Subklasse pro Suite-Durchlauf nur einmal getestet wird und nicht bei jeder neuen Konstruktion.

**3. `runTests()` definieren** Die abstrakte Methode aus `TestCompleteInheritanceAspect` gibt die Tests an, die für jedes Subklassenobjekt durchgeführt werden sollen:

```
public void runTests(Object obj)
{
   runAll((Vector)obj);
}
```

### 3.6.5. Testsuites mit Mock Objects

**1. Aspekt von `TestAspectEntity` ableiten** Das dient dem Zweck einige praktische Methoden zu erben:

`public aspect MockRadio extends TestAspectEntity`

2. **Testabläufe definieren, die getestet werden sollen** Der Mock-Aspekt simuliert in der Regel ein Server-Objekt und testet, ob der Client die richtigen Anfragen in der richtigen Reihenfolge mit den richtigen Parametern stellt. Jeder dieser Abläufe muss explizit definiert werden, damit der Mock-Aspekt weiß, was er testet:

   ```
   static final int MOCK_GET_IMAGE_DATA = 0;
   ```

   Welcher Testlauf gerade aktiv ist, wird in einer Member-Variable festgehalten:

   ```
   int _nToBeMocked;
   ```

   Der Zustand des Testlaufs in einer anderen:

   ```
   int _nStepCount = 0;
   ```

3. **Ablaufeintrittspointcuts und Advices definieren** Das Signal zum Beginn des Testlaufs wird durch einen Pointcut, üblicherweise der Aufruf einer Testmethode, gegeben:

   ```
   pointcut testGetImageData():
   call(* TestProbe.testGetImageData());
   ```

   Der zugehörige Advice initialisiert den Testlauf:

   ```
   before():
     testGetImageData()
     {
       _nToBeMocked = MOCK_GET_IMAGE_DATA;
       _nStepCount = 0;
     }
   ```

4. **Pointcuts und Advices für die Client Aufrufe definieren** Alle relevanten zu testenden Aufrufe an das zu simulierende Objekt müssen mit Pointcuts abgefangen und mit `around()`-Advices umgeben werden. Die Advices prüfen, ob der Client die richtige Anfrage zur richtigen Zeit gestellt hat und ändern den Testablaufzustand, z.B. durch Inkrementierung des Ablaufzählers:

   ```
   pointcut submitMessage(int msg) :
   cflowbelow(call(* TestProbe.testGetImageData()))
   && call(* Radio.submitMessage(int))
   && args(msg);

   void around(int msg) :
     submitMessage(msg)
   ```

```
{
  switch(_nToBeMocked)
  {
    case MOCK_GET_IMAGE_DATA:
      switch(msg)
      {
        case Probe.CONTACT_PROBE:
          if(_nStepCount == 1)
          {
             _nStepCount++;
             return;
          }
          break;
        case Probe.ASK_IMAGE_DATA:
          if(_nStepCount == 3)
          {
            _nStepCount++;
            return;
          }
          break;
      }
      break;
  }
  _nStepCount = Integer.MAX_VALUE;
}
```

**5. Verifizierung** Der Eintrittspointcut dient gleichzeitig als Austrittspointcut. Am Ende des Tests muss verifiziert werden, ob alles korrekt abgelaufen ist:

```
after():
testGetImageData()
{
  check(verify(), thisJoinPoint);
}

public boolean verify()
{
  switch(_nToBeMocked)
  {
     case MOCK_GET_IMAGE_DATA:
       if (_nStepCount == 6) return true;
       break;
  }
  return false;
```

```
}
```

### 3.6.6. Klasseninvarianten durchsetzen

1. **Pointcut für Invariante definieren** Wo der Pointcut und seine Advices für die Invariante definiert werden, hängt davon ab, ob die Invariante nur in der Testphase oder immer (auch während der Anwendung durch den Nutzer) geprüft werden soll.

   Wird die Invariante nur in der Testphase benötigt, wird sie am besten im `TestAspect` definiert, der dem Klassen-`TestCase` zugeordnet ist. Soll sie allgemein immer geprüft werden, legt man einen neuen Invarianten-Aspekt außerhalb des Testframeworks an.

   In beiden Fällen muss die Invariante vor und nach jeder Methodenausführung der Klasse gelten. Der Pointcut muss daher alle Methodeneintritte, also alle calls zu Methoden,erfassen.

   ```
   pointcut invariant(MovingObject movObj) :
     !call (* MovingObject.getInstance(MatrixDouble44))
     && call(* MovingObject.*(..))
     && target(movObj);
   ```

   Im Beispiel werden alle Methodenaufrufe der Klasse `MovingObject` mit Ausnahme der statischen Methode `getInstance()` erfasst. `getInstance()` darf von diesem Pointcut nicht erfasst werden, da die statische Methode nicht im Zusammenhang mit einer Objekt-Instanz aufgerufen wird.

2. **Prüfende Advices definieren** Die Advices enthalten die Prüfung der Bedingung der Zusicherung. Im Beispiel muss die Positionsmatrix eines `MovingObject`s immer eine Transformationsmatrix sein. Ist die Bedingung nicht erfüllt, wird als Zusicherungs-Aktion ein Fehler generiert.

   ```
   before(MovingObject movObj) :
     invariant(movObj)
   {
     check(movObj.m_matWhere.isTransformationMatrix(),
           thisJoinPoint, "Klassen-Invariantenverletzung:
            MovingObject-Positionsmatrix ist keine Transformationsmatrix.");
   }

   after(MovingObject movObj) :
     invariant(movObj)
   {
     check(movObj.m_matWhere.isTransformationMatrix(),
           thisJoinPoint, "Klassen-Invariantenverletzung:
            MovingObject-Positionsmatrix ist keine Transformationsmatrix.");
   }
   ```

Bei den beiden Advices handelt es sich um einen `before()`-Advice, der vor allen Methodenaufrufen von `MovingObject` eingewebt wird, und um einen `after()`-Advice, der nach jedem Aufruf aktiv wird. Beide Advices prüfen die Invariante.

## 3.7. Zusammenfassung

Die praktische Implementierung des implementierten Frameworks ist der schriftlichen Arbeit teilweise in Codeausschnitten im Anhang beigefügt. Dieses Kapitel hatte die Aufgabe die praktische Implementierung in Struktur und in der Art der Verwendung zu beschreiben.

Um dieses Ziel zu erreichen wurde dem Leser die statische Struktur und die Dynamik der Elemente des Frameworks erklärt. Die Tutorials am Ende des Kapitels bieten ihm Schritt-für-Schritt-Anleitungen zum Kennenlernen des Frameworks.

Die folgende Zusammenfassung verbindet noch einmal das Thema der Arbeit, die Implementierung und eine wertende Aussage aus der Praxis.

# 4. Zusammenfassung

## 4.1. Der aspektorientierte Ansatz im Unit-Test

Das Thema *Entwicklung eines aspektorientierten Frameworks für den objektorientierten Unit-Test* wurde in dieser Arbeit auf theoretische und praktische Weise bearbeitet.

Es ging in der Arbeit nicht nur darum, die Möglichkeit des Einsatzes aspektorientierter Konzepte im Unit-Test aufzuzeigen, sondern auch darum zu demonstrieren, dass der Einsatz der AOP in der Praxis konkrete Vorteile mit sich bringt. Dazu musste eine Softwareeinheit entwickelt werden, in der die neuen Konzepte einen sichtbaren Vorteil bewirken.

Das Ergebnis dieser Arbeit ist ein Unit-Testframework, das in seiner Struktur an Testframeworks wie JUnit [51] angelehnt ist, aber Zusatzfunktionalitäten anbietet, an die objektorientierte Testframeworks schon aufgrund ihrer Beschränkung auf objektorientierte Hilfsmittel nicht heran reichen.

Die auffälligsten Vorteile werden im folgenden Abschnitt zusammengefasst. Der übernächste Abschnitt beschreibt die Risiken, die bei einem Einsatz des Frameworks zum Testen auftreten. Obwohl es keine „Baustellen“ im Sinne fehlender Funktionalitäten im Framework gibt, ist es offen für Erweiterungen und neue Ideen. Einige davon werden im Ausblick als Abschluss der Zusammenfassung vorgestellt.

## 4.2. Vorteile des implementierten Frameworks gegenüber konventionellen Frameworks

### 4.2.1. Komfort

AspectJ verfügt über mächtigere Reflexionsfähigkeiten als die Java-Reflection-Classes. Im implementierten Testframework werden diese Fähigkeiten dazu ausgenutzt, Fehlerbeschreibungen automatisch zu generieren. Das ermöglicht unter dem äußerst geringen Aufwand eines Pointcuts und seines zugehörigen Advices für alle Testfälle Fehlerbeschreibungen zu generieren, die den Namen der testenden Klasse, die testende Funktion und sogar die Zeilennummer des Aufrufs enthalten. Diese Funktionalität wird automatisch angewandt. Der Tester muss nur seine Testcases schreiben, um die Strings zur Fehlerbeschreibung muss er sich nicht kümmern.

Unerwartet auftretende Ausnahmen in einem Testfall werden nicht auf höhere Ebenen weiter propagiert. Sie werden behandelt wie einfache Fehler. Falls im Test eine unerwartete Ausnahme auftritt, wird der Testlauf nicht unterbrochen. Stattdessen wird eine Instanz der Klasse `TestUnexpectedException` generiert, die den Ort des Auftretens enthält, und der Testlauf fortgesetzt.

Flexibles Logging zu beliebigen Zielen ist für ein Testframework eine Selbstverständlichkeit. Das aspektorientierte Framework dieser Diplomarbeit nutzt zum Loggen die Möglichkeiten der AOP, weshalb sich auch der Client-Teil des Logging-Vorgangs in einem abgekapselten Aspekt mit einer allgemeinen Regel für das Loggen befindet. Im konventionellen Logging verstreut sich der Logging-Code über die gesamte Implementierung.

### 4.2.2. Zeit- und Platz-Ersparnis

Im aspektorientierten Framework ist es leicht, über AOP-Quantifizierungen verschiedene Formen von zusammenfassenden Tests zu definieren. Das reicht von der Überprüfung von Testfallgruppen mit ähnlichen Ergebnissen in nur einer Zeile bis zum Testen auf Liskov-Konformität von Subklassen.

Das erspart dem Tester Schreibarbeit und dient der Übersicht. Der Liskov-Konformitätstest kann sogar Kunden von Unternehmen, die erweiterbare Bibliotheken anbieten, Testarbeit abnehmen.

### 4.2.3. Überwindung klassischer Testprobleme

Klassische Testframeworks bieten dem Tester keine Hilfsmittel zur Überwindung von Problemen, die beim Test von objektorientierten Programmen auftreten. JUnit [51] und Boost [37] z.B. helfen dem Tester nur dabei seine Tests zu strukturieren und darzustellen. Das Durchbrechen der Kapselung nur in der Testsituation, Instrumentierungen, usw. sind Probleme, die der Tester in klassischen Frameworks selbst immer wieder lösen muss.

Das zu dieser Diplomarbeit entstandene Framework gibt dem Tester im aspektorientierten Teil kleine, aber mächtige Werkzeuge in die Hand, die wie z.B. im Fall der Instrumentierung oder des automatischen Subklassen-Tests Aufgaben übernehmen, für die sonst kommerzielle Tools oder schwerfällige Eigenbau-Lösungen herangezogen werden müssten.

### 4.2.4. Unkomplizierte Implementierung von Mock Objects

Zur praktischen Implementierung des Mock Object-Konzepts stehen Tools wie [59] zur Verfügung. Überraschenderweise hat sich als Nebenergebnis bei der Implementierung des Testframeworks gezeigt, dass Aspekte selbst ideale Mock Objects sind[1].

Das Mock Object-Konzept wurde zwar nicht als fester Bestandteil in das Framework integriert. Wie im Tutorial 3.6.5 beschrieben, ist es aber günstig, angehende Mock Objects von einer Klasse des Frameworks abzuleiten. Der Rest der Programmierung des Mock Objects besteht nur noch darin, die zu simulierenden Methoden mit Pointcuts auszuwählen, mit Advices zu simulieren und einen Verifikationsmechanismus zu schreiben.

[1] Warum sie das sind, steht in Abschnitt 2.3.

### 4.2.5. Modularität

Testern, denen keine aspektorientierten Hilfsmittel zur Verfügung stehen, müssen den zu testenden Code vor dem Testen vorbereiten, um ihm überhaupt erst testbar zu machen. Z.B. müssen gekapselte Member entkapselt werden, Instrumentierungsanweisungen eingefügt werden, das Verhalten von kommunizierenden Objekten so verändert werden, dass der zeitliche Testaufwand nicht zu hoch ist usw.

Den Code für jede zu testende Klasse manuell zu verändern und wieder rückzusetzen ist aufwändig und sehr gefährlich. Code-Generatoren und Präprozessoren bieten etwas mehr Sicherheit. Präprozessor-Makros durchsetzen aber den Code mit ihrer Makro-Sprache, die sich u.U. stark von der Implementierungssprache unterscheidet. Code-Generatoren sind der Implementierungssprache noch ferner und müssen separat konfiguriert werden.

Die aspektorientierte Programmierung ist für dieses Problem die geeignetere Lösung. Die Implementierungssprache ist Teil der aspektorientierten Sprache und die Quantifizierungen erlauben an verschiedenen Stellen ein Sonderverhalten zu spezifizieren, das nur für wenige Tests benötigt wird. Es ist nicht mehr nötig, den zu testenden Quellcode selbst mit Makros oder Code-Generatoren vorzubereiten. Der Cross-Cutting Concern „Versetze eine Menge von Methoden in einen Zustand, der für die folgenden Tests benötigt wird“ wurde aus den einzelnen Stellen im Code herausgezogen und in einen zur Testklasse zugeordneten Testaspekt abstrahiert.

## 4.3. Kritik des Ansatzes

Einige Probleme bei der Anwendung der AOP allgemein und speziell beim Unit-Test dürfen nicht verschwiegen werden.

In den einzelnen Abschnitten des theoretischen Kapitels wurde schon auf individuelle Probleme bei bestimmten Ansätzen eingegangen. Die Hauptkritikpunkte lassen sich aber in wenigen Punkten zusammenfassen.

### 4.3.1. Höherer Aufwand beim Kompilieren

Das Weben des Aspekt-Codes vor dem Kompilieren ist ein Schritt, der zusätzlich Zeit kostet. Das Kompilieren dauert spürbar länger als mit konventionellen Methoden.

### 4.3.2. Einarbeitung

Der aspektorientierte Ansatz unterscheidet sich stark vom imperativen oder objektorientierten Stil. Auch wenn man alle Konzepte begriffen hat, muss man noch lernen, wo und wie der Einsatz der AOP sinnvoll ist, damit man mit ihr über ein mächtiges Werkzeug verfügt. Ihre große Mächtigkeit macht ihren Einsatz gefährlich in den Dimensionen:

- Korrektheit
- Übersichtlichkeit, Nachvollziehbarkeit
- Performance (besonders beim Kompilieren)

Bevor man die AOP also in einem wichtigen Projekt als Unterstützung fürs Testen anwendet, sollte man in ihrem Einsatz ausreichend geübt sein.

### 4.3.3. Unsicherheit

Wie oben angesprochen, birgt der Einsatz der AOP auch Gefahren. Besonders beim Testen, einem Qualitätssicherungsprozess, ist es jedoch sehr wichtig, dass alles korrekt abläuft. Das ist wohl der Hauptkritikpunkt an der Idee des Einsatzes der AOP beim Unit-Test, weil er eine kritische Anforderung des Prozesses Testen selbst betrifft.

Darum gilt, wie oben schon geschrieben, dass die aspektorientierte Programmierung nur mit ausreichender Kenntnis angewandt werden sollte.

## 4.4. Ausblick

Im Folgenden werden Ideen vorgestellt, mit denen die praktische Implementierung noch erweitert werden kann, die aber aufgrund ihrem Abstand zum eigentlichen Thema oder ihrem unangemessen hohen Implementierungs-Aufwand keinen Eingang mehr in diese Diplomarbeit fanden.

### 4.4.1. Makro-Sprache

Wie im Kritik-Teil schon angesprochen, kann die aspektorientierte Methode zu etwas unübersichtlichen Zusammenhängen führen. Das gilt besonders, wenn die Aspekte sich auf viele kleine Fragmente im testenden Code beziehen. Dabei entsteht eine größere Menge von Aspekten, die eine wachsende Fehlergefahr bedeuten.

Wenn die AOP-Erweiterungen nur für kleinere Aufgaben angewandt werden, möchte man sich als Tester auch nicht unbedingt tief in AOP-Sprachen einarbeiten müssen.

Man könnte dem Tester als Erleichterung eine Makro-Sprache anbieten, die für den aspektorientierten Teil des Testens verantwortlich ist. Der Tester müsste nur den konventionellen Teil schreiben und kann den aspektorientierten Teil mit einigen kurzen Makros im konventionellen Testcode erledigen.

Im folgenden Beispiel wird die Methode `VectorDouble4.isNormal()` unter Zuhilfenahme der AOP getestet, um mehrere Testergebnisse zusammenzufassen:

```
public void testIsNormalNegative()
{
  VectorDouble4 vec = new VectorDouble4(3, 2, 1, 5);
  vec.isNormal();
  VectorDouble4 vec2 = new VectorDouble4(0, 1, 6, 0);
  vec2.isNormal();
  VectorDouble4 vec3= new VectorDouble4(.3, 0, 0, 0);
  vec3.isNormal();
}
```

Der nötige zugehörige Aspekt-Code in `TestVectorDouble4Aspect` lautet:

```
pointcut vectorDouble4NegativeIsNormalTest() :
  call (* VectorDouble4.isNormal())
  && cflowbelow(call(* TestVectorDouble4.testIsNormalNegative()));

after() returning(boolean retValue):
  vectorDouble4NegativeIsNormalTest()
{
  if (retValue) makeError(thisJoinPoint);
}
```

Weniger kompliziert könnte das mit einem Makro aussehen:

```
public void testIsNormalNegative()
{
  VectorDouble4 vec = new VectorDouble4(3, 2, 1, 5);
  vec.isNormal();
  VectorDouble4 vec2 = new VectorDouble4(0, 1, 6, 0);
  vec2.isNormal();
  VectorDouble4 vec3= new VectorDouble4(.3, 0, 0, 0);
  vec3.isNormal();
  CHECK_ASPECT_ALL_RETVALUES(TestVectorDouble4.testIsNormal(),
                             isNormal(), false)
}
```

Der Makro-Präprozessor müsste dann, wenn es noch keinen zugehörigen Aspekt zur Testklasse gibt, einen solchen generieren, und Pointcut und Advice mit den gelieferten Informationen in ihn schreiben.

Leider wird bei Java - anders als z.B. für C++ - kein standardisierter Präprozessor mitgeliefert, der diese Parse-Arbeit übernehmen könnte. Es gibt zwar viele kommerzielle und auch einige freie Präprozessoren (z.B. [56] und [49]) für Java. Einen davon vorauszusetzen, wäre aber übertrieben. Der Code könnte nicht mehr von jedem ohne weiteres übersetzt werden.

Außerdem ist es nicht trivial, mit einer Makro-Sprache all die nötigen Aspekt-Konstrukte aus den Parse-Informationen zusammenzustellen.

Darum wurde von der Verwendung einer Makro-Sprache abgesehen.

### 4.4.2. Höhere Sicherheit

Für keinen der AOP-Mechanismen kann garantiert werden, dass er auf die richtige Weise eingesetzt wird. Dem könnte mit überwachenden Aspekten abgeholfen werden. Sie könnten sicherstellen, dass Testaspekte z.B. nur zu `TestCase`s beim Aufruf der Methode `initTestAspect()` generiert und zugeordnet werden. Auch für die anderen Mechanismen könnte man sich überwachende Aspekte vorstellen.

Würde es sich bei dem Framework um ein kommerzielles Produkt handeln, sollte die Mühe zusätzliche Sicherungen einzubauen nicht gescheut werden. Da an ein Framework,

das zu Demonstrationszwecken für eine Diplomarbeit geschrieben wurde, aber andere Ansprüche gestellt werden als an ein kommerzielles Produkt, wurde auf die Implementierung zusätzlicher Sicherheitsmechanismen verzichtet, die übrigens in den meisten frei verfügbaren Testframeworks auch nicht enthalten sind.

Es geht darum, auch dem in der aspektorientierten Methode weniger bewandten Leser eine praktische Anwendung der aspektorientierten Möglichkeiten zu demonstrieren. Das Hinzufügen von Sicherheitsmechanismen ist an und für sich keine sehr anspruchsvolle Aufgabe. Das Problem ist, dass diese Sicherheitsmechanismen sich auf feingranularer Ebene auf das feingranulare Auswirken von Aspekten auf konventionellen Code beziehen würden, was den ungeübten Leser nur verwirrt und von der eigentlichen Aufgabe des Frameworks ablenkt.

Es ging nicht darum ein Produkt für die Anwendung zu entwickeln, sondern darum ein kompaktes Beispiel zur Demonstration vorzustellen. Das heißt nicht, dass das Framework nicht für die Praxis geeignet wäre. Es lässt sich bei grob fahrlässiger Benutzung nur auch auf die falsche Weise einsetzen.

### 4.4.3. Mock Objects als festen Bestandteil ins Framework integrieren

Im theoretischen und im praktischen Teil wurde der Einsatz von Mock Objects demonstriert. Das Framework enthält jedoch keine Klassen oder Aspekte, die eigens zur Unterstützung von Mock Objects geschrieben wurden.

Das ist darin begründet, dass so ein Aspekt dem Tester keine Arbeit abnehmen würde. Mock Objects enthalten Pointcuts zum Abfangen von Methodenaufrufen, deren Advices, Variablen, die die Schritte zählen, und eine Verifikationsmethode.

Außer der bloßen Existenz der Verifikationsmethode sind Mock Objects allgemein verschieden. Einen Superaspekt zu definieren, der nur die abstrakte Deklaration der Verifikationsmethode enthält, sonst keine weiteren Mechanismen implementiert und mit anderen Klassen in keiner Beziehungen steht, vererbt seinen Klassen nur die zusätzliche Schreibarbeit für den Text `extends MockObject` und ist darum überflüssig.

Enthielte das Framework eine Makro-Sprache, könnte man sie für Mock Objects auf sinnvolle Weise einsetzen. Ein Makro `MOCK_OBJECT(Radio, MockRadio)` würde z.B. dafür sorgen, dass MockRadio als Mock Object für Radio verwendet wird. Da das Framework aber keine Makro-Sprache enthält, sind Mock Objects kein fester Bestandteil des Frameworks.

### 4.4.4. Mutationstests

Nach mehreren gescheiterten Experimenten wurde der Versuch aufgegeben, Mutationstests allein mit aspektorientierten Methoden elegant als Test-Instrument einzusetzen. Es ist zwar leicht möglich, Aspekte als Mutationsaspekte ähnlich wie Mock Objects einzusetzen. Der Aufwand ist aber zu groß, jeder Mutation einen eigenen handgeschriebenen Aspekt zuzuordnen.

Code-Generierungs-Tools für den Mutationstest wie Jester [50] generieren in kurzer Zeit automatisch eine größere Menge von Mutationen. Sie eignen sich darum viel besser

für die Aufgabe Tests durch Mutationstests zu testen als handgeschriebene Aspekte eines Frameworks.

Wie in [17] ausführlich beschrieben, ist es außerdem ein großes Problem, die richtigen Mutationen effizient zu generieren. Da Code-Generierung in einem Testframework sowieso keinen Platz hat, wurde auf den Mutationstest in der praktischen Implementierung verzichtet und die Idee nur in der Theorie kurz angedacht.

### 4.4.5. Testen von aspektorientiertem Code

Das Testen von aspektorientiertem Code birgt ganz eigene Schwierigkeiten, wie im theoretischen Teil der Arbeit in Abschnitt 2.5 schon angerissen wurde. Allein mit den theoretischen Überlegungen zum Unit-Testen von aspektorientiertem Code ließen sich Bücher füllen[2]. Das Thema ist theoretisch noch nicht ausgereift. Darum beschränken sich die Fähigkeiten des Frameworks auf den Test von objektorientiertem Code.

Eine Erweiterung der Fähigkeiten des Frameworks, so dass es auch aspektorientierten Code testen kann, wäre u.U. ein interessantes Thema für eine weitere Diplomarbeit.

[2]Siehe dazu auch [33].

# A. Diagramme

## A.1. Klassendiagramme

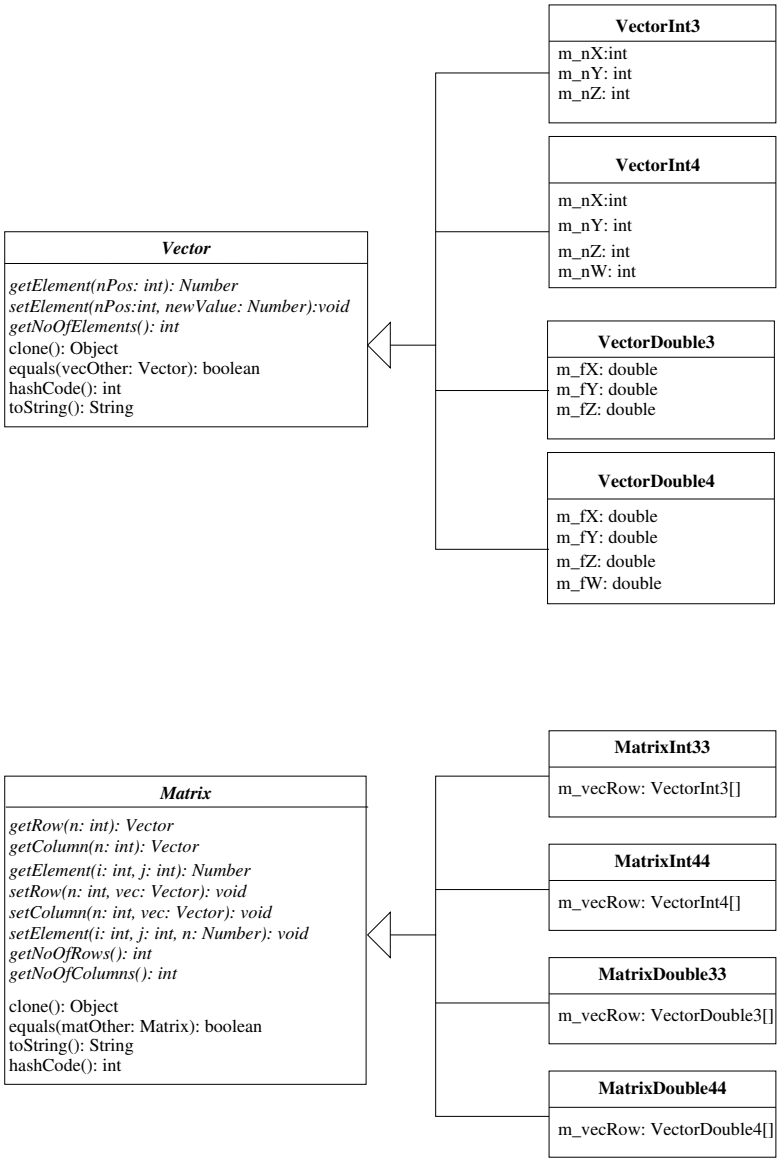


Abbildung A.1.: linearAlgebra Package

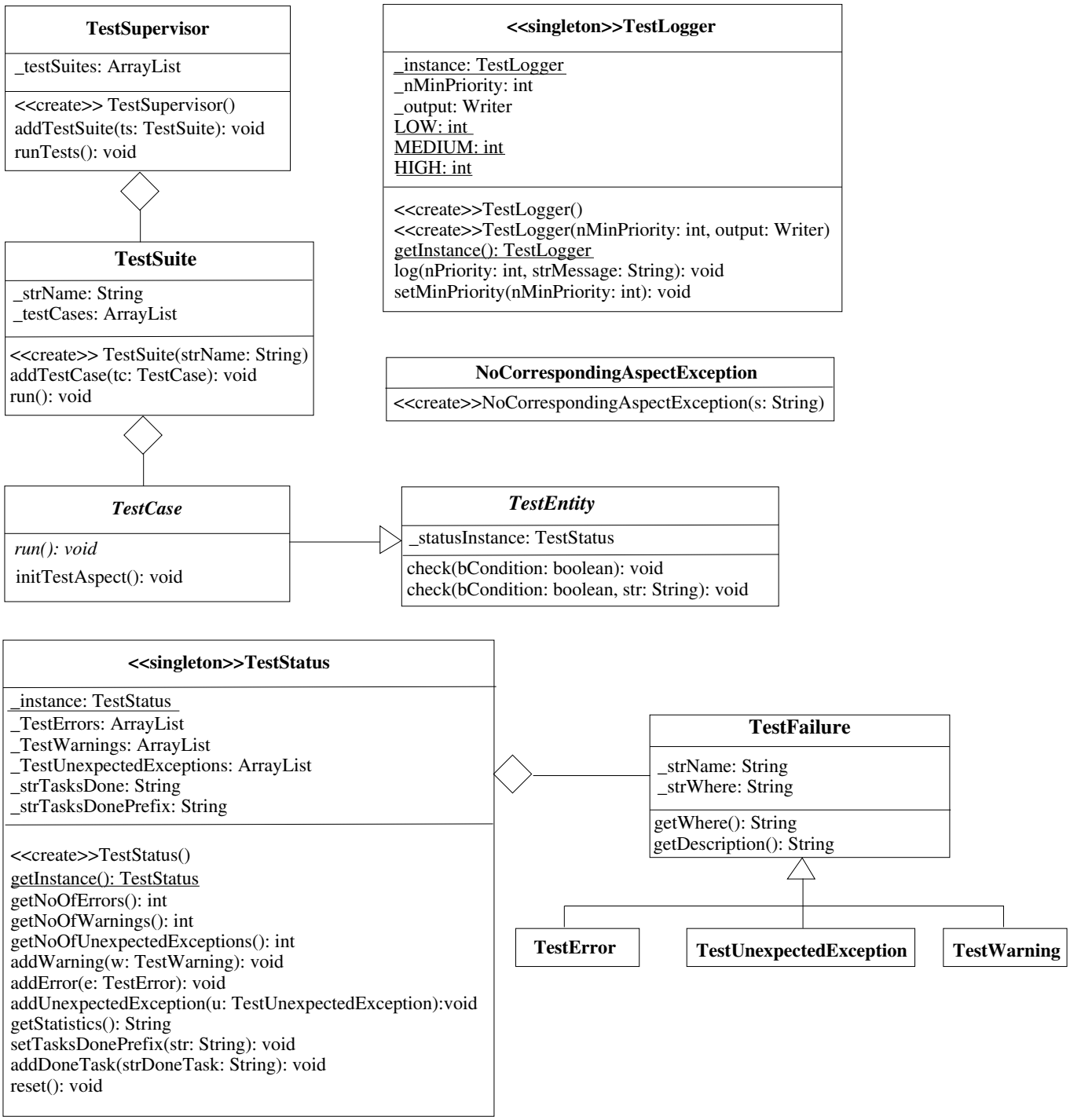


Abbildung A.2.: Klassenhierarchie und Aggregationen im Testframeworks

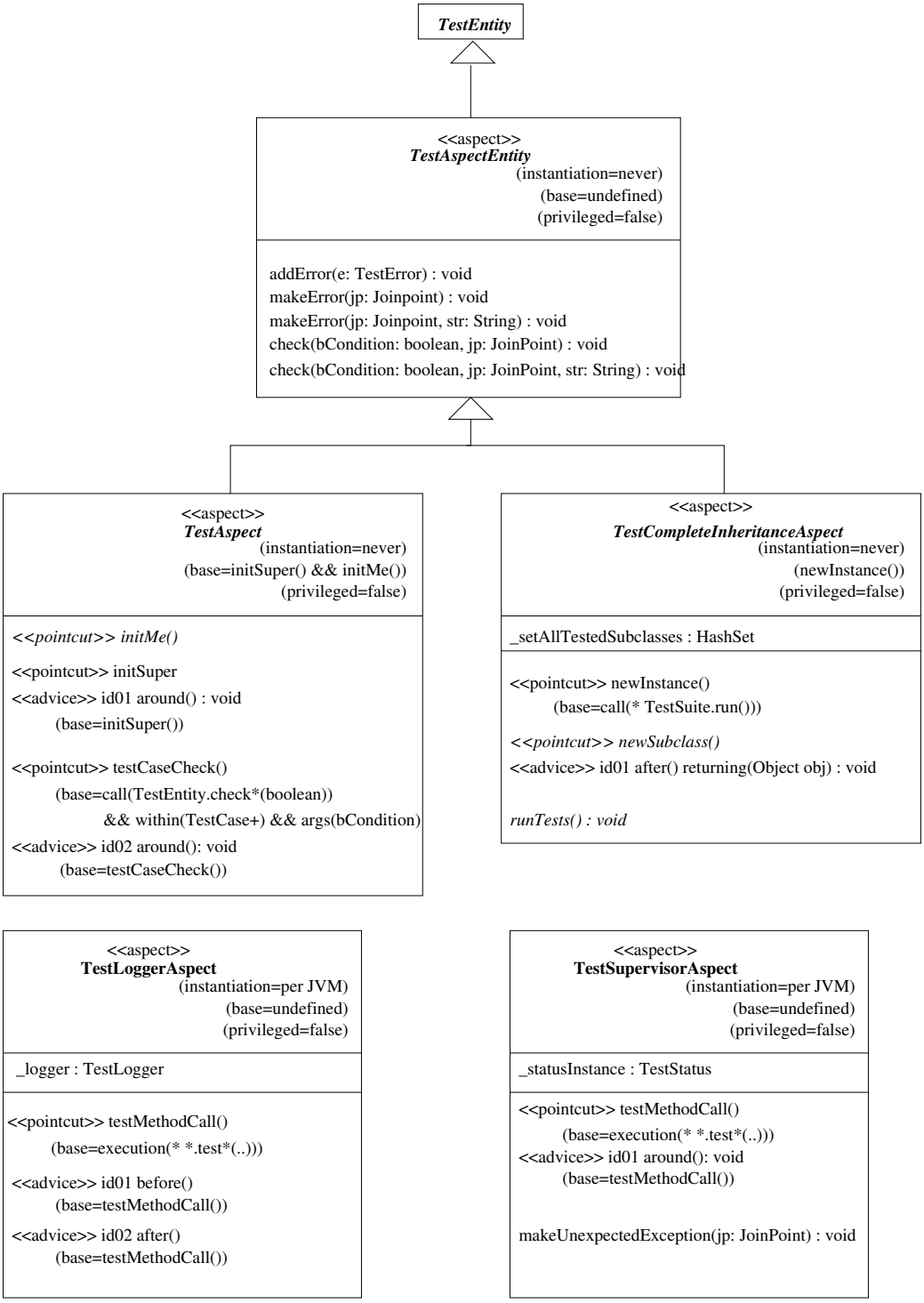


Abbildung A.3.: Aspekthierarchie im Testframework

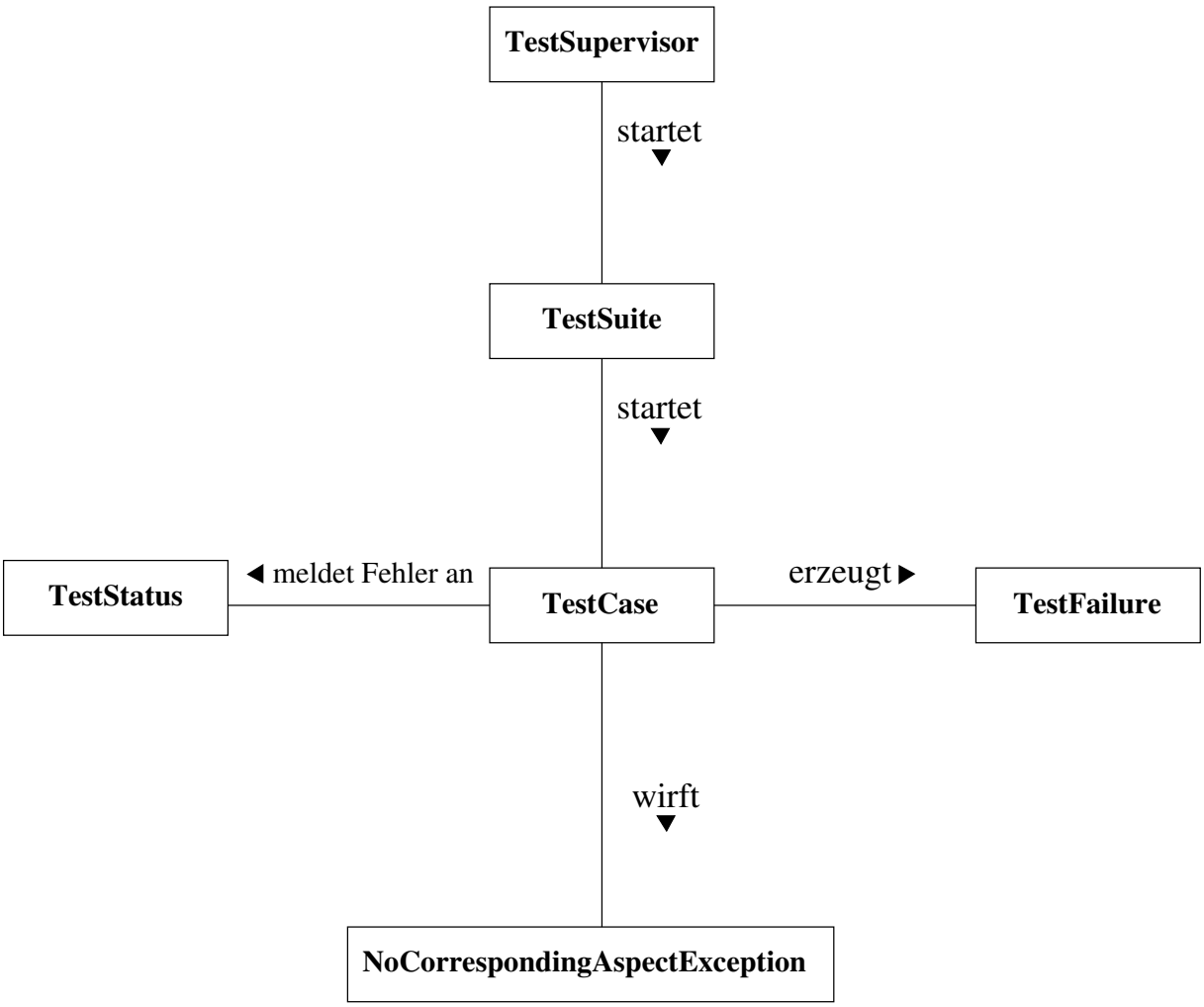


Abbildung A.4.: Funktionale Beziehungen zwischen Klassen im Framework

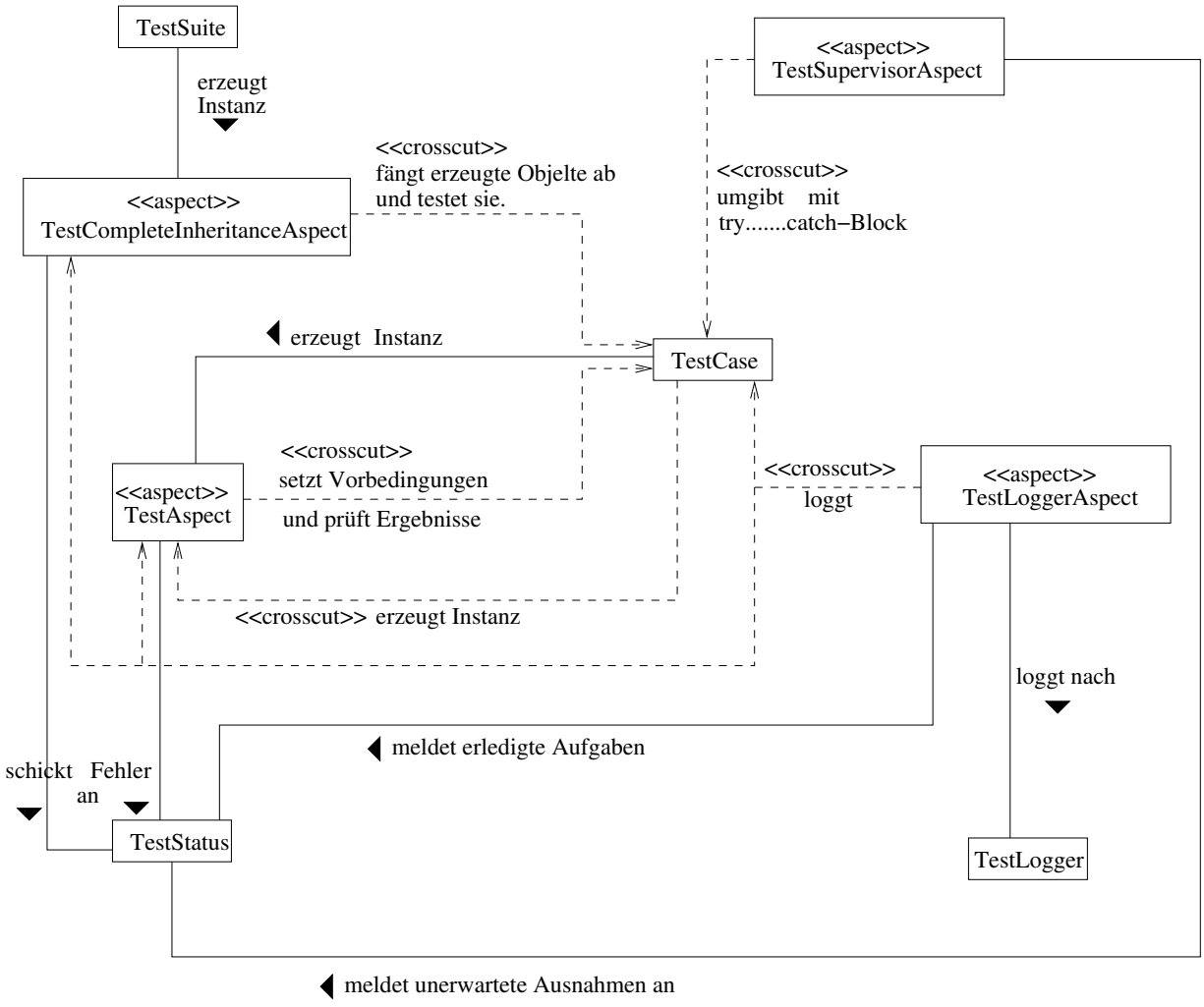


Abbildung A.5.: Wirken der Aspekte

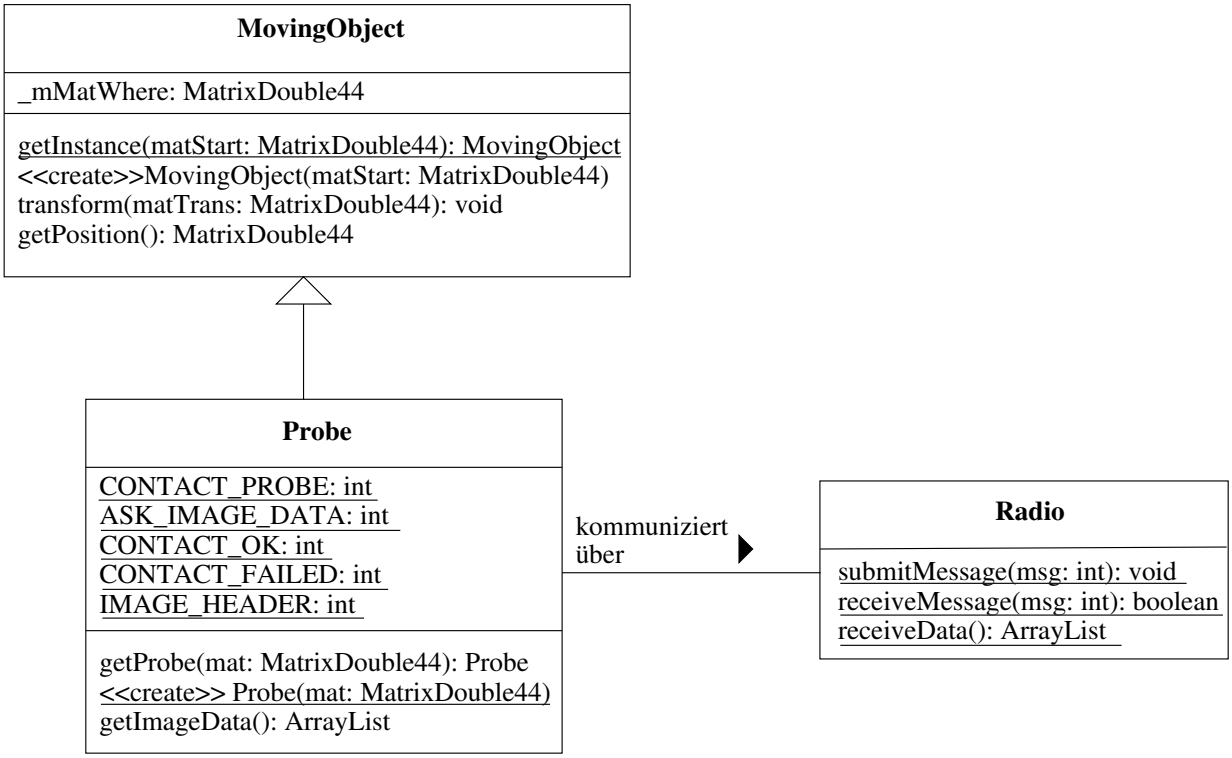


Abbildung A.6.: applicationUnderTest Package

## A.2. Sequenzdiagramme

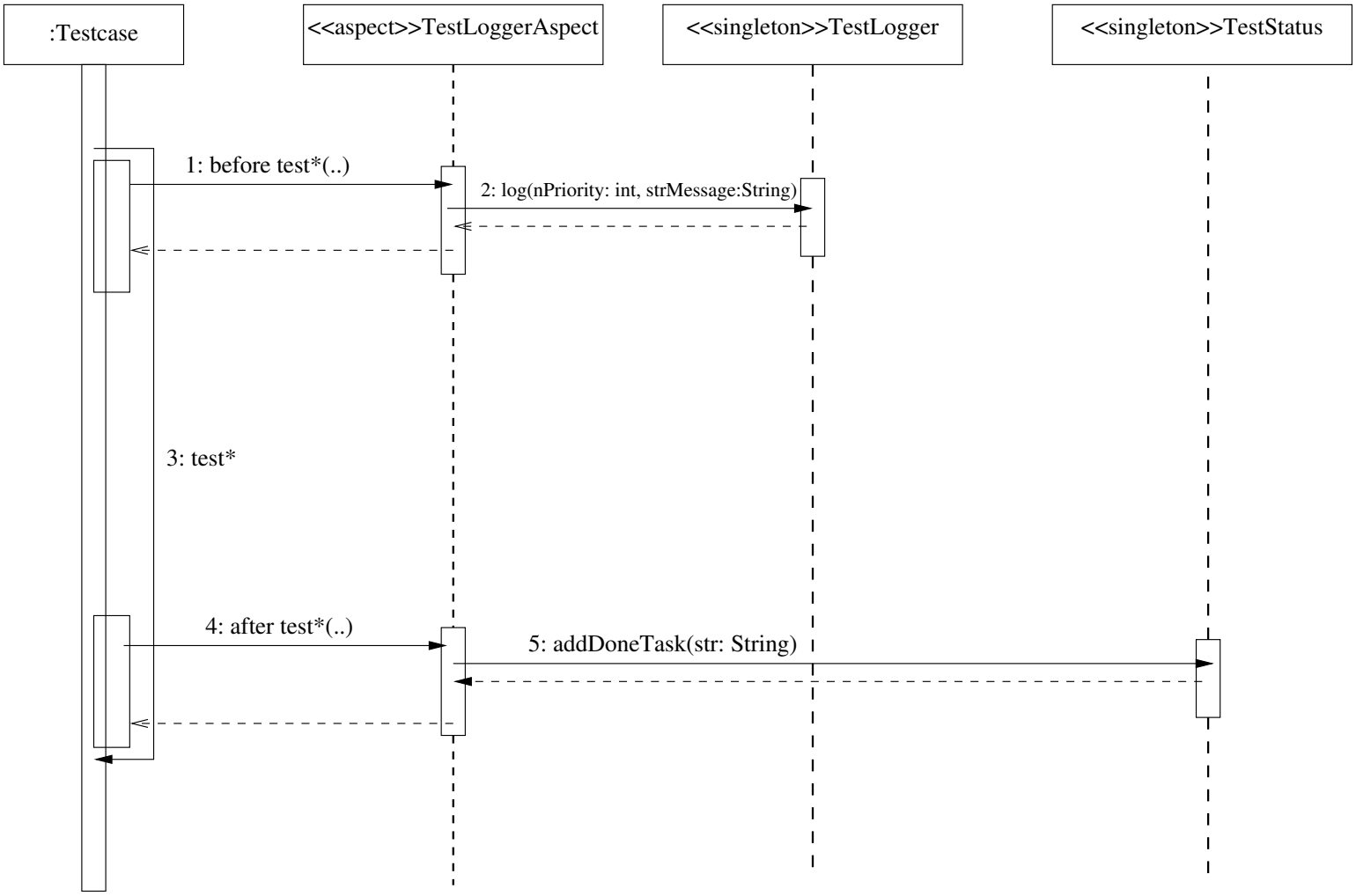


Abbildung A.7.: Ablauf des Loggings im Framework

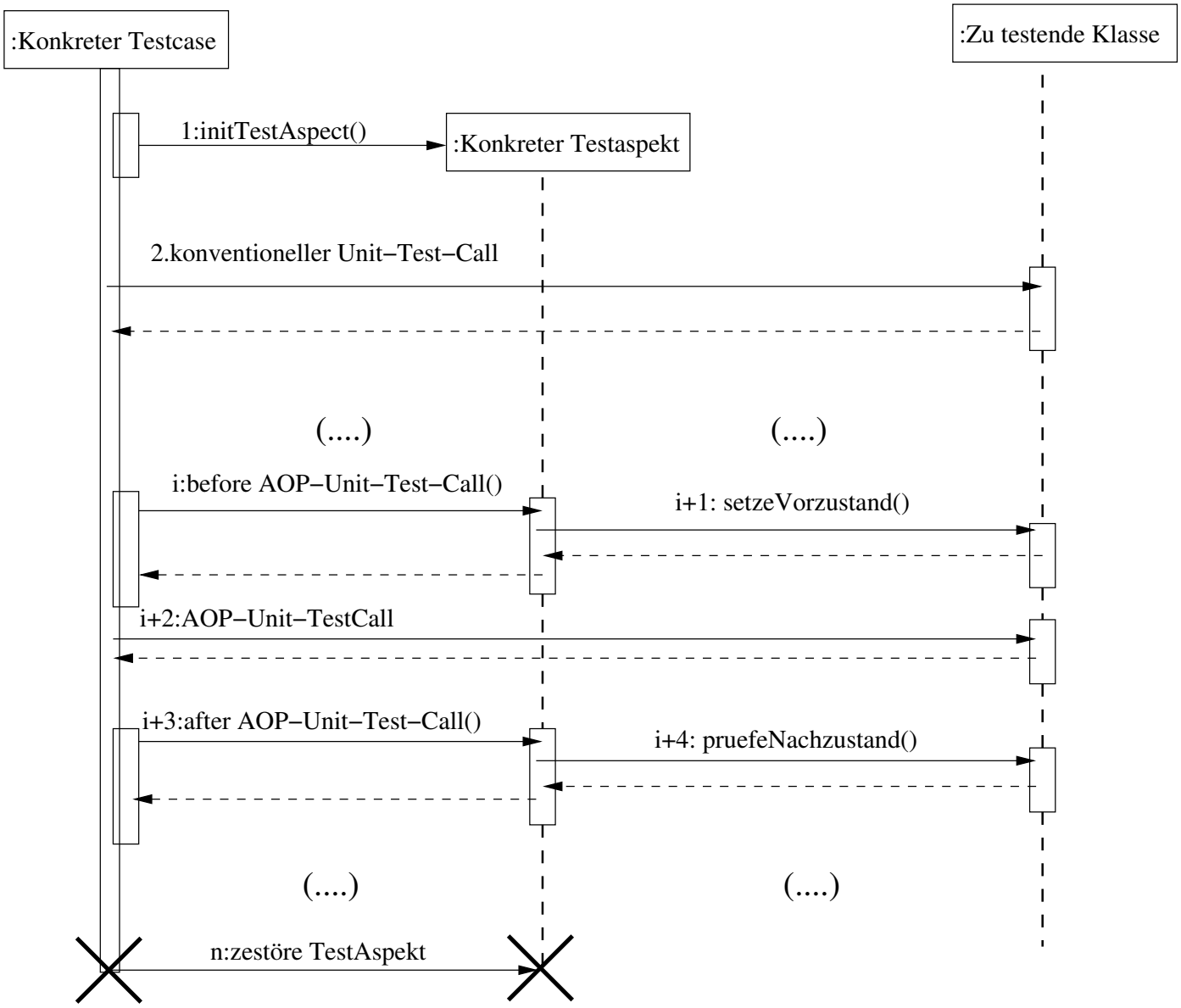


Abbildung A.8.: Lebenszyklus und Wirken von TestAspects

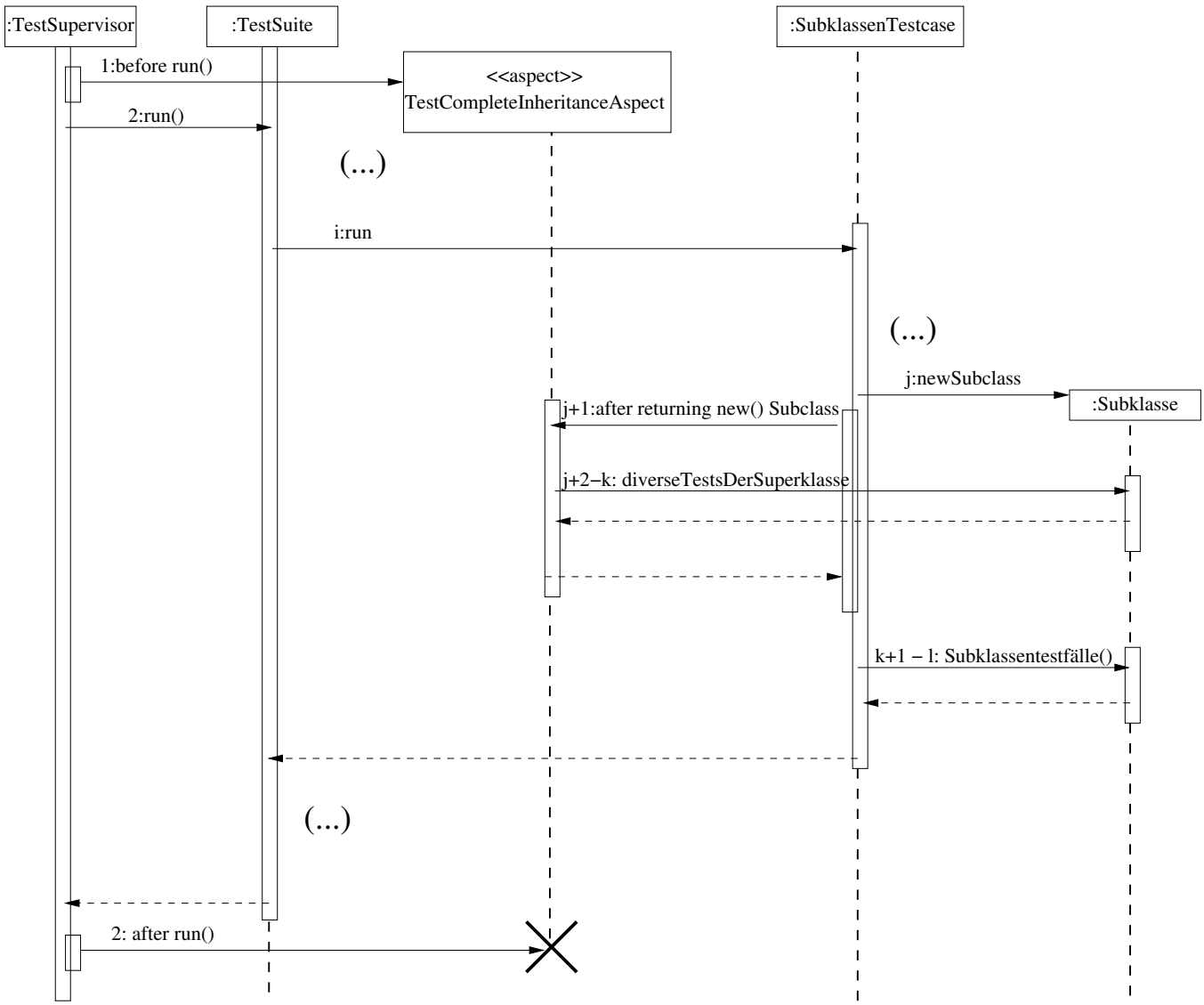


Abbildung A.9.: Wirkung von TestCompleteInheritanceAspect

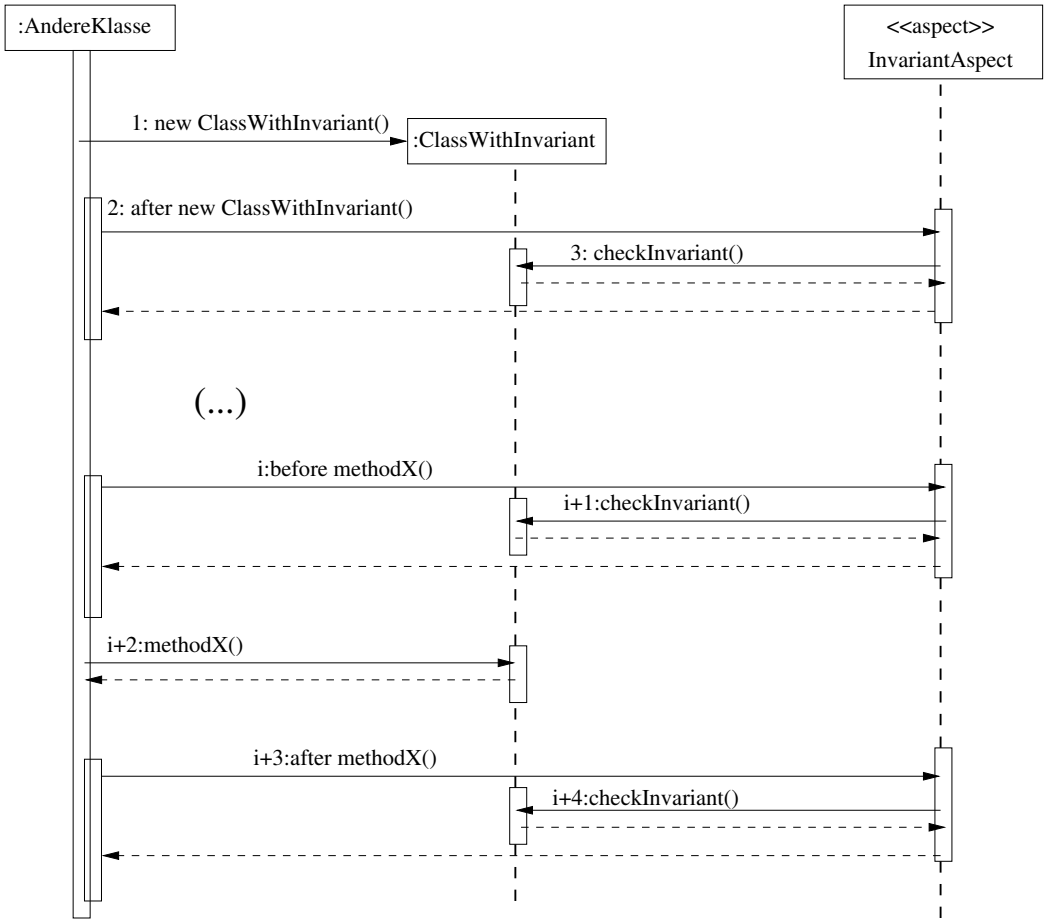


Abbildung A.10.: Durchsetzen von Klasseninvarianten

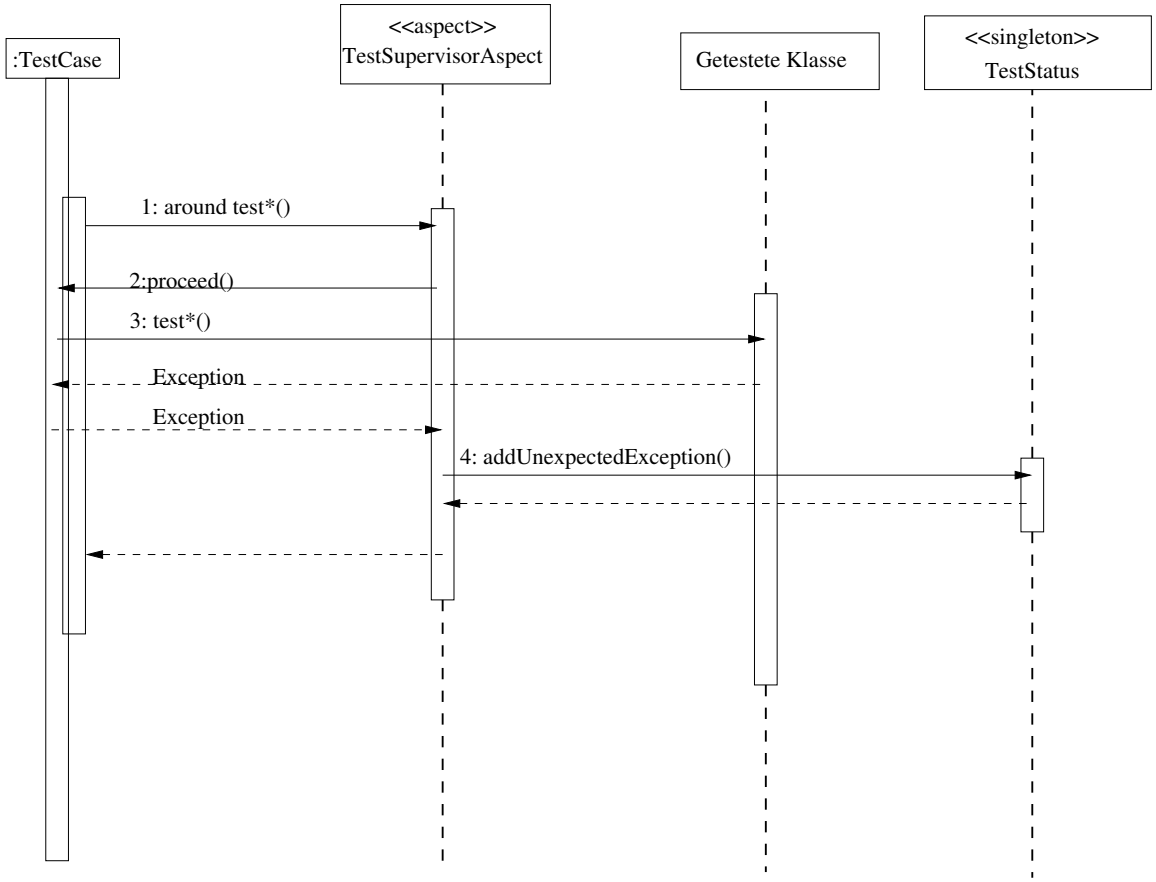


Abbildung A.11.: Abfangen unerwarteter Ausnahmen

# B. Codeausschnitte

Der hier abgebildete Code entspricht der Version des Frameworks vom 30.4.2004.

In der endgültigen Version kann in einigen Kleinigkeiten vom hier abgedruckten differieren. Am Wesentlichen ändert sich nichts.

## B.1. Ausgewählte Klassen des Testframeworks

### B.1.1. TestEntity

```
public abstract class TestEntity
{
  /**
   * Fügt einen Fehler mit dem übergebenen Fehlerstring hinzu,
   * falls die zu checkende Bedingung false ist.
   * @param bCondition Die zu checkende Bedingung
   * @param strText        Beschreibungstext
   */
  public void check(boolean bCondition, String strText)
  {
    if (!bCondition)
      _statusInstance.addError(new TestError(this.getClass().toString(),
                                             strText));
  }


  /**
   * Fügt einen Fehler hinzu, falls die zu checkende Bedingung false ist.
   * @param bCondition    Die zu checkende Bedingung
   */
  public void check(boolean bCondition)
  {
    if (!bCondition)
      _statusInstance.addError(new TestError(this.getClass().toString()));
  }


  final static protected  TestStatus _statusInstance  = TestStatus.getInstance();
}
```

### B.1.2. TestCase

```
public abstract class TestCase extends TestEntity
{
  // Stellt sicher, dass immer wenn ein Aspekt initialisiert werden, dieser auch
  //  mit Sicherheit existiert.
  // Es wird geprueft, ob er diese Methode umgeht.
  public final void initTestAspect() throws Exception
  {
    throw new NoCorrespondingAspectException("Corresponding Test
                                              aspect not in scope!");
  }


  public abstract void run() throws Exception;
}
```

### B.1.3. TestSuite

```
public class TestSuite
{
  /**
   * Konstruktor mit Name für die TestSuite
   * @param strName  Jede Testsuite erhält einen Namen mit diesem String.
   */
  public TestSuite(String strName)
  {
    _strName = strName;
    _testCases = new ArrayList();
  }


  /**
   * Fügt der Liste einen neuen TestCase hinzu.
   * @param tc  Der neue TestCase
   */
  public void addTestCase(TestCase tc)
  {
    _testCases.add(tc);
  }
```

```
  /**
   * Hier werden alle TestCases, die in der Liste enthalten sind, gestartet und
   * durchlaufen.
   * @throws Exception
   */
  public void run() throws Exception
  {
    long nBeforeTime = System.currentTimeMillis();
    TestLogger.getInstance().log(TestLogger.MEDIUM, "Starte TestSuite" + _strName);
    ListIterator it = _testCases.listIterator();
    while(it.hasNext())
    {
      TestCase currentCase = (TestCase) it.next();
      currentCase.run();
    }
    long nTime = System.currentTimeMillis() - nBeforeTime;
    TestStatus.getInstance().addDoneTask("TESTSUITE " + _strName + " in "
                                         + nTime + " ms durchlaufen.");
  }


  private String _strName;
  private ArrayList _testCases;
}
```

### B.1.4. TestSupervisor

```
public final class TestSupervisor
{
  /**
   * Konstruktor, erzeugt eine leere TestSuite-Liste
   *
   */
  public TestSupervisor()
  {
    _testSuites = new ArrayList();
  }


  /**
   * Fügt dem Supervisor eine TestSuite hinzu.
   * @param ts Die neue TestSuite
```

```
   */
  public void addTestSuite(TestSuite ts)
  {
    _testSuites.add(ts);
  }


  /**
   * Durchläuft alle TestSuites.
   * @throws Exception
   */
  public void runTests() throws Exception
  {
    ListIterator it = _testSuites.listIterator();
    long nBeforeTime = System.currentTimeMillis();
    while(it.hasNext())
    {
      TestSuite currentSuite = (TestSuite) it.next();
      currentSuite.run();
    }
    long nTime = System.currentTimeMillis() - nBeforeTime;
    TestStatus.getInstance().addDoneTask("TESTSUITE-SAMMLUNG durchlaufen in "
                                         + nTime + " ms.");
  }

  /// Die Liste mit den TestSuites
  private ArrayList _testSuites;
}
```

### B.1.5. TestFailure

```
public abstract class TestFailure
{
  /**
   * Liefert den String, der den Ort beschreibt.
   * @return Der String, der den Ort beschreibt
   */
  public String getWhere()
  {
    return _strWhere;
  }
```

```
  /**
   * Liefert den String, der den Fehler allgemein beschreibt
   * @return Der String, der den Fehler allgemein beschreibt.
   */
  public String getDescription()
  {
    return _strDescription;
  }

  protected String _strWhere;
  protected String _strDescription;
}
```

### B.1.6. TestStatus

```
public final class TestStatus
{
  /// Die Instanz von TestStatus
  private static TestStatus _instance;

  /**
   * Privater Konstruktor, der von getInstance aufgerufen wird, falls nötig.
   * Alle Variablen werden initialisiert und auf null oder leer gesetzt.
   *
   */
  private TestStatus()
  {
    _TestErrors = new ArrayList();
    _TestWarnings = new ArrayList();
    _TestUnexpectedExceptions = new ArrayList();
    _strTasksDone = "";
    _strTasksDonePrefix = "";
  }


  /**
   * Falls es im System eine gültige TestStatus-Instanz gibt, wird diese geliefert,
   * ansonsten ein neues Objekt konstruiert.
   * Es gibt immer nur ein Objekt vom Typ TestStatus im System.
   * @return Das gültige TestStatus-Objekt.-
   */
  public static TestStatus getInstance()
  {
```

```
  if (_instance == null) _instance = new TestStatus();
  return _instance;
}


/**
 * Liefert die Anzahl der Fehler
 * @return Die Anzahl der Fehler
 */
public int getNoOfErrors()
{
  return _TestErrors.size();
}

(.... Getter für Anzahl Warnungen und Anzahl UnexpectedExceptions ....)


/**
 * Fügt der _TestWarnings-Liste eine Warnung w hinzu
 * @param w   Die neue Warnung für die Liste
 */
public void addWarning(TestWarning w)
{
  _TestWarnings.add(w);
}



/**
 * Fügt der _TestErrors-Liste einen Fehler e hinzu.
 * @param e   Der neue Fehler für die Liste
 */
public void addError(TestError e)
{
  _TestErrors.add(e);
}

/**
 * Fügt der _TestUnexpectedExceptions-Liste eine neue unerwartete Ausnahme hinzu
 * @param u   Die neue unerwartete Ausnahme für die Liste
 */
public void addUnexpectedException(TestUnexpectedException u)
{
  _TestUnexpectedExceptions.add(u);
}
```

```
  /**
   * Fasst die aktuelle Statistik in einem deutschen String zusammen.
   * @return Der zusammenfassende String
   */
  public String getStatistics()
  {
    String strStat = _strTasksDonePrefix + _strTasksDone;
    if (_TestErrors.isEmpty()) strStat +=
                  "\n\nKeine Fehler oder Warnungen im Testdurchlauf.\n\n";
    else
    {
      strStat += "\n\nFolgende Fehler und Warnungen sind aufgetreten:\n";
                (....viel Text....)
    }
  }
```

## B.2. Ausgewählte Aspekte des Testframeworks

### B.2.1. TestAspect

```
public abstract aspect TestAspect extends TestAspectEntity
      perthis(initMe() && initSuper())
{
  /**
   * Sub-Klassen des Aspekts sollen über diesen Pointcut initialisiert werden.
   */
  protected pointcut initSuper() :
    call(* TestCase.initTestAspect(..));

  /**
   *  Dieser Pointcut definiert, zu welcher Klasse der Aspekt gehoert
   */
  public abstract pointcut initMe();

  /**
   * Verhindert die Ausführung von TestCase.initTestAspect(), die sonst
   * eine Ausnahme werfen würde.
   */
  void around() :
    initSuper()
```

```
  {}

  /**
   *  Dieser Pointcut umgibt alle check()-Aufrufe, in dem Test-Case, der der
   * jeweiligen Sub-klasse zugeordnet
   * ist.
   */
  public pointcut testCaseCheck(boolean bCondition) :
    call (* TestEntity.check(boolean))
    && within(TestCase+)
    && args(bCondition);

  /**
   * TestCase.check() wird durch TestAspectEntity ersetzt. So können auch die
   * konventionellen Tests von den AOP-
   * Reflexionsfähigkeiten profitieren.
   */
  void around(boolean bCondition) :
    testCaseCheck(bCondition)
  {
    check(bCondition, thisJoinPoint);
  }
}
```

### B.2.2. TestCompleteInheritanceAspect

```
public abstract aspect TestCompleteInheritanceAspect extends TestAspectEntity
        percflow(newInstance())
{
  /**
   * Dieser pointcut gibt den Punkt an, an dem eine neue Instanz
   * des TestCompleteInheritanceAspects erzeugt werden soll. Bei jedem
   * neuen Durchlaufen einer Testsuite
   */
  public pointcut newInstance() :
    call (* TestSuite.run());


  /**
   * Dieser abstrakte Pointcut dient dazu, Kreationen von Klassen abzufangen,
   * die getestet werden sollen.
   */
  public abstract pointcut newSubclass();
```

```
  /**
   * Nach erfolgreicher Kreierung einer zu testenden Klasse. Wird geprüft, ob
   * eine Klasse dieses Typs schon getestet wurde. Wenn nein, wird sie getestet
   * und danach ihr Klassenobjekt in die Liste der geprüften Klassen geschrieben.
   * HINWEIS: Um Seiteneffekte zu vermeiden sollte in einem konkreten
   * TestCompleteInheritanceAspect ein Klon getestet werden.
   */
  after()  returning (Object newObj):
    newSubclass()
  {
    if (!_setAllTestedSubclasses.contains(newObj.getClass()))
    {
      _setAllTestedSubclasses.add(newObj.getClass());
      runTests(newObj);
    }
  }


  /**
   * Enthält alle Tests, die die Sub-klasse durchlaufen muss.
   */
  public abstract void runTests(Object obj);


  /// Liste mit allen schon getesten Klassen
  HashSet _setAllTestedSubclasses = new HashSet();
}
```

### B.2.3. TestSupervisorAspect

```
public aspect TestSupervisorAspect
{
  // Die gültige Instanz von TestStatus
  TestStatus _statusInstance = TestStatus.getInstance();

  /// Definiert die Prioritäten beim Weben der Aspekte
  declare precedence : TestLoggerAspect, TestAspect+,
                       TestCompleteInheritanceAspect+, TestSupervisorAspect;


  /**
```

```
   * Umgibt alle durch den Namen als testMethoden identifizierten Blöcke
   * mit einem try-catch-Block, um unerwartete Ausnahmen zu fangen.
   */
  pointcut testMethodCall() :
      execution (void *.test*(..));

  /**
   * Tritt in einer TestMethode eine unerwartete Ausnahme auf,
   * wird makeUnexpectedException aufgerufen.
   */
  void around() : testMethodCall()
  {
    Object obj = null;
    try
    {
      proceed();
    }
    catch(Throwable ex)
    {
      makeUnexpectedException(thisJoinPoint);
    }
  }

  /**
   *
   */
  public void makeUnexpectedException(JoinPoint jp)
  {
    String errorWhere = "";
    SourceLocation sl = jp.getSourceLocation();
    errorWhere += sl.getFileName() + ": Zeile " + sl.getLine();
    _statusInstance.addUnexpectedException(new TestUnexpectedException(errorWhere));
  }
}
```

# C. Definitionen häufig benutzter Begriffe

Hier werden einige Begriffe definiert, die in der Arbeit häufiger Verwendung finden, und den nur überfliegenden oder mit der Informatik weniger vertrauten Leser unbekannt sein könnten.

**Advice** Legt fest, was auf benutzerdefinierten Pointcuts getan werden soll.

**AOP, Aspektorientierte Programmierung** Siehe Aspektorientierung.

**AspectJ** Die wohl aktuell am weitesten verbreitete, aspektorientierte Sprache. Sie erlaubt unter anderem Bindung von Aspekten an Objekte und Kontrollflüsse, Vererbung von Aspekten, privilegierte Aspekte und Aspektpriorisierungen.

**Aspekt** Ein Objekt, welches definierte Pointcuts und Advices enthält. In vielen Sprachen kann ein Aspekt wie ein Objekt in einer objektorientierten Sprache behandelt werden, was z.B. Vererbung erlaubt. Aspekte sind in der Regel global und statisch, können aber auch je nach Sprache für ein Objekt oder einen Programmflussabschnitt definiert werden. Sie werden dann an den jeweiligen Punkten erzeugt und vernichtet. Ihre Pointcuts beziehen sich nur auf die instanziierende Entität.

**Aspekte, privilegierte** AspectJ Sprachmittel. Privilegierten Aspekten ist es erlaubt, auf durch Kapselung geschützte Methoden und Variablen einer Klasse zuzugreifen.

**Aspektorientierung** Softwaretechnisches Konzept, dessen Ziel es ist, Cross-Cutting Concerns auf modulare und flexible Weise behandeln zu können.

**Client** Eine Einheit in einem System, die von anderen Einheiten etwas fordert.

**Concern** Anforderung an ein Programm, ein semantisch herausstellbares Anliegen.

**Cross-Cutting Concern** Ein Concern, der sich nicht leicht modularisieren lässt, weil er in vielen verschiedenen Stellen des Systems auftrittwie im Logging oder weil er aus verschiedenen Stellen im System zusammengesetzt ist.

**Extreme Programming** Von Kent Beck begründeter, umstrittener Softwareentwicklungsprozess. [2].

**Framework** Der Rahmen eines Softwaresystems, in dem allgemeine Aufgaben schon erfüllt sind. Der Benutzer eines Frameworks muss es noch mit spezifischen Verhalten füllen.

**Implementation-based Testing** Testverfahren, in dem die Testcases nach Kriterien erstellt werden, die in der konkreten Implementierung des Programms begründet sind. Beispielsweise werden Testcases so erzeugt, dass sie den gesamten Code abdecken.

**Integrationstest** Test des Zusammenspiels eines Subsystems mit anderen Subsystemen.

**Instrumentierung** Ermöglichen der Informationsgewinnung über Abläufe und Zustände des Programms.

**Joinpoints** Punkte im Programmablauf, die eine aspektorientierte Sprache identifizieren kann. Sie können statisch sein, wie der Aufruf einer bestimmten Methode oder dynamisch wie das Überschreiten eines Schwellenwertes in einer Variablen.

**Klasse** Eine Einheit zusammengehöriger Methoden und Variablen mit klar definierter Schnittstelle zu anderen Teilen des Systems.

**Liskov-konform** Eigenschaft einer Klassenhierarchie. In Liskov-konformen Klassenhierarchien müssen die Methoden von Subklassen dasselbe äußere Verhalten ausweisen wie ihre Superklassen.

**Mock Object** Objekt, das ein anderes ersetzt und meistens ein einfacheres Verhalten aufweist. Es unterscheidet sich vom Stub darin, auch das Verhalten seiner Clients zu überprüfen.

**Mutationstest** Veränderungen am Code, die der Testsuite als Fehler auffallen sollten. Dienen dazu, Testsuites zu testen.

**Objektorientierte Softwareentwicklung** Softwareentwicklungs-Paradigma. Ein System wird mit Hilfe von Objekten mit klar definierten Verhalten und Eigenschaften - Klassen - modelliert.

**Pointcut** Ein Joinpoint, für den ein Advice in einem Aspekt definiert wurde.

**Quantifizierung** Das bezeichnende Erfassen von Objekten. In der Mathematik kann man Zahlen z.B. mit Hilfe der Quantoren $\forall$ und $\exists$ quantifizieren.

**Reflexion** Fähigkeit einer Programmiersprache ihre eigenen Sprachkonstrukte in einem ihrer Programm zu analysieren.

**Regressionstest** Das wiederholte Testen des gesamten Codes auch bei nur stellenweisen Änderungen.

**Responsibility-Based Testing** Testcases werden basierend auf den Aufgaben der zu testenden Einheit entwickelt. Die Implementierung wird dabei weniger beachtet.

**Scope** Wirkungsreichweite oder Gültigkeitsbereich eines Objekts [4].

**Server** Einheit in einem System, die Dienste an andere Einheiten anbietet.

**Stub** Minimalimplementierung eines Subsystems. Wird benötigt, wenn das System auf höherer Ebene getestet werden soll und das Original-Subsystem nicht verfügbar ist.

**Testsuite** Eine Sammlung von Testfällen.

**Treiber** Ein Objekt, das einen Testfall aufruft [4].

**UML** Unified Modelling Language. Modellierungssprache für objektorientierte Softwareentwicklung [57].

**Unit-Test** Testmethode, die aus dem Extreme Programming stammt. Die Module der niedrigsten Granularitätsebene, i.d.R. Methoden, werden isoliert getestet. Alle Tests werden möglichst oft wiederholt, damit Fehler durch Seiteneffekte neuer Ergänzungen nicht übersehen werden.

**Weben** Das Einfügen von Advice-Code an die durch Pointcuts bezeichneten Stellen in der Kompilierphase (Compile-Time Weaving) oder zur Laufzeit (Run-time Weaving).

# Abbildungsverzeichnis

# Literaturverzeichnis


[1] Omar Aldawud, Tzilla Elrad und Atef Bader. UML Profile for Aspect-Oriented Software Development. International Conference on Aspect-Oriented Software Development, 2003. URL: `http://lglwww.epfl.ch/workshops/aosd2003/papers/AldawudAOSD_UML_Profile.pdf`.

[2] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[3] Boris Beizer. The Black Box Vampire, 1998. URL: `http://www.pori.tut.fi/~stenberg/TheBlackBoxVampire.pdf`.

[4] Robert Binder. *Testing Object-Oriented Software*. Addison-Wesley, 1999.

[5] R.J.A. Buhr. A Possible Design Notation for Aspect Oriented Programming. European Conference on Object-Oriented Programming, 1998. URL: `http://trese.cs.utwente.nl/aop-ecoop98/papers/Buhr.pdf`.

[6] Morgan Deters und Ron K. Cytron. Introduction of Program Instrumentation using Aspects. Conference on Object-Oriented Programming Systems, Languages, and Applications, 2001. URL: `http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/15-deters.pdf`.

[7] Robert E.Filman und Daniel P. Friedman. Aspect Oriented Programming is Quantification and Obliviousness. Conference on Object-Oriented Programming Systems, Languages, and Applications, 2000. URL: `http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/filman.pdf`.

[8] Robert E.Filman und Klaus Havelund. Source-Code Instrumentation and Quantification of Events. International Conference on Aspect-Oriented Software Development, 2002. URL: `http://www.cs.iastate.edu/~leavens/FOAL/papers2002/TR.pdf`.

[9] Erik Ernst. Seperation of Concerns. International Conference on Aspect-Oriented Software Development, 2003. URL: `http://www.daimi.au.dk/~eernst/splat03/papers/Erik_Ernst.pdf`.

[10] Erich Gamma, Richard Helm und Ralph E. Johnson. *Design Patterns*. Addison-Wesley, 1997.

[11] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. Net.ObjectDays, 2002. URL: http://www.netobjectdays.org/pdf/02/papers/node/0249.pdf.

[12] Christine Hundt. Diplomarbeit: Bytecode-Transformation zur Laufzeitunterstützung von aspekt-orientierter Modularisierung mit Object-Teams/Java, 2003. URL: http://www.objectteams.org/publications/Diplom_Christine_Hundt.pdf.

[13] Cem Kaner. The Impossibility of Complete Testing, 1998. URL: http://www.kaner.com/pdfs/imposs.pdf.

[14] Cem Kaner, Elisabeth Hendrickson und Jennifer Smith-Brock. Managing the Proportion of Testers to (Other) Developers, 2001. URL: http://www.kaner.com/pdfs/pnsqc_ratio_of_testers.pdf.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William G. Grisworld. An Overview of AspectJ. International Conference on Aspect-Oriented Software Development, 2001. URL: http://www.cs.ubc.ca/~gregor/kiczales-ECOOP2001-AspectJ.pdf.

[16] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda und Anurag Mendhekar. Aspect-Oriented Programming, 1996. URL: http://www-sal.cs.uiuc.edu/~kamin/dsl/papers/kiczales.ps.Z.

[17] Sun-Woo Kim. Testing Object-Oriented Programs Using Mutation Techniques, 1998. URL: http://www.cs.york.ac.uk/testsig/publications/sunwoo-jun98.ps.

[18] Michael Kircher, Prashant Jain und Angelo Corsaro. XP + AOP = Better Software? http://www.agilealliance.com/articles/articles/XPplusAOP.pdf.

[19] Irene Koo. Mutation Testing and Three Variations, 1996. URL: http://www.ee.ubc.ca/home/comlab1/irenek/etc/www/techpaps/mutate/mutation.html.

[20] Ramnivas Laddad. *Aspectj in Action: Practical Aspect-Oriented Programming (In Action).* Manning Publications, 2003.

[21] Nicholas Lesiecki. Test Flexibly with AspectJ and Mock Objects. URL: ftp://www6.software.ibm.com/software/developer/library/j-aspectj2.pdf.

[22] Barbara Liskov. Data Abstraction and Hierarchy. Conference on Object-Oriented Programming Systems, Languages, and Applications, 1987. URL: http://www.isse.gmu.edu/~lili/619/Liskov87.pdf.

[23] Barbara H. Liskov und Jeannette M. Wing. A Behavioral Notion of Subtyping, 1994. URL: http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/venari/papers/subtype-toplas/paper.ps.

[24] Tim Mackinnon, Steve Freeman und Philip Craig. Endo-Testing: Unit Testing with Mock Objects, 2000. URL: http://www.connextra.com/aboutUs/mockobjects.pdf.

[25] Robert C. Martin. The Liskov Substitution Principle, 1996. URL: http://www.objectmentor.com/resources/articles/lsp.pdf.

[26] Bertrand Meyer. *Object-Oriented Software Development*. Prentice Hall, 1997.

[27] Andrei Popovici, Gustavo Alonso und Thomas Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for JAVA. International Conference on Aspect-Oriented Software Development, 2003. URL: http://www.iks.inf.ethz.ch/publications/publications/aosd03.ps.

[28] Dominik Stein, Stefan Hanenberg und Rainer Unland. An UML-based Aspect-Oriented Design Notation For AspectJ. International Conference on Aspect-Oriented Software Development, 2002. URL: http://dawis.informatik.uni-essen.de/site/site/publications/papers/aop/StHaUn_AspectOrientedDesignNotation_AOSD_2002.pdf.

[29] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 1998.

[30] Stanley M. Sutton. Multiple Dimensions of Concern in Software Testing. Conference on Object-Oriented Programming Systems, Languages, and Applications, 1999. URL: http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws13-sutton.pdf.

[31] Junichi Suzuki und Yoshikazu Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. European Conference on Object-Oriented Programming, 1999. URL: http://trese.cs.utwente.nl/aop-ecoop99/papers/suzuki.pdf.

[32] Peri Tarr, Harold Ossher und Stanley M. Sutton Jr. Hyper/J: Multi-Dimensional Seperation of Concerns for Java. Net.ObjectDays, 2001. URL: http://www.netobjectdays.org/pdf/01/slides/tutorial/sutton.pdf.

[33] Jianjun Zhao. Unit Testing for Aspect-Oriented Programs. Technical Report SE-141-6, Information Processing Society of Japan (IPSJ), 2003. URL: http://www.fit.ac.jp/~zhao/pub/ps/ipsj-tr-se-141-6.pdf.

[34] Artisan Real-Time-UML. URL: http://www.artisansw.com/.

[35] AspectC++. URL: http://www.aspectc.org.

[36] Aspectj bei eclipse.org. URL: http://www.eclipse.org/aspectj/.

[37] BOOST-Test-Framework. URL: www.boost.org.

[38] Cricket-Cage-Framework. URL: http://sourceforge.net/projects/cricketcage/.

[39] Diskussion über den Begriff Mock Objects und über die Anwendung von Mock Objects. URL: http://groups.yahoo.com/group/testdrivendevelopment/message/5110.

[40] EasyMock. URL: http://www.easymock.org.

[41] Eclipse IDE. URL: http://www.eclipse.org.

[42] Eiffel Software. URL: http://www.eiffel.com/.

[43] Einführung ins Extreme Programming. URL: http://www.extremeprogramming.org/.

[44] Extreme Programming. URL: http://www.xprogramming.com/.

[45] Glen McCluskey & Associates LLC C++ / Java(tm) Consulting. URL: http://www.glenmccl.com.

[46] JAVA 2 Platform SE v1.4.2. URL: http://java.sun.com/j2se/1.4.2/docs/api/.

[47] Java Instrumentation API. URL: http://jiapi.sourceforge.net/.

[48] Java Instrumentation Engine. URL: http://www.forum2.org/eran/jie/.

[49] Java Macro Preprocessor. URL: http://sourceforge.net/projects/jmp/.

[50] Jester - The Test Tester. URL: http://jester.sourceforge.net.

[51] JUnit. URL: www.junit.org.

[52] MockCreator. URL: http://http://mockcreator.sourceforge.net/.

[53] MockObjects.com. URL: http://www.mockobjects.com.

[54] Object Teams. URL: http://www.objectteams.org.

[55] OCL 2.0 specification, release 1.6. URL: http://www.klasse.nl/ocl/ocl-specification-v1-6.zip.

[56] The Java Syntactic Extender. URL: http://jse.sourceforge.net/.

[57] UML-Spezifikation. URL: http://www.uml.org/.

[58] Using Java Reflection. URL: http://java.sun.com/developer/technicalArticles/ALT/Reflection/.

[59] Virtual Mock. URL: www.virtualmock.org.

[60] Yahoo-XP-Newsgroup. URL: http://groups.yahoo.com/group/extremeprogramming/.