Aspektorientierte Programmiertechniken im Unit-Testen

Matthias Vösgen, Dehla Sokenou

Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Softwaretechnik, Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin,
EMail: {mvoesgen|dsokenou}@cs.tu-berlin.de

Zusammenfassung Der Artikel untersucht, wie Probleme, die speziell beim Unit-Testen objektorientierter Programme auftreten, mit Hilfe des Paradigmas aspektorientierte Programmierung gelöst werden können.

Die Untersuchung stellt Schritt für Schritt die einzelnen Teilprobleme vor, führt konventionelle Lösungen auf, zeigt aspektorientierte Alternativen für die gestellten Probleme und erläutert anhand von Code-Beispielen ihre Implementierung in der Sprache AspectJ.

Die Untersuchung begleitend wird das Testframework FlexTest vorgestellt, das die praktische Umsetzbarkeit der vorgestellten Ideen demonstriert.

Schlüsselwörter: Testen, aspektorientierte Programmierung

Abstract The article examines whether problems that occur specifically during unit testing of object-oriented programs can be solved using the aspect-oriented programming paradigm.

It presents the various subproblems step by step, shows conventional solutions, describes aspect-oriented solutions to the problems in a general and language-independent manner, and concludes by looking at the aspect-oriented implementation in the language AspectJ.

Parallel to this, we present the test framework FlexTest, which demonstrates the practical implementability of our ideas.

Key words: Testing, aspect-oriented programming

CR Subject Classification: D.2.5

---

1 Einleitung

Solange die vollständige Verifikation großer Softwaresysteme mit einem (von wenigen Einzelfällen abgesehen) unverhältnismäßig hohen Aufwand verbunden ist, bleibt gründliches Testen eine tragende Säule bei der Entwicklung qualitativ hochwertiger Software. Diese Erkenntnis rechtfertigt die Bemühungen in Forschung [1] und Praxis, die in dieses Themas investiert werden.

Besonders beim Testen objektorientierter Programme treten Konflikte auf, deren Ursache in den Grundkonzepten der objektorientierten Programmierung selbst liegen. Die Konstrukte, die Abstraktion und Wiederverwendung ermöglichen, stellen für den Test ein Hindernis dar, das in der Praxis meist auf umständliche, unpraktikable und manchmal auch gefährliche Weise umgangen wird.

In diesem Artikel wird untersucht, wie mit den Problemen, die speziell beim Unit-Testen objektorientierter Programme auftreten, mit Hilfe des vergleichsweise neuen Paradigmas aspektorientierte Programmierung, kurz AOP, umgegangen werden kann.

Die einzelnen Testprobleme werden dazu nacheinander untersucht und aspektorientierte Lösungen heraus gearbeitet, die konventionellen Herangehensweisen gegenübergestellt werden.

Um die Realisierbarkeit der Ideen zu demonstrieren, wurde ein Testframework in der aspektorientierten Sprache AspectJ [1,13] erstellt, in welchem die Theorie auf reale Testprobleme angewendet werden kann. Dieses Testframework wird die theoretische Untersuchung begleitend vorgestellt.

2 Aspektorientierte Programmierung

Eines der zentralen Konzepte zur Verbesserung der Qualität von Software ist die Modularisierung. Sie hilft, komplexe Probleme auf einfache Teilprobleme (in Form von Modulen) abzubilden und zur Lösung des Problems die Einzelteile zu kombinieren (modulare Dekompositionalität bzw. Kompositionalität [22]). Modularisierung bie-

[1] Siehe dazu z.B. das IFE-Themenheft „Aktuelle Entwicklungen im Softwaretest“ [18].

tet Vorteile in allen Phasen der Softwareentwicklung. In der Analyse werden Probleme zerlegt, im Entwurf einzeln betrachtet und modelliert, in der Implementierung erleichtern Module Teamarbeit und Verantwortlichkeiten und in der Wartung können Änderungen und Erweiterungen bei guter Modularisierung lokal gehalten werden. Zudem können Module durch Rekombination zu neuen Systemen zusammengesetzt und so wiederverwendet werden.

Einige Eigenschaften und Anforderungen (*Concerns*) des System lassen sich jedoch mit objektorientierten Zerlegungstechniken nicht modularisieren. Oft muss bei Paaren von Concerns entschieden werden, nach welchem der Concerns das System zu modularisieren ist. Der andere Concern, der nicht Grundlage des Systementwurfs war, verteilt sich dann über das gesamte System und bildet einen *Crosscutting-Concern*. Typische Beispiele für Crosscutting-Concerns sind Logging, Sicherheit, Synchronisation oder Verteiltheit. Zum Beispiel lässt sich der Server für das Logging sehr gut in einer Logger-Klasse kapseln, die Aufrufe in den Clients sind jedoch über das gesamte System verteilt (siehe Abbildung 1). Das führt zu unübersichtlichen Code, der schwer zu warten ist, da der Logging-Concern über viele Klassen des Systems verteilt ist (*Scattering*) und gleichzeitig mit der Business-Logik des Systems eng verwebt ist (*Tangling*).

Aspektorientierte Programmierung [14] löst diesen Konflikt, in dem das Scattering und Tangling nicht mehr im Quellcode sichtbar ist und der Crosscutting-Concern in einem eigenem Modul als *Aspekt* gekapselt wird. Das Einweben des Aspekts in die Business-Logik des Systems wird nachträglich von einem *Aspektweber* (je nach verwendeter aspektorientierter Sprache zur Compile- oder zur Ladezeit) durchgeführt.

Das Einweben zusätzlicher Funktionalität kann nur an bestimmten Punkten im Kontrollfluss, den *Joinpoints*, erfolgen. Ein Joinpoint kann zum Beispiel der Aufruf, das Eintreten oder das Verlassen einer Methode, der Zugriff auf eine Variable oder das Werfen einer Ausnahme sein. Joinpoints können durch die Quantifizierungsmöglichkeiten [9] der aspektorientierten Sprache zu *Pointcuts* zusammengefasst werden[2]. Ein Pointcut kann zum Beispiel der Aufruf aller Methoden sein, die schreibenden Zugriff auf ein Attribut bieten (Setter-Methoden). Je nach verwendeter Sprache ist eine Quantifizierung mehr oder weniger präzise möglich.

Die zusätzliche Funktionalität wird in einem *Advice* spezifiziert. Dieses Konstrukt ähnelt einer Methode in der objektorientierten Programmierung. Anders als eine Methode wird ein Advice nicht explizit aufgerufen, sondern an Pointcuts gebunden. Dabei gibt es verschiedene Möglichkeiten, wann der Advice-Code ausgeführt werden soll, sowohl in zeitlicher Hinsicht (z. B. vor oder nach dem ursprünglichen Code) als auch kontrollflussabhängig (z. B. nur, wenn die zu adaptierende Methode aus einer bestimmten anderen Methode aufgerufen wurde oder der Aspekt aktiv ist).

Abbildung 2 zeigt die Implementierung des Logging-Beispiels mit Hilfe aspektorientierter Programmierung. Das eigentliche Programm (Module 1 bis n) enthält ausschließlich die Business-Logik. Der Logging-Code ist vollständig in zwei weitere Module ausgelagert. Das Logging-Modul enthält wie in Abbildung 1 den serverseitigen Logging-Code. Die zuvor verteilten Aufrufe der Clients sind nun gekapselt im Logging-Aspekt. Das Weben in den Code geschieht automatisch durch den Aspektweber und ist für den Entwickler transparent. Änderungen am Logging-Code betreffen jetzt nur noch wenige Module. Der Programmierer der Business-Logik muss sich um die Integration des Logging-Codes nicht mehr kümmern (*Obliviousness* [9]). Dies wird verdeutlicht durch die umgedrehte Pfeilrichtung in Abbildung 2.

Unser Ansatz verwendet aspektorientierte Programmiertechniken für das Unit-Testen. Dabei wird das Unit-Testen als ein Crosscutting-Concern betrachtet. Aspektorientierte Programmiertechniken helfen, den Testcode vom zu testenden Code zu trennen und in Aspekte einzukapseln, bieten jedoch darüber hinaus weitere Vorteile. Bevor im Abschnitt 5 die Einsatzmöglichkeiten von aspektorientierter Programmierung im Unit-Testen untersucht werden, folgt zunächst im Abschnitt 3 eine Einführung in das objektorientierte Unit-Testen.

3 Unit-Testen

Softwaretesten findet auf verschiedenen Granularitätsstufen statt. In einem objektorientierten System werden auf der niedrigsten Ebene zunächst Methoden und Klassen getestet, dann die Zusammenarbeit mehrerer zusammengehöriger Klassen und auf der höchsten Ebene schließlich das System als Ganzes.

Unit-Tests befinden sich in dieser Hierarchie auf der untersten Ebene. Der Wirkungsbereich eines Unit-Test ist die kleinste ausführbare Softwareeinheit [4].

Darunter wird in der Objektorientierung meist eine Klasse verstanden. Obwohl zum Testen Nachrichten an Methoden geschickt werden, wird durch die enge Kopplung der Methoden in einer Klasse durch den gemeinsamen Zustand, auf dem sie arbeiten, die Klasse fast ausnahmslos als kleinste testbare Einheit betrachtet [3, 17, 24, 27].

Der Begriff Unit-Test wird hier nicht synonym mit dem Begriff Klassentest verwendet. Wir verwenden den Begriff Unit-Test analog zu seiner Bedeutung im Umfeld des Extreme Programming [11]. Der Wirkungsbereich des Unit-Tests ist in diesem Fall eine einzelne Methode. Für jede Methode werden voneinander unabhängige Testfälle geschrieben. Das Ergebnis des Unit-Tests

[2] Der Begriff *Quantifizierung* ist in diesem Kontext mit dem engl. Begriff *quantification* synonym, der eine Aussage bezeichnet, die den Gültigkeitsbereich einer Sache (in der Prädikatenlogik z.B. eines Prädikats) angibt.

Abbildung 1 Beispiel Logging ohne aspektorientierte Programmierung

Abbildung 2 Beispiel Logging mit aspektorientierter Programmierung

macht eine Aussage über die getestete Methode, nicht jedoch über komplexere Zusammenhänge innerhalb der Klasse oder des Systems.

Zur Unterstützung des Tests dient ein Werkzeug zur automatischen Ausführung und Auswertung der Testfälle. Meist werden diese in der Zielsprache des Systems codiert [2]. So werden z. B. beim JUnit-Framework [12] Testfälle in Java codiert.

Im Folgenden wird untersucht, ob es sich beim Unit-Test um einen Crosscutting-Concern im Sinne der AOP handelt.

4 AOP und Unit-Test

Die aspektorientierte Programmierung birgt wie das objektorientierte Programmierparadigma die Gefahr, an ihrer Intention 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.[3]

Bei der AOP empfiehlt es sich, noch strenger auf ihre sinnvolle Anwendung zu achten. Ihre vor allem durch die Ereignisquantifizierungen mächtigen Ausdrucksmittel üben eine große Macht über den Kontrollfluss aus. Einige wenige unpassend eingesetzte Aspekte können die Lesbarkeit eines Programms drastisch verschlechtern, anstatt die Modularität zu verbessern.

[3] Die objektorientierte Vererbung kann z.B. zum Zwecke der Code-Ersparnis missbraucht werden. Klassen können direkt miteinander verwandt sein, aber unterschiedliches Verhalten oder unterschiedliche Struktur aufweisen. Ein Beispiel dafür ist die Klasse `Stack` in Java, die von der Klasse `Vector` erbt.

4.1 Unit-Testen als Crosscutting-Concern

Aspektorientierte Programmierung ist ein Werkzeug, das entwickelt wurde, um die Beherrschbarkeit von Crosscutting-Concerns zu verbessern. Wenn es sich beim Testen also um einen solchen Crosscutting-Concern handelt, sind sinnvolle Einsatzmöglichkeiten der AOP zumindest wahrscheinlich.

Da der Unit-Test meist als eine Menge von Methodenaufrufen an die zu testenden Objekte implementiert ist, kann bei flüchtiger Betrachtung der Eindruck entstehen, beim Unit-Test handele es sich nicht um einen Crosscutting-Concern.

Praktische Erfahrung mit dem Testen objektorientierter Programme zeigt dem Tester jedoch schnell die Grenzen dieser einfachen Sichtweise.
Beim Testen stellt man fest, dass

- sich die zu testenden Methoden in einem Kontext befinden, von dem zum Testen abstrahiert werden muss.
- Methoden von gekapselten Variablen abhängen und sich auf solche auswirken. In der Testphase (und nur dann) muss es möglich sein, diese Variablen zu lesen und zu verändern.

Diese beiden Anforderungen an den Code in der Testphase lassen sich zwar in wenigen Worten formulieren, bedeuten aber in der Implementierung, dass sich die Methoden und Klassen im Testkontext in ihrer Struktur und in ihrem Verhalten auf tiefer Ebene unterscheiden. Speziell auf die Kapselung bezogen (auf die in 5.1 näher eingegangen wird) heißt das z.B., dass die Klassen zum Testen transparenter und manipulierbarer sein müssen.

Man kann zusammenfassend sagen: *Testbarkeit ist ein Crosscutting-Concern für die zu testende Klasse.*

Auch der Einsatz von Zusicherungen lässt sich als Crosscutting-Concern verstehen. Zusicherungen verfolgen ähnlich wie der Testprozesses das Ziel der Qualitätssteigerung. Dieses Thema wird in 5.5.3 behandelt.

4.2 Quantifizierung und Reflexion

Der zufriedenstellende Test eines Softwaresystems setzt einen hohen Grad der Abdeckung von Funktionalität und Implementierungsstruktur voraus. Leider muss dazu insbesondere im Unit-Test auch eine große Menge trivialen Test-Codes implementiert werden. In der AOP können allgemeine Bedingungen für eine Menge von Aufrufen vorgeschrieben werden. Auf diesem Wege lassen sich auch Eigenschaften für eine allgemein quantifizierbare Menge von Testfällen festlegen. In 5.2 und 5.3 werden Anwendungen vorgestellt, die vor allem die mächtigen Quantifizierungsfähigkeiten der AOP ausnutzen.

Reflexion ist die Fähigkeit eines Programms, seine eigenen Konstrukte zur Laufzeit wahrzunehmen. Java z.B. ermöglicht über das Reflection-Package ein gewisses Maß an Reflexion. AspectJ, eine aspektorientierte Erweiterung für Java, verfügt über weit stärkere Reflexionsfähigkeiten, die dazu genutzt werden können, die Aufgaben eines Testframeworks zu erleichtern. Während im Java Reflection-Package nur über Strukturen wie Klassen und Methoden reflektiert werden kann, können in AspectJ auch Informationen über den Kontrollfluss gewonnen werden. Ausführlicher wird auf dieses Thema in 5.4 eingegangen.

5 Konkrete Anwendungen der AOP im Unit-Test

Nachdem das Problemgebiet Unit-Testen und das Werkzeug AOP vorgestellt worden sind, geht dieser Abschnitt auf konkrete Anwendungen ein. Dazu werden Teilprobleme des objektorientierten Unit-Testens theoretisch betrachtet und mit JAVA und AspectJ-Beispielen verdeutlicht, warum konventionelle, objektorientierte Sprachmittel nicht ausreichen, die jeweiligen Teilprobleme modular zu lösen. Auf jede Einzelbetrachtung aufbauend wird erörtert, ob und wie die AOP das Teilproblem lösen kann.

Die Beispiele zur Veranschaulichung werden aus einer für diesen Zweck erzeugten Bibliothek mit Datenstrukturen und einigen Operationen aus der linearen Algebra entnommen.

Um den praktischen Nutzen des AOP-Einsatzes zu belegen, entstand das Unit-Testframework FlexTest, das in AspectJ implementiert wurde. Es ähnelt vom Aufbau JUnit, ist aber mit AOP-Zusatzfunktionalität ausgestattet. Die AOP-Lösungsansätze werden also nicht isoliert, sondern als Teil des FlexTest-Frameworks demonstriert. FlexTest schreibt feste Strukturen für die Beziehungen zwischen objektorientiertem und aspektorientiertem Code vor und wirkt so einem zweckentfremdenden Gebrauch der AOP entgegen. Ausserdem nimmt ein Framework dem Tester Arbeit bei der Strukturierung der Tests und der Fehlersuche ab.

FlexTest hätte auf einem existierenden, objektorientierten Framework aufbauen können. Da aber einige zentrale Funktionalitäten eines Testframeworks wie das Logging oder die automatische Fehlerlokalisation mit aspektorientierten Mitteln modularer gelöst werden konnten und Beziehungen zwischen objektorientierten Teilen und aspektorientierten Teilen aufgebaut werden mussten, waren die Erfordernisse so verschieden von denen eines objektorientierten Frameworks, dass es sich als praktischer erwies, ein neues Framework aufzubauen.

FlexTest soll hier aus Platzgründen und zum besseren Verständnis an Anwendungsbeispielen demonstriert werden. Eine vollständige Beschreibung einer älteren Version des Frameworks findet sich in [28].

Zum Verständnis der praktischen FlexTest-Implementierungen sind Kenntnisse in AspectJ erforderlich. Diesem Artikel steht nicht der Platz zur Verfügung, eine elementare Einführung in AspectJ zu geben (siehe dazu [1]).

5.1 Kontrolliertes Durchbrechen der Kapselung

Kapselung ist ein elementarer Bestandteil der objektorientierten Programmierung. Sie behindert aber den Unit-Test, der auf alle Member der getesteten Klassen zugreifen können muss, weil sich die getesteten Methoden auf gekapselte Member-Variablen auswirken oder von ihnen abhängen können. Praktische Erfahrungen zeigen, dass das Durchbrechen der Kapselung in folgenden Fällen unbedingt erforderlich ist:

- Setzen von Zuständen,
- Zustandsüberwachung,
- Testen von Setter- und Getter-Methoden,
- Ausschluss von Fehlermaskierungen.

5.1.1 Konventionelle Lösungen. Das Problem der gezielten Kapselungsdurchbrechung in der Testphase stellt sich mit jedem Software-Projekt, das in einer objektorientierten Sprache geschrieben wurde, neu. Aus der Praxis sind zahlreiche Lösungsansätze und Workarounds zu diesem Problem bekannt, die im folgenden zusammengefasst dargestellt werden.

Manuelles Durchbrechen der Kapselung. Die Kapselung kann in der Testphase umgangen werden, indem der zu testende Code für die Testphase entsprechend vorbereitet wird. Manuelle Vorbereitung, also z.B. die Änderung von gekapselten `private`-Membern zu `public`-Membern, bedeutet einen unverhältnismäßig hohen Arbeitsaufwand, da diese Vorbereitung auch nach jeder Änderung am Anwendungs-Code wiederholt werden muss. In der Entwicklungsphase müssen zwei Versionen des Systems verwaltet werden, die ständig miteinander abgeglichen werden müssen, eine für die eigentliche Anwendung und eine für die Testphase.

Automatisiertes Durchbrechen der Kapselung. Der Vorbereitungsprozess lässt sich mit Präprozessoren oder Code-Generatoren automatisieren, was aber nichts daran ändert, dass zwei Versionen verwaltet werden müssen, was u.U. zu Abstimmungsproblemen zwischen einer Testabteilung und der eigentlichen Entwicklungsabteilung führen kann.

Außerdem wird der Code beim Einsatz von Präprozessor-Makros mit sprachfremden Teilen durchsetzt. Code-Generatoren müssen von außen konfiguriert werden, was je nach Granularität der Kapselungsdurchbrechung nicht unerheblich viel Zeit kosten kann.

Sprachmittel. Die Programmiersprache C++ 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.

Diese Lösung ist praktikabel und wird in der Praxis oft gewählt. Der exklusive Bruch mit der Kapselung führt aber leider eine Abhängigkeit der zu testenden Klasse von den Testklassen ein. Wenn keine bedingte Kompilierung durch Präprozessor-Makros stattfindet, müssen die Testklassen sich also auch bei Release-Kompilierungen immer im Compiler-Scope befinden. Der Test-Concern ist nicht scharf von der eigentlichen Anwendung getrennt.

Bedingte Kompilierung mit Hilfe von Präprozessor-Makros kann dieses Problem beheben. Dadurch weicht man im Code stellenweise von der Implementierungssprache ab oder muss von außen konfigurieren, wie bereits im vorigen Abschnitt über das automatisierte Durchbrechen der Kapselung angesprochen.

5.1.2 Aspektorientierter Ansatz. In [15] wird der Einsatz von privilegierten Aspekten in der Testphase zur gezielten Durchbrechung der Kapselung 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 keine Trennung zwischen Test- und Release-Sourcecode nötig. Im Release ist der Aspekt nicht enthalten.

Soll der eingewobene Code nur das Ergebnis eines Testfalls oder mehrerer Testfälle prüfen, reicht es, einen Pointcut für die Testfallaufrufe, deren Ergebnis im privilegierten Aspekt geprüft werden soll, zu definieren. Im zugehörigen Advice kann auf alle Member getesteter Klassen zugegriffen werden. Möchte man einen Vorzustand setzen, bindet man an denselben Pointcut einen Advice, der einen Vorzustand vor dem Ausführen der Methode setzt.

Auf diese Weise ermöglichen die Sprachmittel der AOP das Durchbrechen der Kapselung exklusiv in einem dafür bestimmten Aspekt. Außerhalb des Aspekts bleibt die Kapselung bestehen.

5.1.3 Implementierung mit FlexTest. In AspectJ lässt sich die Kapselung in einem Aspekt mittels des Schlüsselworts `privileged` bei der Aspektdefinition umgehen.

Die Umsetzung dieser speziellen Idee ist trivial. Der Anspruch an ein Testframework, das durch die AOP ergänzt wird, ist hier weniger, die Funktionalität anzubieten, sondern vielmehr eine klare Struktur einzuhalten, die auf die getesteten Strukturen passt. In objektorientierten Unit-Testframeworks wie z.B. JUnit [12] fasst eine Testklasse verschiedene Testfälle für eine zu testende Klasse zusammen. Die Struktur des Frameworks und die Namen der Testklassen lehnen sich an die Namen und die Struktur des getesteten Codes an.

FlexTest übenimmt dieses Vorgehen. Alle objektorientierten Testklassen erben von der abstrakten Klasse `TestCase`, deren Testfälle sich auf eine Klasse beziehen. `TestCase`s werden in `TestSuite`s vereinigt, welche wiederum in einem `TestSupervisor` enthalten sind.

Zur Ergänzung des Frameworks mit AOP-Hilfsmitteln gibt es den abstrakten Superaspekt `TestAspect`. Die Subklassen von `TestAspect` passen sich an die Struktur des objektorientierten Teils an, indem jede Subklasse von `TestAspect` eindeutig einer Subklasse von `TestCase` zugeordnet wird.

In Abbildung 3 werden die ersten Zeilen eines Aspekts aufgeführt, der die Testklasse `TestVectorDouble4` ergänzt.

Durch das Schlüsselwort `privileged` wird diesem Aspekt erlaubt, die Kapselung von Klassen in seinem Wirkungsbereich zu durchbrechen. Sein Wirkungsbereich wird insofern eingeschränkt, als dass er fest an eine Testklasse gebunden ist. Welche Klasse das ist, wird im Pointcut `initMe()` definiert, der in der Superklasse `TestAspect` abstrakt deklariert wurde.

Von `initMe()` und dem Aufruf der Methode `initTestAspect()` ist die Existenz und die Wirkung der Subklassen von `TestAspect` abhängig, wie man im Code zu `TestAspect` sieht (Abbildung 4).

Die `perthis(...)` Zuordnung besagt mehr, als dass eine Instanz des Aspektes nach `initTestAspect()` erzeugt und von der entsprechenden Instanz der Testklasse existenzabhängig ist. In der Zuordnung nimmt AspectJ eine implizite Einschränkung der Pointcuts auf Joinpoints innerhalb der Instanz der Testklasse vor, so dass nicht unerwünscht Joinpoints aus anderen Klassen adaptiert werden können.

Angenommen der Tester von `VectorDouble4` möchte testen, ob der Konstruktor, dessen Parameterliste aus vier `doubles` besteht, das gewünschte Verhalten (Erzeugen einer neuen `VectorDouble4`-Instanz mit den übergebenen Werten als Komponenten) implementiert. Die Klasse `TestVectorDouble4` kann zwar Testtreiber für diesen Test sein, sie kann die Ergebnisse der Testfälle aber nicht auf direktem Wege prüfen, da die Komponenten des Vektors gekapselt sind (siehe Abbildung 5).

Der oben eingeführte `TestDouble4Aspect` ist privilegiert und der Klasse `VectorDouble4` zugeordnet. Er

```
public privileged aspect TestVectorDouble4Aspect extends TestAspect
{
  public pointcut initMe() :
      within(TestVectorDouble4);
  [...]
}
```

Abbildung 3 Ergänzung der Testklasse TestVectorDouble4

```
public abstract aspect TestAspect extends TestAspectEntity perthis(initTotal())
{
  protected pointcut initTotal() :
   initSuper() && initMe();

  protected pointcut initSuper() :
   call(* TestCase.initTestAspect(..));

  public abstract pointcut initMe();
  [...]
}
```

Abbildung 4 Ausschnitt aus TestAspect

```
  public void testConstructor()
  {
    test1 = new VectorDouble4(1., 2., 3., 4.);
    [...]
  }
```

Abbildung 5 Konstruktortest in TestVectorDouble4

kann die Kapselung in Testfällen von `VectorDouble4` durchbrechen. Dazu muss er den Testfall natürlich mit einem Pointcut selektieren (siehe Abbildung 6).

5.2 Verallgemeinerte Tests

Klassisches Unit-Testen kann eine zeitraubende, repetetive Tätigkeit sein, wenn keine geeigneten Tools existieren, die dem Tester auf der Struktur des zu testenden Systems aufbauend Arbeit bei der Erzeugung und Zuordnung von Testfällen abnehmen.

5.2.1 Der aspektorientierte Ansatz. Methoden, deren Rückgabewerte aus einer kleinen Menge kommen, zeichnen sich dadurch aus, dass viele Überprüfungen wenige verschiedene Ergebnisse liefern. In den vorigen Abschnitten wurde dargestellt, 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, mit einem Pointcut selektieren.

Auf diese Weise kann der Tester bewusst mit Konventionen arbeiten. Testfallaufrufe, die z.B. immer `true` zurückliefern sollen, werden in eine Methode mit einem fest gelegten Namensmuster geschrieben. Die Überprüfung all dieser Aufrufe erfolgt in einem Aspekt mit einem Pointcut, dessen Definition sich auf die Kontrollflusseinheit der Methode bezieht. Das bedeutet, dass jeder Aufruf eines Testfalls innerhalb der durch den Namen kenntlich gemachten Methode in diesem Aspekt geprüft wird.

Der Tester ordnet die Aufrufe, die z.B. `true` liefern in einen Kontext und die, die `false` liefern in einen anderen; die Überprüfung für alle Fälle erfolgt in jeweils einer Zeile in den betreffenden Aspekten.

Wenn der Tester die entsprechenden Pointcuts vorsichtig[4] definiert und mit der AOP vertraut ist, wird ihm durch die Zusammenfassung nicht nur Schreibarbeit abgenommen, es erleichtert u.U. auch die Übersicht. Hunderte von Testfällen können einer Überprüfung zugeordnet und ohne Zwischenüberprüfungen in einem Block dargestellt werden.

[4] Das bedeutet, man achtet darauf, dass die Pointcuts sich exakt auf die intendierten Joinpoints beziehen. Bei unvorsichtiger Definition können die Pointcuts zu allgemein wirken und zu unvorhergesehenen, schwer nachvollziehbaren Ergebnissen führen.

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

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)));
}
[...]
```

Abbildung 6 Aspektorientierte Unterstützung des Konstruktortests in TestVectorDouble4Aspect

5.2.2 Implementierung mit FlexTest. Verallgemeinerungen der Testfälle, wie sie in diesem Abschnitt besprochen wurden, werden von Flextest unterstützt, da sie bereits durch die Quantifizierungsmöglichkeiten von AspectJ zur Verfügung gestellt werden.

Die Pointcuts in den Testaspekten definieren, welche Testfälle allgemein geprüft werden oder zu welchen Testfällen eine allgemeine Vorbedingung gesetzt wird. FlexTest beschränkt den möglichen Einfluß eines allgemeinen Tests: Ein `TestAspect` wirkt sich immer nur auf einen `TestCase` aus. Verwendet man das FlexTest-Framework der Intention der Autoren nach, können nur Testfälle innerhalb *einer* Klasse verallgemeinert werden.

Das Beispiel zur Kapselungsdurchbrechung aus dem vorigen Abschnitt enthielt bereits eine Verallgemeinerung, weil alle `VectorDouble4`-Konstruktionen in der Klasse `TestVectorDouble4` getestet wurden.

Zur Illustration hier ein Beispiel, in dem der Kontext des Testfallaufrufs eine Rolle spielt:

Getestet wird die Funktion `isNormal()`, die einen vierstelligen Vektor auf Normalität prüft. In `testIsNormalPositive()` werden alle Testfälle zusammengefasst und gestartet, die den Wert `true` liefern sollen. Die Testfälle aus `testIsNormalNegative()` sollen alle `false` liefern (siehe Abbildung 7).

Die Aufteilung wurde vorgenommen, damit die verschiedenen Fälle durch Pointcuts über einen Kontext selektiert werden können. In den zugehörigen Advices erfolgt die Überprüfung der Rückgabewerte (siehe Abbildung 8). Wie man sieht, bringt das Verfahren einen Mindestaufwand mit sich, der sich mit der Anzahl der Testfälle und dem Nutzen aus der verallgemeinerten Aussage amortisieren muss.

5.3 Testen auf Liskov-Konformität

Die Ideen der vorigen Abschnitte bezogen sich auf Instrumentierung im weitesten Sinne und das Einsparen von Test-Code. Die hier folgende Anwendung geht in eine andere Richtung. Der aspektorientierte Code wird nun auch als Testtreiber zum automatisierten Testen von Vererbungshierarchien eingesetzt.

Vererbung hat in einem Softwaresystem zweifachen Nutzen [4]:

- Methoden, Subklassen und Strukturen sind leicht wiederverwendbar. Eine Vererbungshierarchie, in der der Schwerpunkt auf diesem Nutzen liegt, wird als *Convenience Inheritance* bezeichnet.
- Sie drückt verwandtschaftliche Beziehungen zwischen den Klassen bezüglich ihres Einsatzgebietes und ihrer Funktionsweise aus, wodurch die Struktur übersichtlicher gestaltet wird. Eine Vererbungshierarchie, in der der Schwerpunkt auf diesem Nutzen liegt, wird als *Representation Inheritance* bezeichnet.

Convenience Inheritance-Hierarchien sparen Zeit und Platz, weil in ihr benötigte Funktionalitäten mit geringer Rücksichtnahme auf die Bedeutung der Struktur vererbt werden. Geschlossene Klassenhierarchien, auf die nur wenige Entwickler zugreifen, können auf diese Weise organisiert werden.

Representation Inheritance kommt den Benutzern der Klassen entgegen, weil die Klassen in ihr so strukturiert sind, dass sie sich leicht in den richtigen Kontext einordnen lassen. In einer sorgfältig entworfenen Representation Inheritance-Hierarchie kann schnell entschieden werden, wie und wo bestimmte Klassen eingesetzt werden können und sollen. Systeme oder Bibliotheken, mit denen sich viele Entwickler beschäftigen, sollten besser nach dem Prinzip der Representation Inheritance organisiert 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.“ [19]*

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. Eine überschriebene Methode in einer Subklasse muss dasselbe Ergebnis zurückliefern wie die Superklassenme-

```
public void testIsNormalPositive()
  {
    VectorDouble4 vec2 = new VectorDouble4(0, 1, 0, 0);
    vec2.isNormal();
    VectorDouble4 vec3= new VectorDouble4(1, 0, 0, 0);
    vec3.isNormal();
    [...]
  }

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

Abbildung 7 Testfallaufrufe in TestVectorDouble4

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

  after() returning(boolean retValue) :
    vectorDouble4PositiveIsNormalTest()
  {
    if (!retValue) makeError(thisJoinPoint);
  }


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

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

Abbildung 8 Test der Aufrufe aus Abbildung 7

thode, darf aber auf andere Wegen zum Ergebnis kommen. Beispielsweise sollte die Methode `getSmallest()` einer Superklasse `SortedOutput` sowohl in einer sortierten, verketteten Liste als auch in einem Heap das kleinste Element zurückliefern.

Eine Testsuite, in der Liskov-konforme Klassen getestet werden, sollte die Testfälle der Liskov-konformen Superklassen auch auf die Subklassen anwenden.

5.3.1 Konventionelle Lösungen. In [4] wird ein Testverfahren zum Testen Liskov-konformer Klassen vorgeschlagen. Das Verfahren schreibt vor, jede Superklassenmethode im Kontext jeder Subklasse auszuführen. Darüber hinaus wird die Vererbung von Testfällen der Superklassen-Testsuites für die Subklassen-Testsuites empfohlen.

Die Testfälle der Superklassen-Testsuite müssen dann in jeder Subklassen-Testsuite mit der richtigen Subklasse als Argument aufgerufen werden. Bei einfachen Hierarchien ist der Zusatzaufwand eher gering. In komplexen 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 analog zur veränderten Klassenhierarchie umstrukturieren.

Zumindest bei komplexen Hierarchien kann man darin einen Crosscutting-Concern sehen.

5.3.2 Aspektorientierter Ansatz. 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 ausgewählte Tests einer Klasse für jede ihrer Subklassen durchlaufen werden; vorausgesetzt es wird in der Testsuite eine Instanz der Subklasse erzeugt. Dazu genügt ein Joinpoint in der Superklassen-Testsuite, der die erste Kreation eines Subobjekts in einer Klasse, die als Testklasse identifizierbar ist, selektiert und dieses Subobjekt 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 das Ergebnis des Konstruktors, damit ihm ein korrekt konstruiertes Subklassenobjekt zum Testen zur Verfügung steht.

Die Superklassentests werden auf diese Weise aus dem Kontext des Subklassentests im Quellcode herausgezogen. Die Entwickler der Subklassen, die für ihre eigenen Tests verantwortlich sind, müssen sich weder um die Superklassentests kümmern, noch müssen sie Testhierarchien verwalten.

Der Tester erhält mit diesem Testmechanismus nicht nur ein Werkzeug zum Unit-Testen, das ihm die Arbeit abnimmt, Methoden in Subklassen zu testen, die in Superklassen definiert wurden. Es bietet ihm zusätzlich eine Qualitätskontrolle der Vererbungshierarchie.

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 abgeleiteten Klassen testen.

5.3.3 Implementierung mit FlexTest. Die abstrakte Superklasse für alle Vektoren in der `linearAlgebra`-Hierarchie `Vector` implementiert die Methode `clone()`. Diese Methode hängt von den Konkretisierungen der Methoden `getNoOfElements(), setElement()` und `getElement()` in den Subklassen ab.

Ohne etwas über die Implementierung dieser Methoden zu wissen, können an ein durch `clone()` geklonten Subklassenobjekt folgende Forderungen gestellt werden:

- Es ist vom selben Typ wie das Ursprungsobjekt.
- Die Werte der Attribute sind identisch.
- Es ist ein anderes Objekt mit eigener Referenz, d.h. die Werte sind gleich, aber es gilt nicht `Objekt == ObjektGeklont`.

Der Testcode, den man auf alle Subklassen anwenden möchte, ist in Abbildung 9 abgebildet. `TestCompleteInheritanceAspect` aus dem TestFlex-Framework enthält die nötigen Mechanismen, mit denen komplette Hierarchien auf Liskov-Konformität getestet werden können.

Instanzen von Subaspekten von `TestCompleteInheritanceAspect` werden immer einem Durchlauf einer `TestSuite` zugeordnet. Ein solcher Durchlauf ist definiert als der Programmfluss zwischen dem Eintritt in die und dem Austritt aus der Methode `TestSuite.run()` (siehe Abbildung 10). Jede zu testende Subklasse wird nur einmal getestet. Welche Klassen Subklassen sind, wird durch den Pointcut `newSubclass()` im jeweiligen Subaspekt von `TestCompleteInheritanceAspect` bestimmt. Die Tests, die auf diesen Subklassen wirken, stehen in der Methode `runTests()`.

Abbildung 11 zeigt eine Implementierung des Beispiels. Beim Durchlauf der Testsuite geschieht folgendes: Wird eine Subklasse von `Vector` in einer Subklasse von `TestVector` erzeugt, wird geprüft, ob der Konformitätstest für ein Objekt dieses Typs schon durchlaufen wurde. Wenn ja, passiert nichts. Sollte ein Objekt dieses Typs noch nicht auf Konformität geprüft worden sein, wird die Instanz an die Methode `runTests()` übergeben, wo die nötigen Tests stattfinden.

Die Zuordnung zu einer `TestSuite` anstatt zum vollständigen Durchlauf aller Tests bewirkt die wiederholte Ausführung des Konformitätstests bei zweimaligen Durchlaufen derselben `TestSuite`.

5.4 Komfort-Funktionalität

In den vorigen Abschnitten wurde gezeigt, wie spezielle Probleme des Testens mit der AOP gelöst werden können. Dieser Abschnitt behandelt AOP-Ergänzungen, die den Aufbau und die Funktionalität eines Testframeworks komfortabler und übersichtlicher gestalten können.

5.4.1 Logging. Logging ist das klassische Anwendungsbeispiel für aspektorientierte Programmierung. In nahezu jedem Lehrbuch für aspektorientierte Sprachen wie z.B. [15] wird diese Anwendung ausführlich besprochen.

In einem Testframework ist es sinnvoll, den Eintritt in Testfallmethoden zu loggen. Ein Pointcut mit zugehörigen Advice, der die Logging-Anweisung enthält, reicht aus, um das Problem zu lösen.

5.4.2 Fehlerlokalisation. Die Forderung nach der Fehlerlokalisation[5] 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 spricht die große Menge trivialer Tests, deren vollständige Kommentierung zu viel Zeit kosten würde.

[5] Der Ort, an dem der Fehler bemerkt wurde (z.B. in einer bestimmten Methode) ist gemeint, nicht die Ursache des Fehlers.

```
[...]
void testClone()
  {
    Vector _vecExample2 = (Vector)_vecExample.clone();
    check(_vecExample.equals(_vecExample2));
    check(_vecExample != _vecExample2);
  }
[...]
```

Abbildung 9 `clone()`-Methode in `Vector`

```
 public abstract aspect TestCompleteInheritanceAspect extends TestAspectEntity percflow(newInstance())
{
  public pointcut newInstance() :
    call (* TestSuite.run());
  [...]
}
```

Abbildung 10 Pointcut zur Selektion der Konstruktion einer neuen Instanz einer zu testenden Subklasse

```
public aspect TestVectorAspect extends TestCompleteInheritanceAspect
{
  public pointcut newSubclass():
    call (Vector+.new()) && within (TestVector*);
  [...]
}
```

Abbildung 11 Konkreter Pointcut zur Selektion der Erzeugung einer neuen Subklasse

Betrachtet man die Aufgabe genauer, stellt man fest, dass sich eine Kommentierung der Testfälle, die zur Auftrittslokalisation ausreicht, auf einige wenige verallgemeinerbare 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 mit einem Pointcut zu selektieren, dessen Advice über die Reflexion Auskunft über die Quellcodedatei, den Klassennamen, den Methodennamen und die Zeilennummer, in der der Testfall aufgerufen wurde, erlangen kann. Diese Informationen kann der Advice im Fehlerfall ausgeben oder einer Liste für die Statistik hinzufügen.

In JUnit werden die Strings zur Fehlerlokalisation automatisiert aus Exception-Stacktraces gewonnen. Dabei handelt es sich um eine Zweckentfremdung des Ausnahme-Mechanismusses, da dieser dazu dient, Ereignisse zu behandeln, die den Programmfluss der Anwendung kritisch unterbrechen. Fehlgeschlagene Testfälle sind aber in einer Testsuite normale Ereignisse, die den Programmfluss nicht unterbrechen. Man könnte von einem „Hack“ sprechen. Eine Exception, die in einem Testframework auftritt, sollte sich auf ein kritisches Problem im Framework selbst beziehen, nicht auf einen fehlgeschlagenen Testfall bei der zu testenden Anwendung.

Im FlexTest-Framework wird die Fehlerlokalisation kurz und modularisiert implementiert (siehe Abbildung 12). Damit werden alle Testfälle, die objektorientiert oder in Aspektinstanzen über `check(boolean)` getestet werden, mit einem Advice umgeben. Dieser Advice lässt beim Auftritt eines Fehlers von der Methode `makeError (JoinPoint, String)` ein Fehlerobjekt generieren, das die AspectJ-Reflexion als Informationsquelle nutzen kann (siehe Abbildung 13).

5.4.3 Ausnahmebehandlung. Der Mechanismus zur Propagierung von Ausnahmen, der ein sinnvoller Bestandteil vieler objektorientierter Sprachen ist, stellt für den Unit-Test ein Problem dar.

Tritt eine Ausnahme in einem Testfall auf, mit der auch der Tester nicht gerechnet hat, wird der Testlauf unterbrochen und die Ausnahme weiter gereicht, bis sie schließlich auf höherer Ebene gefangen wird. Es erscheint lohnend, einen Mechanismus zu implementieren, der solche unerwarteten Ausnahmen in einem Testframework automatisch fängt, in die Statistik aufnimmt und dann den Testlauf beim nächsten Testfall fortsetzt.

```
[...]
  public pointcut testCaseCheck(boolean bCondition) :
    call (* TestEntity.check(boolean))
    && within(TestCase+)
    && args(bCondition);

  void around(boolean bCondition) :
    testCaseCheck(bCondition)
  {
    check(bCondition, thisJoinPoint);
  }

  public void check(boolean bCondition, JoinPoint jp)
  {
    if (!bCondition) makeError(jp, "");
  }
[...]
```

Abbildung 12 Ausschnitt aus `TestAspectEntity`

```
[...]
Testfall aufgerufen in TestMatrixDouble44.java: Zeile 165 erzeugte ein fehlerhaftes Ergebnis.
[...]
```

Abbildung 13 Ausgabe eines Fehlers

Der Tester kann das veranlassen, indem er jeden seiner Testfälle 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, dass es sich bei dem Concern „Umgebe jeden Testfall mit einem `try...catch`-Block und melde unerwartete Ausnahmen“ um einen Crosscutting-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.

TestFlex kapselt das Fangen von Ausnahmen, die unerwartet geworfen wurden, zentral in `TestSupervisorAspect` (siehe Abbildung 14). Wie `TestLoggerAspect` kann auch `TestSupervisorAspect` spezialisiert und andere Methoden als Testmethoden identifiziert werden.

5.5 Weitere Anwendungen

Der Fokus dieses Artikels liegt auf dem eigentlichen Unit-Test. Drei weitere AOP-Anwendungen, die zur Unterstützung des Unit-Tests angewandt werden können, sind ausführlich in [28] beschrieben und werden hier nur skizziert, um den Rahmen des Artikels nicht zu sprengen.

5.5.1 Instrumentierung allgemein. In der Testphase können einige Informationen über den laufenden Code für den Tester interessant sein, an die er ohne geeignete Maßnahmen nicht gelangen kann. Solche Informationen sind z.B.: Die Anzahl der Aufrufe einer aufwändigen Funktionalität oder die Zeit, die eine Methode für eine bestimmte Aktion benötigt.

Ein Ausschnitt des Instrumentierungsproblems wurde in Abschnitt 5.1 thematisiert, das Durchbrechen der Kapselung. Hier soll das AOP-Potenzial zur Unterstützung der allgemeinen Instrumentierung thematisiert werden.

Wenn sich die Punkte, an oder zwischen denen instrumentiert werden soll, in der verwendeten Sprache quantifizieren lassen, ist eine quasi von außen wirkende Instrumentierung durch die AOP möglich; der Instrumentierungs-Concern wird in einen Aspekt gekapselt, so dass die Instrumentierungswerkzeuge (Ausgaben, Zähler, etc) nicht-invasiv als eigenständige Code-Einheiten eingesetzt werden können.

5.5.2 Mock Objects. Mock Objects [23] ähneln Rumpfimplementierungen, sogenannten Stubs[6]. Anders als diese beschränken sich Mock Objects nicht nur auf Trivialimplementierungen von bisher nicht fertig gestellten Funktionen. Ein Mock Object muss genügend Funktionalität implementieren, um ein unvollständig implementiertes Server-Objekt zu simulieren, damit die Interaktion des Clients mit dem Server getestet werden kann.

Aspektorientierter Lösung. Die grössten praktischen Schwierigkeiten bei der Implementierung von Mock Objects in einem Testsystem sind:

– Das Mock Object muss zuverlässig im richtigen Kontext für das Originalobjekt eingesetzt werden.

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

```
1  public abstract aspect TestSupervisorAspect
2  {
3    /**
4     * Umgibt alle durch den Namen als testMethoden identifizierten Blöcke mit einem try-catch-Block,
5     * um unerwartete Ausnahmen zu fangen.
6     */
7    pointcut testMethodCall() :
8        execution (void *.test*(..));
9
10   /**
11    * Tritt in einer TestMethode eine unerwartete Ausnahme auf, wird makeUnexpectedException aufgerufen.
12    */
13   void around() : testMethodCall()
14   {
15    // Object obj = null;
16     try
17     {
18       proceed();
19     }
20     catch(Throwable ex)
21     {
22       makeUnexpectedException(thisJoinPoint);
23     }
24   }
25   [...]
```

Abbildung 14 Ausschnitt aus `TestSupervisorAspect`

– Das Mock Object darf nur in der Testphase eingesetzt werden. Für die Release-Version sollte es problemlos zu entfernen sein.

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

5.5.3 Zusicherungen. Zusicherungen (engl. Assertions) [4] definieren Bedingungen, die nicht verletzt werden dürfen, damit sich die Anwendung in einem konsistenten Zustand befindet. Klasseninvarianten sind Zusicherungen, die nach jeder Operation von außen auf das Objekt zu wahr evaluieren müssen. Mit rein objektorientierten Mitteln ist die Implementierung von Klasseninvarianten aufwändig, da sie an vielen Stellen im Code geprüft werden müssen.

Klasseninvarianten mit AOP. Bei den Punkten, an denen die Invariante gelten soll, handelt es sich um Joinpoints und darum eignet sich die AOP, insbesondere AspectJ, für diese Aufgabe, weil es sich beim Überprüfen von einer Bedingung an allen Ein- und Austrittspunkten einer Klasse um einen leicht beschreibbaren Crosscutting-Concern handelt. Außerdem kann zwischen internen und externen Aufrufen unterschieden werden, da die meisten aspektorientierten Sprachen kontrollflussabhängige Definitionen für Pointcuts zulassen.

Zur Integration von Vor- und Nachbedingungen kann analog die gleiche Technik verwendet werden.

5.6 Zusammenfassung

Die Einsatzmöglichkeiten der AOP, die hier erörtert wurden, können grob in drei Gruppen eingeteilt werden:

– Instrumentierung im weitesten Sinne, siehe 5.1, 5.5.1, 5.2 und 5.5.3.
– Testtreiber, siehe 5.3 und 5.5.2.
– Unterstützung für ein Testframework, siehe 5.4.

Dass der Schwerpunkt der Anwendungen im Bereich Instrumentierungen liegt, kann nicht überraschen, da es sich bei der AOP im Grunde um ein sehr flexibles Instrumentierungswerkzeug handelt.

6 Verwandte Arbeiten

Die gute Ergänzung von Testen und Aspektorientierung wurde bereits in anderen Arbeiten festgestellt. Insbesondere die nicht-invasive[7] Integration des Testcodes wird in allen Arbeiten hervorgehoben.

[7] Nicht-invasiv bzgl. des Source-Codes, nicht bzgl. des Verhaltens des Systems.

Fast ausschließlich wird AspectJ als aspektorientierte Sprache beim Testen eingesetzt. Eine Ausnahme ist der in [26] beschriebene Ansatz, der sich auf Object-Teams/Java stützt. Dort wird insbesondere die Fähigkeit der Object Teams genutzt, Aspekte zur Laufzeit aktivieren und deaktivieren zu können, um Zustandsautomaten als Testorakel zu benutzen. Dass eine Implementierung von Zustandsautomaten als Testorakel auch mit AspectJ möglich ist, zeigt [7].

Weitere Anwendungsdomains werden in [8], [10], [20] und [21] beschrieben. Bei diesen Arbeiten liegt der Schwerpunkt auf dem Monitoring des Laufzeitverhaltens, speziell von Realzeit- und nebenläufigen Systemen, während unser Ansatz sich auf den Test einzelner Methoden einer Klasse konzentriert und damit eine andere Granularität besitzt.

Mock Objects durch Aspekte in das zu testende System zu integrieren, wird in [16] vorgeschlagen. Als Basis wird das JUnit-Framework verwendet, das um aspektorientierte Fähigkeiten erweitert wurde. Dieser Ansatz ähnelt sehr stark unserer Idee zur Integration von Mock Objects in das zu testende System.

Zusicherungen mit Hilfe von Aspekten zu realisieren, wird in [25], [6] und [29] vorgeschlagen. In allen Fällen werden Zusicherungen in einer Spezifikationssprache (in [25] und [6] die *Object Constraint Language OCL* als Teil der UML, in [29] die *Java Modeling Language JML*) notiert, die automatisch ausgewertet werden. In [29] wird dazu ein Generator benutzt, der JML-Spezifikationen in Java-Code transformiert und diese als Aspekte in das zu testende System einbindet. Das in [29] vorgeschlagene aspektorientierte Framework auf der Basis des JUnit-Frameworks wertet zudem in einer aspektorientierten Testbeschreibungssprache definierte Testfälle aus und generiert aus dieser Beschreibung den AspectJ-Code. Unser Ansatz benutzt keine unabhängige Spezifikations- oder Testbeschreibungssprache, Zusicherungen und Testfälle werden in Advices spezifiziert. Dieses bietet den Vorteil, dass der Tester neben AspectJ keine weitere Sprache erlernen muss, was jedoch mit der geringen Mächtigkeit der Sprache im Gegensatz zu Spezifikationssprachen wie OCL oder JML erkauft wird.

Die oben vorgestellten Arbeiten verfolgen im Gegensatz zu unserer Arbeit jeweils die Untersuchung eines Teilaspekts beim Testen und seine Unterstützung durch aspektorientierte Programmiertechniken. Unsere Arbeit versucht eine umfassende Untersuchung der Unterstützung aspektorientierter Programmiertechniken im Testen zu leisten, konzentriert sich dabei aber auf das Unit-Testen.

7 Fazit

Die aspektorientierte Programmierung liefert dem objektorientierten Unit-Test eine Reihe von neuen Lösungsansätzen. Ob sie sich für ein bestimmtes Problem eignen, muss abgewogen werden.

Zum Abschluss werden hier noch einmal die Vorteile zusammengefasst und den Nachteilen gegenübergestellt. Der Ausblick befasst sich schließlich damit, wo zu dem Thema noch Beiträge geleistet werden können.

7.1 Vorteile des FlexTest-Frameworks gegenüber konventionellen Frameworks

In den folgenden Punkten werden einige Vorteile des aspektorientierten Unit-Testansatzes, wie er in Flextest realisiert worden ist, aufgeführt.

7.1.1 Komfort. AspectJ verfügt über mächtigere Reflexionsfähigkeiten als die Java-Reflection-Classes. Im FlexTest-Framework werden diese Fähigkeiten dazu ausgenutzt, Fehlerbeschreibungen automatisch zu generieren. Der Tester kann sich darauf beschränken, seine Testcases schreiben. Für die Strings zur Fehlerbeschreibung ist er nicht zuständig.

Flexibles Logging ist für ein Testframework eine Selbstverständlichkeit. FlexTest nutzt zum Loggen die Möglichkeiten der AOP, weshalb sich der Client-Teil des Logging-Vorgangs in einem gekapselten Aspekt mit einer allgemeinen Regel für das Loggen befindet. Im konventionellen Logging verstreut sich der Logging-Code über die gesamte Implementierung.

7.1.2 Zeit- und Platz-Ersparnis. FlexTest macht es dem Tester 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 und die Tests auf Liskov-Konformität mitliefern, Testarbeit abnehmen.

7.1.3 Überwindung klassischer Testprobleme. Klassische Testframeworks bieten dem Tester keine Hilfsmittel zur Überwindung von Problemen, die beim Test von objektorientierten Programmen auftreten. JUnit [12] und Boost [5] 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.

FlexTest 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.

7.1.4 Unkomplizierte Implementierung von Mock Objects. Überraschenderweise hat sich als Nebenergebnis bei der Implementierung des Testframeworks gezeigt, dass Aspekte selbst sich als Mock Objects eignen.

Das Mock Object-Konzept wurde zwar nicht als fester Bestandteil in das Framework integriert. Es ist aber günstig, Mock Objects von einer Klasse des Frameworks abzuleiten. Die Restarbeit bei der Programmierung des Mock Objects besteht nur noch darin, die zu simulierenden Methoden mit Pointcuts auszuwählen, mit Advices zu simulieren und Überprüfungsmechanismus zu schreiben.

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

AOP-Sprachen wie z.B. AspectJ können diese objektübergreifenden Aufgaben modular und zentral lösen.

7.2 Kritik des Ansatzes

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

7.2.1 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.

7.2.2 Konflikte mit System- und Komponententest. In der Entwicklung hat man es immer mit zwei Varianten des kompilierten Codes oder des Byte-Codes zu tun: Einer Release-Variante und einer Variante für den Unit-Test. Unter Umständen kann das zu Konflikten mit System- und Komponententests führen, die die Zusammenarbeit unveränderter Release-Einheiten testen sollen.

7.2.3 Einarbeitung. Der aspektorientierte Ansatz unterscheidet sich stark vom imperativen oder objektorientierten Stil. Selbst wenn man alle Konzepte verstanden 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.

7.3 Wertung

Die am schwersten wiegenden Nachteile von FlexTest werden durch wachsende Erfahrung der Entwickler und Tester kompensiert. Einzig der etwas höhere Kompilieraufwand im Vergleich zu Instrumentierungs-Tools bleibt über längere Dauer als Kritikpunkt bestehen. Rechnet man aber die Zeitersparnis für die Tester in der Programmierung der Testfälle gegen, verliert auch dieser Kritikpunkt an Bedeutung.

Die oben erwähnten Vorteile des aspektorientierten Ansatzes sollten überzeugend genug sein, sich weiter gehend mit dem Thema zu beschäftigen.

7.4 Ausblick

Das FlexTest-Framework ist eines der ersten Frameworks seiner Art und schöpft darum natürlich noch nicht das gesamte Potenzial der aspektorientierten Programmierung auf dem Gebiet Unit-Testen aus.

Einige Ideen zur Verbesserung und Erweiterung sollen hier zum Abschluss noch präsentiert werden.

7.4.1 Makro-Sprache. Wenn die AOP-Erweiterungen nur für kleinere Aufgaben angewandt werden, möchte man sich als Tester 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 könnte den aspektorientierten Teil mit einigen kurzen Makros im konventionellen Testcode erledigen.

Auch im Hinblick auf die Sicherheit wäre eine Makro-Sprache vorteilhaft, da durch sie garantiert würde, dass die aspektorientierten Teile des Frameworks nicht falsch eingesetzt werden.

7.4.2 Testen von aspektorientiertem Code. Das Unit-Testen von aspektorientiertem Code ist ein Thema, das auch in der Forschung noch nicht befriedigend gelöst wurde.

Es gibt einige Arbeiten zu dem Thema wie z.B. [30], wo der Versuch unternommen wird, Einheiten als Units für den Unit-Test zu isolieren. Von einer standardisierten Methode kann noch nicht die Rede sein.

Eine Erweiterung des Frameworks, die es befähigen würde, aspektorientierten Code zu testen, würde sich anbieten.

Literatur

1. AspectJ-Homepage. htpp://www.aspectj.org.
2. K. Beck and E. Gamma. Test Infected: Programmers Love Writing Test. http://members.pingnet.ch/gamma/html/junit.htm.

3. R. V. Binder. Object-oriented software testing. *Communications of the ACM*, 37(9), 1994.
4. R. V. Binder. *Testing Object-Oriented Systems.* Object Technology Series. Addison-Wesley, 2000.
5. BOOST-Testframework-Homepage. http://www.boost.org.
6. L. C. Briand, W. Dzidek, and Y. Labiche. Using Aspect-Oriented Programming to Instrument OCL Contracts in Java. Technical report, Carlton University, Kanada, 2004.
7. J.-M. Bruel, J. Araújo, A. Moreira, and A. Royer. Using Aspects to Develop Built-In Tests for Components. In *AOSD Modeling with UML Workshop, 6th UML Conference*, San Francisco, USA, 2003.
8. M. Deters and R. K. Cytron. Introduction of Program Instrumentation using Aspects. In *Workshop of Advanced Separation of Concerns in Object-Oriented Systems, Proc. of the 16th OOPSLA*, ACM Sigplan Notices, Tampa, USA, 2001.
9. R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, 19th OOPSLA*, Minneapolis, USA, 2000.
10. R. E. Filman and K. Havelund. Source-Code Instrumentation and Quantification of Events. In *Workshop on Foundations of Aspect-Oriented Languages, 1st AOSD Conference*, Enschede, Niederlande, 2002.
11. R. E. Jeffries. Extreme Testing: Why Aggressive Software Development Calls for Radical Testing Efforts. *Software Testing & Quality Engineering*, 1999.
12. Junit-homepage. http://www.junit.org.
13. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of the 15th ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, Budapest, Ungarn, 2001. Springer-Verlag.
14. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of the 11th ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, Jyväskylä, Finnland, 1997. Springer-Verlag.
15. R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications, 2003.
16. N. Lesiecki. Test Flexibility with AspectJ and Mock Objects. http://www-106.ibm.com/developerworks/java/library/j-aspectj2/.
17. P. Liggesmeyer. Ein Überblick über objektorientiertes Testen. In *8. Treffen des Arbeitskreises TAV*, volume 15 of *Softwaretechnik-Trends*, Hamburg, 1995. Gesellschaft für Informatik (GI).
18. P. Liggesmeyer and A. Spillner, editors. *Themenheft: Aktuelle Entwicklungen im Softwaretest*, volume 15 of *Informatik - Forschung und Entwicklung.* Springer-Verlag, September 2000.
19. B. Liskov. Data Abstraction and Hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications*, pages 17 – 34, Orlando, USA, 1987.
20. T. Low. Designing, Modelling and Implementing a Toolkit for Aspect-oriented Tracing (TAST). In *Workshop on Aspect-Oriented Modeling with UML, 1st AOSD Conference*, Enschede, Niederlande, 2002.
21. D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Proc. of The 5th ISORC*, Crystal City, USA, 2002.
22. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.
23. MockObjects.com. http://www.mockobjects.com.
24. J. Overbeck. Testing Object-Oriented Software: State of the Art and Research Directions. In *Proc. of the 1st EuroStar*, London, Großbritannien, 1993.
25. M. Richters and M. Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In *AOSD Modeling With UML Workshop, 6th UML Conference*, San Francisco, USA, 2003.
26. D. Sokenou and S. Herrmann. Using Object Teams for State-Based Class Testing. Technical report, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, 2004.
27. C. D. Turner and D. J. Robson. The Testing of Object-Oriented Programs. Technical report, University of Durham, Department of Computer Science, 1992.
28. M. Vösgen. Implementierung eines aspektorientierten Frameworks für den objektorientierten Unit-Test. Diplomarbeit, Technische Universität Berlin, 2004.
29. G. Xu, Z. Yang, and H. Huang. A Basic Model for Aspect-Oriented Uni Testing. www.cs.ecnu.edu.cn/sel/ harryxu/research/ papers/fates04_aspect-oriented
30. J. Zhao. Unit Testing for Aspect-Oriented Programs. Technical Report SE-141-6, Information Processing Society of Japan, 2003.