Empirische Untersuchungen von aspektorientiertem Programmieren

Benjamin Schröter (schroete@inf.fu-berlin.de)

Einleitung

In der Informatik wurden bisher verschiedene Methoden verwendet, um Computerprogramme zu schreiben. Von einfachem Assembler über funktionale und prozedurale Systeme, über die Verwendung von abstrakten Datentypen bis hin zur Objektorientierung. Mit jedem dieser Systeme wurde es einfacher, die Aufgaben des Systems in einzelnen Modulen klar voneinander zu trennen (*seperation of concerns*) [1]. Das aspektorientierte Programmieren (*AOP*) erweitert die Möglichkeit dieser Trennung um Stellen die bisher in objektorientierten Systemen nicht vorhanden waren. Dadurch kann erreicht werden, dass wir die Modularität eines Programms so aufbauen können, wie wir es gewohnt sind darüber nachzudenken und nicht darauf angewiesen sind, es so zu machen, wie es uns die verwendeten Werkzeuge vorschreiben. [2, 3]

Durch diese verbesserte Modularität verspricht die aspektorientierte Programmierung einfacheren Code. Auch die Wiederverwendung dieses Codes soll dadurch gefördert werden. Als Konsequenz daraus erhofft man sich eine gesteigerte Produktivität bei der Softwareentwicklung. Allgemein wird dies aufgrund der Modularität und des vereinfachten Codes auch angenommen. Doch ob der Code wirklich einfacher wird und vor allem ob sich daraus tatsächlich Produktivitätssteigerungen ergeben kann nur schwer überprüft werden. Um diese Fragen zu beantworten haben Walker, Baniassad und Murphy eine Untersuchung durchgeführt, die hier beschrieben wird.

In den folgenden Abschnitten werde ich zuerst die Grundgedanken von AOP und AspectJ (eine aspektorientiere Erweiterung für Java) vorstellen. Der dritte Abschnitt befasst sich dann mit der Untersuchung von Walker, Baniassad und Murphy. Danach (Abschnitt 4) werden zwei weitere Untersuchungen beschrieben, die zu ähnlichen Ergebnissen kommen. Der letzte (fünfte) Abschnitt fasst diese Ergebnisse zusammen.

1 Aspektorientieres Programmieren

Heutzutage ist Objektorientierung eins der wichtigsten Prinzipien in der Softwareentwicklung. Es gibt spezielle Analyse- (OOA) und Entwurfsmethoden (OOD) die helfen sollen, Software mit einem objektorientierten Blickwinkel zu analysieren bzw. zu entwerfen. Des Weiteren existieren objektorientierte Notationen mit entsprechender Softwareunterstützung für den Entwurf (z.B. UML), sowie Programmiersprachen um objektorientiert zu Entwickeln. Die Dominanz objektorientierter Programmiersprachen wie z.B. Java, C++ oder C# könnte den Schluss zulassen, dass die objektorientierte Programmierung (*OOP*) ausgereift und nicht verbesserungsfähig sei. Doch es gibt durchaus Situationen und Anforderungen, in denen OOP Schwächen zeigt und das Problem auch auf andere Weise betrachtet werden könnte. An diesen Stellen kann die aspektorientierte Programmierung möglicherweise weiterhelfen.

Ein Ziel von OOP ist es, die Zuständigkeiten von Objekten klar zu trennen (separation of concern). Diese Trennung gliedert das zu erstellende Programm in klare Einheiten und ermöglicht so eine einfache Entwicklung, Weiterentwicklung und Fehlersuche und stellt die Grundlage für den Austausch und die Wiederverwendung von Modulen dar.
OOP eignet sich zwar hervorragend um Anwendungen nahe an (real existierenden) Objekten zu modellieren, und gerade aus diesem Grund erfreut sich OOP einer großen Beliebtheit, doch sobald sich einzelne Anforderungen nicht mehr klar den modellierten Objekten zuordnen lassen, erreicht man die Grenzen von OOP.

Typische Anforderungen an die Performance, Speicherverwaltung, Fehlerbehandlung, Logging und Sicherheit lassen sich in einem solchen Objektmodell meist nicht ohne weiteres von den fachlichen Anforderungen trennen. Solche Anforderungen können zwar als einzelne Klassen modelliert werden, aber diese Klassen weisen meist einen anderen Detaillierungs- und Abstraktionsgrad als die fachlichen Klassen auf. Außerdem ist damit meist nur ein Teil der Anforderung in einer Klasse gekapselt doch ein weiterer Teil zieht sich durch viele andere Klassen des Programms.
Als Beispiel kann man hier das Logging von gewissen Methodenaufrufen nennen. Zwar kann man das Formatieren, Ausgeben und Speichern von Informationen in einer Klasse vom Rest der Anwendung trennen, aber die Methoden, die protokolliert werden

sollen, müssen Methoden der Logging-Klasse weiterhin explizit aufrufen.
Als Resultat wird der Code vieler Methoden, die eigentlich ganz unterschiedliche Aufgaben erfüllen, mit Code zum Logging vermischt (*tangled code*) oder anders betrachtet ist der Code für das Logging über viele Klassen und Methoden verteilt.
Daraus entstehen natürlich Schwierigkeiten beim Debugging, wenn sich die Anforderungen ändern und eine Wiederverwendung einzelner Klassen in einem anderen (Logging-)Kontext ist so gut wie unmöglich.

Solche Anforderungen, die nicht als Objekte entworfen werden können, werden als *Aspekte* bezeichnet.
Die aspektorientierte Programmierung ermöglicht es, diese Anforderungen als Aspekte zu modellieren. Die Aspekte enthalten dann allen Code der zur Erfüllung der Anforderungen nötig ist und verteilen ihn nicht auf viele Klassen. Dadurch wird die Zuständigkeit von Objekten und Aspekten wieder klar deutlich und das Entwickeln und Debuggen wird erleichtert. Auch die Wiederverwendung sowohl auf Objekt- als auch Aspektebene wird dadurch gefördert.

Aspekt- und Objektcode werden beim compilieren oder zur Laufzeit miteinander *verwoben* und erzeugen so das gewünschte Verhalten.
Für viele Programmiersprachen gibt es erste aspektorientierte Erweiterungen und auch erste Unterstürzung in den Entwicklungswerkzeugen. Diese Erweiterungen arbeiten auf verschiedenen Ebenen, so dass sie entweder Quellcode generieren oder eigene Compiler bzw. Laufzeitbibliotheken anbieten.
Im folgenden Kapitel wird die sehr bekannte und stabile Erweiterung AcpectJ für Java vorgestellt.

2 AspectJ

Ursprünglich wurde *AspectJ* [4] 1997 vom *Xerox Palo Alto Research Center* als beispielhafte Implementierung von Aspektorientierung in Java entwickelt und ist nun Teil des *Eclipse*-Projekts [5]. Die ersten Versionen von AspectJ waren dabei spezielle Aspektsprachen um spezielle Probleme zu lösen, entwickelten sich dann aber zu einer allgemeinen Sprache mit der man nahezu beliebige Aspekte in Java definieren kann. Die aktuelle Version ist 1.1.1.
AspectJ erweitert Java um aspektorientierte Elemente und kann heute als quasi Standard für AOP unter Java angesehen werden. Die hier beschriebenen Experimente und Studien verwenden im Java Umfeld ebenfalls AspectJ.
Viele Entwicklungsumgebungen (u.a. Eclipse, Emacs, JBuilder) unterstützen AspectJ bereits oder es werden entsprechende Plug-Ins angeboten.

AspectJ stellt einen eigenen Compiler zur Verfügung der die eigentlichen Java-Klassen und die Aspekte verarbeitet, beim compilieren miteinander verwebt und dann Java-Bytecode erzeugt.
Dabei ist AspectJ in folgender Form kompatibel zu Java [6]:

- *Upward compatibility*: alle syntaktisch korrekten Java Programme sind auch syntaktisch korrekte AspectJ Programme.
- *Platform compatibility*: AspectJ Programme laufen auf einer normalen (d.h. unveränderten) Java virtual machine, da der Compiler Bytecode erzeugt.
- *Tool compatibility*: vorhandene Tools (z.B. IDEs, Dokumentationstools, Designtools) können ohne große Schwierigkeiten an AspectJ angepasst werden.
- *Programmer compatibility*: AspectJ ist so in Java integriert, dass der Programmierer es als natürliche Erweiterung betrachtet.

Durch AspectJ werden Java neue Sprachkonstrukte hinzugefügt, die es ermöglichen Aspekte und deren Verhalten zu beschreiben.

Ein Aspekt besteht aus einem oder mehreren *Join Points* (Schlüsselwort `pointcut`) die angeben, an welchen Stellen der Aspektcode mit dem Java-Objektcode verwoben werden soll. Join Points können z.B. folgende sein: alle setter-Methoden, alle Konstruktoraufrufe, Konstruktoraufrufe bestimmter Klassen oder gewisse Methoden, die einem angegebenen Muster entsprechen.

Des Weiteren enthält ein Aspekt einen oder mehrere *Advices*, die Javacode enthalten, der an den Join Points eingewoben werden soll. Dabei stehen die folgenden drei Advices zur Verfügung:

- `before`: der Code wird vor dem Erreichen des Join Points ausgeführt
- `after`: der Code wird nach dem Erreichen des Join Points ausgeführt
- `around`: der Code wird anstatt des Join Points (beispielsweise anstatt einer bestimmten Methode) ausgeführt. Innerhalb des Advices kann dann z.B. die Methode ausgeführt oder eben dies verhindert werden.

Der Compiler verbindet (*verwebt*) den Code der Advices an den definierten Join Points so das der Aspektcode an diesen Stellen automatisch mit dem Objektcode ausgeführt wird. Dabei hat man auch die Möglichkeit, Parameter, Rückgabewerte oder sogar das Verhalten von den als Join Point definierten Methoden zu beeinflussen.

3 Untersuchung von Walker, Baniassad und Murphy

AOP verspricht, unter anderem durch die Möglichkeit miteinander verflochtenen (*tangled*) Code zu vermeiden, Module (in diesem Fall Klassen und Aspekte) besser voneinander trennen zu können, so das Software einfacher und übersichtlicher wird. Dadurch erhofft man sich auch Produktivitätsvorteile bei der Anpassung und der Fehlersuche in Anwendungen.

Walker, Baniassad und Murphy haben, um diese Annahme zu stützen, ein kontrolliertes Experiment durchgeführt und die Ergebnisse 1999 in [7] unter dem Titel „*An Initial Assessment of Aspect-oriented Programming*“ veröffentlicht. Dabei hatte das Experiment einen eher erkundenden Charakter um die Grundlage für weitere Untersuchungen auf diesem sehr neuem Forschungsgebiet zu bilden.

Der nächte Abschnitt beschreibt das durchgeführte Experiment und im darauf folgendem werden die Ergebnisse dargestellt.

3.1 Durchführung

Es wurden zwei voneinander getrennte Experimente durchgeführt, die zwar ähnlich aufgebaut waren, aber unterschiedliche Situationen simulierten. Zum einen wurde das Debuggen einer bestehenden Anwendung, zum anderen das Verändern dieser Anwendung untersucht.

Als Grundlage diente in beiden Experimenten ein Bibliothekssystem in dem es möglich war, Bücher auszuleihen und Anfragen von Benutzern an die Bibliothek, sowie von Bibliotheken an andere Bibliotheken zu senden.
Dieses System stand in einer AspectJ und einer Java Varianten für das erste Experiment zur Verfügung.
Für das zweite Experiment wurde eine verteilte Version dieses Bibliothekssystems verwendet, die zum einen in AspectJ und zum anderen in Emerald zur Verfügung stand. Hier wurde Emerald als Kontrollsprache verwendet, da Emerald eine objektorientierte Sprache ist, die von Haus aus verteilte Anwendungen und Synchronisation unterstützt.

Verwendet wurde die Version 0.1 von AspectJ, eine Version die mit der aktuellen nicht mehr viel gemeinsam hat. In dieser Version waren nur zwei Aspektsprachen (`Cool` und `Ridl`) implementiert und es war noch nicht möglich beliebige Aspekte zu schreiben, so wie es oben beschrieben ist. Dies schränkt zwar den Nutzen von AspectJ im Vergleich zur aktuellen Version sehr ein, erlaubt es aber durchaus erste Erfahrungen mit aspektorientierter Programmierung zu sammeln. Die hier vorgestellten Experimente sind so angelegt, dass sie genau in den Aufgabenbereich dieser beiden Aspektsprachen fallen:

- `Cool` kapselt die Synchronisierung von Methoden, indem es ermöglicht, die nötigen Sperrungen zu definieren.
- `Ridl` ermöglicht es bei Remoteaufrufen von Methoden das Transferverhalten von Objekten (Parametern und Rückgabewerten) zu regeln. Es gibt an, welche Parameter kopiert und welche als Remotereferenz verwendet werden sollen. Dabei bietet `Ridl` die Möglichkeit, dieses Verhalten nicht nur für Objekte, sondern auch für deren Daten (*properties*) zu definieren. D.h. einzelne Properties eines Objekts können als Kopie übertragen werden und andere als Remotereferenz.

Die Programmierer, die an dem Experiment teilnahmen wurden als Paare zusammengestellt. Drei Paare arbeiteten mit AspectJ und drei andere mit der Kontrollsprache. Als Kontrollsprache wurde Java (im Debugexperiment) bzw. Emerald (im Changeexperiment) eingesetzt.

Während des Experiments wurden die Programmierer gebeten laut zu denken und zur späteren Auswertung auf Video aufgezeichnet. Zusätzlich wurden sie während des Programmierens und danach interviewt.
Jedes der beiden Experimente dauerte maximal 90 Minuten. Die Zeit zum Erfüllen der einzelnen Aufgaben wurde gemessen.

Im Vorfeld wurden die Teilnehmer mit den Programmiersprachen und Entwicklungstools vertraut gemacht. Außerdem wurde ihnen eine kurze Übersicht der Themen Synchronisierung und verteilte Anwendungen zur Auffrischung ihres Wissens gegeben.

3.1.1 Debugexperiment

Im ersten Experiment sollten die Programmierer Fehler im gegebenen Code finden und korrigieren. Dazu wurde der Code mit drei Fehlern versehen, die alle die Synchronisierung betrafen:

1. lesende Anfragen verschiedener Benutzer wurden sequenziell abgearbeitet
2. zwei Benutzer konnten das selbe Buch gleichzeitig ausleihen
3. es traten Deadlocks auf

Die aspektorientierte Implementierung mit AspectJ verwendet in diesem Beispiel die `Cool`-Aspektsprache um die Synchronisierung zu realisieren.

Während dem Experiment wurden Messungen durchgeführt und Beobachtungen festgehalten, die nun im einzelnen beschrieben werden:

Allen Teilnehmern (sowohl den Java- als auch den AspectJ-Gruppen) gelang es innerhalb der gegebenen Zeit alle Fehler zu finden und zu beheben. Die AspectJ-Gruppen (22, 34 und 35 Minuten) benötigten in der Summe weniger Zeit als die Java-Gruppen (36, 45 und 61 Minuten) um alle Fehler zu beheben. Deutlicher wurden die Unterschiede, wenn man die Fehler einzeln betrachtet. Den ersten Fehler konnten alle AspectJ-Gruppen viel schneller korrigieren als die Vergleichsgruppen. Bei den anderen beiden Fehlern fällt die Differenz geringer aus.
Des Weiteren wurde gemessen, wie oft die Entwickler zwischen den betrachteten Dateien wechseln mussten. Dies wird als Indiz dafür genommen, wie gut die Zuständigkeiten einzelner Module (Dateien) getrennt sind bzw. wie sehr die betrachteten Probleme (hier die Synchronisierung) über mehrere Dateien verteilt sind. Bei der Korrektur des ersten Fehlers wechselten die AspectJ-Gruppen seltener die Dateien, beim zweiten Fehler öfter und beim dritten in etwa genauso oft wie die Java-Gruppen.
Es wurde versucht zu messen, wieviele semantische Analysen die einzelnen Gruppen bei der Bearbeitung der einzelnen Fehler durchführten. Dazu wurde vor allem die Kommunikation der Paare bei der Arbeit betrachtet. Aus den daraus gewonnenen Daten wird sehr schnell deutlich, dass die AspectJ-Gruppen viel seltener das Verhalten von Objektcode analysieren mussten um die Fehler innerhalb der Aspekte zu korrigieren.

Die Entwickler, die mit Java arbeiteten, fügten den Code zur Synchronisierung innerhalb der Methoden ein und konnten in einer sehr feinen Granularität das gewünschte Verhalten erzeugen. Um mit AspectJ eine solch feine Granularität zu erreichen hätten die Entwickler die Struktur des Programms (der Methoden) verändern müssen, da AspectJ nur das einweben von Aspektcode auf Methodenebene (vor oder nach der Methode) erlaubt. Keins der AspectJ-Teams hat den Code so verändert. Außerdem schien keins der AspectJ-Teams die Granularität der `Cool`-Aspektsprache in Frage zu stellen.

Die drei Aufgaben, die an die Programmierer gestellt wurden, können zwei Gruppen zugeordnet werden: lokale (*localized*) und nicht-lokale (*non-localized*) Fehler. Fehler, die an einer Stelle behoben werden können (z.B. in einer Klasse oder einem Aspekt) werden als lokal bezeichnet. Nicht-lokal hingegen sind die Fehler, zu deren Korrektur die Entwickler an mehreren Stellen arbeiten müssen.
In diesem Experiment fällt auf, dass die AspectJ-Gruppen immer dort einen deutlichen Vorteil hatten, wenn das zu behebende Problem lokal ist. Bei nicht-lokalen Problemen hingegen ist dieser Vorteil nicht so klar deutlich, aber es scheint sich hier auch kein Nachteil durch die Verwendung von AspectJ zu ergeben.

3.1.2 Changeexperiment

In diesem Experiment wurde die Auswirkung von AspectJ auf die Veränderbarkeit von Programmen untersucht. Dazu sollten die Programmierer drei Änderungen an der Anwendung vornehmen. Auch hier wurde das oben beschriebenen Bibliothekssystem in der verteilten (*distributed*) Version verwendet. Die AspectJ-Variante verwendet die Aspektsprache `Cool` für die Synchronisierung und `Ridl` um den Datentransfer der Remoteaufrufe zu steuern. Als Kontrollsprache wurde Emerald verwendet.

Als erstes sollten die Programmierer eine Funktion zum Zurückgeben ausgeliehener Bücher hinzufügen. Als zweite Aufgabe sollte eine Bibliothek das Ausleihen an Benutzer zufällig verbieten. Dazu muss automatisch klargestellt werden, welche Bibliothek im System diese Aufgabe übernimmt und dass alle Bibliotheken vor dem Ausleihen diese befragen.
Die dritte Aufgabe bestand darin, die Performanz zu verbessern. Dazu konnten die Programmierer an jeder Stelle des Programms Verbesserungen vornehmen, aber es wurde auch bewusst an einer Stelle im ursprünglichen Code mit ineffizientem Code gearbeitet.

Auch hier werden nun die Ergebnisse der einzelnen Messungen und Beobachtungen beschrieben:
Da die dritte Aufgabe beliebig viel Zeit in Anspruch nehmen kann, wurde nur die Zeit für die Bearbeitung der ersten beiden Aufgaben miteinander verglichen. Dabei benötigte die AspectJ-Gruppe diesmal mehr Zeit als die Vergleichsgruppe, die Emerald verwendet. Um dies genauer betrachten zu können, wurde untersucht, für welche Tätigkeiten die Zeit genutzt wurde. Dabei wurde deutlich, dass die Emerald-Gruppe mehr Zeit für die Analyse benötigte als die AspectJ-Gruppe, wohingegen die AspectJ-Gruppe mehr Zeit für das Coding benötigte.
Es wurde auch betrachtet wie viel und wo die beiden Gruppen Code geschrieben haben. Die AspectJ-Gruppen schrieben zwischen 50 und 150 Zeilen Code von denen $\frac{1}{3}$ `Ridl`-Code und $\frac{2}{3}$ `Cool`-Code waren. Die Emerald-Gruppe schrieb zwischen 50 und 80 Zeilen Code von denen die Hälfte Code für die Synchronisierung und $\frac{1}{3}$ Code für das Transportieren von Objekten war.

Für Performanzverbesserungen, wie sie in der dritten Aufgabe vorgesehen war, hatten nur zwei der drei AspectJ-Programmierer noch genügend Zeit (auch dieses Experiment war auf 90 Minuten begrenzt), und nur einer von ihnen konnte tatsächlich Performanzverbesserungen erreichen. Im Gegensatz dazu haben alle Emerald-Programmierer Performanzverbesserungen in dem Programm durchführen können. Dies lag unter anderem auch daran, dass `Ridl`

in Version 0.1 verschiedenen Probleme mit sich brachte, da es auf Java RMI aufsetzt. Dadurch müssen Objekte, die kopiert werden sollen, das `Serializable`-Interface implementieren. Dazu hätten große Teile des Codes neu geschrieben oder zumindest überarbeitet werden müssen.

Durch `Ridl` schien das Kopieren von Objekten bei Remoteaufrufen genauso gekapselt zu sein, wie `Cool` die Synchronisierung kapselt. Daher begannen die AspectJ-Programmierer meist sehr schnell mit dem Programmieren, ohne vorher den Objeketcode genauer zu analysieren. Dies führte jedoch nicht zum Ziel und sie mussten durchaus semantische Analysen durchführen, da das Zusammenspiel des Javacodes mit dem Aspektcode an dieser Stelle von Bedeutung ist.
Die `Ridl`-Aspektsprache scheint den Code nicht nur in die Aspekte zu kapseln, sondern ist durchaus sehr vernetzt mit dem Anwendungscode.

3.2 Experimentkritik

Obwohl das Experiment nur erkundenden Charakter hatte und nicht darauf ausgerichtet war, konkrete Zahlen und Faktoren zu ermitteln, gibt es einige Kritikpunkte an dem Experiment. Diese Kritikpunkte sollen nicht das Ergebnis schmäler, denn die erkundende Aufgabe kann und hat das Experiment so wie es durchgeführt wurde durchaus erfüllt, aber an mehreren Stellen sind die Ergebnisse sehr vage. Die Ergebnisse hätten mit einem leicht veränderten Experimentaufbau vielleicht konkreter ausfallen können.

Das Experiment war nicht darauf ausgelegt verallgemeinerbar zu sein. Dazu waren zu wenige Versuchspersonen daran beteiligt. Auch ist es fragwürdig, ob die Aussagen über die recht kleinen Aufgaben, die an die Programmierer gestellt wurden, auf größere Aufgaben und Projekte übertragen werden können. Aber natürlich konnten trotz dieser Einschränkungen Beobachtungen gemacht und Tendenzen festgestellt werden. Dies war die Hauptaufgabe dieses Experimentaufbaus.

Weitaus komplizierter scheint es mir, die Ergebnisse des Changeexperiments zu verallgemeinern. Hier wurden zwei von Grund auf verschiedene Programmiersprachen verwendet, und es ist nicht klar, ob vielleicht Emerald an sich schon große Vorteile hatte und ein Vergleich mit aspektorientiertem Java (AspectJ) gar nicht legitim ist.
Dieser Einwand scheint mir durchaus berechtigt und wird von Walter auch nicht weiter betrachtet. Es wird lediglich angeführt, dass Emerald hier Eigenschaften besitzt, die Java fehlen und die Java durch AspectJ hinzugefügt werden. Doch dies ist gerade der Punkt, der untersucht werden sollte. Auch wenn Emerald nicht aspektorientiert arbeitet bringt es für diesen Aufgabenbereich Funktionen mit sich, die AspectJ Java hinzufügen soll.
Dieser Vergleich scheint mir sehr Fragwürdig, so dass zumindest die quantitativen Ergebnisse aus diesem Experiment nicht zu hoch bewertet werden sollten. Natürlich wurden hier auch allgemeine Beobachtungen zu AOP gemacht, die durchaus eine weitergehende Untersuchung bedürfen.

3.3 Ergebnisse

Auch wenn die beiden Experimente recht verschiedene Ergebnisse lieferten, bemerkt man doch, dass das Aspekt-Interface eine große Rolle dabei spielt, ob aspektorientierte Programmierung Vorteile gegenüber der objektorientierten Programmierung bringt oder nicht.
Auch wenn man die Konzepte in verschiedene Teile des Codes trennen kann, bedeutet dies nicht, das diese Codeteile unabhängig voneinander sind. Diese Codeteile müssen immer zusammenarbeiten um die Funktion des Systems zu erzeugen [7, Seite 126].

Es wurden große Unterschiede zwischen `Cool` und `Ridl` aufgedeckt. Die Teilnehmer an dem Experiment sahen in `Cool` eine klar abgegrenzte Aufgabe für den Java-Objektcode. Dies wird in der Studie als „*narrow aspect-code interface*“ bezeichnet und erlaubt es den Programmierern den `Cool`-Code zu verstehen, ohne den Java-Objektcode analysieren zu müssen.

Im Gegensatz dazu ist `Ridl` komplexer und das Interface wird als „*wide aspect-code interface*“ bezeichnet. Um den `Ridl`-Code verstehen und modifizieren zu können muss nicht nur dieser verstanden, sondern auch der Java-Objektcode analysiert werden. Dadurch gehen die Vorteile, die AOP verspricht, nämlich durch die Trennung von Objekten und Aspekten den Code zu vereinfachen, verloren.
Die Probleme, die die Programmierer mit `Ridl` hatten, könnten vielleicht verringert werden, wenn das Aspekt-Interface anders entworfen wäre. Auch würden entsprechende Werkzeuge unter Umständen das Arbeiten mit solchen Aspekt-Interfaces erleichtern.

Als wichtigstes Ergebnis muss daher genannt werden, dass durch AOP die versprochenen Vorteile nur dann Wirkung zeigen, wenn die Aspekte und das Aspektinterface klare Aufgaben haben und von dem übrigen Code klar getrennt sind. Aspektorientierung erleichtert zwar diese Trennung, aber reicht als alleiniges Mittel nicht dazu aus. Zum einen muss das Aspektinterface klar durchdacht sein und zum anderen muss geprüft werden ob und wie sich ein Vorhaben überhaupt durch Aspekte umsetzen lässt.
Leider lassen Walker et al. die Frage offen, ob ein anderes Interface für `Ridl` oder sogar ein anderer

Ansatz für diesen Aspekt (so wie man ihn erst heute mit einer neuen AspektJ Version entwickeln könnte) bessere Ergebnisse erreicht hätten, oder ob das Vorhaben des Changeexperiments tatsächlich mit objektorientierten Methoden (so wie in der Vergleichsgruppe) besser zu realisieren ist.

Des Weiteren scheint es, dass aspektorientierte Programmierung auch das Herangehen der Programmierer an die Probleme verändert. Sie versuchen, wenn Aspekte vorhanden sind, die den Bereich der zu untersuchenden Aufgaben betreffen, dort die Probleme zu lösen. Beim ersten Experiment war genau dies die richtige Strategie, aber im zweiten Experiment haben die meisten angefangen die `Ridl`-Aspekte zu bearbeiten, bevor die Programmierer den gesamten Code analysiert und verstanden haben, obwohl es vielleicht besser gewesen wäre, nicht den Aspektcode sonder den Objektcode zu verändern.

4 Weitere empirische Untersuchungen

Walker et al. haben in der oben beschriebenen Studie [7] schon zu einem frühen Zeitpunkt der AOP-Entwicklung interessante Beobachtungen gemacht. Weitere Untersuchungen und Studien bekräftigen diese Ergebnisse zum Teil.
In den folgenden Abschnitten sollen zwei dieser Studien kurz vorgestellt werden.

4.1 „An Empirical Study about Separation of Concerns Approachs“

Diaz Pace und Campo untersuchten in [8] verschiedene Systeme, die die Trennung von Zuständigkeiten (*separation of concerns*) begünstigen. Dazu analysierten sie vier Programme, die die gleiche Aufgabe (eine mathematische Simulation) erfüllten und in verschiedenen Sprachen bzw. mit verschiedenen Hilfsmitteln entwickelt wurden.
Als Referenz diente hier eine objektorientierte Implementierung in Java, die mit zwei aspektorientierten Systemen verglichen wurde.
Zum einen wurde AspectJ benutzt und als zweiter aspektorientierter Ansatz wurde ein aspektorientiertes Framework (TaxonomyAop) verwendet. Der Unterschied zwischen diesen beiden Varianten liegt darin, dass AspectJ den Code bei der Übersetzung miteinander verwebt, und das TaxonomyAop Framework dies erst zur Laufzeit macht, indem es Reflection[1] verwendet.
Als viertes System wurde ein architekturbasiertes Framework (Bubble) verwendet, auf das ich hier allerdings nicht weiter eingehen werden.

Der Performancevergleich zeigt, dass die aspektorientierte Variante in AspectJ nahezu genauso schnell ist, wie der normale Javacode. Das aspektorientierte Framework war allerdings um den Faktor drei langsamer.

Um die Komplexität der verschiedenen Varianten zu messen, wurden die Anzahl der Klassen, der Methoden und die Codezeilen gezählt, sowie mehrere Quotienten gebildet (Methoden pro Klasse sowie Codezeilen pro Klasse und Methode). Des Weiteren wurde ein Komplexitätsfaktor (CCN, *cyclomatic complexity number*) bestimmt.
Dabei fällt auf, dass die aspektorientierten Varianten zwar mehr Klassen und mehr Codezeilen enthalten, aber diese Klassen weniger komplex sind (die Klassen enthalten im Durchschnitt weniger Methoden und die Methoden weniger Codezeilen). Auch der durchschnittliche CCN-Wert, der die Komplexität der Methoden angibt, ist bei den aspektorientierten Varianten kleiner.

Das Bewerten zu welchem Grad Code, der fachlich nicht zusammengehörig ist, miteinander vernetzt (*tangled*) ist, gestaltet sich nicht einfach. Hierzu wurde ein Faktor eingeführt, der aus den Werten, die zur Messung der Komplexität bestimmt wurden, berechnet werden kann. Auch hier erreichten die aspektorientierten Varianten das erwartete Ergebnis, indem sie weniger miteinander vernetzten Code enthielten.

An dieser Stelle machen Diaz Pace und Campo auf die Ergebnisse der Walker Studie [7] aufmerksam, die besagt, dass auch nicht verwobener Code voneinander abhängig sein kann. Auch sie haben bei der Entwicklung der verschiedenen aspektorientierten Varianten bemerkt, dass die Aspekte, die ein klar abgegrenztes Aufgabengebiet hatten, einfacher zu verstehen waren. Als Beispiel hierzu nennen sie auch die Synchronisierung im Gegensatz zu mathematischen Aufgaben die durch Aspekte realisiert wurden. Um diese mathematischen Aspekte zu realisieren war auch hier ein komplizierteres (*wide*) Interface notwendig. Somit bestätigen Diaz Pace und Campo explizit die Erkenntnisse aus [7].

4.2 „A Study on Exception Detection and Handling using Aspect-Oriented Programming“

In [9] beschreiben Lippert und Lopes eine Fallstudie, in der sie die Fehlerbehandlung eines Frameworks für interaktive Business-Applikationen (das JWAM Framework der Universität Hamburg) mit aspektorientierten Werkzeugen überarbeiten. Das Framework besteht aus über 600 Klassen und Interfaces und ist

[1] Reflection bezeichnet den Mechanismus, Klassen zur Laufzeit zu untersuchen und in gewissem Maße zu verändern. Hier wird also erst zur Laufzeit der Aspektcode hinzugefügt ohne die Klassen bei der Kompilierung kennen zu müssen. Leider ist dies meist sehr inperformant.

in Java geschrieben.

Es wurden zwei Kernaspekte der Fehlerbehandlung in eine aspektorientierte Form überführt.
Zum einen besteht ein Großteil des Exceptionhandlings des Frameworks aus redundantem Code, der wiederum in den meisten Fällen eine Exception nicht behandelt, sondern diese protokolliert und dann fortfährt.
Zum anderen basiert das Framework auf dem Design-by-Contract-Prinzip. Innerhalb jeder Methode wurde daher geprüft, ob die übergebenen Parameter semantisch korrekt sind (z.B. dürfen Parameter vom Type `object` hier niemals `null` sein). Auch die Rückgabewerte der Methoden wurden vor dem Beenden der Methode auf ihre Korrektheit überprüft.

Dieser Code, der in vielen Klassen der selbe oder sehr ähnlicher war, wurde in Aspekte ausgelagert. Dazu wurde AspectJ Version 0.4 verwendet.

In dem ursprünglichen Framework waren 11% des Codes (4856 Zeilen) für Exceptionhandling und das Prüfen von Vor- und Nachbedingungen zuständig. Dieser Anteil konnte durch die Verwendung von Aspekten erheblich auf 3% (1160 Zeilen) reduziert werden. Es konnten z.B. 1510 Vorbedingungen der Form `arg != null` auf einige wenige Aspekte reduziert werden.
Außerdem ermöglicht dieser Aspektcode, dass auch die Vor- und Nachbedingungen von Interfaces, die Benutzer des Frameworks implementieren, geprüft werden können. Dies ist mit Java ohne Erweiterung nicht möglich bzw. der Benutzer, der Interfaces implementiert, muss dies manuell tun.

Neben der enormen Reduzierung des Quellcodes wurden aber auch andere Vorteile der aspektorientierten Version beobachtet: Da nun die Fehlerbehandlung vom restlichen Code getrennt ist, ist es einfacher bzw. überhaupt erst möglich, das Framework mit unterschiedlichen Fehlerbehandlungskonfigurationen auszustatten. Auch Änderungen in den Anforderungen, die die Fehlerbehandlung betreffen, können einfacher umgesetzt werden.

Lippert und Lopes weisen des Weiteren darauf hin, dass die Walker-Studie [7] zwar einen anderen Fokus hatte, aber sie die Ergebnisse, dass Aspekte ein klar abgegrenztes Aufgabengebiet haben sollten, durchaus bestätigen können.
In ihrer Fallstudie entsprach sowohl die Fehlerbehandlung, als auch das Prüfen von Vor- und Nachbedingungen dieser Anforderung und dem was Walker als ein „*narrow aspect-code interface*“ bezeichnet und ließen sich dadurch, ohne größere Probleme, als Aspekte modellieren.

5 Zusammenfassung der Ergebnisse

Die hier vorgestellten Untersuchungen haben, zum Teil zu einem sehr frühen Zeitpunkt, interessante Ergebnisse zur aspektorientierten Programmierung hervorgebracht.
Man ist sich darüber einig, dass AOP in gewissen Bereichen große Vorteile mit sich bringen kann, und hat dafür auch mehrere Beispiele angegeben. Aber es ist auch aufgefallen, dass es durchaus Fälle gib, in denen auf den ersten Blick zwar Aspektorientierung angebracht erscheint, aber sie die gewünschten Vorteile nicht ausspielen kann.
Walker macht dafür unter anderem das Aspektinterface verantwortlich und andere stimmen ihm da zu. Wenn ein Aspekt ein klar abgegrenztes Interface und ebenso klar abgegrenzte Aufgaben hat kann AOP seine versprochenen Vorteile am besten ausspielen. In diesem Fall können Aspekte und Klassen unabhängig voneinander programmiert und auch verwendet werden, die dann durch Zusammenfügen ihre neue Funktionalität entwickeln.
Sobald das Interface komplexer wird, oder aber der Aspektcode nicht mehr unabhängig vom Objektcode ist, gehen einige Vorteile von AOP verloren. Dies wurde im Changeexperiment von Walker deutlich, indem er zeigte, dass die AspectJ-Programmierer hier nicht effektiver arbeiten konnten als die Programmierer in der Vergleichssprache.
Doch er zeigt leider nicht auf, in welchem Maße die aspektorientierte Programmierung hier andere Vorteile mit sich bringt. So erhält man beispielsweise trotz eines komplizierten Aspektinterfaces getrennte Einheiten von Aspekten und Klassen und vermeidet redundanten Code. Auch wird der Java-Objektcode weniger umfangreich, da gewisse Teile in die Aspekte ausgelagert wurden. Dies könnte durchaus von Vorteil sein, wenn man Arbeiten (z.B. Fehlersuche und Veränderungen) an Code durchführt, der nicht den Bereich der Aspekte berührt.

Es stellt sich des Weiteren die Frage, in wie weit man die speziellen Ergebnisse von Walker in die heutige Zeit übertragen kann. Er verwendete eine der ersten Versionen von AspectJ, die bis zur heutigen Version eine große Evolution durchgemacht hat.
Würde man heute zu den gleichen Ergebnissen kommen, wenn man anstatt der Aspektsprachen (die heute nicht mehr Teil von AspectJ sind und nicht weiterentwickelt werden) `Cool` und `Ridl` diese Aspekte mit AspectJ Version 1.x entwickeln würde? Auch die heutige Verfügbarkeit von Entwicklungsumgebungen mit Unterstützung für AspectJ könnte das Ergebnis verändern.

Das durchaus wichtigere Ergebnis, neben den absoluten Zahlen und Messergebnissen, ist aber, dass

das Aspektinterface eine beachtliche Rolle für den Nutzen und die Akzeptanz der Aspekte sowie für die Effizienz der Programmierer spielt. Dieses Ergebnis wurde von weiteren Untersuchungen (auch mit neueren Versionen von AspectJ) bestätigt.

Während sich in der Objektorientierung Mehtoden zum Entwurf von Objektmodellen entwickelt haben, fehlt etwas vergleichbares noch für diese relativ junge Technologie.
Es fehlt den Anwendern an Erfahrung, wie sie Aspektinterfaces entwerfen und was dabei Vor- und Nachteile mit sich bringt. Auch eine klare Liste an Kriterien, mit der man entscheiden kann, ob etwas als Aspekt oder als Klasse entworfen werden soll, fehlt noch.

Nachdem prinzipiell gezeigt wurde, dass AOP, *richtig* angewendet, durchaus Vorteile hat, muss nun erforscht werden, was es bedeutet AOP *richtig* anzuwenden!
Entscheidungshilfen für und gegen AOP bei gewissen Problemstellungen und Richtlinien zum Entwurf „guter“ Aspektinterfaces wären wünschenswert. Auch eine Sammlung bewährter Aspekte (entweder als Code für eine gewisse Umgebung oder als Entwurfsmuster so wie die Design Patterns von Gamma et al.) wäre erstrebenswert.

Aspektorientierte Programmierung hat ein großes Potenzial und kann, durch die Verfügbarkeit von AspectJ und Implementierungen für andere Systeme, auch heute schon eingesetzt werden.
Dabei sollte man allerdings die oben beschriebenen Schwierigkeiten beachten und nicht versuchen zu viele Probleme mit Aspekten zu lösen, wenn es bewährte objektorientierte Methoden gibt. Denn ein Sprichwort besagt, dass für denjenigen der nur einen Hammer besitzt, die Welt aus lauter Nägeln besteht. Diesen Fehler kann man auch mit neuen Technologien in der Softwaretechnik begehen!

Die hier beschriebenen Studien haben alle gezeigt, dass Aspektorientierung die bekannte Objektorientierung sehr gut erweitert, aber es auch viele Stellen gibt, an denen die bekannten und erprobten Verfahren ihre Daseinsberechtigung behalten.

Literatur

[1] Elrad, Filman, and Bader. Aspect-oriented programming. *CACM: Communications of the ACM*, 44, 2001.

[2] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[3] Clemens Hackl, editor. *Programming Methodology, 4th Informatik Symposium, IBM Germany, Wildbad, September 25-27, 1974*, volume 23 of *Lecture Notes in Computer Science*. Springer, 1975.

[4] Aspectj project homepage. www.eclipse.org/aspectj/.

[5] Eclipse project homepage. www.eclipse.org/.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[7] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *International Conference on Software Engineering*, pages 120–130, 1999.

[8] D. Pace. An empirical study about separation of concerns approaches, 2001.

[9] Martin Lippert and Cristina V. Lopes. A study on exception detection and handling using aspect-oriented programming. Technical Report CSL-99-1, 1999.