Empirische Untersuchungen von aspektorientiertem Programmieren

Benjamin Schröter
29. März 2004

Inhalt

1. Was ist aspektorientiertes Programmieren (AOP)?
2. Ein Beispiel: AspectJ für Java
3. Produktivitätssteigerungen durch AOP?
   „An Initial Assessment of Aspect-oriented Programming“
   von Robert J. Walker, Elisa L.A. Baniassad und Gail C.Murphy
4. Weitere Untersuchungen zu AOP
5. Zusammenfassung

Inhalt

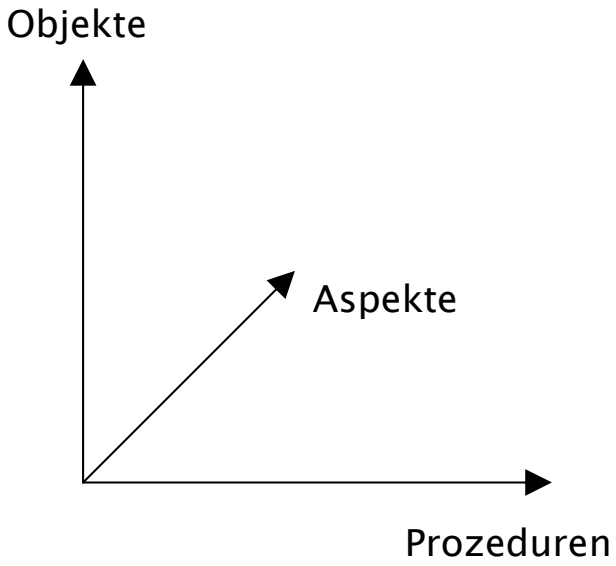
1. Was ist aspektorientiertes Programmieren (AOP)?
2. Ein Beispiel: AspectJ für Java
3. Produktivitätssteigerungen durch AOP?
   „An Initial Assessment of Aspect-oriented Programming“
   von Robert J. Walker, Elisa L.A. Baniassad und Gail C.Murphy
4. Weitere Untersuchungen zu AOP
5. Zusammenfassung

Was ist aspektorientiertes Programmieren (AOP)?

- Das Prinzip des aspektorientierten Programmierens erweitert die objektorientierte Programmierung um eine weitere Dimension
- In dieser Dimension kann auf einer Metaebene das Verhalten der Software beeinflusst werden
- Dadurch können Programme besser strukturiert werden

Die Dimensionen zur Strukturierung

- Prozedurale Programmierung enthält eine Dimension zur Strukturierung von Programmen
- Objektorientierte Programmierung (OOP) erweitert die prozedurale Programmierung um eine weitere Dimension (die Objekte)
- Aspektorientierte Programmierung (AOP) erweitert OOP wiederum um eine (die dritte) Dimension (die Aspekte)



Erhoffte Vorteile

- Bessere Trennung der Zuständigkeiten von Modulen (Code)
  seperation of conserncs
- Klareres Design
- Einfacherer Code
- Weniger redundanter Code
- Bessere Wiederverwendung von Modulen
- Einfachere Fehlersuche
- Besser Veränderbar (z.B. bei Anforderungsänderungen)

→ Bessere Qualität von Software

→ Produktivitätssteigerung bei der Entwicklung

Typische Anforderungen an Software

fachliche Anforderung

1. Fachliche Anforderungen
   - Meist Anforderungen an die Logik des Programms
   - Verhalten und Aufbau der Anwendung
     - Maskenfluss
     - „Ein Kunde kann ein oder mehrere Konten haben“
     - „Eine Adresse des Kunden ist die Postanschrift“
   - Kann gut objektorientiert modelliert werden
   - Beschreibt die Business-Objekte
   - Kann durch objektorientierte Analyse- und Designmethoden sehr gut gefasst werden

Typische Anforderungen an Software

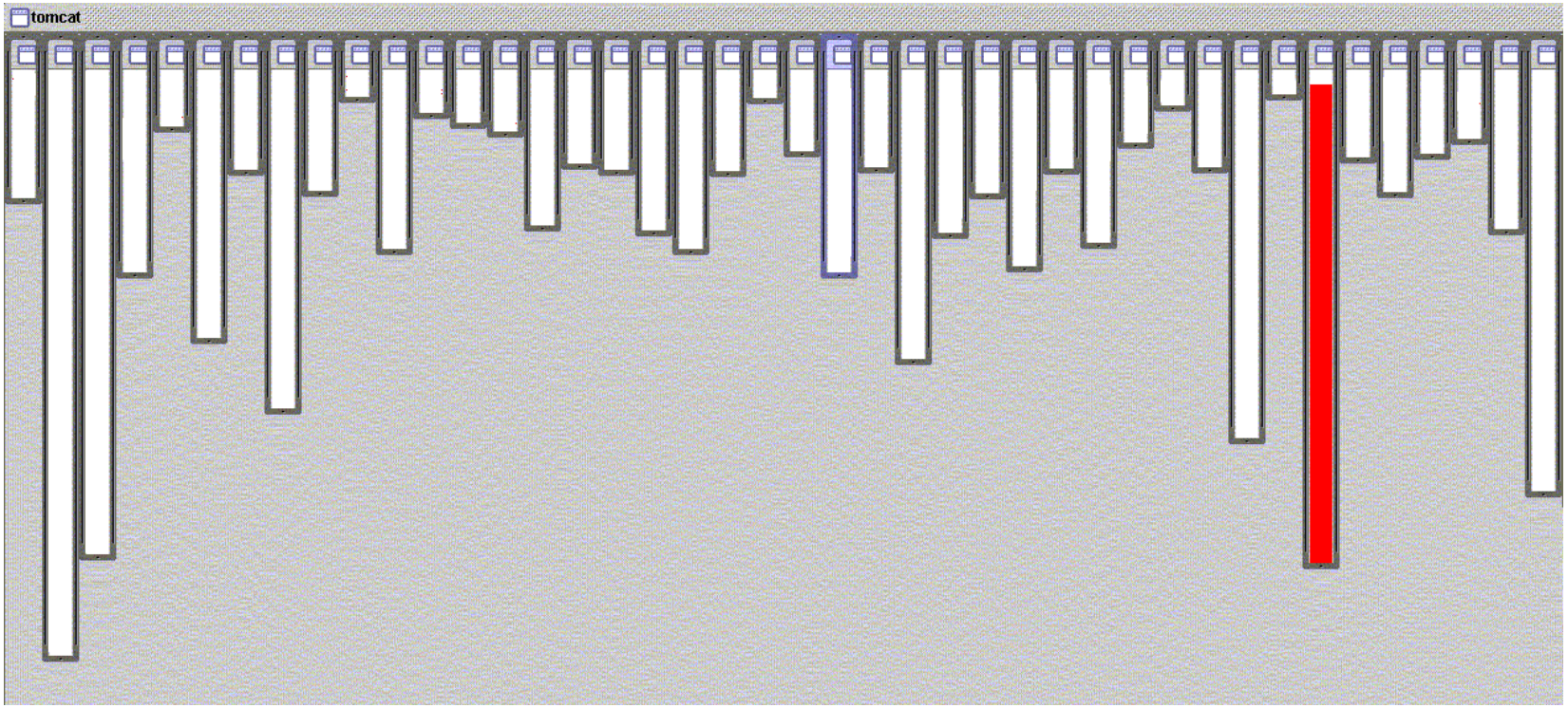
technische Anforderungen

2. Technische Anforderungen
   - z.B. Performanz, Sicherheit, Korrektheit, Fehlerbehandlung, Logging
   - Kollidieren oder vermischen sich meist mit fachlichen Anforderungen
   - Es ist meiste unklar ‚wo' sie untergebracht werden sollen
   - Können das klare Design von Business-Objekten zerstören
   - Können oft nicht von den Business-Objekten getrennt werden

→ Fachliche und technische Anforderungen vermischen sich und die Modularität von Software geht verloren

Beispiel für gute Modularität

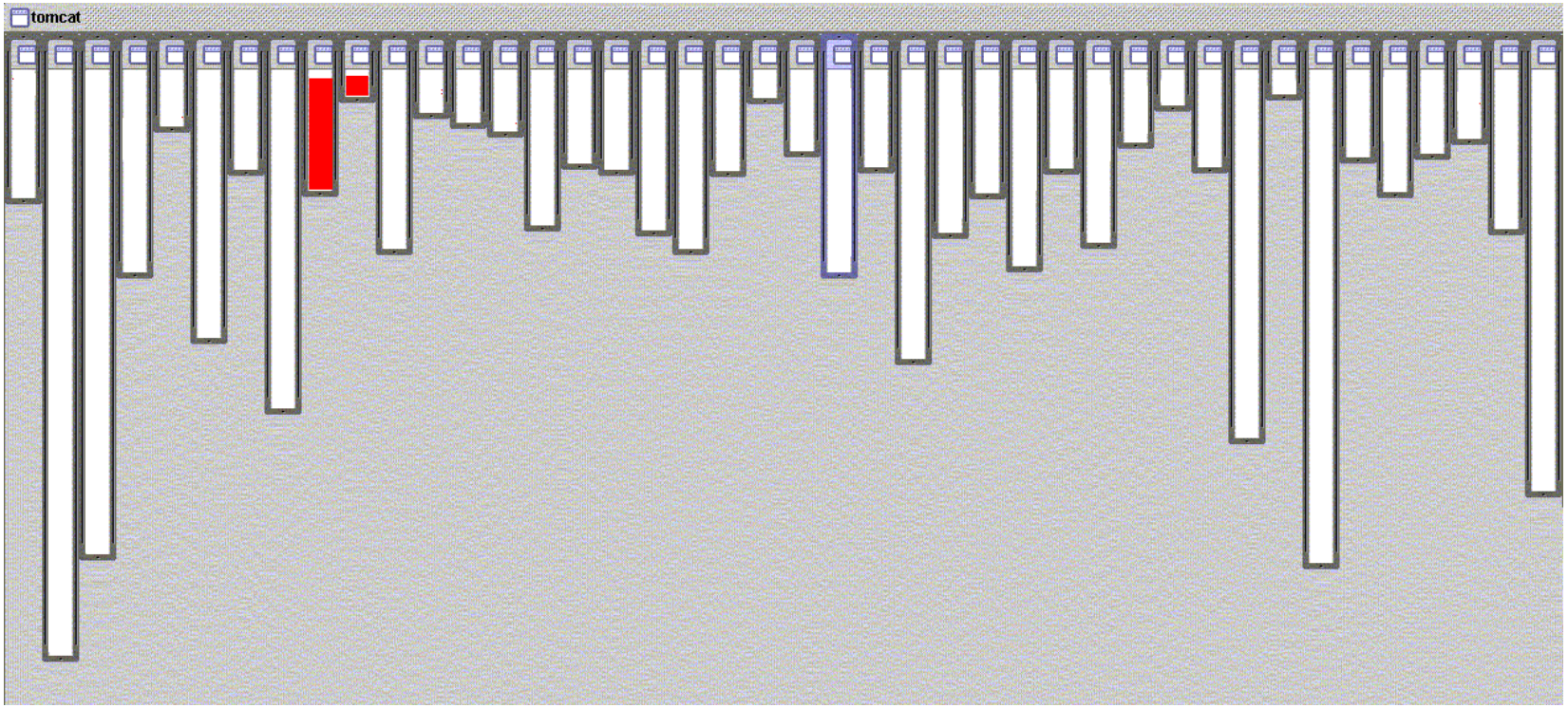
Das Parsen von XML in *org.apache.tomcat*



ist in einer Klasse gekapselt

Beispiel für gute Modularität

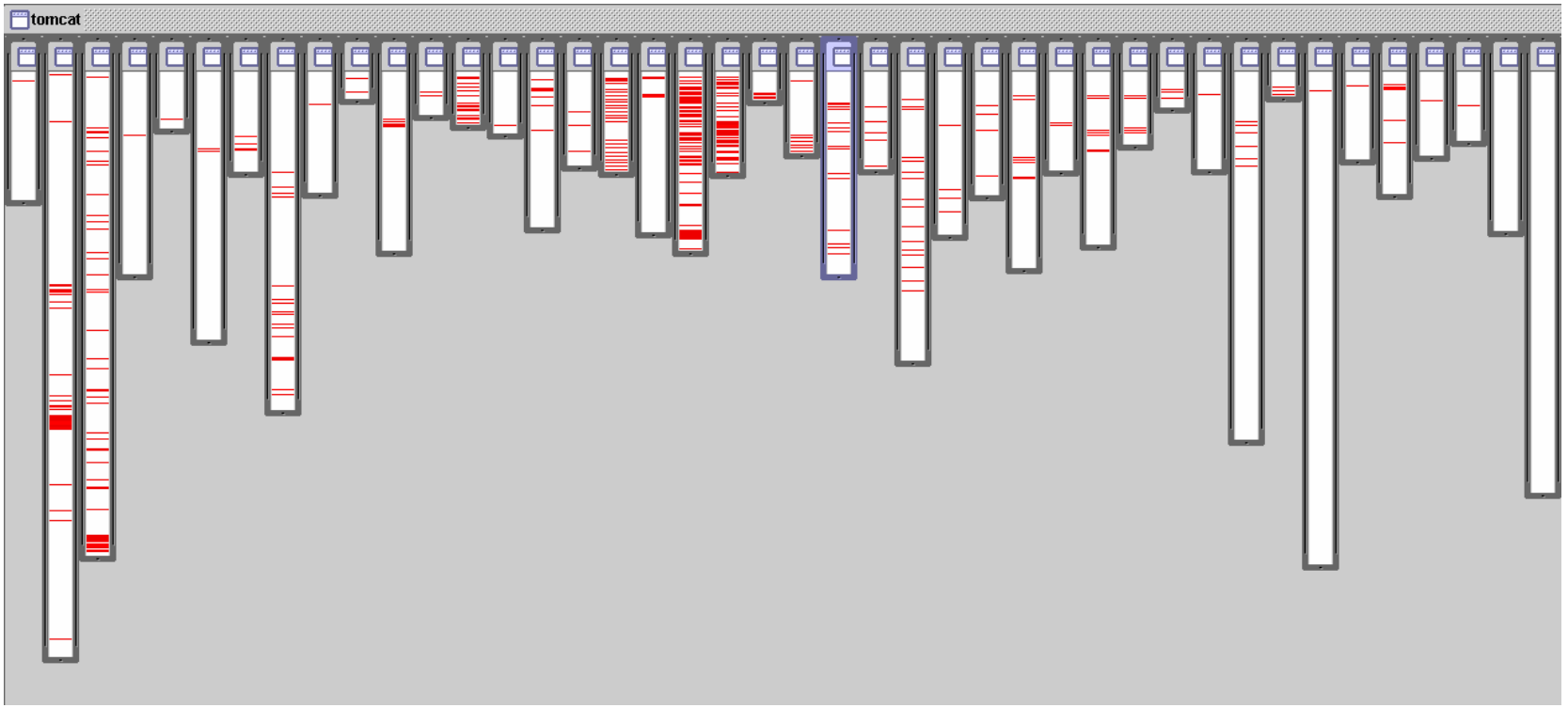
UML pattern matching in *org.apache.tomcat*



ist in zwei Klasse gekapselt (hier wurde Vererbung verwendet)

Beispiel für schlechte Modularität

Logging in *org.apache.tomcat*



verteilt sich über fast alle Klassen

Beispiel für schlechte Modularität

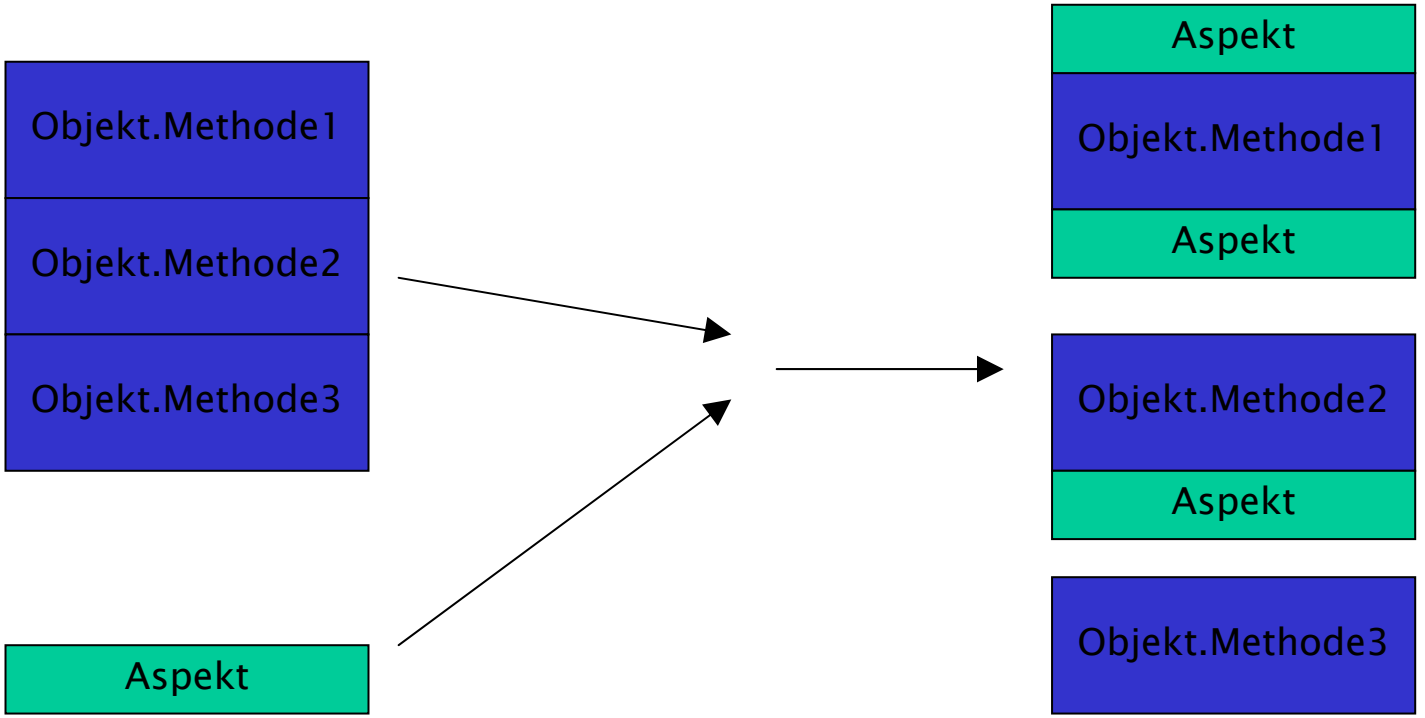
Warum ist dieser verteilte Code (*tangled code*) schlecht?

- Die Klassen erfüllen mehr als nur eine Aufgabe
- Der Quellcode wird unübersichtlich und ist schwer zu warten
- Bei Änderungen an Anforderungen, die so verteilt sind, muss der gesamte Code bearbeitet und getestet werden
- Die Wiederverwendung wird erschwert

Wie will AOP diese Probleme lösen?

- Aufgaben, die nicht klar Objekten zugeordnet werden können, werden als **Aspekte** bezeichnet
- Aspekte können getrennt von Objekten programmiert, getestet und bearbeitet werden
- Der Objektcode wird nicht vom Aspektcode beeinflusst
- Die Aspekte und Objekte werden automatisch miteinander verbunden (verwebt) um das gewünschte Verhalten zu erzeugen

Wie will AOP diese Probleme lösen?



Inhalt

1. Was ist aspektorientiertes Programmieren (AOP)?
2. Ein Beispiel: AspectJ für Java
3. Produktivitätssteigerungen durch AOP?
   „An Initial Assessment of Aspect-oriented Programming“
   von Robert J. Walker, Elisa L.A. Baniassad und Gail C.Murphy
4. Weitere Untersuchungen zu AOP
5. Zusammenfassung

AspectJ

- AspectJ ist eine aspektorientierte Erweiterung für Java
- Es ermöglicht Aspekte unter Java zu beschreiben
- Ein eigener Compiler verwebt diese Aspekte mit den Javacode

- Wurde 1997 vom Xerox Palo Alto Research Center entwickelt
- Ist heute in Version 1.1.1 Teil des Eclipse-Projekts
- www.eclipse.org
- www.eclipse.org/aspectj

AspectJ

- Wird von vielen Entwicklungsumgebungen (z.B. Eclipse, Emacs, JBuilder) unterstützt
- Kann als quasi Standard für aspektorientierte Programmierung unter Java angesehen werden

Kompatibilität

- AspectJ ist kompatibel zu Java :
  - *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.

Konzepte von AspectJ

- AspectJ führt drei neue Konzepte in Java ein:
  1. Aspekte
  2. Join Points
  3. Advices

Konzepte von AspectJ

Aspekte

- Schlüsselwort **`aspect`**
- Kapselt zusammengehörigen Code von Aspekten ähnlich wie Klassen (**`class`**) den Code von Objekten kapseln
- Besteht aus Join Points und Advices

Konzepte von AspectJ

Join Points

- Schlüsselwort **`pointcut`**
- Beschreibt die Stellen an denen ein Aspekt mit dem Objektcode verwoben werden sollen
- AspectJ unterstütz das einweben auf Methodenebene
- Mögliche Join Points sind:
  - alle setter-Methoden (einer Klasse, mehrerer Klassen, eines Packages)
  - alle Konstruktoraufrufe
  - Konstruktoraufrufe bestimmter Klassen
  - Methoden, die einem angegebenen Muster entsprechen

Konzepte von AspectJ

Advices

- Schlüsselworte **`before, after, around`**
- Enthalten den Code der vor, nach oder anstatt eines Join Points ausgeführt werden soll
- Wenn ein Advice den Objektcode eines Join Points ersetzt (**`around`**) kann innerhalb dieses Advice die ursprüngliche Methode kontrolliert oder z.B. mit anderen Parametern aufgerufen werden
- Die Advices erlauben den Zugriff auf die Parameter und Rückgabewerte (**`after`**, **`around`**) der Methoden

Demo

Inhalt

1. Was ist aspektorientiertes Programmieren (AOP)?
2. Ein Beispiel: AspectJ für Java
3. Produktivitätssteigerungen durch AOP?
   „An Initial Assessment of Aspect-oriented Programming“
   von Robert J. Walker, Elisa L.A. Baniassad und Gail C.Murphy
4. Weitere Untersuchungen zu AOP
5. Zusammenfassung

„ An Initial Assessment of Aspect-oriented Programming“ von Robert J. Walker, Elisa L.A. Baniassad und Gail C.Murphy

- Zwei kontrollierte Experimente
  - Debugexperiment
  - Changeexperiment
- Mit dem Ziel zu überprüfen, ob AOP die Produktivität von Entwicklern steigert
- Es wurde das Debugging und das Ändern eines vorhandenen Programms untersucht
- Als AOP-Sprache wurde Java mit AspectJ 0.1 verwendet

Durchführung

- Es wurde AspectJ mit einer Kontrollsprache verglichen
  - Java im Debugexperiment
  - Emerald im Changeexperiment
- Bestehende Anwendungen in diesen Sprachen galt es zu bearbeiten
- Jeweils drei Paar (Debugexperiment) bzw. drei Programmierer (Changeexperiment) lösten die gestellten Aufgaben in einer der beiden Sprachen

- Dabei wurde u.a. die Zeit für das Erledigen der Aufgaben gemessen,
- das Verhalten der Programmierer beobachtet und
- zusätzlich wurden sie während und nach dem Experiment interviewt

Anmerkungen zu AspectJ 0.1

- Die ersten Versionen von AspectJ verfolgten einen anderen Ansatz als die oben beschriebene aktuelle Version
- Anstatt einen Mechanismus zur Definition allgemeiner Aspekte zu enthalten, bestand AspectJ aus vordefinierten Aspektsprachen:
  - **`Cool`** zur Synchronisierung von Methodenaufrufen in einer Multithread-Umgebung
  - **`Ridl`** um in verteilten Anwendungen das Transferverhalten von Objekten bei Remoteaufrufen zu steuern
- Diese Aspektsprachen sind heute kein Bestandteil mehr von AspectJ und werden nicht weiterentwickelt

Ausgangssituation

- Ein Bibliothekssystem
  - Benutzer können Bücher ausleihen
  - Benutzer können abfragen ob Bücher vorhanden sind
  - Bibliotheken können diese Anfragen an andere Bibliotheken weiterleiten
- Für das Debugexperiment standen eine Java- und eine AspectJ-Variante zur Verfügung die **`Cool`** verwendet
- Für das Changeexperiment wurde eine verteilte (distributed) Version der selben Anwendung benutzt:
  - Die AspectJ-Variante verwendet **`Cool`** und **`Ridl`**
  - Als Vergleichssprache wurde Emerald verwendet, da Emerald eine objektorientierte Sprache ist, die von Haus aus verteilte Anwendungen und Synchronisation unterstützt

Debugexperiment

Durchführung

- Die Programme enthielten drei Fehler, die die Synchronisation betrafen:
  1. lesende Anfragen verschiedener Benutzer wurden sequenziell abgearbeitet
  2. zwei Benutzer konnten das selbe Buch gleichzeitig ausleihen
  3. es traten Deadlocks auf
- Die Zeit war auf 90 Minuten begrenzt

Debugexperiment

Messungen

- Die AspectJ Gruppen konnten die Fehler in kürzerer Zeit korrigieren als die Vergleichsgruppen
  - Beim ersten Fehler war der Unterschied am größten
  - Bei den anderen fiel der geringer aus
- Es wurde gemessen, wie oft die Programmierer zwischen den Dateien wechseln mussten
  - Beim ersten Fehler wechselten die AspectJ-Gruppen seltener die Dateien
  - Beim zweiten Fehler häufiger
  - Beim dritten in etwa genauso oft wie die Vergleichsgruppe
- Die AspectJ-Gruppen mussten seltener das Verhalten des Objektcodes analysieren um die Fehler in den Aspekten zu korrigieren

Debugexperiment

Ergebnisse

- Die Aufgaben lassen sich zwei Gruppen zuordnen:
  - lokale Fehler
  - nicht lokale Fehler
- Lokale Fehler lassen sich an einer Stelle korrigieren
- Nicht lokale Fehler müssen an mehreren Stellen des Codes bearbeitet werden

- Bei lokalen Fehlern haben die AspectJ-Programmierer deutliche Vorteile gegenüber der Vergleichsgruppe
- Bei nicht lokalen Fehlern haben sie zumindest keine messbaren Nachteile

Changeexperiment

Durchführung

- Das vorhandene Programm sollte an drei Stellen verändert werden
  - Eine Methode zum Zurückgeben von Büchern sollte hinzugefügt werden
  - Das Ausleihen sollte zufällig von einer zentralen Bibliothek verboten werden können.
    Dazu muss automatisch diese Bibliothek bestimmt werden und alle anderen Bibliotheken müssen diese Bibliothek befragen.
  - Die Performanz der gesamten Anwendung sollte verbessert werden.
- Auch hier war die Zeit auf 90 Minuten begrenzt

Changeexperiment

Messungen

- Es wurde nur die Zeit für die ersten beiden Aufgaben miteinander verglichen
  - Die AspectJ-Gruppen benötigten hier deutlich länger als die Vergleichsgruppe
- Dadurch blieb einem AspectJ-Programmierer keine Zeit mehr für die dritte Aufgabe
- Von den verbleibenden zwei Programmierern konnte nur einer Performanzverbesserungen erzeilen
- Alle drei Emerald-Programmierer konnten die Performanz verbessern

Changeexperiment

Ergebnisse

- Die Aspektorientierung verändert auch das Herangehen der Programmierer an den Code
  - Sie versuchten die Fehler in den Aspekten zu korrigieren, obwohl in diesem Experiment auch der Objektcode von Bedeutung war
  - Im Gegensatz zur Vergleichsgruppe verzichteten sie auf tiefer gehende semantische Analysen des Objektcodes
  - Und bearbeiteten z.T. übereilt die Aspekte

Ergebnisse

- Weder AOP noch OOP geht als klarer Sieger aus diesem Vergleich hervor
- Die Aspektsprache (**`Cool`** bzw. **`Ridl`**) oder allgemein das Aspektinterface haben einen großen Einfluss auf die Effizienz der Entwickler
- Die Aufgabe von **`Cool`** ist klar abgegrenzt und kann unabhängig von Objektcode betrachtet werden
  narrow aspect-code interface
- **`Ridl`** ist in weitaus größerem Maße mit dem Objektcode verzahnt. Auch wenn der Aspektcode örtlich getrennt ist, ist er nicht unabhängig vom Objektcode
  wide aspect-code interface

Offene Fragen

- Wären die Ergebnisse die selben gewesen, wenn das Aspektinterface von Ridl anderes entworfen wäre?
  Wenn man anstatt Ridl 0.1 einen ähnlichen Aspekt mit AspectJ 1.x schreiben würde?
- Wären die Programmierer effektiver gewesen, wenn sie bessere Unterstützung durch die Entwicklungstools gehabt hätten (z.B. so wie die Unterstützung heute ist)?
- Welche anderen Vorteile ergaben sich durch AOP in diesem Beispiel?
  - bessere Trennung von verschiedenem Code?
  - bessere Änderbarkeit an Code der nicht die Aspekte betrifft?

Inhalt

1. Was ist aspektorientiertes Programmieren (AOP)?
2. Ein Beispiel: AspectJ für Java
3. Produktivitätssteigerungen durch AOP?
   „An Initial Assessment of Aspect-oriented Programming“
   von Robert J. Walker, Elisa L.A. Baniassad und Gail C.Murphy
4. Weitere Untersuchungen zu AOP
5. Zusammenfassung

Weitere Untersuchungen zu AOP

- Zwei weitere Untersuchungen bestätigen die beschriebenen Ergebnisse
  - „An Empirical Study about Separation of Concerns Approchs“
    von Andrés Díaz Pace und Marcelo Campo
  - „A Study on Exception Detection and Handling Using Aspect-Oriented Programming“
    von Martin Lippert und Cristina Videira Lopes

„An Empirical Study about Separation of Concerns Approchs“ von Andrés Díaz Pace und Marcelo Campo

- Es wurde eine Java Version mit zwei aspektorientierten Versionen der gleichen Anwendung (einer mathematischen Simulation) verglichen
- Dazu wurde AspectJ und ein aspektorientiertes Framework benutzt
- Um die Komplexität zu vergleichen wurden mehrere Quotienten gebildet:
  - Methoden pro Klasse
  - Codezeilen pro Klasse
  - Codezeilen pro Methode
- Sowie ein Komplexitätsfaktor berechnet
- Über einen weiteren Faktor wurde ein Wert bestimmt, der angibt wie sehr Code, der nicht zusammengehörig ist, miteinander verwoben (*tangled*) ist

„An Empirical Study about Separation of Concerns Approchs“

Ergebnisse

- Die aspektorientierten Versionen enthalten zwar mehr Klassen und mehr Code sind aber weniger komplex
- Im Durchschnitt enthalten die Methoden weniger Code
- und die Klassen weniger Methoden
- Der Komplexitätsfaktor ist geringer
- Der Code ist weniger miteinander verwoben

„An Empirical Study about Separation of Concerns Approchs“

Ergebnisse

- Es wurde die Erkenntnis von Walker explizit durch Erfahrungen bei der Entwicklung der AOP-Varianten bestätigt:
  - Aspekte die klar vom Objektcode getrennt sind (z.B. Synchronisation), sind einfacher zu verstehen und machen den Vorteil von AOP klar deutlich
  - komplexere Aspekte (hier mathematische Berechnungen) lassen sich nicht vollständig vom Objektcode trennen und haben keine so klaren Effektivitätsvorteile bei der Entwicklung

„An Empirical Study about Separation of Concerns Approchs“

Laufzeit

- Zusätzlich wurde die Laufzeit der verschiedenen Programmen betrachtet:
  - Die Java- und AspectJ-Implementierungen waren nahezu gleich schnell.
    Ein Overhead durch den Aspektcode war nicht messbar!
  - Die Version mit dem aspektorientierten Framework war um den Faktor drei langsamer!
    Dies lässt sich darauf zurückführen, dass dieses Framework Refelction verwendet.

„A Study on Exception Detection and Handling Using Aspect-Oriented Programming“
von Martin Lippert und Cristina Videira Lopes

- In dieser Fallstudie wurde ein bestehendes Framework um aspektorientierte Elemente erweitert
- Grundlage bildete das JWAM Framework für Business-Applikationen der Universität Hamburg
- Es besteht aus über 600 Klassen und Interfaces
- Die Funktionen zur Fehlerbehandlung sollen als Aspekte umgesetzt werden

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

Fehlerbehandlung im Framework

- Exceptionhandling
  - Das Exceptionhandling enthält in vielen Fällen redundanten Code der über das gesamte Framework verteilt ist
  - In vielen Fällen wird eine Exception nicht behandelt, sonder nur protokolliert und dann fortgefahren

- Vor- und Nachbedingungen
  - Das Framework basiert auf dem Design-by-Contract-Prinzip
  - Innerhalb jeder Methode wird die Gültigkeit der Parameter geprüft
    - z.B. dürfen Parameter vom Type **`object`** niemals **`null`** sein
  - Auch die Rückgabeparameter werden in den Methoden auf ihre Gültigkeit geprüft

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

Ergebnisse

- Ursprünglich waren 11% des Codes (4856 Zeilen) für die Fehlerbehandlung und das Prüfen von Vor- und Nachbedingungen zuständig.
- In der aspektorientierten Version nur noch 3% (1160 Zeilen)

- 1510 Vorbedingungen der Form **`arg != null`** konnte auf wenige Aspekte reduziert werden
- Es ist nun auch ohne Weiters möglich die Vor- und Nachbedingungen von Interfaces, die von Benutzen implementiert werden, zu prüfen

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

Ergebnisse

- Auch Lippert und Lopes bestätigen direkt die Ergebnisse von Walker:
  - Bei der Anpassung des Frameworks bemerkten sie ebenfalls, dass es wichtig ist, dass die Aspekte klare abgegrenzte Aufgaben mit entsprechendem Aspektinterface haben
  - Im Fall dieser Fallstudie war dies sowohl für das Exceptionhandling als auch für das Prüfen von Vor- und Nachbedingungen gegeben

Inhalt

1. Was ist aspektorientiertes Programmieren (AOP)?
2. Ein Beispiel: AspectJ für Java
3. Produktivitätssteigerungen durch AOP?
   „An Initial Assessment of Aspect-oriented Programming“
   von Robert J. Walker, Elisa L.A. Baniassad und Gail C.Murphy
4. Weitere Untersuchungen zu AOP
5. Zusammenfassung

Zusammenfassung

- Aspektorientierung kann durchaus Vorteile mit sich bringen
- Besonders wichtig dafür ist es, dass die Aspekte klare Aufgaben haben
- und klar vom Objektcode getrennt werden können
- Wenn die Aspekte vom Objektcode abhängig sind, verschwinden ihre Vorteile zum Teil. Dies bedeutet aber nicht zwingend Nachteile gegenüber der Objektorientierung.
- Neben dem Effektivitätsvorteil bei der Entwicklung sollte man auch andere Vorteile nicht außer Acht lassen:
  - Bessere Trennung von Modulen
  - Bessere Möglichkeiten zur Wiederverwendung von Objekten und Aspekten

Offene Frage

- Wie Entwirft man *gute* Aspekte und *gute* Aspektinterfaces?
  Es fehlt noch an Entwurfsmethoden, wie es sie für die Objektorientierung gibt
- Wann sollte man ein Problem aspektorientiert, wann objektorientiert lösen?
  Hilfsmittel zur Entscheidungsfindung gibt es kaum
- Design Patterns für AOP wären wünschenswert
  Ähnlich den Design Patterns von Gamma
- Auch eine einheitliche (grafische) Notation, wie UML für die Objektorientierung, gibt es bisher nicht

Danke