Kommt mit AOP das Aus von Java und OO?

(AOP = Aspekt-Orientierte Entwicklung)

oliver.boehm@agentes.de

Abstract

Trotz Einsatz moderner Tools und Generatoren scheitern viele größere Projekte. Warum? Warum bekommt man die Komplexität nicht in den Griff? Wo bleiben die versprochenen wiederverwendbare Komponenten? Viele Frage, auf die die Objekt-Orientierung Antworten schuldig geblieben ist. Und während wir alle noch mit den Unzulänglichkeiten von Java und java-basierten Frameworks herumkämpfen, bahnt sich im akademischen Bereich bereits der nächste Paradigmenwechseln an – die Aspekt-Orientierte Programmierung (AOP). Aber ist AOP wirklich die Lösung künftiger Probleme? Eine Einführung in die wunderbare Welt der Aspekte und eine Fallstudie mit AspectJ, einem weiterentwickelten Java-Compiler, geben einen Ausblick auf morgen und die Fragen: Was bringt's? Was kann ich sparen? Wieviel bin ich schneller?

agentes

Inhalt

- von **A**ssember bis **A**spectJ
- Warum? (Hype or Hope?)
- Freie Sicht auf die Businesslogik
- Qualitäts-Aspekte

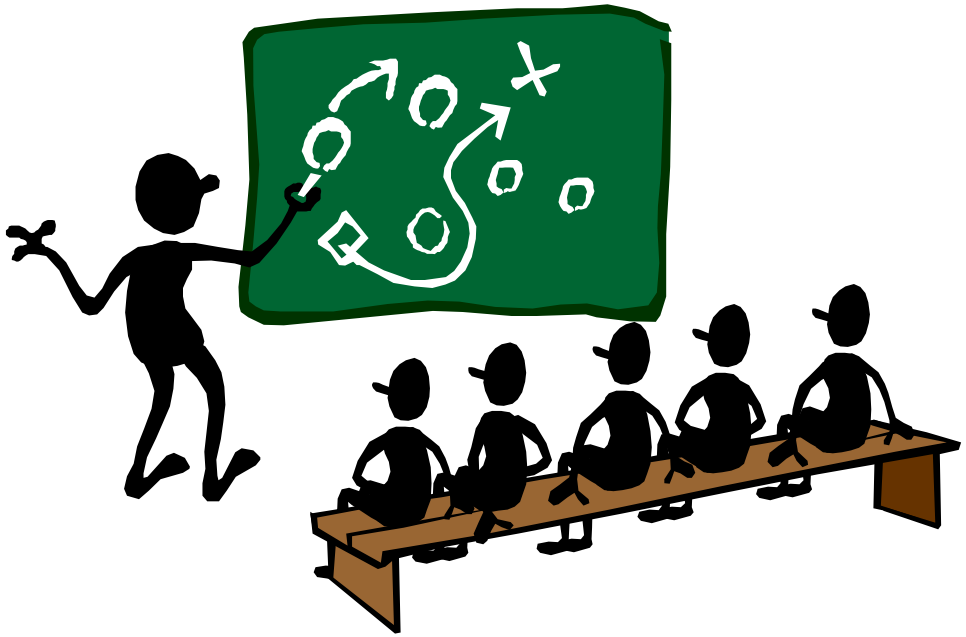
Achtung, ab hier wird's technisch

- Do you speak AspectJ (das AspectJ-Vokabuar)
- AspectJ mit Eclipse
- Achtung, Code!
- ~~ApsectJ for Hacker / IOACC~~

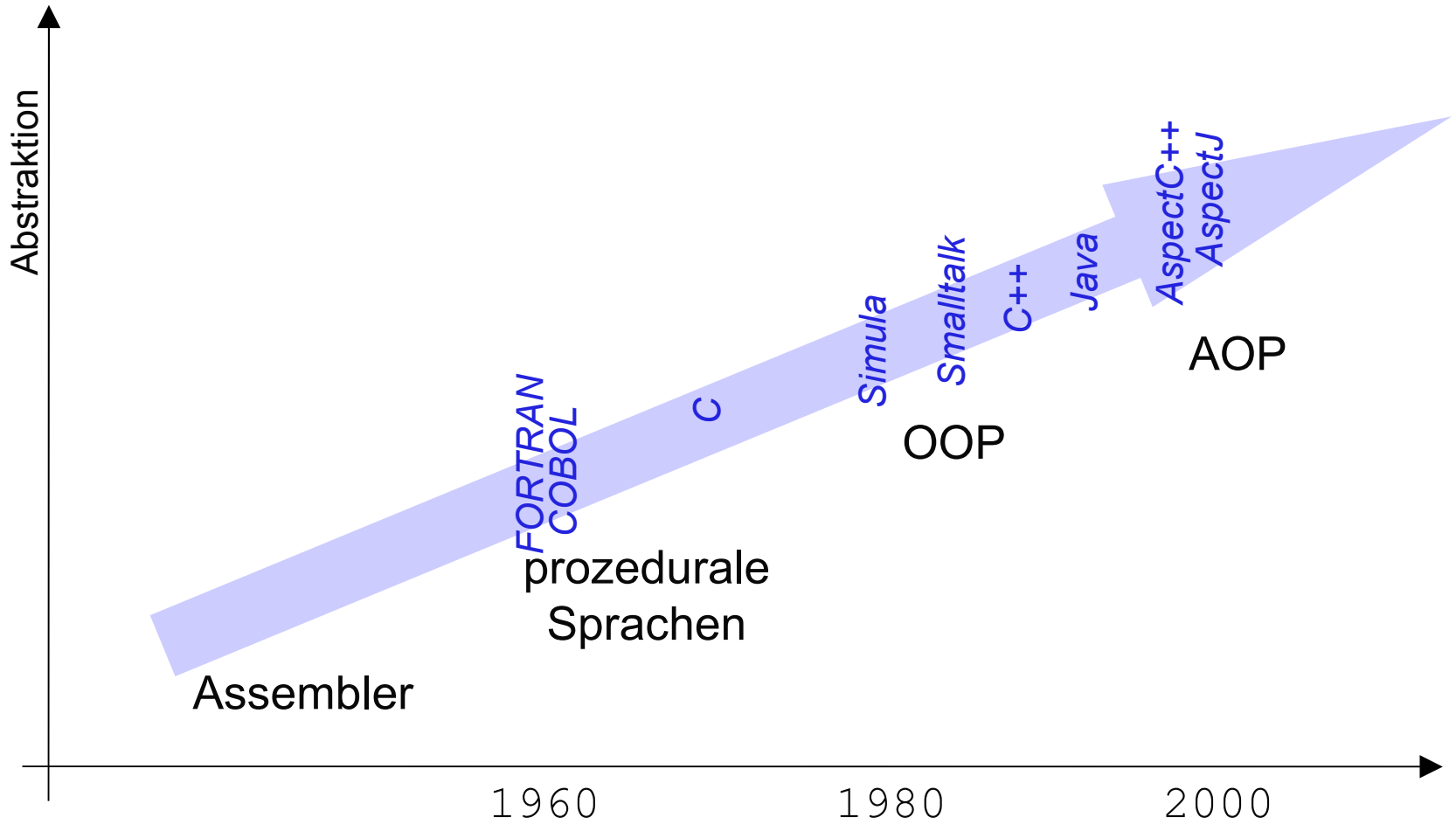
ab hier werfen wir einen Blick in die Glaskugel

- Fit For Future?
- Diskussion

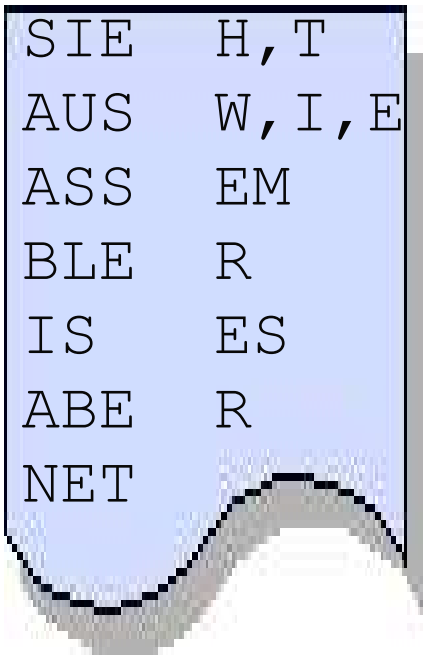
von Assembler bis AspectJ



Die Geschichte der Programmiersprachen



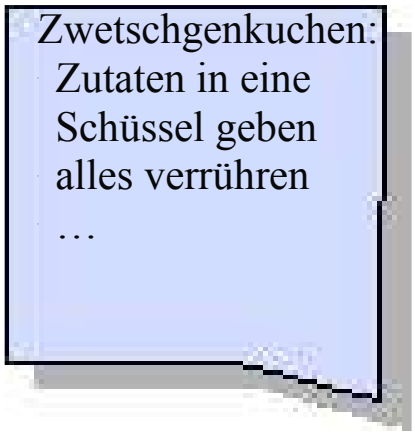
Assembler / Maschinensprache



- viele Dialekte
  - Prozessor-abhängig
  - HW-abhängig
- schnell
- nur von Spezialisten zu verstehen

prozedurale Sprachen

- Bsp: Backanleitung



PASCAL
C
COBOL
FORTRAN

```
kuchen = backen(butter, mehl,
  zucker, eier, ...)
```

agentes

OO-Sprachen

Ruby

SIMULA

Smalltalk

C++

Python

Eiffel

Java

Kuchen
Zutaten Größe
add backe

```
kuchen = new Kuche(butter, mehl, eier);
kuchen.backe();
```

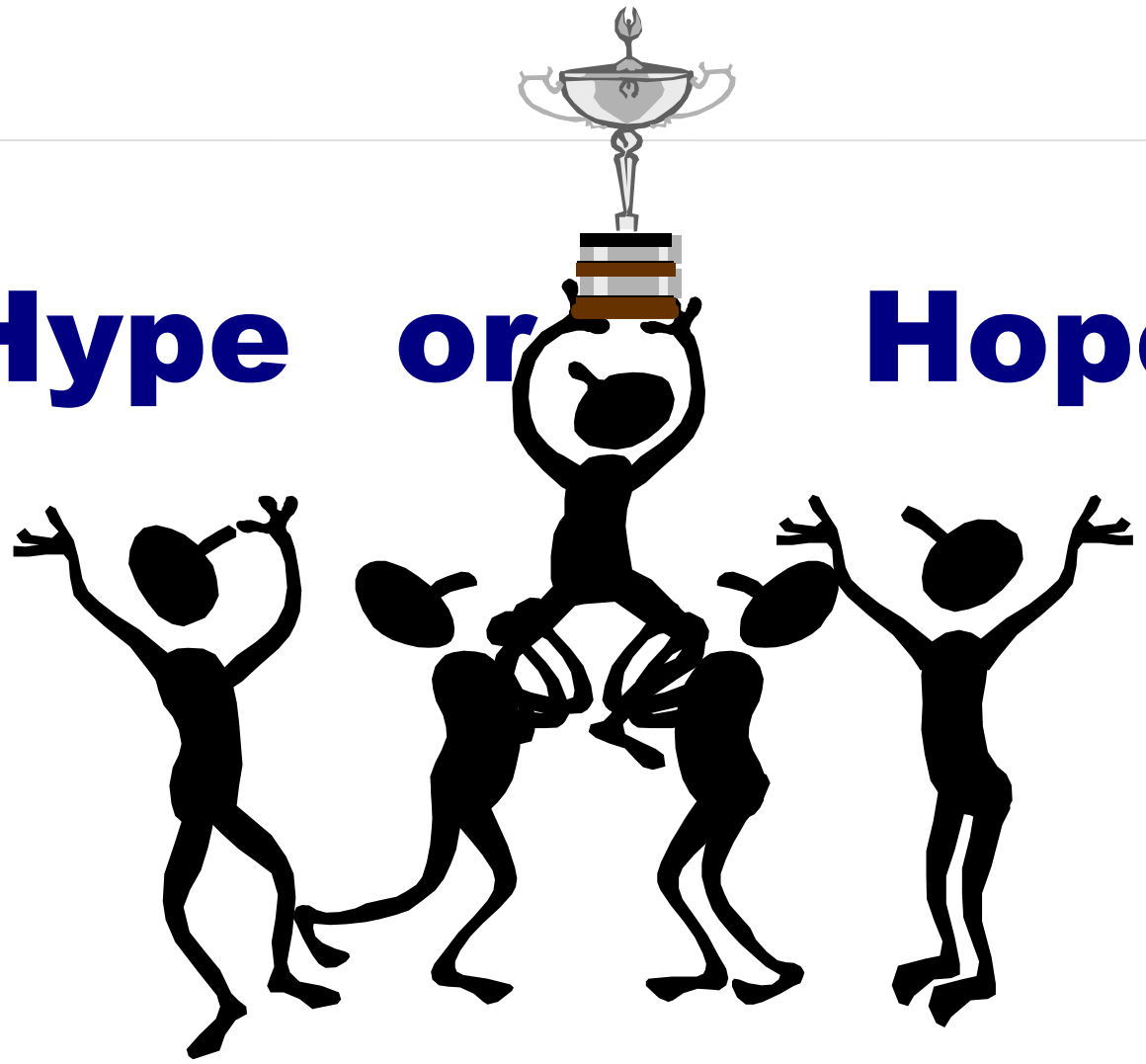
Aspektorientierte Sprachen

AspectJ AspectS AspectC++

= OOPS, um Aspekte angereichert, z.B.

- Logging-Aspekt
- Security-Aspekt
- Transaktion-Aspekt
- ...

Hype or Hope?



agentes

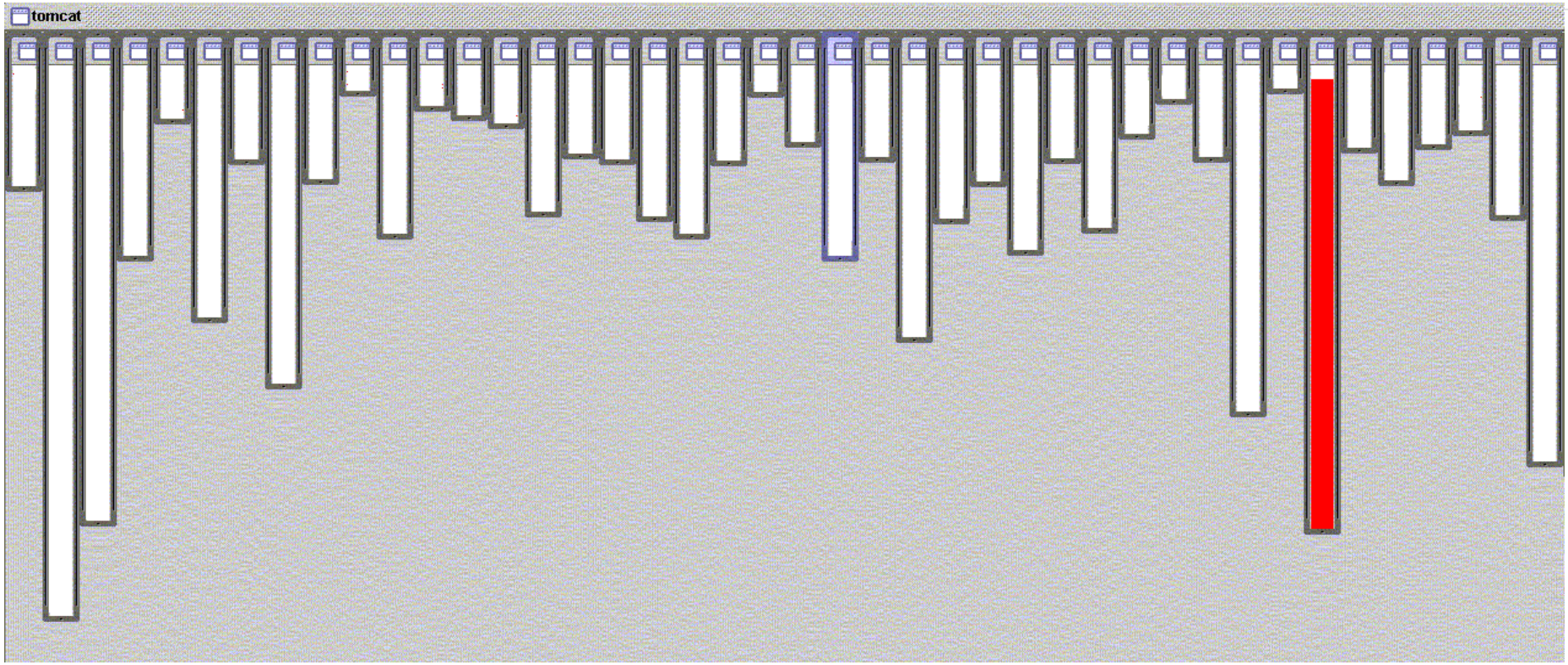
Ziel der SW-Entwicklung seit 30 Jahren

- “Divide and Conquer”
- Separations of Concern
  - Concern = eigenes Interessengebiet, Konzept, Ding, ...
  - es gibt fachliche Concerns (Geschäftslogik)
  - und nichtfachliche Concerns (Logging, Sicherheit, ...)
- Jeder Concern soll in einem eigenen Modul, Klasse, ... gekapselt werden

agentes

Modularität

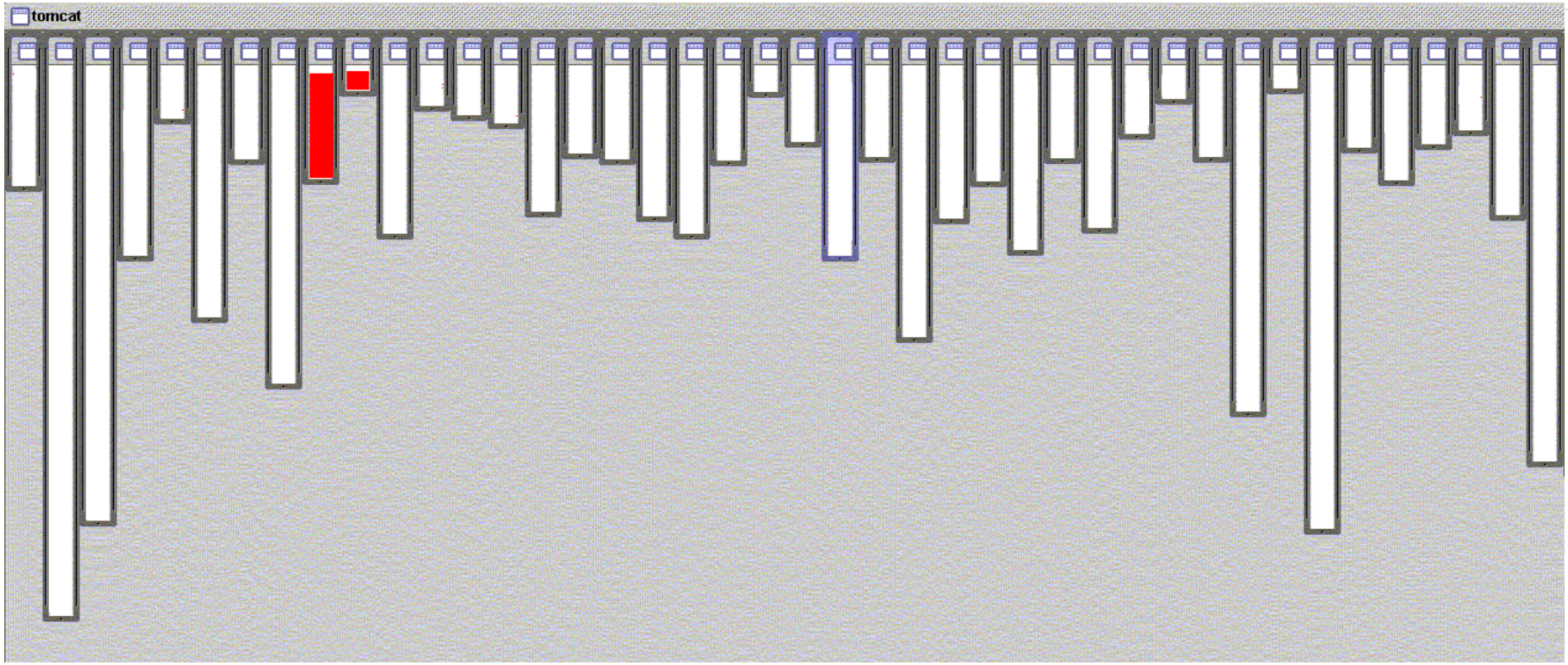
XML parsing



- XML-Parsing in org.apache.tomcat
  - rot = relevanter Code-Bereich
  - begrenzt auf 1 Modul

Modularität ok

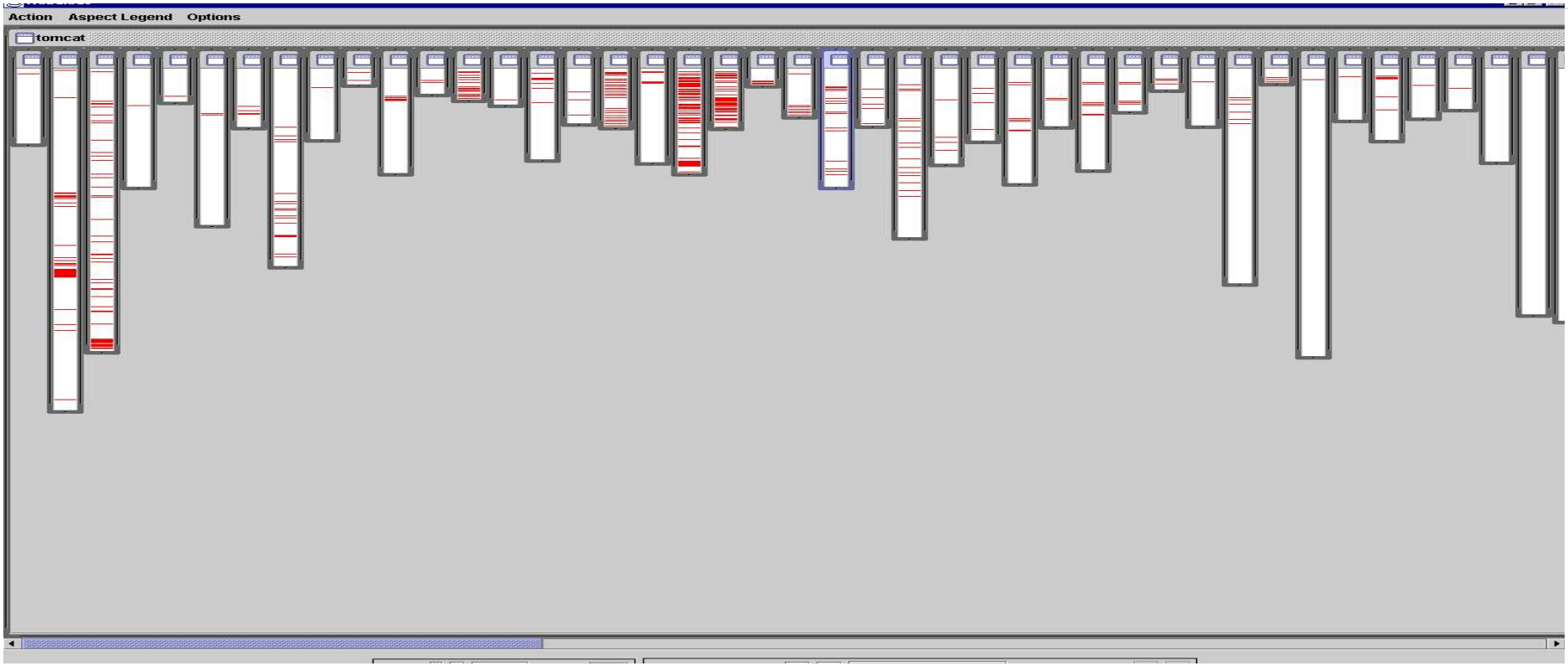
URL Pattern Matching



- URL Pattern Matching in org.apache.tomcat
  - begrenzt auf 2 Module (Verwendung von Vererbung)

problems like…

Logging ist nicht modularisiert

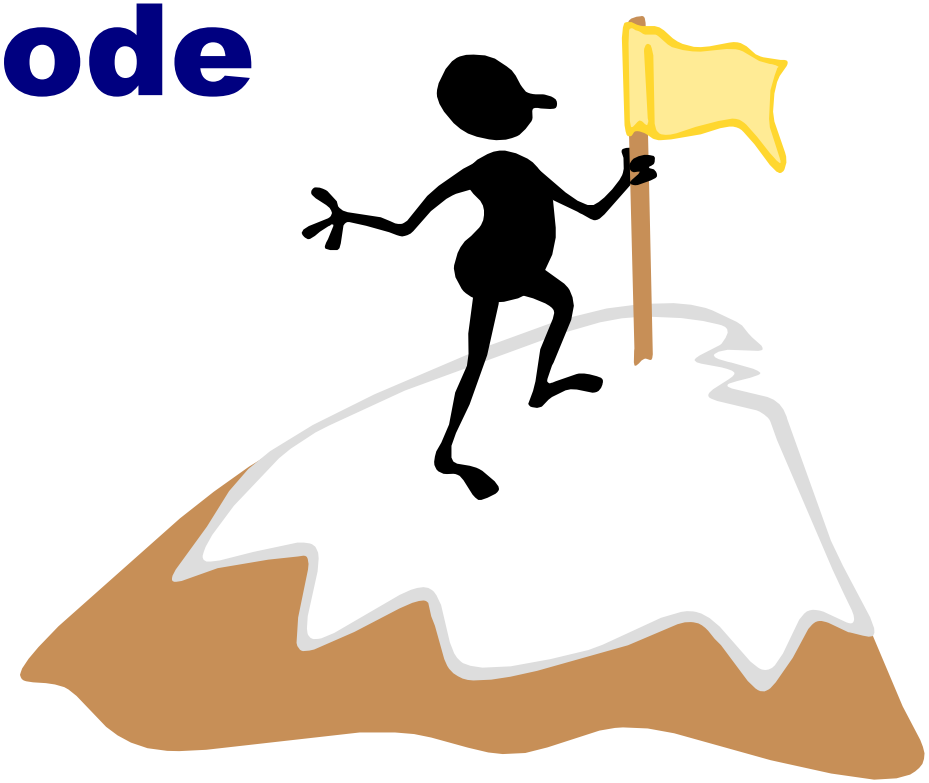


- Logging in org.apache.tomcat
  - rot: dort wird geloggt
  - überall wird geloggt

Was tun bei Log-Änderungen?

- manuell
  - aufwändig trotz IDE-Unterstützung (Refactoring)
  - fehleranfällig
- Framework?
  - Framework kann nicht entscheiden, wann geloggt werden soll
- autom. Einfügen/Anpassung von Log-Anweisungen (Log-Generator)
  - bei Änderungen muss Generator angepasst werden
  - wie konfigurieren? wie versionieren?
- Log-Container + (XML-)Deskriptoren
  - EJB-Ansatz
  - aufwendig
- vielleicht AOP?
  - mal sehn...

Freie Sicht auf den Code



fachliche Concerns
Business-Logik
nichtfachliche Concerns
Logging
Persistenz
Transaktionen
Client-Server
Exception-Handling
Fehlerbehandlung
GUI
Performance
Datenhaltung
Lastverteilung
Sicherheit

agentes

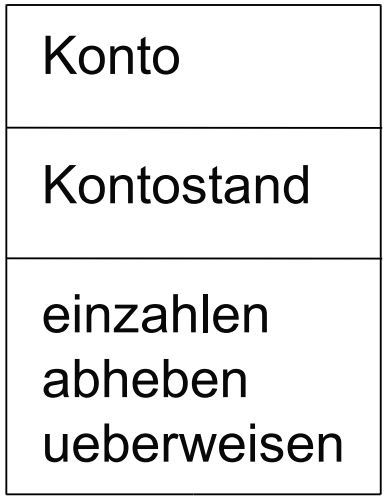
Beispiel: Bankkonto

fachliche Concerns
- Einzahlung
- Auszahlung
- Überweisung
- Bonitätsprüfung
- ...

nichtfachliche Concerns
- Logging
- Performance
- Authorisierung
- Sicherheit
- ...

Ein Fallbeispiel

```
public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        kontostand = kontostand + betrag;
    }

    public void abheben(double betrag) {
        kontostand = kontostand - betrag;
    }

    public void ueberweisen(double betrag, Konto anderesKonto) {
        this.abheben(betrag);
        anderesKonto.einzahlen(betrag);
    }

}
```



agentes

mit Fehlerbehandlung

```
public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        if (betrag < 0) {
            throw new IllegalArgumentException("Betrag negativ");
        }
        kontostand = kontostand + betrag;
    }

    public void abheben(double betrag) {
        if (betrag < 0) {
            throw new IllegalArgumentException("Betrag negativ");
        }
        kontostand = kontostand - betrag;
        if (kontoUeberzogen()) {
            kontostand = kontostand + betrag;
            throw new RuntimeException("Konto überzogen");
        }
    }

    public void ueberweisen(double betrag, Konto anderesKonto) {
        this.abheben(betrag);
        anderesKonto.einzahlen(betrag);
    }

}
```

Code verdoppelt noch lesbar

agentes

...mit Logging

```
public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        System.out.println(betrag + " soll eingezahlt werden");
        if (betrag < 0) {
            System.err.println("Einzahlung mit neg. Betrag verweigert");
            throw new IllegalArgumentException("Betrag negativ");
        }
        kontostand = kontostand + betrag;
        System.out.println(betrag + " auf Konto eingezahlt.");
    }

    public void abheben(double betrag) {
        System.out.println(betrag + " soll ausgezahlt werden");
        if (betrag < 0) {
            System.err.println("Auszahlung mit neg. Betrag verweigert");
            throw new IllegalArgumentException("Betrag negativ");
        }
        kontostand = kontostand - betrag;
        System.out.println(betrag + " von Konto ausgezahlt.");
    }

    public void ueberweisen(double betrag, Konto anderesKonto) {
        System.out.println(betrag + " soll ueberwiesen werden");
        abheben(betrag);
        anderesKonto.einzahlen(betrag);
        System.out.println(betrag + " auf " + anderesKonto + " ueberwiesen");
    }

}
```

Code vervierfacht
schwer lesbar

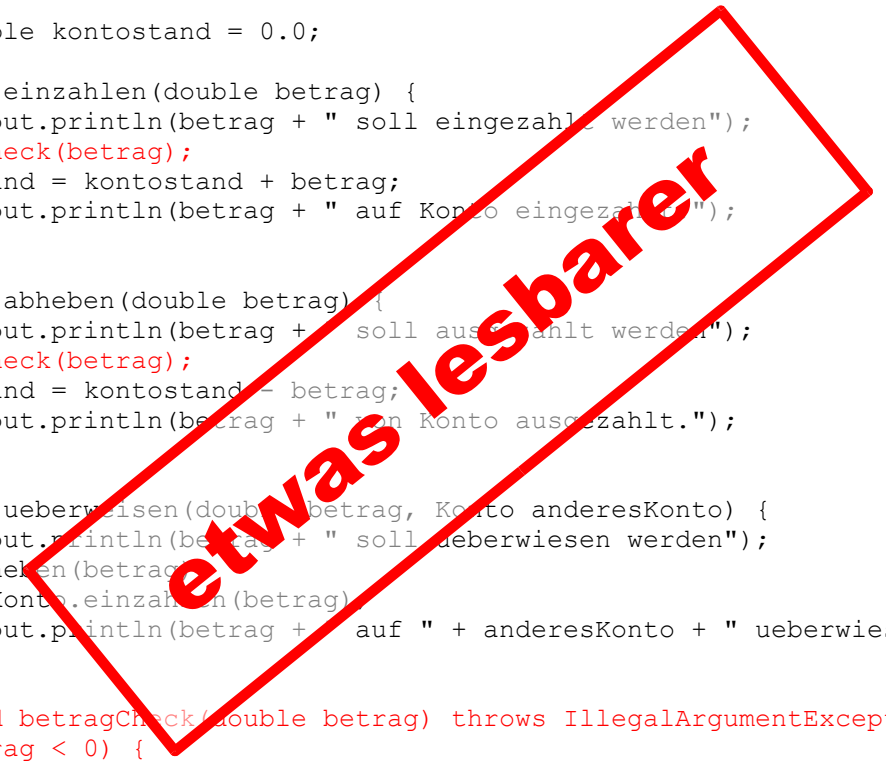
Abhilfe: Refactoring

```
public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        System.out.println(betrag + " soll eingezahlt werden");
        if (betrag < 0) {
            System.err.println("Einzahlung mit neg. Betrag verweigert");
            throw new IllegalArgumentException("Betrag negativ");
        }
        kontostand = kontostand + betrag;
        System.out.println(betrag + " auf Konto eingezahlt.");
    }

    public void abheben(double betrag) {
        System.out.println(betrag + " soll ausgezahlt werden");
        if (betrag < 0) {
            System.err.println("Auszahlung mit neg. Betrag verweigert");
            throw new IllegalArgumentException("Betrag negativ");
        }
        kontostand = kontostand - betrag;
        System.out.println(betrag + " von Konto ausgezahlt.");
    }

    public void ueberweisen(double betrag, Konto anderesKonto) {
        System.out.println(betrag + " soll ueberwiesen werden");
        this.abheben(betrag);
        anderesKonto.einzahlen(betrag);
        System.out.println(betrag + " auf " + anderesKonto + " ueberwiesen");
    }

}
```

extract method

Abhilfe: Refactoring

```
public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        System.out.println(betrag + " soll eingezahlt werden");
        betragCheck(betrag);
        kontostand = kontostand + betrag;
        System.out.println(betrag + " auf Konto eingezahlt");
    }

    public void abheben(double betrag) {
        System.out.println(betrag + " soll ausgezahlt werden");
        betragCheck(betrag);
        kontostand = kontostand - betrag;
        System.out.println(betrag + " vom Konto ausgezahlt.");
    }

    public void ueberweisen(double betrag, Konto anderesKonto) {
        System.out.println(betrag + " soll ueberwiesen werden");
        this.abheben(betrag);
        anderesKonto.einzahlen(betrag);
        System.out.println(betrag + " auf " + anderesKonto + " ueberwiesen");
    }

    private void betragCheck(double betrag) throws IllegalArgumentException {
        if (betrag < 0) {
            System.err.println("Aktion mit neg. Betrag verweigert");
            throw new IllegalArgumentException("Betrag negativ");
        }
    }

}
```



Abhilfe: Aspekte

```
public class Konto {

    private double kontostand = 0.0;

    public void einzahlen(double betrag) {
        kontostand = kontostand + betrag;
    }

    public void abheben(double betrag) {
        kontostand = kontostand - betrag;
    }

    public void ueberweisen(double betrag, Konto anderesKonto) {
        this.abheben(betrag);
        anderesKonto.einzahlen(betrag);
    }

}
```

Wo ist die Fehlerbehandlung? Logging?

Hier! Fehlerbehandlung über Aspekte

```
public aspect KontoAspect {

    pointcut zahlen(double betrag):
        call(public void Konto.einzahlen(double))
            || call(public void Konto.abheben(double)
            && args(betrag);

    void around(double betrag): zahlen(betrag) {
        if (betrag < 0) {
            throw new IllegalArgumentException("negativer Betrag");
        }
        proceed(betrag);
    }
}
```

Fehlerbehandlung in Aspekt herausgezogen

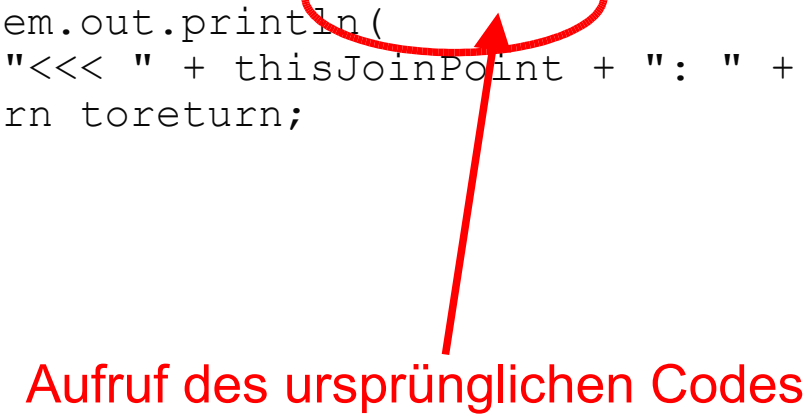
weitere Prüfungen

```
pointcut kontoMethods():
    call(public * Konto.*(..));

after(): kontoMethods() {
    if (((Konto)thisJoinPoint.getTarget()).kontostand < 0) {
        throw new RuntimeException("Konto ueberzogen");
    }
}
```

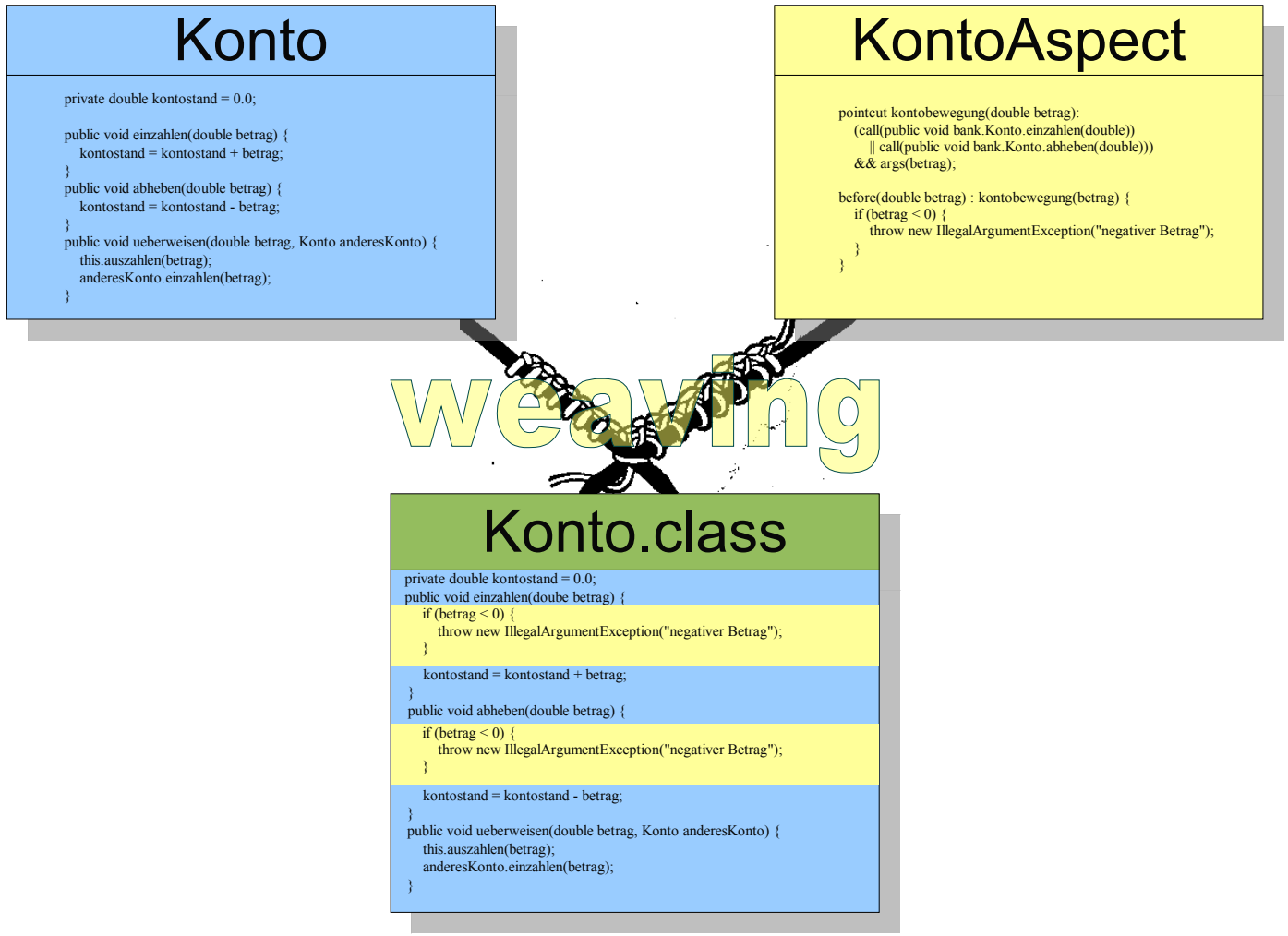
wird *after* jeder Methode “eingewebt”

Logging-Aspekt

```
pointcut allPublic():
    call(public * summer.camp.*.*(..));

Object around(): allPublic() {
    System.out.println(
        ">>> " + thisJoinPoint + ": " + thisJoinPoint.getTarget());
    Object toreturn = proceed();
    System.out.println(
        "<<< " + thisJoinPoint + ": " + thisJoinPoint.getTarget());
    return toreturn;
}
```



Der Webe-Vorgang (Weaving)



Qualitäts-Aspekte

agentes

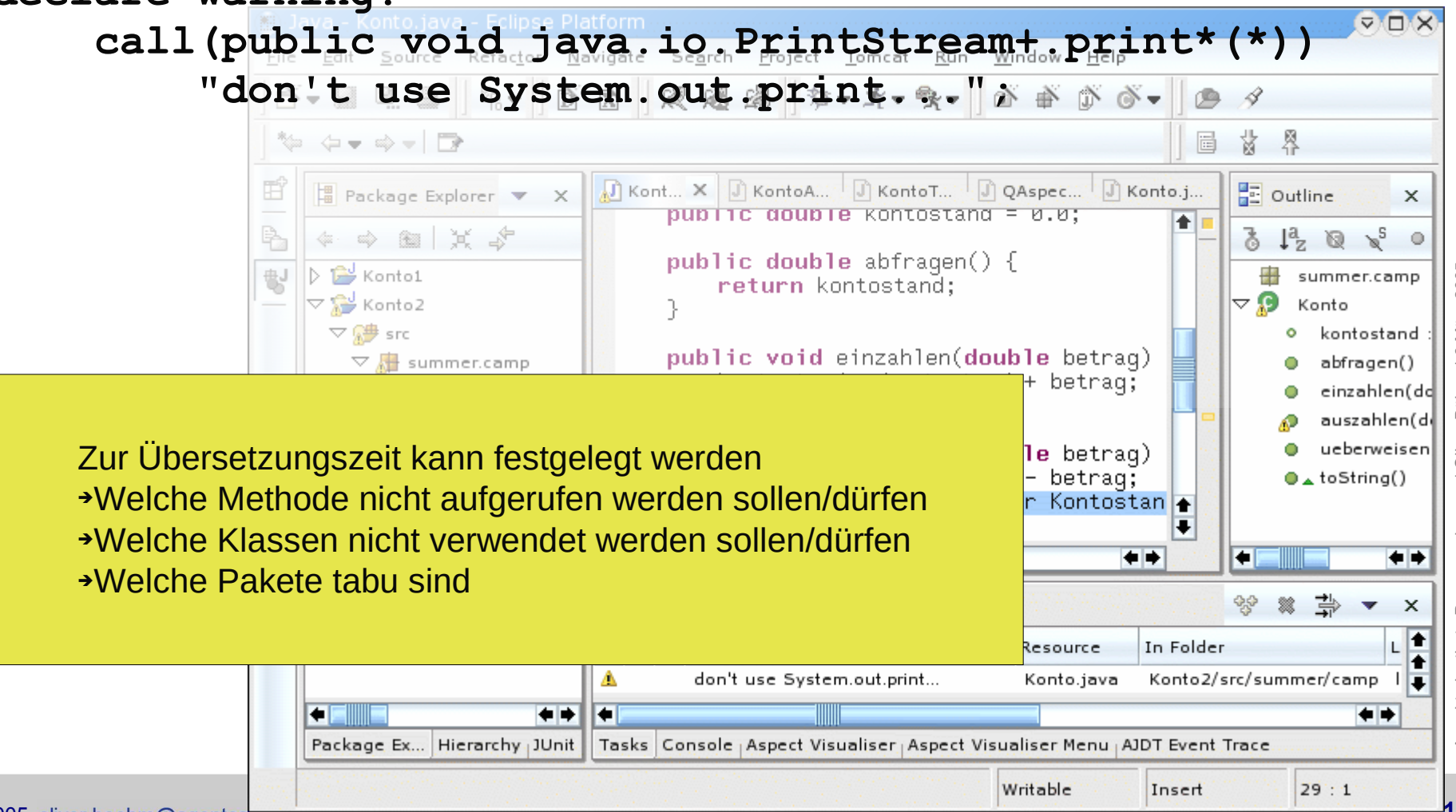
Qualitätssicherung

- declare error / declare warning
- Exception Handling
- Aspekte können eingesetzt werde für
  - Pre-/Postconditions
  - DbC (Design by Contract)

agentes

declare warning

```
declare warning:
    call(public void java.io.PrintStream+.print*(*))
        "don't use System.out.print...";
```



Zur Übersetzungszeit kann festgelegt werden
- ➔Welche Methode nicht aufgerufen werden sollen/dürfen
- ➔Welche Klassen nicht verwendet werden sollen/dürfen
- ➔Welche Pakete tabu sind

agentes

Exception

- Default-Implementierung herausziehen
- Exception automatisch abfangen
- Automatische Protokollierung
  - aller Exceptions
  - ausgewählter Exceptions

```
pointcut handleException(Throwable ex):
    handler(java.lang.Throwable+) && args(ex);

/**
 * protokolliere alle abgefangen Exceptions
 */
before(Throwable ex) : handleException(ex) {
    System.err.println("*** " + ex);
}
```

agentes

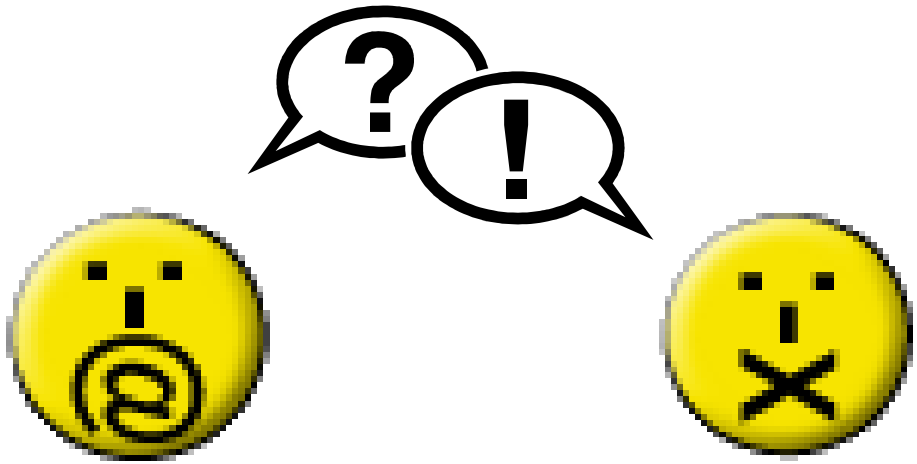
Design by Contract (DbC)

- durch Bertrand Meyer / Eiffel populär
- Idee: Aufrufer und aufgerufene Methode schließen Vertrag
- Realisierung in Eiffel:
  - über require, ensure, invariant
  - Überprüfungen können abgeschaltet werden (Performance)
- Java:
  - iContract: @pre, @post
  - assert (seit JDK 1.4)
- AspectJ:

```
//  Post-Condition
after() : call(public * Konto.*(..)) {
    assert(((Konto)thisJoinPoint.getTarget()).kontostand >= 0)) :
        "Konto darf nicht ueberzogen werden";
}
```

Technischer Teil

Do you speak AspectJ?



agentes

Begrifflichkeiten

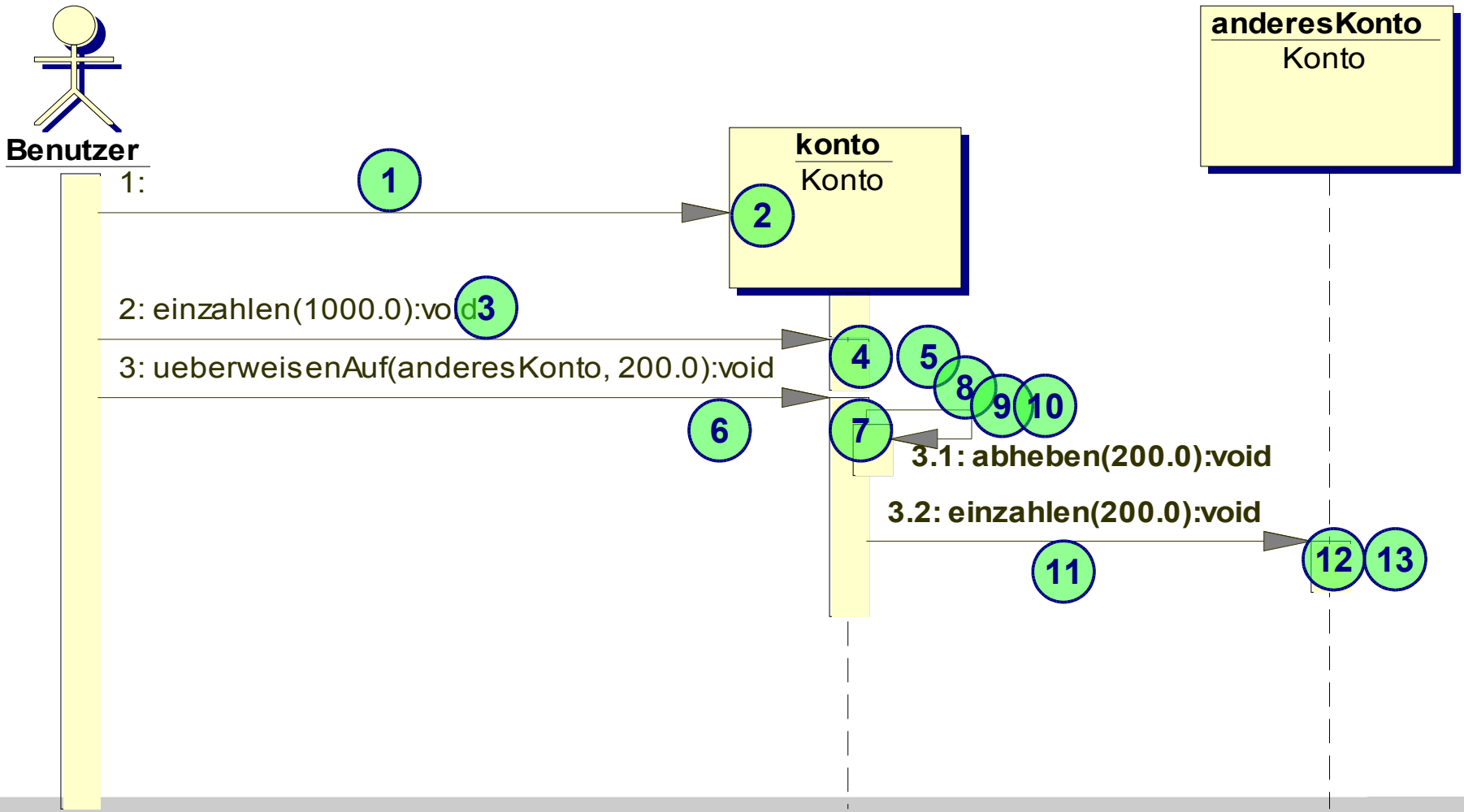
- **Joinpoint**
  - Punkte im Programm, an denen Code erweitert oder modifiziert werden soll
- **Pointcut**
  - eine Auswahl von Joinpoints
- **Advice**
  - der Code für den Pointcut
- **Introduction**
  - Erweiterung anderer Klassen, Interfaces, Aspekte um zusätzliche Funktionalität
- **Aspect**
  - Konstrukt, in dem das obige abgelegt wird

agentes

Joinpoints können sein

- Aufrufen einer Methode
- Ausführen einer Methode
- Zugriff auf eine Variable
- Behandeln einer Exception
- Initialisierung einer Klasse
- Initialisierung eines Objekts

Joinpoints im Programm (Ausschnitt)



agentes

Pointcut

- Pointcut = Pattern oder Abfrageprädikat über Joinpoints
  - Wildcards möglich
  - Logische Operatoren
- Primitive Pointcut = Pointcut ohne Namen
- Named Pointcut

Name

```
pointcut allPublic():
   call(public * summer.camp.*.*(..));
```

alle public-Methoden aus summer.camp mit beliebigen Parametern

```
pointcut zahlen(double betrag):
   call(public void summer.camp.Konto.*zahlen(double))
      && args(betrag);
```

agentes

before-Advice

- Aktion wird *vor* dem ursprünglichen Joinpoint ausgeführt

```
before(): allPublic() {
    System.out.println(thisJoinPoint);
}
```

für alle public-Methode gib den Joinpoint aus

after-Advice

- Aktion wird *nach* dem eigentlichen Joinpoint ausgeführt

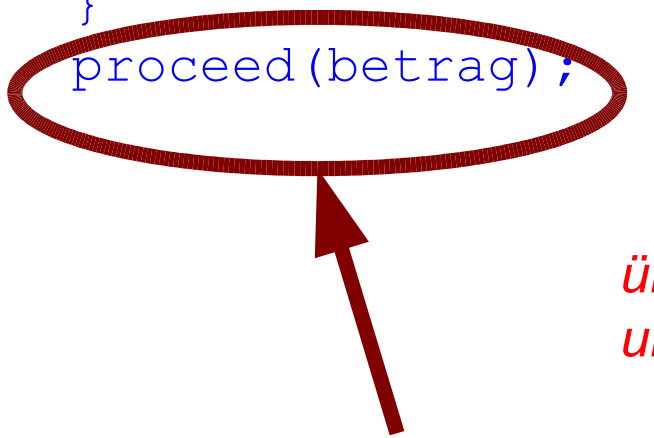
```
after(): kontoMethods() {
    if (((Konto)thisJoinPoint.getTarget()).kontostand < 0) {
        throw new RuntimeException("Konto ueberzogen");
    }
}
```

überprüfe nach jeder Konto-Methode den Kontostand

around-Advice

- Aktion wird *anstelle* des Codes ausgeführt

```
void around(double betrag): zahlen(betrag) {
    if (betrag < 0) {
        throw new IllegalArgumentException("negativer Betrag");
    }
    proceed(betrag);
}
```



überprüfe für die zahlen-Methoden den Parameter und rufe danach die eigentliche Methode auf

ruft den ursprünglichen Code auf

agentes

Aspekte

- Aspekt = Container für die neuen Konzepte
  - können Advices und Pointcuts enthalten
  - Methoden, Variablen und Inner Classes
- Aspekte sind Klassen sehr ähnlich
  - verhalten sich zu Klassen wie Klassen zu structs in C++
- können interfaces implementieren
- können von Klassen abgeleitet werden

agentes

Beispiel

```
package summer.camp;

public aspect KontoAspect {

    pointcut zahlen(double betrag):
        call(public void summer.camp.Konto.*zahlen(double))
            && args(betrag);

    void around(double betrag): zahlen(betrag) {
        if (betrag < 0) {
            throw new IllegalArgumentException("negativer Betrag");
        }
        proceed(betrag);
    }

}
```

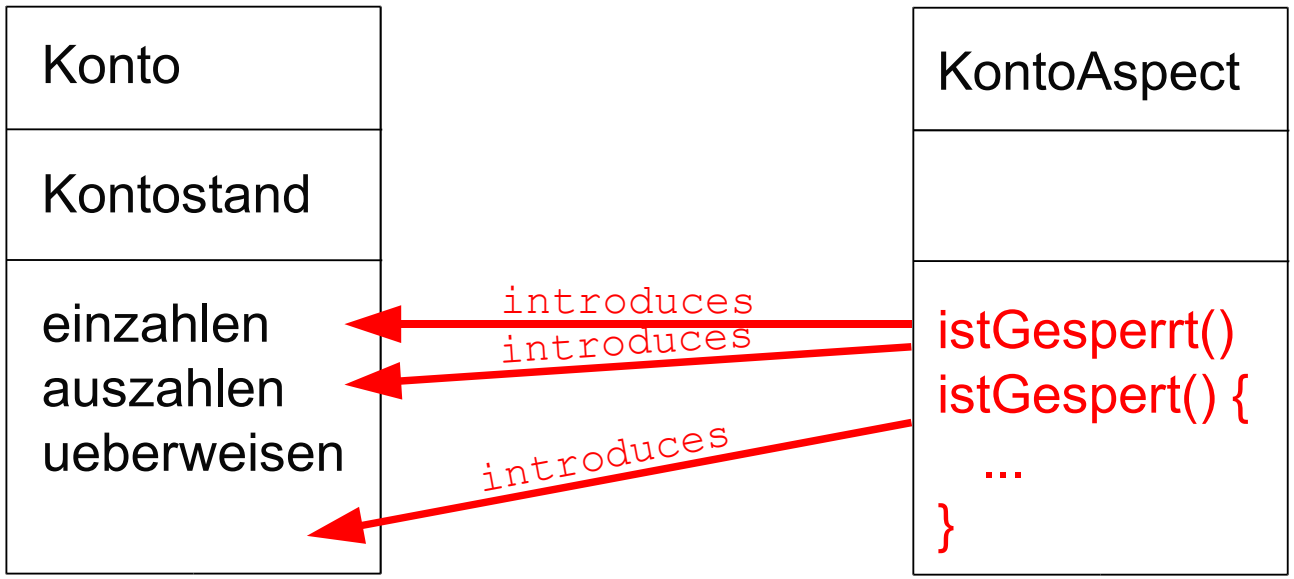
agentes

Introduction

- auch als “intra-class declaration” bekannt
- mit Introduction können andere Aspekte, Interfaces oder Klassen erweitert werden
- hinzugefügt werden können:
  - Variablen
  - Methoden
  - Interfaces
  - Vaterklassen
- bewährtes Konzept (“open classes”), z.B. in Python

Idee

- für ein/auszahlen soll zuerst überprüft werden, ob das Konto gesperrt ist



AspectJ mit Eclipse

agentes

Zutaten

3 Dinge braucht der Entwickler:

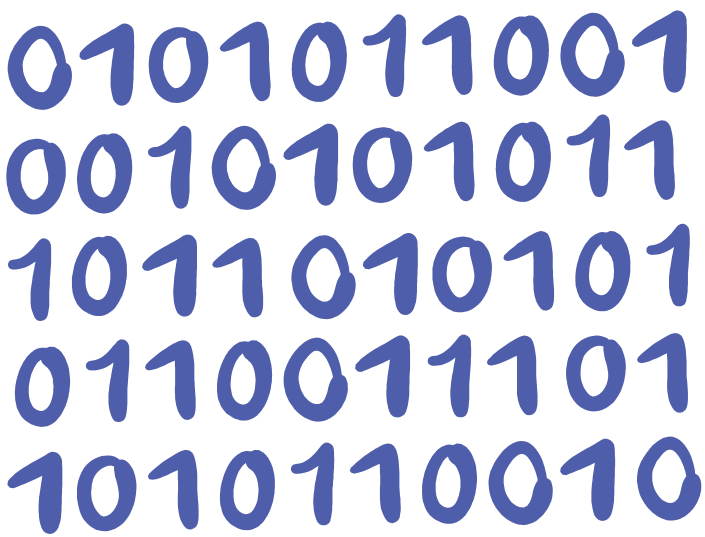
- Compiler
  - www.eclipse.org/aspectj/
- Editor
  - Emacs, ...
- Debugger
  - jdb

Oder: alles Integriert

- Eclipse
  - www.eclipse.org
- AspectJ-Plugin
  - www.eclipse.org/ajdt

gut zu wissen:
- Compiler ist OpenSource
- gesponsert von Eclipse/IBM
- User-Feedback erwünscht

Achtung, Code!



Der Blick nach vorn

AspectJ 5 & Friends

- aktuell: AspectJ 1.2
- AspectJ 5
  - Basis: Java 5 (= JDK 1.5)
  - IDE: Eclipse 3.1
  - @Annotations
  - voraussichtlich: Mitte 2005
  - http://www.eclipse.org/aspectj/
- AspectWerkz
  - Aspekte über XML
  - viele Ähnlichkeiten mit AspectJ
  - Zusammenarbeit mit AspectJ 5 angekündigt (19.1.)
  - sponsored by Bea Systems

AOP meets MDA

- noch keine graphische Notation für Aspekte
- techn. Aspekte für techn. Randbedingungen
- flexibler
- Gefahr: Aspekte ufern aus

- Idee: Modell enthält nur Business-Logik, Rest wird generiert
- restliche Logik im Generator manifestiert
- Gefahr: Generator-Entwicklung, Abhängigkeit vom Generator

agentes

erwarteter Nutzen

- Modularität auch für übergreifende Dinge (“Concerns”)
  - Java-Code konzentriert sich auf Business-Logik
  - kürzerer Code
  - einfachere Wartung und Weiterentwicklung
- erhöhte Wiederverwendbarkeit über
  - Library-Aspekte
  - Plug n' Play-Aspekte (“Patterns”) für verschiedene Anforderungen

Aber!!!

- Ähnlich wie mit Reflection kann man viel “Schweinereien” anstellen, die Änderungen zum Himmelfahrtskommando und Wartung zum Abenteuer machen können, daher:
  - nur wohldosiert verwenden
  - Programmier-Disziplin
  - Erfahrung notwendig
- Vorsicht bei Refactoring!
  - noch keine Tool-Unterstützung

Erfahrungen

- deutliche Verbesserung gegenüber der OOP
- Codereduktion
- Reduzierung der Wartungskosten
- Reduzierung der Entwicklungszeit
- Verbesserung der Codequalität
- Verbesserung der Softwarearchitektur
- Verbesserung der Performanz möglich

Quelle: http://www.jugs.de/protokolle2003.htm#300103

Code-Reduktion

- redundanter Code konnte modularisiert werden
- realisierte Codereduktion 30-95%
  - abhängig vom Einzelfall
  - abhängig vom Entwickler
- Serverseitige Komponenten: 90%
- EAI-Szenarien: 95%
- GUI-Swing-Code: 50%

agentes

Entwicklungszeit

- Neuentwicklung:
  - 20-40% schneller fertig bei gleicher Qualtität oder
  - deutlich bessere Qualität in der gleichen Zeit
- nicht funktionaler Code kann mit deutlich weniger Aufwand realisiert werden
- Unterstützung durch
  - konsequentes Exception-Handling
  - gutes Tracing
  - einfaches Profiling, etc.

Verbesserung der Architektur

- bessere Muster sind möglich
- existierende Muster können effizienter implementiert oder als Aspekte in Bibliotheken ausgelagert werden
- neue Architekturen werden möglich (z.B. virtueller, geteilter interner Nachrichtenbus)
- Architekturbedingungen lassen sich durch Aspekte zur Compilezeit überprüfen
- zirkuläre Abhängigkeiten können elimiert werden

Verbesserung der Performance

- viele “Flexibilitätslayer” für künftige (meist dann doch nicht stattfindende) Erweiterungen können eliminiert werden (z.B. setter und getter)
- viele Reflection-Aufrufe werden überflüssig
  - James Gosling (auf OOP 2003): *...reflection is bug no. 1 in Java...*
- Kommunikation kann virtuell indirekt, physikalisch direkt erfolgen
- Einsatz von effizienteren Pattern

AOP in freier Wildbahn

- JBoss4
  - Transaktionen, Sekurity, Caching mit AOP
  - Aspekt-Deployment
- WebLogic 8.1
  - WebLogic Aspect Framework

agentes

Einführungs-Strategien

- Top-Down (mit Unterstützung des Managements)
- Bottom up (don't tell)
- über die QA-Schiene

agentes

Adaptionsaufwand

- Ungefähr so groß wie der Umstieg von Prozedural auf OOP
- Aber:
  - er wird nicht kleiner
  - er wird kommen
  - warum nicht jetzt?
- können wir aus der Vergangenheit lernen?
- erste OO-Projekte waren oft Fehlschläge
  - Lernkurve unterschätzt
  - fehlende Begleitung von erfahrenen Experten
  - überzogene Erwartungen
- erst folgende OO-Projekte waren erfolgreich
  - klein mit unkritischem Projekt anfangen
  - Erfahrungen sammeln
  - Multiplikatoren ausbilden

Zusammenfassung

- setzt auf OO / Java auf
- AOP / AspectJ gewinnt an Bedeutung
- Entwicklung wird effektiver
- kleinere Programme möglich

- maßloser Einsatz kann genau das Gegenteil bedeuten
- Fallstricke lauern
- Selbstdisziplin nötig
- Lernkurve vergleichbar mit C → C++

agentes

Links & Literatur

- Oliver Böhm
  Aspektorientierte Programmierung mit AspectJ
  (vorraussichtlich ab ~~April~~ Ju[n,l]i 2005)
  http://www.dpunkt.de/buch/3-89864-330-1.html
- Joseph D. Gradecki, Nicholas Lesiecki
  Mastering AspectJ
  John Wiley & Sons, ISBN: 0471431044
- Aspekt-Orientierte Programmierung mit AspectJ
  http://www.agentes.de/artikel/aspectj.html
- AspectJ Summer Camp
  http://www.aspectsoft.de/ajsc/AspectJSummerCamp.pdf
- http://aspectj.swiki.net/

agentes

Bedeutung von Aspekten in 2-3 Jahren

Diskussion

info@agentes.de
boehm@javatux.de