Einführung in die Aspect Orientierte Programmierung (AOP)

Arno Schmidmeier, AspectSoft

JUGS

Zürich, Januar 2003

Inhalt

- AOP Einführung
- Meine Erfahrungen
- Fragen und Antworten

AOP Einführung

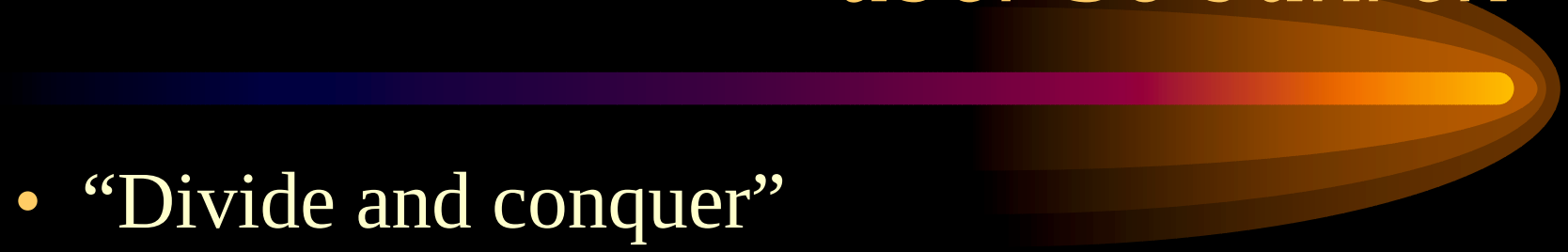
Arno Schmidmeier

AspectSoft

+49/9151/90 50 30

A@Schmidmeier.org

Ziel der Softwareengeneering seit über 30 Jahren

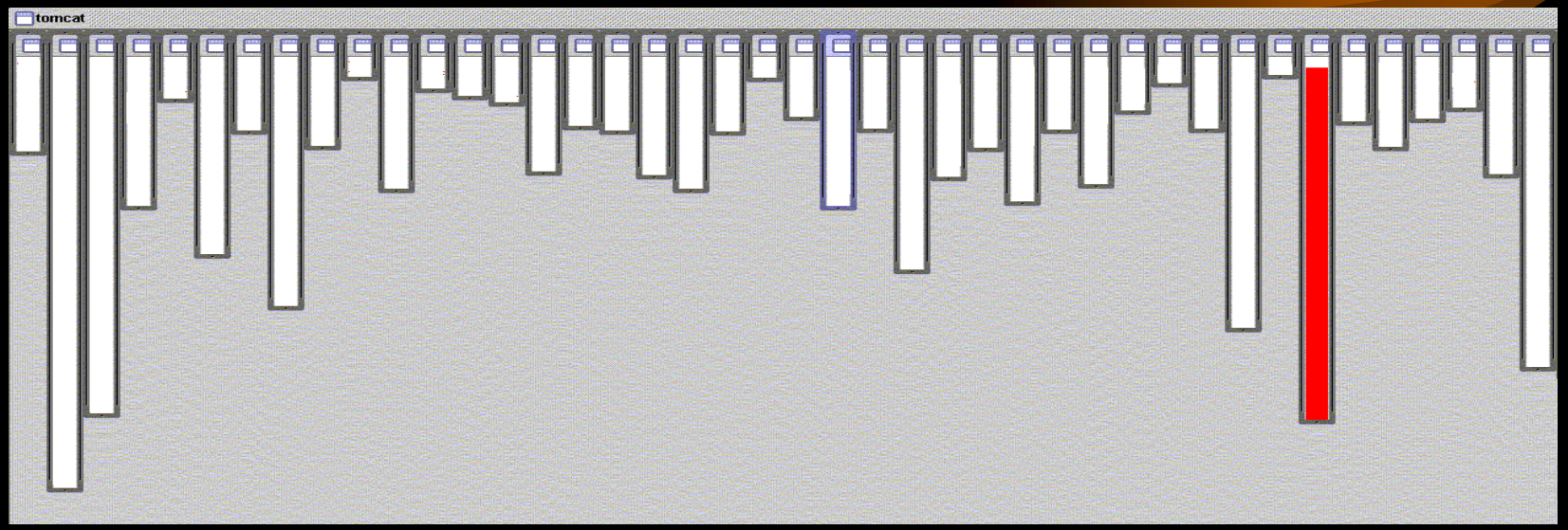


- “Divide and conquer”
- Stichwort: “Separation of Concerns”
- Jeder Concern soll in einem eigenem Modul, Klasse Funktion, etc. gekapselt werden

Was ist ein Concern?

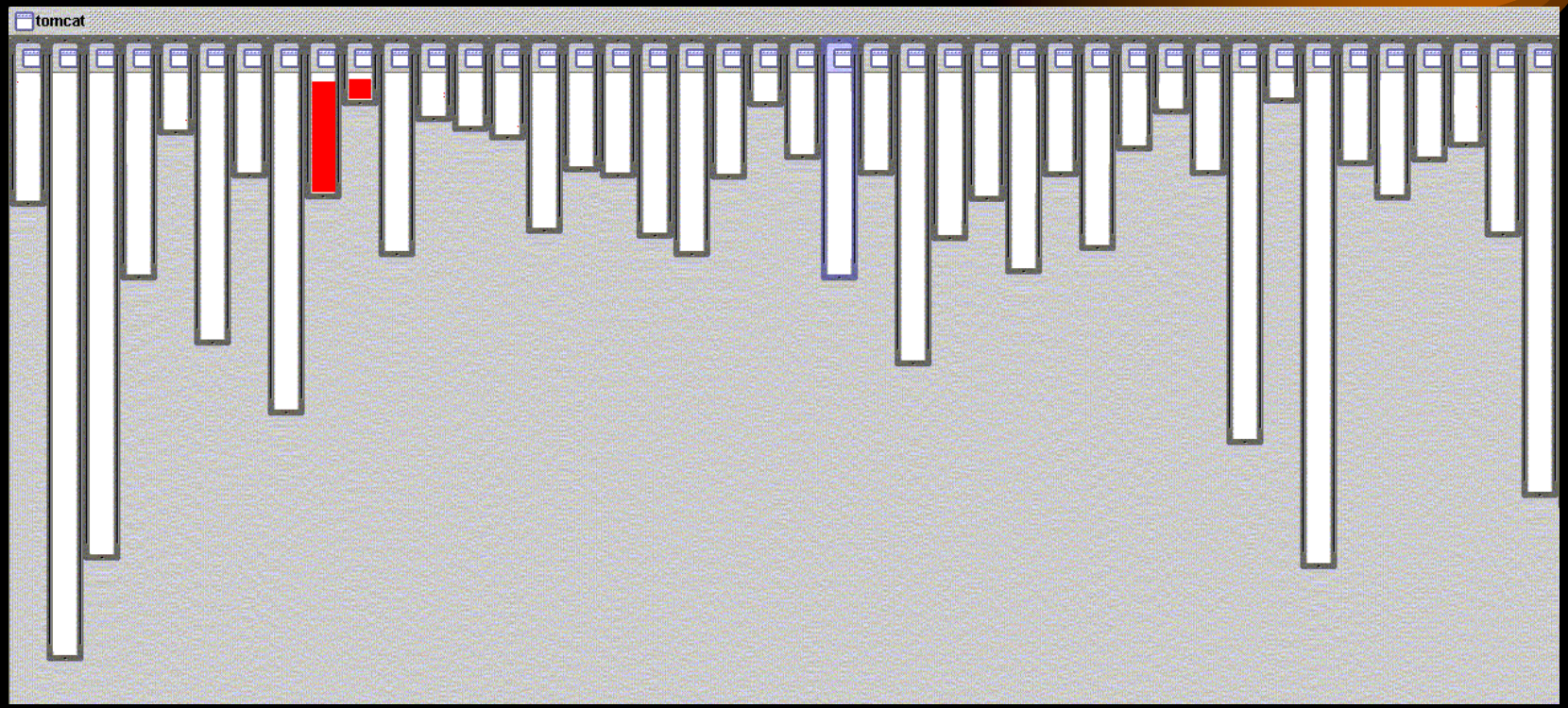
- Ein bestimmtes Ziel, Konzept oder Interessensgebiet
- Ein Softwaresystem besteht aus:
  - Fachlichen Concerns (Geschäftslogik)
  - Nicht Fachlichen Concerns

Gute Modularität XML-parsing



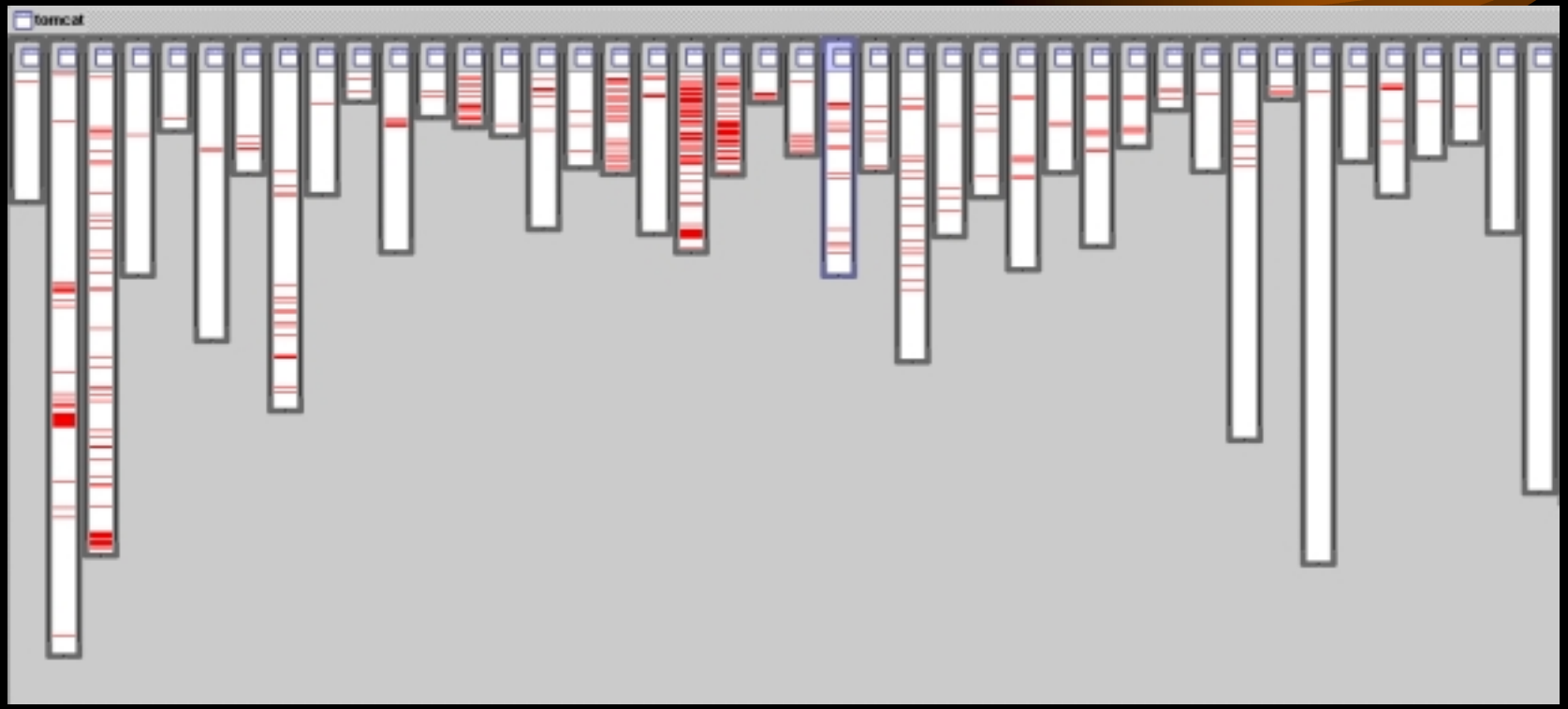
- XML parsing in org.apache.tomcat
  - –Rot zeigt die relevanten Code Stellen
  - –Passt wunderbar in ein Modul

Und bei: URL-pattern matching



- URL pattern matching in org.apache.tomcat

Logging ist nicht modularisiert



- Wo ist logging in org.apache.tomcat

Das Bankkonto-Beispiel

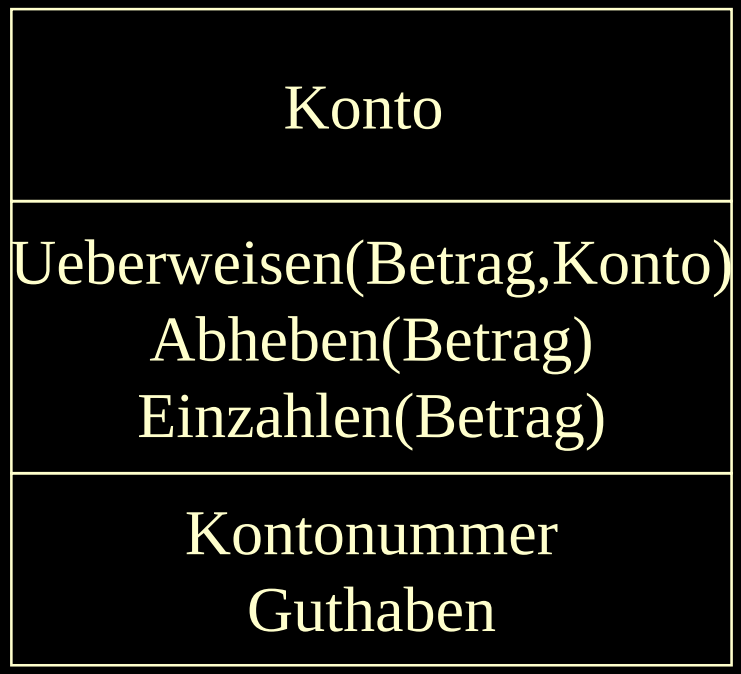
Fachliche Concerns:

- Überweisung, Einzahlung, Abhebung, ...
- Gesperrtes Konto
- Bonitätsprüfung, Kundenstatus, etc.

Nicht fachliche Concerns:

- Logging
- Transaction integrity
- Authorisation
- Security
- Performance, etc.

Klasse Konto



- Einfache Klasse
- „Drei“ fachliche Methoden ueberweisen, abheben einzahlen
- Attribute Kontonummer, Guthaben

Im Source-Code

```
public class Konto {

   String Kontonummer;
   double Guthaben;

    public void abheben(float Betrag) {
      Guthaben=Guthaben-Betrag;
   }

   public void einzahlen(float Betrag) {
      Guthaben=Guthaben+Betrag;
   }

   public void ueberweisen(float Betrag,Konto AnderesKonto) {
      Guthaben=Guthaben-Betrag;
      AnderesKonto.einzahlen(Betrag);
   }
}
```

Mit fachlichen Concerns

```
private void inlog_uberweisen(float Betrag,Konto AnderesKonto)   throws
            NichtMoeglich{
      if (Betrag<=0)
         throw new NichtMoeglich(" Programfehler, negativer Betrag“);
      if (AnderesKonto==null)
         throw new NichtMoeglich(" Programfehler, Konto==null“);
      Guthaben=Guthaben-Betrag;
      AnderesKonto.einzahlen(Betrag);
      checkIstMoeglich();

}
```

2,5 mal soviel Zeilen, aber immerhin noch halbwegs lesbar

Mit Logging

```
public void ueberweisen(float Betrag,Konto AnderesKonto)throws
    Logger.log("ueberweisen called with"+Betrag+" "+Ander
    if (Betrag<=0){
      NichtMoeglich ex=new NichtMoeglich(" Progr
      Logger.log("Exiting ueberweisen with Exc
        throw ex;
    }
    if (AnderesKonto==null){
      NichtMoeglich ex=new N                    null“);
      Logger.log("Exiting
        throw ex;
    }
    Guthabe
    try{

    }
      L                weisen with Exception "+ex);
        th
    }catch(            eption ex){
      Logger           Exiting ueberweisen with Exception "+ex);
        throw ex;
    }
    Logger.log("Exiting ueberweisen normally");
}
```

Lines of Code nochmal verdoppelt

Unakzeptabel!!!!

Martin Fowler:
Refactoring!

Nach dem Refactoring (1)

```
private void real_uberweisen(float Betrag,Konto AnderesKonto)throws NichtMoeglich{
      Guthaben=Guthaben-Betrag;
      AnderesKonto.einzahlen(Betrag);
   }

   private void condition_uberweisen(float Betrag,Konto AnderesKonto)throws
     NichtMoeglich{
      if (Betrag<=0)
         throw new NichtMoeglich(" Programfehler, negativer Betrag");
      if (AnderesKonto==null)
         throw new NichtMoeglich(" Programfehler, Konto==null");
      real_uberweisen(Betrag,AnderesKonto);
      checkIstMoeglich();
   }
```

Nach dem Refactoring (2)

```
public void uberweisen(float Betrag, K                    NichtMoeglich{
   Logger.log("ueberweisen called
   boolean mustCloseTransac                                tTransaction();
   try{
      condition_ub
   }catch(Nic
      Lo
   }
   Lo
}
```

Lesbarer!, aber:
Lines of Code versechsfacht
Anzahl der Methoden verdreifacht
Mind. ¾ des Codes ist redundant
Unakzeptabel!!!!

Folgen

Solcher Code ist mein Alptraum

- Schlecht Lesbar
- Geringe Produktivität
- Geringe Code Wiederverwendung
- Geringe Codequalität
- Schwierige Code-Evolution

Eigentlich wollten wir doch nur...(1)

- Logging:
  - Bevor eine Methode aufgerufen wird, gib ihre Signatur und Parameter aus
  - Wenn eine Methode ordentlich terminiert, gib die Signatur aus
  - Wenn eine Methode mit einer Exception terminiert, gib die Signatur und die Methode aus

Eigentlich wollten wir doch nur...(2)

- Business Concern:
  - Bevor eine Methode mit einem float Parametern aufgerufen wird, prüfe ob der Parameter >0 ist
  - Bevor eine Methode mit einem Konto als Parameter aufgerufen wird, prüfe ob der Parameter !=null ist
  - Nach Beendigung einer Methode teste ob das Objekt noch seine Invarianten einhält

...,dann mach´mas doch so (1)

- Logging:

```
before():execution(* Konto.*(..)){
   Logger.logentry(thisJoinPoint, thisJoinPoint.getArgs());
}

after() returning : execution(* Konto.*(..)){
   Logger.logexit(thisJoinPoint);
}

after() throwing (Throwable ex):execution(* Konto.*(..)){
   Logger.logexit(thisJoinPoint,ex);
}
```

...,dann mach´mas doch so (2)

- Business Concern :

```
before(float Betrag):execution(* Konto.*(..))&&args(..,Betrag,..){
    if (Betrag<0)
        throw new NichtMoeglich(" Programfehler, negativen Betrag unzulässig");
 }

 before(Konto konto):execution(* Konto.*(..))&&args(..,konto){
    if (konto==null)
        throw new NichtMoeglich(" Programfehler, Konto==null unzulässig ");
 }

 after(Konto konto) returning: execution(public * Konto.*(..))&&target(konto){
    konto.checkIstMoeglich();
 }
```

Wie geht das?

- Wir betrachten wichtige (Zeit)Punkte (Ereignisse) im Ablaufverhalten des Programms (Joinpoints)
- Wir suchen uns, die für uns relevanten Joinpoints mit einer Abfragesprache aus (pointcuts)
- Wir sagen: (advice)
  - (Bevor) Wann immer und wo immer ein bestimmtes Ereignis eintritt, mache zuerst ...,
  - (nachdem) Wann immer und wo immer ein bestimmtes Ereignis eintrat, mache dann ...
  - (Anstatt) Wann immer und wo immer ein bestimmtes Ereignis eintreten soll, mache stattdessen ...

Joinpoints

- Das Aufrufen einer Methode
- Das Ausführen einer Methode
- Zugriff auf eine Variable
- Das Behandeln einer Exception
- Die Initialisierung einer Klasse
- Die Initialisierung eines Objects

Primitive pointcuts

Ein Pointcut ist eine Art Abfrageprädikat über Join Points:

- Auf das ein gegebener Join Point passt oder nicht
- Das optional Werte aus dem Join Point veröffentlichen kann.
- Wildcards sind möglich

execution(public void Konto.*(double))

Passt auf die Ausführung aller Methoden mit dieser Signatur

Pointcut Composition

- Pointcuts können mit den logischen Operatoren ||, &&, ! verknüpft werden

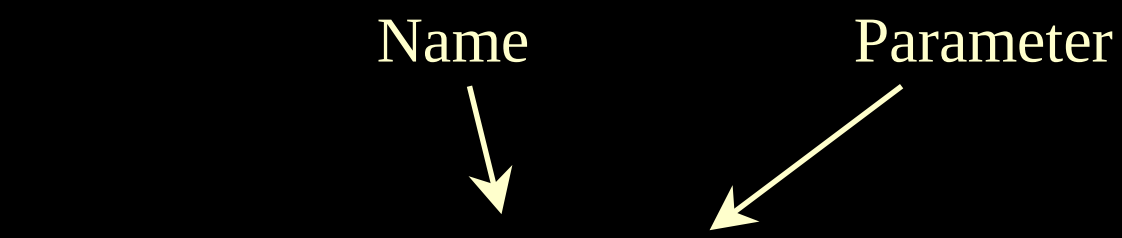
execution(public void Konto.*(double))||

execution(public void OtherClass.*(..))

Wann immer (public void Konto.*(double)) oder (public void OtherClass.*(..)) ausgeführt wird.

Benutzerdefinierte Pointcuts

- Benutzerdefinierte Pointcuts (auch bekannt als named Pointcuts) können genauso verwendet werden, wie primitive Pointcuts




```
pointcut mustValidate(  double Betrag):
        execution( public void Konto.*(double)) &&args(Betrag);
```

Before advice

- Aktion wird vor dem ursprünglichem Codestück am Joinpoint ausgeführt.

```
before():execution(* *.*(..)){
      Tracer.trace(thisJoinPoint);
}
```

Wann immer eine Methode ausgeführt werden soll, schreibe zuvor den Joinpoint in das Tracefile

After advice

- Aktion wird nach dem ursprünglichem Codestück am Joinpoint ausgeführt

```
after():execution(* *.*(..)){
     Tracer.trace(thisJoinPoint);
}
```

Wann immer eine Methode ausgeführt wurde, schreibe danach den Joinpoint in das Tracefile

Around Advice

- Anstelle des Codes an dem Join Point führe die Aktion aus

```
Object around():execution(* *.*(..)){
      Tracer.trace(„before “+thisJoinPoint);
      Object toreturn=proceed();
      Tracer.trace(„after “+thisJoinPoint);
      return toreturn;
}
```

Ruft den ursprünglichen Code auf

Wann immer eine Methode ausgeführt werden soll, trace den Joinpoint, führe dann die Methode aus und trace ihn dann wieder

Aspekt (1)

- Um die neuen Konzepte modular anwenden zu können, existiert eine Art Container namens Aspekt.
- Aspekte sind Klassen sehr ähnlich
- Aspekte verhalten sich im allgemeinen zu Klassen, wie Klassen zu structs in C++

Aspekte (2)

- Können nur vom Laufzeitsystem instanziert werden
- Können Methoden und Variablen und inner classes haben,
- Können advices, und pointcuts enthalten
- Können interfaces implementieren
- Können von Klassen abgeleitet werden

Was fehlt noch?

- Die Methode checkIstMoeglich() gehört irgendwie zur Klasse Konto, wird aber nur vom Aspekt benutzt.
- Irgendwie sollte daher der Aspekt der Klasse Konto, die Methode „zur Verfügung stellen“.

Introduction (aka Open Classes)

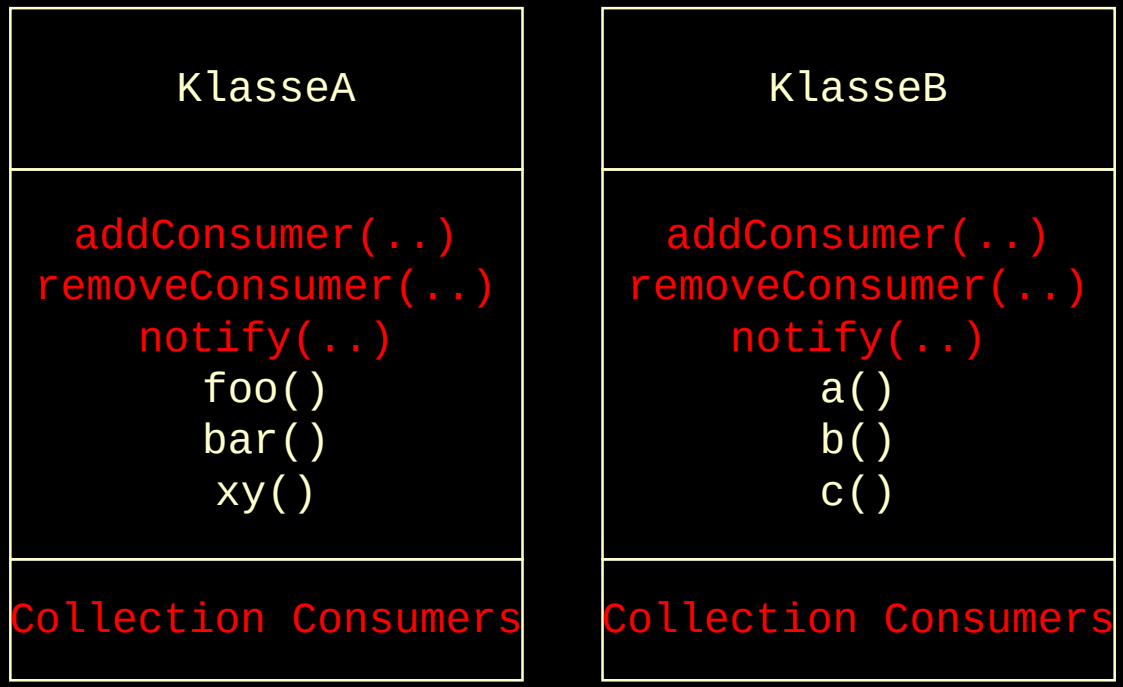
- Mit Introduction ist es einem Aspekt möglich andere Aspekte, Interfaces oder Klassen um Funktionalität zu erweitern.
- Hinzugefügt werden können:
  - Variablen,
  - Methoden
  - Interfaces
  - Vaterklassen, (nur bei einer Ableitung von Object)
- Bewährtes Konzept, z.B. von Python oder von der Subjektbasierten Programmierung

Container Introduction

- Ein Interface wird mit Methoden und Variablen erweitert.
- Das Interface wird einem Aspekt oder Klasse hinzugefügt
- Anwendung z.B., Listenernotification
- Aka Mixins

Beispiel Listener Notification (1)

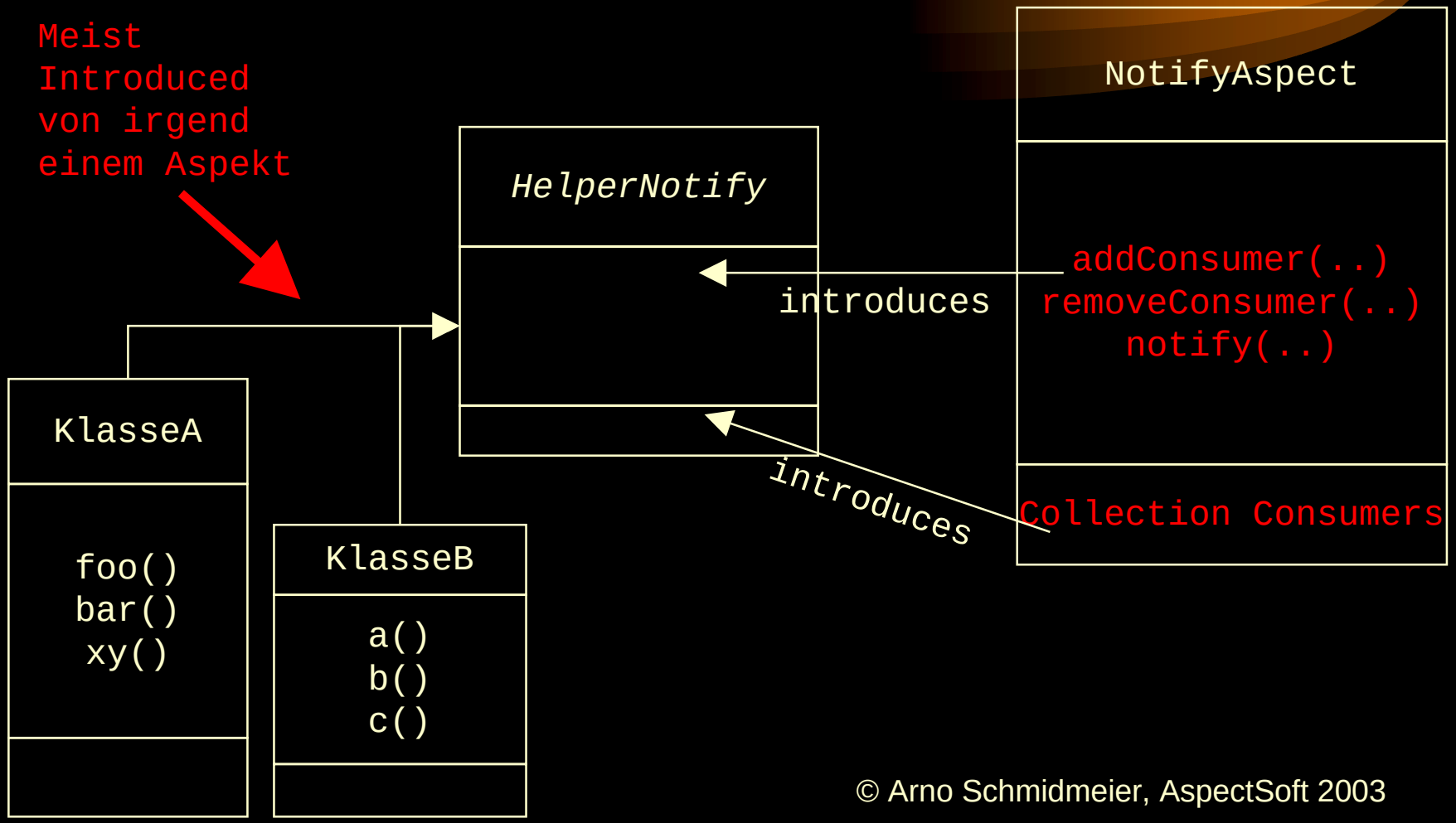
- Zwei Klassen, die beide die Listener notification realisieren müssen



Beispiel: Listener Notification (2)

Problem wir haben noch immer redundanten nicht modularisierten Code, wir haben viele Aspekte mit redundanten Code...

Beispiel: Listener Notification (3)



Declare error / declare warning

- Das Java Typssystem ist suboptimal
- Viel zu viele Methoden müssen als public deklariert werden, und diese werden zur falschen Zeit, vom falschem Ort vom falschem Modul aufgerufen.
- Beispiel:
  - closed subsystem
  - Kooperierende abstrakten Klassen

Declare warning/error (2)

- Syntax:
  - declare error: <pointcut> : <„Fehlertext“>
  - declare warning: <pointcut> : <„Fehlertext“>

Wo immer eine Methode aus dem sql package aufgerufen wird, erzeuge einen compile Fehler, außer im Paket sqladapter

```
declare warning:
        call(* java*.sql.*.*(..))&&
      && ! within(com.aspectsoft.sqladapter.*):
  "JDBC must not be used directly“
```

AspectJ (1)

- Basiert auf dynamic join points von Java
  - “Zeitpunkte im Ablauf” eines Java Programms
- Fügt „fünf kleine“ Erweiterungen hinzu
  - pointcuts
    - Anfragesprache über Join Points
  - advice
    - (Zusätzliche) Aktionen an den Join Points eines Pointcuts
  - “open classes”
  - Declare error, declare warning
  - aspect
    - Eine Modulare Einheit für überschneidende Funktionalitäten

AspectJ (2)

- Die führende AOP-Programmiersprache
- basiert auf der Java Plattform
- Vollständig abwärtskompatibel zu Java
- Ursprünglich entwickelt von PARC, jetzt eclipse
- Open Source
- Support für/von populäre IDEs (eclipse, JBuilder, Forte, JDE, emacs, ...)
- Erfolgreich in diversen large Scale-Projekten eingesetzt

Erfahrungen

Arno Schmidmeier

AspectSoft

+49/9151/90 50 30

A@Schmidmeier.org

Erfahrungen

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

Codereduktion

- Redundanter Code konnte modularisiert werden.
- Realisierte Codereduktion 30-95%
  - Abhängig vom Anwendungsfall
  - Fähigkeiten der Entwickler
- Serverseitige Komponenten 90%
- EAI-Szenarien: 95%
- GUI-Swing Code: ca. 50% auf den manuell erzeugten Anteil

Wartungskosten

- Änderungen im Tangling Code können zentral modifiziert werden.
- Nicht funktionale Anforderungen werden einheitlich implementiert und können wiederverwendet werden
- Unterstützender Code für Wartungsarbeiten lässt sich günstig realisieren bzw. wiederverwenden, Tracing, Logging, Profiling, Debug-APIs, etc.

Entwicklungszeit

- Zwei Extreme (für Neuentwicklungen):
  - 20-40% schneller fertig bei gleicher Qualität oder
  - Die doppelte Qualität in der gleichen Zeit
- Nicht funktionaler Code wird implementiert, bzw. kann bei mit deutlich weniger Aufwand realisiert werden
- Unterstützung des Entwicklers durch
- ->Konsequentes Exceptionhandling
- ->gutes Tracing
- ->einfaches Profiling, etc.

Codequalität

- Die Klassenstruktur von AspectJ-Programmen richtet sich viel besser nach der Struktur des Anwendungsbereichs anstelle von technischen Strukturen
- Methoden und Klassen werden „entrümpelt“, und sind damit deutlich einfacher lesbar

Verbesserung der Architektur (1)

- Es stehen bessere Muster zur Verfügung
- Viele existierende Muster können effizienter implementiert werden bzw. werden zu Bibliotheken degradiert
- Neue Architekturen werden möglich. (z.B. virtueller, geteilter interner Nachrichtenbus)

Verbesserung der Architektur (2)

- Die Einhaltung vieler Architekturbedingungen kann durch Aspekte zur Compilezeit überprüft werden
- Circular Dependencies können vollständig eliminiert werden.

Verbesserung der Performance

- Viele „Flexibilitätslayer“ für künftige Erweiterungen können eliminiert werden. (z.B. setter und getter)
- Oft: Die Kommunikation erfolgt nur noch virtuell indirekt, physikalisch direkt
- Oft: Einsatz von effizienteren Pattern

Einführung

- Top-Down (mit Unterstützung des Managements)
- Bottom up (don´t tell the manager)

Businesscase

- Zwei Extreme (für Neuentwicklungen):
  - 20-40% schneller fertig bei gleicher Qualität oder
  - Die „doppelte“ Qualität in der gleichen Zeit

Erfahrungswert von Sirius, (von Ende 2001)

Adaptionsaufwand

- Ungefähr so groß wie von Prozedural zu OO
- Aber:
  - Er wird nicht kleiner
  - Er muß irgendwann einmal durchgeführt werden
  - Also, warum nicht gleich jetzt?

- Beratenes Team (bei Sirius Software)
  - 3 monatiges Projekt, trotz Adaption eine Woche früher fertig
  - Bei besser Qualität -> geringere Wartungskosten
  - Erhöhte Wiederverwendung

Beispiel Adaptionsaufwand (betreuter Entwickler)

- 3 mal vergleichbares Projekt, 3 mal der gleiche Entwickler:
- Projekt 1: Plain Java+Vitria 1 Monat
- Projekt 2: Plain Java+Vitria +AspectJ
  - 2,5 Wochen
- Projekt 3: Plain Java+Vitria +AspectJ+Wiederverwendete Library aus Projekt 2, 3 Tage

Adaptionsstrategien

- „Fucked up beyond any recovery“-Ansatz (nur mit externer Hilfe)
- Businesscase driven (externe Hilfe zumindest für den Businesscase empfohlen)
- „Don´t tell the manager“
- Über die QA-Schiene

Arno Schmidmeier

AspectSoft

+49/9151/90 50 30

A@Schmidmeier.org