

AOP by Examples



3. August 2011



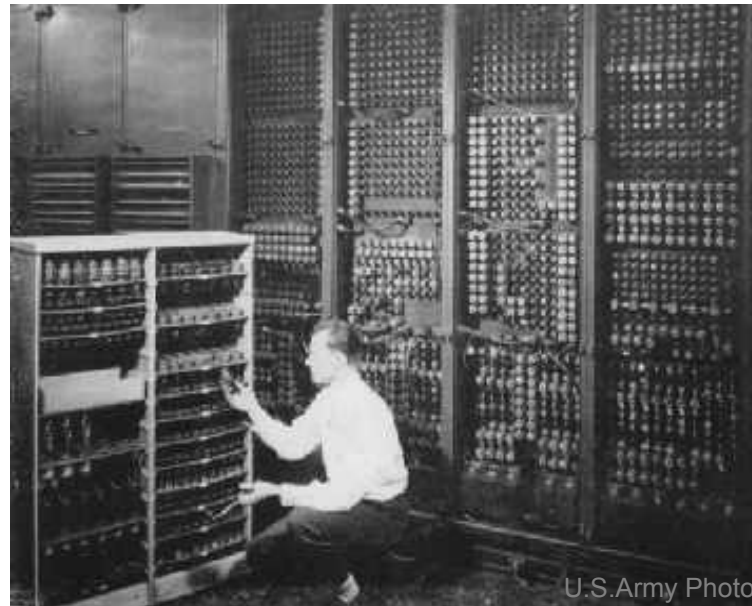
JUG Karlsruhe

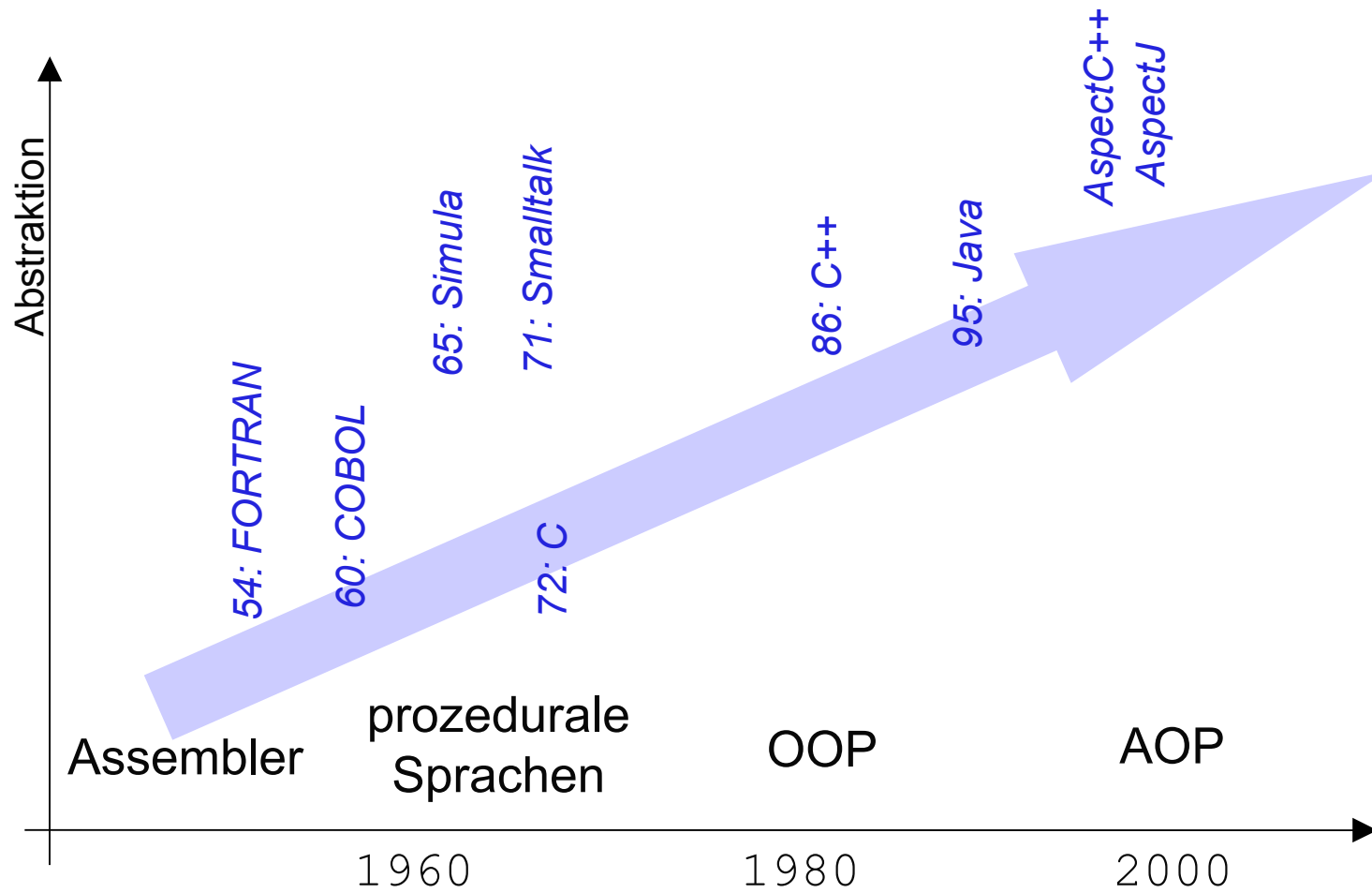
jug-karlsruhe.de | [twitter.com/@jugka](https://twitter.com/jugka)

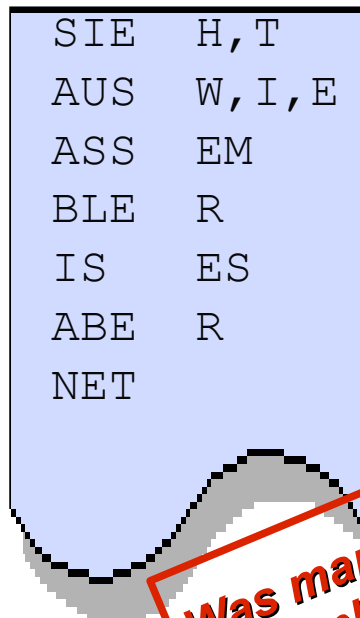
oliver.boehm@agentes.de

- ☛ **von Assembler bis AspectJ**
- ☛ **Hello-World-Beispiel**
- ☛ **Begrifflichkeiten**
- ☛ **noch mehr Beispiele**
- ☛ **Was ist neu in AspectJ 1.6?**
- ☛ **Zusammenfassung**

von Assembler bis AspectJ







**Was man nicht in
Assembler machen kann,
muss man löten!**

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

▪ Bsp: Backanleitung

Zweckschgen Kuchens

Mürbeteig: 250gr. Mehl, $\frac{1}{2}$ Backpulver
80gr. Fett, 80gr. Zucker

1 Vanillezucker, 1 Prise Salz, 2 Eier

Belag: 1 kg Zweckschgen, 2 Eßl. Hagelzucker

Aus den Zutaten einen Mürbeteig herstellen und
1 Stunde kaltstellen.

Zweckschgen aufsteinen und einschneiden.

Rundes Backblech beschmieren und mit Teig
auslegen. Den Teigboden mehrmals mit einer
Gabel einstechen und mit Semmelbrösel bestreuen.

Zweckschgen rosettenförmig legen.

Im Backofen 20-30 Minuten backen bei 180-200°C.

Nach dem Backen mit Hagelzucker bestreuen.

PASCAL
C COBOL
FORTRAN

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

Ruby

SIMULA
C++ Python
Java Eiffel
Smalltalk



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

Kuchen
Zutaten Größe
add backe



- = OOPS, um Aspekte angereichert, z.B.
- Logging-Aspekt
 - Security-Aspekt
 - Transaktion-Aspekt
 - ...

AspectJ AspectS
 AspectC++



<http://www.flickr.com/photos/oskay/472097903/>

fachliche Concerns

- **Einzahlung**
- **Auszahlung**
- **Überweisung**
- **Bonitätsprüfung**
- ...

nichtfachliche Concerns

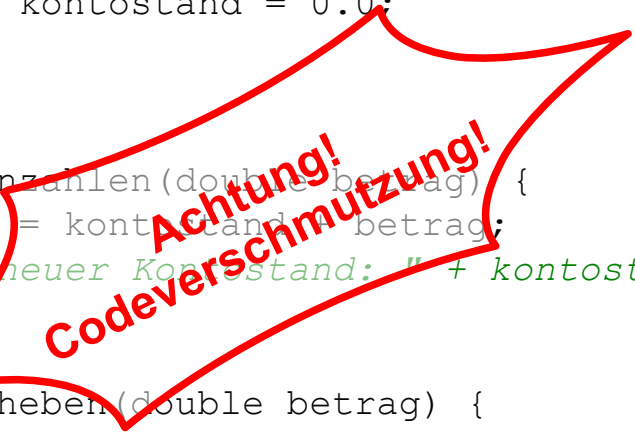
- **Logging**
- **Performance**
- **Authorisierung**
- **Sicherheit**
- ...



```
public class Konto {  
  
    private double kontostand = 0.0;  
  
    public double abfragen() {  
        return kontostand;  
    }  
  
    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);  
    }  
  
}
```

Konto
Kontostand
abfragen einzahlen abheben ueberweisen

```
public class Konto {  
  
    private static Logger log = Logger.getLogger(Konto.class)  
    private double kontostand = 0.0;  
  
    ...  
  
    public void einzahlen(double betrag) {  
        kontostand = kontostand + betrag;  
        log.info("neuer Kontostand: " + kontostand);  
    }  
  
    public void abheben(double betrag) {  
        kontostand = kontostand - betrag;  
        log.info("neuer Kontostand: " + kontostand);  
    }  
  
    ...  
  
}
```

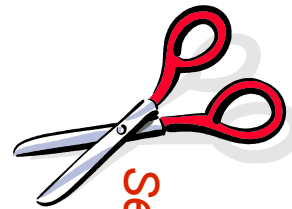


neue
Anforderung:

*alle Konto-
Bewegungen
müssen
protokolliert
werden!*

```
public aspect LogAspect {  
  
    private static Logger log = Logger.getLogger(LogAspect.class);  
  
    after(double neu) : set(double Konto.kontostand) && args(neu) {  
        log.info("neuer Kontostand: " + neu);  
    }  
  
}
```

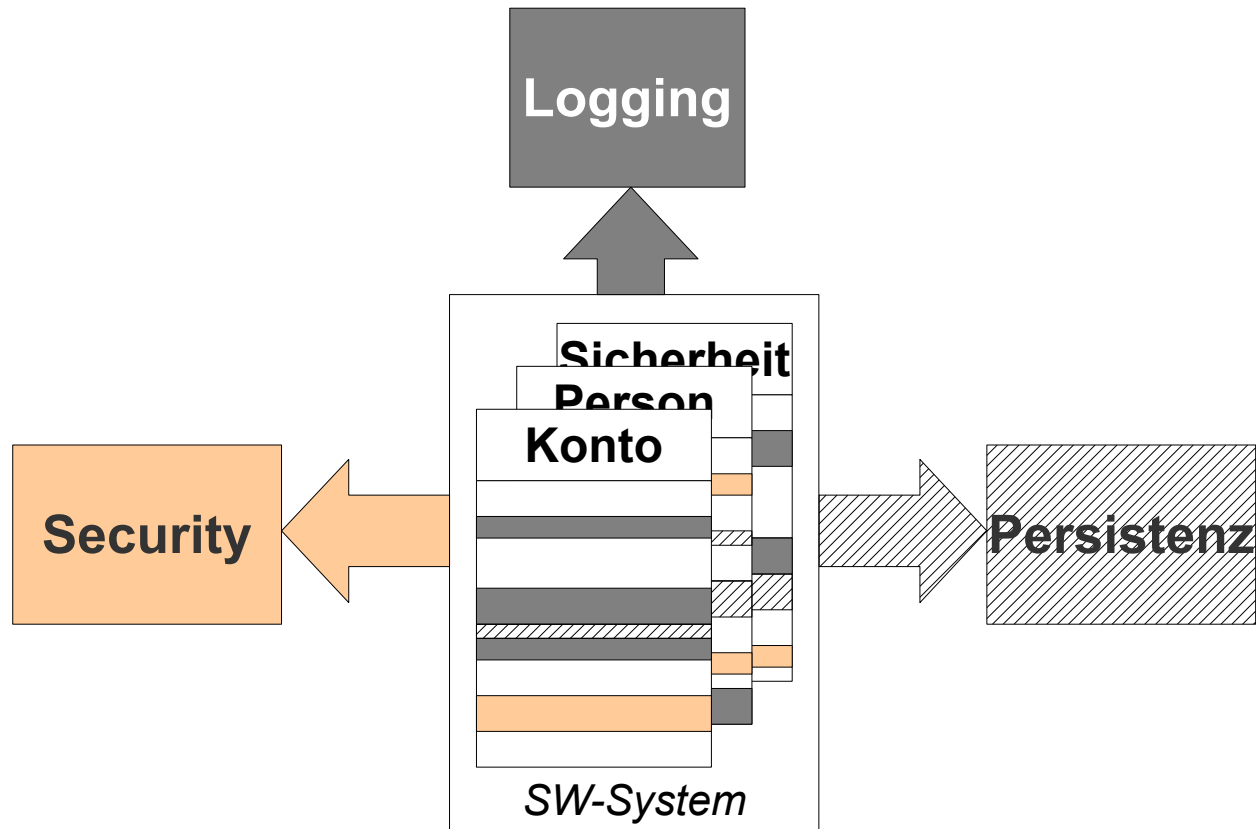
Konto
Kontostand
abfragen einzahlen abheben ueberweisen



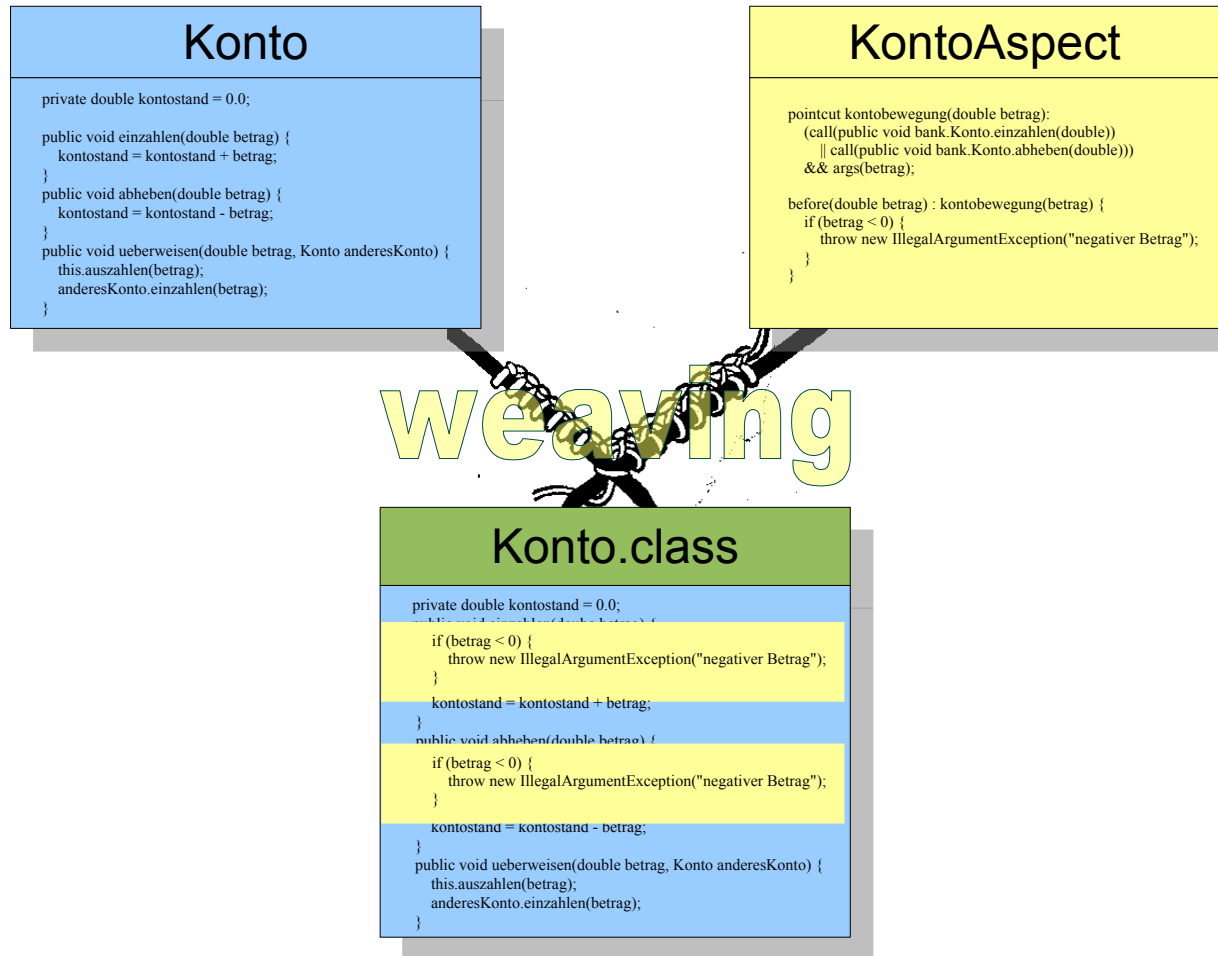
Separation of Concerns

LogAspect

Do you think in AOP?



Das System als Menge
von "Concerns"



Do you speak Aschbegt- Dschei?

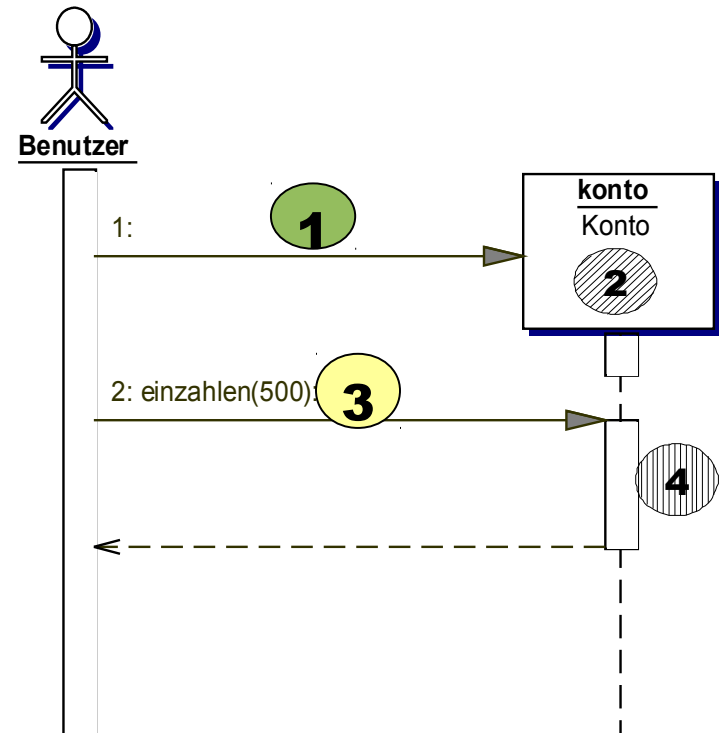


- **Joinpoint**
 - Punkte im Programm, an denen Code erweitert oder modifiziert werden soll
- **Pointcut**
 - eine Auswahl von Joinpoints
- **Advice**
 - der Code für den Pointcut
- **Introduction**
 - Erweiterung anderer Klassen, Interfaces, Aspekte um zusätzliche Funktionalität
- **Aspect**
 - Konstrukt,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

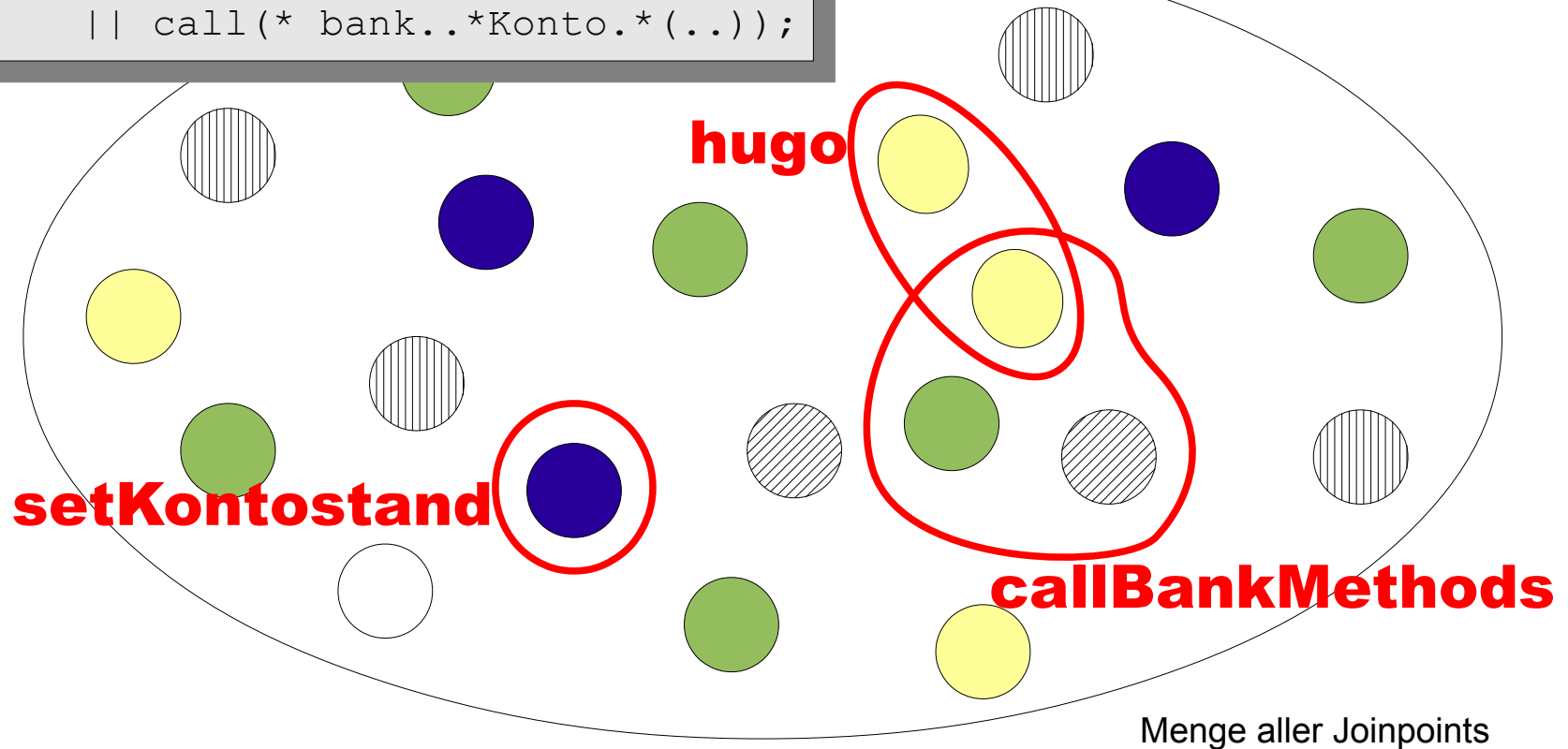
- Aufrufen einer Methode
- Ausführen einer Methode
- Zugriff auf eine Variable
- Behandeln einer Exception
- Initialisierung einer Klasse
- Initialisierung eines Objekts

```
Konto konto = new Konto();  
konto.einzahlen(500.0);
```

- (1) Konstruktor aufrufen
- (2) Objekt initialisieren
- (3) Methode aufrufen
- (4) Methode ausführen



```
pointcut setKontostand() :  
    set(double bank.Konto.kontostand);  
  
pointcut callBankMethods() :  
    call(* bank.*Konto.*(*)  
        || call(* bank..*Konto.*(..));
```



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

pointcut **accessKontostand()** :

Name

```
get(private double bank.Konto.kontostand)  
|| set(private double bank.Konto.kontostand);
```

- **Code, der eingewebt wird**
 - before()
 - after()
 - around()
- **Ähnlichkeit mit Methoden**

```
after() : callBankMethods() {  
    log.debug("TEST: " + thisJoinPoint);  
}
```

```
TEST: call(void bank.Konto.einzahlen(double))  
TEST: call(double bank.Konto.abfragen())  
...
```

- **Aktion wird vor dem ursprünglichen Joinpoint ausgeführt**

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

für alle public-Methode gib den Joinpoint aus

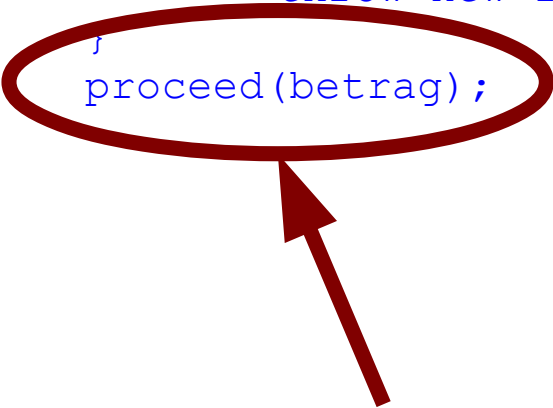
- Aktion wird *nach* dem eigentlichen Joinpoint ausgeführt

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

überprüfe nach jeder Konto-Methode den Kontostand

- Aktion wird *anstelle* des Codes ausgeführt

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



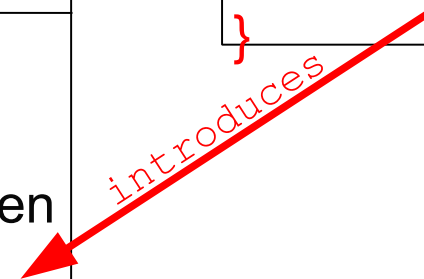
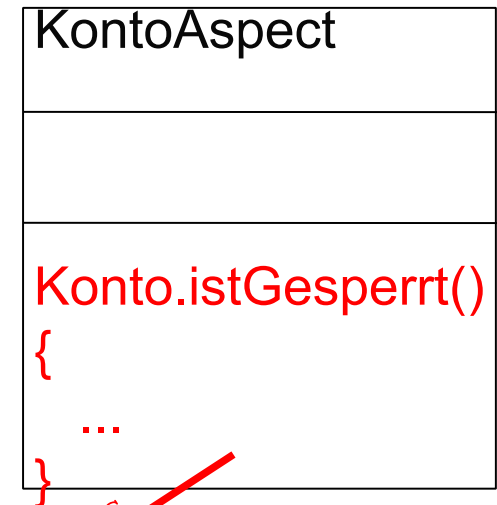
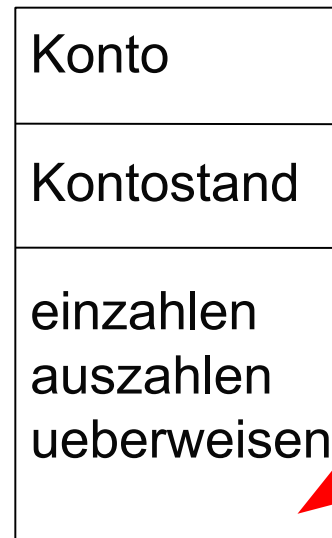
*überprüfe für die zahlen-Methoden den Parameter
und rufe danach die eigentliche Methode auf*

ruft den ursprünglichen Code auf

- **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**

```
aspect WorldAspect {  
  
    /**  
     * Beispiel fuer einen benannten Pointcut ("mainOperation").  
     */  
    public pointcut mainOperation():  
        execution(public static void hello.World.main(String[]));  
  
    /**  
     * say good bye  
     */  
    after(): mainOperation() {  
        System.out.println("Good bye.");  
    }  
  
    /**  
     * Beispiel fuer einen anonymen Pointcut ("execution(..)")  
     */  
    after() : execution(public static void hello.World.main(String[])) {  
        System.out.println("Good bye.");  
    }  
  
}
```

- 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



weitere Beispiele



<http://www.flickr.com/photos/balakov/1544234007/>

- **kein Zugriff auf Kontostand**

```
pointcut accessKontostand() :  
    get(private double bank.Konto.kontostand)  
    || set(private double bank.Konto.kontostand);  
  
before(Konto konto) : accessKontostand() && this(konto) {  
    if (konto.isGesperrt()) {  
        log.warn("Zugriff verweigert (Konto gesperrt)");  
        throw new IllegalStateException("Konto ist gesperrt");  
    }  
}
```

- für gekennzeichnete Methoden ist ein Login erforderlich

```
public class Konto {  
    ...  
    @LoginRequired  
    public void abheben(double betrag) {  
        kontostand = kontostand - betrag;  
    }  
}  
  
pointcut loginRequiredExecutions() :  
    execution(@LoginRequired * bank.Konto.*(..));  
  
Object around() : loginRequiredExecutions() {  
    if (!AccessControl.loggedIn()) {  
        try {  
            AccessControl.login();  
        } catch (LoginException e) {  
            throw new AccessControlException("login failed");  
        }  
    }  
    return proceed();  
}
```

```
/** Marker Interface */
public interface Singleton {}

/** Es kann nur einen geben... */
public class Highlander implements Singleton {
    ...
    public aspect SingletonAspect {

        private Hashtable singletons = new Hashtable();

        pointcut selectSingletons() : call((Singleton +).new(..));

        Object around() : selectSingletons() {
            Class singleton =
                thisJoinPoint.getSignature().getDeclaringType();
            synchronized(singletons) {
                if (singletons.get(singleton) == null) {
                    singletons.put(singleton, proceed( ));
                }
            }
            return singletons.get(singleton);
        }
    }
}
```

Idee:
es wird immer das gleiche Objekt
bei new(..) zurückgegeben

- **Socket aufbauen (teuer):**
 - `Socket socket = new Socket(hostname, 80);`
- **Socket lesen / schreiben**
- **Socket freigeben**
 - `socket.close();`
- **Problem Freigabe:**
 - Hmm, vielleicht brauche ich den Socket noch
 - andererseits: nicht benötigte Ressourcen sollten freigegeben werden
- **Abhilfe**
 - Socket-Pooling
 - lohnt sich aber nur, wenn ich viele Verbindungen habe!!!
- **nächstes Problem:**
 - aber da muss ich ja alle Socket-Kreierungen abändern :-(


```
/**
 * Pointcut fuer das Oeffnen bzw. Anlegen eines Sockets
 */
pointcut socketCreation(String host, int port) :
    call(public Socket.new(String, int))
    && args(host, port)
    && !within(SocketPool);

/**
 * Pointcut fuer das Schliessen eines Sockets
 */
pointcut socketClose(Socket socket) :
    call(public void Socket.close())
    && target(socket)
    && !within(SocketPool);
```

```
/**
 * Das Oeffnen eines Sockets erfolgt nun ueber den socketPool.
 * Falls dies fehlschlaegt, wird der Original-Pointcut aufgerufen.
 */
Socket around(String host, int port)
    throws UnknownHostException, IOException
    : socketCreation(host, port) {
    Socket socket = socketPool.getSocket(host, port);
    if (socket == null) {
        return proceed(host, port);
    } else {
        return socket;
    }
}
```

- **Caching**
- **Better Exceptions**
 - PatternTesting Exception
 - autom. Behandlung (z.B. LazyInitialisationException)
- **Monitoring**
- **Workarounds**
 - Patches
 - Bibliotheken „erweitern“ (finale Klassen)
- **Testen / Mocking**
- **Validierung / Runtime-Checks**
 - nur gültige Parameter zulassen
 - PatternTesting Check-RT
- **Q**
 - PatternTesting Check-CT

- **Oliver Böhm**
Aspektorientierte Programmierung mit AspectJ 5
<http://www.dpunkt.de/buch/3-89864-330-1.html>
- **die Seite zum Buch**
<http://www.aosd.de>
- **Ramnivas Laddad**
AspectJ in Action
Manning Publications Co., 2003, ISBN 1-930110-93-6
- **PatternTesting**
<http://patterntesting.org/>



Was ist neu in AspectJ 6?



- **Unterstützung Java 6**
- verbesserte inkrementelle Kompilierung
- Bug-Fixing (Generics)
- Annotations jetzt auch für Parameter-Matching

- Am Anfang war der Assembler
 - danach: prozedural, OO
 - Problem: Querschnittsbelange (Crosscutting Concerns).
 - Lösung: AOP
 - AOP baut auf OOP aufsetzt auf OO / Java auf
 - AOP / AspectJ gewinnt an Bedeutung
 - Entwicklung wird effektiver
 - kleinere Programme möglich
- maßloser Einsatz kann Gegenteil bedeuten
 - Fallstricke lauern
 - Selbstdisziplin nötig
 - Lernkurve vergleichbar mit C -> C++

AspectJ = Java++





AspectJ

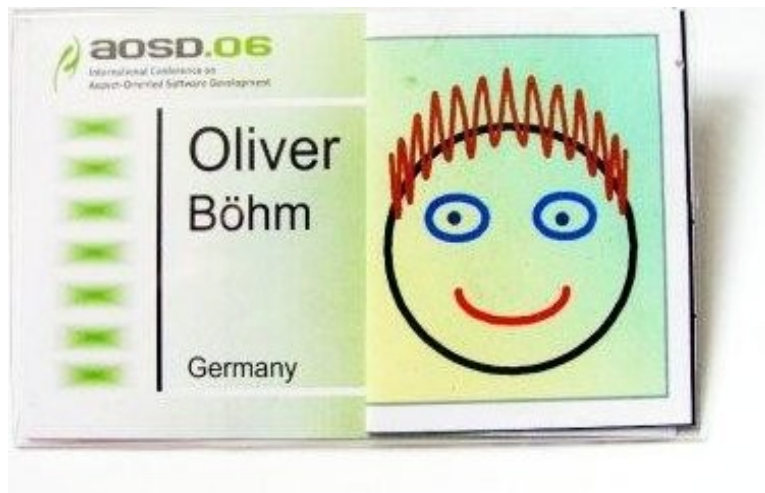
*Java ohne AspectJ ist wie
Linsen ohne Spätzle*

Java

- **1.9.2011: Relevantes schneller finden – mit Lucene und Solr**
 - Alte Scheuer
 - <http://jugs.org/2011-09-01.html>
- **23.3.2012: 10 Jahre PatternTesting**
 - Alte Scheuer
 - (geplant)
- **6.7.2012: Java Forum Stuttgart**
 - über 1000 Besucher
 - <http://www.java-forum-stuttgart.de/>



Vielen Dank



agentes GmbH

Oliver Böhm

oli.blogger.de

oliver.boehm@agentes.de

ob@jugs.org

Telefon 0711 / 25857 - 207

Telefax 0711 / 25857 - 299

Räpplenstraße 17

70191 Stuttgart