RESCO

Aspektorientierte Programmierung (AOP)

Beispiele mit AspectJ

Thomas Baustert
Resco GmbH
(Mai 2004)

Motivation

- AOP ist ein mächtiges „Werkzeug“ für den „Entwickler-Werkzeugkoffer“.
- AOP könnte zunehmend an Bedeutung gewinnen.
  - Kommerzielle Nutzung.
  - JBOSS 4.0, BEA 8.1 mit AOP Framework.
  - Immer mehr Artikel, Bücher, Vorträge.

Ziel

- Grundverständnis über AOP.
- Mächtigkeit und Möglichkeiten erkennen.
- Interesse wecken.

Inhalt

- Modularisierung von Software-Systemen und Grenzen heutiger Programmiersprachen.
- Einsatz der aspektorientierten Programmierung (Lösungsansatz).
- Beispiele mit der Java Erweiterung AspectJ.
- Fazit und Diskussion.

„It‘s all about modularization“

- Modularisierung von Verantwortung(-sbereichen) ist ein wesentliches Ziel bei der Entwicklung von Softwaresystemen („*Separation of Concerns*“).

- Beispiele:
  - N-Tier Applikationen: Frontend, Middletier, Backend.
  - MVC-Pattern: Model, View, Controler.
  - Framework.
  - (Abstrakte) Klasse.
  - Methode.

Concern: *dtsch.: Anliegen.*
Wird in AOP auch als Verantwortung übersetzt. Anliegen wird durch Klassen implementiert, die jeweils die Verantwortung tragen.

Vorteile durch Modularisierung

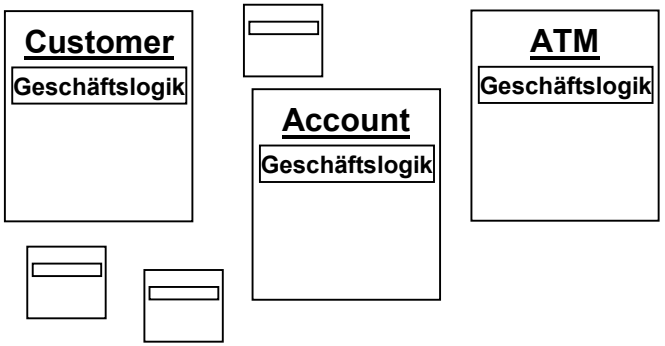
- Bessere Bewältigung der Komplexität eines Softwaresystems.
- Bessere Wiederverwendung von Teilen.
- Vermeidung von redundantem Quellcode.
- Parallele Entwicklung der getrennten Bereiche durch Experten.
- Höhere Produktivität, geringere Kosten.

Evolution der Modularisierung

- Monolithisches Assembler-Programm. Alle Verantwortlichkeiten in einem Programm (Modul).
- Prozedurale Programmierung modularisiert Verantwortlichkeiten in Prozeduren und Module.
- Objektorientierte Programmierung modularisiert Verantwortlichkeiten in Objekten (Klassen).
- *Aspektorientierte Programmierung modularisiert Verantwortlichkeiten in Aspekten.*

Core Concern (Kern-Anliegen/-Verantwortung)

- Jede Anwendung hat Kern-Anliegen/-Verantwortung (in AOP „Core Concerns“ genannt). Beispiel: Geschäftsanwendung: Geschäftslogik.
- Primäre Aufgabe des Entwicklers ist die Umsetzung der Kern-Verantwortung. Beispiel: Geschäftslogik.
- Modularisierung durch Aufteilung der Verantwortlichkeiten in Klassen. Unterstützung: Vererbung, Trennung Schnittstelle/Implementierung.
- Reale Welt (Geschäftsmodel) durch OO-Design gut abbildbar.

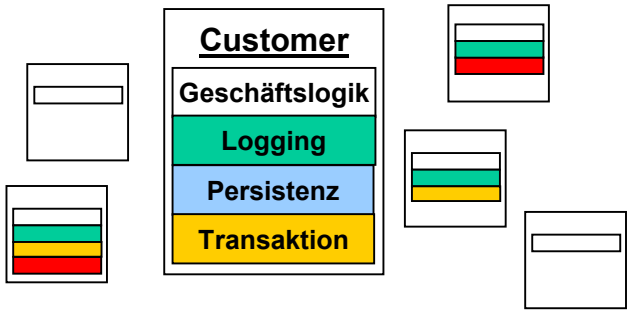


Crosscutting Concerns (Querschneidene Verantwortung)

- Weitere Aufgabe des Entwicklers ist die Umsetzung technischer Anliegen, z.B. Logging, Persistenz, Ausnahmebehandlung, usw.

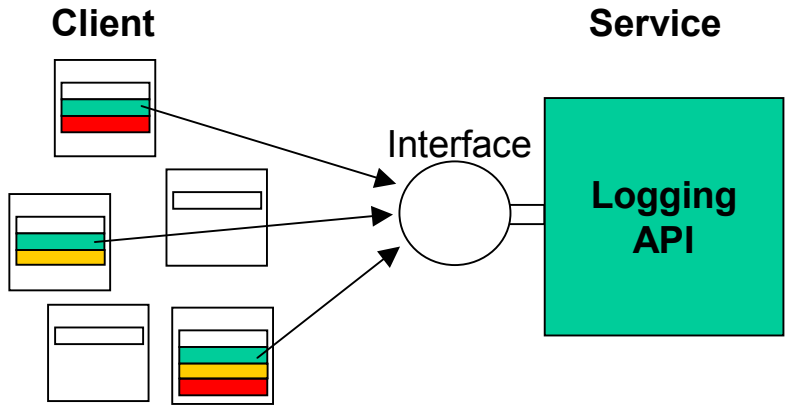
- Logging, Persistenz, usw. sind systemweit verteilt und werden in AOP daher „Crosscutting Concerns“ (querschneidende Verantwortlichkeiten) genannt.

- Crosscutting Concerns werden in Kern-Module (Klassen) der Geschäftslogik integriert.



Modularisierung Service und Client

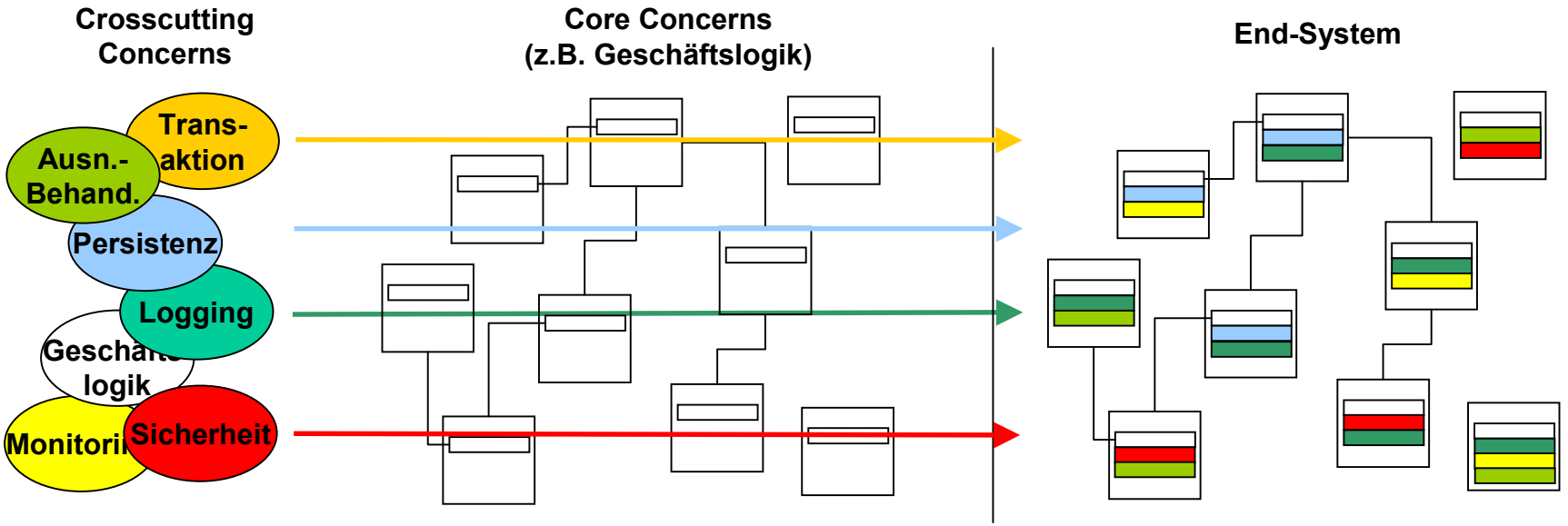
- Dienst/Service wird systemweit benötigt. Beispiel Logging API.
- Service-Teil wird in OO durch Trennung von Schnittstelle und Implementierung gut modularisiert. Austausch der Implementierung möglich.
- Client-Teil (Aufruf des Service) ist über Quellcode verstreut und nicht modularisiert. Schnittstelle statt Implementierung hilft nicht.



Modularisierung Crosscutting Concerns

- Crosscutting Concerns können mit heutigen Methoden und Programmiersprachen nicht modularisiert werden.

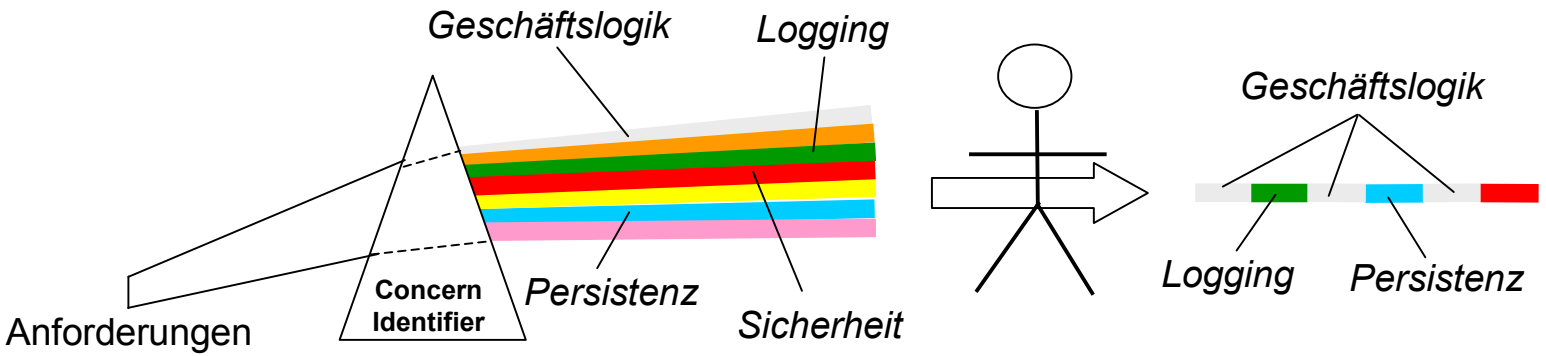
- Crosscutting Concerns sind über das System verteilt und in Klassen der Geschäftslogik integriert.



Design-Phase und Implementierung

- Aufteilung der individuellen Anforderungen in individuelle, sich gegenseitig nicht beeinflussende Verantwortungsbereiche während Design-Phase.

- Heutige Methoden/Sprachen ermöglichen nicht den Erhalt dieser Aufteilung während der Implementierung.

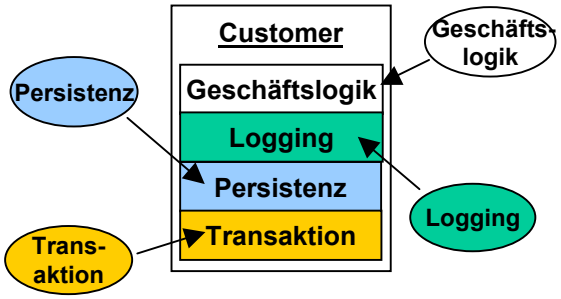


N-dimensionaler Verantwortungsraum

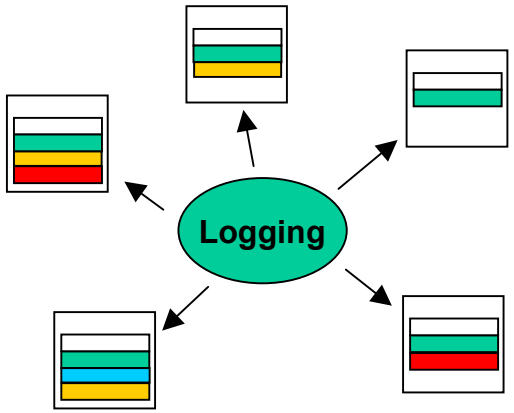
1-dimensionaler Implementierungsraum

Symptome der Crosscutting Concerns

- Code Tangling (Code-Vermischung):
  Eine Klasse ist für mehrere Anliegen verantwortlich.
  Beispiel: Geschäftslogik, Logging, Persistenz, Transaktion.

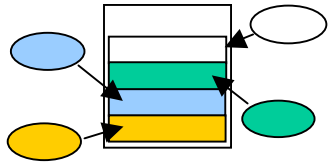


- Code Scattering (Code-Verstreuung):
  Ein Anliegen ist über mehrere Klassen verteilt.
  Beispiel: Logging, Sicherheit, Pooling.

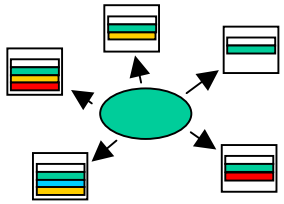


Nachteile durch Crosscutting Concerns

Durch Code-Vermischung und -Verstreuung:



- Sind Änderungen umfangreicher und fehleranfälliger.
- Ist Debugging, Wartung, Erweiterung, Refaktoring schwerer.



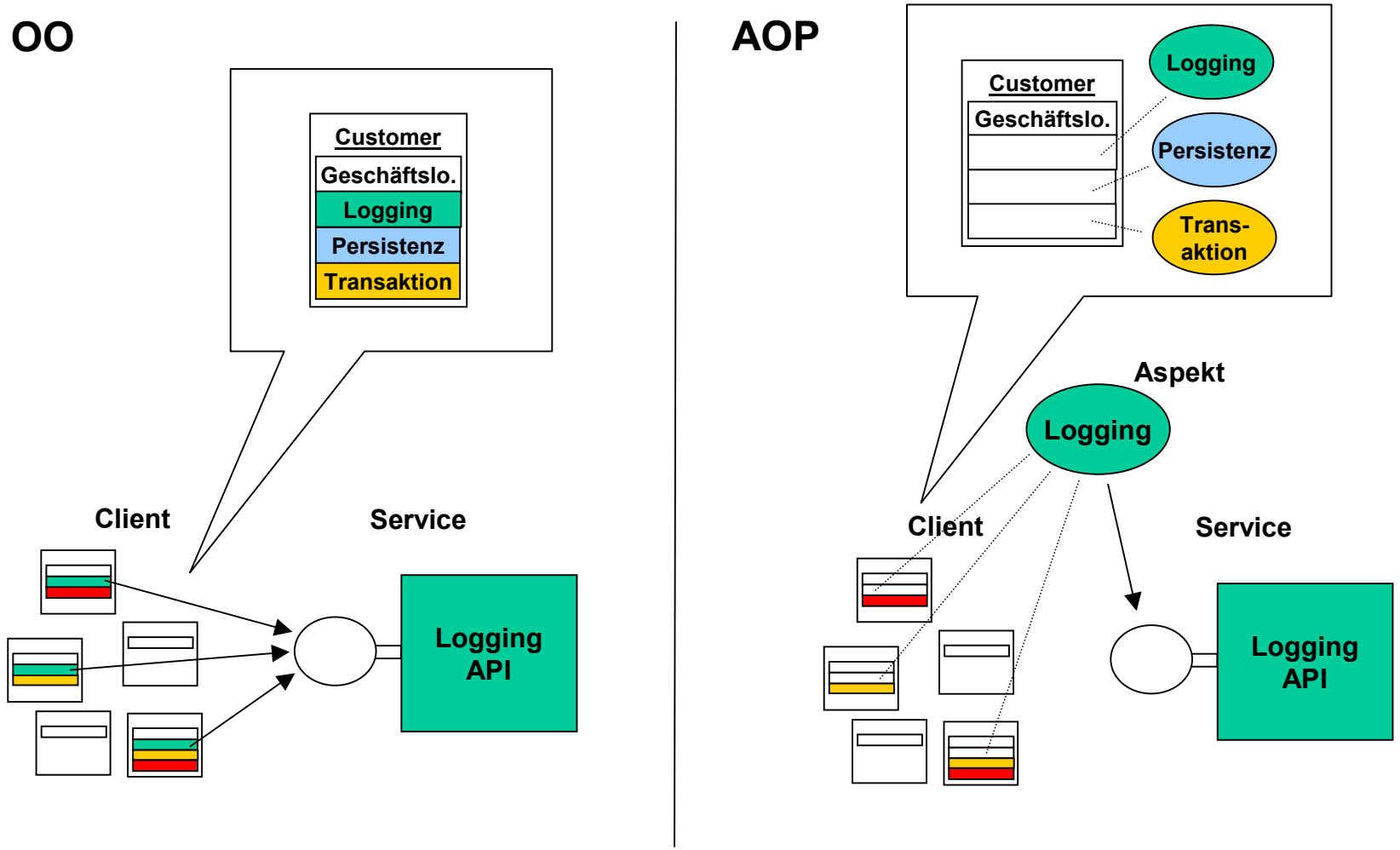
- Ist Wiederverwendung und Produktivität geringer.
  (Z.B. Geschäftslogik und technische Details vermischt.)
- Ist Code-Qualität schlechter.
  (Technik ist zu beherrschen. Mehr oder weniger beiläufige Implementierung).

Aspektorientierte Programmierung (AOP)

- Erweiterung bestehender Paradigmen (OO, Prozedural/Imperativ) zur Modularisierung von Crosscutting Concerns.
- OO: „Betrachten“ von Objekten als Modularisierungseinheit. Crosscutting Concerns als nicht modularisierbare Objekte (Aspekte).
- AOP: „Betrachten“ von Aspekten als Modularisierungseinheit. Crosscutting Concerns als Aspekte modularisieren.
  - Debugging (Logging, Tracing, Profiling, Monitoring),
  - Speichermanagement (Caching, Persistenz), Transaktionsbehandlung,
  - Sicherheit (Authentifizierung, Autorisierung),
  - Ausnahmebehandlung,Thread-Sicherheit,
  - Einhaltung von Programmierrichtlinien, Vor- und Nachbedingungen,
  - Teile der Geschäftslogik.

Aspekt: *Gesichtspunkt unter dem man etwas betrachtet.*

Crosscutting Concerns in OO und AOP



AOP Prinzip – Join Point Model

- Prinzip Programmiersprache: „Wenn Ereignis eintritt, führe Aktion aus“.

Ereignis	Aktion
getAmount()	Aufruf/Ausführung der Methode
x = y	Weise x den Wert von y zu
new Connection()	Erzeuge Instanz von Connection

- Prinzip AOP: „Wenn Ereignis (Join Point) eintritt, führe Aktion (Advice) aus“.

```
...
atm.doTransfer(…);
...
```

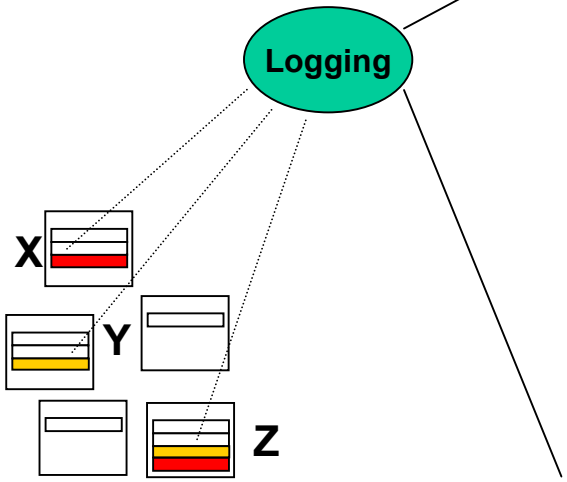
Wenn Methode *doTransfer()* auf Objekt *atm* aufgerufen wird, **dann** logge vorher „starte Transfer“.

```
setCreator(Creator c) {
  this.creator = c;
  ...
```

Wenn *creator* gesetzt wird, **dann** prüfe auf *null*.

AOP Prinzip

- Ausführen von Aktionen an verteilten Stellen, ohne den dafür notwendigen Quellcode zu verteilen.
- Selektion und Aktionen modularisiert im (zentralen) Aspekt.

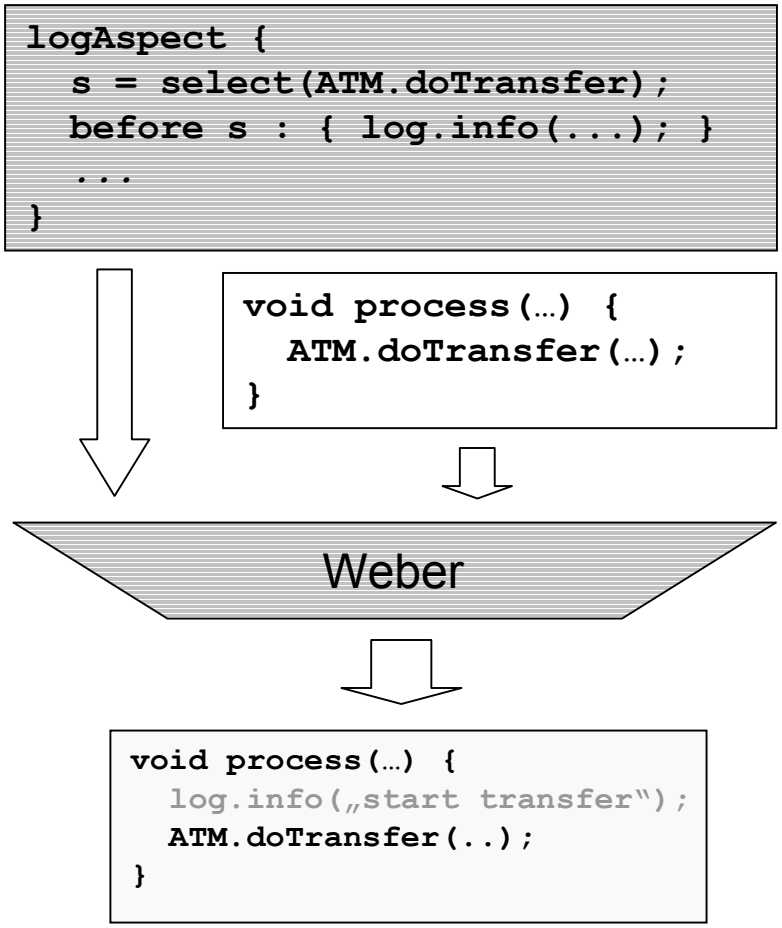


LoggingAspekt:

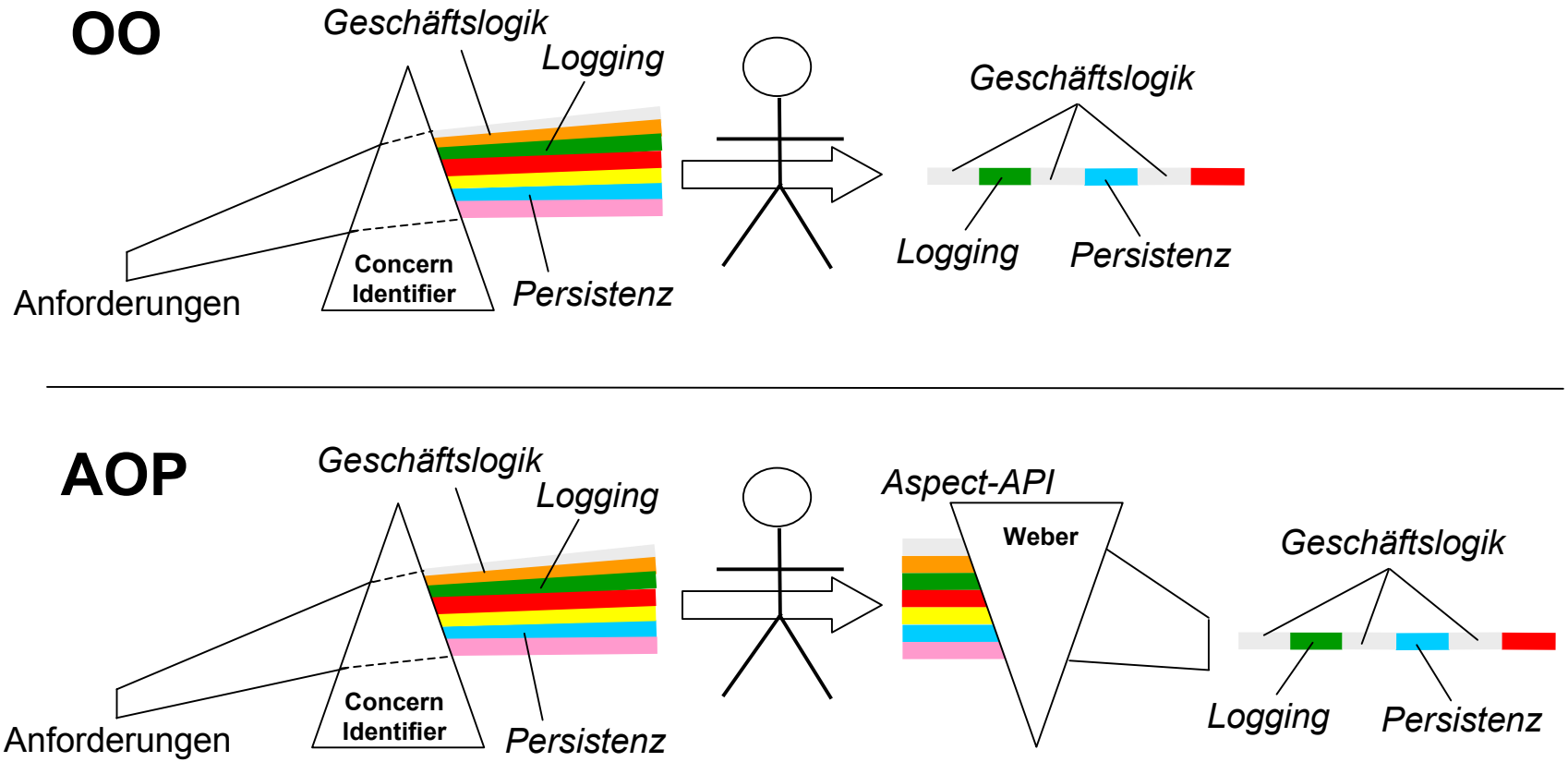
```
Selektiere Ereignispunkte (Join Points)
X, Y und Z und führe bei dessen
Eintreten die Aktion (Advice) A aus
(z.B. logge Signatur).

  S = select(X || Y || Z);
  before(S) : { log(...); }
  after(S) : { log(...); }
```

Bestandteile AOP Sprache

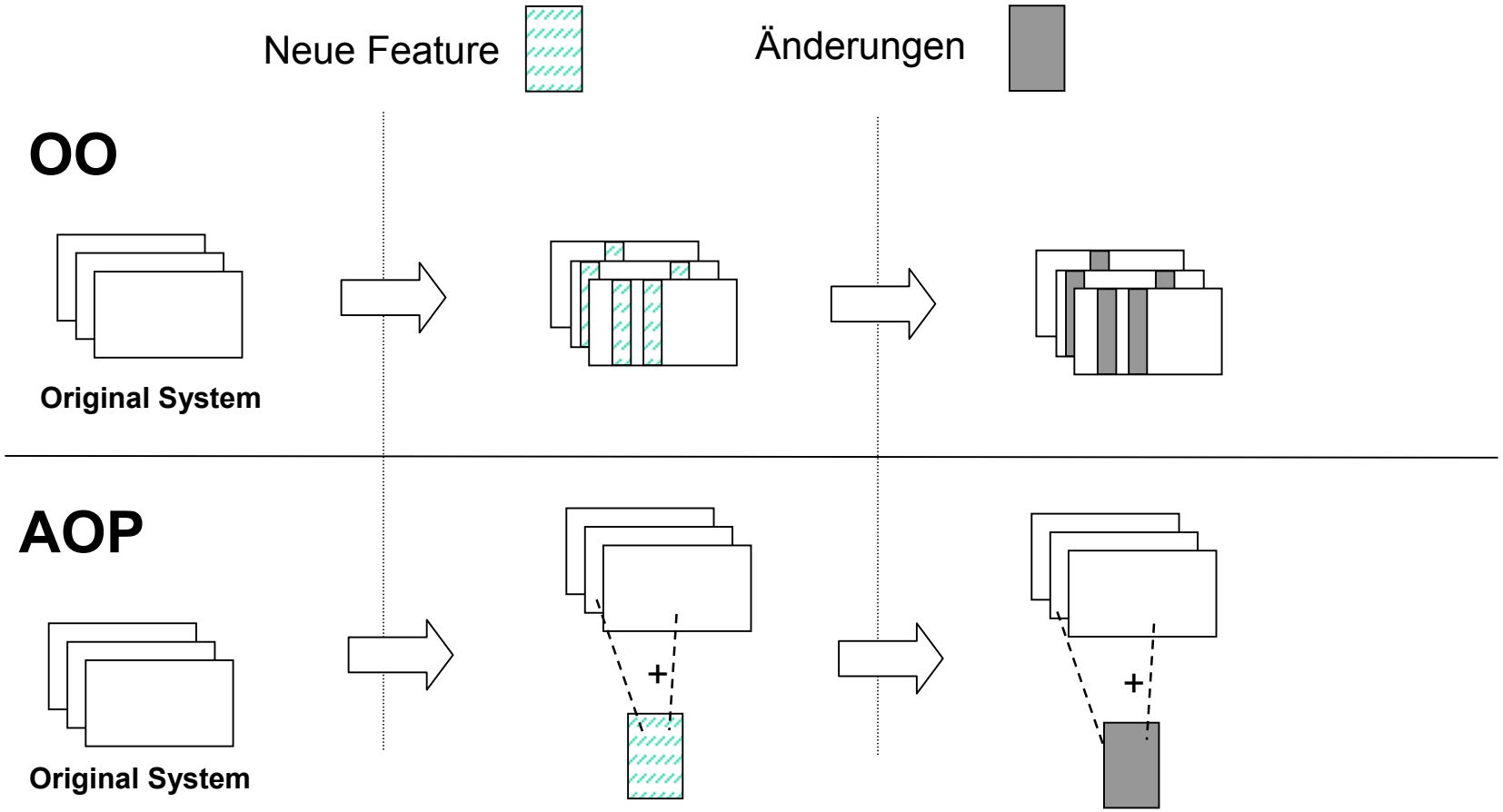
- Sprache:
  Für welchen Ereignispunkt welche Aktion ausführen.
  - Ereignisse selektieren (z.B. Methodenaufruf)
  - Aktionen definieren (z.B. logging)
- Compiler (Weber):
  Aktionen an selektierte Ereignispunkte einweben.
  - Z.B. Logging vor Aufruf der Methode.



Zeitpunkt der Komposition verschoben

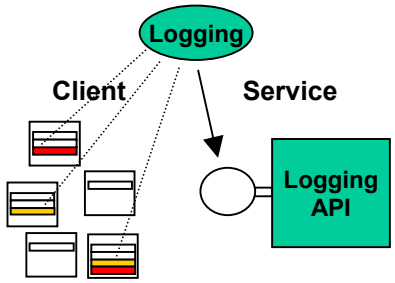


Entwicklungsphase OO und AOP

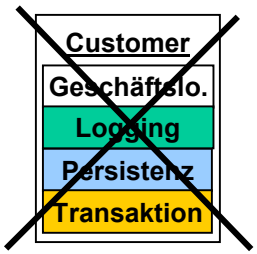


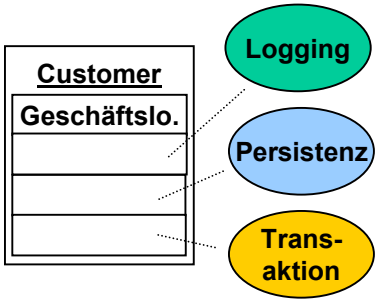
Vorteile durch AOP

- Stärkere Modularisierung:
  - Zentrale Entwicklung/Wartung. Keine Code-Verstreuung.
  - Weniger Coderedundanz, kürzerer Code.
  - Saubere, systemweit einheitliche Umsetzung. Unternehmensweite Richtlinien besser umsetzbar.



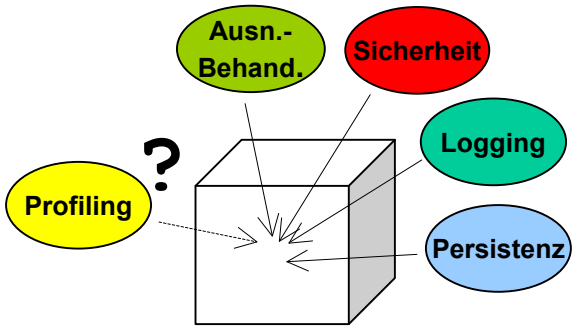
- Klarere Trennung der Verantwortung:
  - Ein Modul, eine Verantwortung. Keine Code-Vermischung.
  - Verständlicher Quellcode (Geschäftslogik, Technische Details).





Vorteile durch AOP

- Stärkere Wiederverwendung:
  - Lose gekoppelte Module.
  - System konfigurieren. Module Einweben und Herausnehmen (z.B. Profiling).
  - Aspekt-Bibliothek.



- Höhere Produktivität:
  - Entwickler konzentriert sich auf seinen Expertenbereich. Höhere Code-Qualität.
  - Unabhängige Module parallel entwickeln.
  - Eigenen Entwicklungszyklus pro Modul, da entkoppelt.

„Architect‘s dilemma“ (How much design is too much?)

- Architektur basiert auf Anforderungen. Vorhersagen über zukünftige Anforderungen nicht möglich. *Nichts ist so konstant wie die Änderung!*
- Mögliche zukünftige (Crosscutting) Concerns von Beginn nicht zu berücksichtigen, kann zu nachträglichen Änderungen oder Neuentwicklung führen.
- Weniger wahrscheinliche Concerns von Beginn an zu berücksichtigen kann zu unüberschaubarem, unhandlichen Quellcode führen.
- Mangelhaftes Design liegt häufig an den Grenzen heutiger Design Methoden.

Vorteile durch AOP in Bezug auf Architektur

- Design-Endscheidung nicht über Quellcode verstreut.
- Design-Entscheidungen können später fallen.
- Nachträgliche Einführung von Aspekten (z.B. Profiling) möglich.
- Konzentration auf Kern-System (Geschäftslogik).
- Umsetzung von Anforderungen, wenn sie gestellt werden.
  (XP: „YAGNI“: „You aren‘t gonna need it“)

AOP mit AspectJ

http://eclipse.org/aspectj/

- Erweitert Java um statische (Klassenstruktur) und dynamische (Programmablauf) Sprachkonstrukte.
- Ab ca. 1997 im Xerox PARC entwickelt, Dezember 2002 als Eclipse Projekt.
- Definiert Syntax. Bietet Weber, Debugger, Browser (Visualisierung).
- Kommandozeilen-basiert, ANT-Unterstützung. Plug-Ins für Eclipse, JBuilder, Emacs, usw.
- Aktuelle Version 1.2rc1 (Stand 11. Mai 2004).

AspectJ Sprachelemente

Aspect:
Modularisierungs-einheit, analog Klasse in OO.

Join Point:
Ereignispunkt.

Pointcut:
Relevante Ereignispunkte selektieren.

```
[abstract] anAspect {

  pointcut transferCalls() :
    call( public void
          ATM.doTransfer(Account,Account) );

  before() : transferCalls() {
      log.info(„call doTransfer");
  }

  declare error : noUIinEJBCode() :
    „No UI in EJB code!";

  declare parents: (TestCase+ && !TestCase)
          extends CustomTestCase;
}
```

Advice:
Für selektierte Ereignispunkte Aktionen ausführen.

Compile-time Declarations:
Fehler/Warnungen zur Übersetzungszeit auslösen.

Inter-type Declarations:
Modifiziert Klassenstruktur.

```
public void process(…) {
  ...
  atm.doTransfer(fromAcc,toAcc);
  ...
}
```

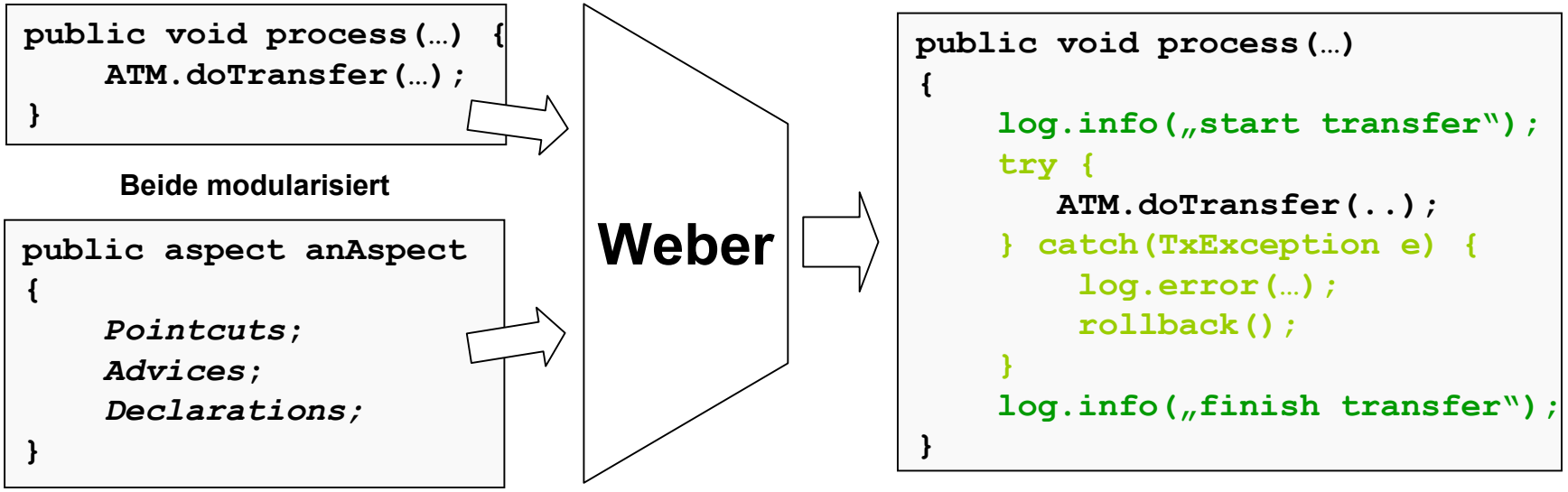
Context-Information:
Zum Pointcut vorhandene Infos (Parameter,Aufrufer,…).

= dynamisch

= statisch

AspectJ Weber

- Webt Advices an durch Pointcuts selektierte Join Points im Code ein.
- Arbeitet auf Bytecode, Sourcecode nicht erforderlich.
- Erzeugter Bytecode JVM kompatibel.



AspectJ – Join Point

- Zeitpunkt/Ereignis im Programm.

- AspectJ kennt zur Zeit:
  - Methoden-aufruf/-ausführung.
  - Konstruktoraufruf/-ausführung.
  - Attribut-Zugriff.
  - Ausnahme behandeln.
  - Klassen-Initialisierung.
  - Objekt-Initialisierung.

AspectJ – Pointcut

- Sprach-Konstrukt zur Selektierung von konkreten Ereignissen (Join Points).
  - call(public void ATM.doTransfer(Account,Account)).
  - set(Address.name).
  - withincode(void Member.doRegistration()).
- Named Pointcuts:
  - pointcut move(): call(void Point.setX(int));
- Kontext ermitteln:
  - pointcut move(int xpos): call(void Point.setX(int)) && args(xpos);
  - this(), target()

AspectJ – Pointcut

- Pattern möglich:
  - call(* javax..*.add*Listener(EventListener+));
  - handler(TransactionException+).
- Logische Operatoren: Und (&&), Oder (||), Nicht(!):
  - call(void Address.set*(..)) && withincode(void Member.doRegistration()).

AspectJ – Advice

- Definiert den Code, der bei einem Join Point ausgeführt werden soll. Der Code kann vor, nach oder anstelle des original Codes ausgeführt werden.

```
pointcut theCall() : call(* HelloWorld.sayHello(..));

before() : theCall() {
    System.out.println(„Before call to sayHello“);
}
after() : theCall() {
    System.out.println(„After call to sayHello“);
}
void around() : theCall() {
    System.out.println(„Around call to sayHello“);
    proceed();
}
```

Ursprünglicher Code

AspectJ – Kontext-Informationen

- thisJoinPoint
  - Ausgeführtes Objekt (Instanz von Foo)
  - Zielobjekt (Instanz von File)
  - Argumente (filename)
- thisJoinPointStaticPart
  - Signatur (void java.io.File.open(String))
  - Quellcode Position (Foo.java:42)
  - Join Point Art (Method-Call)
- thisEnclosingJoinPointStaticPart
  - Methode Aufrufer (Foo.foo())

```
public class Foo {
     …
     private void foo() {
          …
          file.open(filename);
          …
     }
}
```

AspectJ – Inter-type Declaration

- Modifiziert eine Klasse, ein Interfaces, die Vererbungshierachie oder einen Aspekt. Methoden und Attribute können hingefügt werden.

```
aspect NameableAspect {

    private String Nameable.name;

    public void Nameable.setName(String n) {
        name = n;
    }
    public void Nameable.getName() {
        return name;
    }

    declare parents : (Connection || Request) implements Nameable;
}
```

AspectJ – Compile-time declaration

- Kann Fehler und Warnungen zur Übersetzungszeit auslösen, wenn bestimmte Bedingungen nicht erfüllt werden.

```
public aspect LoggingRuleAspect {

    declare warning : call(* Logger.log(*))
        : „log is not performant. Use logp instead!“

    declare error : call(public PrinterQueue.new(..))
        : „Create PrinterQueue via ResourceFactory!“

}
```

AspectJ – Aspect

- Zentrale Einheit in AOP, wie Klasse zentrale Einheit in OOP. Quellcode enthält dynamische und statische Konstrukte.
- Abstrakter Aspekt und Vererbung möglich.
- Privilegierter Aspekt mit Zugriff auf private Attribute der Klasse.

```
[privileged] public [abstract] aspect FooAspect {

    // [abstract] pointcuts

    // advices

    // inter-type declarations

    // compiler-time declarations
}
```

AspectJ Sprachelemente

- Aspect:
  Modularisierungseinheit, analog Klasse in OO.

dynamisch

- Join Point:
  Ereignispunkt.
- Context-Information:
  Zum Pointcut vorhandene Infos (Parameter, Aufrufer, ...).
- Pointcut:
  Relevante Ereignispunkte selektieren.
- Advice:
  Für selektierte Ereignispunkte Aktionen ausführen.

statisch

- Inter-type Declarations:
  Modifiziert Klassenstruktur.
- Compile-time Declarations:
  Fehler/Warnungen zur Übersetzungszeit auslösen.

Einfaches Beispiel

```
public class HelloWorld {
    public void sayHello() {
        System.out.println(„Hello World!“);
    }
    public static void main(String[] args) {
        new HelloWorld().sayHello();
    }
}
```

Output:

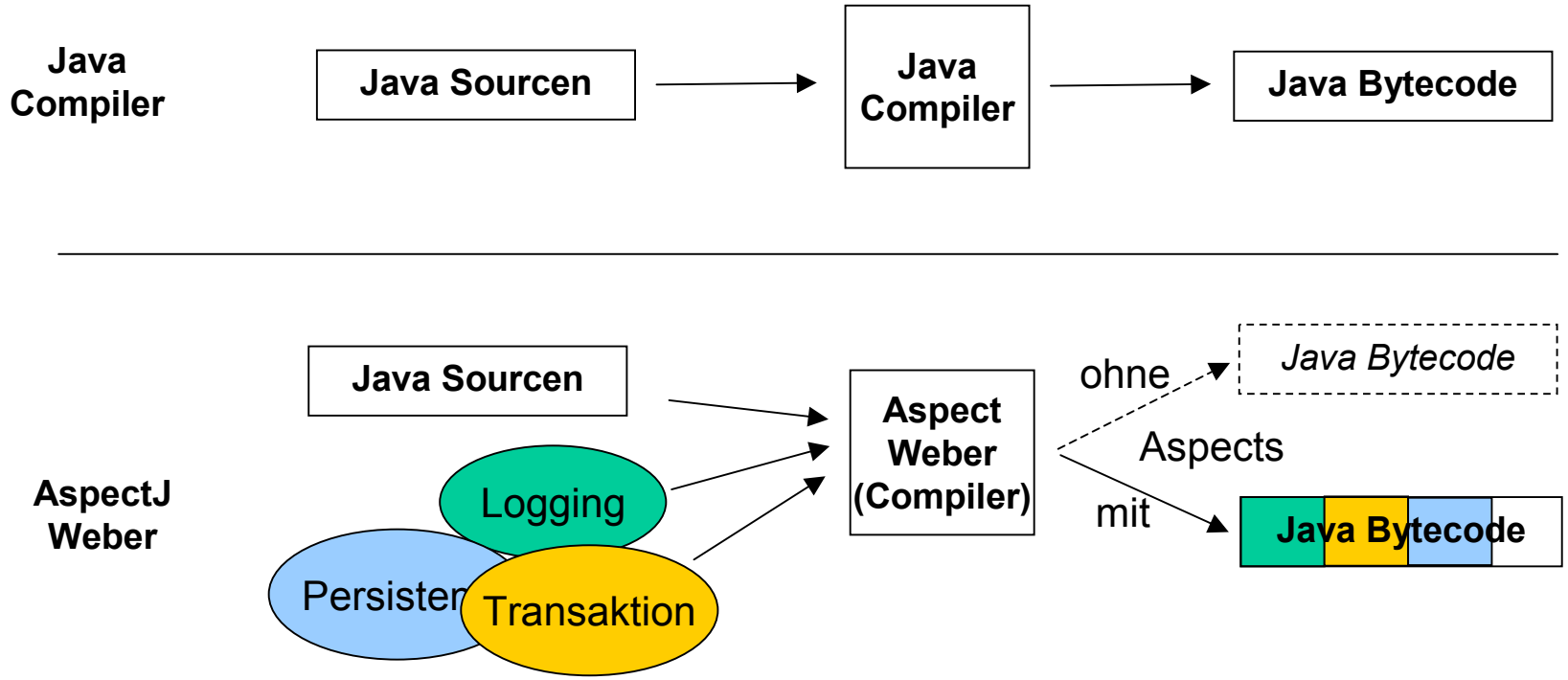
```
Before call to sayHello
Hello World!
After call to sayHello
```

Verbindung

(Im Programm erzeugt durch Weber)

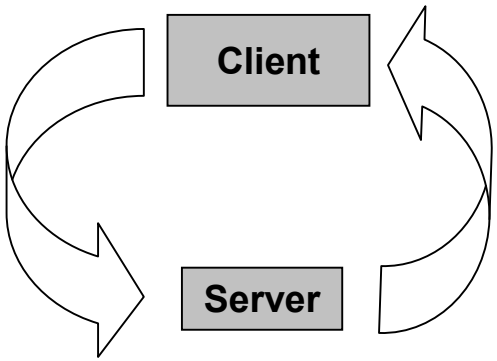
```
public aspect HelloWorldAspect {
    pointcut theCall() : call(void HelloWorld.sayHello());

    before() : theCall() {
        System.out.println(„Before call to sayHello“);
    }
    after() : theCall() {
        System.out.println(„After call to sayHello“);
    }
}
```

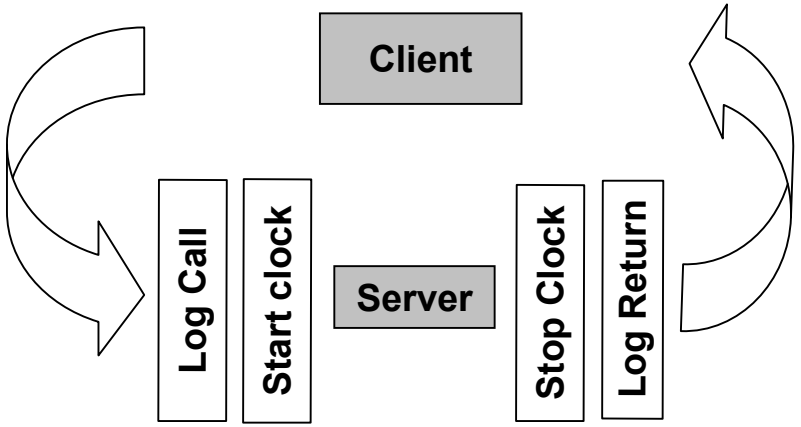
AspectJ – Weber vs. Java Compiler



Beispiel Debugging

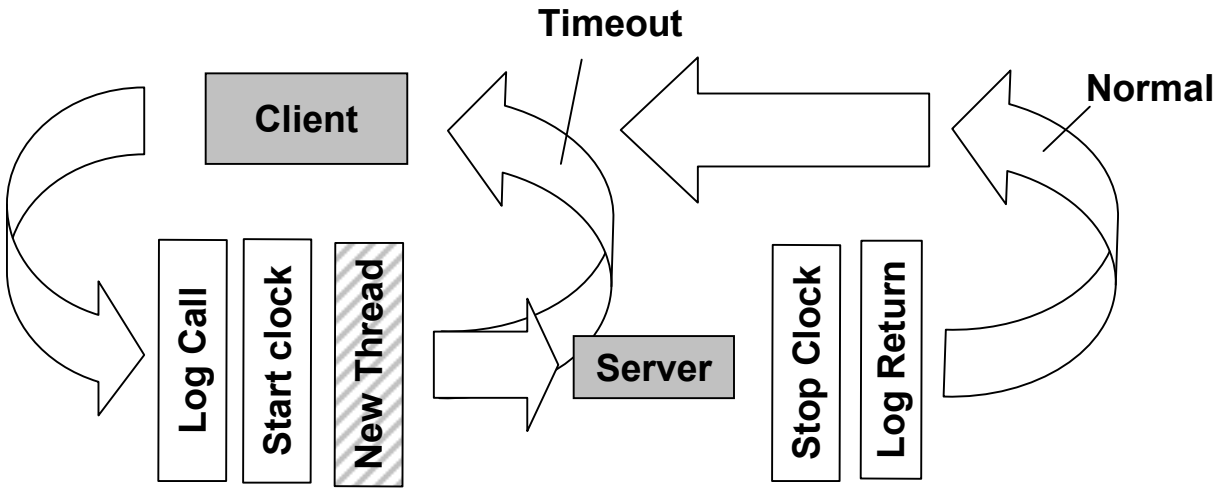
- Logging, Monitoring, Profiling am Beispiel eines Client-/Server-Aufrufs.
- Aspect stellt Logik ohne Änderung des Quellcodes bereit.





Beispiel Timeout

- Erweiterung des Client-/Server-Aufrufs um einen Timeout.
- Aspect stellt Logik ohne Änderung des Quellcodes bereit.



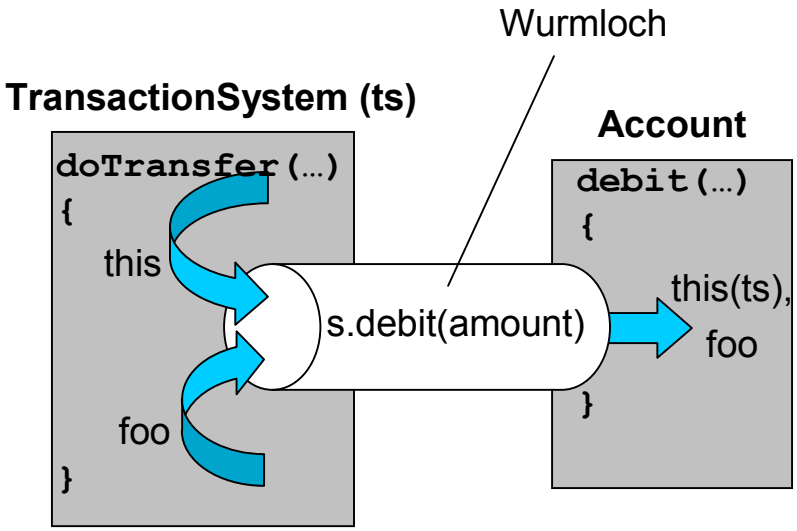
Beispiel Thread-Safe („Aspect In Action“, Ramnivas Laddad)

- Swing ist nicht Thread-Safe, trotzdem parallele Abarbeitung notwendig.
- Aktualisierung von Komponenten nur über EventQueue:
  - Synchron: EventQueue.invokeAndWait()
  - Asynchron: EventQueue.invokeLater()
- Einhaltung mit konventionellen Mitteln aufwendig und unübersichtlich.
- Aspect stellt parallele Verarbeitung und Einhalten der Thread-Regel bereit.
- Keine Änderung des original Quellcode.

Beispiel „Wurmloch“ („AspectJ In Action“, Ramnivas Laddad)

- Kontext vom Aufrufer an den Aufgerufenen weiterleiten ohne neue Parameter hinzu zufügen.

```
public class TransactionSystem {
    …
    private void doTransfer(
        Account s,
        Account t,
        float amount)
    {
        s.debit(amount);
        t.credit(amount);
    }
}
```



Beispiel Default-Implementierung

- Beispiel: Connection und Request sollen Namen bekommen.
- In Java Mehrfachvererbung nicht möglich, daher als Interface. Dadurch immer wieder neue Implementierung des Interface.
- In AOP eine Default-Implementierung für Interface.

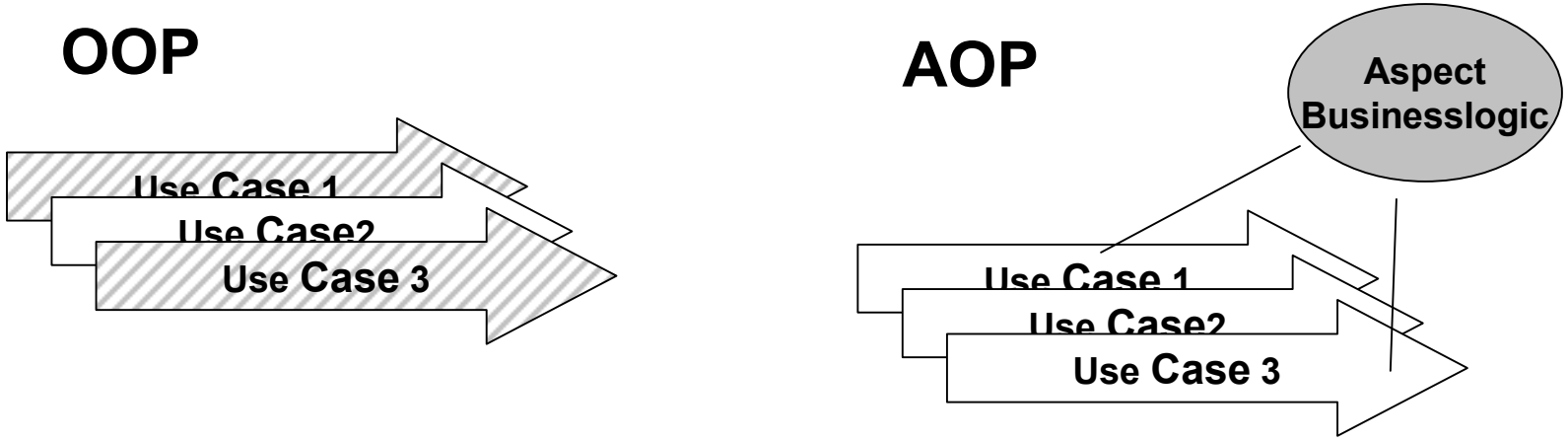
```
aspect NameableAspect {
  private String Nameable.name;

  public void Nameable.setName(String n) {
    name = n;
  }
  public void Nameable.getName() {
    return name;
  }

  declare parents : (Connection || Request)
      implements Nameable;
}
```

Beispiel Design-Pattern

- Beispiel: Beobachter (Observer): Subjekt: *Knopf*, Beobachter: *Liste*.
- Abstrakter Aspekt als Adapter zwischen Pattern (Rollendefinition) und Objekten (Rollenspieler).
- Pattern(-protokoll) ist modularisiert und nur einmal im System vorhanden.
- Konkrete Aspekte als Brücke zwischen Subjekt und Beobachter.
- Weder *Knopf* noch *Liste* weiß, dass es Subjekt bzw. Beobachter ist. Beide Objekte kennen sich nicht.

Beispiel Geschäftslogik

- Bonuspunkt je nach Aktion (Order, Fonds, Überweisung, LiveTrading, …)
- Statt Quellcode in jeden Use Case Aspekt schreiben.
  Wenn Aktion vorbei, Aspekt wieder raus!



Einsatz in Entwicklungsphasen (Adaption)

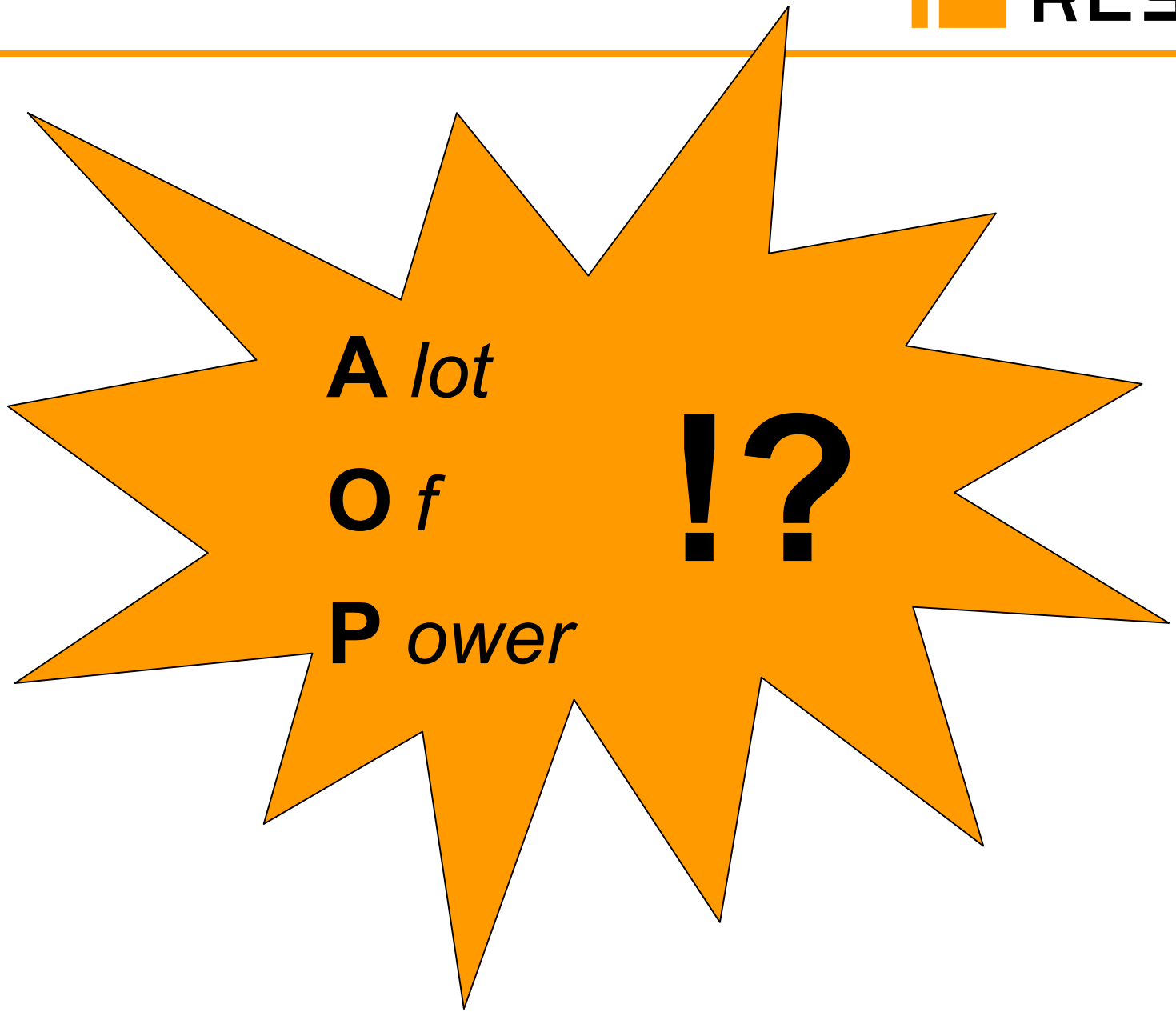
- Entwicklung
  - Erfahrungen machen.
  - Aspekte vom Originalquellcode getrennt, Produktion ohne Aspekte möglich.
  - Einsatz: Logging, Debugging, Tracing, Monitoring, Vor-/Nachbedingungen, systemweite Konventionen.
  - Aufbau Aspekt-Bibliothek.
- Test
  - Zugriff auf private Implementierungsdetails ohne Codeänderung.
  - Im Falle von Fehlern mehr Kontext zur Verfügung stellen.
- Produktion
  - Nach Erfahrung während Entwicklung in Produktion übernehmen.

Aspektorientierte Programmierung (AOP)

- AOP ist:
  - Erweiterung bestehender Paradigmen (OO, Prozedural/Imperativ) zur Modularisierung von Crosscutting Concerns
  - Weniger komplexere Lösung für bestehende Modularisierungsprobleme.
  - Nicht Domain-spezifisch, sondern universell.
  - Niemals ideal, aber deutlich besser als bisherige Mittel (SD=>OD).
- AOP ist nicht:
  - Ein neues eigenes Paradigma.
  - Ein Patch für schlechtes Design.
  - Lösung für alle Probleme.

Resourcen

- Bücher
  - „AspectJ in Action“, Ramnivas Laddad, Manning 2003.
  - „Mastering AspectJ“, Joseph D. Gradecki, Nicholas Lesiecki, Wiley 2003.
- Internet
  - AspectJ: eclipse.org/aspectj
  - Eclipse Plug-In AJDT: eclipse.org/ajdt.
  - Aspect-Oriented Software Development: aosd.net/.
  - Folien: www.thomas-baustert.de


A lot
O f
P ower
!?