Resco – the business enabler.

RESCO

RESCO

Einführung in die aspektorientierte Programmierung (AOP)

Beispiele mit AspectJ

Thomas Baustert
Resco GmbH

Motivation

- AOP ist ein mächtiges „Werkzeug“
- AOP wird zunehmend an Bedeutung gewinnen.
  - Kommerzielle Nutzung
  - Immer mehr Artikel, Bücher, Vorträge, insbesondere über AspectJ.
  - JBOSS 4.0, BEA 8.1 starten mit AOP Framework.

Ziel

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

Inhalt

- Modularisierung und Grenzen heutiger Programmiersprachen.
- Aspektorientierte Programmierung.
- AspectJ und Beispiele.
- 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: d*tsch.: Anliegen.*
Wird in AOP als Verantwortung/ Verantwortlichkeit übersetzt. Anliegen wird durch Klassen implementiert, die jeweils die Verantwortung tragen.

Vorteile durch Modularisierung

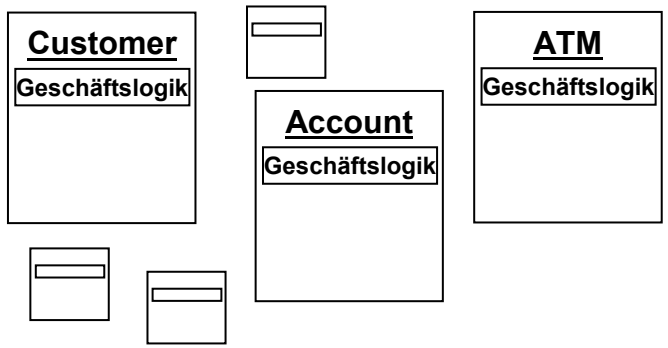
- Bessere Bewältigung der Komplexität eines Softwaresystems.
- Bessere Wiederverwendung von Teilen.
- Kein redundanter Quellcode.
- Parallele Entwicklung der getrennten Bereiche durch Experten.
- Höhere Produktivität, geringere Kosten.

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 Objekte (Klassen).
- *Aspektorientierte Programmierung modularisiert Verantwortlichkeiten in Aspekte.*

Entwicklung einer Geschäftsanwendung mit OO

- Geschäftslogik ist primäres Anliegen, der Kern der Anwendung.
  In AOP „Core Concerns“ genannt.
- Primäre Aufgabe des Entwicklers ist die Umsetzung der Geschäftslogik.
- Modularisierung durch Aufteilung der Verantwortlichkeiten in Klassen.
  Unterstützung: Vererbung, Trennung Schnittstelle/Implementierung.



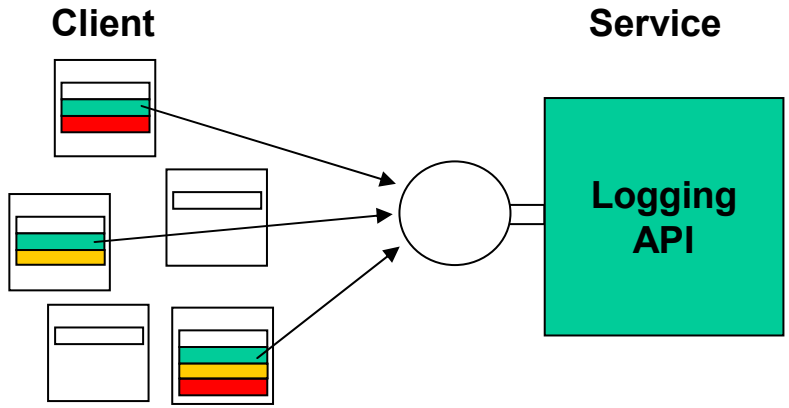
Systemweite Aspekte („Crosscutting Concerns“)

- Primäre Aufgabe des Entwicklers ist die Umsetzung der Geschäftslogik.
- Weitere Aufgabe des Entwicklers ist die Umsetzung technischer Aspekte, z.B. Logging, Persistenz, Ausnahmebehandlung, usw.
- Aspekte wie Logging, Persistenz, usw. sind systemweit verteilt und werden in AOP daher „Crosscutting Concerns“ (querschneidende Verantwortlichkeiten) genannt.
- Systemweite Aspekte werden in Kern-Module (Klassen) der Geschäftslogik integriert.

Aspekt:
„Gesichtspunkt unter dem man etwas betrachtet“.

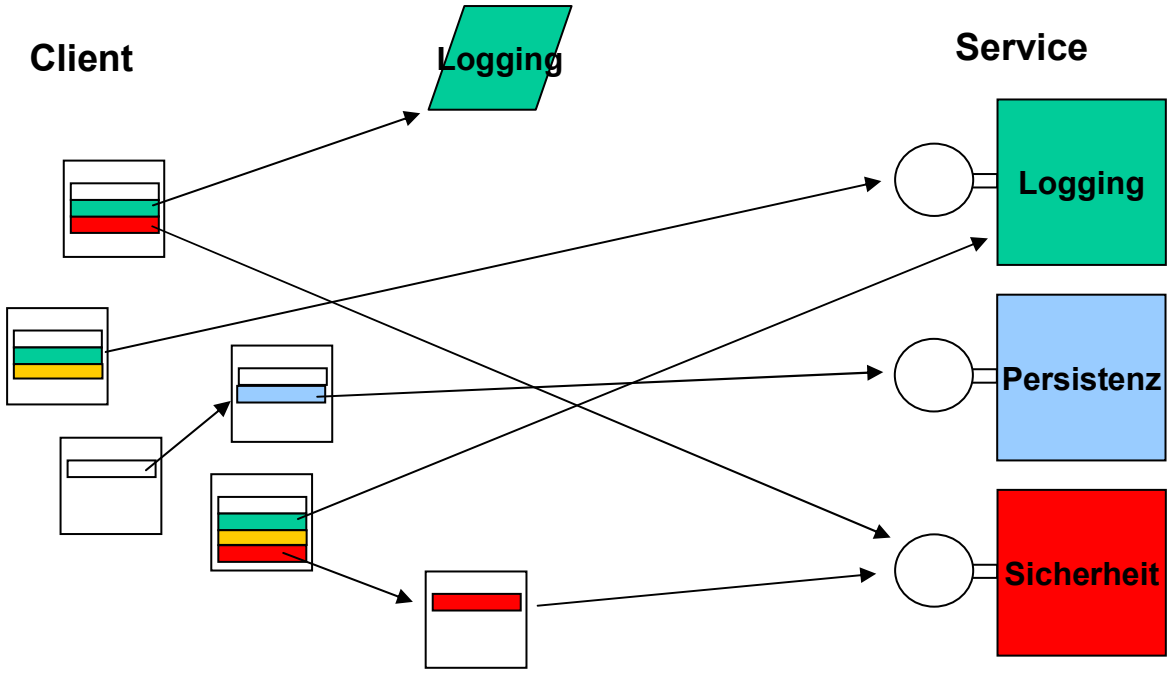
OO Modularisierung systemweiter Aspekte

- 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. Nutzung Schnittstelle hilft nicht.



OO Modularisierung systemweiter Aspekte

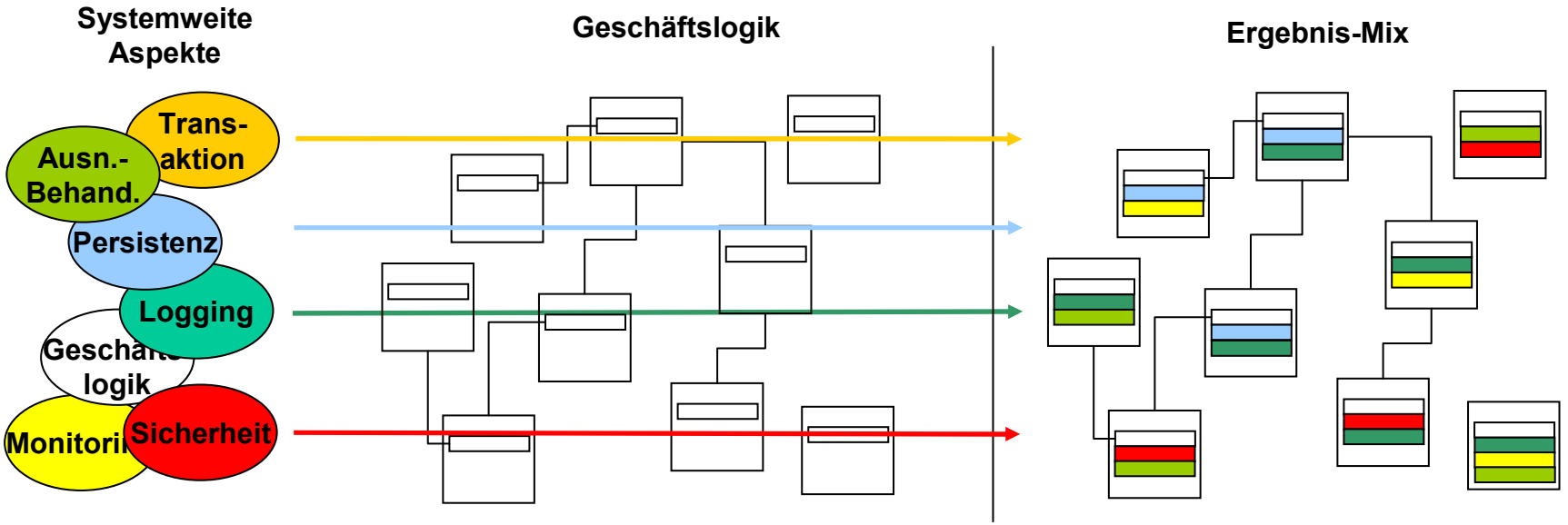
- Unterschiedliche Service-Verwendung.



OO Modularisierung systemweiter Aspekte

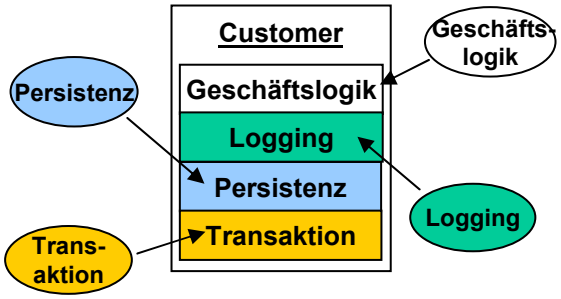
- Crosscutting Concerns können mit heutigen Methoden und Programmiersprachen nicht modularisiert werden.

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

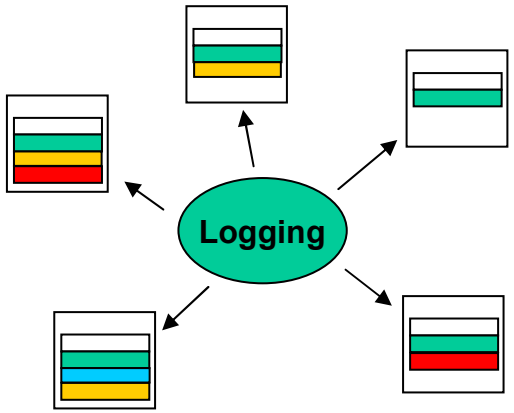


Symptome der Crosscutting Concerns

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



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



Nachteile durch Crosscutting Concerns

- Geringere Produktivität: Entwickler muss sich neben der Geschäftslogik auch mit technischen Details beschäftigen.
- Mindere Code-Qualität: Entwickler muss Technik (Framework, EJB) gut beherrschen. Mehr oder weniger beiläufige Implementierung.
- Geringere Wiederverwendung: Geschäftslogik und technische Details sind verbunden.
- Erschwerte Erweiterbarkeit: Änderungen sind umfangreich und können technische Details wie auch Geschäftslogik beeinflussen.

Rückblick

- Vorteile durch Modularisierung
- Modularisierung in OO (Geschäftsanwendung)
  - Trennung Schnittstelle/Implementierung, Vererbung, Design-Pattern.
  - Modularisierung Core Concerns, Server-Teil gut.
  - Grenzen (Fehlende Modularisierung von Crosscutting Concerns).

Lösungsansätze

Lösungsansätze zur Modularisierung der Crosscutting Concerns:

- Design Pattern, z.B. Visitor, Observer.
- EJB und Deskriptoren. Auslagerung Persitenz, Security, etc.
- Entwicklung einer Business Language (Sun ACE).
- Aspektorientierte Programmierung.

Aspektorientierte Programmierung

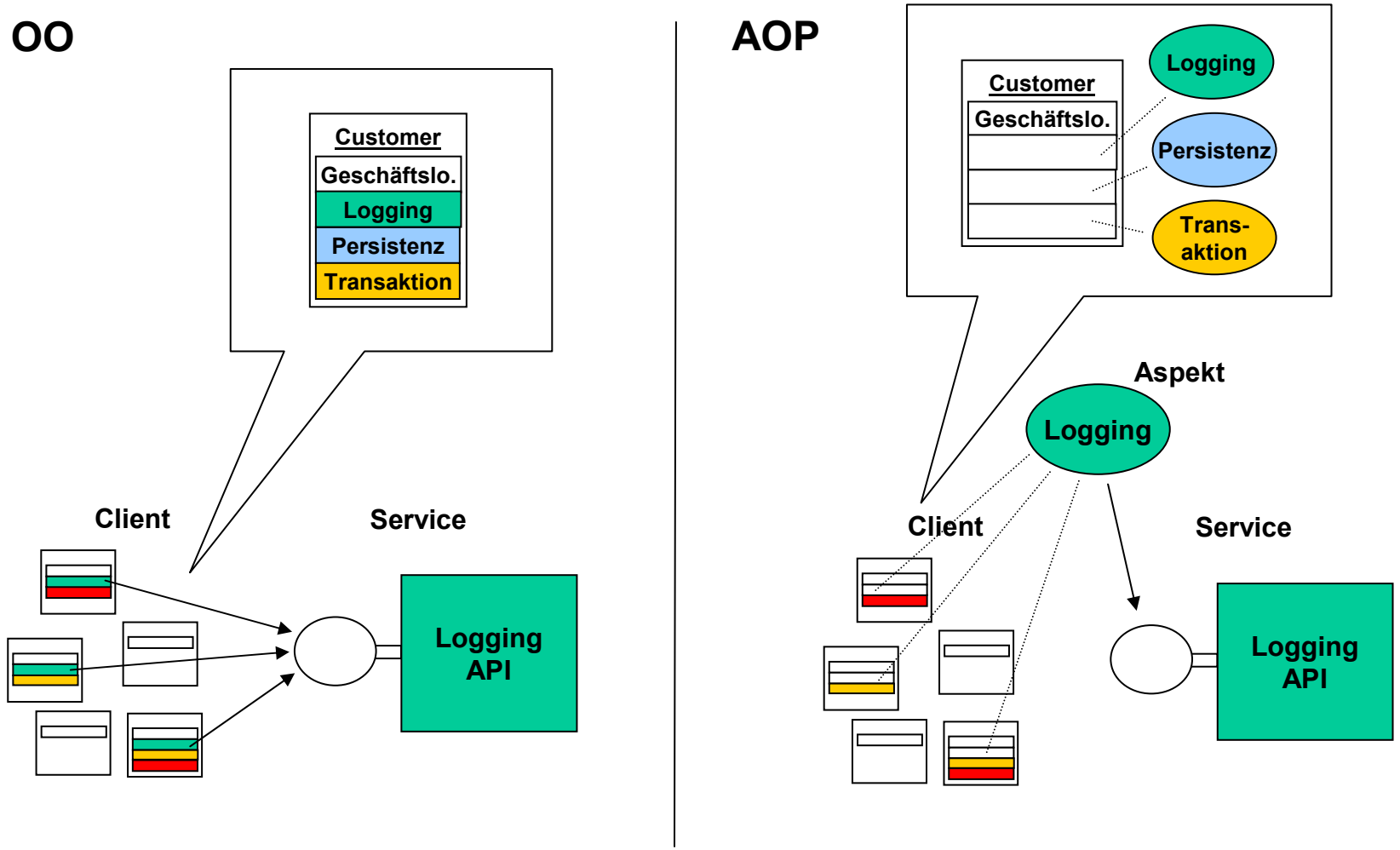
- Aspekt: „Gesichtspunkt unter dem man etwas betrachtet“.

- Aspekte im Sinne der AOP: Crosscutting Concerns.
  - Debugging (Logging, Tracing, Profiling, Monitoring),
  - Speichermanagement (Caching, Persistenz),
  - Transaktionsbehandlung,
  - Sicherheit (Authentifizierung, Autorisierung),
  - Ausnahmebehandlung,
  - Thread-Sicherheit,
  - Einhaltung von Programmierrichtlinien,
  - Vor- und Nachbedingungen,
  - Teile der Geschäftslogik.

Aspektorientierte Programmierung

- OO führt Modularisierung für Core Concerns ein.
- AOP führt Modularisierung für Crosscutting Concerns ein.
- AOP ergänzt/erweitert OO um Modularisierung von Crosscutting Concerns.
- AOP beginnt da, wo OO an seine Grenzen stößt.

Crosscutting Concerns in OO und AOP

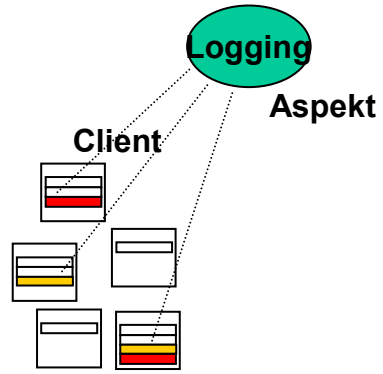


„It‘s not a trick, it‘s AOP!“

- Zeitpunkte/Ereignisse im Programm betrachten (Joinpoints).
  Beispiel: Methodenaufruf.

- Mit AOP-Sprache für Kontext relevante Ereignisse herausfiltern (Pointcuts).
  Beispiel: *doTransfer()*-Methode auf Objekt von *ATM*.

- Für relevante Ereignisse ein Verhalten erzwingen (Advice).
  Beispiel:
  - Vor Aufruf von *doTransfer()* logge „start doTransfer“.
  - Nach Aufruf von *doTransfer()* prüfe Invarianz.
  - Anstelle von *new Connection()* nehme Connection aus Pool.



Joinpoints (AspectJ)

- Methoden-aufruf/-ausführung.
- Konstruktoraufruf/-ausführung.
- Attribut-Zugriff.
- Ausnahme behandeln.
- Klassen-Initialisierung.
- Objekt-Initialisierung.

RESCO

Pointcuts (AspectJ)

- Sprach-Konstrukt zur Filterung von Ereignissen:
  - execution(public void ATM.doTransfer(..)).
  - call(public void *.set*(Foo)).
  - set(Address.name).
  - handler(TransactionException+).
  - withincode(void Member.doRegistration()).
- Logische Operatoren: Und (&&), Oder (||), Nicht(!):
  - call(void Address.set*(..)) && withincode(void Member.doRegistration()).

Advices (AspectJ)

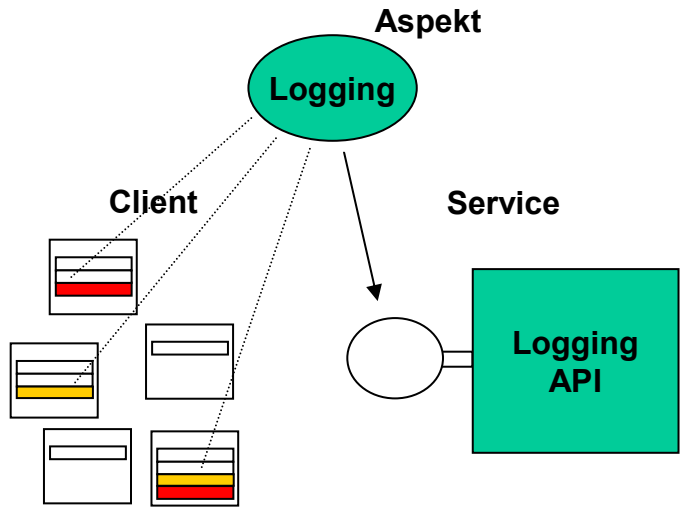
- Sprach-Konstrukt zur Definition von Aktionen an Pointcuts.
- Vor Ereignis führe Aktion aus: before execution(ATM.doTransfer(..)).
- Nach Ereignis führe Aktion aus: after execution(ATM.doTransfer(..)).
- Anstelle Ereignis führe Aktion aus: around call(new Connection()).

Modulierungseinheit Aspekt (AspectJ)

- Was in OO die Klasse, ist in AOP der Aspekt.
- Aspekt definiert Elemente zur Modularisierung von Crosscutting Concerns. (AspectJ: Pointcuts, Advices, Introduction, Compile-time declarations, Java-Code).

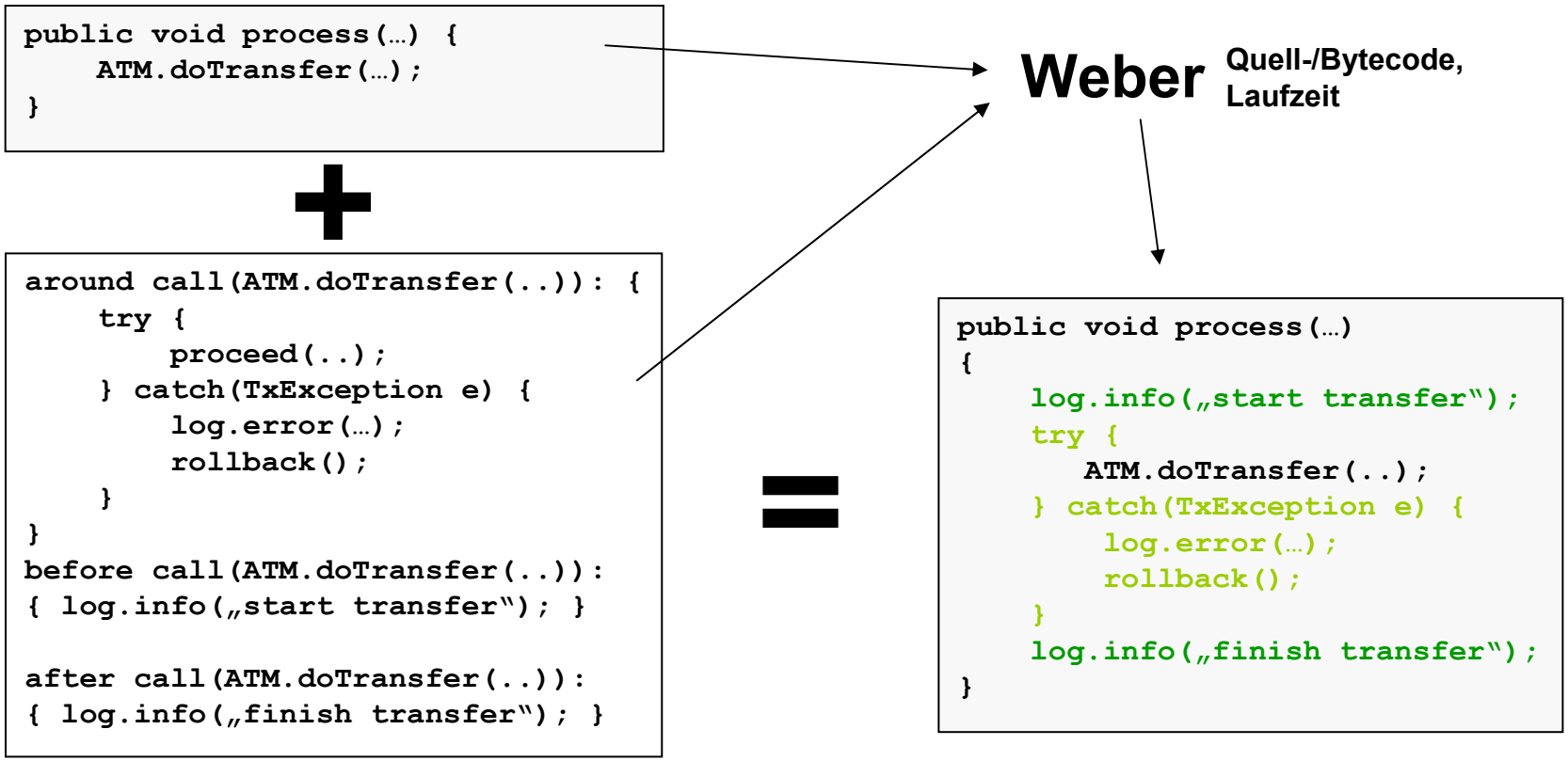
```
public aspect anAspect
{
    define pointcut(s);

    define advice(s) on pointcut(s);
}
```

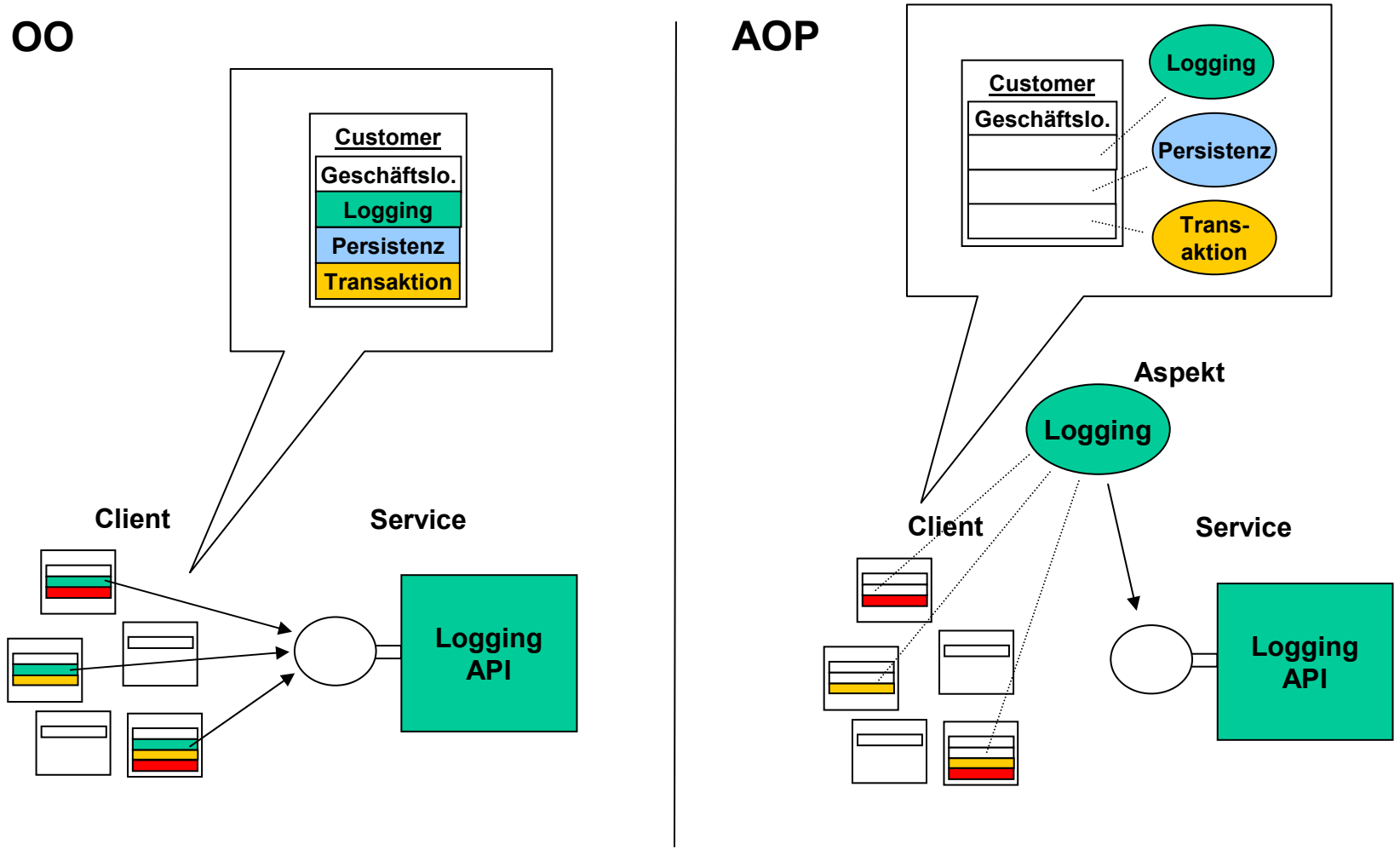


Weber

- Webt Advices an durch Pointcuts definierte Punkte im Code ein.

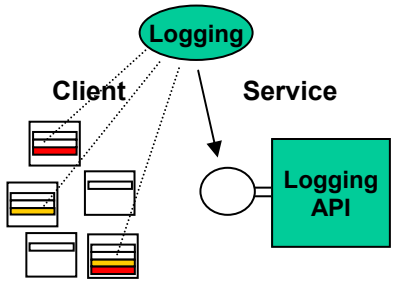


Crosscutting Concerns in OO und AOP

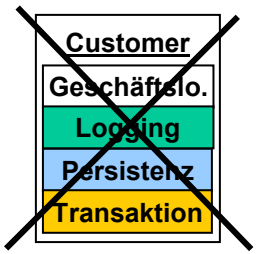


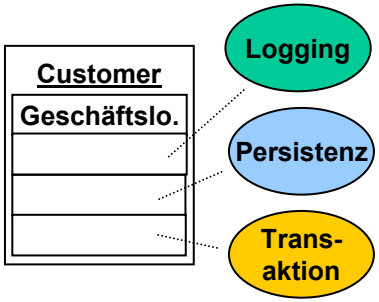
Vorteile durch AOP

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

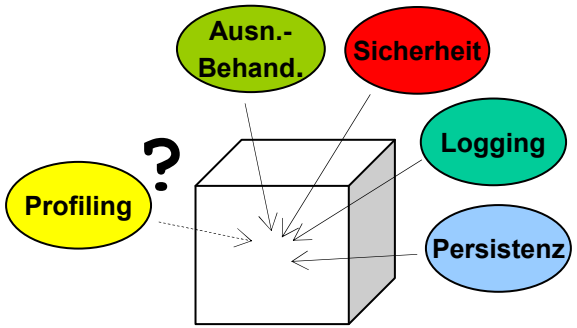


- Klarere Trennung der Verantwortung:
  - Ein Modul, eine Verantwortung. Keine Code-Vermischung.
  - Verständlicher Quellcode (Geschäftslogik, Technische Details).



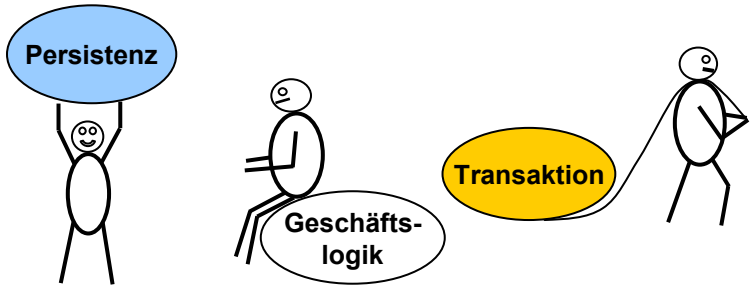


Vorteile durch AOP



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

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



Rückblick

- Vorteile durch Modularisierung
- Modularisierung in OO
  - Trennung Schnittstelle/Implementierung
  - Modularisierung Core Concerns, Server-Teil gut.
  - Grenzen (Fehlende Modularisierung von Crosscutting Concerns).
- Modularisierung in AOP
  - Modularisierung Crosscutting Concerns in Aspekten.
  - Joinpoint, Pointcut, Advice.
  - Syntax und Weber.

AOP mit AspectJ

aspectj
crosscutting objects for better modularity

www.aspectj.org

- Ca. 1997 im Xerox Palo Alto Research Center entwickelt und im Dezember 2002 an das Eclipse Projekt übergeben.
- Erweitert Java um statische (Klassenstruktur) und dynamische (Programmablauf) Sprachkonstrukte.
- Definiert Syntax, bietet Weber.
- Weber arbeitet auf Bytecode Ebene. Erzeuge Klassen erfüllen die Java Bytecode-Spezifikation und laufen in jeder JVM.
- Plug-Ins für Eclipse, JBuilder, Emacs, usw.

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
```

Aspect

Pointcut

```
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“);
    }
}
```

Advice

Verbindung

(Im Programm erzeugt durch Weber)

AspectJ – Sprachkonstrukte

- Dynamische Sprachkonstrukte: Neues Verhalten bei Ausführung des Programms.
  - *Join Point*.
  - Pointcut.
  - Advice.
- Statische Sprachkonstrukte: Modifizieren statische Struktur des Programms (z.B. Vererbungshierachie) oder erzeugen Fehler zur Übersetzungszeit .
  - Introduction.
  - Compile-time declaration.
- Aspekt als Modularisierungseinheit.

AspectJ – Join Point

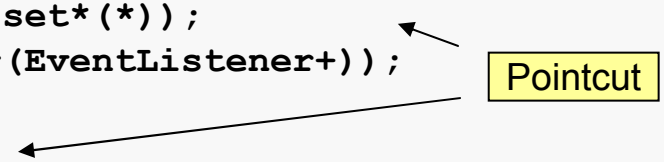
- Methoden-aufruf/-ausführung.
- Konstruktoraufruf/-ausführung.
- Attribut-Zugriff.
- Ausnahme behandeln.
- Klassen-Initialisierung.
- Objekt-Initialisierung.

AspectJ – Pointcut

- Ein *Pointcut* ist ein dynamisches Programmkonstrukt.
- Ein *Pointcut* selektiert Join Points und sammelt den Kontext auf, z.B. das Objekt, auf dem die Methode aufgerufen wird und die Argument, die übergeben werden.

```
execution(void Account.credit(float));
execution(!public * Account.set*(*));
call(* javax..*.add*Listener(EventListener+));

set(Account.balance);
get(public !final *.*);
```



AspectJ – Advice

- Ein *Advice* (Hinweis) ist ein dynamisches Programmkonstrukt.

- Ein *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();
}
```

Advice

AspectJ – Introduction

- Eine *Introduction* (Einführung, Neuerung) ist ein statisches Programmkonstrukt.

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

```
public aspect MinimumBalanceRuleAspect {
    private float Account.minimumBalance;

    public float Account.getAvailableBalance() {
        return getBalance() - minimumBalance;
    }
    …
}
```

Introduction

AspectJ – Compile-time declaration

- Eine *Compile-time declaration* ist ein statisches Programmkonstrukt.
- Es 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!“

}
```

CTD

AspectJ – Aspect

- *Aspect* modularisiert die „Crosscutting Concerns“.
- *Aspect* ist zentrale Einheit in AOP, wie Klasse zentrale Einheit in OOP. Quellcode enthält dynamische und statische Konstrukte.
- Abstrakte Aspekte und Vererbung möglich

```
public [abstract] aspect FooAspect {
    // [abstract] pointcuts
    // advices
    // introduction
    // compiler-time declaration
}
```

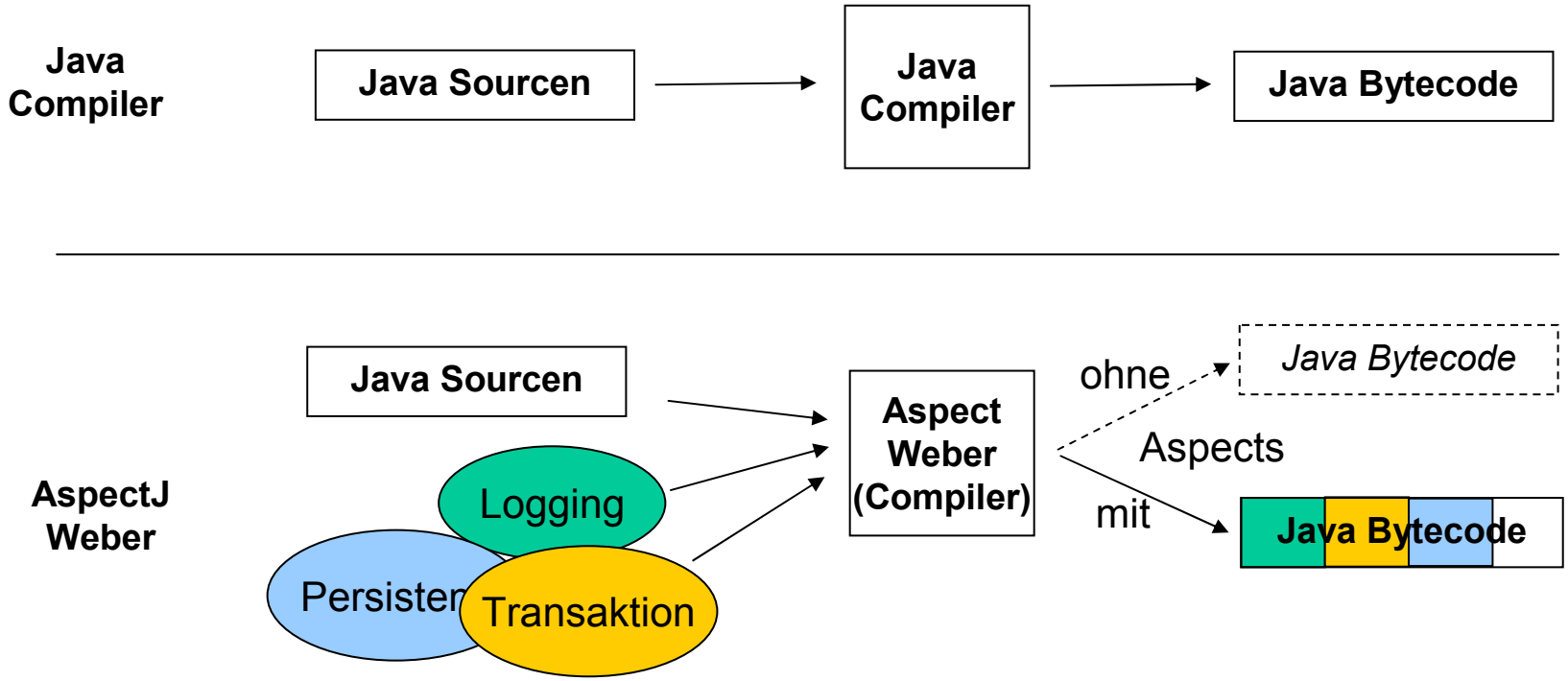
Aspect

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())

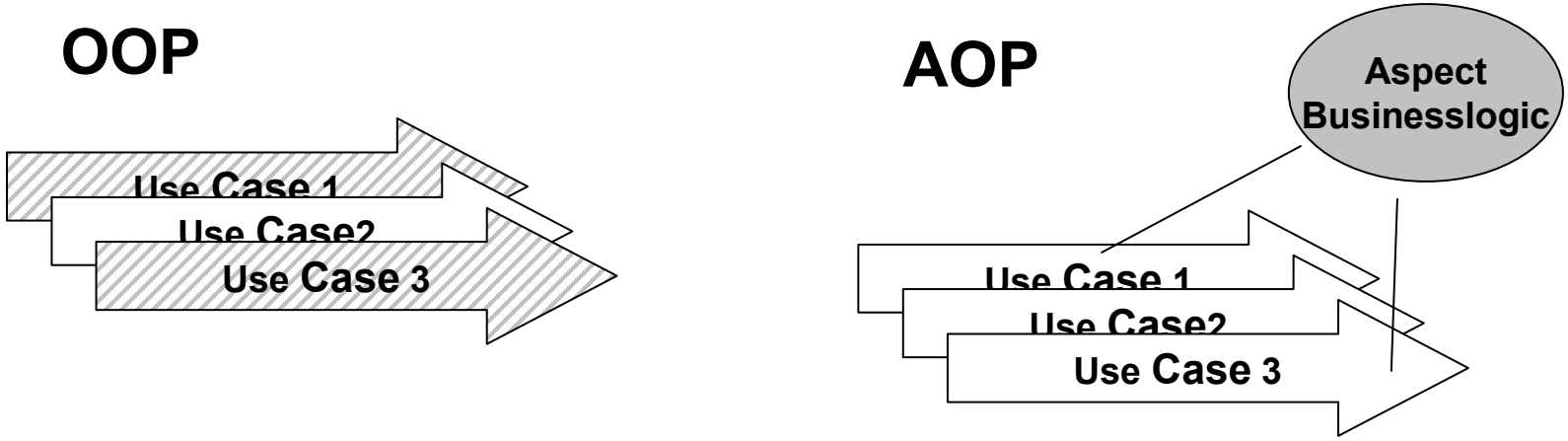
```
public class Foo {
     …
     private void foo() {
          …
          file.open(filename);
          …
     }
}
```

AspectJ – Weber vs. Java Compiler



Beispiel Geschäftslogik

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



Aspektorientierte Programmierung

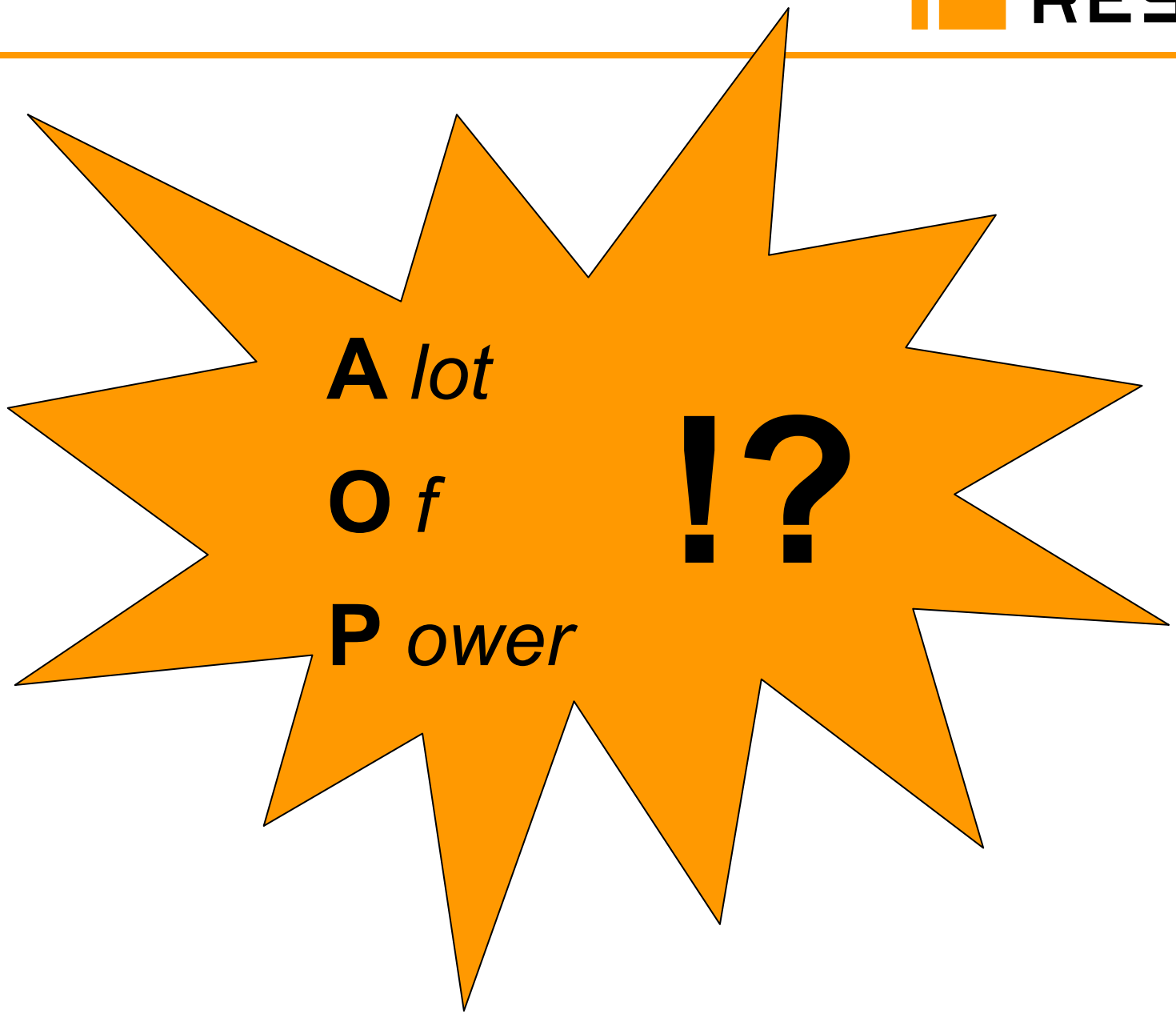
- Ermöglicht Modularisierung von Crosscutting Concerns.
- Erweiterung/Ergänzung bestehender Paradigmen.
- Kein entweder OOP oder AOP.
- Für Initiales Design oder Refactoring.
- Nicht Domain-spezifisch, sondern universell.
- Kein Mittel gegen schlechtes Design.

Fazit

- AOP
  - Ist kein Hype-Thema.
  - ist ein mächtiges Werkzeug.
- Theorie und Praxis
  - Niemals ideal, aber deutlich besser als bisherige Mittel (SD=>OD).
  - Aller Anfang ist schwer.
- Probleme
  - Abgleich Aspekte, Quellcode.
  - AspectJ: Dauer Compilerzeit.
  - Wie adaptieren?

Resourcen

- Bücher
  - „AspectJ in Action“, Ramnivas Laddad, Manning 2003.
  - „Mastering AspectJ“, Joseph D. Gradecki, Nicholas Lesiecki, Wiley 2003.
- Internet
  - AspectJ: www.aspectj.org


A lot
O f
P ower
!?