

Bessere Software  
durch AOP?

ex|Xcellent  
solutions

Achim Demelt  
OOP 2008

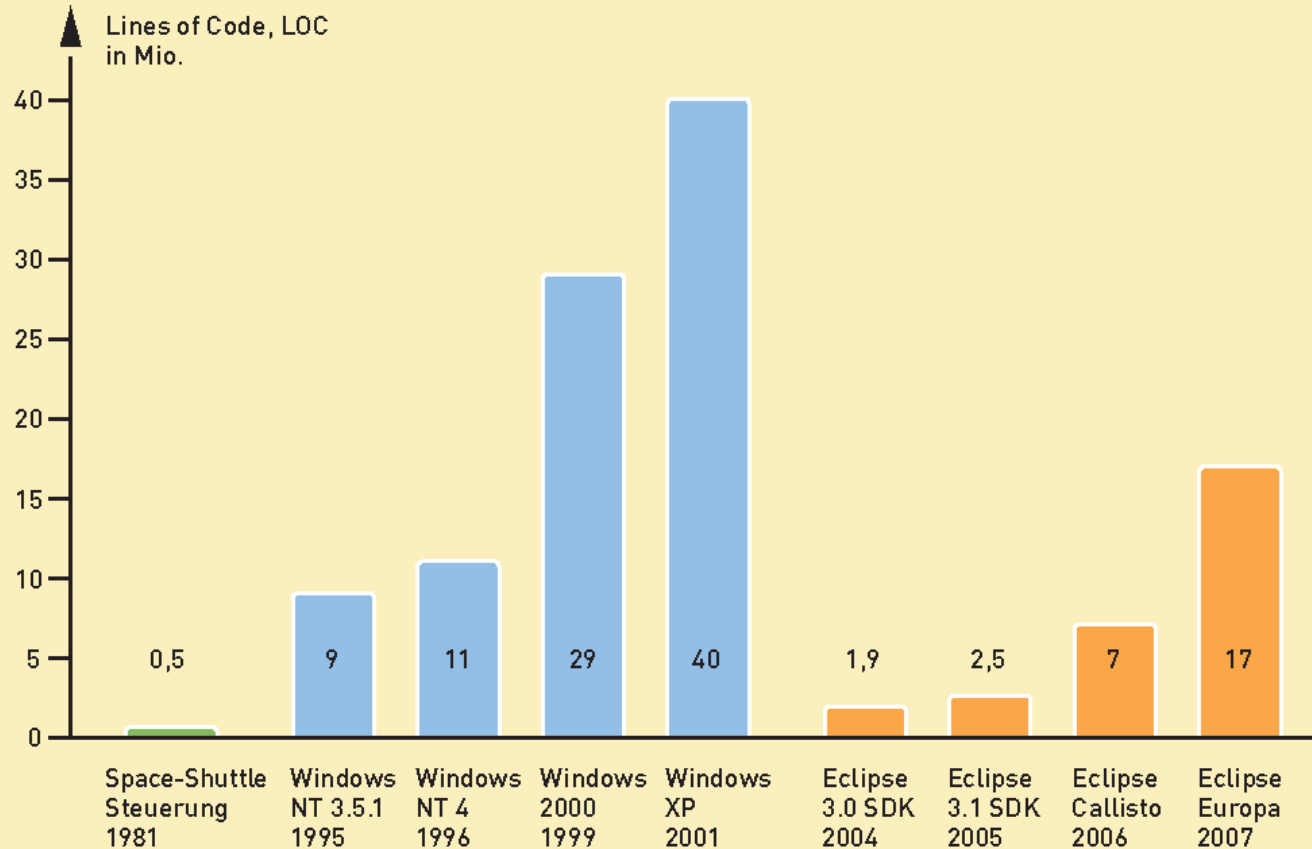


- x| Motivation
- x| Konzepte der Aspektorientierung
- x| Aspektorientierung in der Praxis
- x| Bessere Software?
- x| Die Zukunft von AOP

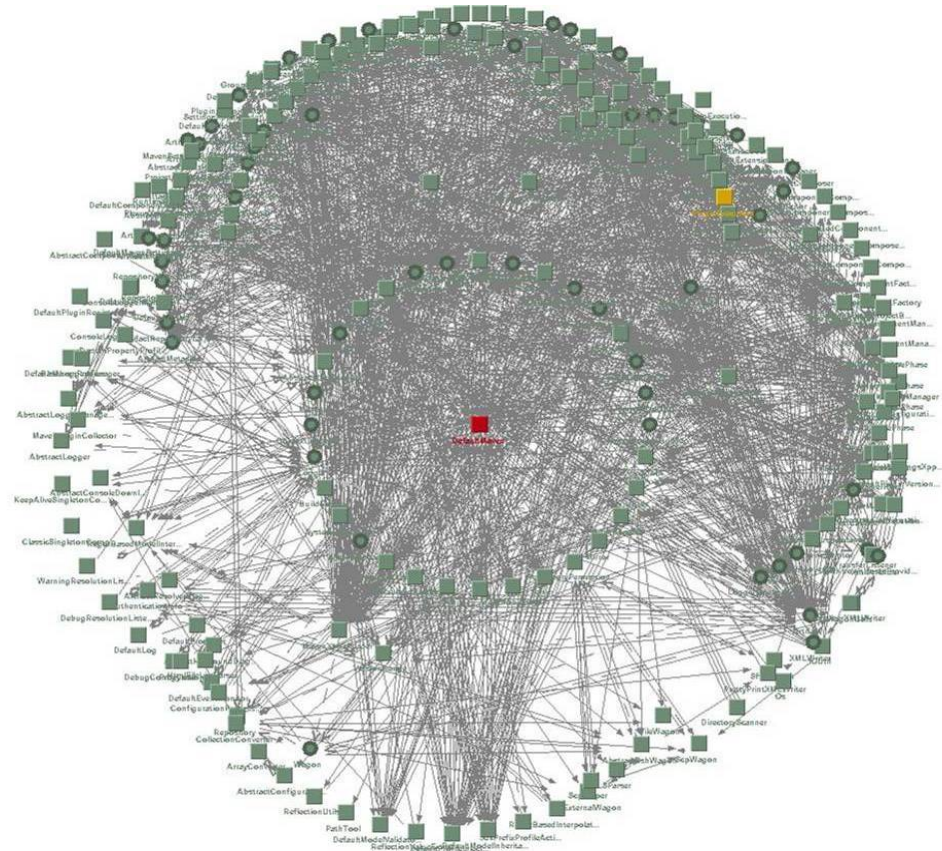
## Software wird immer größer und komplexer

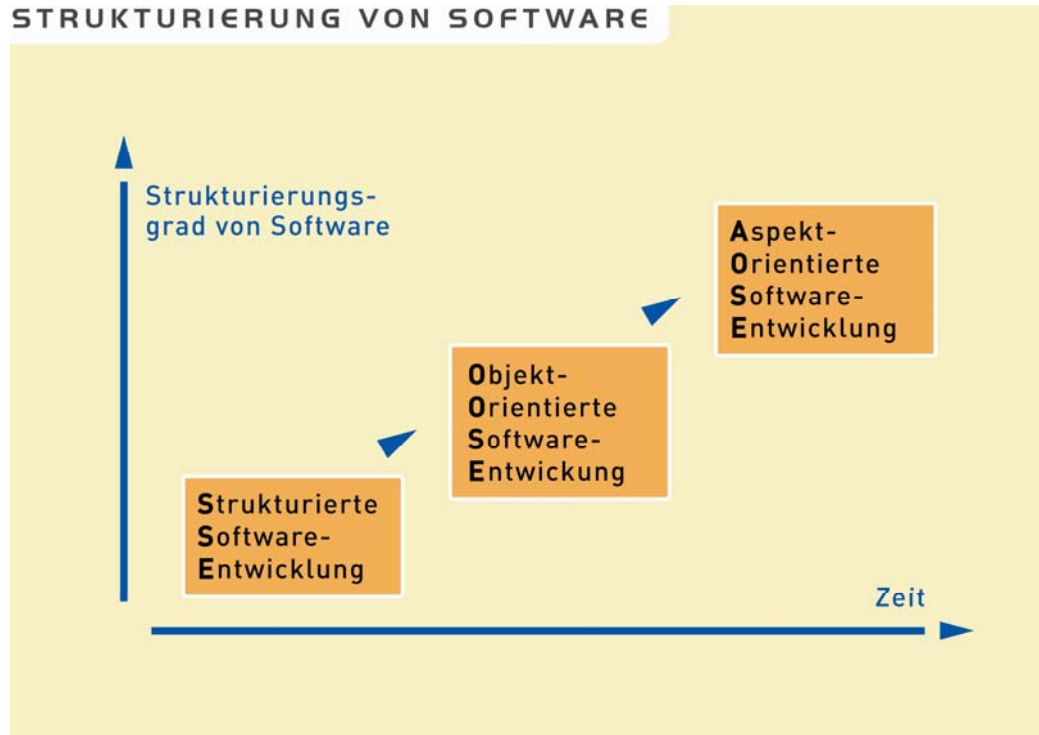
- x| Komplexe und dynamische Geschäftsprozesse
- x| Großer Funktionsumfang
- x| Hoher Bedienkomfort
- x| Integration mit anderen Systemen

## GRÖSSE VON SOFTWARE-SYSTEMEN



- x| Grenzen der Strukturierungsmöglichkeiten von OO
- x| Klassen bilden eng verwobene Abhängigkeitsnetze
- x| Verbesserungen durch:
  - | Komponenten (OSGi)
  - | Services (SOA)





- x| SSE strukturiert einzelne Befehle in definierten, wiederholbaren Blöcken (Schleifen, Funktionen, ...)
- x| OOSE strukturiert Daten mit ihrem dazugehörigem Verhalten in Objekten
- x| AOSE strukturiert Blöcke/Daten/Verhalten/ Objekte in Aspekten
- Die Einführung neuer Strukturierungsebenen ermöglicht die Entzerrung komplexer Konstrukte aus den darunterliegenden Ebenen

# Bessere Software durch AOP?

*„Aspect-oriented programming (AOP), and aspect-oriented software development (AOSD) attempt to aid programmers in the separation of concerns, specifically cross-cutting concerns, as an advance in modularization.“*

Wikipedia

*„Aspects are one kind of concern in software development. With respect to a primary or dominant decomposition aspects are concerns that crosscut that decomposition.“*

aosd.net

*„A concern is an area of interest or focus in a system.“*

aosd.net

*„A concern is any code related to a goal, feature, concept, or ,kind‘ of functionality.“*

Clarke, Baniassad

## *The tyranny of the dominant decomposition*

- x|** Bekanntes Problem bei traditionellen Sprachen und Systemen
- x|** Modularisierung ist immer nur in einer Dimension möglich
  - | OO => Klasse
  - | OSGi => Bundle
- x|** Übergreifende Themen (cross-cutting concerns) werden über das gesamte System verteilt
- x|** Aspektorientierung ermöglicht Modularisierung auf orthogonalen Dimensionen

## Modularisierung == Separation of Concerns

- x| Ein Concern ist modular, wenn:
  - | Sein Code lokal zusammenhängend ist
  - | Er ein wohldefiniertes Interface hat, das beschreibt, wie der Concern mit dem Rest des Systems interagiert
  - | Das Interface eine Abstraktion der Implementierung ist, und diese ausgetauscht werden kann
  - | Ein automatischer Mechanismus die Einhaltung des Interfaces sicherstellt
  - | Der Concern mit anderen Concerns automatisiert zusammengesetzt werden kann, um ein vollständiges System zu erzeugen

- x| Aspekte haben zum Ziel, Teile eines Softwaresystems zu erfassen, die mit den regulären Mitteln der Modularisierung über das System verteilt wären
- x| Wie Aspekte definiert und ausgeführt werden, ist abhängig von der Sprache und Systemumgebung
- x| Concerns können statische und dynamische Einflüsse haben

- x| Concern:** Ein Belang oder Bereich in einem Softwaresystem. Wird i.d.R. als Anhaltspunkt für Modularisierung der Software verwendet.
- x| Crosscutting:** Die Manifestation eines Concerns, der die (dominante) Dekomposition durchbricht.
- x| Joinpoint:** Elemente, mit denen Aspekte interagieren können. Abhängig von Sprache und AO-Engine.
- x| Pointcut:** Eine Selektion bestimmter Joinpoints.
- x| Advice:** Ratschlag, der an bestimmten Pointcuts von der AO-Engine verwoben (eingebaut) wird.
- x| Aspect:** Gruppiert Pointcut-Definitionen und Advices, die zu einem (crosscutting) Concern gehören.
- x| Weaving:** Das Zusammensetzen von Aspekten mit anderen Aspekten und den Basismodulen.

# Bessere Software durch AOP?

- x|** Spracherweiterung von Java
  - | AOP auf Java Code-Ebene
  - | Realisierung (Weaving) durch Bytecodemanipulation, entweder durch Compiler oder durch ClassLoader zur Laufzeit
  
- x|** Joinpoints sind Ereignisse bei der Ausführung von Java Code
  - | Aufrufe von Methoden und Konstruktoren
  - | Ausführung von Methoden und Konstruktoren
  - | Lesen und Schreiben von Membervariablen
  - | Initialisierung von Klassen und Objekten
  - | Ausführung eines Exception Handlers
  
- x|** Inter-type Declarations für statische Ergänzungen von Klassen
  - | Ergänzung neuer Membervariablen und Methoden
  - | Erweiterung der implementierten Interfaces
  - | Definition von Basisklassen

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }

    pointcut changes(Point p): target(p) && call(void Point.set*(int));
    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }

    static void updateObserver(Point p, Screen s) {
        s.display(p);
    }
}
```

- x| Proxy-basierter Ansatz (Java-Interface Proxies oder CGLIB)
  - | Plain Java zur Laufzeit
- x| Arbeitet nur auf Spring-managed Beans
- x| Einzige Art von Joinpoints:
  - | Ausführung von Methoden
- x| Syntaxvarianten:
  - | @AspectJ
  - | XML

```
@Aspect
public class BeforeExample {

    @Before("execution(* com.xyz.myapp.dao.*.*(..))")
    public void doAccessCheck() {
        // ...
    }
}
```

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..)) and this(service)"/>
        <aop:before pointcut-ref="businessService" method="monitor"/>
        ...

    </aop:aspect>

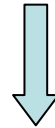
</aop:config>
```

- x| Templatesprache zur Erzeugung textueller Artefakte aus Modellen
- x| Wenige, einfache Sprachkonstrukte
  - | DEFINE: Definiert ein Template für eine Art von Modellelement
  - | EXPAND: Delegiert in ein Unter-Template
  - | FOR, FOREACH, IF: Kontrollstrukturen
  - | <<, >>: Bezug auf Modellelemente und Funktionen
  - | Alles andere wird als Klartext übernommen

```
<<DEFINE Entity FOR data::Entity>>  
  <<FILE baseClassName() >>  
    // generated at <<timestamp()>>  
    public abstract class <<baseClassName()>> {  
      <<EXPAND Impl>>  
    }  
  <<ENDFILE>>  
<<ENDDEFINE>>  
  
<<DEFINE Impl FOR data::Entity>>  
  <<EXPAND GettersAndSetters>>  
<<ENDDEFINE>>  
  
<<DEFINE Impl FOR data::PersistentEntity>>  
  <<EXPAND GettersAndSetters>>  
  public void save() {  
  }  
<<ENDDEFINE>>
```

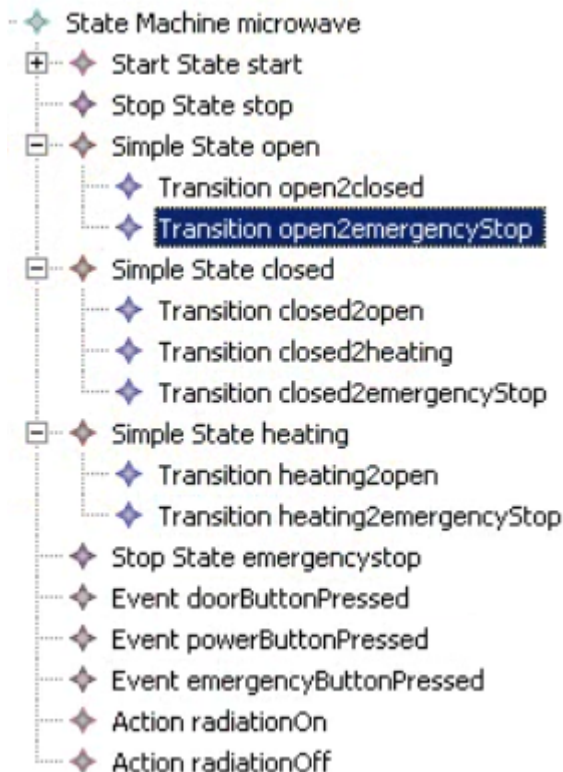
- x| Templates sind Joinpoints
- x| Können durch Aspekt-Templates advised werden

```
«AROUND Impl FOR data::Entity»  
  «FOREACH attribute AS a»  
    public static final AttrInfo «a.name»Info = new AttrInfo(  
      "«a.name»", «a.type».class );  
  «ENDFOREACH»  
  «targetDef.proceed()»  
«ENDAROUND»
```



```
public abstract class PersonImplBase {  
    public static final AttrInfo  
        nameInfo = new AttrInfo("name", String.class);  
    public static final AttrInfo  
        name2Info = new AttrInfo("name2", String.class);  
    private String name;  
    private String name2;  
  
    public void setName(String value) {  
        this.name = value;  
    }  
}
```

- x| Statisches Merging von Modellen
- x| Joinpoints: Metamodellelemente
- x| Pointcut-Definition mit Expressions



```

: m.states.typeSelect(SimpleState);
states(StateMachine m) : m.states.typeSelect(SimpleState);
t ) : transName( Transition t ): ((State)t.eContainer).name+"2emergencyStop

```

# Bessere Software durch AOP?

**x|** Wann wird Software besser?

**x|** Allgemeine Kriterien:

- | Qualität der Entwickler
- | Qualität des Projektmanagements
- | Effizienz des Prozesses und der Tools
- | ...

**x|** AO-relevante Kriterien

- | Hoher Grad an Modularisierung
- | Reduktion der Komplexität
- | Geringes Fehlerpotenzial
- | Akzeptanz von AO

- x| Vollständige, mächtige AO-Realisierung für Java
- x| Stabil, praxiserprobt
- x| Gute Tools
  
- x| Aber die Komplexität...
  - | 11 Typen von Joinpoints
  - | Jede Java-Klasse ist advise-bar
  - | 25+6 Schlüsselwörter für Pointcut Expressions
  - | Signaturmatching mit Wildcards
  - | 6 Schlüsselwörter für Advices
  - | 6 Schlüsselwörter für Intertype declarations, etc.
  - | 7 Schlüsselwörter für Aspektdeklaration
  - | 14 Annotations für @AspectJ Syntax

### x| Gefahr der „Un“Strukturierung

- | Pointcuts an prinzipiell allen Stellen des Java-Programms möglich
- | Aspekte haben Zugriff auf private Members
- | Verhalten von Methoden kann durch Aspekte verändert werden
- | OO-Kapselung kann komplett unterwandert werden
- | Der Contract von Interfaces kann gebrochen werden

### x| Hohes Fehlerpotenzial

- | Definition nicht-trivialer Pointcuts kann schwierig sein
- | Unbemerkte Selektion nicht beabsichtigter Joinpoints
- | Überblick nur mit Toolunterstützung

- x| Pragmatische statt vollständige AO-Lösung für Java
- x| Wesentlich geringere Komplexität
  - | 1 Typ von Joinpoint
  - | Nur Spring-managed Beans advise-bar
  - | 5+4 Schlüsselwörter für Pointcuts
  - | Signaturmatching mit Wildcards
  - | 6 Schlüsselwörter für Advices
  - | 1 Schlüsselwort für Inter-type declarations, etc.
  - | 3 Schlüsselwörter für Aspektdeklaration
- x| Geringerer Umfang erfordert weniger Einarbeitung
  
- x| Aber:
  - | Schlechte Toolunterstützung für @AspectJ und XML Syntax
  - | Proxy-Ansatz birgt subtile Laufzeitkuriositäten
  - | Fehlererkennung oft erst zur Laufzeit

- x| Aspektorientierung auf Template-Ebene, nicht auf Java-Code
- x| Andere Abstraktionsebene mit inhärent geringerer Komplexität
- x| Sehr geringe Komplexität der AO-Erweiterung:
  - | 1 Typ von Joinpoint
  - | Nur Templates advise-bar
  - | 1 neues Schlüsselwort
  - | Sehr einfache Wildcards zur Selektion von Pointcut-Templates
- x| Sehr kurze Einarbeitungszeit
  
- x| Aber:
  - | Schlechte Toolunterstützung für AO-Features
  - | Detailverbesserungen notwendig

- x| Aspektorientierung auf (Meta-)Modellebene
- x| Dadurch hohe Abstraktion
- x| Mit DSLs ist die Abstraktionsebene sogar frei wählbar
- x| Geringe Komplexität:
  - | Keine neuen Schlüsselwörter
  - | Einfache Wildcards für Pointcut-Definition im Modell
  - | Standard XPand Expressions zur Pointcut-Selektion
  
- x| Aber:
  - | Sehr junges Gebiet
  - | Erfordert andere Denkweise der Entwickler
  - | Integration von Modellen und Java-Code ist anspruchsvoll

- x| Akzeptanz von Aspektorientierung teilweise erschreckend schlecht
  - | Schlechte Ergebnisse vorprogrammiert
  
- x| Häufige Reaktionen:
  - | Ablehnung
    - „Wozu brauch ich das?“
    - „Das macht doch alles komplizierter!“
  - | Angst
    - „Darf ich jetzt nicht mehr Java machen?“
    - „Das kapiert ich nicht!“
  
- x| AOP wird als Ablösung von OOP wahrgenommen, nicht als Ergänzung
  
- x| AOP ist ein Paradigmenwechsel
  - | Ähnlich wie beim Wechsel von SP zu OOP bleiben einige „auf der Strecke“

# Bessere Software durch AOSD!

## Aspect-Oriented Software Development

- x| AOP leitet einen Paradigmenwechsel ein
  
- x| Aus aspektorientierter Programmierung (AOP) wird aspektorientierte Softwareentwicklung (AOSD)
  - | Zunächst Manifestierung in Programmiersprachen
    - AspectJ, Spring AOP, JBoss AOP, Aspect#, AspectC++
  - | Dann Arbeit und Forschung an AO-Theorie
    - Universitäten, Publikationen, Konferenzen, Bücher, ...
  - | Einzug von AOSD in der breiten Praxis
    - > 2010

## Ubiquitous AOSD

- x|** Höhere Wahrnehmung der Aspektorientierung
  - | Festigung der AO-Konzepte und Begriffe
  - | Aufnahme von AO in das Curriculum der Hochschulen
  - | Präsenz durch Publikationen
  - | „Thinking in Aspects“
  
- x|** Höhere Abstraktion von AO
  - | Weg von AOP im Code
  - | Verbindung von AOSD und MDSD
  - | „Early Aspects“ in Analyse und Designphase

## x| Bei mir:

- | Hier und jetzt
- | Im Ausstellungsbereich
- | Email: [a.demelt@excellent.de](mailto:a.demelt@excellent.de)

## x| Im Internet:

- | <http://aosd.net>
- | <http://www.eclipse.org/aspectj>
- | <http://www.springframework.org/>
- | <http://www.openarchitectureware.org/>

## x| Literatur:

- | Clarke, Baniassad: „Aspect-Oriented Analysis and Design“
- | Jacobsson, Ng: „Aspect-Oriented Software Development with Use Cases“