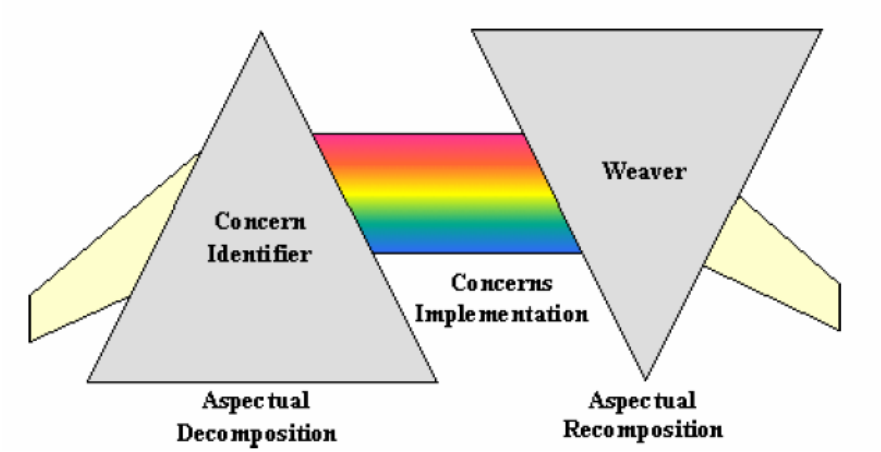
Aspektorientierte Softwareentwicklung



Eine Einführung mit Schwerpunkt Aspektorientierte Programmierung

Modularisierung

- **Vorteile**
  - Bessere **Bewältigung der Systemkomplexität**
  - Bessere **Wiederverwendung** von Teilen
  - **Kein redundanter Quellcode**
  - **Bessere Planbarkeit in der Entwicklung**

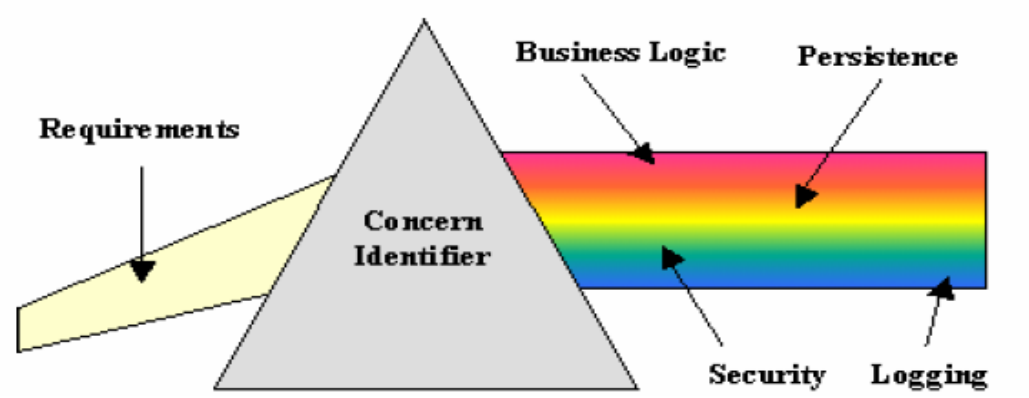
- Designprinzipien
  - *Divide and Conquer ➔ OO!*
    - Funktionale Dekomposition (eindimensional)
  - *Information Hiding / Problem Hiding ➔ OO!*
    - Trennung von Schnittstelle und Implementierung
  - ***Separation of Concerns ➔ OO?***
    - Klare Zuordnung von Zuständigkeiten auf einzelne Module

Die Prisma Metapher

Concern

„… something that relates or belongs to one … matter for consideration … marked interest or regard …“ (Merriam Webster)

“... those interests which pertain to the system’s development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.” (IEEE [6])



- Wie modularisiert man die einzelnen Zuständigkeiten?
- Wie ermöglicht man lose Kopplung zwischen den Zuständigkeiten?

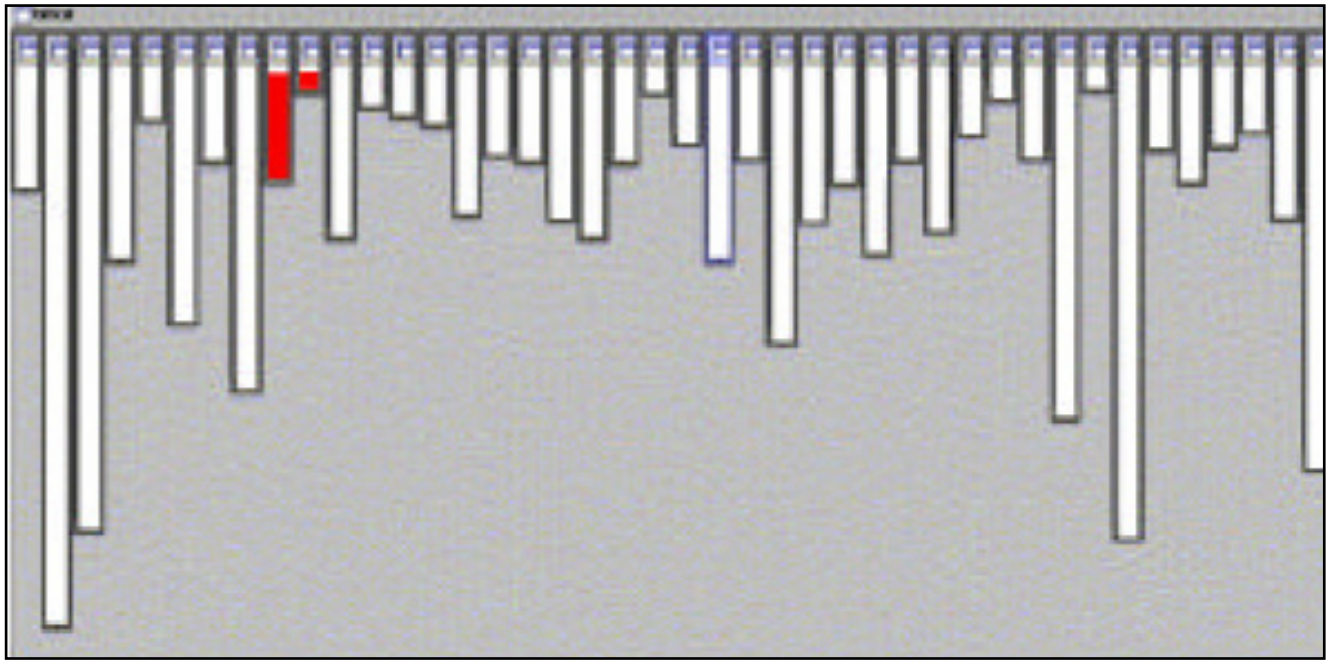
Abbildung von Zuständigkeiten auf Programmeinheiten

- **Variante 1: Isomorphismus**

- **Vorteile einer objektorientierten Lösung**
  - Modularisierung der Zuständigkeiten durch OO-Mittel wie Klassen und Namensräume möglich.
  - Einfach zu lokalisieren und zu modifizieren.
  - Hohe Kohäsion

Beispiel: Isomorphismus

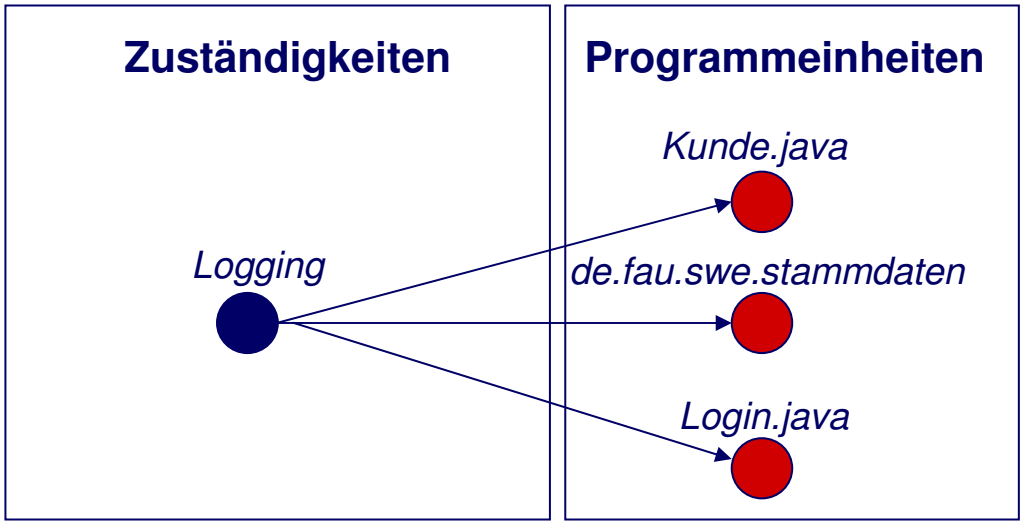
- **URL Pattern Matching im Apache Tomcat Server**



- Vertikale Balken = Quellcode-Pakete
- Rote Färbung = Lokalisierung der Zuständigkeit

Abbildung von Zuständigkeiten auf Programmeinheiten

- **Variante 2: Tangling (Verstreuung)**



- **Vorteil einer objektorientierten Lösung**
  - Zugriff auf Kontext einfach möglich (z.B. Logging kundenspezifischer Daten)
- **Nachteile einer objektorientierten Lösung**
  - Modularisierung der Zuständigkeit durch OO-Mittel nur in Ausnahmefällen möglich (z.B. per Vererbung).
  - Schwer zu lokalisieren und zu modifizieren.

Beispiel: Tangling

- **Logging im Apache Tomcat Server**

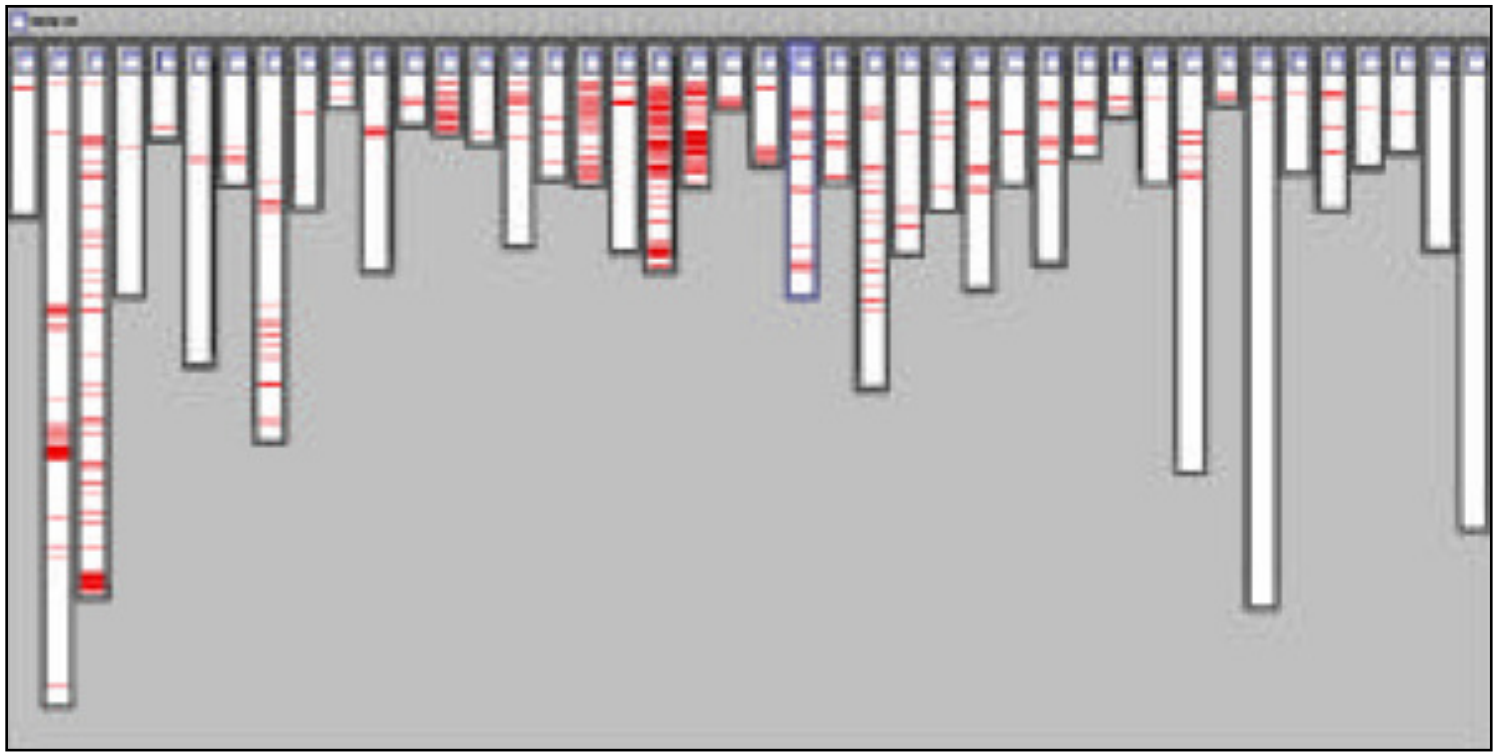
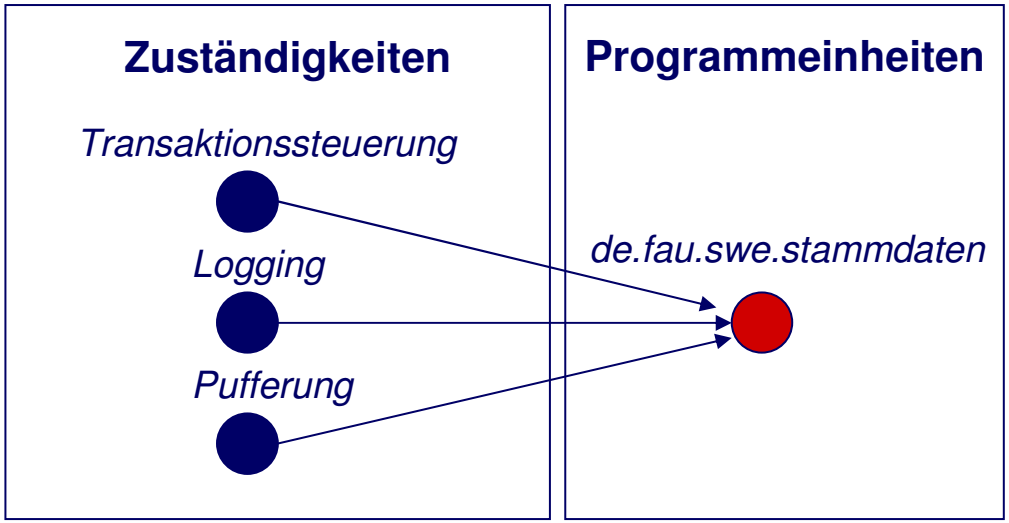


Abbildung von Zuständigkeiten auf Programmeinheiten

- **Variante 3: Scattering (Vermischung)**



- **Vorteil einer objektorientierten Lösung**
  - siehe Tangling
- **Nachteile einer objektorientierten Lösung**
  - Modularisierung der Zuständigkeit durch OO-Mittel nur in seltenen Ausnahmefällen möglich (z.B. per Mehrfach-Vererbung oder Decorator-Muster).
  - Extrem schwer zu modifizieren, da Entwickler viele Zuständigkeiten kennen muss
  - Ungewollte Wechselwirkungen zwischen den Zuständigkeiten
  - Niedrige Kohäsion

Auswirkungen Tangling & Scattering

Software Slums

Inhomogene Infrastruktur

Spaghetti Code

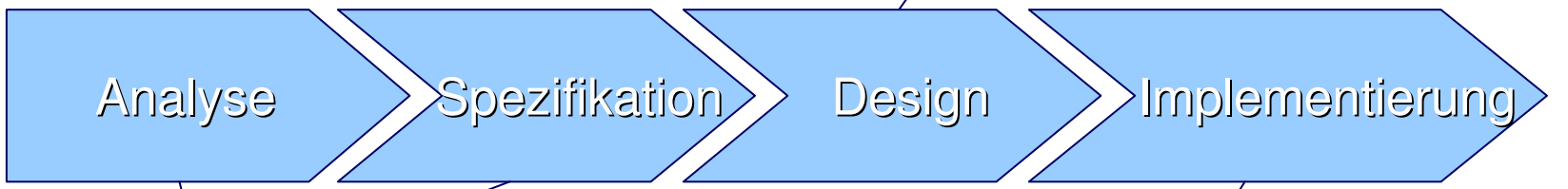
Undurchsichtige Programmstruktur

Crosscutting Concerns

- Zuständigkeiten, die sich per *Tangling* oder *Scattering* auf OO-Programmeinheiten abbilden, nennt man *Crosscutting Concerns*.
  - Adressieren oft nicht-funktionale Anforderungen und systembezogene Anforderungen.
- Isomorph abbildbare Zuständigkeiten nennt man *Common Concerns*.

- Lösungsansätze zur Modularisierung von Crosscutting Concerns:
  - Design Patterns
    - z.B. Observer für Logging und Sicherheitsprüfungen
    - z.B. Command für Undo-/Redo-Mechanismus
    - z.B. Decorator für die Anreicherung von Funktionalität
  - Inversion of Control
    - Indirektion des Kontrollflusses über einen Container (z.B. EJB Container) oder Interpreter.
    - Anpassung des Verhaltens über Metadaten.
  - **Aspektorientierte Programmierung ➔**

Aspektorientierung im Entwicklungsprozess

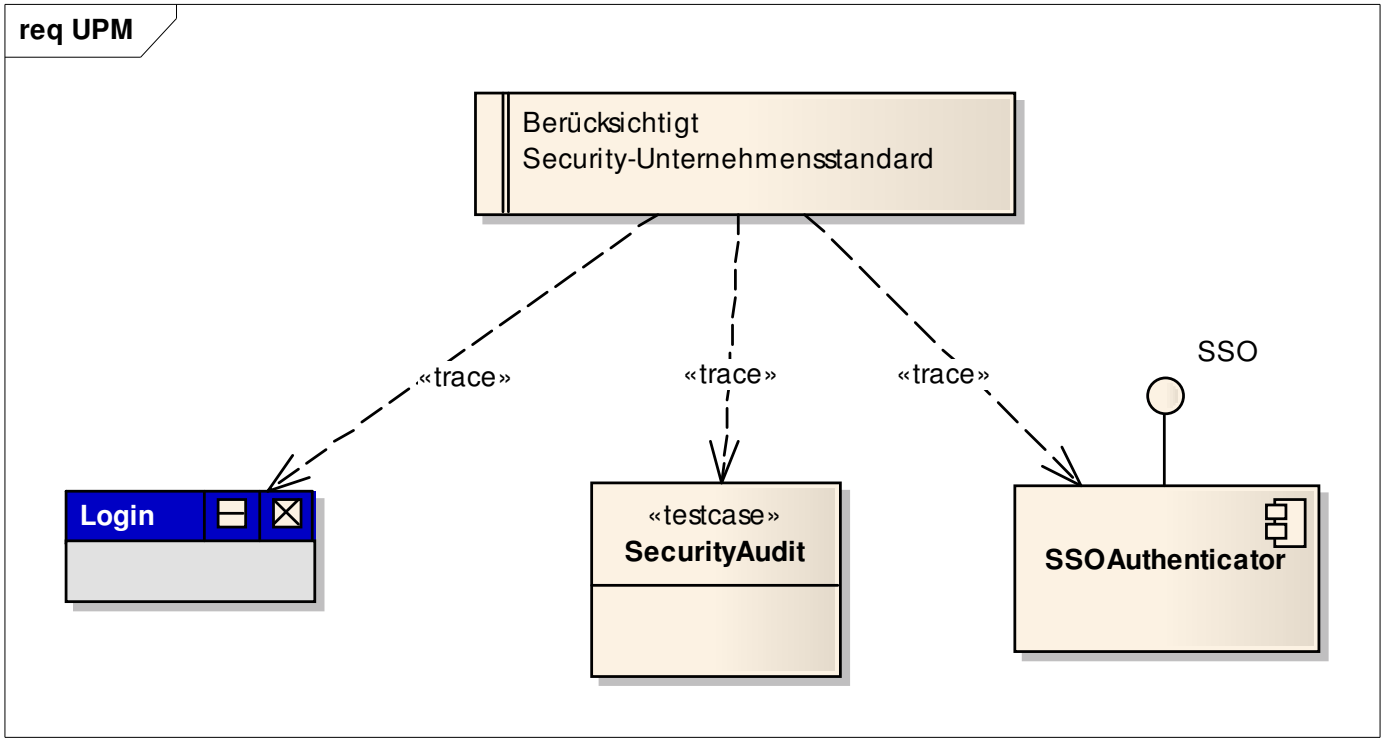


Aspect-Oriented Modelling
Aspekte und ihre Wechselwirkungen mit OO-Designelementen modellieren

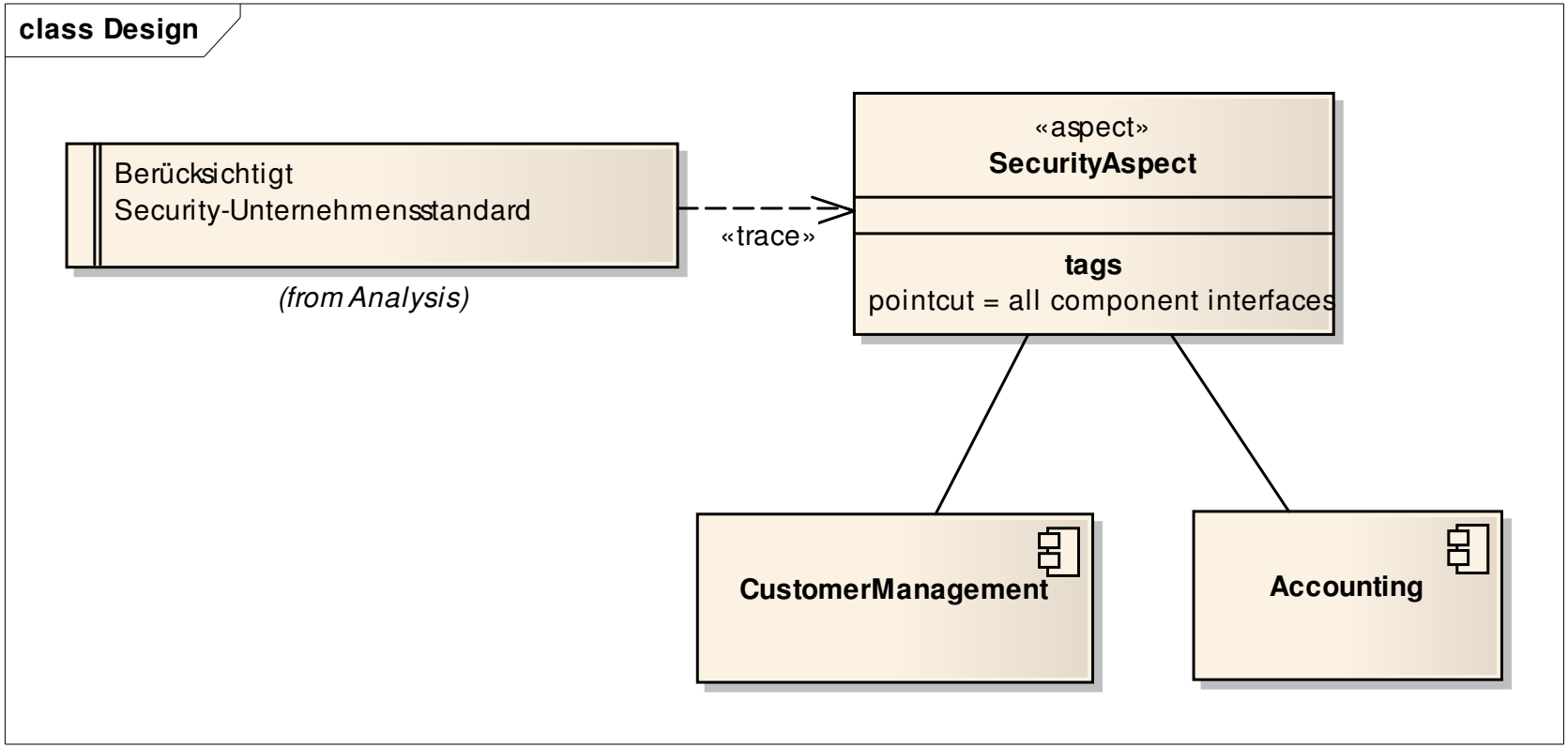
Early Aspects
Crosscutting Concerns identifizieren und beschreiben

Aspect-Oriented Programming
Aspekte implementieren und kompilieren

Aspekte in der Analysephase



Aspekte in der Designphase



Aspekt-orientierte Programmierung (AOP)

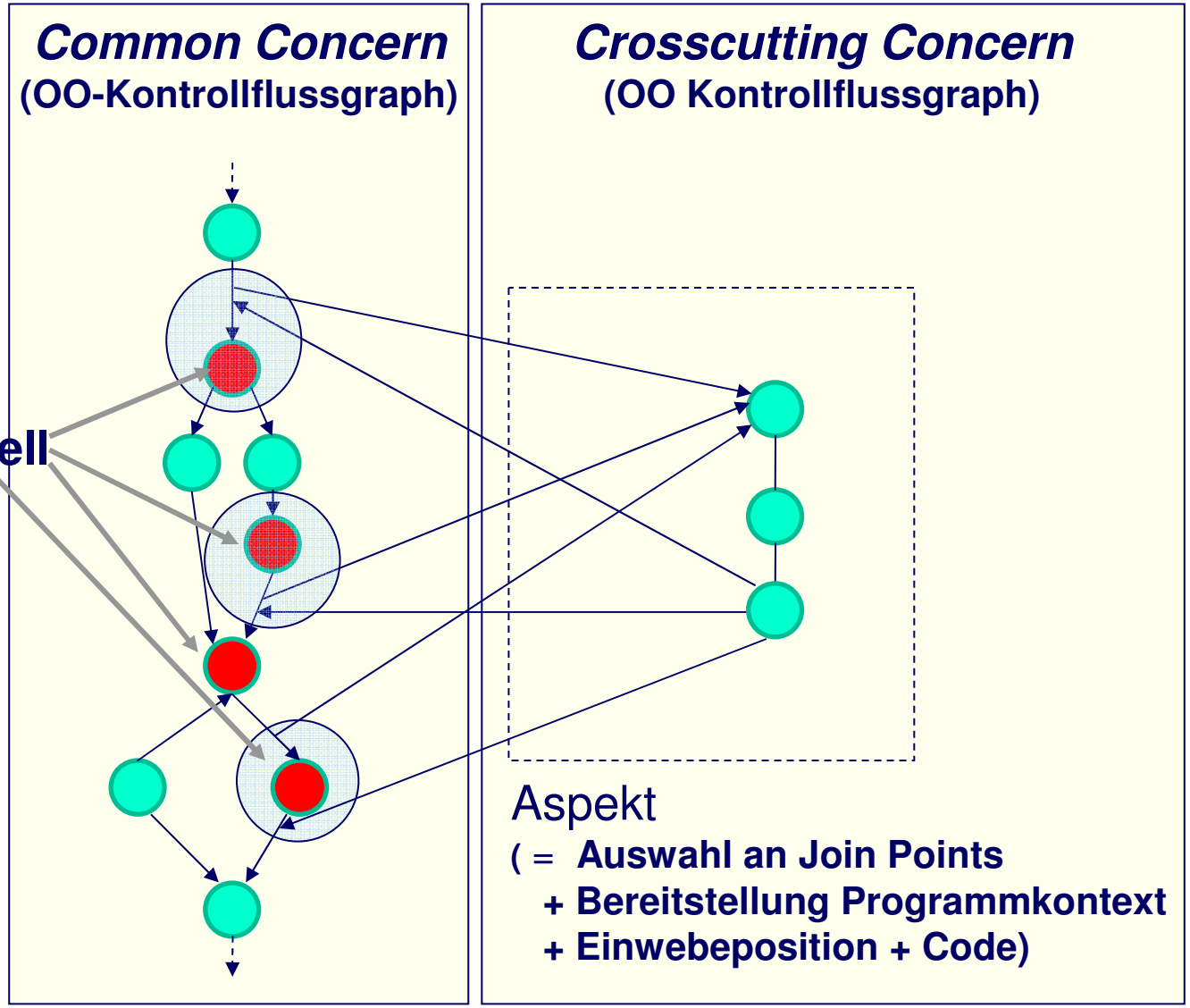
- Am *Xerox Palo Alto Research Lab* (PARC) Mitte der 90er Jahre unter Federführung von Gregor *Kiczales* entstanden.
- Ende der 90er Jahre: AOP-Implementierung AspectJ für Java durch gleiches Team.

- **Motivation**: Untersuchung der Grenzen der objektorientierten Entwicklng und Entwicklung von Ansätzen zur Steigerung der Produktivität durch vereinfachte Modularisierbarkeit von Code.

- **Zentrale Ideen:**
  - Unterscheidung
    - **Komponenten: Modularisierung von Common Concerns**
    - **Aspekte: Modularisierung von Crosscutting Concerns**
  - Bereitstellung von Regeln, um Aspekte und Komponenten miteinander kombinieren zu können.

AOP: Das Konzept

Join Point Modell
(Auswählbare Knotentypen)



AspectJ: Die Sprache

- Zentrales Sprachkonstrukt: **Aspect**
  - Kombination aus Java-Code und einer java-ähnlichen Aspektbeschreibungs-Syntax.

Das Aspekt-Grundgerüst

```
public aspect MannersAspect {

public void greet(){
        System.out.println("Good Day");
        }

public void sayGoodbye(){
        System.out.println("Thank you. Goodbye!");
        }
}
```

Die Anwendung

```
public class Talker {

public static void say(String message){
        System.out.println(message);
        }
}
```

Sprachkonstrukt: **Pointcut**

- **Wählt eine Menge an Joinpoints aus („Abfangen“).**
- Beschreibt den Kontext, der übergeben wird (mehr dazu später…)

```
public aspect MannersAspect {

pointcut callSay():call(public static void Talker.say(String));

public void greet(){
        System.out.println("Good Day");
        }

public void sayGoodbye(){
        System.out.println("Thank you. Goodbye!");
        }
}
```

```
public class Talker {

public static void say(String message){
        System.out.println(message);
        }
}
```

Sprachkonstrukt: **Pointcut**

```
pointcut callSay():call(public static void Talker.say(String));
```

Name für spätere Referenzierung

Join Point Typ

Signatur (Einschränkung)

Joinpoint Typ	Syntax
Methode-Aufruf	***call(Methoden-Sigantur)***
Methoden-Ausführung	***execution(Methoden-Sigantur)***
Objektinitialisierung	***initialization(Konstruktur-Signatur)***
Lesender Feldzugriff	***get(Feld-Signatur)***
Schreibender Feldzugriff	***set(Feld-Signatur)***
Exception Handler Ausführung	***handler(Typ-Signatur)***

- **Kombination von Signaturen per „||“ möglich.**
- **Einsatz von Wildcards möglich:**

Wildcard	Bedeutung
*	beliebige Anzahl von Zeichen außer dem Punkt.
..	beliebige Anzahl von Zeichen einschließlich von Punkten.
+	alle Subtypen des geg. Typs.

Sprachkonstrukt: **Pointcut**

- **Zusammengesetzte Pointcuts**
  - Verknüpfung von mehreren Pointcuts
  - Alle bool'schen Operatoren verfügbar

```
pointcut methodsToTrace() :
        execution(public * de.fau.i11..*(..)) &&
        !execution(public * *.set*(..)) &&
        !execution(public * *.get*(..));
```

Wählt alle Join Points „Methodenausführung" innerhalb aller Klassen des Pakets de.fau.i11 und seiner Unterpakete aus.
Nicht jedoch Methodenausführungen von Methoden, deren Name mit „get" oder „set" beginnt.

Sprachkonstrukt: **Advice**

- Sprachkonstrukt: **Advice**
  - Verbindung zwischen einem Pointcut und der Implementierung des Crosscutting Concerns.

```
public aspect MannersAspect {

    pointcut callSay():call(public static void Talker.say(String));

    before(): callSay() {
        greet();
    }
    after(): callSay()  {
        sayGoodbye();
    }
}
```

```
// Alternative:
void around(): callSay() {
        greet();
        proceed();
        sayGoodbye();
}
```

Bereitstellen von Kontext

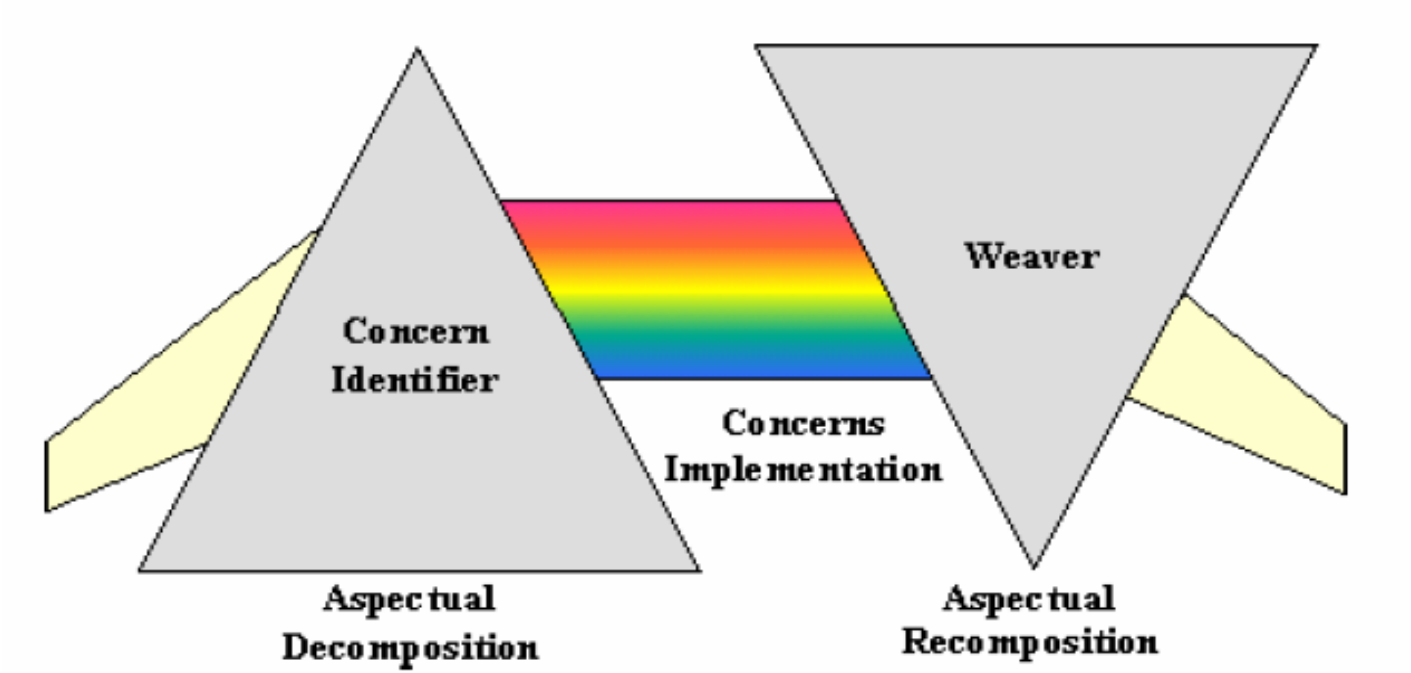
- Methoden-Parameter können an Pointcut gebunden werden. Dies wirkt auch als zusätzliche Bedingung.
- Advice muss dann die gebundenen Parameter in seine Signatur aufnehmen ➔ Parameter steht zur Verfügung.

```
pointcut callSay(String message):call(* *.say(..)) && args(message);

before(String msg): callSay(msg) {
        greet();
}

after(String msg): callSay(msg)  {
        System.out.println("and the second time: " + msg);
        sayGoodbye();
}
```

AOP Einsatzgebiete

- Fehlerbehandlung
- Logging / Tracing
- Sicherheitsprüfungen
- Transaktionssteuerung
- Persistenz
- Caching und Replikation
- Verifikation, Policy Enforcement
- Nebenläufigkeitskontrolle
- Profiling
- …

Die Prisma Metapher (rev.)



- Dekomposition von *Crosscutting Concerns* und *Common Concerns*.
- Modularisierung der *Crosscutting Concerns* durch Aspekte
- Rekomposition durch einen *Weaver*
  - ➔ Produziert ausführbaren Programmcode.
  - ➔ Ist Teil des Kompilierens oder Ladens einer Software.

Vorteile AOP

- Durch die AOP ist es möglich, Crosscutting Concerns
  - … besser zu definieren
  - … mit umfassenderer Funktionalität zu entwickeln
  - … besser planen zu können
  - … konsistenter implementieren zu können

  und die sonstigen Vorteile der Modularisierung zu erhalten.

Nachteile AOP

- Wiederverwendbarkeit und Stabilität der Aspekte hängt stark von deren Implementierung ab (richtige Wahl der Abstraktion).
- Ein Crosscutting Concern kann Seiteneffekte auf Common Concerns haben.
  - Die Kapselung von Modulen kann gebrochen werden.
  - Die Semantik eines Moduls ist nicht mehr allein dem Quellcode des Moduls zu entnehmen.
- Das Testen und Verifizieren eines Systems mit Aspekten ist deutlich komplizierter (z.B. Nachweis, dass keine Seiteneffekte existieren).
- Verminderte statische Analysierbarkeit des Kontrollflusses

Quellen

- **[1] Gregor Kiczales et al., „Aspect-Oriented Programming“**, Proceeding of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
- **[2] Gregor Kiczales et al., „An Overview of AspectJ“**, Lecture Notes in Computer Science, Volume 2072, 327—355. 2001.
- **[3] Stanley M. Sutton Jr., Peri Tarr, „Aspect-Oriented Design Needs Concern Modeling“,** Workshop AODS 2002
- **[4] IEEE. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.** IEEE Std. 1471- 2000. Approved 21 September 2000.
- **[5] Ramnivas Laddad, „Aspect-Oriented Programming Will Improve Quality“**, IEEE Software, Quality Time, 0740 – 7459. 2002.
- **[6] Ramnivas Ladda, „AspectJ in Action“,** Manning, 2003
- **[7] Ramnivas Laddad, „AOP myths and realities“,** IBM developerWorks AOP@Work series, 2006 http://www-128.ibm.com/developerworks/java/library/j-aopwork15
- **[8] The AspectJ Language,** http://www.eclipse.org/aspectj/doc/released/progguide/language.html

Tutorial AspectJ: Ziele

- Ziele:
  - Erste Gehversuche mit AspectJ
  - Eclipse AspectJ Toolset kennenlernen
  - Ausgangsbasis für eigene Versuche / Beispiele
- Übungen:
  - *Talker/Manners:* Erstellen eines einfachen Aspekt / Klasse Paars
  - *Simple Tracing:* Anwendung eines Aspekts auf viele Klassen und Analyse der Advices.
  - *UPMBugPatch*: Manipulation von (falschem) Programmverhalten

Tutorial: AspectJ

- **Vorbereitung:**
  - Account einrichten und eigenes Verzeichnis erstellen (im Home-Verzeichnis)
  - Eclipse 3.3 auf Rechner kopieren
  - Plugins installieren (AspectJ Tools, Subversive)
  - Workspace anlegen
  - UPM Beispielanwendung aus Subversion-Repository holen: https://upm.svn.sourceforge.net/svnroot/upm
  - Source-Folder *aspects* erstellen (hier wird der Übungscode abgelegt)
  - Projekt auf AspectJ Projekt umstellen

- **Hilfestellungen:**
  - Eclipse Help
  - Vortragsfolien
  - Dozenten ;-)

Tutorial: Übung 1

- **Talker/Manners-Beispiel** aus dem Vortrag implementieren
  - Klasse *Talker* erstellen (New-Dialog) – inkl. *main()* Methode
  - Aspect *Manners* erstellen (New-Dialog)
  - Klasse Talker als AspectJ Anwendung laufen lassen

```
public aspect MannersAspect {

pointcut callSay():call(public static void Talker.say(String));

public void greet(){
        System.out.println("Good Day");
        }

public void sayGoodbye(){
        System.out.println("Thank you. Goodbye!");
        }
}
```

```
public class Talker {

public static void say(String message){
        System.out.println(message);
        }
}
```

Tutorial: Übung 2

- Implementierung eines **Tracing-Aspekt**s:

```
public aspect SimpleTracingAspect {
    pointcut methodsToTrace()
    : execution(public * com._17od.upm..*(..));


    before() : methodsToTrace(){
        System.out.println("Enter: " + thisJoinPoint.getSignature().toString());
    }

    after() : methodsToTrace(){
        System.out.println("Leave:" + thisJoinPoint.getSignature().toString());
        }
}
```

- Möglichkeiten der Eclipse AspectJ IDE ausnutzen:
  - **Cross References View:** Anzeige aller Advices eines Aspekts nach einem Build.
  - **Advice Markers:** Markierung von Codestellen an denen sich ein Aspekt anlegt mit konfigurierbaren Icons.
  - **Aspect Visualization Perspective:** Grafische Darstellung auf Paketebene, wo sich alle definierten Aspekte anlegen.

Tutorial: Übung 3

- **UPM hat einen Bug** in der Methode *com._17od.upm.util.Util.loadImage(String)*

- Folgender Code müsste anstatt des gegebenen Methodenrumpfes ausgeführt werden:

```
URL imageURL = ClassLoader.getSystemClassLoader().getResource(name);
return new ImageIcon(imageURL);
```

- Vorgehen zur Korrektur:
  - In den *Project Properties* das Verzeichnis „image“ in den *Libraries Java Build Path* legen.
  - Aspekt *UPMBugPatch* erstellen, der um die gg. Methode herum angewendet wird.
  - Siehe Folie „Bereitstellen von Kontext“ zu dem Thema, wie der Methodenparameter übernommen werden kann.