

Aspektorientierte Programmierung - Einführung

von Assembler bis AspectJ

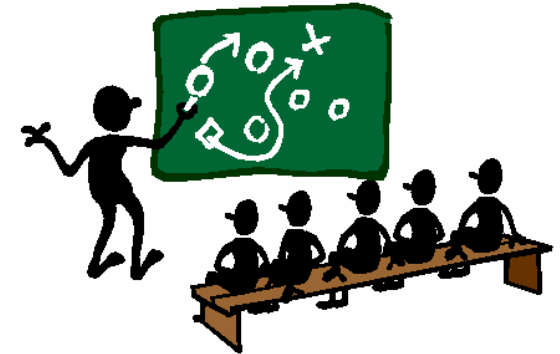
AOP wird als nächstes Glied der Evolutionskette der Komplexitätsbeherrschung diskutiert.

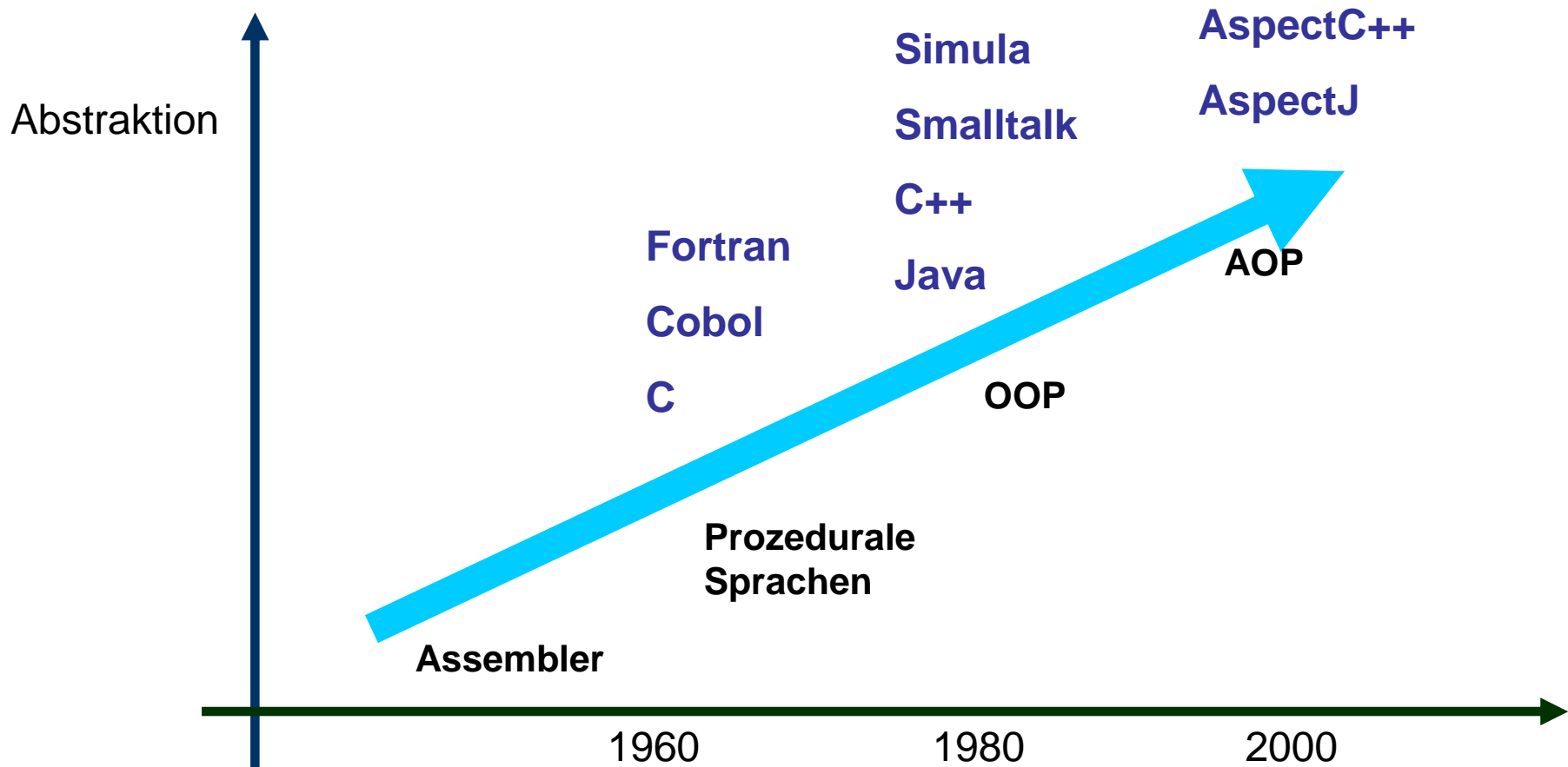
Assembler -> Prozedurale Programmierung-> OOP
-> AOP

AOP modularisiert Aspekte die beim klassischen Design über viele Module verteilt sind.

Bestehende Programmiersprachen können um AOP Fähigkeiten erweitert werden.

z.B. Java -> AspectJ





SIE H,T
AUS W,I,E
ASS EM
BLE R
IS ES
ABE R
NET

- **Viele Dialekte**
Processor-abhängig
HW-abhängig
- **Schnell**
- **Nur von Spezialisten zu verstehen**

Prozedurale Sprachen

Beispiel: Backanleitung

Zwetschkuchen:

Zutaten in eine Schüssel geben
alles verrühren

...

Prozedurale Sprachen:

PASCAL

COBOL C

FORTRAN

`Kuchen=backen(butter, mehl,
zucker, eier, ...)`

OO-Sprachen



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

Kuchen

Zutaten

Größe

add()

backe()

Objektorientierte Sprachen:

Ruby, SIMULA, C++

Python, Java, C#, Smalltalk

Eifel

AO-Sprachen

AOP=OOP + Aspekte

- Logging-Aspekt
- Security-Aspekt
- Transaktion-Aspekt
- ...



Aspektorientierte Sprachen:
AspectJ, AspectS, AspectC++

Notwendigkeit von AOP

Es gibt Designanforderungen, deren Lösung weder mit funktionalen noch mit OO-Sprachen sauber implementierbar ist.

Implementierung von Aspekten im gängigen Programmierstil führt häufig zu:

- **Streuung der Lösungsfunktionalität im Code**
- **Minderung der Lesbarkeit und Wartbarkeit**
- **erschwerter Wiederverwendbarkeit**

Definition: Concern

Spezifische Anforderung oder Gesichtspunkt, welcher in einem Software-System behandelt werden muss, um die übergreifenden Systemziele zu erreichen

Definition: Core Concerns

Realisieren die Kernfunktionalität eines Systems

Definition: Crosscutting Concerns

Realisieren diejenigen Funktionen eines Systems, welche die Core Concerns oder andere Crosscutting Concerns quer schneiden, z.B. Logging



Nichtfachliche Concerns:

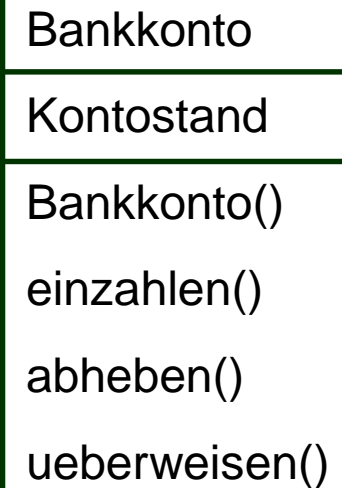
- **Logging**
- **Performance**
- **Autorisierung**
- **Sicherheit**



Fachliche Concerns:

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

```
package aspekt2;
class Bankkonto
{
    int Kontostand;
    Bankkonto(int betrag)
    { Kontostand=betrag;}
    void einzahlen(int betrag)
    { Kontostand+=betrag;
    }
    void abheben(int betrag)
    { Kontostand-=betrag;
    }
    void ueberweisen(int betrag, Bankkonto empfaenger)
    { this.abheben(betrag);
      empfaenger.Kontostand+=(betrag);
    }
}
public class aspekt2 {
    public static void main(String[] args) {
        Bankkonto b=new Bankkonto();
        b.einzahlen(100);
        System.out.println("Kontostand = " + b.Kontostand );
    }
}
```



Code Fehlerbehandlung

```
package aspekt2;
class Bankkonto
{
    int Kontostand;
    Bankkonto(int betrag)
    {
        Kontostand=betrag;
    }
    void einzahlen(int betrag) throws IllegalArgumentException
    {
        if(betrag<0) throw new IllegalArgumentException("Betrag negativ");
        Kontostand+=betrag;
    }
    void abheben(int betrag)
    {
        if (betrag<0) throw new IllegalArgumentException("Betrag negativ");
        Kontostand-=betrag;
    }
    void ueberweisen(int betrag, Bankkonto empfaenger)
    {
        this.abheben(betrag);
        empfaenger.Kontostand+=(betrag);
    }
}
```

```
package aspekt2;
class Bankkonto
{
    int Kontostand;
    Bankkonto(int betrag)
    {
        Kontostand=betrag;
    }
    void einzahlen(int betrag) throws IllegalArgumentException
    {
        if (betrag<0) System.err.println("Fehler");
        if(betrag<0) throw new IllegalArgumentException("Betrag negativ");
        Kontostand+=betrag;
    }
    void abheben(int betrag)
    {
        if (betrag<0) System.err.println("Fehler");
        if (betrag<0) throw new IllegalArgumentException("Betrag negativ");
        Kontostand-=betrag;
    }
    void ueberweisen(int betrag, Bankkonto empfaenger)
    {
        this.abheben(betrag);
        empfaenger.Kontostand+=(betrag);
    }
}
```

**Code vervielfacht
schwerer lesbar**

```
class Bankkonto
{
    int Kontostand;
    Bankkonto(int betrag)
    {
        Kontostand=betrag;}
    void einzahlen(int betrag) throws IllegalArgumentException
    {
        testBetrag(betrag);
        Kontostand+=betrag;
    }
    void abheben(int betrag)
    {
        testBetrag(betrag);
        Kontostand-=betrag;
    }
    void ueberweisen(int betrag, Bankkonto empfaenger)
    {
        this.abheben(betrag);
        empfaenger.Kontostand+=(betrag);
    }
    void testBetrag(int betrag)
    {
        if (betrag<0)
        {
            System.err.println("Fehler");
            throw new IllegalArgumentException("Betrag negativ");
        }
    }
}
```

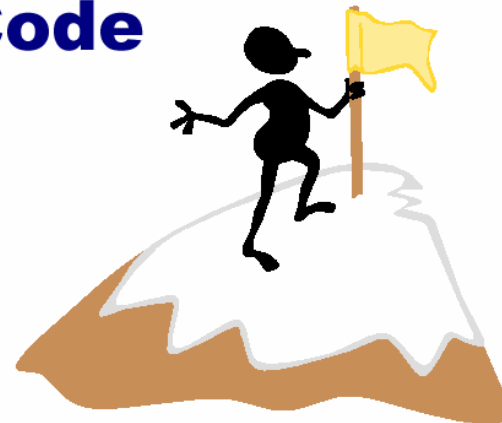
etwas lesbarer

```
class Bankkonto
{
    int Kontostand;
    void einzahlen(int betrag)
    { Kontostand+=betrag;
    }
    void abheben(int betrag)
    { Kontostand-=betrag;
    }
    void ueberweisen(int betrag, Bankkonto empfaenger)
    { this.abheben(betrag);
      empfaenger.Kontostand+=(betrag);
    }
}
```

Bankkonto mit Aspekten!

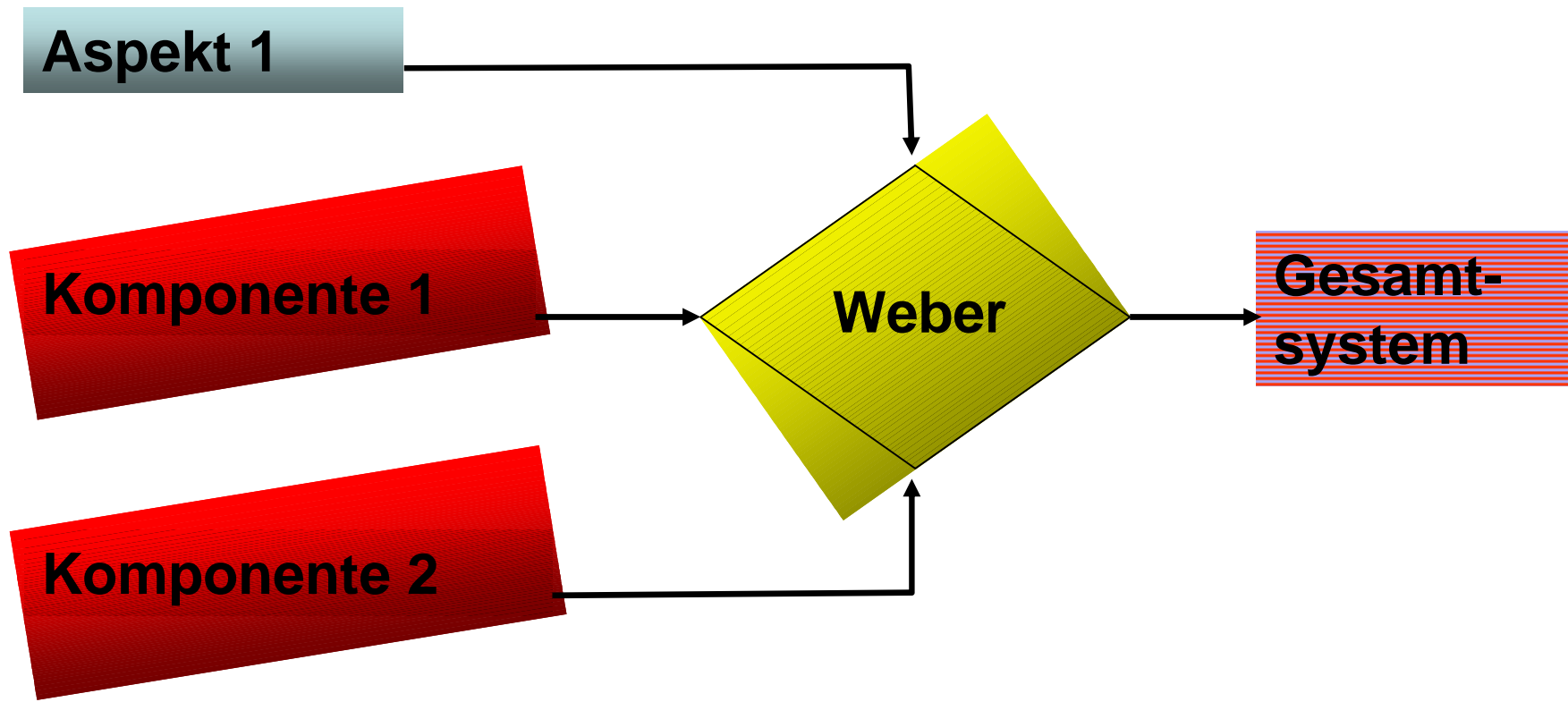
Wo ist die Fehlerbehandlung?

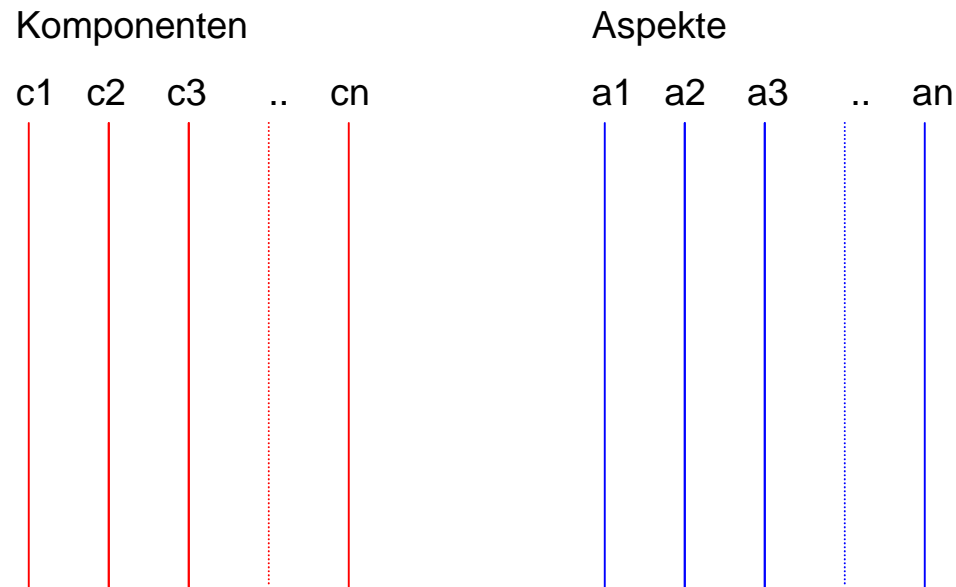
Freie Sicht auf den Code



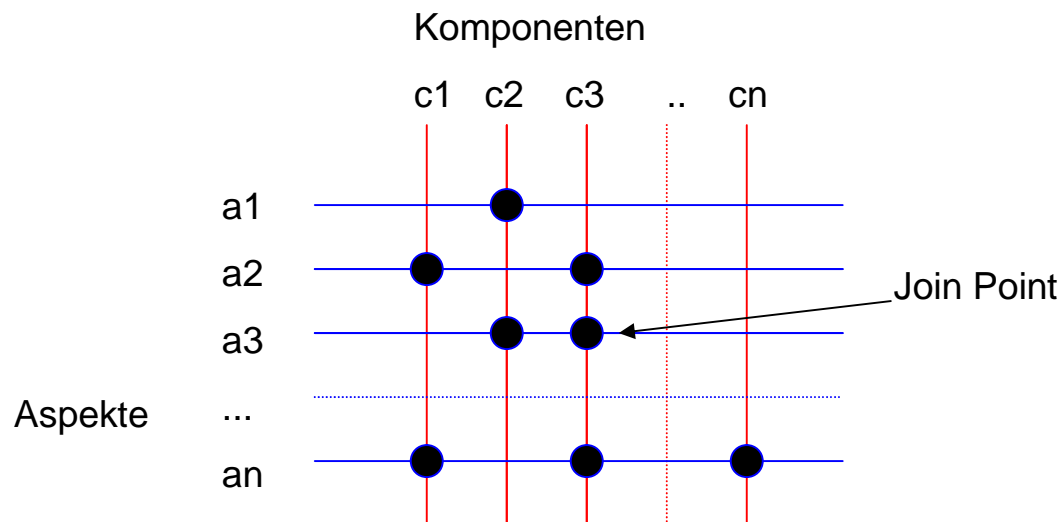
Fehlerbehandlung in Aspekt ausgelagert

```
public aspect A1
{
    pointcut zahlen(int betrag):
    call(void Bankkonto.einzahlen(int))&&args(betrag)
    ||call(void Bankkonto.abheben(int))&&args(betrag);
    void around(int betrag):zahlen(betrag){
        if (betrag<0){
            throw new IllegalArgumentException("negativer Wert");
        }
        proceed(betrag);
    }
}
```





Grundidee der aspektorientierten Programmierung ist es, die Separation of Concerns von Komponenten und „crosscutting“ Aspekten zu meistern



Integration von Komponenten und Aspekten

Kategorisierung von Aspekten:

- Fehlerbehandlungsaspekte
- Geschäftsaspekte
- Loggingaspekte
- Nebenläufigkeitsaspekte
- Optimierungsaspekte
- Programmierrichtlinienaspekte
- Sicherheitsaspekte

Erwarteter Nutzen

- **Modularität auch für übergreifende Dinge („Concerns“)**
 - Java-Code konzentriert sich auf Business-Logik**
 - Kürzerer Code**
 - Einfachere Wartung und Weiterentwicklung**
- **Erhöhte Wiederverwendbarkeit über**
 - Library-Aspekte**

Erfahrungen

- **Deutliche Verbesserung gegenüber der OOP**
- **Codereduktion**
- **Reduzierung der Wartungskosten**
- **Reduzierung der Entwicklungskosten**
- **Verbesserung der Codequalität**
- **Verbesserung der Softwarearchitektur**
- **Verbesserung der Performanz möglich**