Einsatzmöglichkeiten aspektorientierter Programmierung

Dipl.-Inf. (FH) Christian Schommer
Prof. Dr. Jörg Hettel

Fachhochschule Kaiserslautern,
Standort Zweibrücken

Übersicht

- Das aspektorientierte Programmierparadigma
  - Ein pädagogisches Beispiel: *Das SpaceGame*
- Sprachkonzepte der AOP
  - *JoinPoint*, *Pointcut* und *Advices* am Beispiel von AspectJ
  - Realisierung der *SpaceGame*-Anwendung mit AspectJ
- Zwei exemplarische Einsatzbereiche der AOP
  - AOP und Design Patterns
  - AOP und J2EE
- AOP quo vadis?

Das aspektorientierte Paradigma

Objektmodellierung (1)

- Innerhalb der Objektorientierung wird die Welt durch Objekte beschrieben
  - Klassen bilden den Bauplan von Objekten

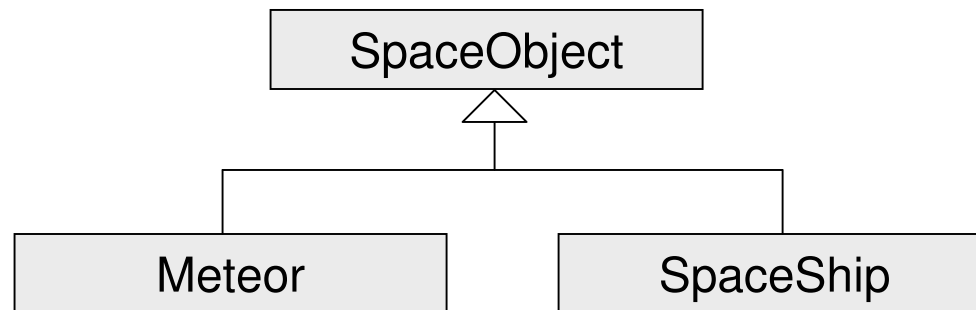
Ein Objekt besitzt einen bestimmten Zustand und reagiert mit einem definierten Verhalten auf seine Umgebung. Außerdem besitzt jedes Objekt eine Identität, die es von allen anderen Objekten unterscheidet. Ein Objekt kann mit anderen Objekten in Beziehung stehen.

- Ein Objekt realisiert das **Geheimnisprinzip**
  - Ein Objekt kapselt Zustand (Daten) und Verhalten (Operationen)
  - Zustand und Verhalten eines Objektes bilden eine Einheit

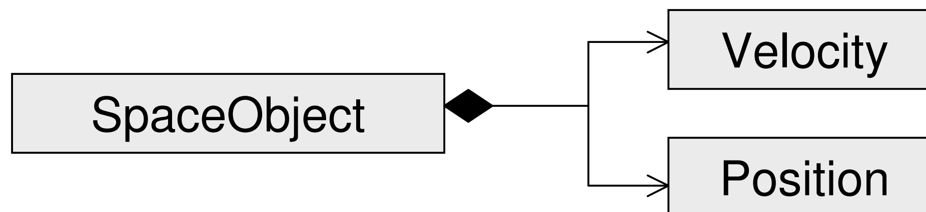
Leitgedanke: Decompose Systems Into Modules

Objektmodellierung (2)

- Strukturierungskonzepte der OOP
  - Die Vererbung realisiert eine *"Typstrukturierung"*

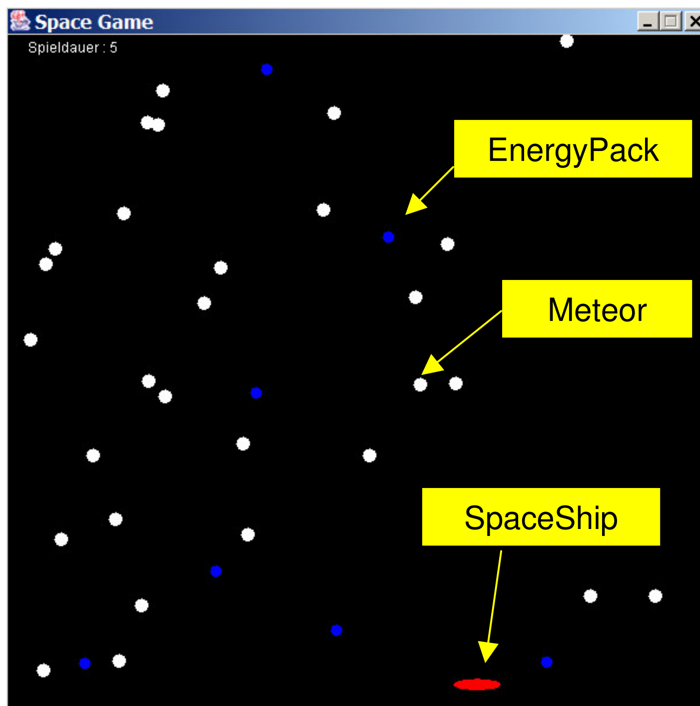


  - Assoziationen realisieren eine *"Beziehungsstrukturierung"*

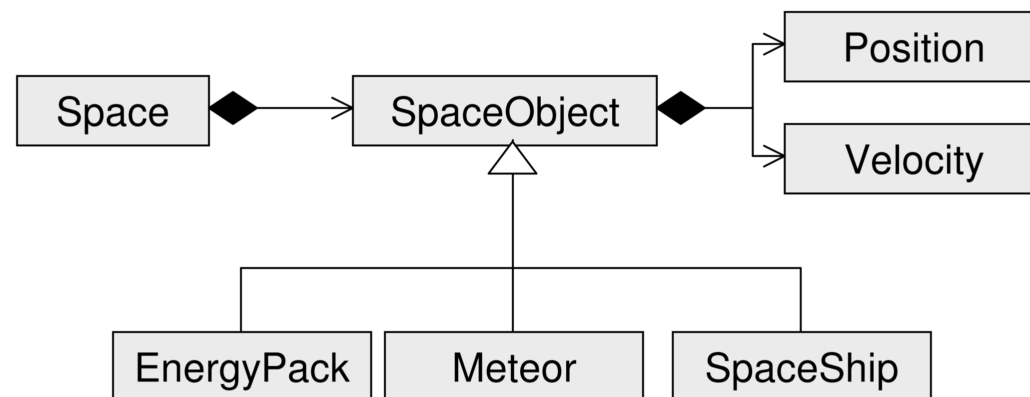


Beispiel: Einteilung der Welt in Klassen (1)

- Ein einfaches Beispiel: *Das SpaceGame*



Klassenmodell:



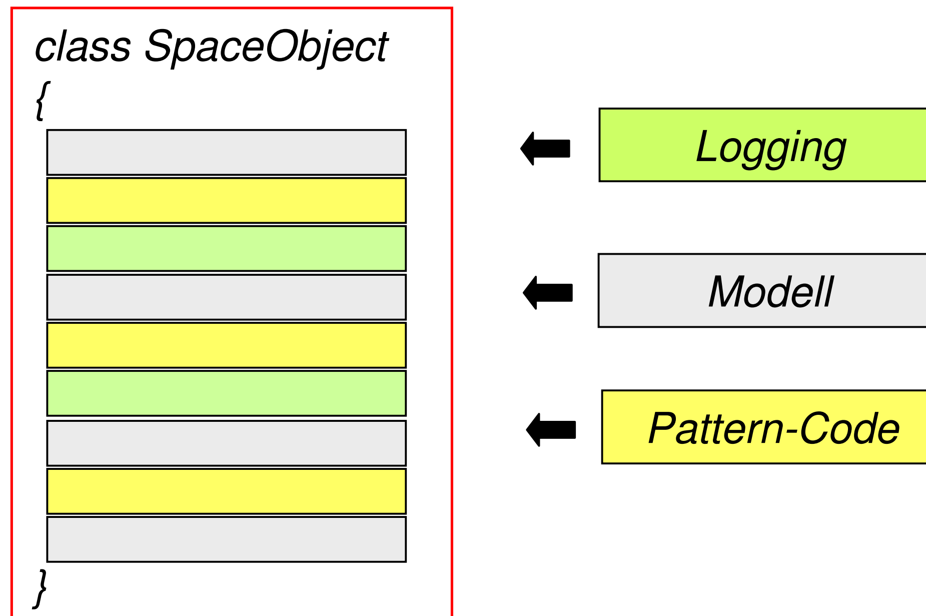
Beispiel: Einteilung der Welt in Klassen (2)

- Das Anwendungsmodell dominiert den Entwurf
  - *"Tyranny of the dominant decomposition"*
  - Modellklassen werden durch weitere Belange (Concerns) verschmutzt!
    - Z.B. Anbindung an View, Logging, etc.

SpaceView
<<Observable>>
Logger
<<Observer>>
Space
SpaceObject
Position
Velocity
SpaceObjectView
EnergyPack
Meteor
SpaceShip
EnergyPackView
MeteorView
SpaceShipView

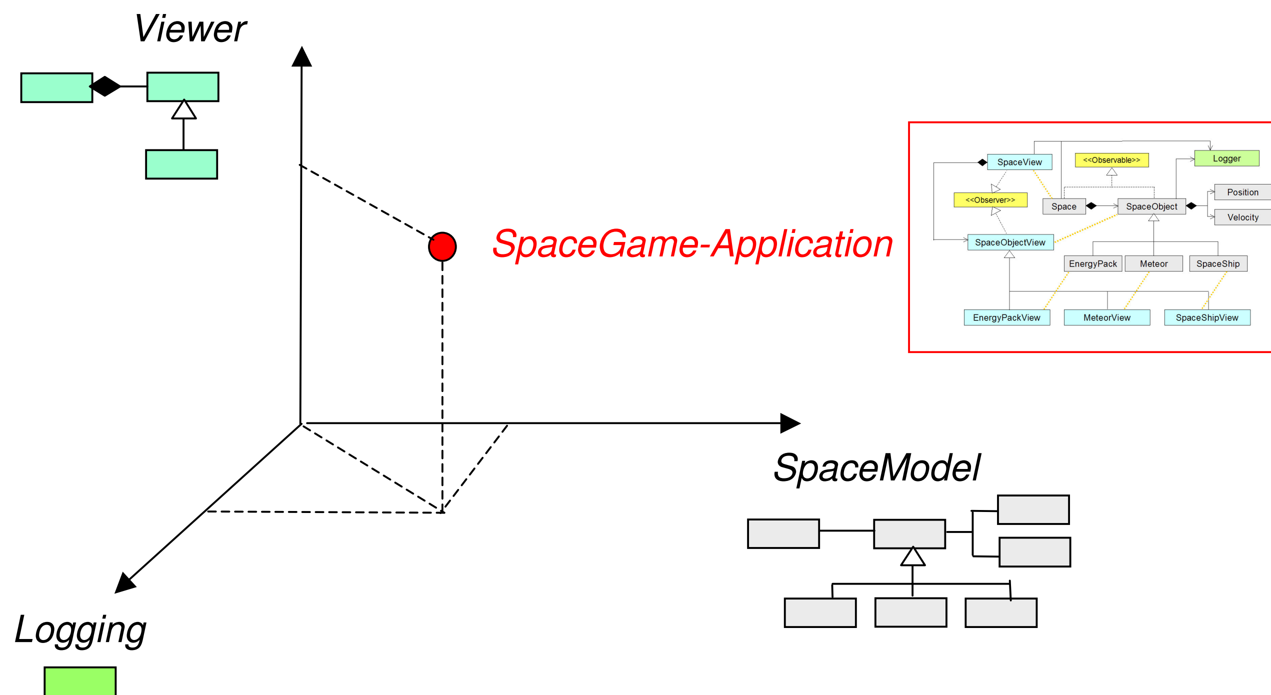
Das Resultat

- Die Modellklassen enthalten modellfremden Code (*crosscutting code*)
  - Zugriff auf das Log-Framework
  - Code zur Realisierung des Observer-Patterns, des Singleton-Patterns, etc.
  - Interface-Implementierungen
- Modellfremder Code verbindet die weiteren Belange mit dem Modell
  - Im *SpaceGame* z.B. View und Logging



Die Idee

- Unabhängige Entwicklung der Concerns als unabhängige Module
  - Entwicklung oder Wiederverwendung eines Modells, eines Viewers und eines Logging-Mechanismus



- Danach Verwebung der Module zu der Anwendung
  - Hierzu werden neue Sprachkonzepte benötigt

Sprachkonzepte der AOP

Sprachkonzepte von AspectJ

- Mit AspectJ lassen sich Code-Fragmente in bestehende Klassen einweben
  - Also kein echtes Zusammensetzen von Klassen aus verschiedenen Concerns

- Um Code einweben zu können brauchen wir folgende Konzepte:
  - Allgemeine Definition der Stellen, an denen Code potentiell eingewoben werden kann
    - Das sogenannte ***JoinPoint* Modell**
  - Definition für die tatsächlich benutzten Stellen, an denen Code eingewoben wird
    - Das sind die sogenannten ***Pointcuts***
    - Bei der Definition der Pointcuts wird das zugrundeliegende *JoinPoint* Modell benutzt
  - Konstrukt zur Gruppierung eines Code-Fragments, dass an einem definierten *Pointcut* eingewoben wird
    - Das sind die sogenannten ***Advices***
  - *Pointcuts* und *Advices* stehen in der Regel in Beziehung. *"Dieser Advice an diesem Pointcut."*
    - *Pointcuts* und *Advices* werden in einem ***Aspect*** zusammengefasst

Weitere Sprachkonzepte von AspectJ

- Klassen können um Attribute und Methoden erweitert werden
  - Man spricht hierbei von ***Introduction***
- Es lassen sich spezifische Compiler-Warnungen und -Fehler definieren
  - Das ist AspectJ spezifisch, da AspectJ intern mit einem Precompiler arbeitet

JoinPoints und Pointcut Syntax von AspectJ

- Auszug aus dem JoinPoint-Modell:

JoinPoint Kategorie	Pointcut Syntax
Class initialisation	staticinitialisation(*TypeSignature*)
Contructor call	call(*ConstructorSignature*)
Constructor execution	execution(*ConstructorSignature*)
Object initialisation	initialization(*ConstructorSignature*)
Object preinitialisation	preinitialization(*ConstructorSignature*)
Field read access	get(*FieldSignature*)
Field write access	set(*FieldSignature*)
Method call	call(*MethodSignature*)
Method execution	execution(*MethodSignature*)
Exception handler execution	handler(*TypeSignature*)
Advice execution	adviceexecution()

Beispiel: Auszug aus der SpaceGame-Anwendung (1)

- Beispiel: Die Meteor-Klasse sind als reine Fachklasse implementiert

```
public class Meteor extends SpaceObject
{
  public Meteor(Position pos, Velocity vel)
  {
    super(pos, vel);
  }

  public double getFieldOfActivity()
  {
    return 10.0;
  }

  public double getMass()
  {
    return 10.0;
  }
}
```

Aspektierung:

Meteore müssen die Rolle von View-Objects annehmen können:

⇒ Implementierung eines entsprechenden Interfaces

⇒ Registrierung des Meteors beim Viewer

Beispiel: Auszug aus der SpaceGame-Anwendung (2)

- Die Modellobjekte müssen auch die Rolle von View-Objekten annehmen können:

```
public class Meteor extends SpaceObject
{
  public Meteor(Position pos, Velocity vel)
  {
    super(pos, vel);
  }
  ...
}
```

Modellklasse: Meteor

Enthält ausschließlich Fachcode.

```
public aspect ViewOfSpaceObjects
{
  declare parents : SpaceObject implements ObjectView;
  public void Meteor.update(Graphics g)
  {
    g.setColor( Color.WHITE );
    g.fillOval( (int) getPosition().getX(),
                              (int) getPosition().getY(), 12, 12 );
  }
  ...
}
```

Introduction:
Die Klasse Meteor erhält eine Interface-Implementierung, damit sie auch die Rolle von View-Objekten annehmen kann.

Beispiel: Auszug aus der SpaceGame-Anwendung (3)

- Verknüpfung der Modellobjekte mit der View

```
abstract public class SpaceObject
{
  ...
  public SpaceObject(Position pos, Velocity vel)
  {
    super();
    this.pos = pos;
    this.vel = vel;
  }
  ...
}
```

Initialization JoinPoint: Nach der Initialisierung kann Code eingewoben werden

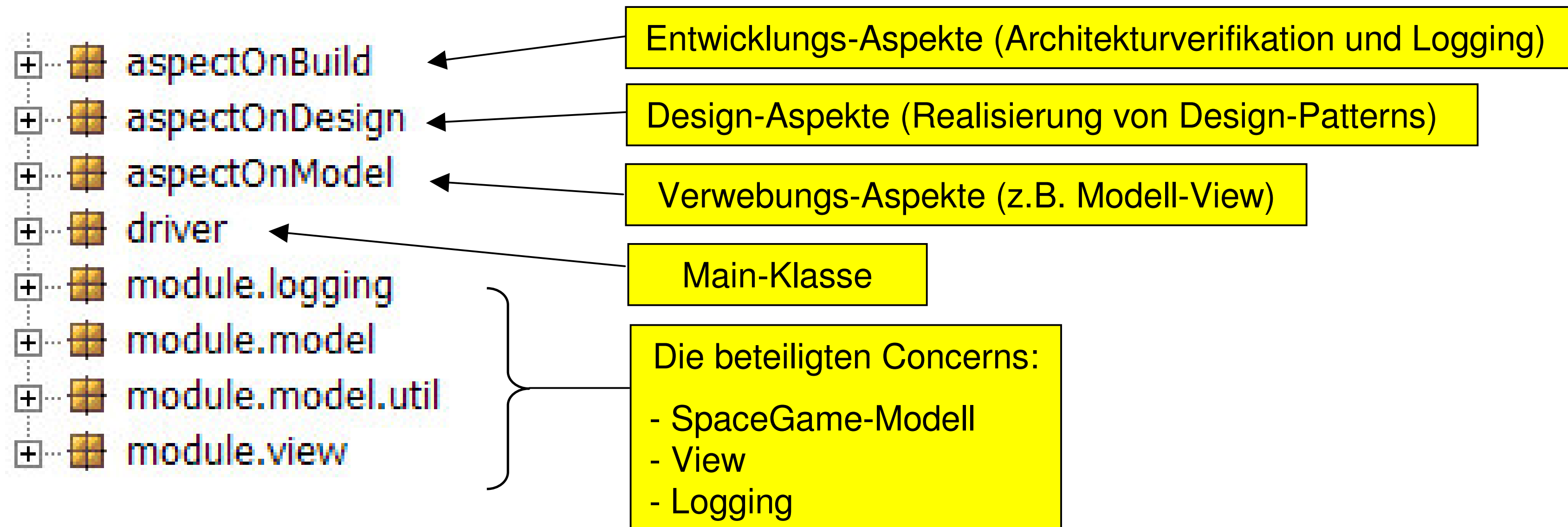
```
public aspect ObjectRegistrationOnView
{
  pointcut spaceObjectContructor(ObjectView obj) :
      initialization( public SpaceObject.new(..) ) && this(obj);

  after(ObjectView obj) returning : spaceObjectContructor(obj)
  {
    View view = new View();
    view.addObjectView( obj );
  }
}
```

Nach der Erzeugung eines Space-Objekts, wird es an der View registriert.

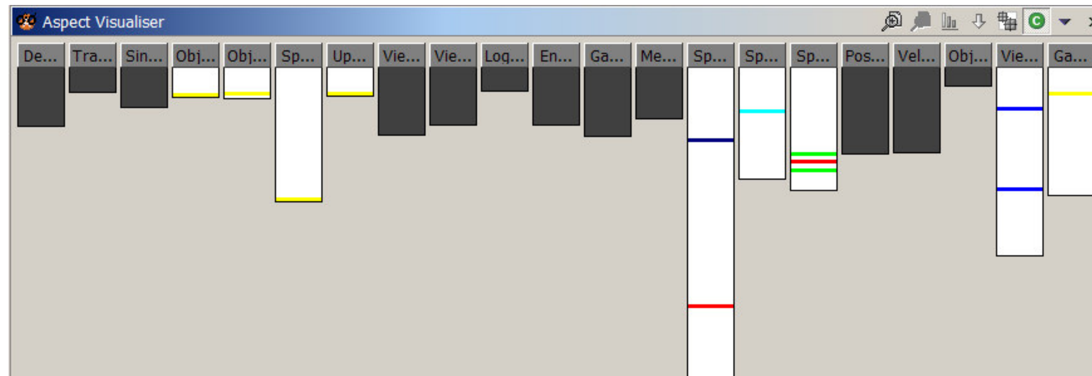
Projektübersicht in Eclipse

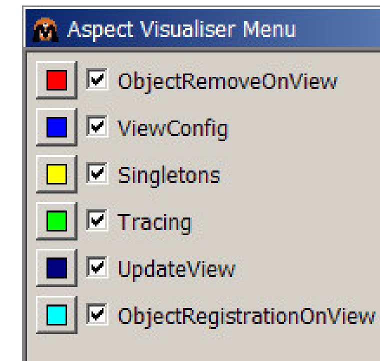
- Die Anwendung wird in einzelne Concerns unterteilt
- Die Aspekte kommen in eigene Packages



AspectJ-Unterstützung in Eclipse

- Graphische Veranschaulichung der Webstellen bei Eclipse:





Beispiel: Architekturverifikation mit AspectJ

- Auf Basis des JoinPoint-Modells können statische Architekturverifikationen durchgeführt werden.
  - Z.B. Überprüfung der Unabhängigkeit der Concerns

```
public aspect DependencyCheck
{
  // Abhängigkeiten des Modells von der View
  public pointcut viewCalls() : call(* concern.view..*(..) )
                               || call( concern.view..*.new(..) );

  public pointcut modelCallsFromView() : viewCalls()
                                        && within( concern.model..* );

  declare warning : modelCallsFromView() : "Modul dependency error!";

  ...
}
```

Einsatzbereiche der AOP

Einsatzbereiche von AOP

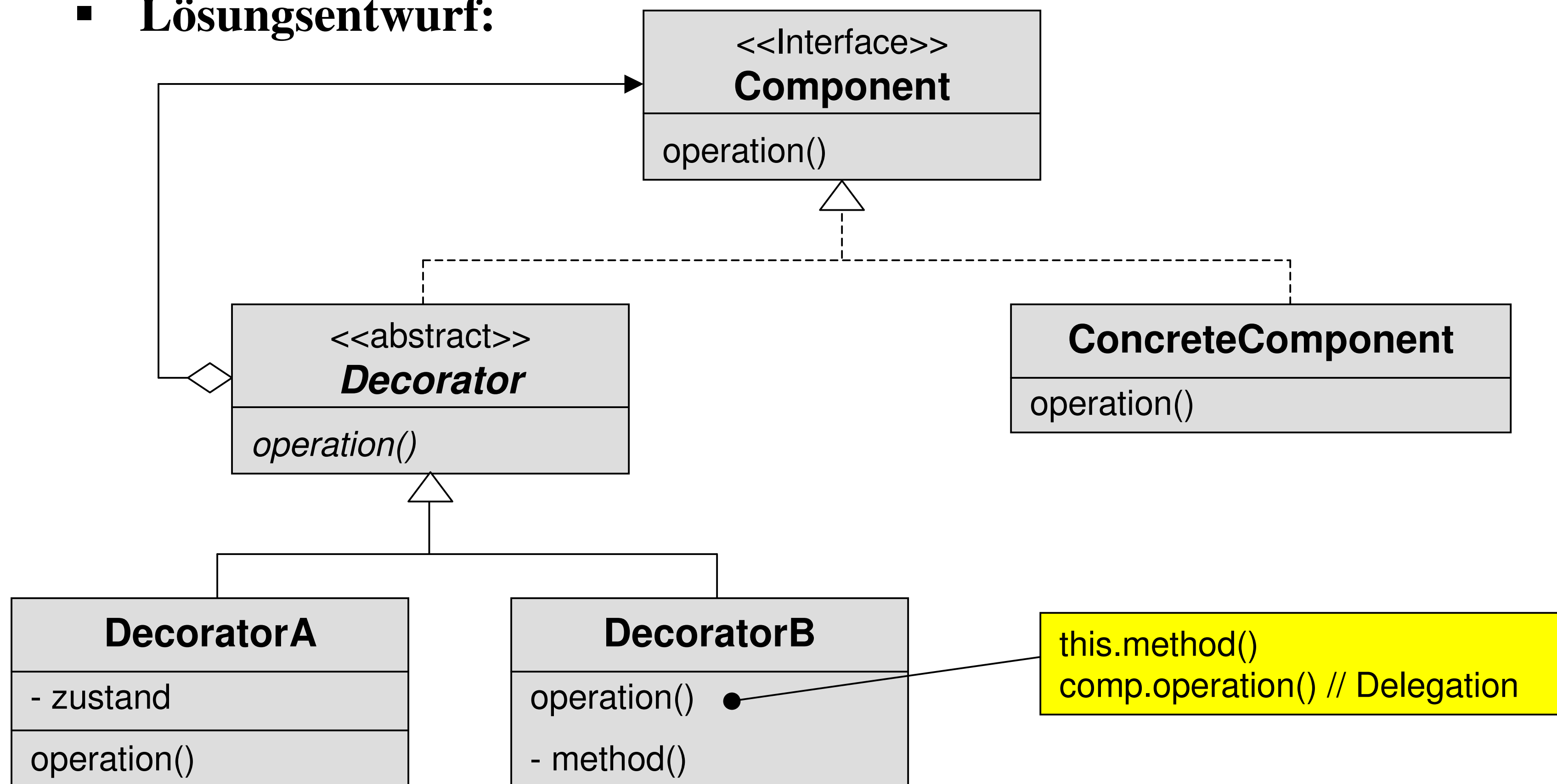
AOP ist mehr als Logging!!

- Einsatz von AOP bei der Softwareentwicklung:
  - Entwicklungsaspekte
    - Tracing und Logging, Profiling, Programm-Slicing, Architektur-Verifikationen (Abhängigeitschecks, Law-Of-Demeter-Check, etc.), etc.
  - Design-Aspekte
    - Pattern-Realisierung, Pooling und Caching
  - Anwendungsaspekte
    - Persistenz, Transaktionen, Authentifizierung und Autorisierung, Verteilung, etc
      - Hauptsächlich Middleware-Thematiken

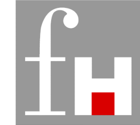
- Ausblick auf weitere Anwendungsgebiete:
  - Der Einsatz von Aspekten bei der Modellierung und Implementierung von Use Cases wurde von Jacobson vorgeschlagen
    - Siehe Literatur

Das Decorator Design Pattern

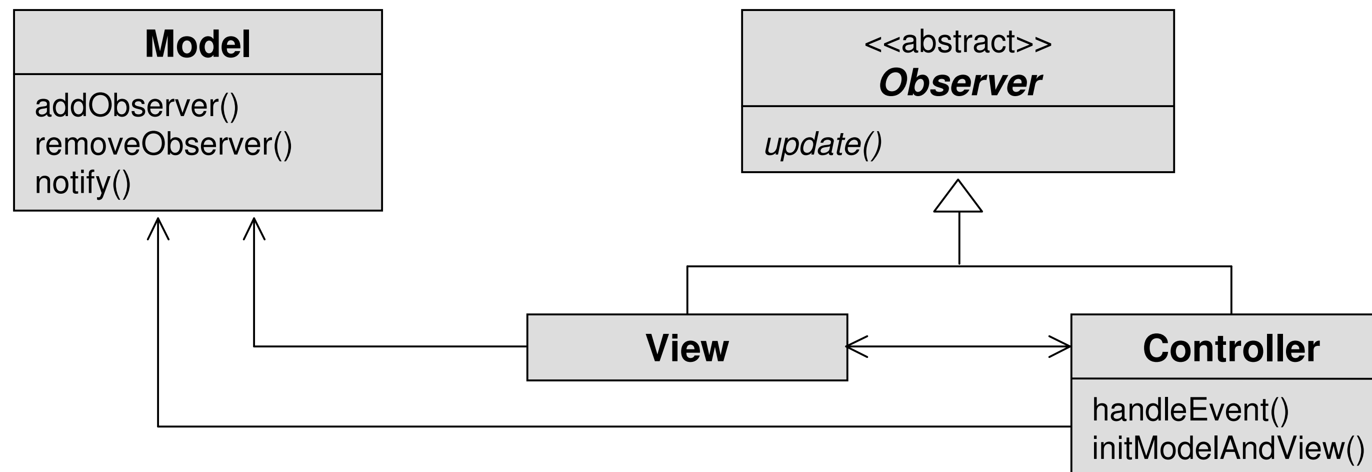
- **Problem:** Einem Objekt müssen neue Zuständigkeiten zugeordnet werden. Die Klasse darf dabei nicht geändert werden.
- **Lösung:** Dynamisches "Vorschalten" von Objekten mit dem gleichen Typ wie die "dekorierte" Klasse.
- **Lösungsentwurf:**



Model-View-Controller Pattern



- **Problem:** Benutzungsschnittstellen müssen häufig geändert oder ausgetauscht werden. Die Änderungen bzw. der Austausch soll ohne großen Aufwand vollzogen werden können.
- **Lösung:** Unterteilung der Anwendung in drei Bereiche: *Verarbeitung (Modell), Ausgabe (View)* und *Eingabe (Controller)*.
- **Lösungsentwurf:**

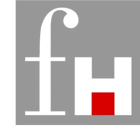


Bemerkungen

- Limitierungen des Decorator-Patterns:
  - Der Einsatz des Decorator-Patterns muss im Entwurf berücksichtigt werden
    - Man benötigt ein Interface bzw. eine abstrakte Klasse, von der die Decorator-Klassen abgeleitet werden können
  - Der Aufrufer arbeitet nicht mit dem Originalobjekt, sondern mit einem Decorator-Objekt.
    - Decorator-Objekt hat eine andere ID als das Originalobjekt
  - Bei der Verwendung des Decorator-Patterns werden viele Objekte erzeugt

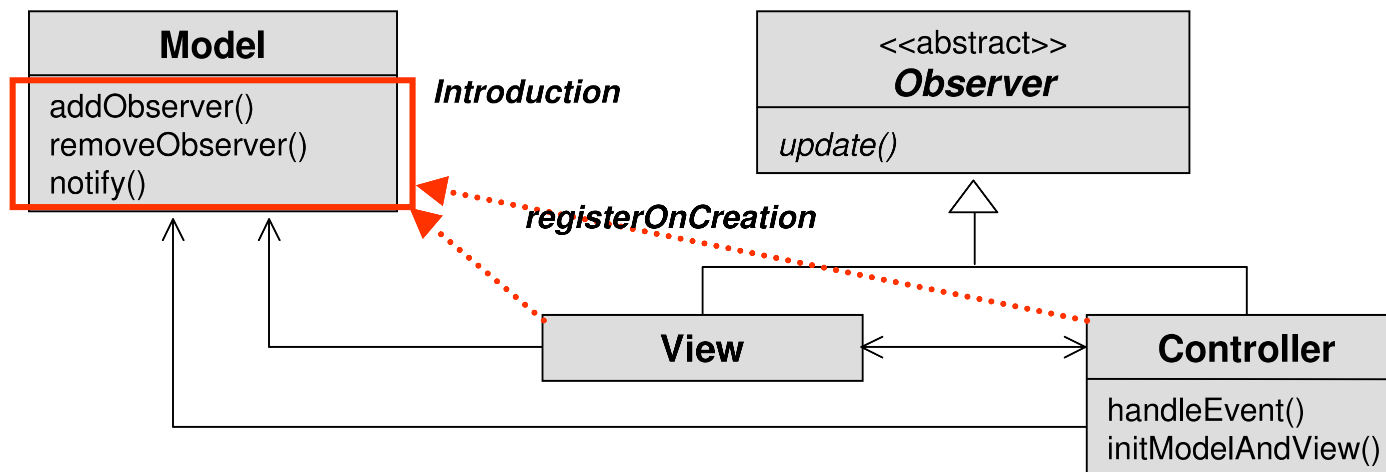
- Limitierungen des MVC-Patterns
  - Bei der Verwendung des Model-View-Controller-Patterns muss das Modell um pattern-spezifische Funktionalität erweitert werden
    - Änderungen am Modell müssen zu expliziten notify-Aufrufen führen.
    - Observable benötigen eine Observer -Verwaltung.
    - Das Pattern ist direkt im Modell-Code "verdrahtet".

AOP-Varianten der beiden Patterns



- Das Decorator-Pattern kann sehr einfach durch AOP-spezifische Sprachkonstrukte realisiert werden
  - Einweben des Decorator-Codes in das "Original-Objekt"

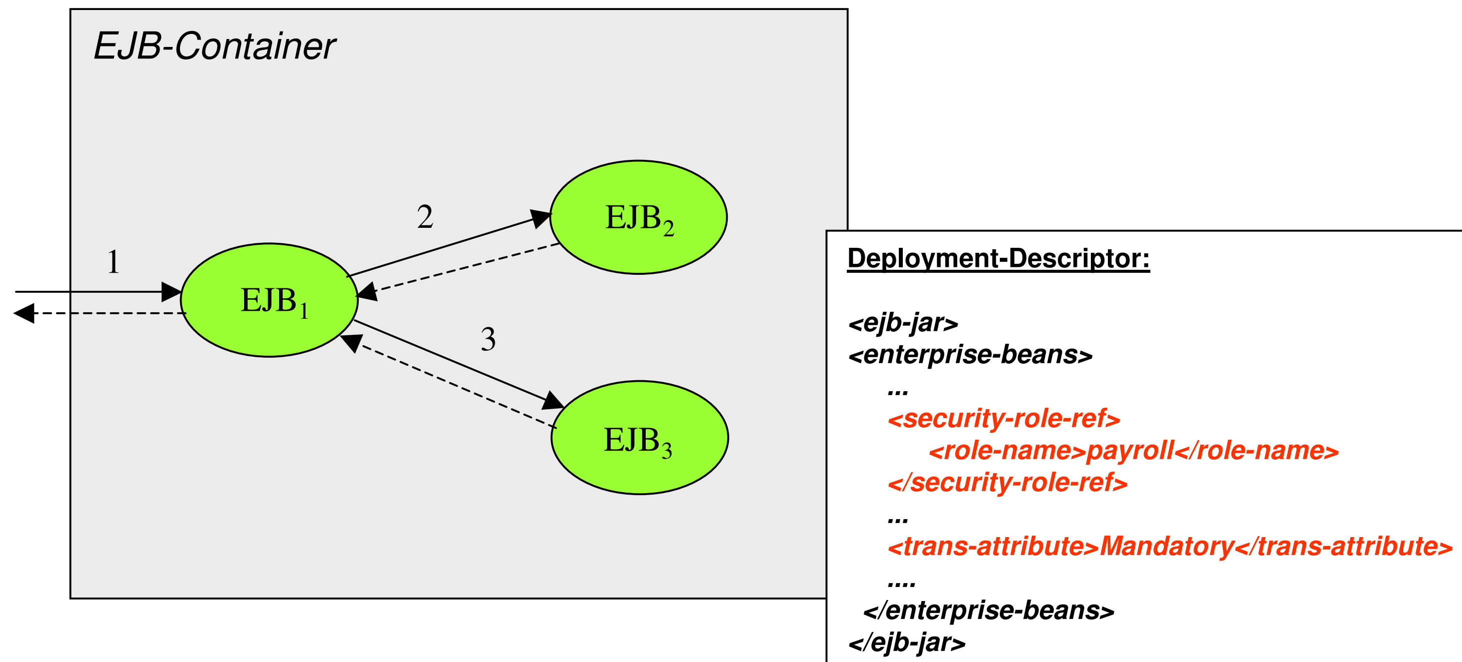
- Bei der Realisierung des MVC-Pattern kann der Pattern-spezifische Code eingewoben werden
  - Das Modell bleibt frei von Pattern-Code



© Prof. Dr. Jörg Hettel, Fachhochschule Kaiserslautern

AOP und J2EE (1)

- Bei EJBs existiert im Prinzip schon eine Art *Separation of Concerns*
- Transaktionales Verhalten und Zugriffskontrolle werden im Deployment-Descriptor hinterlegt
  - Kein Code bezüglich Transaktionen und Berechtigungsprüfung (Security) in der Klasse
    - Das ist die Design-Empfehlung



AOP und J2EE (2)

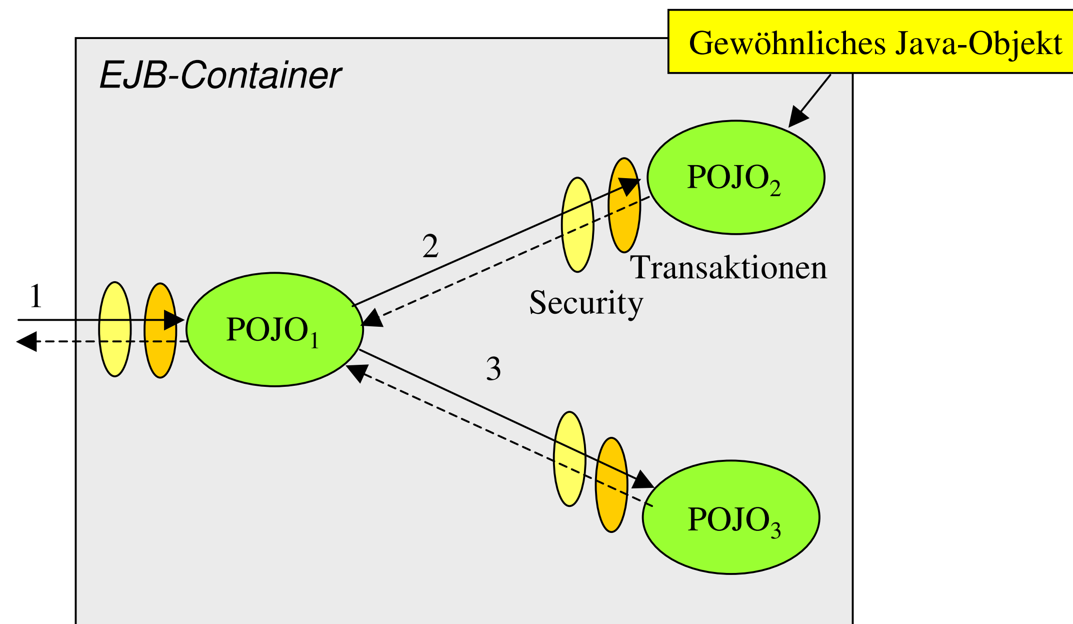
- Limitierungen des J2EE-Ansatzes
  - Separation of Concerns kann durchbrochen werden
    - Es existiert ein API für den Zugriff auf den *Transaktions*- oder *Security-Aspekt*
      - Bsp:

```
public class MySessionBean implements SessionBean {
  public void komplexeOperation() throws ... {
      SessionContext ctx = getSessionContext();
      if( ctx.getRollbackOnly() ){
          return;
      }
      // führe komplexe Operation aus
  }
}
```

  - Nur explizit als EJBs implementierte Klassen können an den "Aspekten" teilnehmen
    - Ausschluss von gewöhnlichen Java-Objekten (POJO)
      - POJO = ***p**lain **o**ld **J**ava **o**bjects*

AOP und J2EE (3)

- **Idee:** Realisierung der Middleware-Concerns als Aspekte
  - Somit kann jedes Java-Objekt *"aspektiert"* werden
- Z.B. JBoss-AOP-Implementierung
  - Realisierung der Aspekte durch *"Interceptors"*
  - *Interceptor*-Objekte werden an vorher definierten *Pointcuts* zur Laufzeit dynamisch "eingewoben"
    - Basiert auf JBoss-AOP-Framework



AOP quo vadis?

Was AOP leistet und was nicht

- AOP bietet neue Strukturierungsmöglichkeiten
- AOP bietet die Möglichkeit Klassen auf einem höheren Abstraktionslevel zusammenzusetzen (verweben)
  - Insbesondere können Crosscutting-Concerns sehr elegant implementiert werden
- AOP ist kein Ersatz für OOP, sondern eine zusätzliche Technik
  - Funktionaler Code muss in einer herkömmlichen Programmiersprache geschrieben werden.
  - AOP bietet nur die Möglichkeit Code (aus Fragmenten) zusammenzusetzen

- AOP führt bei falschem Einsatz genauso zu schlechtem Design und Code, wie das auch bei der OOP der Fall sein kann
  - Man muss die Konzepte und Design-Kriterien richtig anwenden
- AOP verkompliziert den Programmfluss und erschwert somit das Debugging und die Fehlererkennung
  - Das ist auch schon bei OOP der Fall

AOP quo vadis?

- Die AOP-Sprachen bzw. –Frameworks sind, Stand Juli 2004, nur bedingt im großen Stil einsatzfähig.
- Es fehlt noch:
  - Standardisierte Modellierung von Aspekten
  - "Vernünftige" Entwicklungs- und Toolunterstützung
  - Konsolidierung am Markt
    - Es gibt eine große Anzahl von AOP-Frameworks
      - Viele Frameworks sind nur Forschungsprototypen

- AOP hat momentanen Entwicklungstand, wie die OOP vor ca. 10 Jahren
- AOP hat sehr gute Chancen sich als neues Paradigma durchzusetzen
  - Siehe z.B. AOP-Commitment der großen Hersteller
    - Z.B.: IBM, BEA, JBoss, etc.

- AOP ist eine Technologie, bei der es sich lohnt, näher hinzuschauen
  - Sie verändert die Sichtweise auf die Software-Entwicklung

AOP als neues Modularisierungsprinzip

- Der Weg zur modularisierten Software über neue Abstraktionsstufen

Literatur

1. Ramnivas Laddad: *AspectJ in Action*, Manning 2003
2. Gregor Kiczales et al.: *Aspect Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming, p. 220-242, 1997
3. Jan Hannemann und Gregor Kiczales: *Design Pattern Implementation in Java and AspectJ*, Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, p. 161 – 173, 2002.
4. Gamma et al.: *Design Patterns*, Addision Wesley, 1995.
5. Ivar Jacobson: *Use Cases and Aspects - Working Seamlessly Together*, Journal of Object Technology, Vol. 2, No. 4, July-August 2003, pp. 7-28.
6. Karsten Doller: *Einsatz aspektorientierter Programmierparadigmen bei der Entwicklung von J2EE-Anwendungen*, Diplomarbeit, FH-Kaiserslautern, 2004