Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

18.12.2018
Programmieren im Grossen IV

Muster

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

AGENDA

Einführung ins Thema

Was sind Muster?

Entwurfsmuster

(Besucher, Beobachter, Liste gängiger Muster)

Architekturmuster

(3-Schichten, MVC, Liste gängige Muster)

Allgemeine Prinzipien

Fazit

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

01
EINFÜHRUNG INS THEMA

Ziel:
Die Eckpunkte des Themas kennenlernen

WIEDERKEHRENDE PROBLEME

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Oft gibt es immer wiederkehrende Probleme

Aus dem Beispiel der letzten Vorlesung:

- Baum-Struktur mit verschiedenartigen Knoten
- Navigation, zumindest: von oben nach unten
- Verwaltung, zumindest: Knoten einfügen, löschen
- Delegieren von Aufgaben nach unten

→ Solche oder sehr ähnliche Aufgaben werden in vielen Situationen gebraucht - Z.B.:

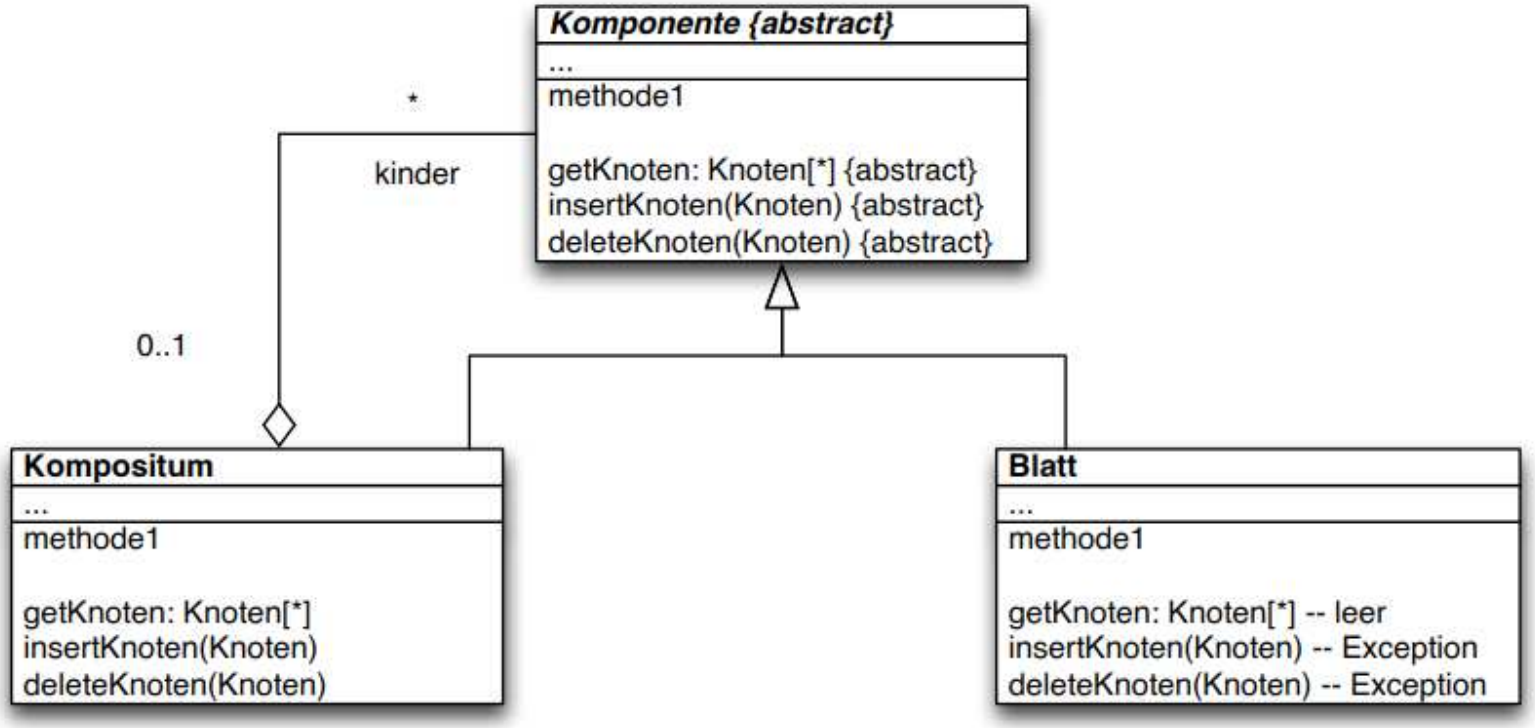
  - Dateisystem,
  - Evtl.: Unsere Programmieraufgabe im 2. Semester

→ Hier helfen Muster!

ZUGEHÖRIGE, TYPISCHE, ERPROBTE LÖSUNG:

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Wir setzen das im allgemeinen Fall so um:



Kompositum.methode1: delegiert Aufgabe an Unterknoten

→ Diese immer wiederkehrende Aufgabenstellung + diese typische, erprobte Lösung (+ zusätzliche Beschreibungen)

wird *Entwurfsmuster „Kompositum“* genannt

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

02
Was sind Muster?

Ziel:
Nochmals den Mustergedanken genauer beleuchten

DEFINITION MUSTER (PATTERNS)

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Muster (in der Software-Entwicklung)
  = typische, erprobte Lösung für ein immer wiederkehrendes Problem in einem bestimmten Kontext

- Konzept wurde von Christopher Alexander (Bauarchitektur) entwickelt
  - Erlaubt Kommunikation von Expertenwissen
    → bietet bewährte Lösung gerade auch für Einsteiger
    → Vereinfacht die Kommunikation zwischen Experten

DEFINITION MUSTER (PATTERNS)

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Grundideen:

- Problem bestehend aus Konflikten oder Kräften
- Musterinvarianten Teil → Kern des Musters
- Mustervarianter Teil → Veränderlicher Teil zur Einbettung in Umgebung des Musters
- Konsequenzen → Neue Kräfte
- Muster kommunizieren miteinander über Konsequenzen/Kräfte:

  Unterstützend ↔ Konflikt verursachend

→ Diese Punkte bilden auch Grundstruktur

→ Oft in Musterschablone gepackt

MUSTER-BESCHREIBUNGEN

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Übliche Form für die Beschreibung eines Musters:
  1. Name + andere Namen
  2. Beispiel
  3. Kontext
  4. Problem
  5. Lösung
  6. Vor- und Nachteile
  7. Verwandte Muster / Kommunikation zw. Mustern
- Musterkataloge:
  - Sammlung von Mustern + deren Zusammenwirken
    (Unterstützend, Widersprechend, Neutral)
  - z.B. Entwurfsmuster von Gamma et al (s. Literaturverzeichnis)

ARTEN VON MUSTERN

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Anforderungsmuster (requirement patterns)
  = adressieren Probleme bei der Spezifikation von Anforderungen
- Analysemuster (analysis pattern)
  = adressieren Probleme beim Erstellen von Fachmodellen, etc.
- Architekturmuster (architectural patten)
  = adressieren Probleme beim Grobentwurf
- Entwurfsmuster (design pattern)
  = adressieren Probleme beim Feinentwurf

ARTEN VON MUSTERN

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Idiome
  = beschreiben gewisse typische „Redewendungen“ in Programmier-sprachen, die zu Lösung spezifischer Probleme verwendet werden
- Prozessmuster (Best Practices)
  = Adressieren Probleme mit Softwareentwicklungsprozessen
  – Auch Best Practices genannt → Agile Methoden bauen darauf
  – BEM: Ward Cunningham & Alistair Cockburn haben Mustergedanken in das SE getragen
- Allgemeine Prinzipien / Heuristiken
  – Sog. Daumenregeln
  – Z.B. GRASP (siehe später)
- Antipatterns
  – Beschreiben schlechte/ungeeignete Lösungen

LEIDER KÖNNEN WIR NICHT ALLE MUSTERARTEN BEHANDELN

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Deshalb besprechen wir:
  - Entwurfsmuster (Designpatterns)
    - Besucher
    - Beobachter
    - (Method-Object-Pattern)
    - Liste gängiger Muster
  - Architekturmuster (Architectural Patterns)
    - 3-Schichten-Architektur
    - MVC-Muster
    - Liste gängiger Architekturmuster
  - Allgemeine Prinzipien
    - GRASP-Patterns

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

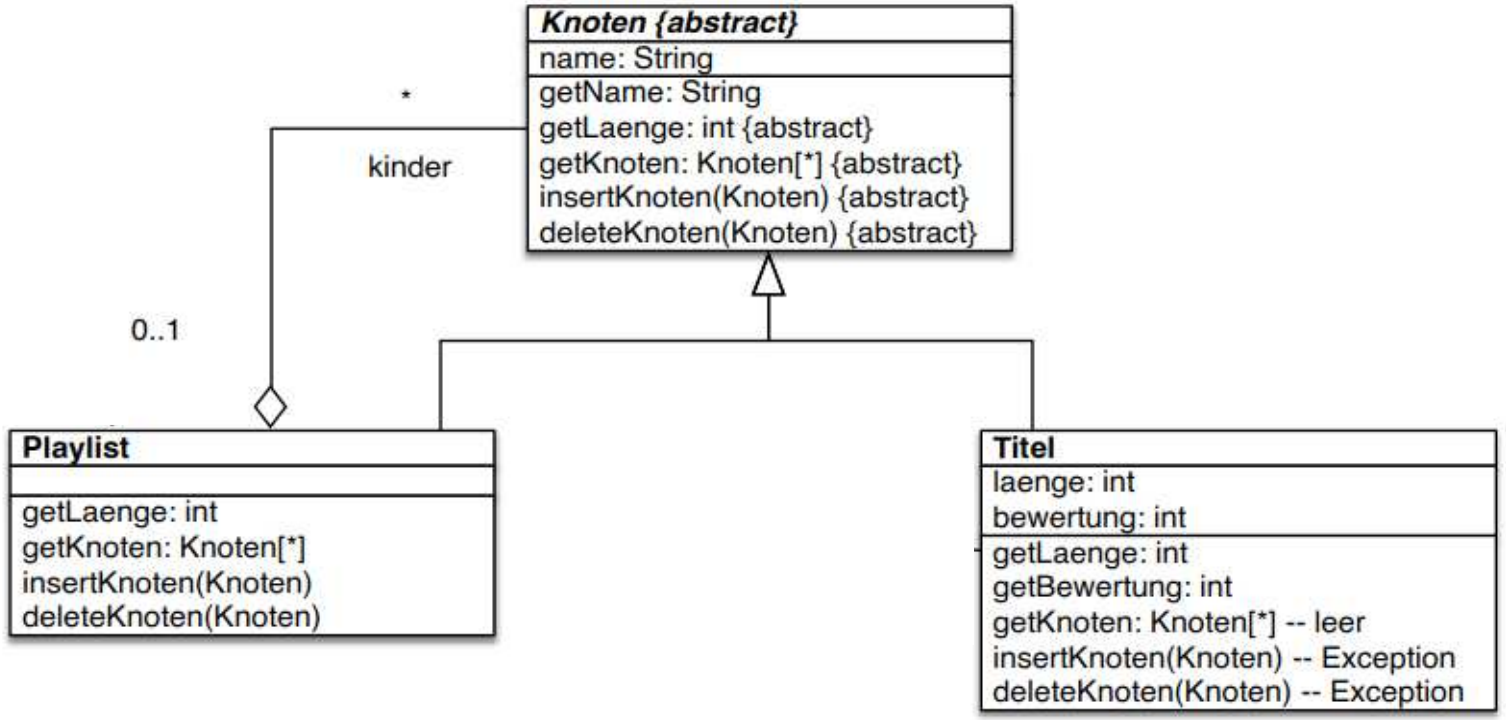
03a
Entwurfsmuster – Besucher (Visitor)

Ziel:
Entwurfsmuster Besucher genauer kennenlernen

BEISPIEL FEINENTWURF „PLAYLISTENVERWALTUNG“

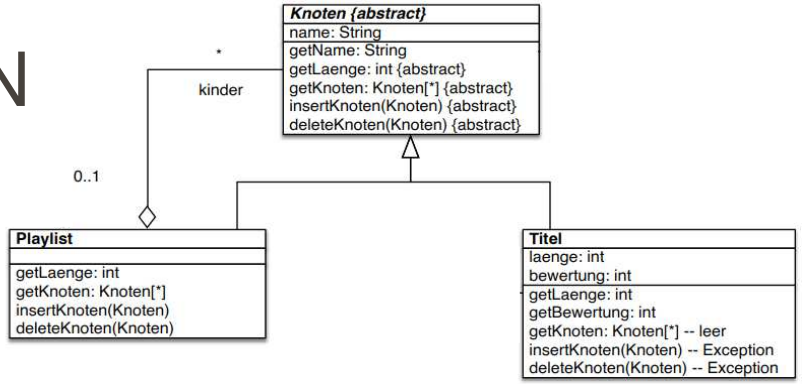
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Unser Beispiel:



Jetzt: Erweiterung um die Berechnung der Bewertung für Playlist-Objekte

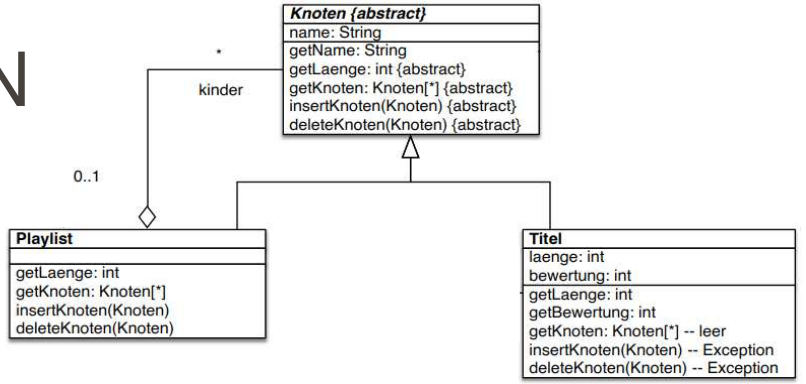
AUSGANGSÜBERLEGUNGEN FÜR BESUCHER-MUSTER



- Mögliche Umsetzung:
  - neue Methode *Playlist.getBewertung*: gibt Durchschnitt von getBewertung der untergeordneten Knoten zurück
  - neue abstrakte Methode *Knoten.getBewertung*

- Mögliche Nachteile:
  - Wenn mehrere solcher Methoden hinzugefügt werden sollen und wenn es viele Unterklassen von Knoten gibt, dann:
    - Schnittstelle von Knoten wird aufgebläht
    - (fast) alle Klassen in der Vererbungshierarchie müssen jedes Mal angefasst werden

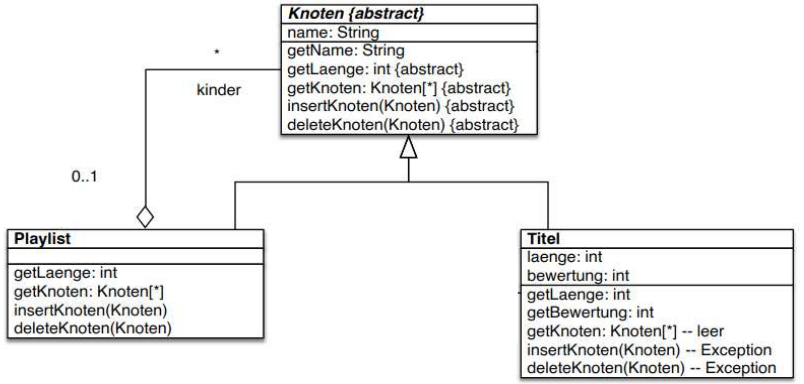
AUSGANGSÜBERLEGUNGEN FÜR BESUCHER-MUSTER



- Bessere Idee:
  - Kapseln der neuen Funktionalität „Berechnung der Bewertung“ in eigener Klasse.

- Konsequenzen der besseren Idee:
  - Wenn mehrere solcher Methoden hinzugefügt werden sollen und wenn es viele Unterklassen von *Knoten* gibt, dann:
    - je neuer Funktionalität eine neue Klasse
    - Schnittstelle von *Knoten* verändert sich nicht
    - keine bestehende Klasse in der Vererbungshierarchie muss angefasst werden

UMSETZUNG: 1. VERSUCH

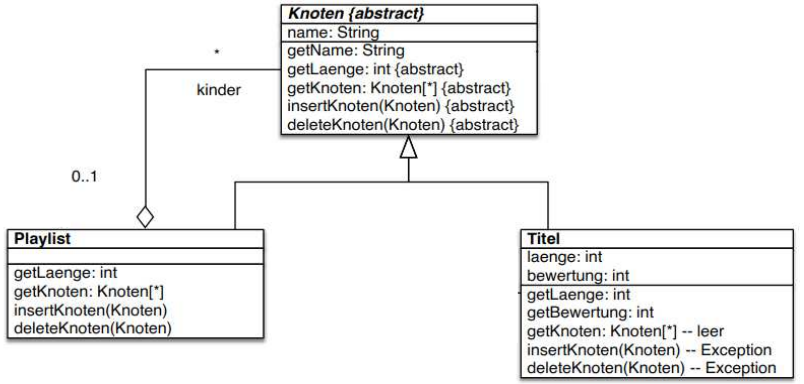


- Eigene Klasse zur Berechnung der Bewertung (fehlerhaft):

```
public class Bewertung {
  public static double getBewertung(Titel t) {
      return t.getBewertung();
  }

  public static double getBewertung(Playlist p) {
      ...
      for (Knoten k: p.getKnoten()) {
          summe += getBewertung(k);
      ...
  }
```

Problem:
Lässt sich nicht kompilieren, da es keine passende Methode *getBewertung(Knoten k)* gibt!

UMSETZUNG: 2. VERSUCH

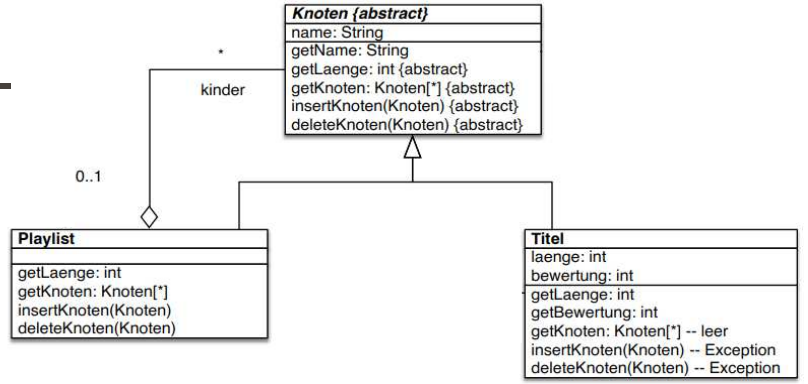


- Eigene Klasse zur Berechnung der Bewertung (fehlerhaft):

```
public class Bewertung {
  public static double getBewertung(Titel t) {...}
  public static double getBewertung(Playlist p) {
      // wie getBewertung(Knoten k)
  }
  public static double getBewertung(Knoten k) {
      ...
      for (Knoten kind: k.getKnoten()) {
          summe += getBewertung(kind);
      ...
```

Problem:
getBewertung(Titel) und *getBewertung(Playlist)* werden niemals aufgerufen, sondern stets *getBewertung(Knoten)*.
→ Polymorphieproblem

UMSETZUNG: 2. VERSUCH – POLYMORPHIEPROBLEM



- Eigentliches Problem:

```
public class A {...}
public class B extends A {...}

public class AndereKlasse{
   public ... f(A a) {...}
   public ... f(B b) {...}
   ...
       A a1 = new A(...);
       ... = f(a1);
       A a2 = new B(...);
       ... = f(a2);
```

f (A)

f (A)

Warum greift der Polymorphismus hier nicht?
Bereits zur Compile-Zeit wird anhand der Parameter entschieden, ob *f(A)* oder *f(B)* in den Bytecode eingesetzt wird.

BEMERKUNG ZU POLYMORPHIE IN JAVA UND C++

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- In Java und C++ funktioniert der Polymorphismus nur für Methoden in derselben Klasse / Klassenhierarchie

```
public class A { ... f(...) {...} }
public class B extends A { ... f(...) {...} }
...
  A a1 = new A(...);
  a1.f(...);
  A a2 = new B(...);
  a2.f(...);
```

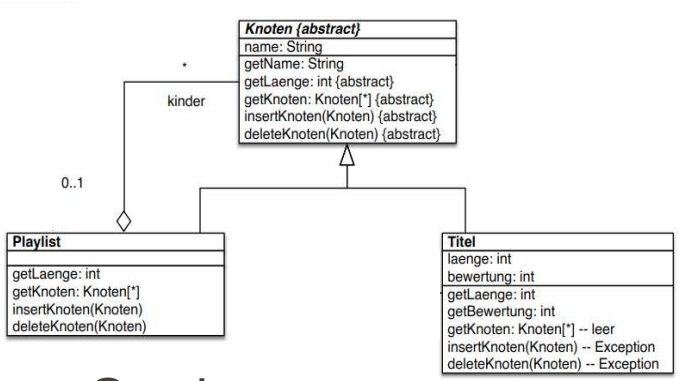
A.f (…)

B.f (…), weil f in A und B definiert

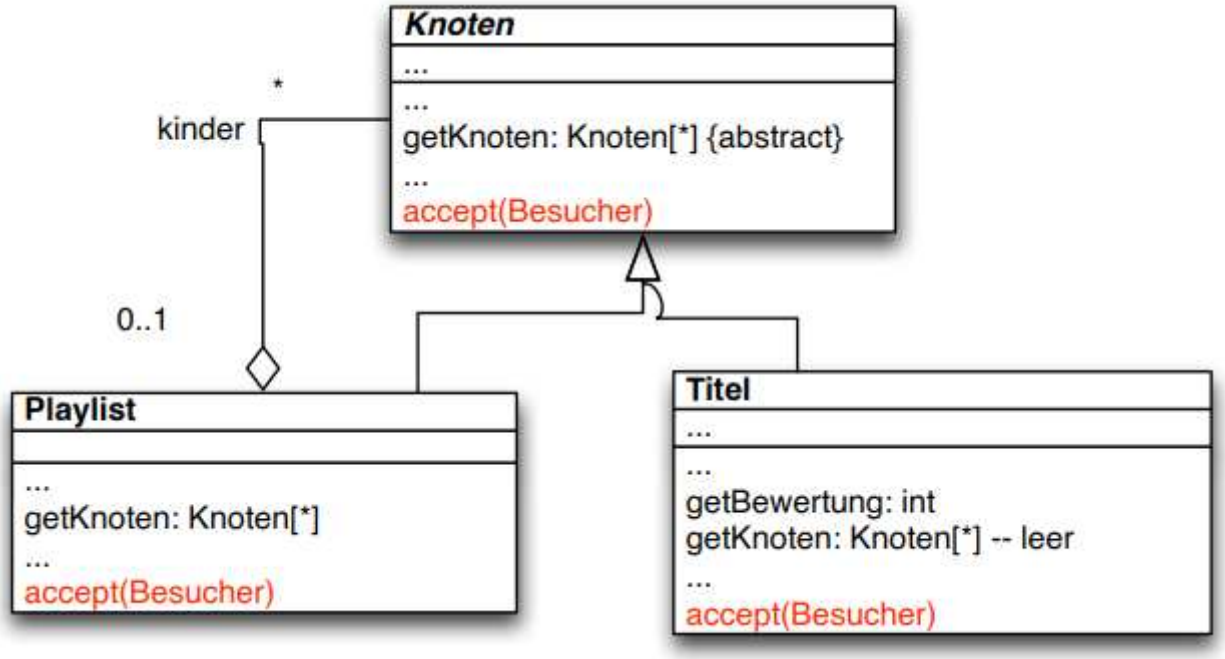
- Funktioniert jedoch nicht wenn die Methode in einer anderen Klasse/ Klassenhierarchie definiert ist (siehe Bsp. vorher)

- **BEM:** Es gibt Programmiersprachen, in denen der Polymorphismus auch für Parameter funktioniert („Multimethoden“)
  - Z.B.: Scala, Groovy (beide Java-basiert und sehr interessant!), Haskell, …

UMSETZUNG: 3. VERSUCH – DIESES MAL KLAPPT‘S!



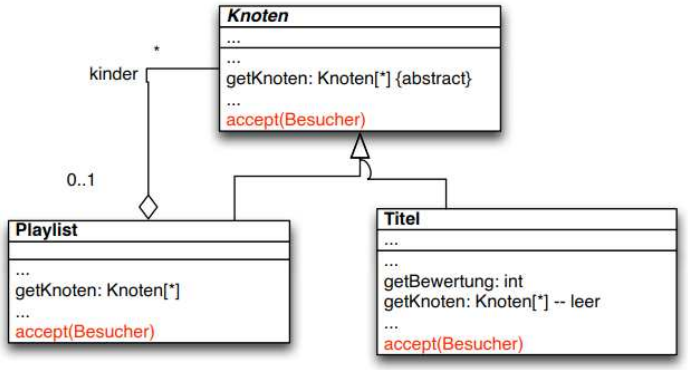
- Entwurfsmuster „Besucher“ – Teil 1:
  - *accept*-Methode für alle Klassen der Baum-Struktur:



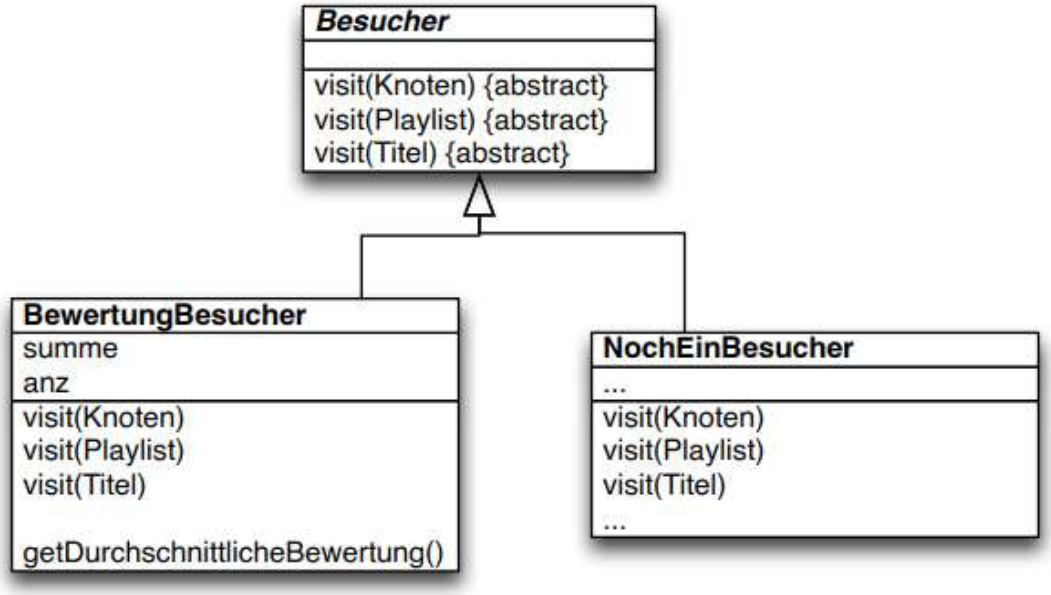
  - Gleiche Implementierung von *accept(Besucher b)* für alle Klassen: *b.visit(this)*

⚠ Vorsicht: Diese eine Code-Zeile muss in der accept-Methode **jeder** Unterklasse stehen (wegen Polym.-Problem)!

UMSETZUNG: 3. VERSUCH – DIESES MAL KLAPPT‘S!

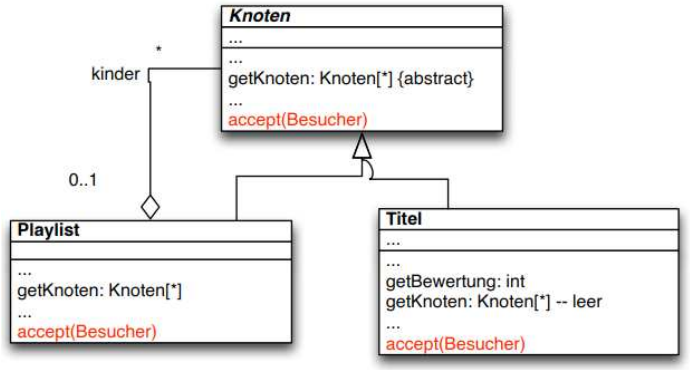


- Entwurfsmuster „Besucher“ – Teil 2:
  - Gemeinsame Schnittstelle/Oberklasse *Besucher* aller Klassen, die Zusatzberechnungen auf Baum-Struktur durchführen:



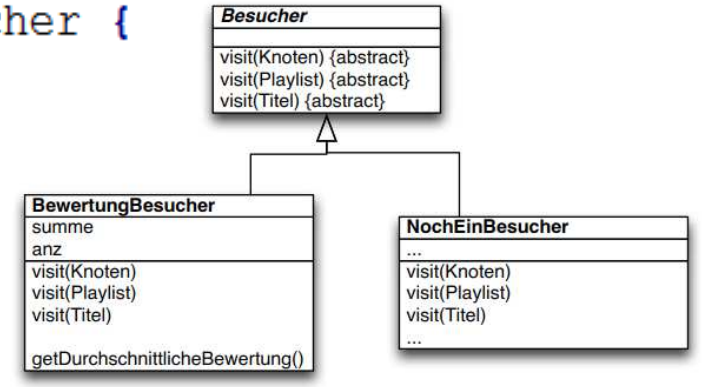
  - und Aufruf von *visit* für Unterknoten **nur** indirekt über Aufruf von *accept*!

UMSETZUNG: 3. VERSUCH – DIESES MAL KLAPPT‘S!

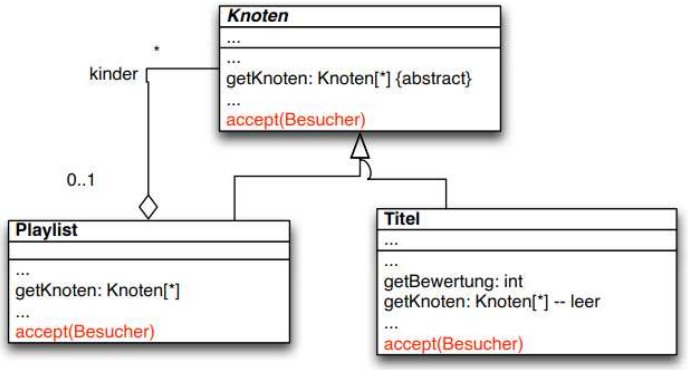


- Implementierung einer Besucher-Klasse:

```
public class BewertungBesucher extends Besucher {
   int summe; int anz;
   public void visit(Titel t) {
      summe += t.getBewertung(); anz++;
   }
   public void visit(Playlist p) {
       for (Knoten k: p.getKnoten()) {
          k.accept(this);
       }
   }
   public void visit(Knoten k) {
      //... wird das jemals aufgerufen? ...
      EH.Assert(false,...,"Unbehandelter Knotentyp {0}",k);
   }
   public double getBewertung() {
      if (anz>0) {return ((double)summe)/anz;}
      else {return 0.0;}
   }
```

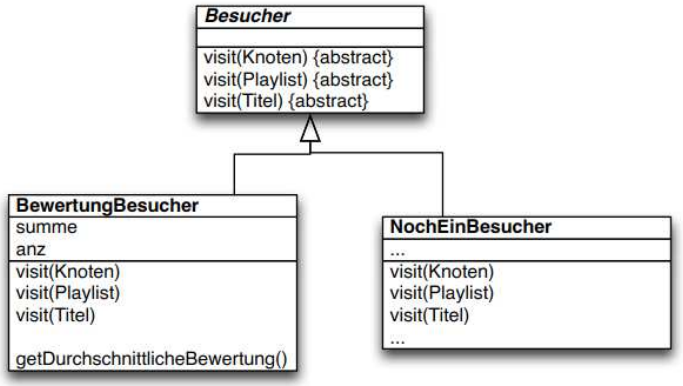


UMSETZUNG: 3. VERSUCH – DIESES MAL KLAPPT‘S!



- Verwendung einer Besucherklasse:

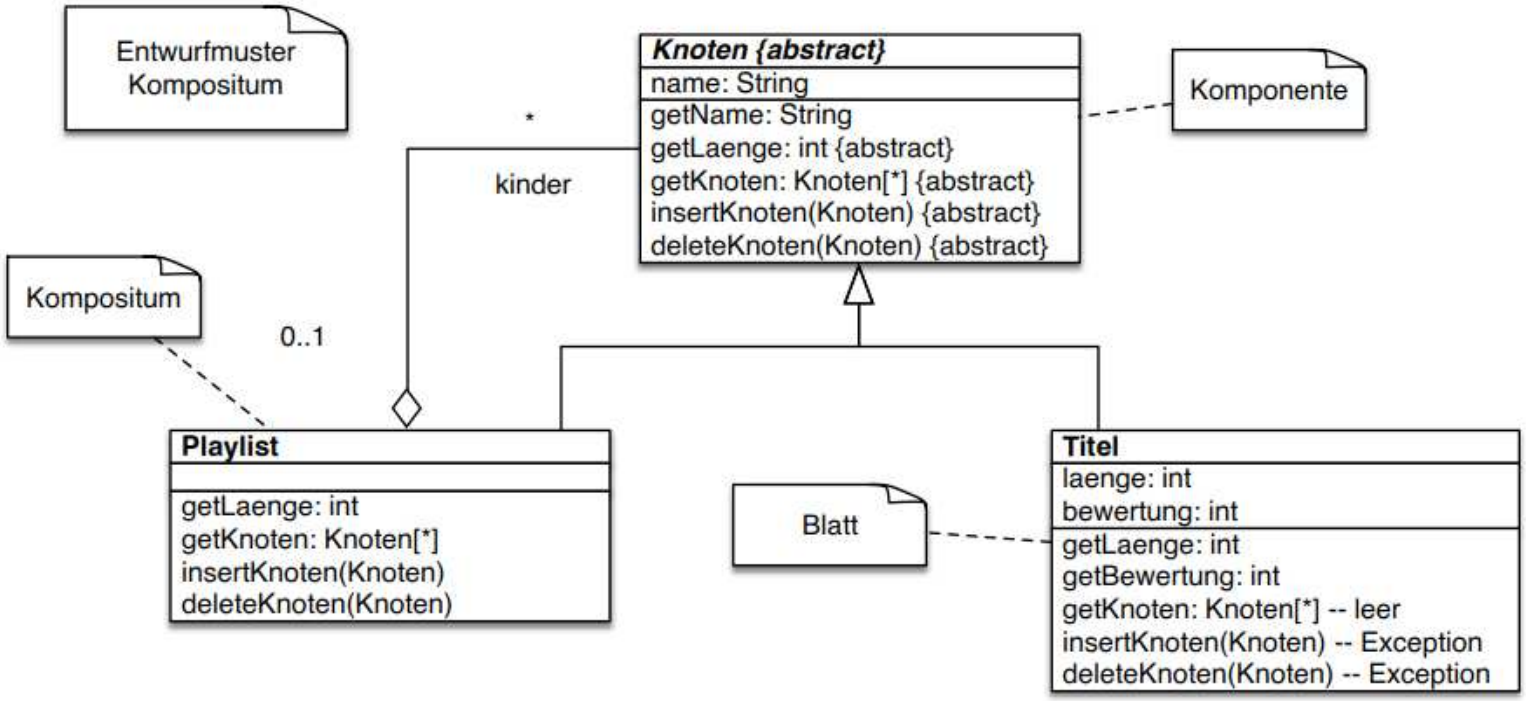
```
// hinter diesem Knoten haengt
// ggfs. ein ganzer Baum
Knoten k = ...;
BewertungBesucher b = new BewertungBesucher();

// Ausloeser fuer rekursiven Abstieg
// bis hin zu allen Blaettern
k.accept(b);

System.out.println(b.getBewertung());
```



BEISPIEL FEINENTWURF „PLAYLISTENVERWALTUNG“

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Wir erinnern uns:



→ Kompositum-Muster

- Jetzt: Besucher-Muster

→ Beide harmonieren miteinander

→ Gutes Beispiel für sich ergänzende / verstärkende Muster

BESUCHER-MUSTER IM ÜBERBLICK

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Kern des Entwurfsmusters:
  - Wechselseitiger Aufruf von *accept* und *visit* (2 Mal Polymorphismus):
    - *knoten.accept(besucher)* → offenbart tatsächlichen Typ von *knoten*
    - *besucher.visit(knoten)* → offenbart tatsächlichen Typ von *besucher*

    ⇒ *visit* kennt tatsächlichen Typ von *knoten* und *besucher*

- Verwendung:
  - oft zusammen mit „Kompositum“
  - ABER: Kompositum ist keine zwingende Voraussetzung
    - Können auch andere komplexere Datenstrukturen sein, die traversiert werden müssen (z.B. Bäume, …)

  → Gut, wenn komplexe Strukturen für verschiedenen Funktionalitäten durchtraversiert werden müssen

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

03b
Entwurfsmuster – Beobachter (Observer)

Ziel:
Entwurfsmuster Besucher genauer kennenlernen

INTERAKTIVE PROGRAMME

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Heutzutage haben wir Programme, die mit den Benutzern interagieren
  → Interaktive Programme

- Interaktive Programme:
  – Programm wartet auf Aktion des Benutzers
  – Aktion löst ein Ereignis aus (Maus-Klick, Button-Klick,…)

- Frage: Wie löst man diese Ereignis-Verarbeitung am geschicktesten?
  – Nicht starr, sondern möglichst flexibel

INTERAKTIVE PROGRAMME-BEOBACHTER-MUSTER

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Wie löst man diese Ereignis-Verarbeitung am geschicktesten?

  → **Beobachter Muster:**

  - Zwei grundlegende Phasen im Ablauf:
    - Phase 1 (vor dem Start der Ereignis-Verarbeitung)
      - Ereignis-Empfänger registrieren
    - Phase 2 (während der Ereignis-Verarbeitung)
      - Ereignis tritt ein → alle registrierten Ereignis-Empfänger benachrichtigen
    - (Evtl. Phase 3)
      - Wenn nicht mehr benötigt, Ereignis-Empfänger abmelden
  - Andere Namen:
    - Call-Back, Listener, Observer, Publisher-Subscriber, . . .

INTERAKTIVE PROGRAMME-BEOBACHTER MUSTER

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

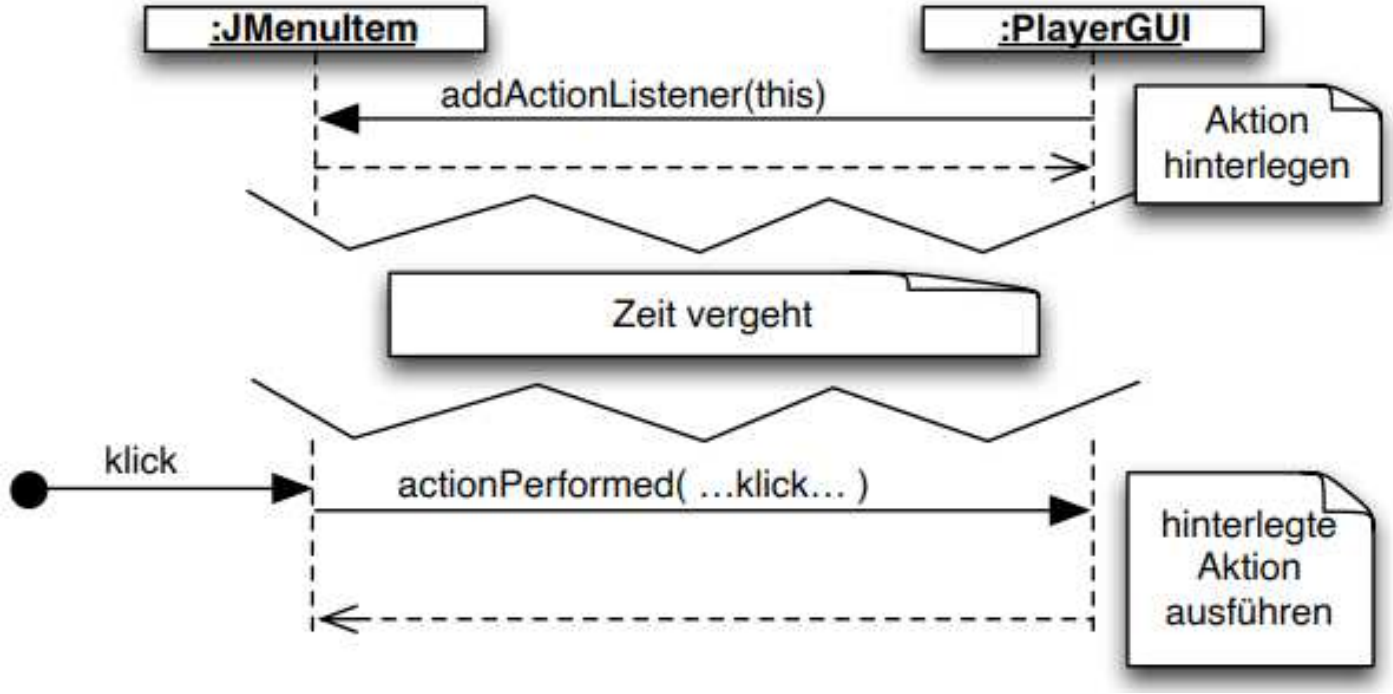
- Code-Beispiel: Ereignis „Klick auf Menüentrag“

```
01 public class PlayerGUI extends JPanel implements ActionListener
02 {
03       JMenuItem newPlaylist;
04       public MeineGUI() {
05           ...
06           newPlaylist=new JMenuItem("New Playlist");
07           // hinterlegte Aktion(en) registrieren
08           newPlaylist.addActionListener(this);
09       }
10
11       public void actionPerfromed(ActionEvent arg0) {
12           // hinterlegte Aktion(en) ausfuehren
13           ...
14       }
15       ...
16 }
```

INTERAKTIVE PROGRAMME-BEOBACHTER MUSTER

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Code-Beispiel: Ereignis „Klick auf Menüentrag“



DAS BEOBACHTER-MUSTER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Kurzbeschreibung:
  - „Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt werden.“

- Andere Namen:
  - Observer, Publisher-Subscriber, Call-Back, Listener

* Nach Gamma et al.

DAS BEOBACHTER-MUSTER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Problem:
  - Mehrere miteinander interagierende Klassen → Aufrechterhalten der Konsistenz zwischen den Objekten

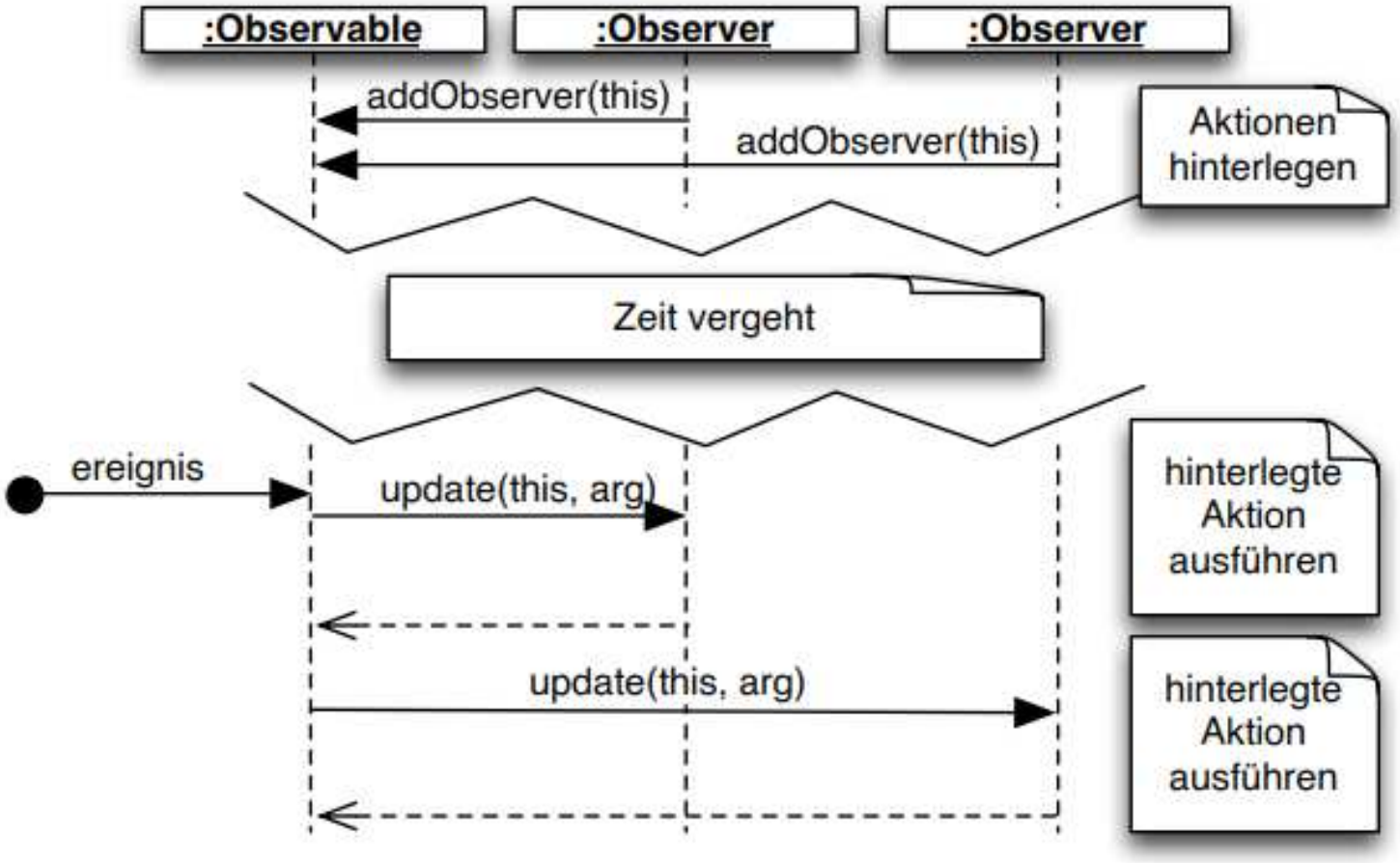
- Lösung: Aufteilung (grob)
  - Subjekt (Observable)
    - kennt seine Beobachter
    - hat Methoden zum An-/Abmelden von Beobachtern
    - hat Methode zum Benachrichtigen der Beobachter
  - Beobachter (Observer)
    - hat Methode, die vom Subjekt bei Veränderungen aufgerufen wird

*Nach Gamma et al.

DAS BEOBACHTER-MUSTER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Grober Ablauf mit zwei Beobachtern:

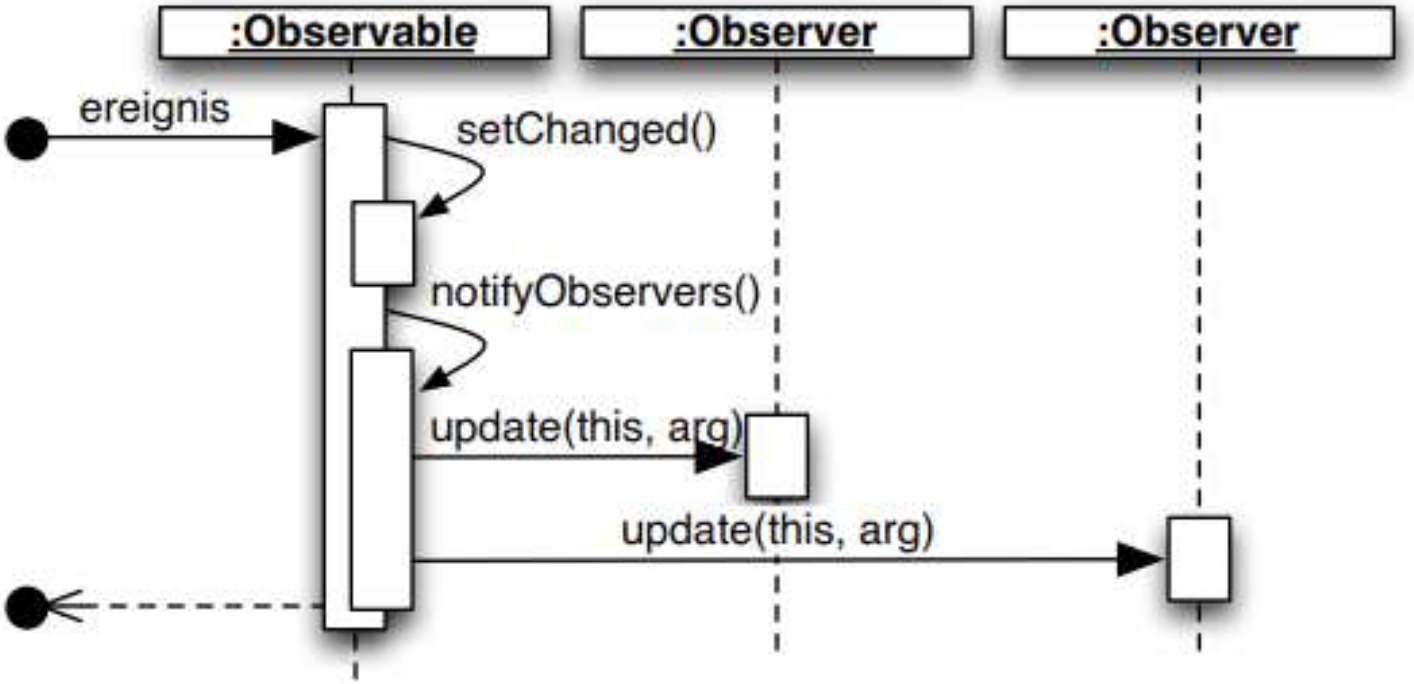


* Nach Gamma et al.

DAS BEOBACHTER-MUSTER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

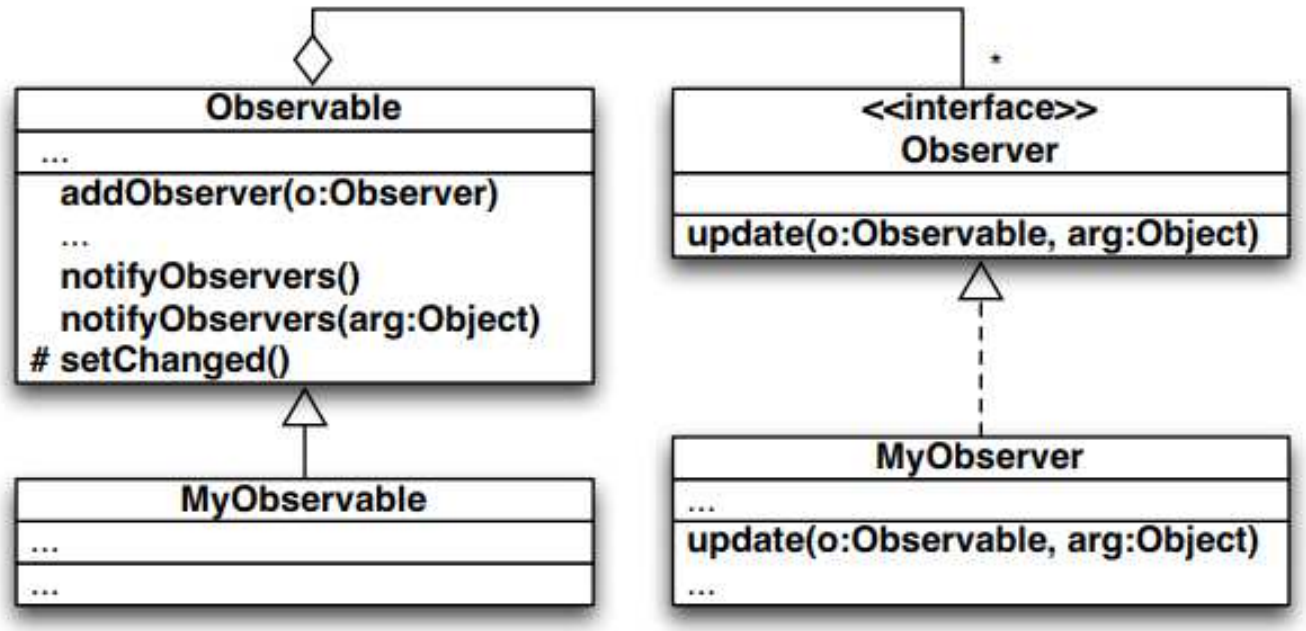
- Details des Ablaufs:



* Nach Gamma et al.

DAS BEOBACHTER-MUSTER*

- Klassendiagramm:

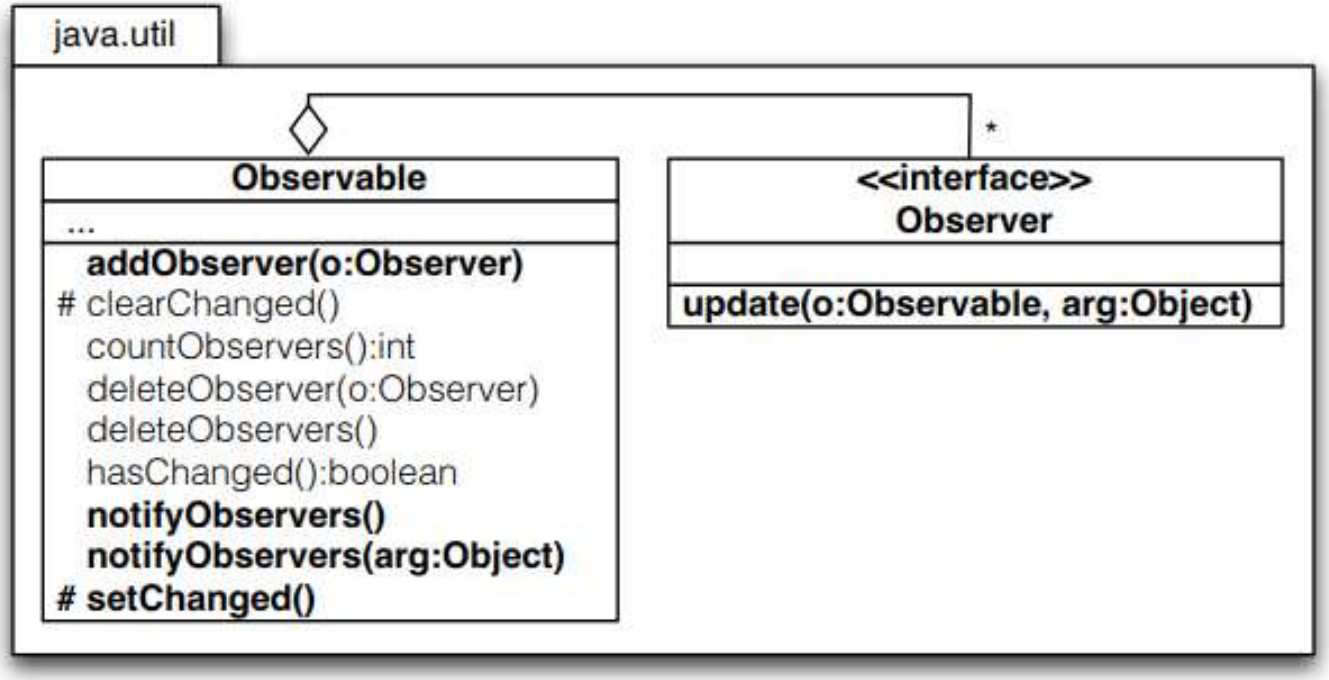


* Nach Gamma et al.

DAS BEOBACHTER-MUSTER IN JAVA

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Der allgemeine Teil des Musters ist sogar in der Java API vorhanden*:



→ Eigenes Subjekt von Observable erben und für die Observer das Observer Interface implementieren

* Leider ab Java 9 deprecated ☹ → stattdessen PropertyChangeListener aus "java.beans“ verwenden

DAS BEOBACHTER-MUSTER IN JAVA

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Aber auch alle typischen Listener-Interfaces und die GUI-Komponenten auf die man sich für die Events registrieren kann folgen dem Observermuster
  - Es heisst nur Listener statt Observer
  - Die Eventmethoden haben andere -spezifischere- Namen und Parameter

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

03c
Entwurfsmuster – Gängige Muster*

Ziel:
Liste gängiger Muster kennenlernen

* Nach Gamma et al.

ERZEUGUNGSMUSTER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Abstrakte Fabrik (Abstract Factory)
- **Erbauer (Builder)**
- **Fabrikmethode (Factory Method)**
- Prototyp (Prototype, ≈ Objekterzeugung in JavaScript)
- Singleton
  - Ist für manche eher ein Idiom (auf Codeebene)
  - Mittlerweile kritisch gesehen, da es aufgrund seiner Starrheit Probleme beim Unittesting (v.a. bei Parallelisierung) geben kann

* Nach Gamma et al.

STRUKTURMUSTER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- **Adapter**
- **Brücke (Bridge)**
- Dekorierer (Decorator)
- Fassade (Facade)
- Fliegengewicht (Flyweight)
- **Kompositium (Composite)**

* Nach Gamma et al.

VERHALTENSMUSTER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- **Befehl (Command)**
- **Beobachter (Observer, Listener)**
- **Besucher (Visitor)**
- Interpreter
- **Iterator**
  - (in Java und C++ inzwischen fest eingebaut)
- Memento
- **Schablonenmethode (Template Method)**
- **Strategie (Strategy)**
- Vermittler (Mediator)
- Zustand (State)
  - (wird von manchen inzwischen als Anti-Muster angesehen)
- Zuständigkeitskette (Chain of Responsibility)

* Nach Gamma et al.

SONSTIGE VERHALTENSMUSTER

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Method-Object-Pattern
  - Hatten wir in Programmiermethoden, 1. Vorlesung
  - Kein GoF-Pattern
    - Sondern aus Kent Beck: Smalltalk Best Practice Patterns
  - IDEE: Eine Klasse steht für einen Algorithmus (Algorithmus wird in einer Klasse gekapselt)
  - Sehr ähnlich zu Template-Method-Pattern (aber ohne Vererbung)
  - Interagiert hervorragend mit:
    - Strategy (Auswahl verschiedener Algorithmen durch User)
    - Template-Method

      (Abstrakt formulierter Grundalgorithmus wovon geerbt und offene Methoden überschrieben werden können)

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

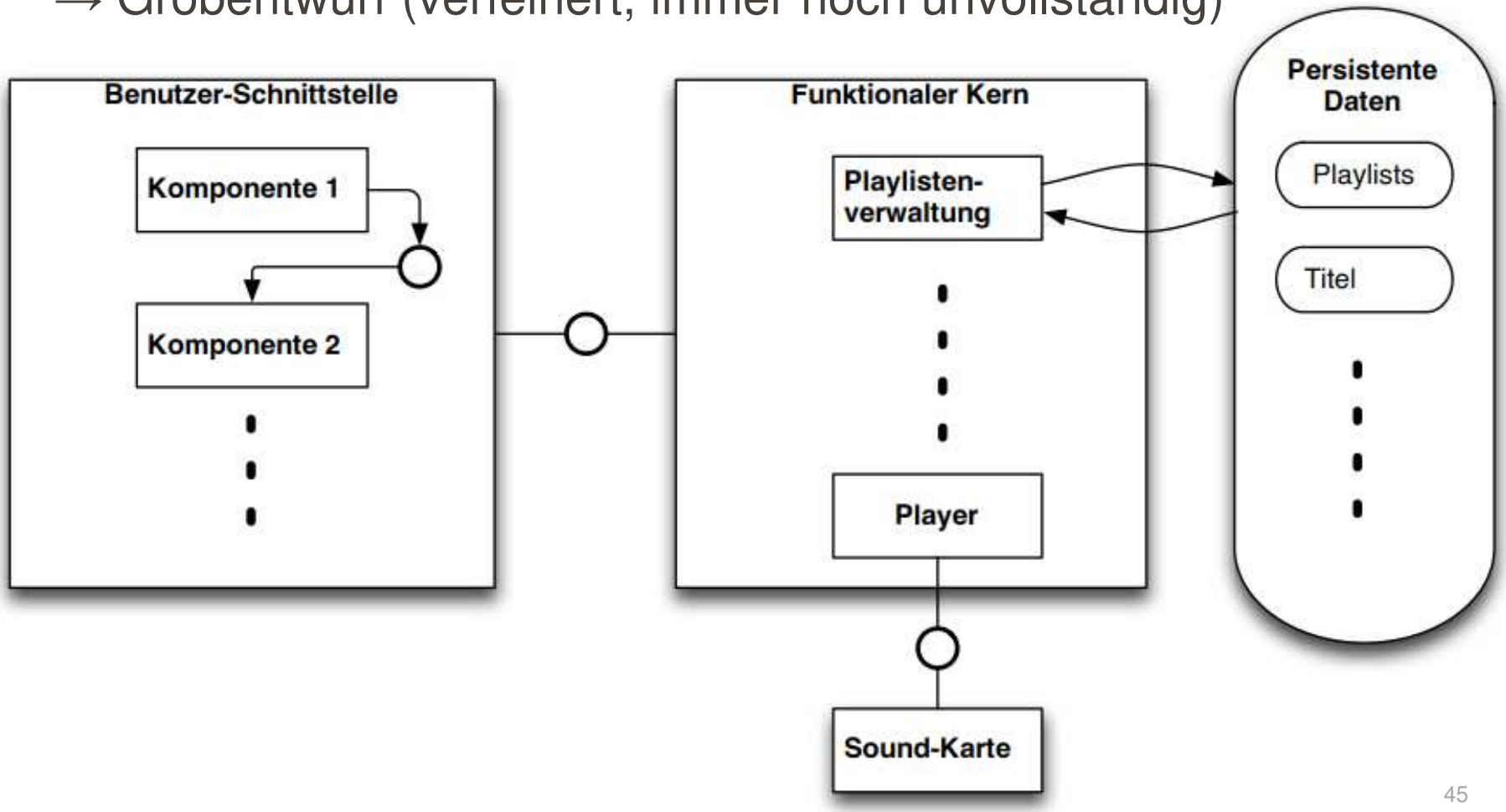
04a
Architekturmuster – 3-Schichten-Arch.

Ziel:
Die 3-Schichten-Architektur kennenlernen
(eigentlich Wiederholung)

BEISPIEL
3-SCHICHTEN-ARCHITEKTUR

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Beispiel: MP3-Player in 3-Schichten-Architektur:
  → Grobentwurf (verfeinert, immer noch unvollständig)

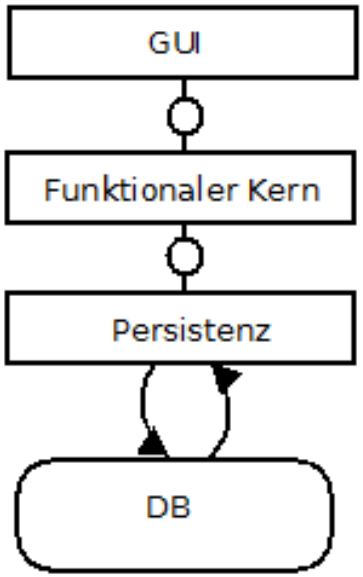


HINWEIS ZUR 3-SCHICHTEN-ARCHITEKTUR

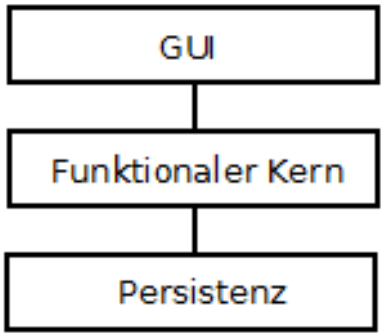
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Wird oft eher umgekehrt von oben nach unten dargestellt
  → Schichten

So (z.B. FMC):



Oder auch so (z.B. UML):

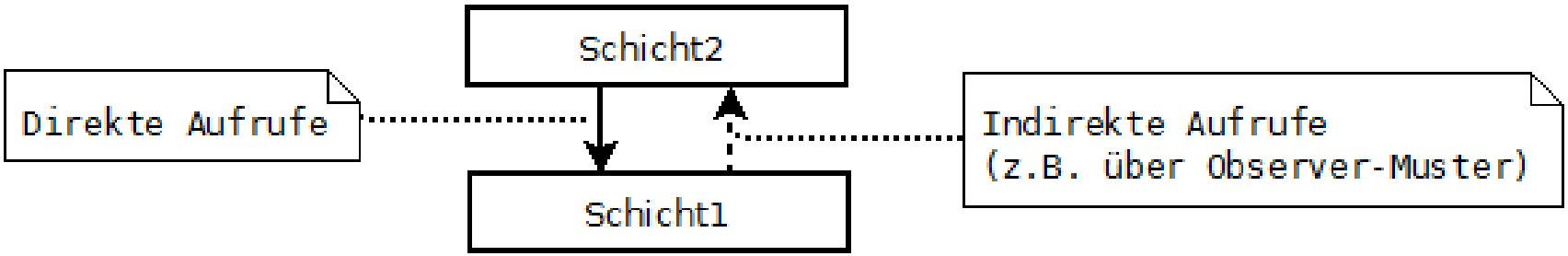


→ Oft wird DB weggelassen und als Teil der Persistenzschicht betrachtet
  - Kann aber auch ohne DB sein (z.B. dann XML-Dateien, …)
  - Deshalb: Besser auch die DB, … darstellen

WEITERER HINWEIS ZU SCHICHTEN-ARCHITEKTUREN

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Meist kennt nur eine Schicht die andere Schicht direkt
  - Meist obere Schicht kennt die darunter liegende Schicht direkt
  - Kommunikation von darunter liegender Schicht erfolgt indirekt
    - Z.B. über Observer-Muster



→ So wird eine Entkoppelung erreicht
(Siehe auch GRASP-Pattern Loose Coupling)

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

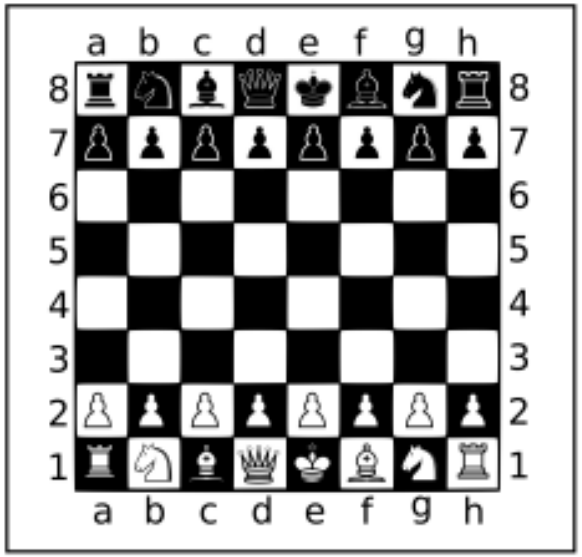
04b
Architekturmuster – MVC

Ziel:
Das ModelViewController-Muster als Architekturmuster kennenlernen

BEISPIEL:
EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Grafische Benutzeroberfläche (ohne Menü):



Bildquelle (und Lizenz):

http://commons.wikimedia.org/wiki/File:AAA_SVG_Chessboard_and_chess_pieces_02.svg

BEISPIEL: EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

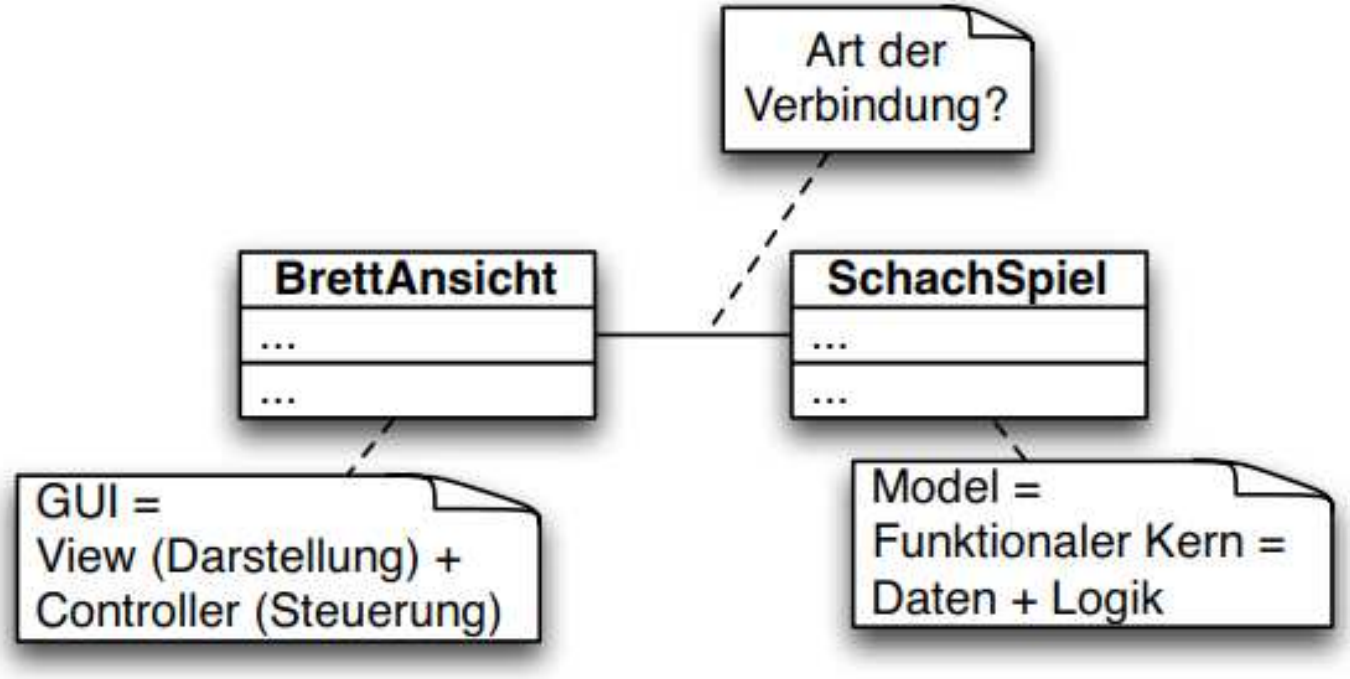
- Nichtfunktionale Anforderungen:
  - Ziel: Flexible GUI
    1. GUI soll leicht veränderbar sein, z.B.:
       - anderes Aussehen des Spielbretts
       - Swing → anderes GUI-Framework (z.B. Java FX)
    2. GUI soll leicht erweiterbar sein, z.B.:
       - neues Fenster: Historie der Spielzüge
       - neues Fenster: Direkteingabe

- Auswirkung auf Entwurf:
  1. Saubere Trennung zwischen GUI und Spielelogik
     - GUI: Darstellung + Steuerung (Maus/Tastatur)
     - Funktionaler Kern: Spielzustand (Daten) + Spielelogik
  2. Spezieller Benachrichtigungs-Mechanismus

BEISPIEL: EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Entwurf:

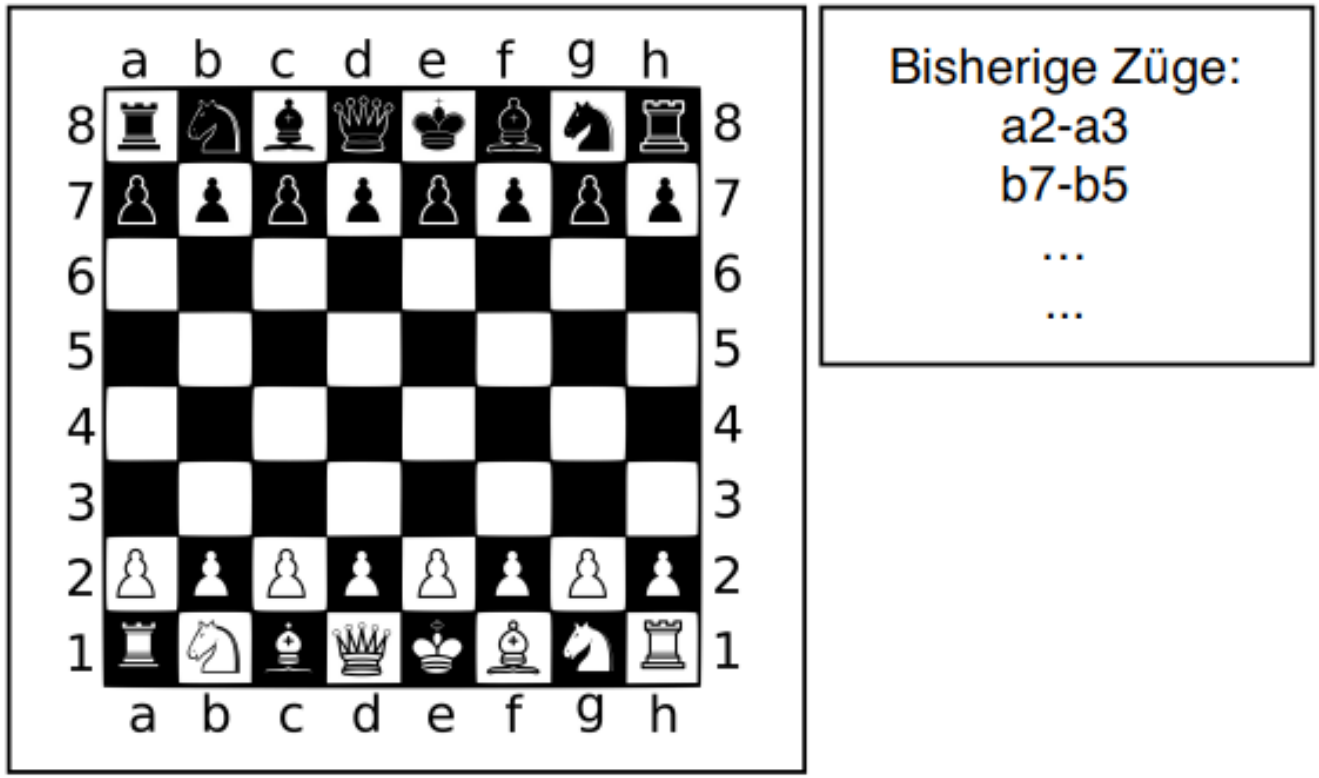


→ Prinzipielle Idee: Saubere Trennung zwischen GUI und Spielelogik

BEISPIEL: EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Erweiterte GUI mit neuem Fenster “Zugansicht”:



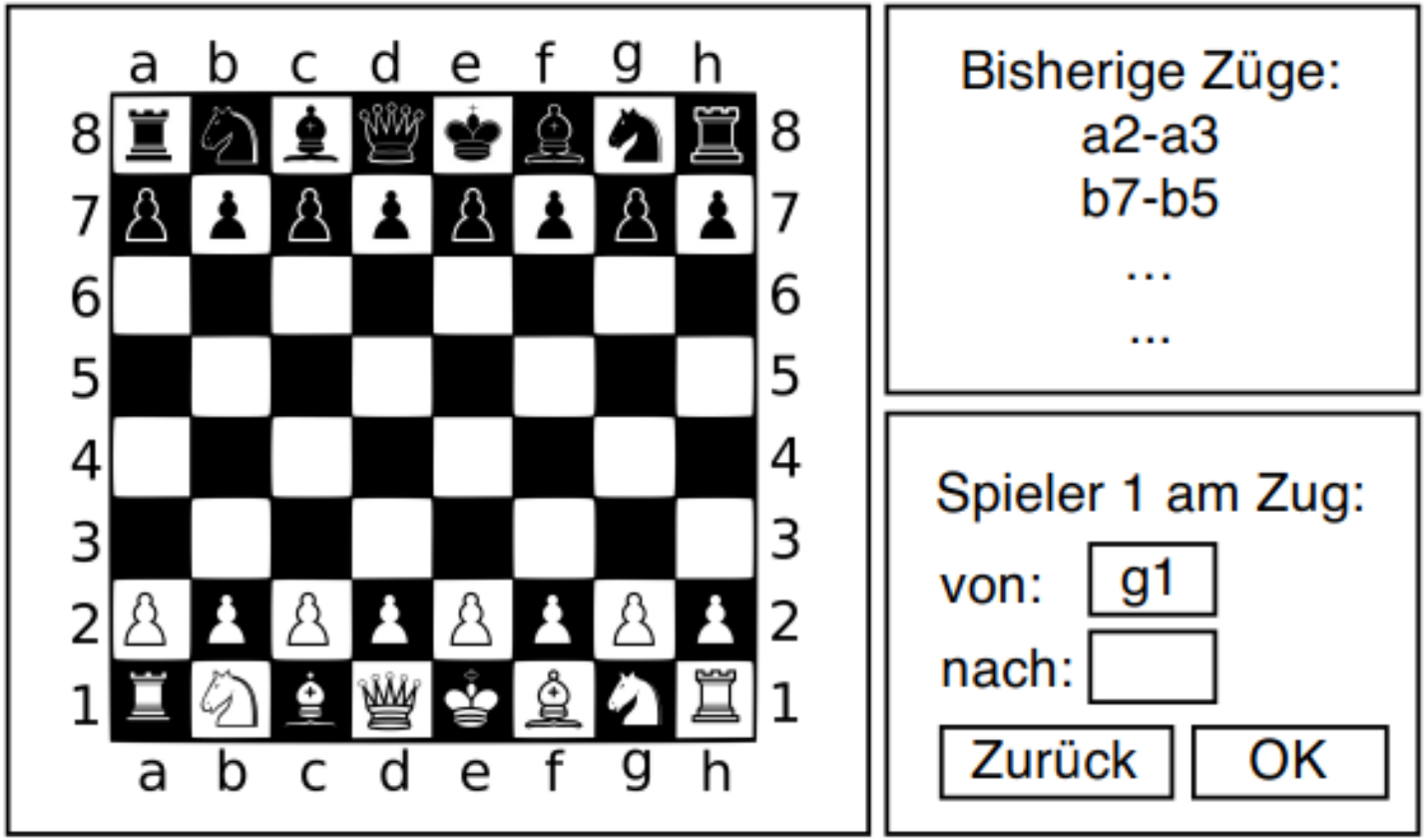
Bildquelle (und Lizenz):

http://commons.wikimedia.org/wiki/File:AAA_SVG_Chessboard_and_chess_pieces_02.svg

BEISPIEL: EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Erweiterte GUI mit neuem Fenster “Direkteingabe”:



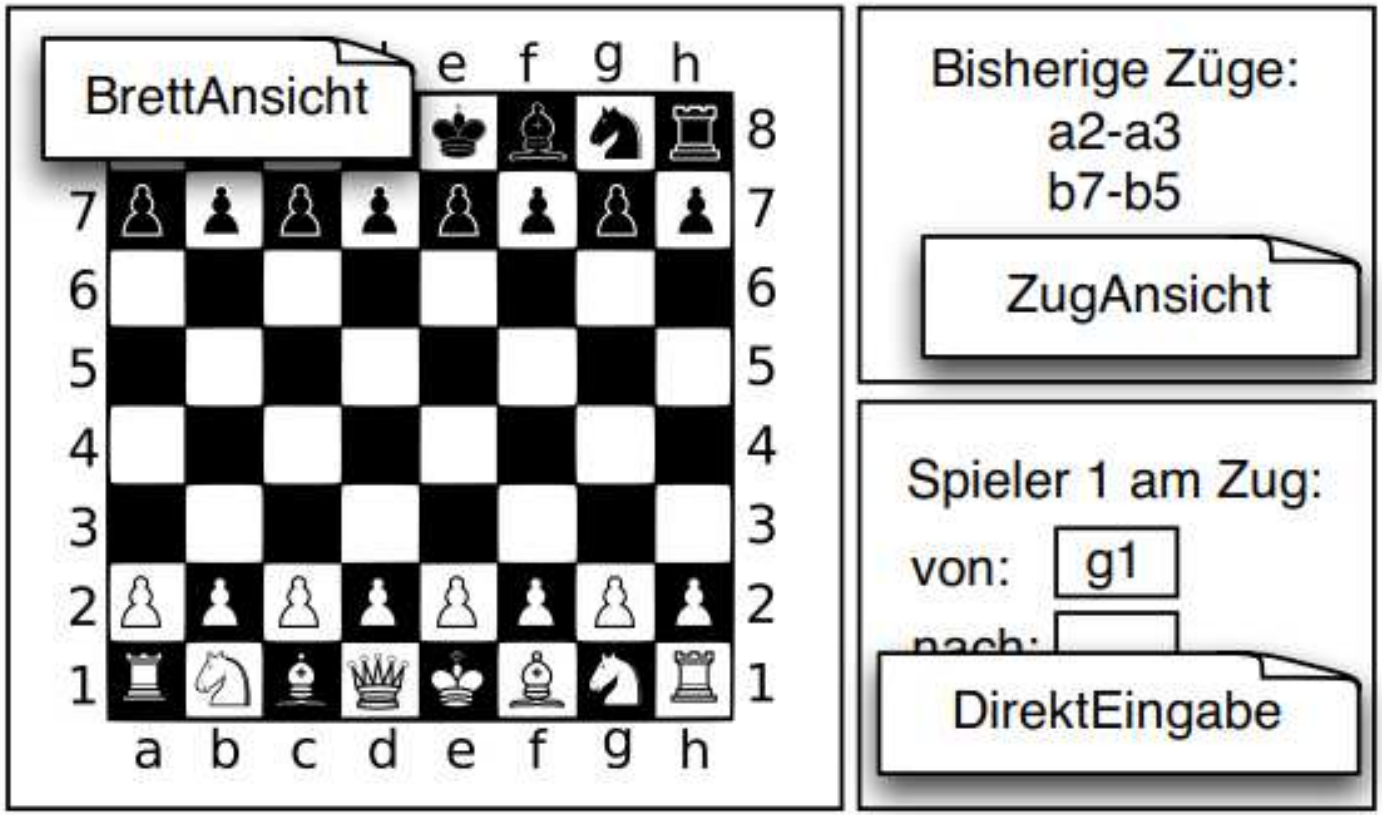
Bildquelle (und Lizenz):

http://commons.wikimedia.org/wiki/File:AAA_SVG_Chessboard_and_chess_pieces_02.svg

BEISPIEL: EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Erweiterte GUI – Klassennamen der Fenster:



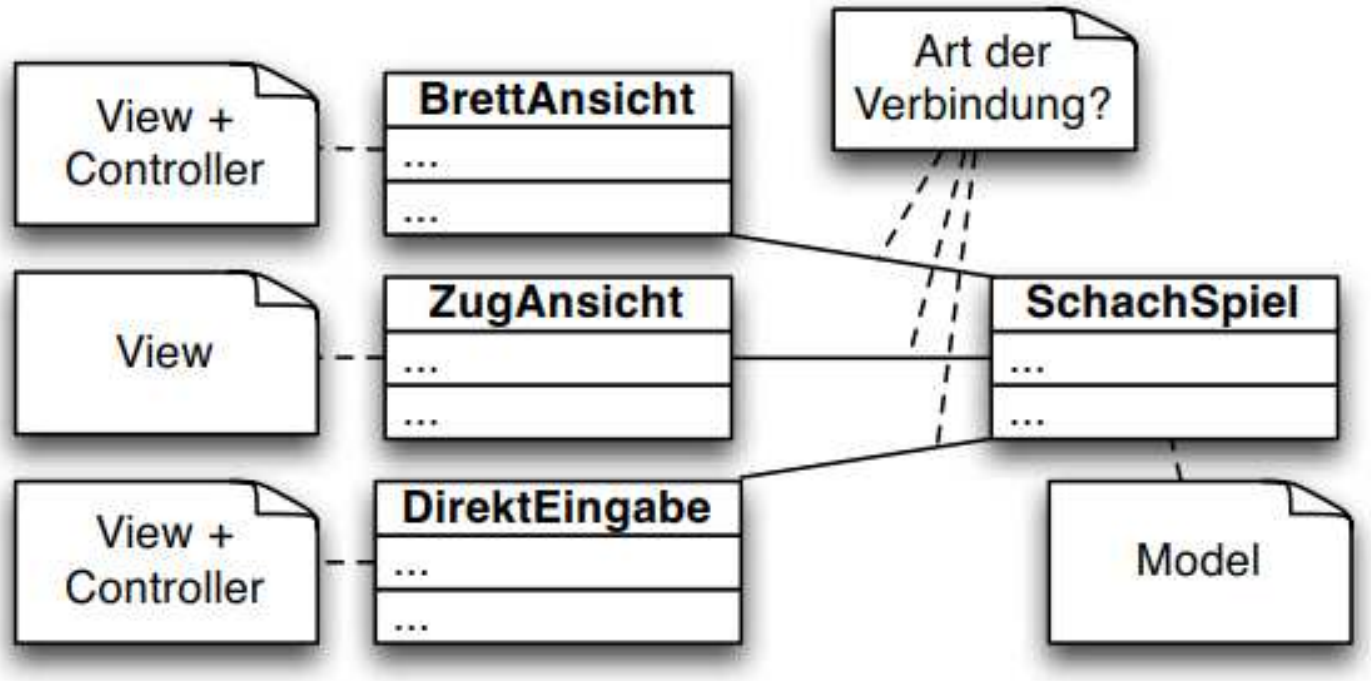
Bildquelle (und Lizenz):

http://commons.wikimedia.org/wiki/File:AAA_SVG_Chessboard_and_chess_pieces_02.svg

BEISPIEL: EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Erweiterter Entwurf:



→ Prinzipielle Idee: Saubere Trennung zwischen GUI und Spielelogik

BEISPIEL: EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Geeigneter Benachrichtigungsmechanismus?

- Mögliche Ereignisse:
  - BrettAnsicht: Bewegen einer Spielfigur
  - DirektEingabe: Eingabe und Bestätigung eines Zugs
- Problem:
  - Alle drei Ansichten (BrettAnsicht, ZugAnsicht, DirektEingabe)
  - Müssen stets den gleichen aktuellen Zustand anzeigen.
- Lösung:
  - Bei Veränderung des Spielzustands:
  - Nachricht an alle Fenster: „Es hat sich was verändert.“

  → Verwendung des Entwurfsmusters „Beobachter“

BEISPIEL: EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Geeigneter Benachrichtigungsmechanismus?

- Mögliche Ereignisse:
  - BrettAnsicht: Bewegen einer Spielfigur
  - DirektEingabe: Eingabe und Bestätigung eines Zugs
- Problem:
  - Alle drei Ansichten (BrettAnsicht, ZugAnsicht, DirektEingabe)
  - Müssen stets den gleichen aktuellen Zustand anzeigen.
- Lösung:
  - Bei Veränderung des Spielzustands:
  - Nachricht an alle Fenster: „Es hat sich was verändert.“

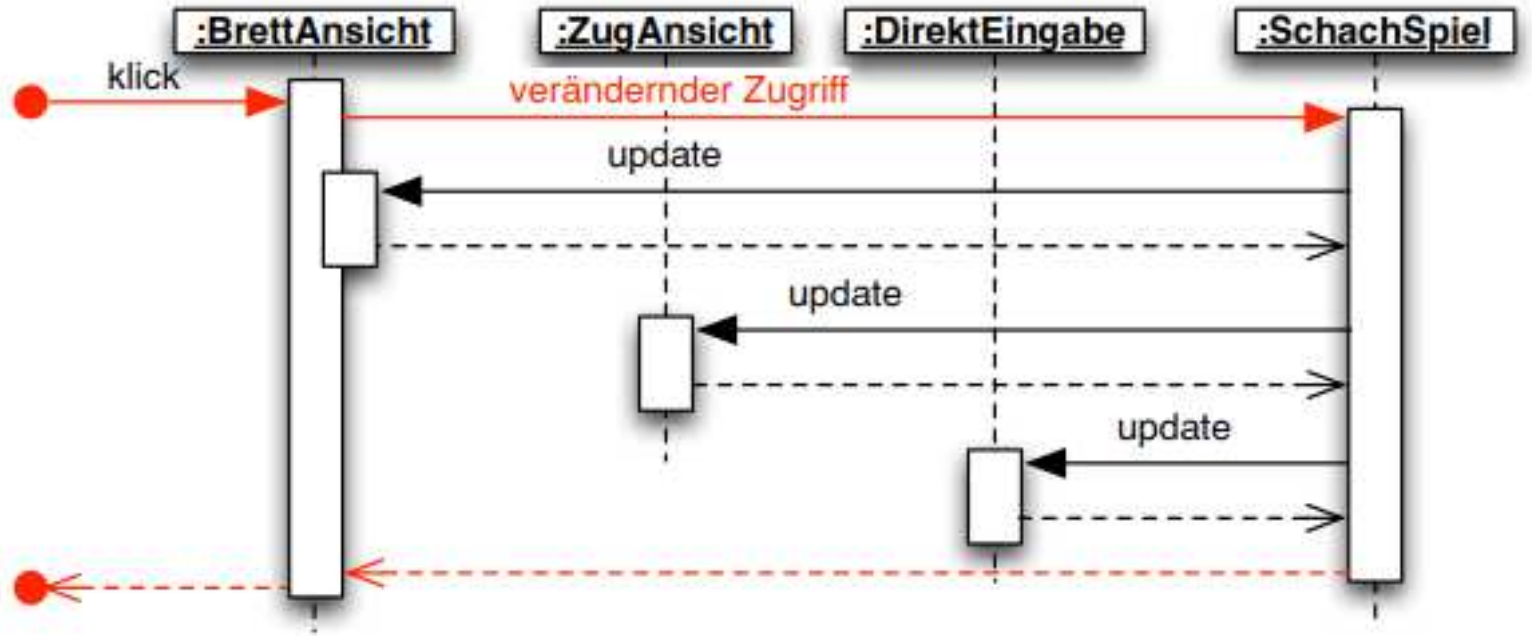
  → Verwendung des Entwurfsmusters „Beobachter“

BEISPIEL:
EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Geeigneter Benachrichtigungsmechanismus?

- Grober Ablauf bei drei Fenstern
  - Auslöser: Klick in BrettAnsicht (z.B. Drag-And-Drop)

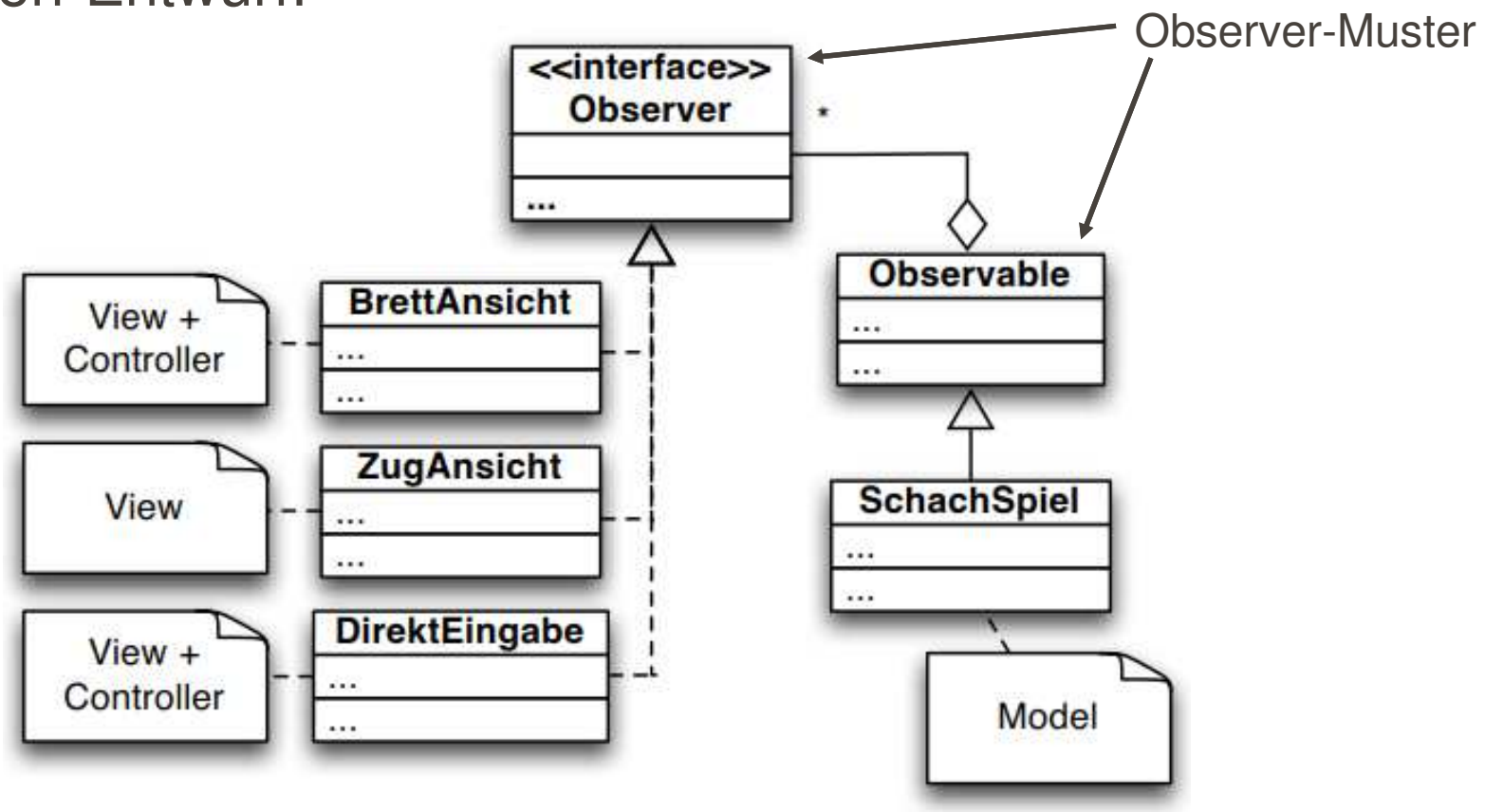


BEISPIEL:
EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Geeigneter Benachrichtigungsmechanismus?

- Klassen-Entwurf:

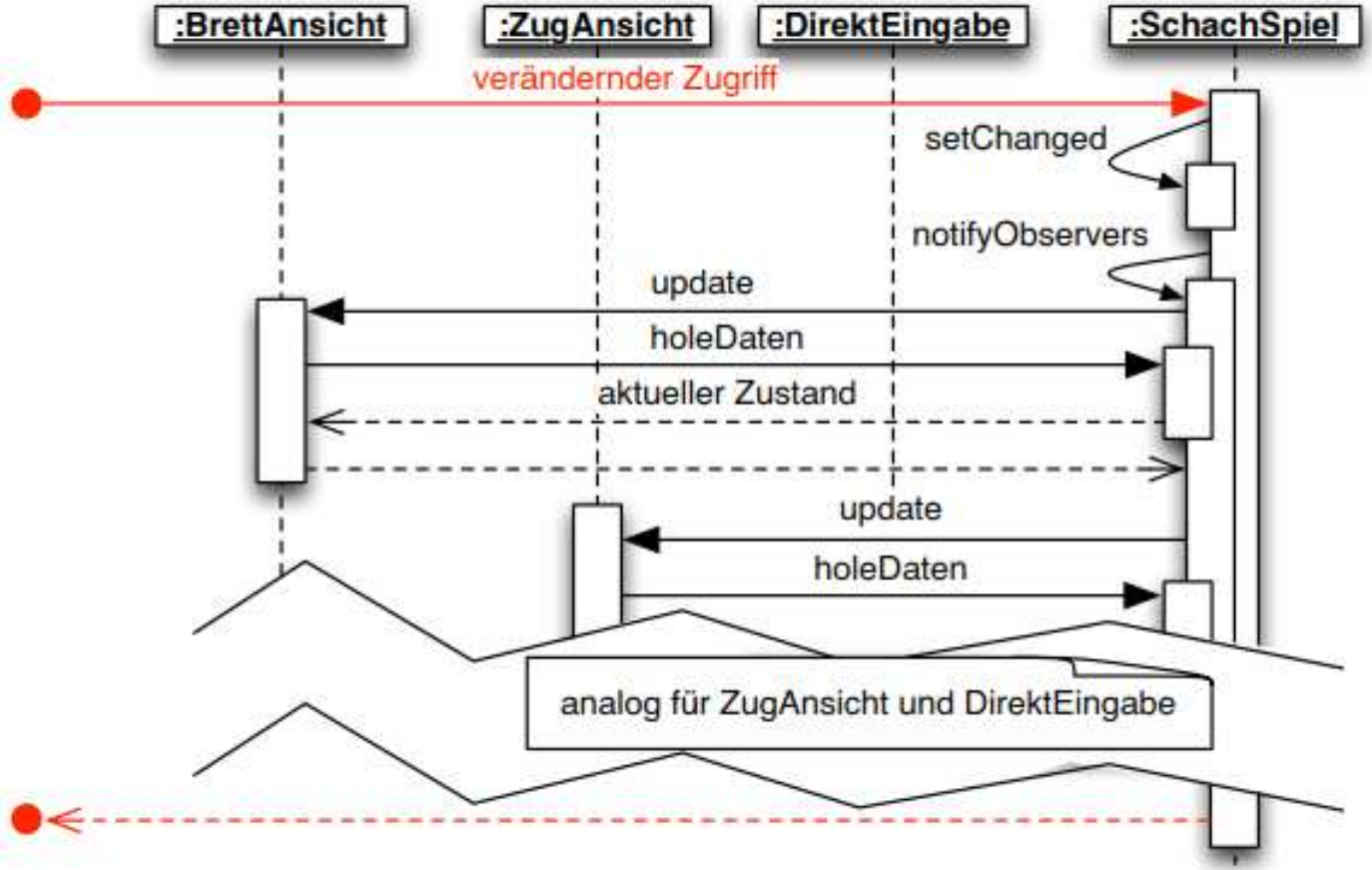


BEISPIEL:
EIN SCHACHPROGRAMM

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Geeigneter Benachrichtigungsmechanismus?

- Ablauf im Detail:



DAS ARCHITEKTURMUSTER MODEL-VIEW-CONTROLLER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Kurzbeschreibung:
  - „Das Model-View-Controller-Muster (MVC) unterteilt eine interaktive Anwendung in drei Komponenten. Das Modell enthält die Kernfunktionalität und die Daten. Ansichten (engl. views) präsentieren dem Anwender Informationen. Steuerungskomponenten sind für die Bedieneingaben verantwortlich. Ansichten und Steuerungskomponenten zusammen umfassen die Benutzerschnittstelle. Ein Mechanismus zur Benachrichtigung über Änderungen [...] sichert die Konsistenz zwischen der Benutzerschnittstelle und dem Modell.“
- Kontext:
  - Interaktive Anwendungen mit einer flexiblen Mensch-Maschine - Schnittstelle

* Nach Buschmann et al.

DAS ARCHITEKTURMUSTER MODEL-VIEW-CONTROLLER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Problem:
  - enge Kopplung zwischen GUI und Funktionalität macht GUI inflexibel
- Lösung:
  - Unterteilung einer Anwendung in drei Bereiche
    - Model = nur Kernfunktionalität
    - View = nur Darstellung der Daten aus dem Model
    - Controller = nur Benutzereingabe entgegennehmen und entsprechende Methode im Model aufrufen
  - Model informiert
    - interessierte Views
    - interessierte Controller

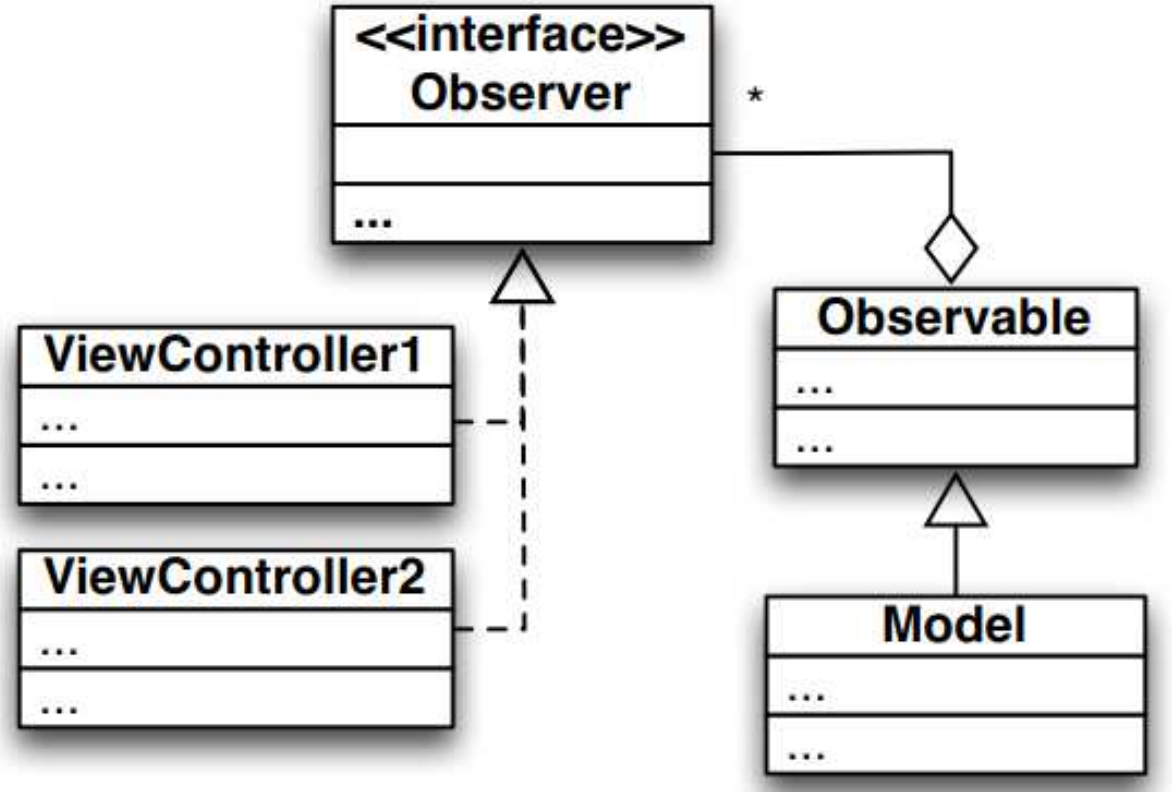
    über Veränderungen

* Nach Buschmann et al.

DAS ARCHITEKTURMUSTER MODEL-VIEW-CONTROLLER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Lösung (Klassen-Entwurf):

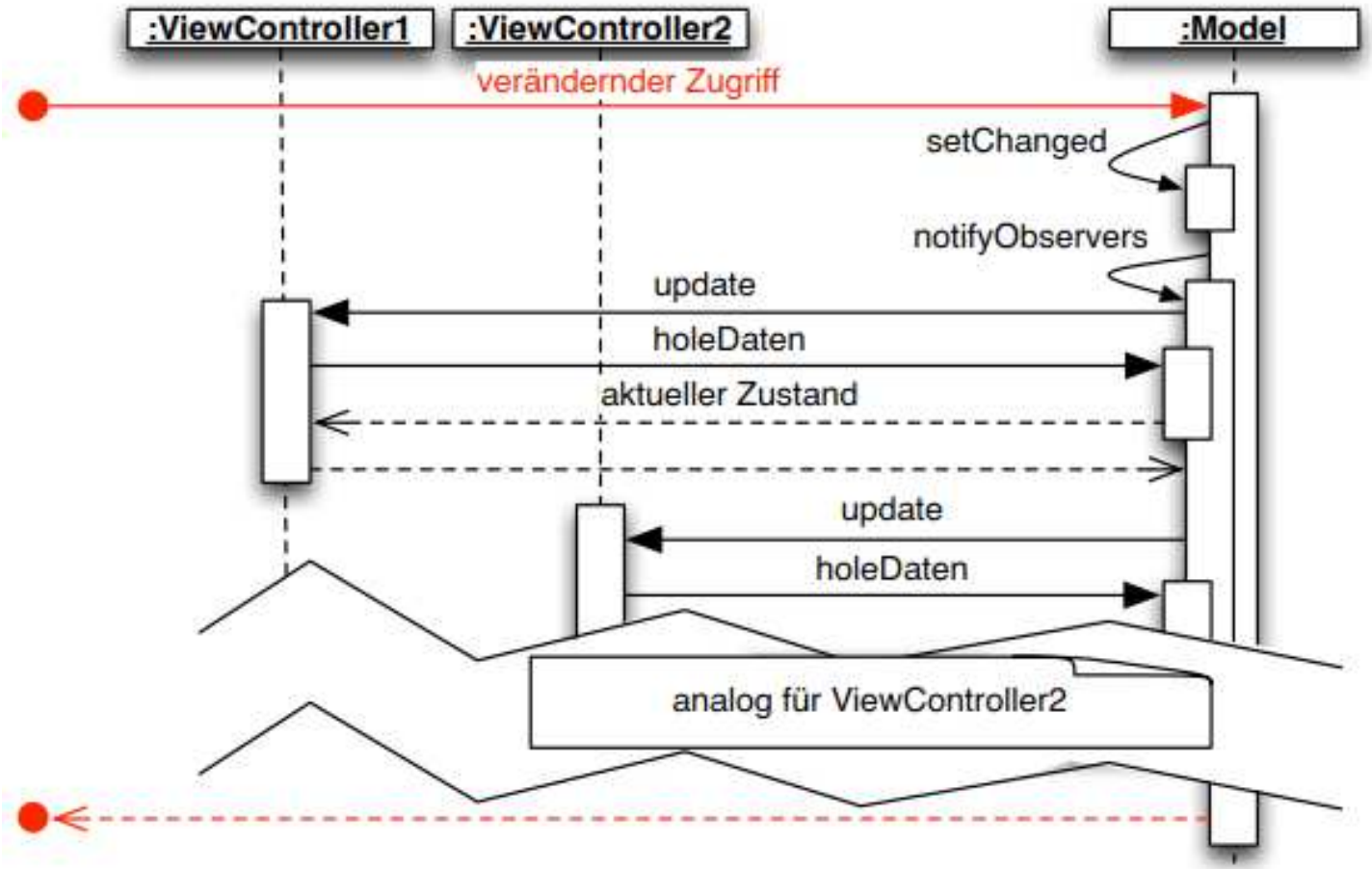


* Nach Buschmann et al.

DAS ARCHITEKTURMUSTER MODEL-VIEW-CONTROLLER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Lösung (Ablauf):



* Nach Buschmann et al.

MODEL-VIEW-CONTROLLER – VARIANTEN:

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Aufteilung von View und Controller:
  - Variante 1: View und Controller in einem Objekt
    - Z.B. Java Swing
  - Variante 2: View und Controller je in einem separatem Objekt
    - Das „Original“ aus den Achtzigern (Smalltalk)
    - War aber damals sehr langsam → Heute nicht mehr so das Problem
- Datenzugriff:
  - Variante 1: View holt aktuellen Zustand vom Model (Methoden in Model für „hole Daten“)
  - Variante 2: aktueller Zustand als Parameter in update

MODEL-VIEW-CONTROLLER – WEITERE BEMERKUNGEN:

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Im letzten Semester Programmiermethoden:
  - JTree, TreeModel,… → Funktioniert nach MVC-Muster
  - Aber: Nur eine einzelne GUI-Komponente

  → Eher ein Entwurfsmuster (für GUI-Komponenten)
    - Tatsächlich der Ursprung des MVC-Musters

    → MVC ist Grundlage für Java AWT/Swing (und andere GUI-Frameworks)
- Hier:
  - Eher auf Ebene des gesamten Programms
  - Das gesamte Programm funktioniert nach diesem Prinzip

  → Architekturmuster

→ D.h. es kann ein Muster wie MVC auch auf verschiedenen Ebenen des Designs geben

MODEL-VIEW-CONTROLLER – WEITERE BEMERKUNGEN:

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Vorsicht: Namensverwirrung

- „MVC“ für Web-Anwendungen ≠ MVC das hier besprochen

- „MVC“ für Web-Anwendungen:
  - Ursprünglicher Name „Model-2“
  - 3-Schichten-Architektur für den Aufbau einer Web-Anwendung:
    - „View“: Darstellung + ggfs. Steuerung
    - „Controller“: Geschäftslogik + ggfs. Steuerung
    - „Model“: Daten
  - zwingende Trennung von „View“ und „Controller“
  - **keine** Verbindung zwischen „View“ und „Model“
  - **keine** Verwendung des Observer-Musters

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

04c
Architekturmuster – Gängige Muster

Ziel:
Liste gängiger Muster kennenlernen

KATALOG AN ARCHITEKTURMUSTERN*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

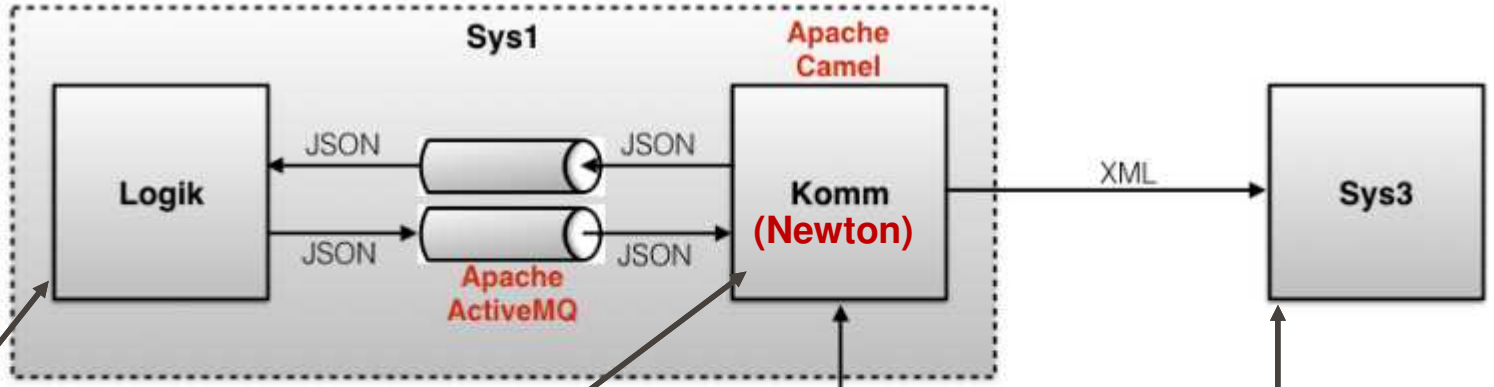
- **Layers (Schichtenarchitektur)**
  - Allgemein Schichtenarchitektur
  - Verbreiteste: 3-Schichtenarchitektur
  - Aber auch: OSI-Schichtenmodell oder TCPIP (vgl. Netzwerk-technik),…
- Pipes-and-Filters (≈ UNIX-Pipes, Funktionale Progr., …)
- Blackboard
- Broker
- **Model-View-Controller (MVC)**
- Presentation-Abstraction-Control
  - Gute Idee bei umfangreichen GUIs
- Microkernel
- **Reflection** (siehe Programmiermethoden (auch Prinzip von Simple))

*Nach [Buschmann et al]

BSP ZU PIPES-AND-FILTERS-ARCHITEKTURMUSTER

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Anwendungsarchitektur in einem Logistikunternehmen:



Sys1.Logik: Frachtanfragen werden mit Transportmitteln gematcht und ein Angebot erstellt

Sys1.Komm: Eigener Newton-Server zum Routen von Nachrichten über Apache Camel
→ Integriert eigentl. 3 Anwendungen miteinander

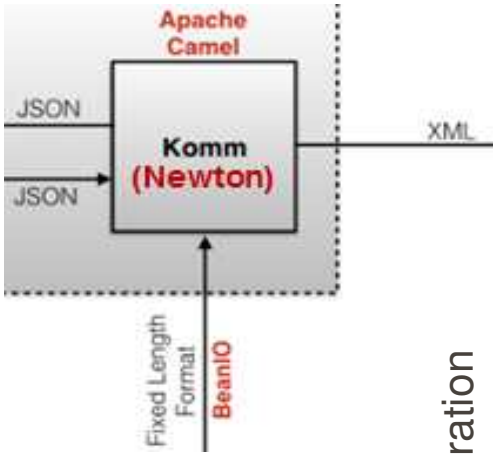
Transportaufträge (Fracht, Transportmittel und Route) gehen über XML an Sys3 zur Frachtverfolgung

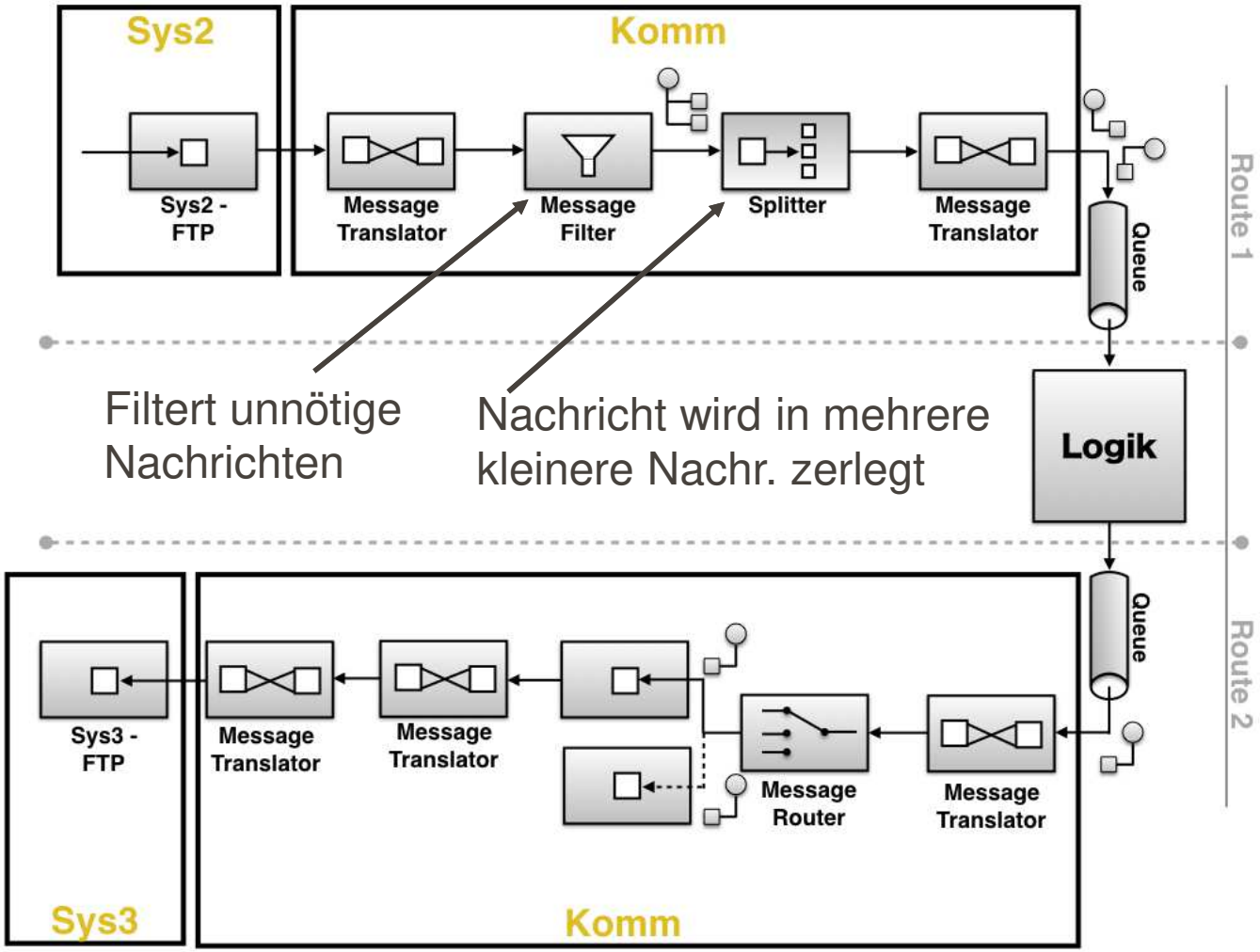
Sys2 (internationales System) stellt per FTP Transportmittel und Transportrouten zur Verfügung

(Vergleiche auch Unternehmensarchitektur!)

BSP ZU PIPES-AND-FILTERS-ARCHITEKTURMUSTER

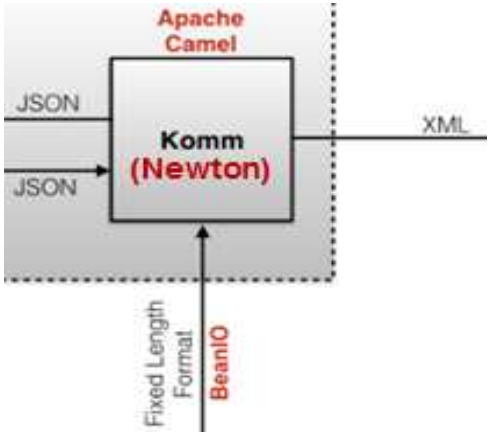
- Routendef. mittels Enterprise Integration Patterns* (EIP):





*Siehe: G. Hohpe, B. Woolf: Enterprise Integration Patterns, Addison Wesley, 2004.

BEISPIEL ZU PIPES-AND-FILTERS



- Codebsp. Route 1 mittels Apache Camel:

```
public void configRoute1() throws Exception{
    Sys2MessageDataFormat dataFormat=   new Sys2MessageDataFormat();
    from("ftp://sys2@xyftp.de:21/infodat?user=usr1&password=pwd123")
        .routeId("Route1")
        .autostartup(autostart)
        .bean(dataFormat, "unmarshal(*)")
        .bean (MessageFilter.class)
        .split().simple("${body}")
        .bean(Sys2ToSys1MessageConverter.class)
        .bean(toJsonFormat.class)
        .to("jms:queue:EingehendeQueue");
}
```

Hier wird auch Reflection benutzt
→ Diese Klassen müssen auch entw. werden

→ Apache Camel ist eine Domänenspezifische Sprache zur Umsetzung von Kommunik. zwischen Systemen mittels EIPs

→ Funktioniert wie **Funktionale Programmierung**

→ Pipes-And-Filters-Prinzip (Muster) auf Architekturebene

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

05
Allgemeine Prinzipien

Ziel:
Die sog. GRASP-Muster als Beispiel
für allgemeine Prinzipien kennenlernen
(siehe auch: Applying UML with Patterns; Kap. 16)

WAS IST GRASP*?

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- GRASP = General Responsibility Assignment Software Patterns
  - engl. “to grasp something” = “etwas verstehen/begreifen”
- Sammlung von allgemeingültigen Entwurfsprinzipien
  - Verteilung von Zuständigkeiten auf Klassen
  - allgemeiner als Entwurfsmuster
  - keine rezepthafte Lösung (wie z.B. Muster)
- Zusammenhang mit Entwurfsmustern:
  - Viele Entwurfsmuster unterstützen eines oder mehrere GRASP-Prinzipien

    → Viele Entwurfsmuster entstanden als eine mögl. Lösung für die Umsetzung eines oder mehrerer GRASP-Prinzipien

* Vgl. z.B.: Applying UML with Patterns; Kap. 16

CREATOR*
(= ERZEUGER-PRINZIP)

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Eine Klasse B ist besser geeignet Objekte einer Klasse A zu erzeugen, wenn eine der folgenden Bedingungen gilt:
  - B ist eine Aggregation von A (B besteht aus A-Objekten)
  - B enthält A-Objekte (contains)
    - Z.B. Liste, die A-Objekte enthält
  - B erfasst A-Objekte (records)
    - Z.B. Fehlerlog, das einzelne Fehlereinträge erfasst
  - B verwendet A-Objekte mit starker Kopplung
    - Starke Abhängigkeit der Klasse B von A
  - B die Initialisierungsdaten für A hat
    - D.h. B ist Experte bezüglich Erzeugung von A
    - B erhält sie z.B. über Konstruktor

*Vgl. Applying UML with Patterns; Kap. 16.7

LOW COUPLING*
(LOSE KOPPLUNG)

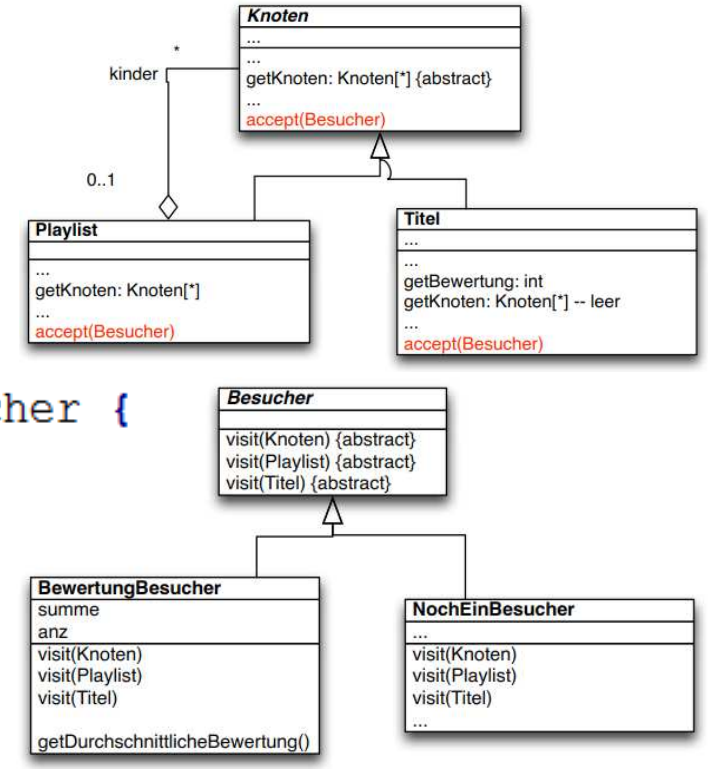
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Kurzbeschreibung:
  - Vermeide unnötige Verbindungen
  - Vermeide insbesondere Verbindungen zu instabilen Klassen/Schnittstellen

- Beispiel: MVC
  - Trennung zwischen GUI und funktionalem Kern
  - Funktionaler Kern ist **nicht abhängig** von (instabiler) GUI

* Vgl. Applying UML with Patterns; Kap. 16.8

HINWEIS: BESUCHER & LOSE KOPPLUNG:




```
public class BewertungBesucher extends Besucher {
   int summe; int anz;
   public void visit(Titel t) {
      summe += t.getBewertung(); anz++;
   }
   public void visit(Playlist p) {
       for (Knoten k: p.getKnoten()) {
          k.accept(this);
       }
   }
   public void visit(Knoten k) {
      //... wird das jemals aufgerufen? ...
      EH.Assert(false,...,"Unbehandelter Kr
   }
   public double getBewertung() {
      if (anz>0) {return ((double)summe)/ar
      else {return 0.0;}
   }
```

Besucher entscheidet wie zum nächsten Datenelement gesprungen wird

→ Jeder Besucher muss die besuchte Datenstruktur genau kennen

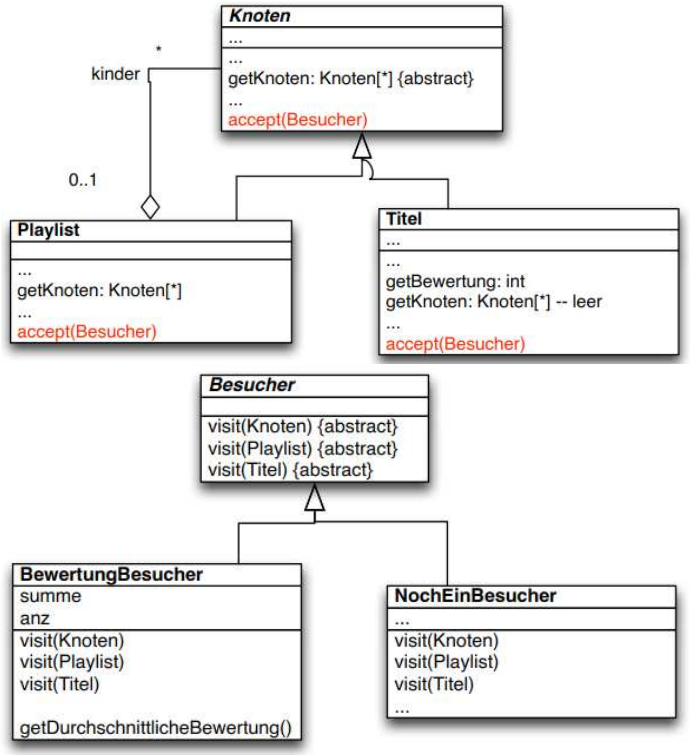
→ **enge Kopplung**

→ Was ist, wenn Datenstruktur komplexer ist / wird? (z.B. auch zyklische Abh.)

HINWEIS: BESUCHER & LOSE KOPPLUNG

- Andere Variante:

  → Besuchter lenkt Besucher



Zum Einstieg beim root

```
public class Playlist ...{
    public void starteBesuch(Besucher bes){
        bes.visit(this);
    }
    public void accept(Besucher bes){
        for (Knoten k: p.getKnoten()) {
            if(k instanceof Titel){bes.visit((Titel)k);}
            else {bes.visit((Playlist)k);}
        }
    }
}
```

Die besuchte Struktur managt den Abstieg in die Unterstrukturen

```
public class BewertungBesucher extends Besucher {
  ...
  public void visit(Titel t) {...}
  public void visit(Playlist p) {p.accept(this); }
  public void visit(Knoten k) { ... }
}
```

WEITERE GRASP-MUSTER*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Controller
  → Vgl. Controller bei MVC-Architekturmuster
- High Cohesion (Hohe Kohäsion)
- Polymorphismus (kennen wir)
  – Vgl. Auch Strategy-Pattern der GoF
- Pure Fabrication
  – Reine Erfindung eines Elements
  – Z.B. Method-Objekt-Pattern
- Indirection
  → Vgl. Controller bei MVC-Architekturmuster
- Protected Variations
  – Z.B. Interfaces als Entkopplungsmechanismus für Variationspunkte

*Vgl. Applying UML with Patterns; Kap. 16

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

06
Fazit

Ziel:
Was haben wir damit gewonnen?

WAS HABEN WIR GELERNT?

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Der Mustergedanke
  - Kondensiertes Expertenwissen
  - Leicht für Anfänger anwendbar
  - Gut zur Kommunikation zw. Experten geeignet
- Es gibt verschiedene Muster
  - Anforderungs-, Analyse-, Architektur-, Entwurfs-, …
- Entwurfsmuster
  - Kompositum, Besucher, Beobachter, …
- Architekturmuster
  - 3-Schichten, MVC, …
- Allgemeine Prinzipien
  - GRASP

WEITERFÜHRENDE LITERATUR

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Freeman et al: Entwurfsmuster von Kopf bis Fuß.
  - Schöne, umfassende Einführung zu Entwurfsmustern
  - Witzig geschrieben
- Starke: Effektive Software-Architekturen
  - Überblick zu Architektur- und Entwurfsmustern, Entwurfsprinzipien.
- Gamma et al: Entwurfsmuster.
  - Klassiker zu Entwurfsmustern (auch „Gang of Four“ genannt)
- Buschmann et al: Pattern-orientierte Software-Architektur. Bd. 1.
  - Klassiker zu Entwurfs- **und v.a. Architekturmustern**
- C. Larman: Applying UML and Patterns [30 BF 500 78].
  - GRASP (Entwurfsprinzipien) ausführlich

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

AUF GEHT‘S!!

SELBER MACHEN UND LERNEN!!