Einführung in Software Engineering

Barbara Paech, Marcus Seiler

Institute of Computer Science
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
paech@informatik.uni-heidelberg.de

Kapitel 4: SWE im Großen – Kommunikation der EntwicklerInnen

4. Kommunikation der EntwicklerInnen (2.Teil)

- 4.1. Einführung Modellierung
- 4.2. Klassendiagramme
- 4.3. Interaktionsdiagramme
- 4.4. Zustandsdiagramme
- 4.5. Klassenentwurf mit OOAD
- 4.6. Kommunikation von Erfahrungswissen (Entwurfsmuster)
- 4.7. Kommunikation von Entscheidungen (Rationale)
- 4.8. Zusammenfassung Modellierungstechniken

4.5.Klassenentwurf mit objekt-orientierter Analyse und Entwurf

- Einführung OOAD
- OOAD: Analyseklassendiagramm
- OOAD: Entwurfsklassendiagramm

Entwurf im Software-Prozess

- In Analyse und Spezifikation werden die Anforderungen ermittelt und dargestellt.
- In der Codierung wird das Zielsystem im Detail realisiert.
- Dazwischen liegt der Entwurf.
- Im Entwurf wird die Architektur festgelegt, also die Struktur.

- Je nach Art der zu entwickelnden Software und je nach der gewählten Entwicklungsstrategie spielen beim Entwurf ganz unterschiedliche Überlegungen eine Rolle.
- Darum ist es unmöglich, eine einzige Strategie zu präsentieren, die immer angemessen ist und immer zum Erfolg führt.

Ziele des Entwurfs

- Wer eine große, komplexe Software realisieren soll, kann zunächst die vielen Facetten des Systems und ihre Wechselwirkungen nicht überblicken => Software untergliedern.
- Wenn damit überschaubare Einheiten entstanden sind => zweckmäßige Lösungsstrukturen festlegen und Software organisieren, um Überblick zu behalten

- Mit dem Entwurf verfolgen wir also drei Ziele, die sich nicht scharf gegeneinander abgrenzen lassen:
  - Gliederung des Systems in überschaubare (handhabbare) Einheiten
  - Festlegung der Lösungsstruktur
  - Hierarchische Gliederung

software engineering heidelberg

Gliederung in überschaubare Einheiten

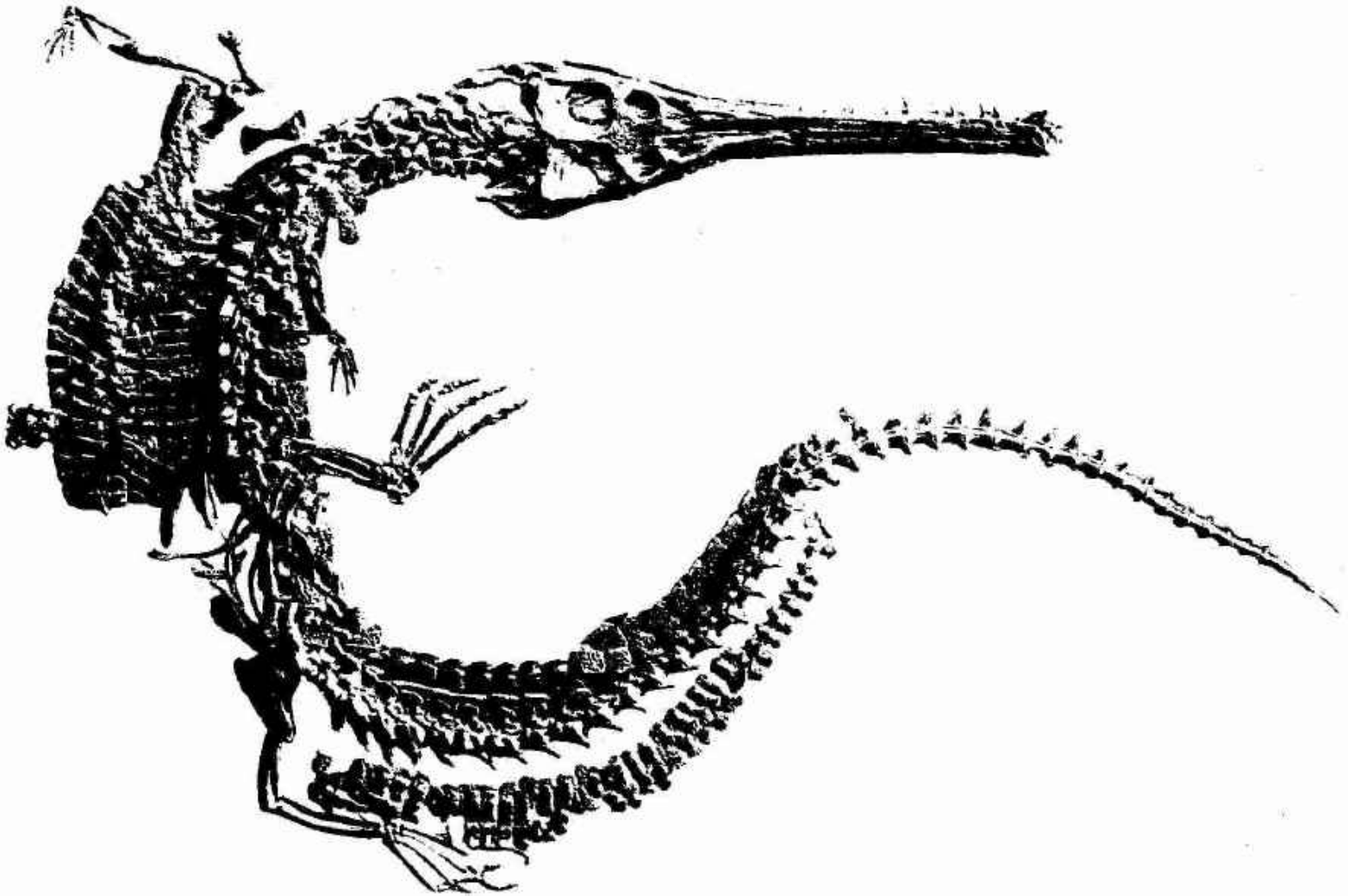
- Vergleich: Ein großes Bauwerk soll an einen anderen Standort versetzt werden.
  - Dazu muss man das Bauwerk zumindest soweit zerlegen, dass die einzelnen Teile handhabbar werden.
  - Die Konkretisierung hängt vom Bauwerk, aber auch von der Ausrüstung und von den Fähigkeiten und Erfahrungen ab.
- Spezifikation liegt also zu Beginn der Entwicklung nicht vollständig vor und ist auch vor dem Entwurf nicht vollständig verstanden
- Oft verhilft erst der Entwurf zum notwendigen Verständnis.
  - Verwendung von Standardstrukturen sinnvoll!
- Wiederverwendung vorhandener Komponenten hat eine ähnliche Wirkung wie die Verwendung einer Standardstruktur.

Festlegung der Lösungsstruktur

- Struktur eines Gegenstands = Menge der Beziehungen zwischen seinen Teilen
  - Gegenstand, der nicht aus (erkennbaren) Teilen aufgebaut ist, heißt unstrukturiert oder amorph.
- Die Struktur bleibt nach Abstraktion vom Material und von den quantitativen Aspekten des Gegenstands übrig.
- Strukturen werden selbststabilisierend und damit sehr lange haltbar, oft viel länger als die Materialien, aus denen sie geformt werde => wirkt sehr weit in die Zukunft!
- Beispiele: Verkehrsplanung, Organisationen, Versteinerungen
- Die Struktur hat großen Einfluss auf die Brauchbarkeit (Effizienz) und (vor allem) auf die Wartbarkeit. → Die Wahl der Struktur ist die wichtigste (technische) Entscheidung der Software-Entwicklung.

[Ludewig,Lichter 2013]

Versteinerung aus Holzmaden



software engineering heidelberg

Hierarchische Gliederung

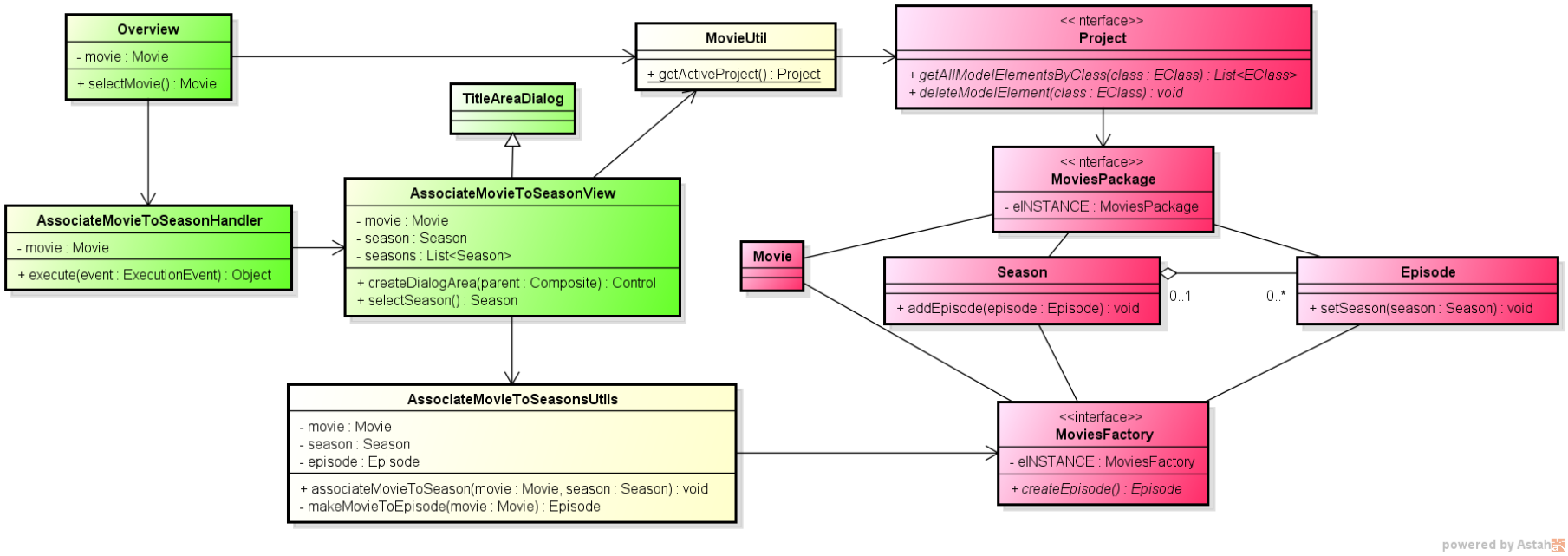
- Miller (1956): Ein Mensch kann nur Systeme überblicken, die aus wenigen, typisch etwa sieben Teilen bestehen.
- Sind es mehr, so muss man die Gegenstände nacheinander betrachten und ihren Zusammenhang systematisch entwickeln.
- Programme (Systeme) bestehen typisch aus einigen bis vielen tausend Deklarationen und Anweisungen. Sie müssen hierarchisch organisiert werden, um für uns überschaubar zu sein.
- Nur kleine Programme kann man direkt nach den Anforderungen codieren; alle anderen müssen schrittweise entwickelt werden.
- Durch die sinnvolle Gliederung (und nur durch sie) entsteht die Möglichkeit zur Abstraktion.

Einführung OOAD

- OOAD = Objektorientierte Analyse und Design
  - Wurde in den 90er Jahren entwickelt

- dient zur systematischen Erstellung des Entwurfsmodells
  - Welche Entscheidungen sind auf dem Weg von den Anforderungen zu einem Klassendiagramm zu treffen?
  - Welche Klassen mit welchen Attributen und Operationen werden benötigt, um alles Wichtige für die Implementierung festzulegen?

Beispiel für Entwurfsmodell mit EMF



Vorgaben aus den Anforderungen

- Arbeitsbereich
  - zeigt grob, welche Daten auf der Benutzungsschnittstelle sichtbar sein sollen und welche Systemfunktionen angeboten werden
- Interaktionsdatendiagramm
  - zeigt die Daten im Detail
- Systemfunktionsbeschreibung
  - Zeigt, was genau zu berechnen ist
- Sicht
  - Zeigt, wie Daten und Funktionen den NutzerInnen angeboten werden sollen

Entscheidungen bei der Umsetzung

- Klassen für Benutzungsoberfläche (Darstellung, Kommunikation zu den Geräten)
  - Setzen die Sichten (und die Navigation aus der UI-Struktur) um
  - Typischerweise durch GUI-Framework unterstützt
- Klassen für Datenverwaltung
  - Setzen das Interaktionsdatendiagramm um
  - Welche Klasse für welche Daten zuständig? Kann man die Entitäten direkt als Klassen übernehmen?
  - Welche Operationen nötig?
- Klassen für interne Verarbeitung
  - Setzen die Systemfunktionen um
  - Welche Teile der Systemfunktionen werden durch Operationen in welchen Klassen berechnet?

Entwurfsprinzip Hohe Kohäsion

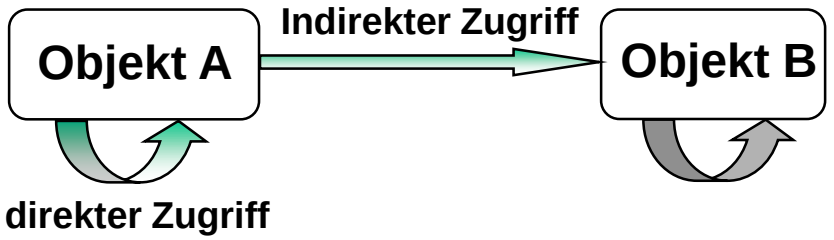
- Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente
- Hohe Kohäsion: starke Abhängigkeit zwischen den Elementen einer Komponente
  - Es gibt keine Zerlegung in Untergruppen von zusammengehörigen Elementen
    - z.B. Menge der Attribute einer Klasse nicht so zerlegbar, dass ein Teil der Operationen nur auf einem Teil der Attribute arbeitet
- Mechanismen zu Erreichung von Kohäsion:
  - Früher: ähnliche Funktionalitäten zusammenfassen
    - Schlecht, wenn Daten verstreut
  - Prinzipien der Objektorientierung (Datenkapselung)
    - Operationen der Klasse zuordnen, auf deren Attribute sie zugreifen
  - Verwendung geeigneter Muster zu Kopplung und Entkopplung

Entwurfsprinzip Niedrige Kopplung

- Kopplung ist ein Maß für die Abhängigkeiten zwischen Komponenten.
- Niedrige Kopplung: geringe Abhängigkeiten zwischen Komponenten
- Mechanismen zur Reduktion der Kopplung:
  - **Schnittstellenkopplung**, d.h. Austausch nur über Schnittstellen
    - z.B. get/set-Operationen statt Attributzugriff (also kein direkter Zugriff auf Attribute von außen)
  - Möglichst wenig Aufrufe zwischen den Komponenten
- Datenkopplung, d.h. gemeinsame Daten von Komponenten vermeiden!
- Strukturkopplung, d.h. gemeinsame Strukturanteile vermeiden !
  - z.B. keine Vererbung über Komponentengrenzen hinweg

Unterscheidung direkter und indirekter Zugriff

- Ein Objekt kann Attributwerte nur durch den Aufruf von Methoden lesen oder verändern.
- Es sind zwei Arten von Attributzugriffen durch Methoden zu unterscheiden:
  - Die Methode eines Objekts greift auf dessen eigene Attribute zu (**direkter Zugriff**).
  - Die Methode eines Objekts greift auf die Attributwerte anderer Objekte durch Aufruf von Methoden dieser Objekte zu (**indirekter Zugriff**).



Definition Grund- und komplexe Operation

- Die Unterscheidung von direktem und indirektem Zugriff ermöglicht die Unterscheidung von Grundoperationen und komplexen Operationen.
  - Eine **Grundoperation** liest und verändert Attribute nur direkt.
  - Eine **komplexe Operation** liest und verändert Attribute auch indirekt.

- Komplexe Operationen erhöhen Kopplung (das ist nicht zu verhindern)
- Komplexe Operationen gefährden Kohäsion (falls falsch zugeordnet) =>
  - Teile komplexe Operation möglichst in Unteroperationen auf, die zusammenhängende Änderungen an einer Klasse machen
  - Ordne komplexe Operation geeignet einer Klasse zu

Grundidee OOAD

- Entwurf beinhaltet viele Detailentscheidungen
- Idee von *Objekt-oriented Analysis and Design (OOAD):* Erstelle den Entwurf in 2 Schritten: Analyseklassenmodell und Entwurfsklassenmodell
- Analyseklassenmodell definiert die Klassenstruktur auf Basis der Anforderungen
  - Modellierung des Systemverhaltens auf abstraktem Niveau
  - Allgemeine *schematische* Beschreibung des Verhaltens
  - Interne Struktur des Systems ("von innen")
- Entwurfsklassenmodell berücksichtigt vorhandene Klassen (Bibliothek, Frameworks) und optimiert im Hinblick auf die Entwurfsziele (insbes. nicht-funktionale Anforderungen)

Entwurfs- vs. Analysemodell

Analyse-Modell	Entwurfs-Modell
Objekte: Fachgegenstände	Objekte: Softwareeinheiten
Klassen: Fachbegriffe	Klassen: Schemata
Vererbung: Begriffsstruktur	Vererbung: Programmableitung
Annahme perfekter Technologie	Erfüllung konkreter Implementierungsziele
Funktionale Essenz	Gesamtstruktur des Systems
Meist projektspezifisch	Ähnlichkeiten zwischen verwandten Projekten
Grobe Strukturskizze	Genaue Strukturdefinition
	Mehr Struktur & mehr Details

4.5.Klassenentwurf mit objekt-orientierter Analyse und Entwurf

- Einführung OOAD
- OOAD: Analyseklassendiagramm
- OOAD: Entwurfsklassendiagramm

Erstellung eines Analyseklassendiagramms

- Im folgenden Erklärung der einzelnen Entscheidungen anhand von Regeln.
- Die Teilschritte 2-4 können in beliebiger Reihenfolge oder auch parallel durchgeführt werden.
  - **Teilschritt 1:** Klassen, Attribute und Assoziationen bestimmen
    Klassenkandidaten aus den Anforderungen identifizieren, Definition passender Namen und Attribute
  - **Teilschritt 2:** Operationen der Klassen bestimmen
  - Systemfunktionen in Form von Operationen auf die Klassen verteilen, Kohäsion und Kopplung berücksichtigen
  - **Teilschritt 3:** Vererbung nutzen und komplexe Assoziationen (wie z.B. Aggregationen) beschreiben
  - **Teilschritt 4:** Klassendiagramm konsolidieren
    Insbes. Dialogklassen evtl. auflösen, Assoziationen konsolidieren.
- Erste Überlegungen sind mit Icons beschreibbar, für Attribute und Operationen sind Klassendiagramme nötig.

Stereotypen von Analyseklassen

UML-Icon:
(Stereotypen)

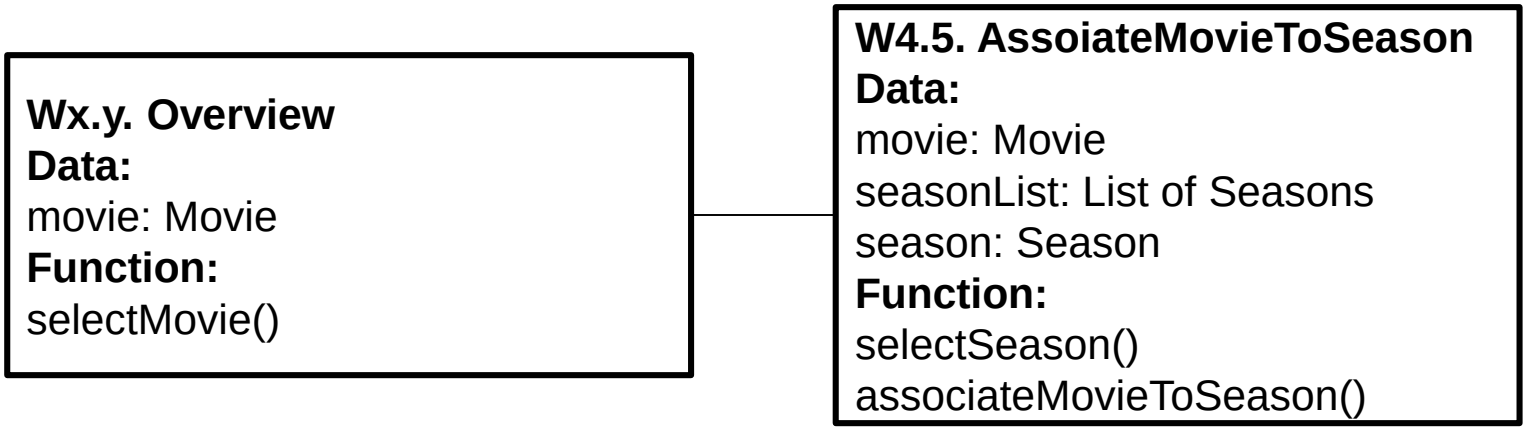
- Verschiedene Stereotypen von Klassenrollen
- Entitäts-Klasse (entity class):
  Beschreibung von Gegenständen mit dauerhafter Existenz

  - Entitätsklassen (siehe Interaktionsdatendiagramm) spiegeln Dinge oder Sachverhalte aus dem Anwendungsbereich wieder (z.B. „StudentIn“ oder „Lehrveranstaltung“). Typischerweise sind das:
    - Informationen, die verwaltet oder ausgetauscht werden
    - Materialien, die ausgetauscht werden
- Steuerungs-Klasse (control class):
  Beschreibung von Vorgängen und Reihenfolgen

  - Semantische Systemfunktionen werden im Analyseklassendiagramm zuerst durch Steuerungsklassen modelliert. Dies sind Platzhalter für die Modellierung komplexer Operationen im Teilschritt 3
- Dialog-Klasse (boundary class):
  Erwähnung von Nutzerinteraktionen und deren Inhalt

  - Dialogklassen (siehe Arbeitsbereiche) bündeln Daten und Operationen, die auf der Benutzungsschnittstelle verfügbar sind.

Beispiel: Analyseklasse für AssociateMovieToSeason



- Arbeitsbereich *W4.5. AssociateMovieToSeason*
- Systemfunktion *AssociateMovieToSeason():* erstellt aus dem Film eine Episode, verlinkt diese mit der Staffel und löscht den Film

Dialog-Klasse (boundary class)

Wx.y. W4.5.

Steuerungs-Klasse (control class)

associateMovieToSeason()

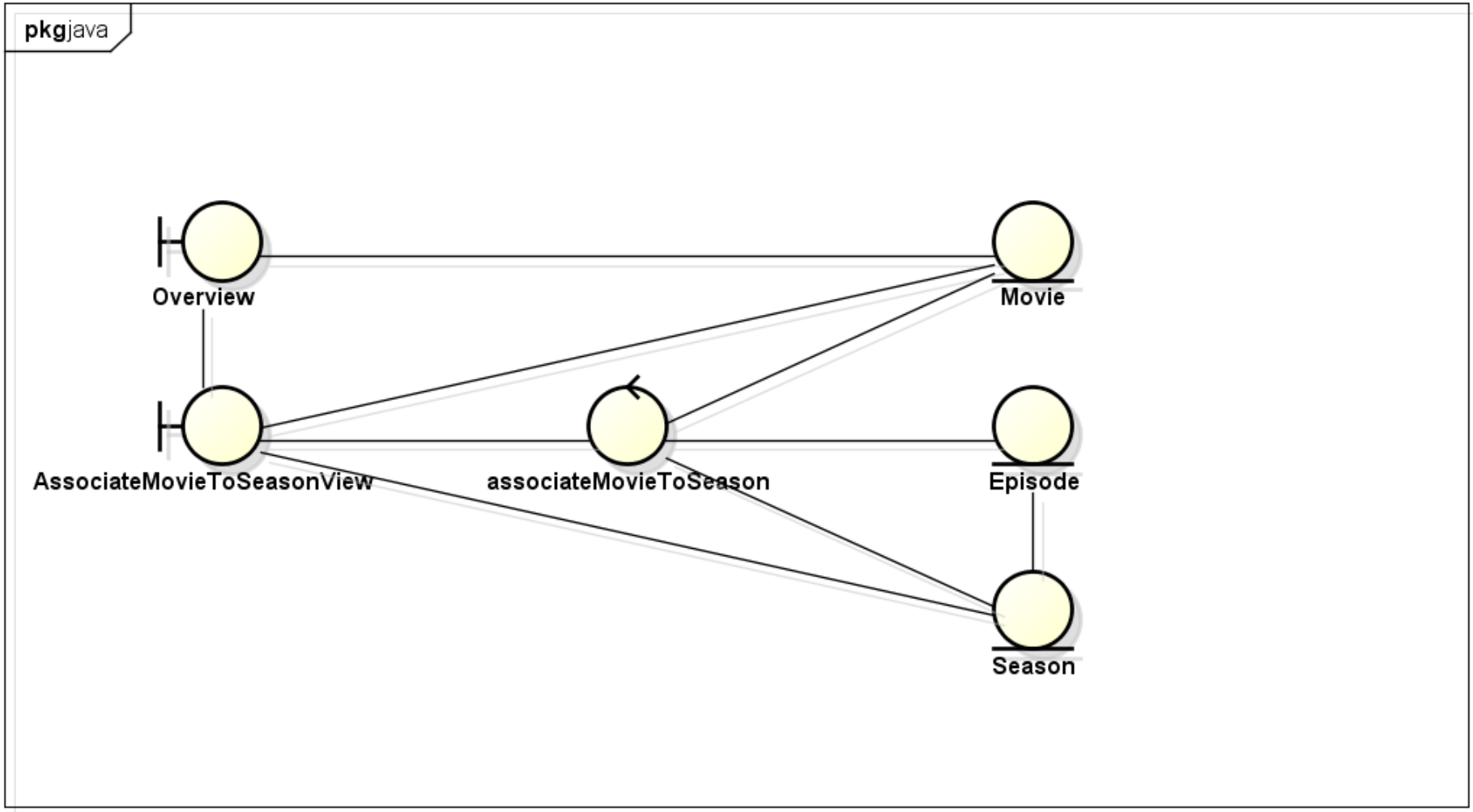
Entitäts-Klasse (entity class)

Movie Season Episode

Teilschritt 1: Regeln für Klassen und Assoziationen

- **Regel 1: Entitätsklassen**
  - Die Entitäten aus dem Interaktionsdatendiagramm werden Entitätsklassen.
  - Assoziationen aus dem Interaktionsdatendiagramm werden Assoziationen im Analyseklassendiagramm.
- **Regel 2: Steuerungsklassen**
  - Semantische Systemfunktionen werden zuerst durch Steuerungsklassen modelliert.
  - Steuerungsklassen werden mit den Entitätsklassen verbunden, deren Attribute sie lesen oder schreiben, sowie ggf. mit weiteren Steuerungsklassen
- **Regel 3: Dialogklassen**
  - Arbeitsbereiche (oder Sichten) werden Dialogklassen.
  - Dialogklassen werden mit den Steuerungsklassen der Systemfunktionen und den Entitätsklassen verbunden, die in dem Arbeitsbereich angeboten werden.
  - Navigationen zwischen Arbeitsbereichen werden zu Assoziationen zwischen Dialogklassen

Beispiel Teilschritt 1:

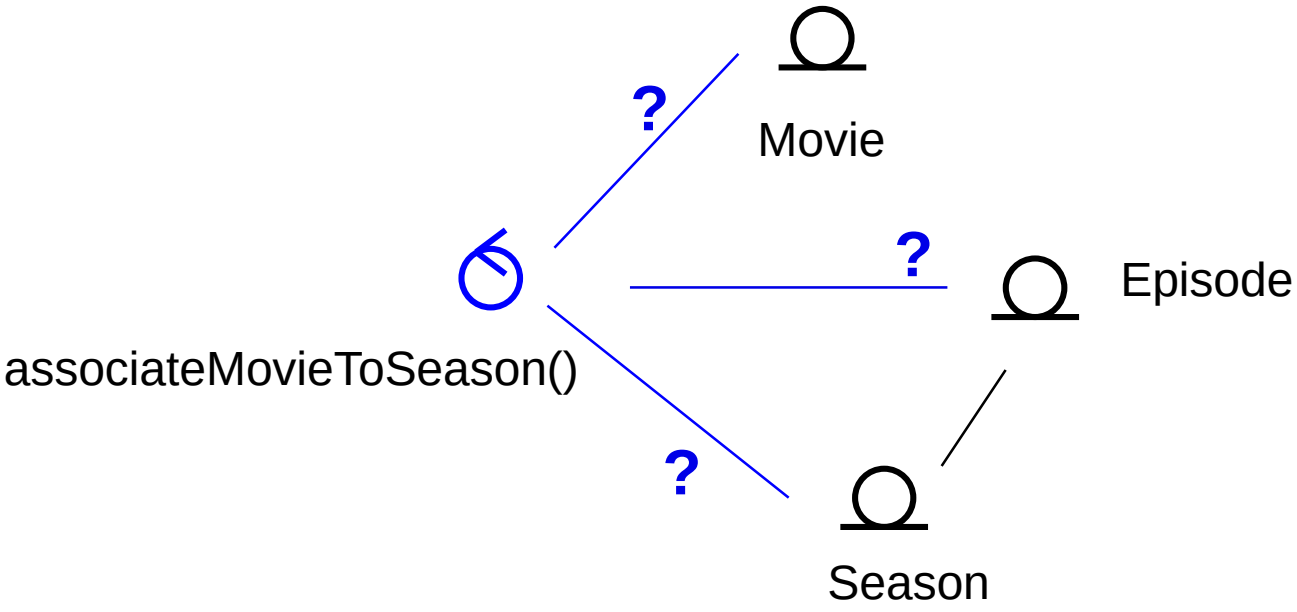


Teilschritt 2: Regeln für Grundoperationen

- **Regel: Verben deuten auf Grundoperationen hin**
  - Grundoperationen spiegeln Überprüfungen oder Berechnungen von und mit Attributen eines Objekts wider. Oft sind diese Überprüfungen oder Berechnungen bzgl. Attributen mit entsprechenden Verben im Anforderungstext genannt. Oft müssen sie aber auch ergänzt werden aus der Kenntnis der Attributbedeutung.

Grundproblem Teilschritt 2: Zuordnung der Operationen

- Eine Steuerungsklasse wird dadurch aufgelöst, dass sie als ein oder mehrere Operationen auf ein oder mehrere Klassen verteilt wird.
- Was ist die richtige Zuordnung?

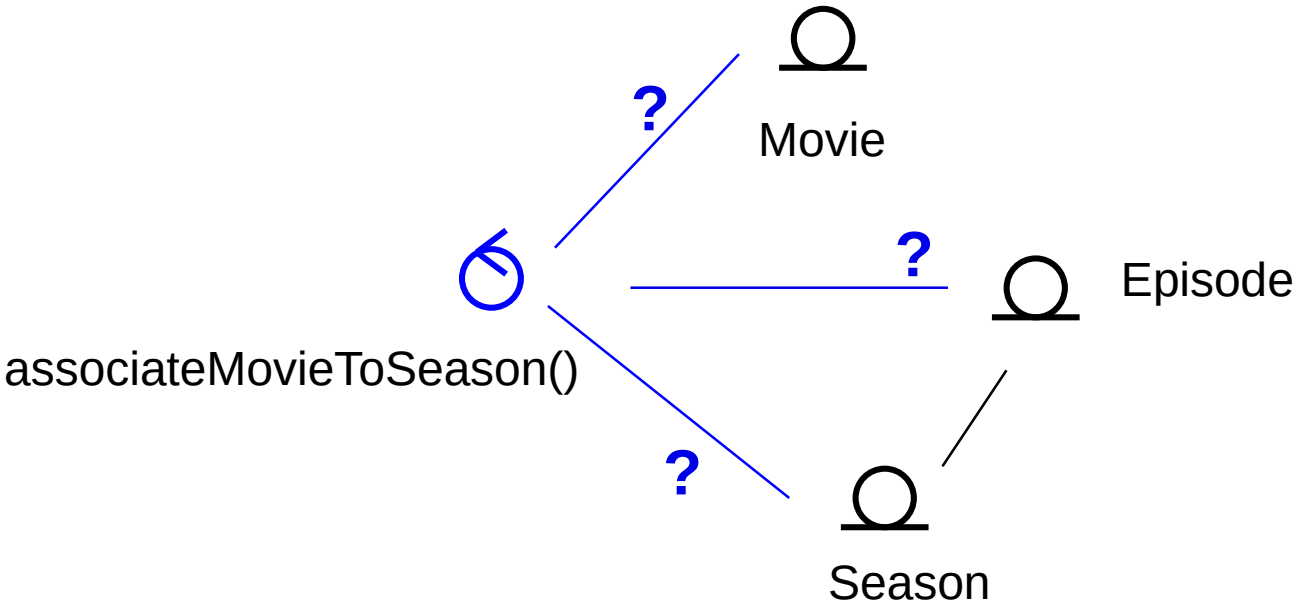


Teilschritt 2: Regeln für komplexe Operationen

- **Regel 1: Verteile Steuerungsklassen auf Entitäts- und Dialogklassen**
  - **Kanonische Lösung**, falls Steuerungsklasse nur auf Attributen (und Assoziationen) **einer** Klasse arbeitet (z.B. Filmdaten ergänzen)
  - **Einfache Lösung**, falls mehrere Klassen betroffen sind, aber als Eingabe nur Attribute einer Klasse (z.B. Löschen einer Serie) oder als Ausgabe nur Attribute einer Klasse verwendet werden
  - Keine „beste Lösung“, wenn Attribute mehrerer Klassen als Eingaben oder als Ausgaben verwendet werden

- **Regel 2: Fasse evtl. Operationen zusammen oder teile sie weiter auf**
  - Operationen können mehrere Systemfunktionen umfassen (Achtung: nur bei kanonischer Lösung)
  - Systemfunktionen evtl. noch in weitere Operationen aufspalten (falls zusammenhängende Änderung, die auch eigenständig sinnvoll ist)

Beispiel Teilschritt 2: Wie Verhalten zu Klassen zuordnen ?



Es gibt keine einfache Lösung, weil die Funktion

- Als Eingabe: Film und Staffel hat
- Als Ausgabe: sowohl eine Episode erzeugt als auch diese mit der Staffel verlinkt (also Episode und Staffel verändert) und Film löscht

=> Objektifizierung der Systemfunktion

Teilschritt 2: „Objektifizierung“ von Verhalten

- Manchmal ist es sinnvoll, Steuerungsklassen in der Implementierung beizubehalten
  - Vermeidung zu großer Klassen
  - Speicherbarkeit von Zwischenzuständen lang andauernder Vorgänge
  - Austauschbarkeit von alternativen Vorgängen durch eine Vererbungshierarchie von Steuervorgängen
  - Wiederverwendbarkeit derselben Steuerungsklasse für unterschiedliche Implementierungen

- Objektifizierung von Verhalten bringt
  - Flexibilität in der Implementierung
  - Entkopplung

- Aber: Organisatorischer Zusatzaufwand und sollte daher nur eingesetzt werden, wenn sinnvoll begründbar!

- Wird hier im Beispiel verwendet

Teilschritt 2: Prüfung durch Sequenzdiagramm

- Die Verteilung der Operationen sollte durch Erstellung von Sequenzdiagrammen für jede Systemfunktion überprüft werden:
  - Zu hohe Kopplung: Sind für die Umsetzung sehr viele Objekte und Methodenaufrufe notwendig?
  - Falls ja: kann die Anzahl durch Umverteilung von Operationen verringert werden?
  - Zu niedrige Kohäsion: Gibt es Operationen einer Klasse, die auf getrennten Attributbereichen arbeiten?
  - Falls ja: wird die Komplexität der Sequenzdiagramme durch eine Aufteilung der Klasse in 2 Klassen erhöht?

Teilschritt 3: Beispiel Klassendiagramm vervollständigen

- Vererbung, Aggregation

Teilschritt 4: Regeln für Umgang mit Dialogklassen

- Regel 1: Dialogklassen in eine separate Schicht legen (GUI-Schicht)
  - Insbesondere wenn nicht-triviale Arbeitsbereiche
- Regel 2: Andernfalls die Dialogklasse einer assoziierten Klasse zuordnen
  - Die Klasse ist dann sowohl für Datenverwaltung als auch Darstellung auf dem GUI zuständig

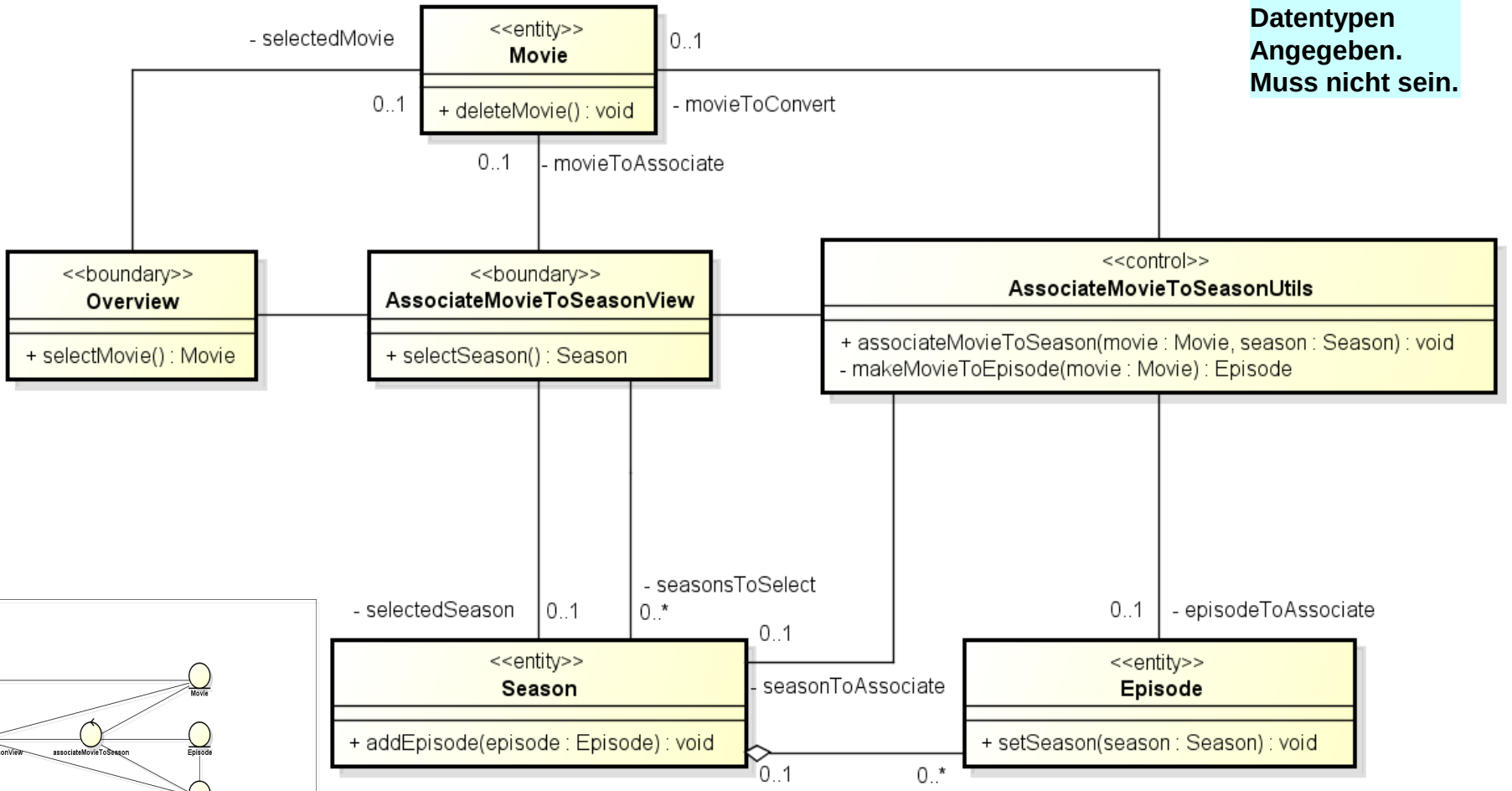
Teilschritt 4: Überprüfung der Assoziationen

- **Regel 1: Alle notwendigen Kommunikationswege abdecken**
  - Jede Klasse mit einer komplexen Operation Op muss eine Assoziation zu allen Klassen haben, deren (Grund-) Operationen bei der Ausführung von Op benötigt werden.

- **Regel 2: Keine unnötigen Kommunikationswege**
  - Es ist zu überprüfen, ob Assoziationen, die in keiner Systemfunktion als Kommunikationsweg genutzt werden, wirklich sinnvoll sind.

Beispiel: konsolidiertes Analyseklassendiagramm mit Operationen

Hier schon Datentypen Angegeben. Muss nicht sein.

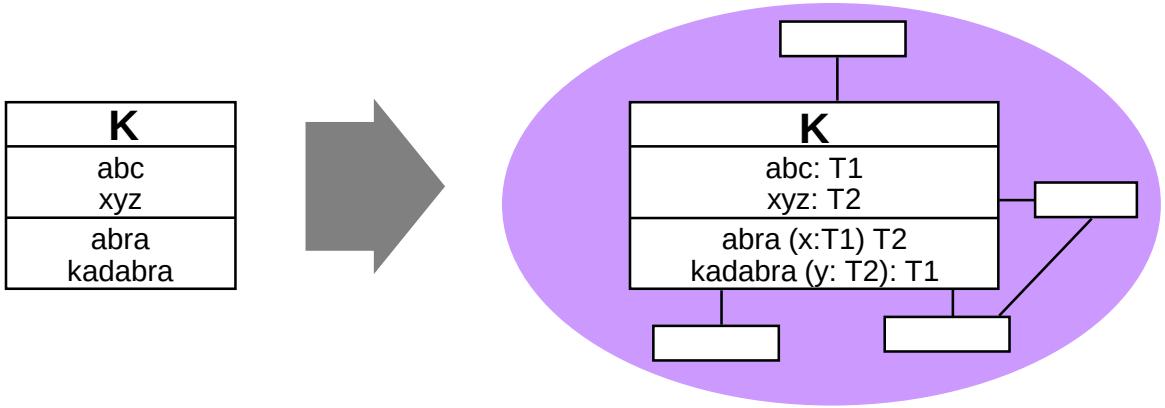


4.5.Klassenentwurf mit objekt-orientierter Analyse und Entwurf

- Einführung von OOAD
- OOAD: Analyseklassendiagramm
- OOAD: Entwurfsklassendiagramm

Erstellung von Entwurfsmodellen

- Ziel: Vorbereitung der Codierung durch Berücksichtigung der Entwurfsziele
- Verfeinerung der Analyseklassenmodelle durch Infrastrukturklassen und Vervollständigung



- Verfeinerung der Zustandsdiagramme (evtl. direkt in Code), siehe später

Entwurfsklasse

- Verfeinert die Analyseklassen durch
  - Detailspezifikation der Klassen: Vervollständigung bzgl. Operationen, Datentypen, Sichtbarkeit, Invarianten
  - Berücksichtigung von Infrastruktur-Klassen
  - Berücksichtigung von Entwurfszielen
  - Detailspezifikation der externen Schnittstellen
  - Aufteilung auf Komponenten: je ein Modell pro Komponente

- Verwendet oft Entwurfsmuster (siehe Kapitel 4.6.)

Wichtige Verfeinerungen

- Erweiterung des fachlichen Kerns: Mehr Details als im Analysemodell
  Nachfolgend typische Ergänzungen:
  - Teilschritt 1: Vollständige Listen der Attribute und Operationen
  - Teilschritt 2: Festlegung von Datentypen, Sichtbarkeit
  - Teilschritt 3: Spezifikation der Operationen (z.B. Vor- und Nachbedingungen)
  - Teilschritt 4: Definition von Ausnahmen
  - Teilschritt 5: Konkretisierung von Datenstrukturen, z.B. bei Assoziationen/ Aggregationen Festlegung von `ordered`
  - Teilschritt 6: Elimination von Mehrfachvererbung

- Zusätzliche Klassen/Pakete:
  - Einbindung in Infrastruktur, Altsysteme etc.
  - Anpassungs- und Entkopplungsschichten für gewählte Technologien (z.B. Datenzugriffsschicht, CORBA-Schnittstellen, XML-Anschluss...)

Teilschritt 3: Spezifikation von Operationen (1)

- Definition: Die Spezifikation einer Operation legt das Verhalten der Operation fest, ohne einen Algorithmus festzuschreiben.

- Grundidee: Design by Contract (Bertrand Meyer)
  - Klasse A ruft Operation Op der Klasse B auf
  - Vorbedingung von Op: muss von A eingehalten werden
  - Nachbedingung von Op: wird von B garantiert, solange bei Aufruf von Op die Vorbedingung V eingehalten wurde
  - Invariante von B: gilt vor und nach jeder Ausführung einer Operation von B

Teilschritt 3: Spezifikation von Operationen (2)

- Spezifikation oft nur textuell
- Wird aber auch in Programmiersprachen unterstützt
  - Kommentare verwenden, z.B. mithilfe von Javadoc
  - Verwendung von Zusicherungen (ASSERT in Java)
    - Kann Zustand überprüfen (Vorbedingung, Nachbedingung, Invariante)
    - Aber braucht dazu manchmal Hilfsvariablen, die den Vorgängerzustand beschreiben
    - Kann zur Laufzeitoptimierung automatisch weggelassen werden (Hilfsvariablen dann aber trotzdem angelegt)
    - Überprüfungen sollten auch nach außen verfügbar gemacht werden
    - Auch beim Testen hilfreich: Ideen für Testfälle, Unterstützung beim Debugging

Teilschritt 4: Definition von Ausnahmen (1)

- Fehlersituationen im Code werden oft durch besondere Ausgabewerte einer Operation angezeigt (Fehlermeldung).
- Dadurch wird aber Behandlung der Fehler beim Aufrufer nicht garantiert => Programmiersprachen erlauben Definition von Ausnahmen
- Beispiel JAVA:

```
try {
// Anweisungen, beinhalten Anzeige von Ausnahme A durch
   throw A}
catch(Typ A) {
// Anweisungen }
```

Kennzeichnet überwachten Bereich

beschreibt die Ausnahmehandlung

Teilschritt 4: Definition von Ausnahmen (2)

- Unterscheide verschiedene Arten von Ausnahmen
  - Ausnahmen in der Domäne (z.B. Geschäftsregeln nicht eingehalten)
  - Technische Fehler (z.B. Verbindung zur Datenbank funktioniert nicht, aufgerufenes Objekt nicht vorhanden)

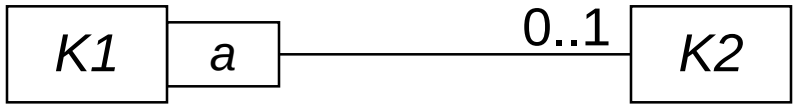
- Unterscheide verschiedene Arten der Ausnahmebehandlung
  - Direkte Behandlung durch besondere Operation
  - Weitergabe nach oben (d.h. umgebenden try-Block)
  - Nicht explizit behandelte Ausnahmen werden vom Laufzeitsystem abgefangen

Teilschritt 5: Konkretisierung von Datenstrukturen

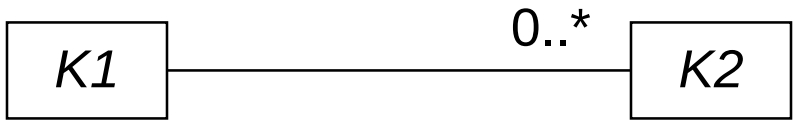
- **Konkretisierung, z.B. Verwendung von qualifizierter Assoziation**
- **Definition:** Eine **Qualifikation (Qualifier)** ist ein **Attribut a** für eine Assoziation zwischen Klassen K1 und K2, durch das die Menge der zu einem K1-Objekt assoziierten K2-Objekte *partitioniert* wird.

  Zweck der Qualifikation ist direkter Zugriff unter Vermeidung von Suche.

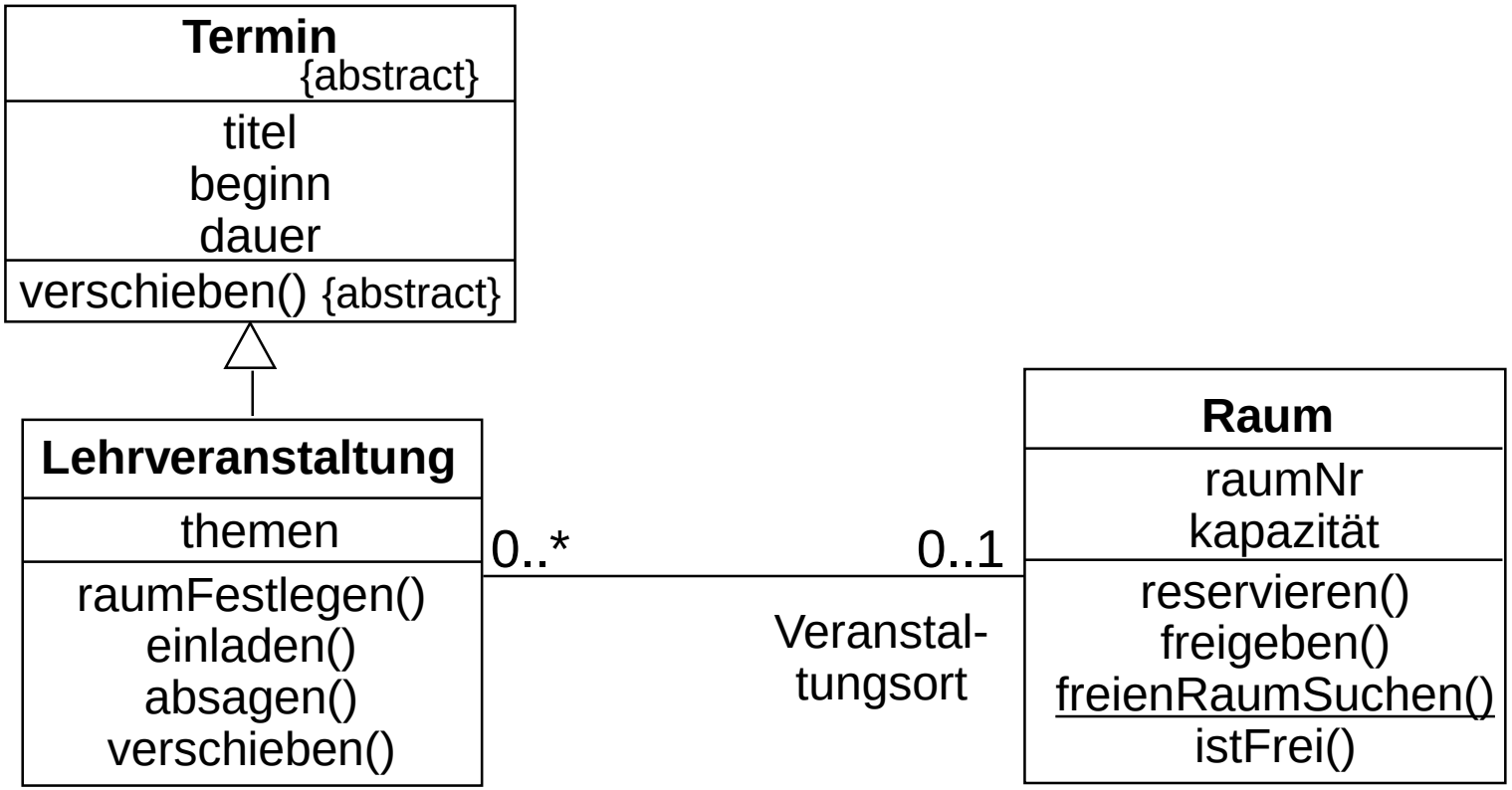
- **Notation:**



als Detaillierung von:



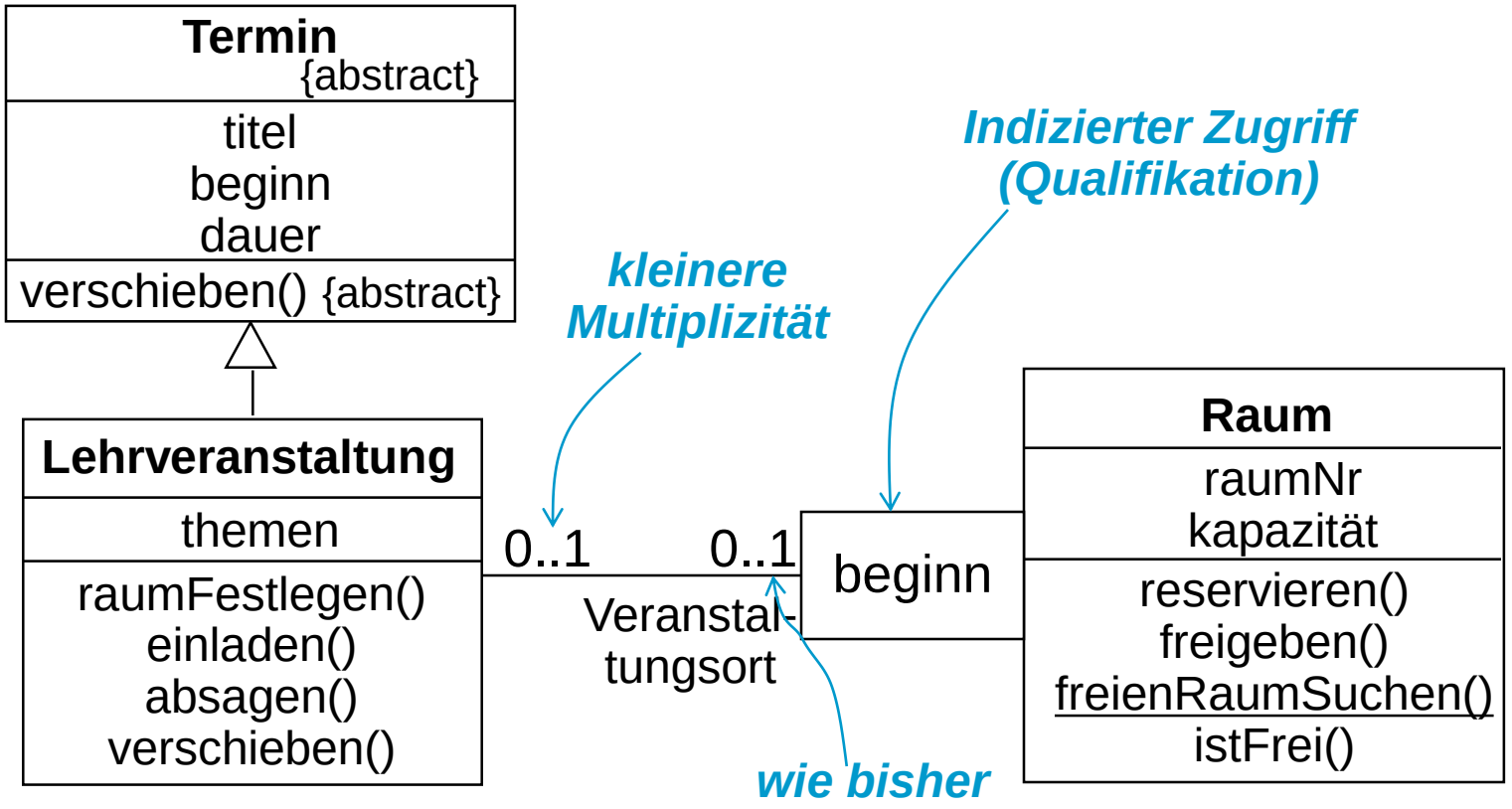
Qualifizierte Assoziation: Beispiel (1)



Raum12.istFrei(start=04.05.02 10:00, dauer=60min);

führt zu einer Suche über alle assoziierten Lehrveranstaltungen !

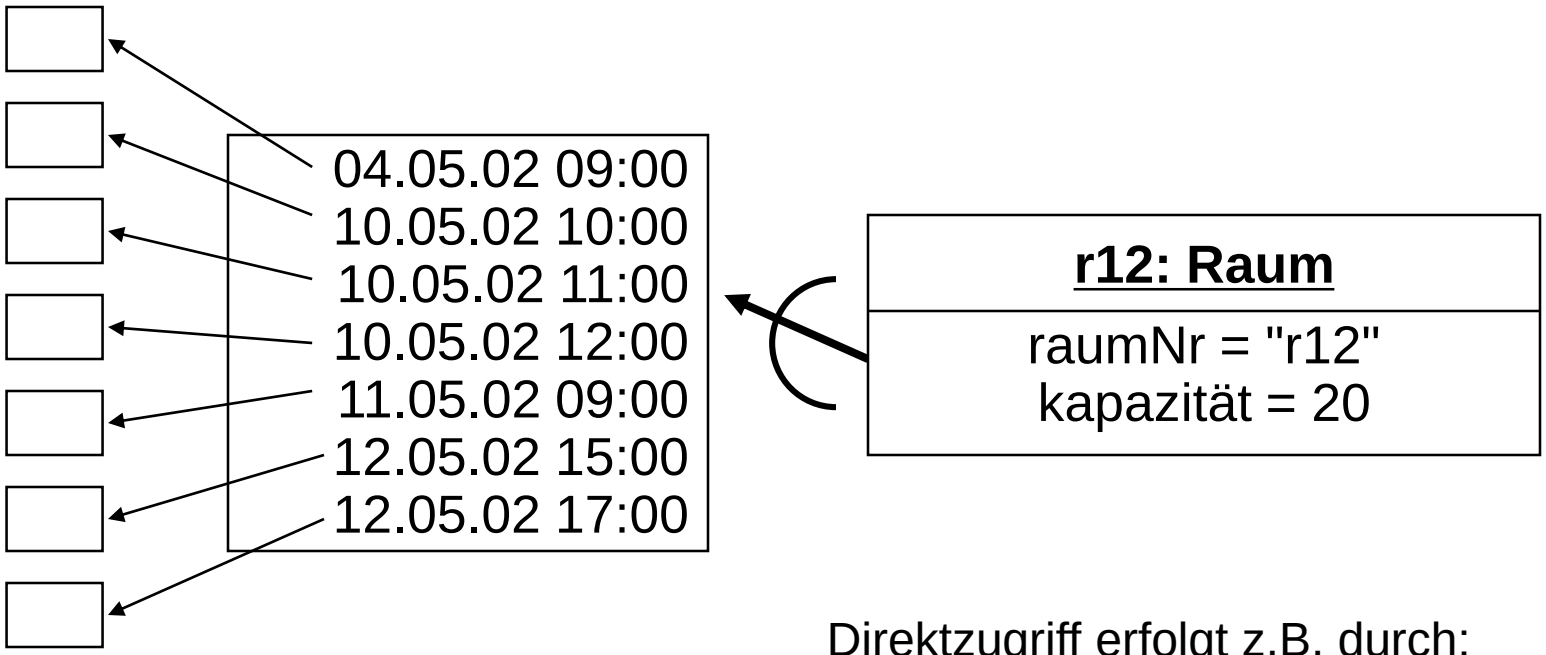
Qualifizierte Assoziation: Beispiel (2)



Raum12.istFrei(start=04.05.02 10:00, dauer=60min);

kann direkt nach Termin abfragen, ob eine Assoziation besteht

Realisierung einer qualifizierten Assoziation



Lehrveranstaltungs-Objekte

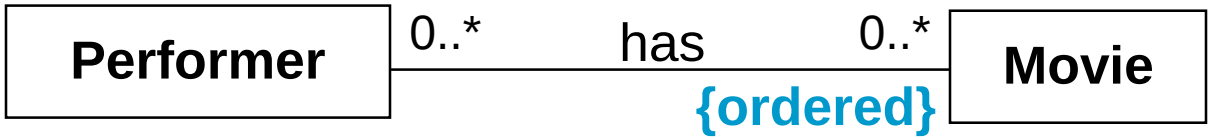
Direktzugriff erfolgt z.B. durch:

Hashfunktion
(Berechnung des Indexwerts aus gegebenem Termin)

Sortierte Baumstruktur

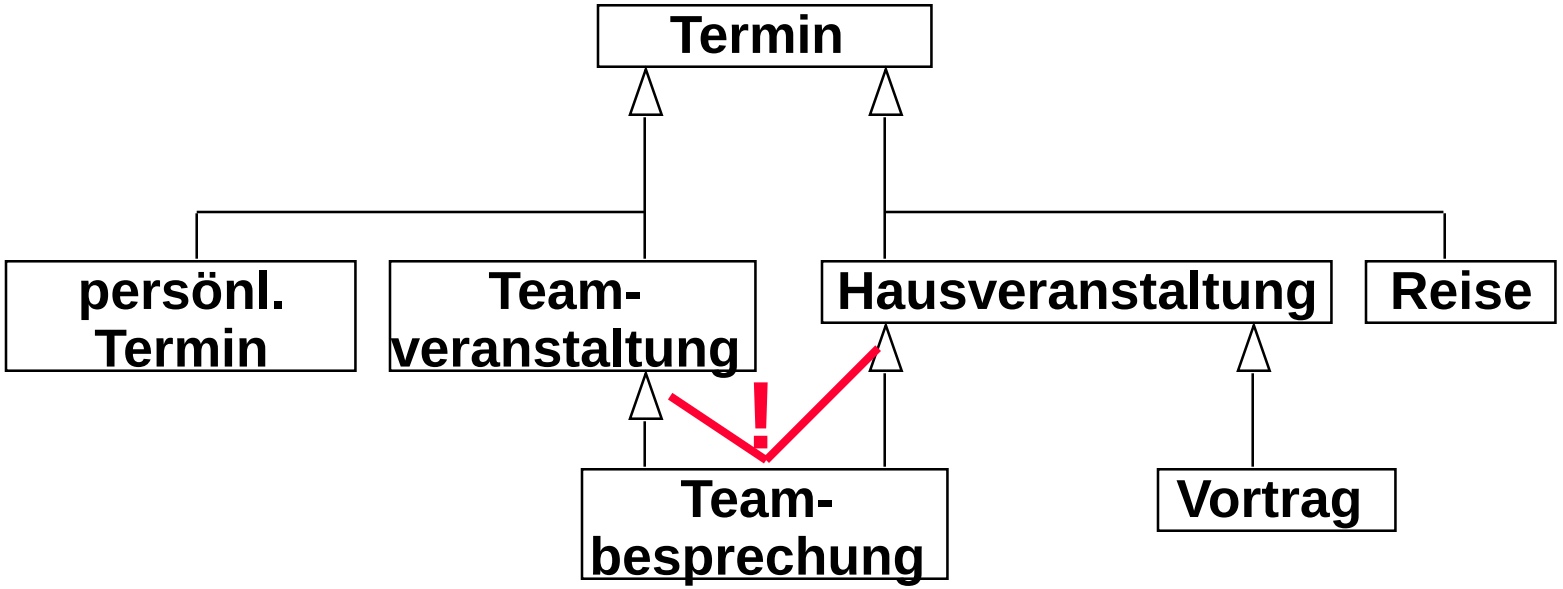
Teilschritt 5: Konkretisierung von Datenstrukturen

- Konkretisierung, Z.B. Festlegung einer Ordnung
  - Es besteht eine feste Reihenfolge, in der die assoziierten Objekte durchlaufen werden können → Oft ist Zugriff über Listen, Iteratoren möglich
  - Mehrfachvorkommen eines Objekts sind verboten



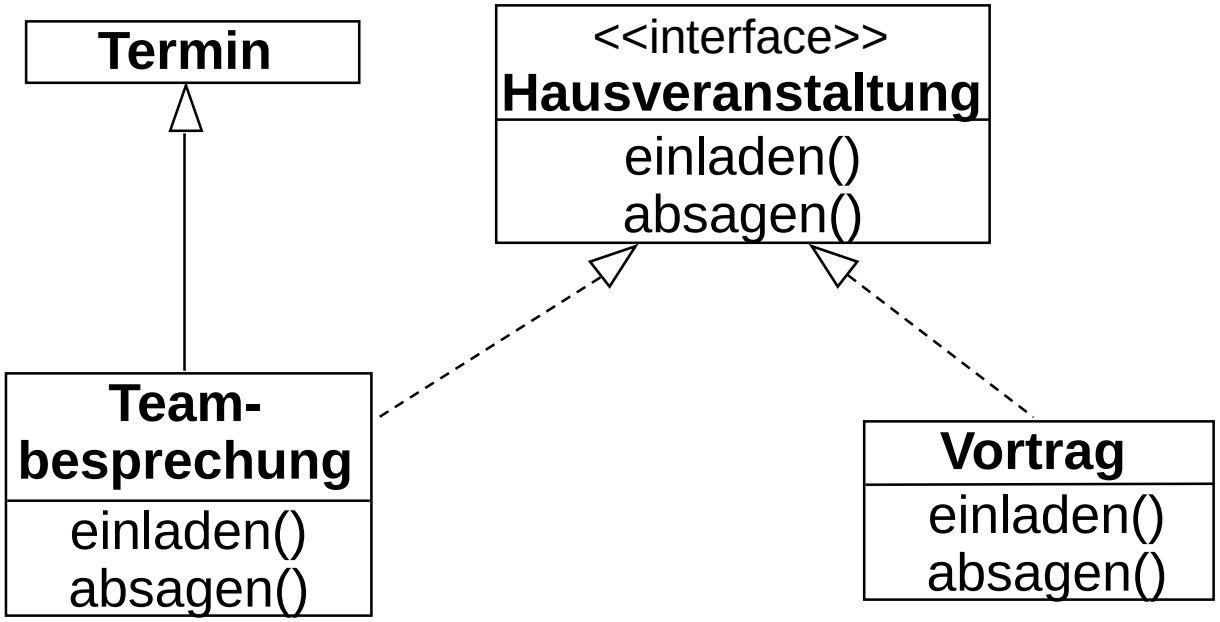
Teilschritt 6: Auflösung von Mehrfachvererbung

- In Analysemodellen treten oft unabhängige Dimensionen der Spezialisierung auf *(Mehrfachvererbung).*

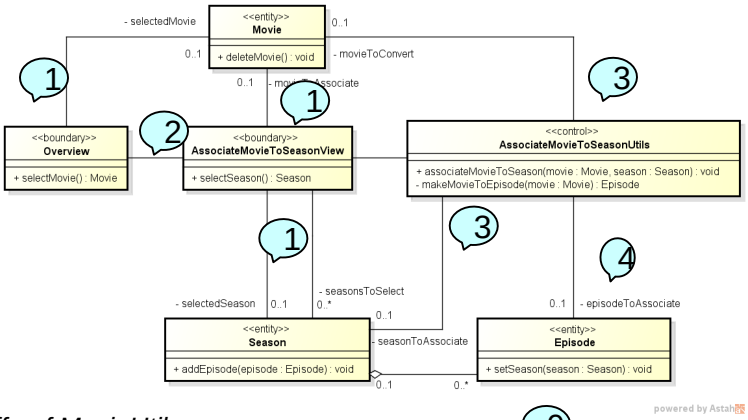


- In Entwurfsmodellen sollten solche Mehrfachvererbungen durch Verwendung von Schnittstellenklassen beseitigt werden

Beispiel: Einfache Vererbung durch Schnittstellen



Beispiel: Entwurfsdiagramm



Analyseklassendiagramm

1 *Zugriff auf MovieUtil, um Daten aus Project zu holen, Werden dann lokal gespeichert*

Verwendete Infrastrukturklasse 0

Verwendete Infrastrukturklasse Impl-Klasse nicht gezeigt 0

Verwendete Infrastrukturklasse 0

Verwendete Infrastrukturklasse, Impl-Klasse nicht gezeigt 0

Einführung einer Handlerklasse In EMF nötig 2

Kein direkter Zugriff Auf Entitäten Erhält Daten aus View 3

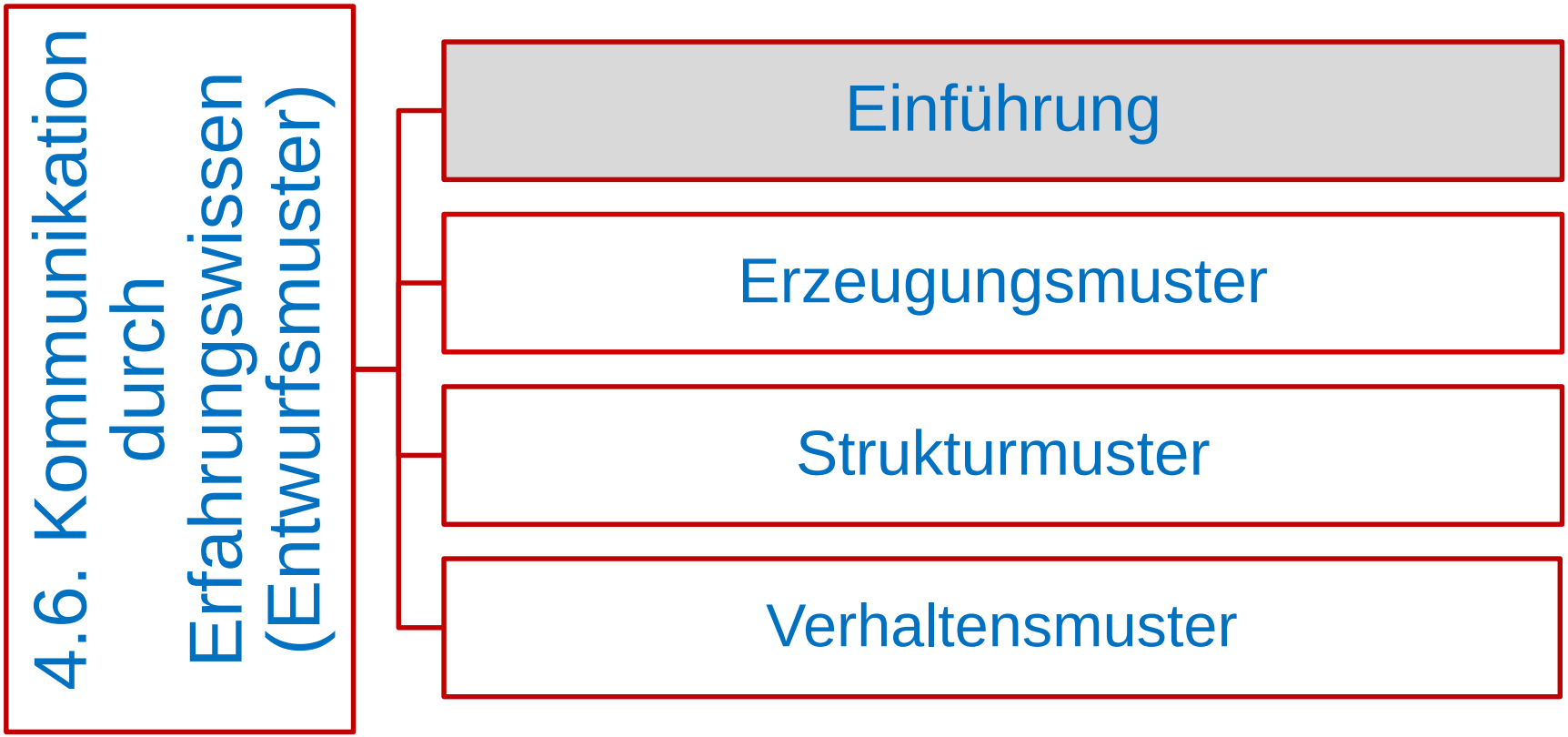
Zugriff auf MoviesFactory, um Episode zu erzeugen 4

Verwendete Infrastrukturklasse, Impl-Klasse nicht gezeigt 0

Zusammenfassung OOAD

- Analyseklassenmodell als **Zwischenschritt** ermöglicht Konzentration auf das Wesentliche:
  - Analyseklassenmodell: grobe Strukturentscheidungen
  - Entwurfsklassenmodell: Implementierungsvorgabe

- Die Schritte und Regeln geben erste Anhaltspunkte für eine systematische Definition der Klassen. Können auch als Checkliste nach dem Entwurf verwendet werden. Häufige Regelverletzung sollte überprüft werden.



Motivation Muster

- Wie kann man Erfahrungswissen weitergeben?
  - Beim Kochen: Rezept
  - In der Architektur: Architekturstile

- Erfahrungswissen für gute Klassenstruktur
  - Beispiele?
  - Grundidee: verwende Entwurfsmuster
  - Ausgehend von typischem Problem
  - Beschreibt Rollen von Klassen und ihre Abhängigkeiten (Terminologie, um Entwürfe zu beschreiben)

Definition Muster

- Muster = schematische Lösung für eine Menge verwandter Probleme
- Lassen sich nur verstehen, wenn praktisch angewendet

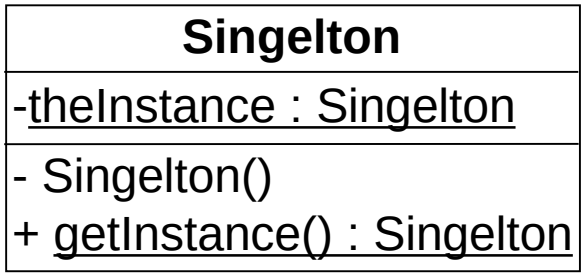
- Arten von Entwurfsmuster
  - Erzeugungsmuster
  - Strukturmuster
  - Verhaltensmuster

Beschreibung eines Musters

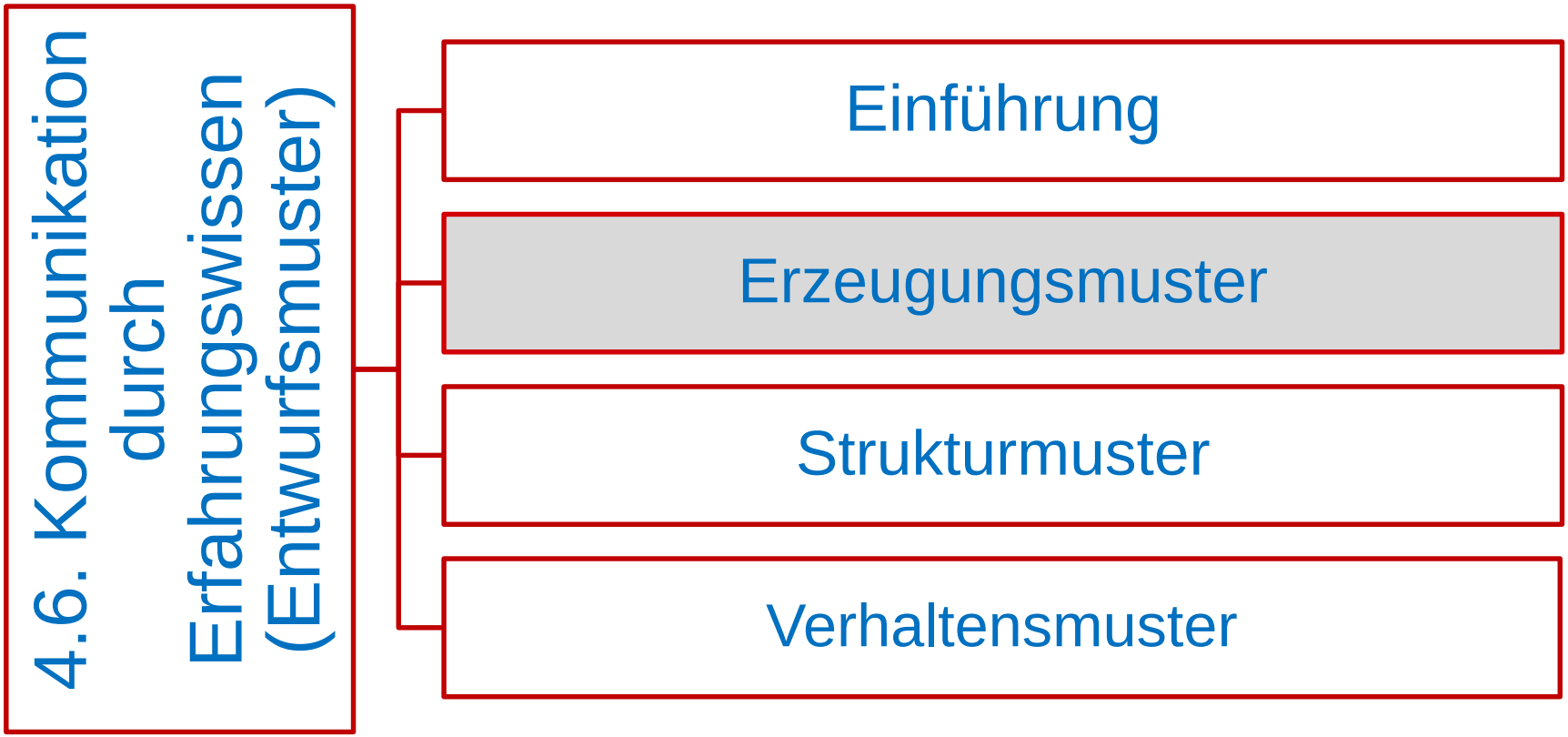
- Name
- Problem (Motivation, Anwendungsbereich, Problemklasse)
- Lösung
  - Struktur (Klassendiagramm)
  - Bestandteile (meist Klassen-, Assoziations- u. Operationsnamen)
  - Objektinteraktion (Abläufe, evtl. Sequenzdiagramme)
- Diskussion
  - Vor-/ Nachteile (beschreibt Begründung (Rationale): warum und wann gut)
  - Abhängigkeiten, Einschränkungen
  - Spezialfälle
  - Bekannte Verwendung (wie reif, wie oft schon eingesetzt)

Beispiel Einzelstück

- engl. *Singelton*
- **Erzeugungsmuster**
- **Problem**: Von einer Klasse darf es höchstens eine Instanz geben
- **Lösung**: Klasse nach Singelton – Schema aufbauen




```
class Singelton {
    private static Singelton theInstance;
    private Singelton();
    public static Singelton getInstance() {
        if (theInstance == null) {
            theInstance = new Singelton();
        }
        return theInstance;
    }
}
```

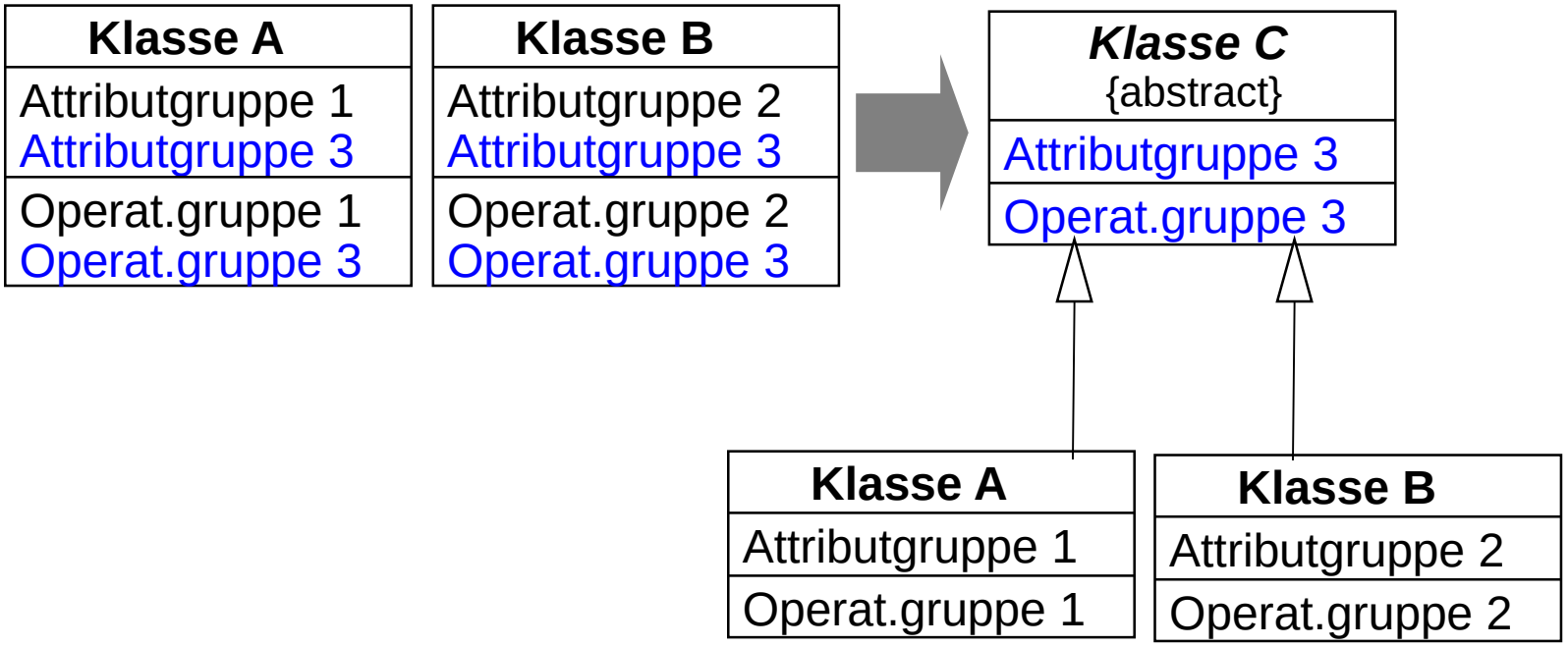


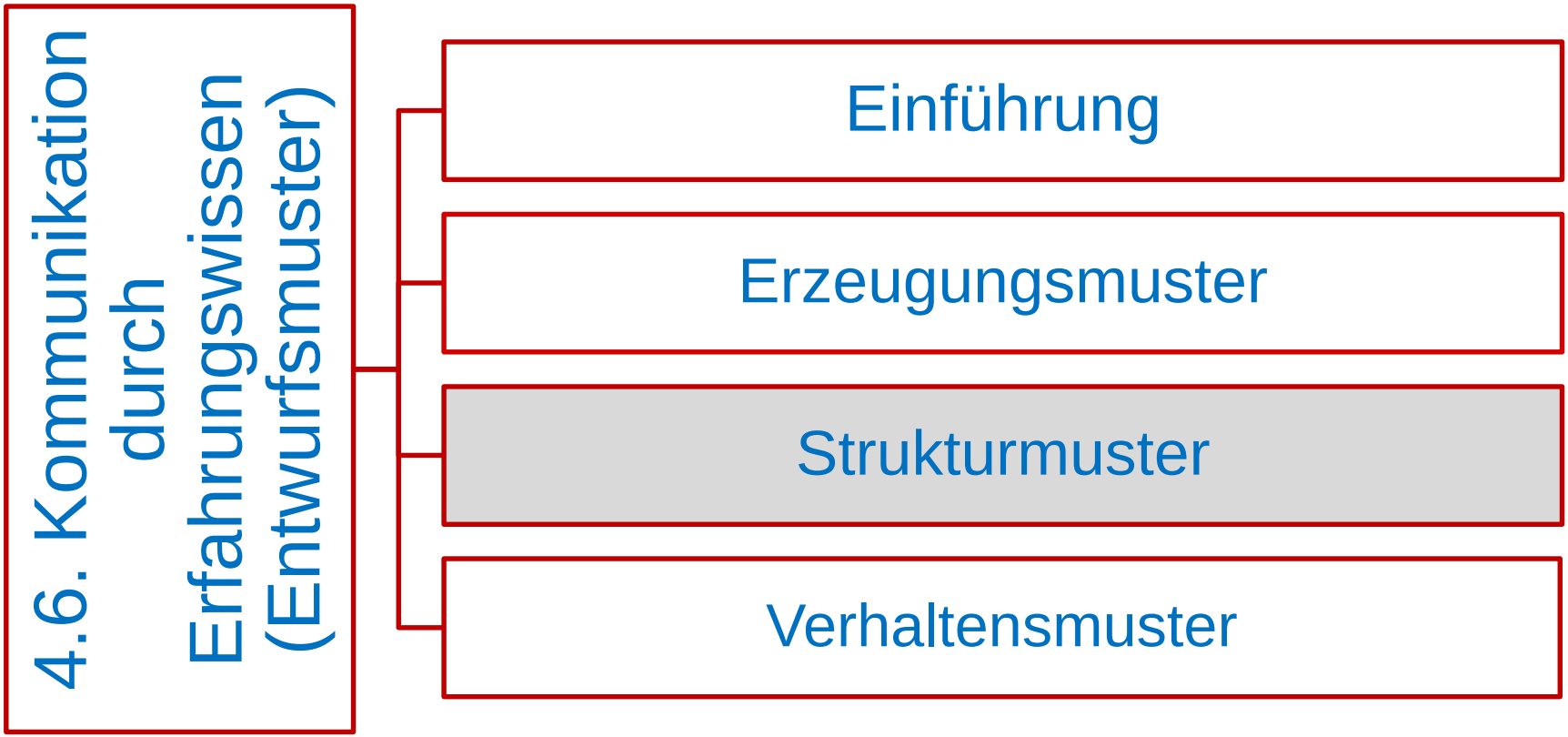
Erzeugungsmuster

- Erzeugungsmuster behandeln Erzeugung von Objekten
  - versuchen Erzeugung zu verstecken, zu vereinheitlichen, zu vereinfachen
  - geben an, was erzeugt wird, wie es erzeugt wird und wann es erzeugt wird
- Beispiele
  - Einzelstück (Singelton)
  - Abstrakte Oberklasse
- Weitere
  - Fabrikmethode (Factory Method)
  - Prototyp
  - Builder

Abstrakte Oberklasse

- **Erzeugungsmuster**
- **Problem**: Klassen enthalten Gruppen von identischen Attributen und Operationen
- **Lösung**: separieren der identischen Bestandteile in einer abstrakten Oberklasse





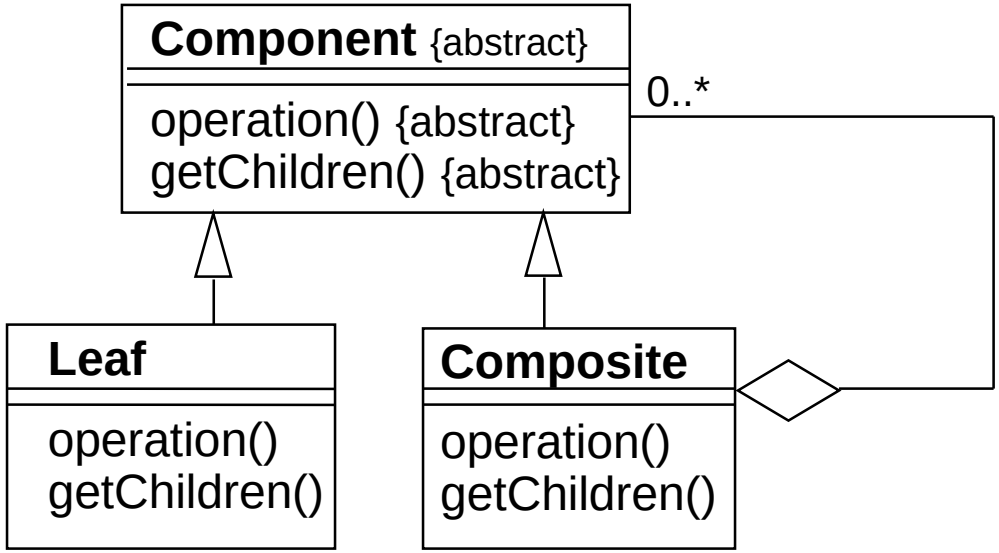
Strukturmuster

- Strukturmuster befassen sich mit Komposition von Klassen => um größere Strukturen bilden zu können
- Beispiele
  - Kompositum
- Weitere
  - Adapter
  - Proxy
  - Bridge
  - Decorator
  - Fassade
  - Fliegengewicht

Kompositum (1)

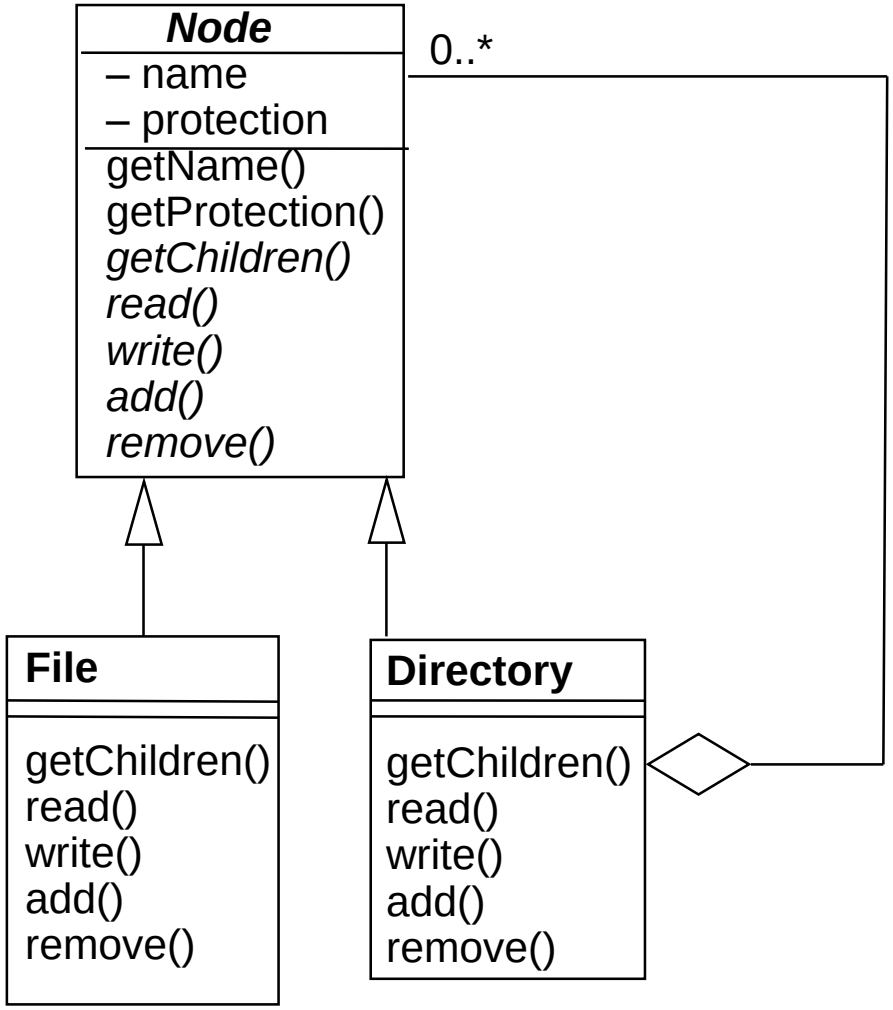
- engl. Composite
- Strukturmuster
- **Problem**:
  - Modellierung und Umsetzung von hierarchischen Strukturen (Baumstrukturen)
- **Lösung**:
  - Einheitliche abstrakte Oberklasse für „Blätter“ und Verzweigungsknoten

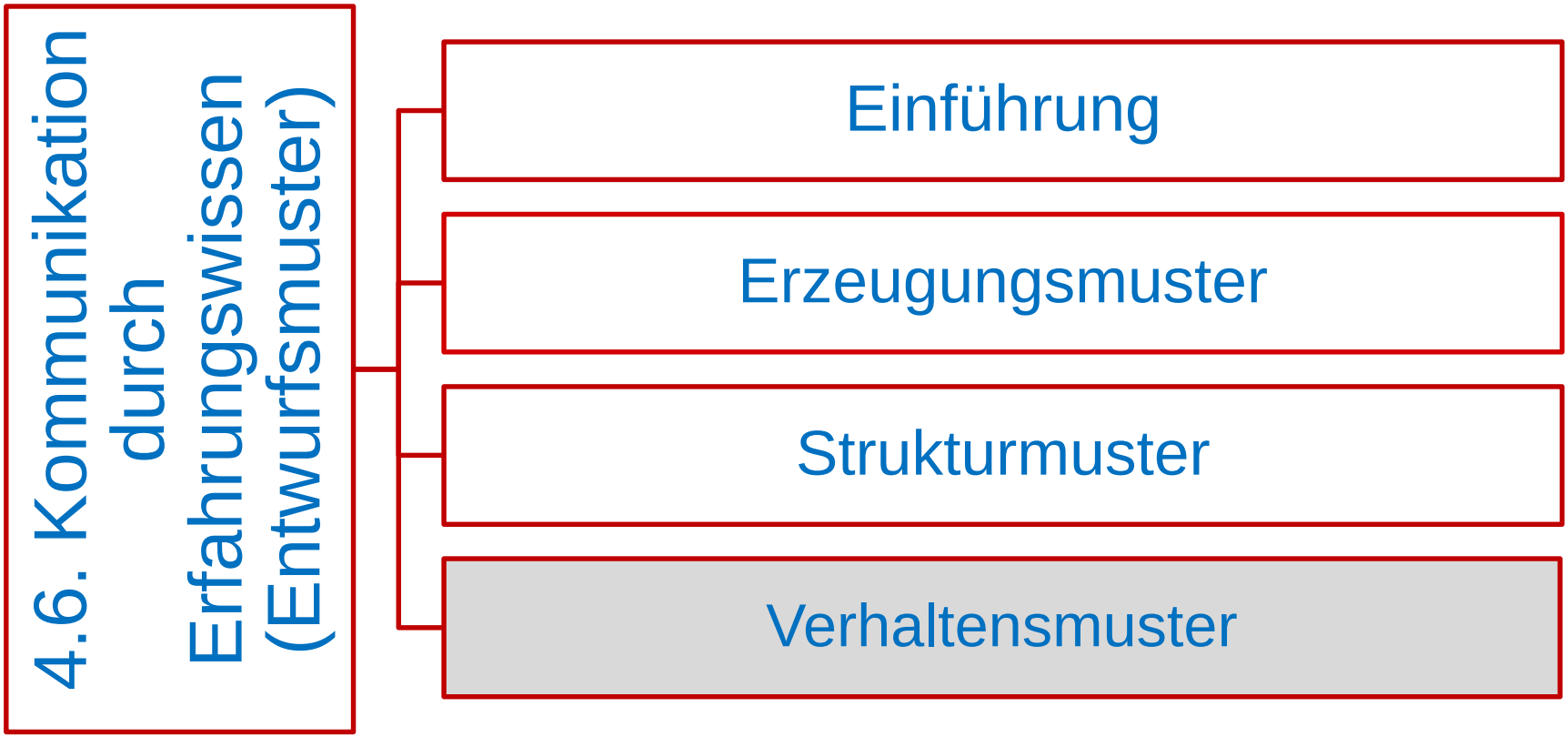
Kompositum (2)



Beispiel Kompositum

- Beispiel Dateisystem





Verhaltensmuster

- Verhaltensmuster beschäftigen sich mit Algorithmen und der Zuweisung von Zuständen zu Objekten
- Beschreiben neben Klassen und Objekten auch die Interaktion zwischen ihnen

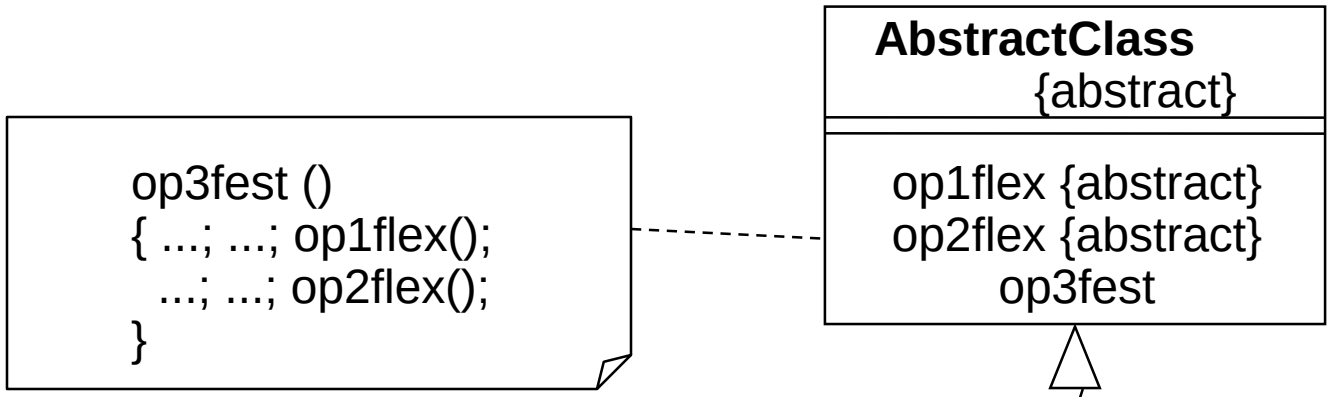
- Beispiele
  - Schablonenoperation

- Weitere
  - Befehl
  - Beobachter
  - Besucher
  - Interpreter
  - Memento
  - Strategie
  - Iterator
  - Vermittler
  - Zustand
  - Zuständigkeitskette

Schablonenoperation (1)

- engl. Template Method
- **Verhaltensmuster**
- **Problem**:
  - Operation besteht aus festen und veränderlichen Bestandteilen
- **Lösung**:
  - Definiere Schablonenoperation, die Skelett eines Algorithmus definiert
  - Algorithmus wird in Unterklassen genau beschrieben
  - Feste Bestandteile in Oberklasse implementiert

Schablonenoperation (2)




```
abstract class Node {
    public void write (String s) {
        if (isWriteable()) {
            writeBody(s);
        }
    }
    public abstract boolean isWritable();
    public abstract void writeBody();
…
}
```

```
class Directory extends Node {
    public boolean isWriteable() {
        return false;
    }
    public void writeBody(String s) { }
…
}
```

Weitere Muster

- In der Zentralübung

Vorteile von Mustern

- Wiederverwendung von bewährten Lösungen für immer wiederkehrende Probleme
- bessere Lesbarkeit/ Wartbarkeit der Software-Entwürfe und des Quellcodes
- einfachere Kommunikation durch gemeinsames "Vokabular"
  - zwischen ArchitektIn und EntwicklerIn
  - zwischen EntwicklerInnen untereinander
  - Hilfestellung für unerfahrene EntwicklerIn

Nachteile von Mustern

- Anwendung eines Musters im falschen Kontext
  - führt zum "Overhead" (Pseudo-Klassen werden eingefügt, nur damit man ein bestimmtes Muster hat)
  - führt zur schlechteren Lesbarkeit/ Wartbarkeit des Entwurfs/ des Codes (Verwirrung über nicht angemessene Verwendung)

Zusammenfassung Entwurfsmuster

- Entwurfsmuster geben Erfahrungswissen wieder
- Sie sind Lösungsvorschläge für eine Menge ähnlicher Problemen
- 3 Arten von Entwurfsmuster
  - Erzeugungsmuster
  - Strukturmuster
  - Verhaltensmuster

Literatur Entwurfsmuster

- E Gamma, R Helm, R Johnson, J Vlisside (2009) Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software“, Addison Wesley
- Viele Webseiten zu Design Pattern, z.B.
  - http://www.oodesign.com/
  - http://en.wikipedia.org/wiki/Software_design_pattern