Übung 11

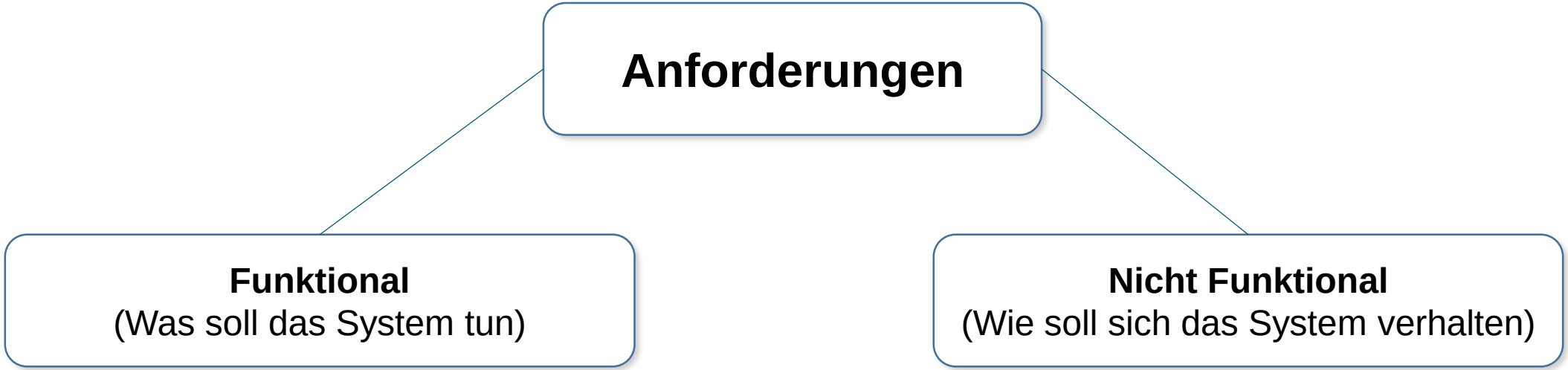
Prüfungsvorbereitung

0. Organisatorisches

- **Klausur:**
  - 17.02.2016, 13-15 Uhr, B11, SR15
  - Keine Hilfsmittel!!!
  - Papier wird bereitgestellt

http://giphy.com/gifs/exam-gNJNnBA5leOoU

01. Funktionale vs. Nicht Funktionale Anforderungen



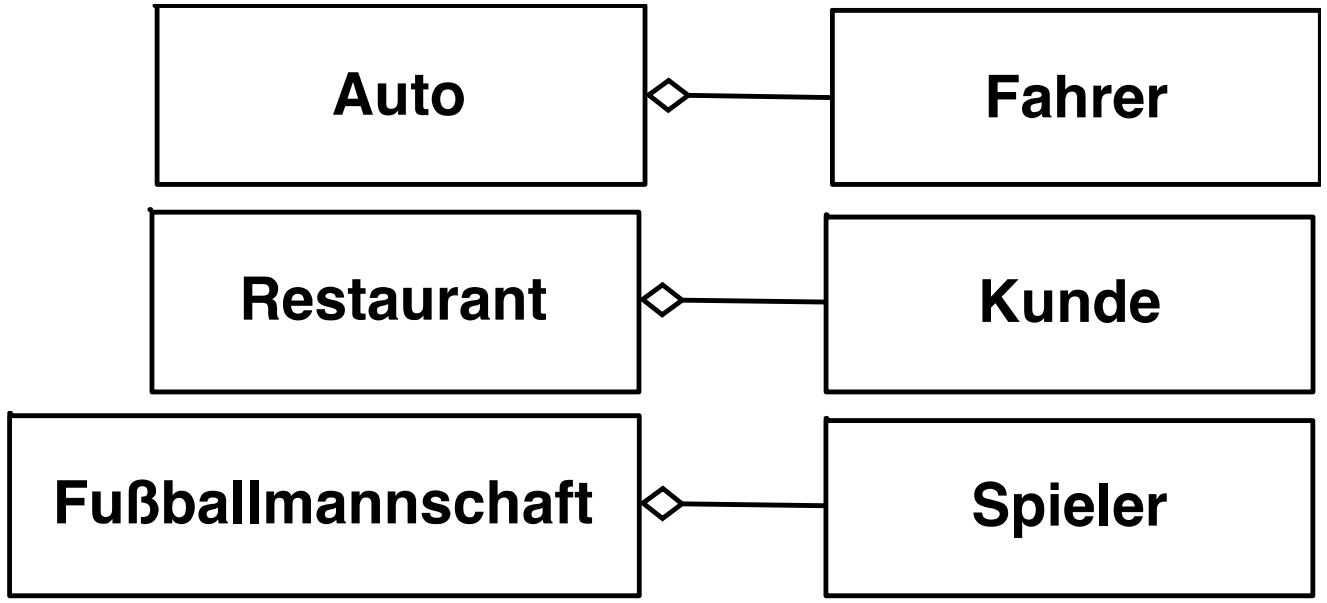
- Benutzerschnittstelle
- Anwendungsfälle
- Dienstleistungen
- Implementierung von Feature XYZ

- Zuverlässigkeit
  (Wiederherstellbarkeit, Fehlertoleranz)
- Look and Feel
- Benutzbarkeit
  (Verständlichkeit, Erlernbarkeit, Bedienbarkeit)
- Leistung und Effizienz
  (Antwortzeiten, Ressourcenbedarf)
- Wartbarkeit

02. UML: Aggregation vs. Komposition

Aggregation

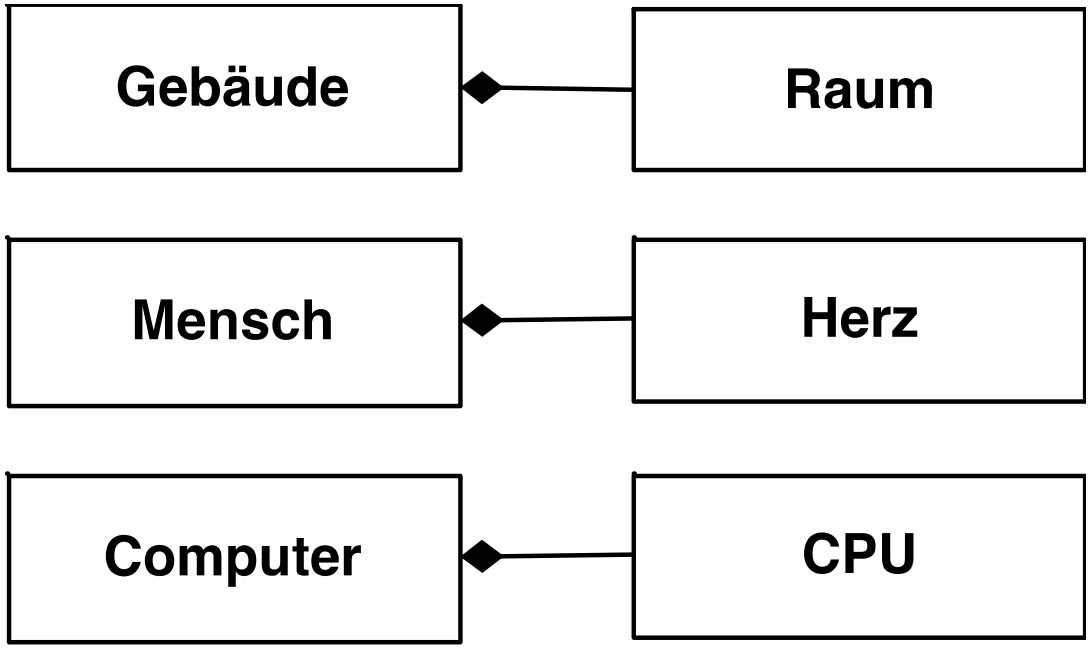
- Teil-Objekt **kann** ohne Aggregat-Objekt weiterhin existieren



02. UML: Aggregation vs. Komposition

Komposition

- Teil-Objekt **kann nicht** ohne Aggregat-Objekt weiterhin existieren



02. UML: Meta-Metamodell

Was ist ein Metamodell?

Eine wohl bekannte Analogie:

Deutsche Sprache als Modell
Welche Wörter dürfen benutzt werden?
Wie werden sie zu Sätzen und Texten zusammengesetzt?

Duden als Metamodell
Beschreibung der deutschen Sprache
Begriffswelt, Grammatik

02. UML: Meta-Metamodell

Metamodell

- Grundlage für Modelle
- Modell zum Aufbau von Modellen
- Definition der Modellierungssprache
- Beschreibung von Metaklassen
- Grundlage von Entwurfstools

Modell

- Basierend auf Metamodell
- Modell zu konkretem Sachverhalt
- Beschrieben mit Modellierungssprache
- Instanzen von Metaklassen
- Ergebnis der Anwendung von Entwurfstools

02. UML: Meta-Metamodell

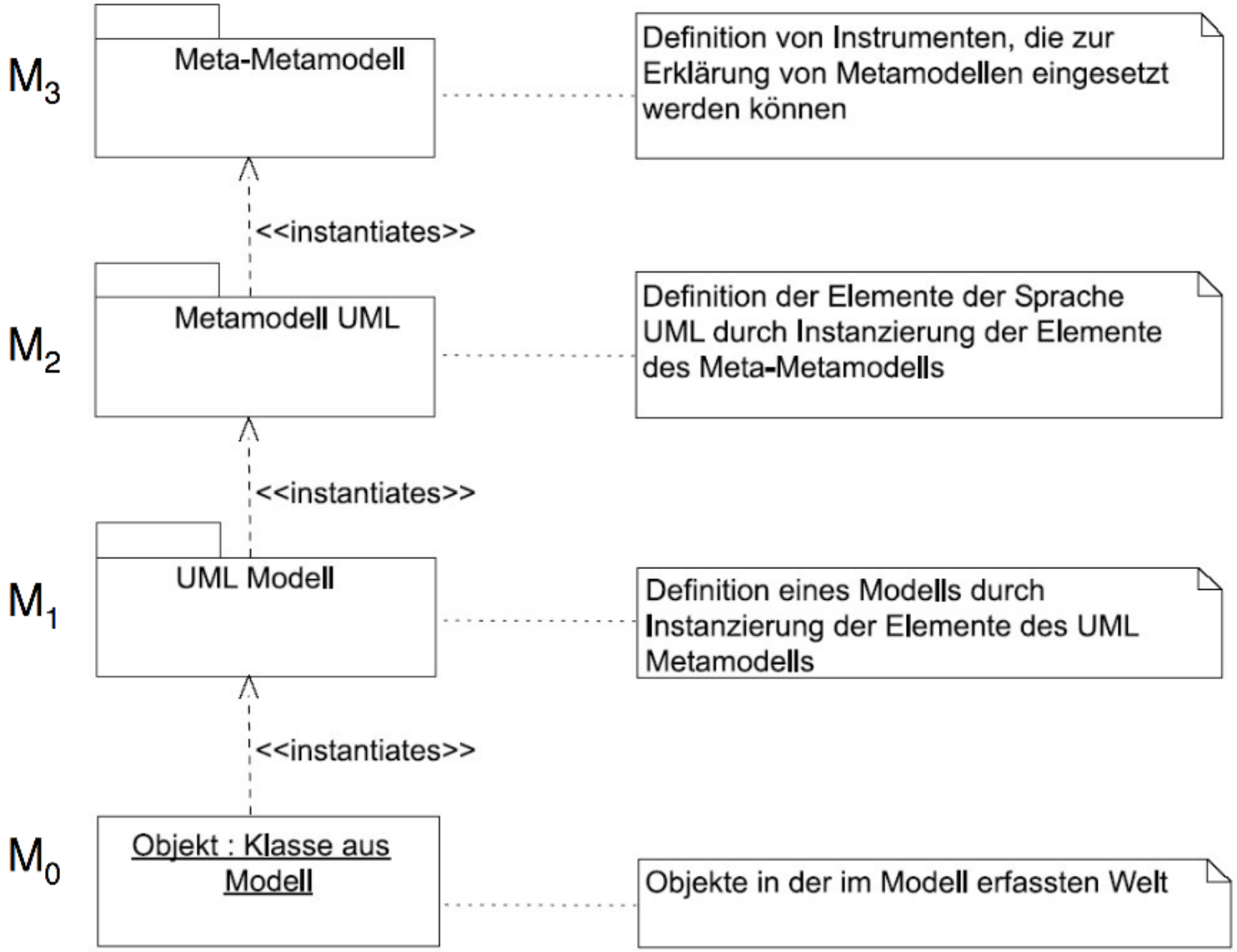
Meta-Metamodell

- Beschreibung von Metamodellen
- Grundkonzepte der Objektorientierung

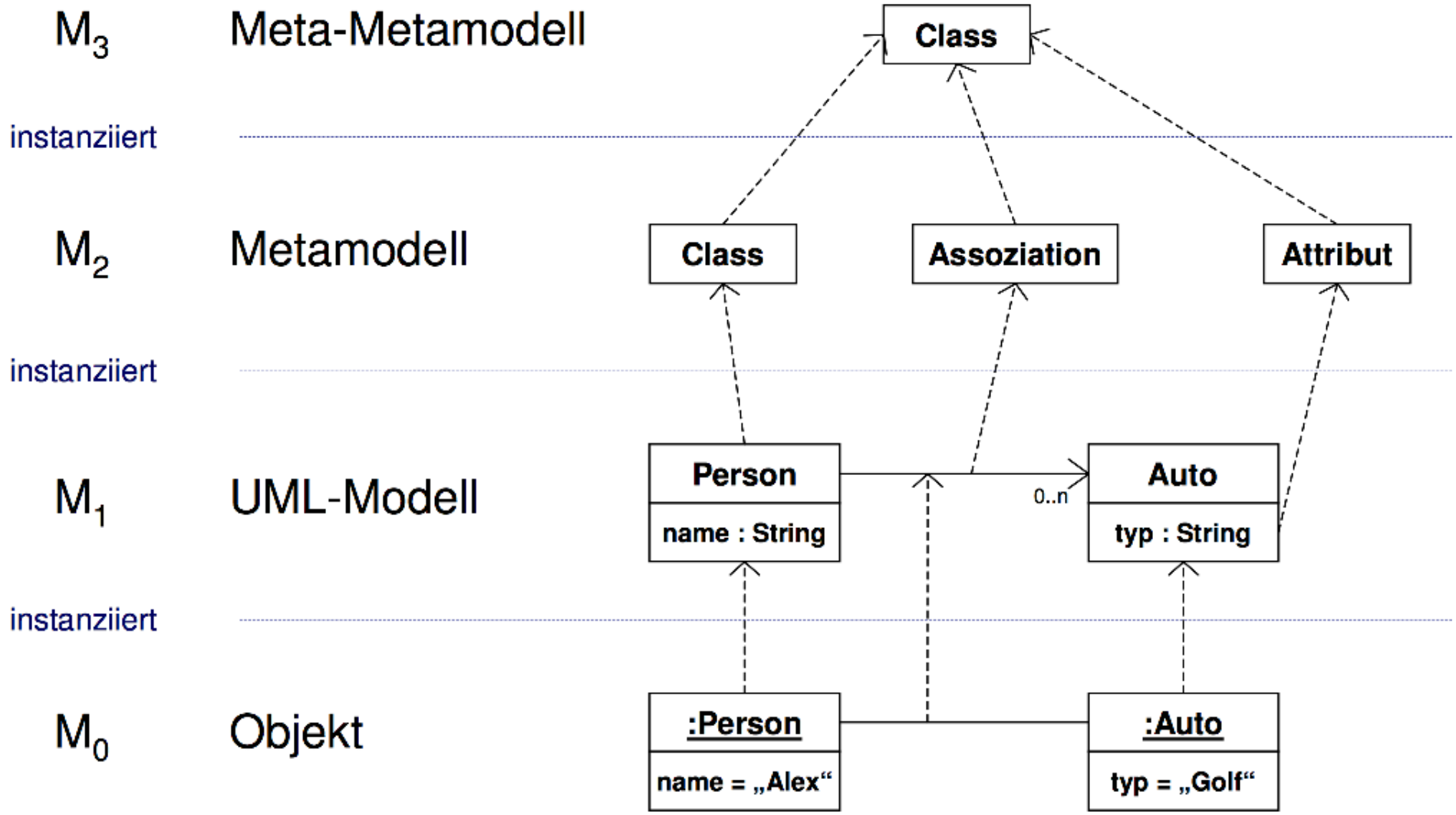
- Meta-Meta-Metamodell denkbar (rekursiv)
  - Kein weiterer Abstraktionsgewinn

➢ Entstehung einer Vier-Schichten-Architektur

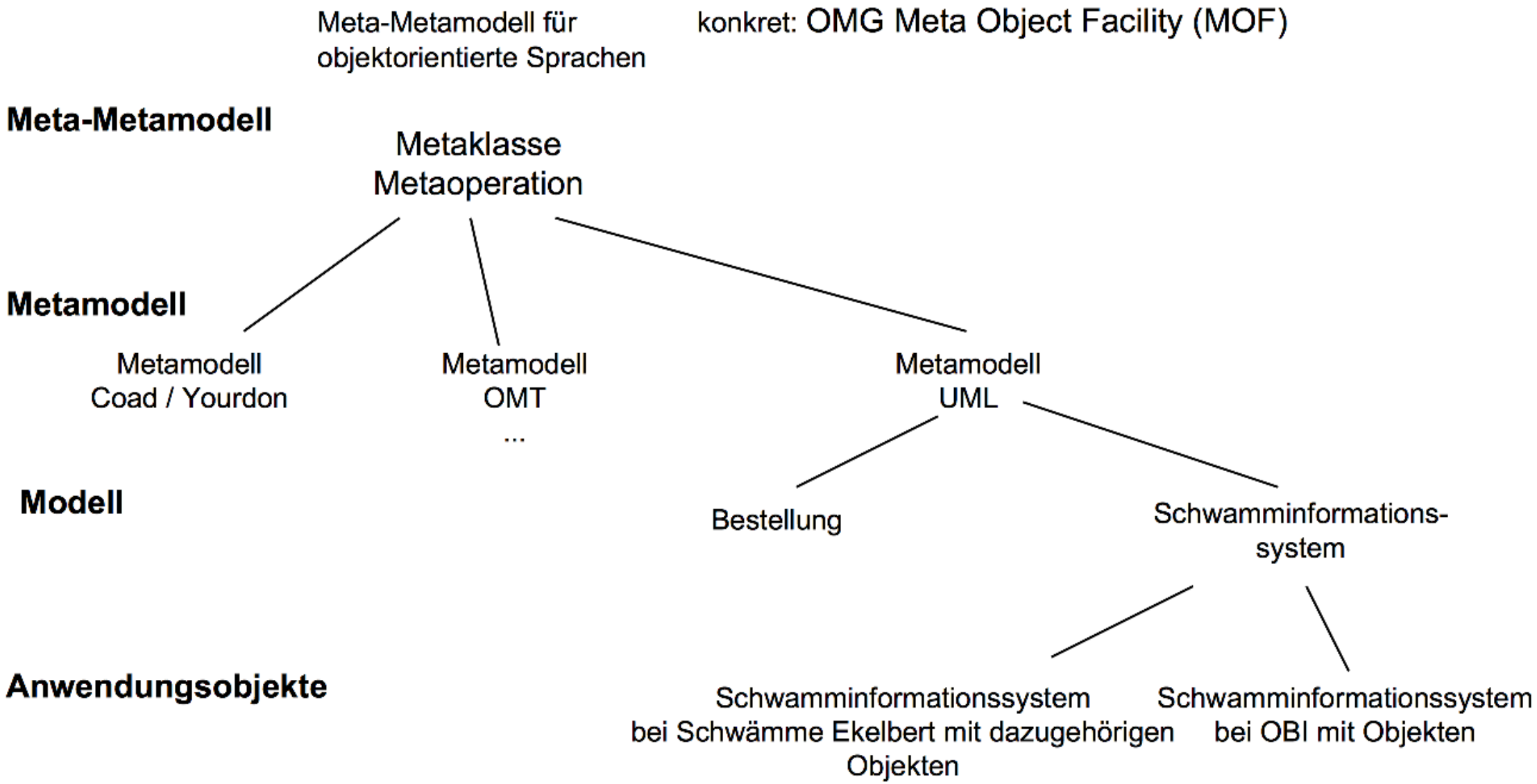
02. UML: Meta-Metamodell



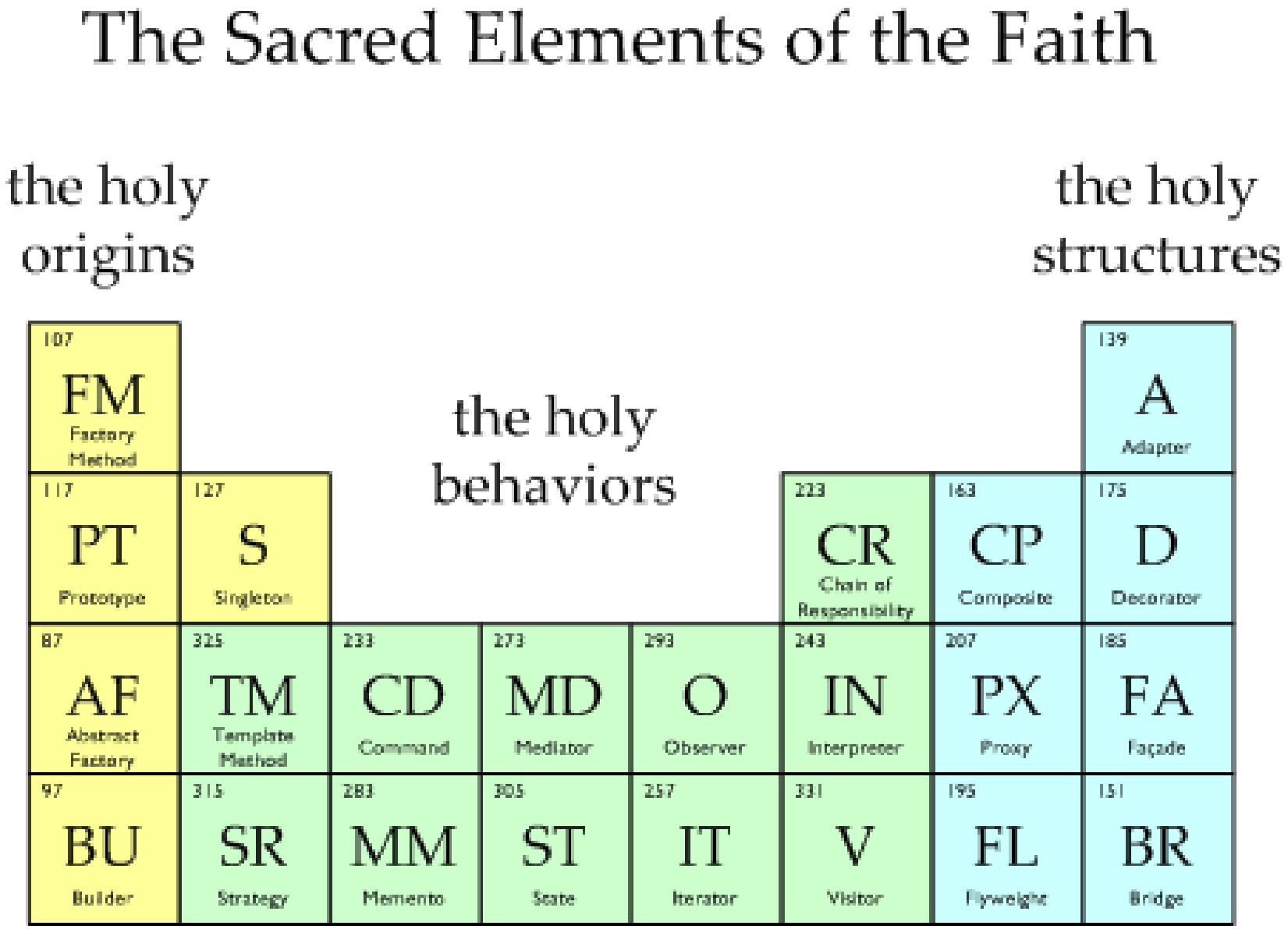
02. UML: Meta-Metamodell



02. UML: Meta-Metamodell



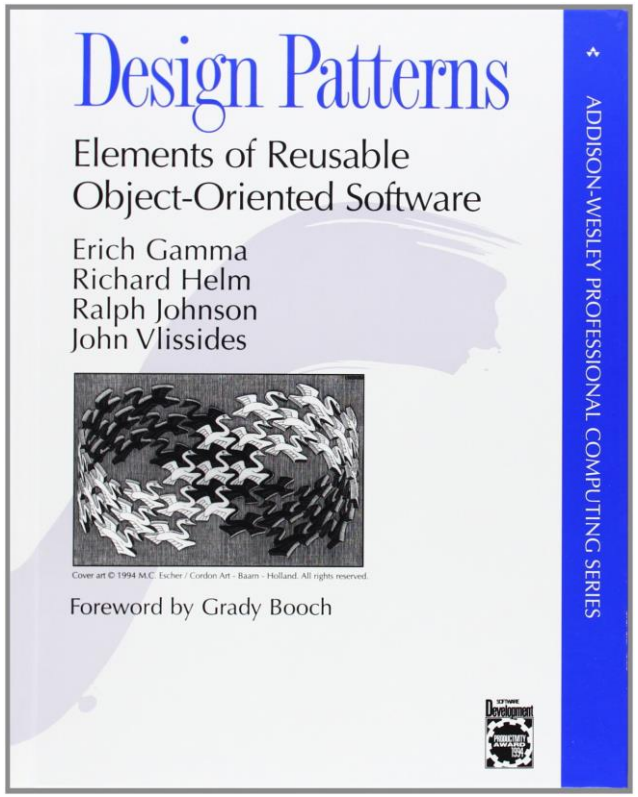
03. Design Patterns



03. Design Patterns

- Hilfreiche Links:
  - http://www.vincehuston.org/dp/
  - https://www.philipphauer.de/study/se/design-pattern.php
  - https://www.tutorialspoint.com/design_pattern/index.htm

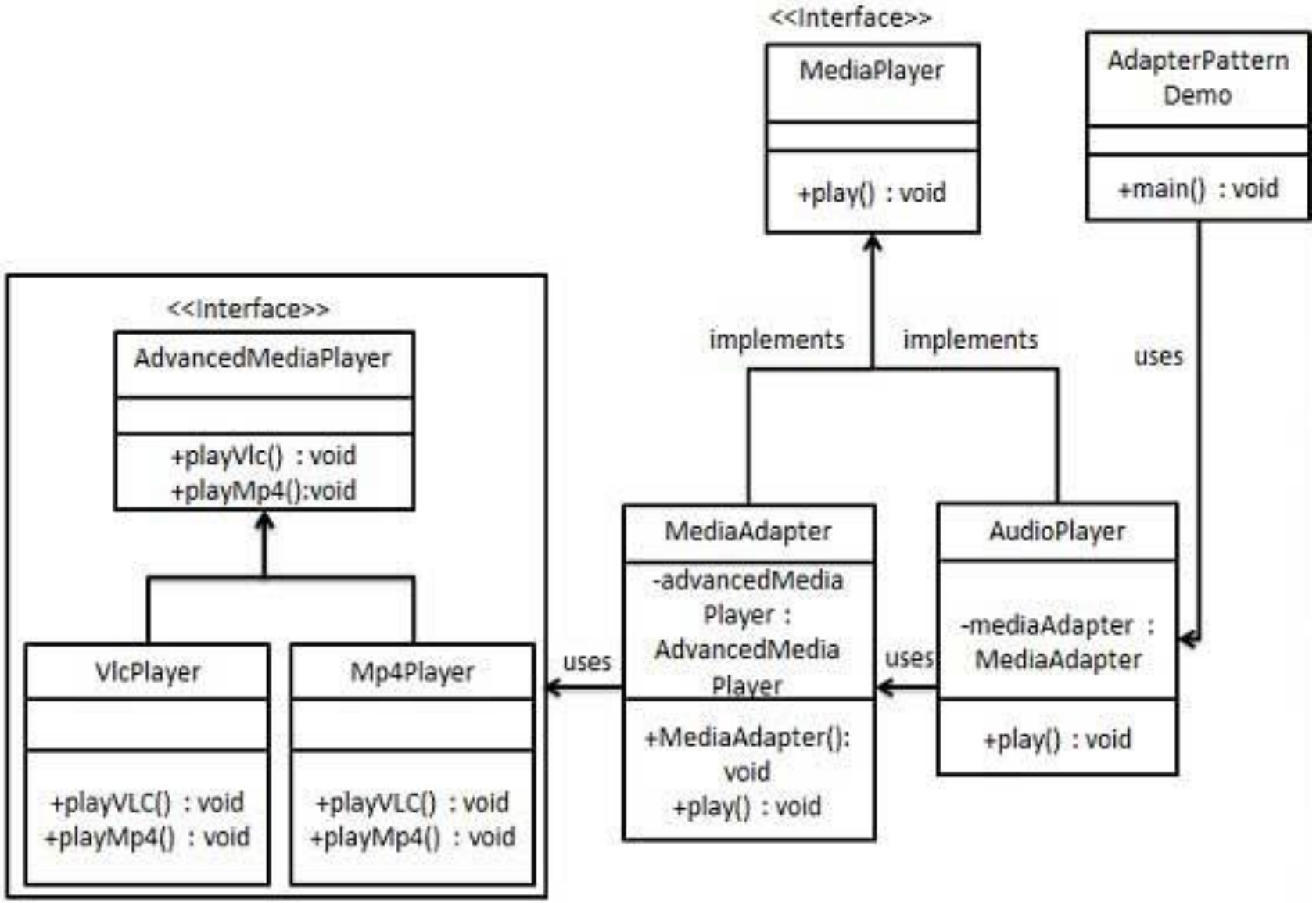
- Standardbuch:
  - Design Patterns. Elements of Reusable Object-Oriented Software



03. Design Patterns: Adapter

- Brücke zwischen zwei inkompatiblen Interfaces bzw. Schnittstellen
- **Analogie**: Kartenlesegerät als Adapter zwischen Speicherkarte und PC
- **Kerninhalte**:
  - Eine Klasse ist verantwortlich für das Verbinden unabhängiger und inkompatibler Schnittstellen

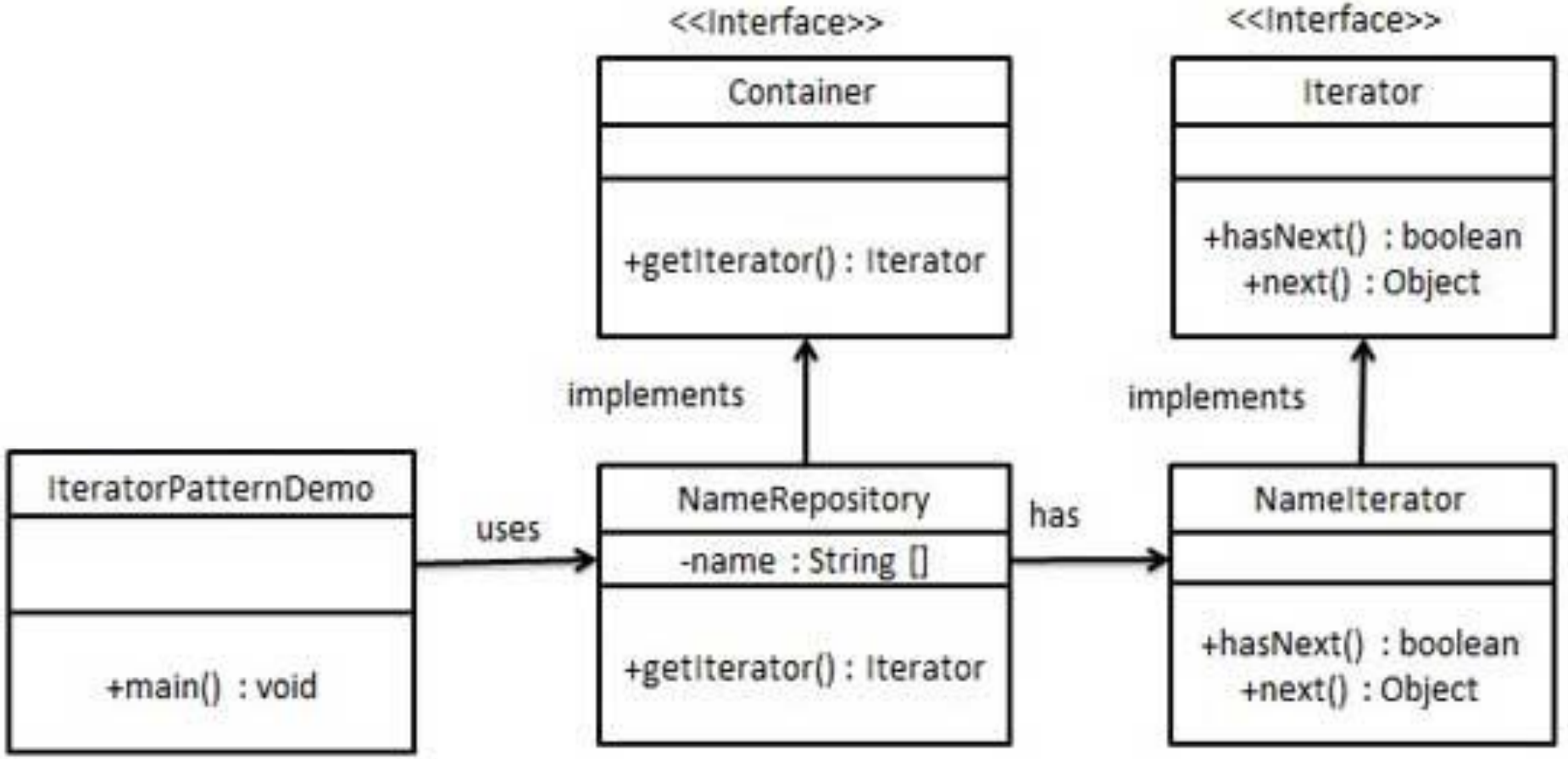
03. Design Patterns: Adapter



https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

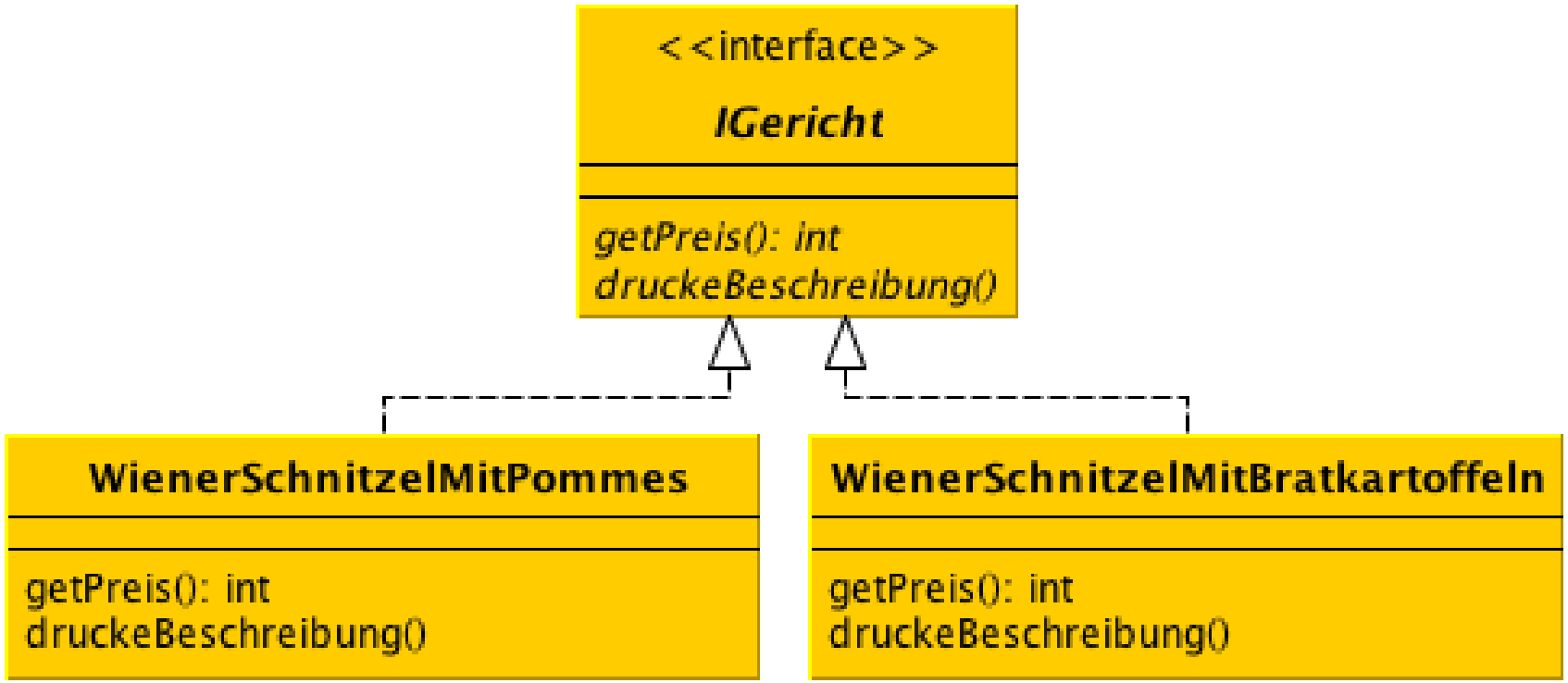
03. Design Patterns: Iterator

- Elementzugriff einer Collection in sequenzieller Art
- Oft in Java und .Net benutzt

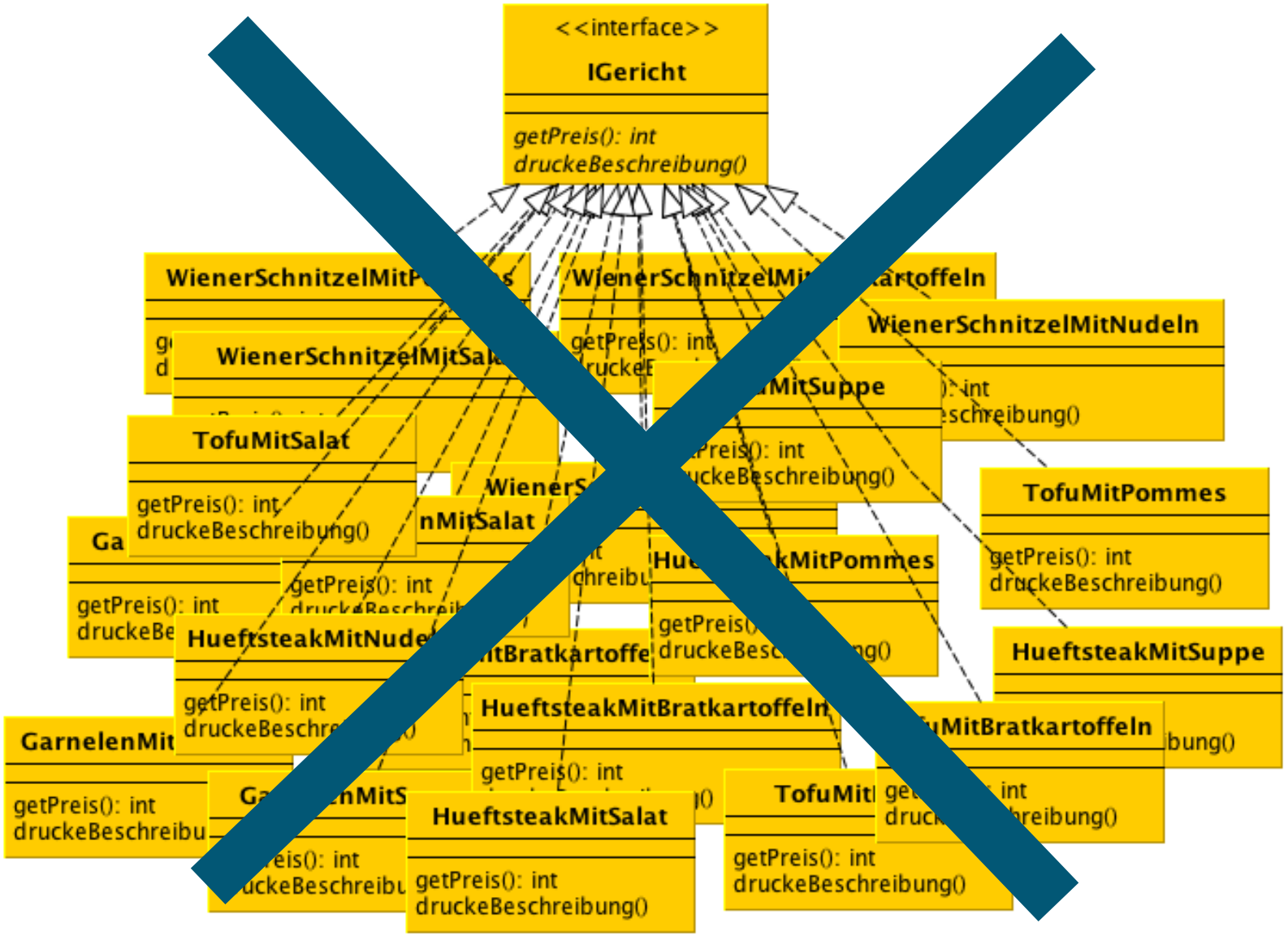


https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm

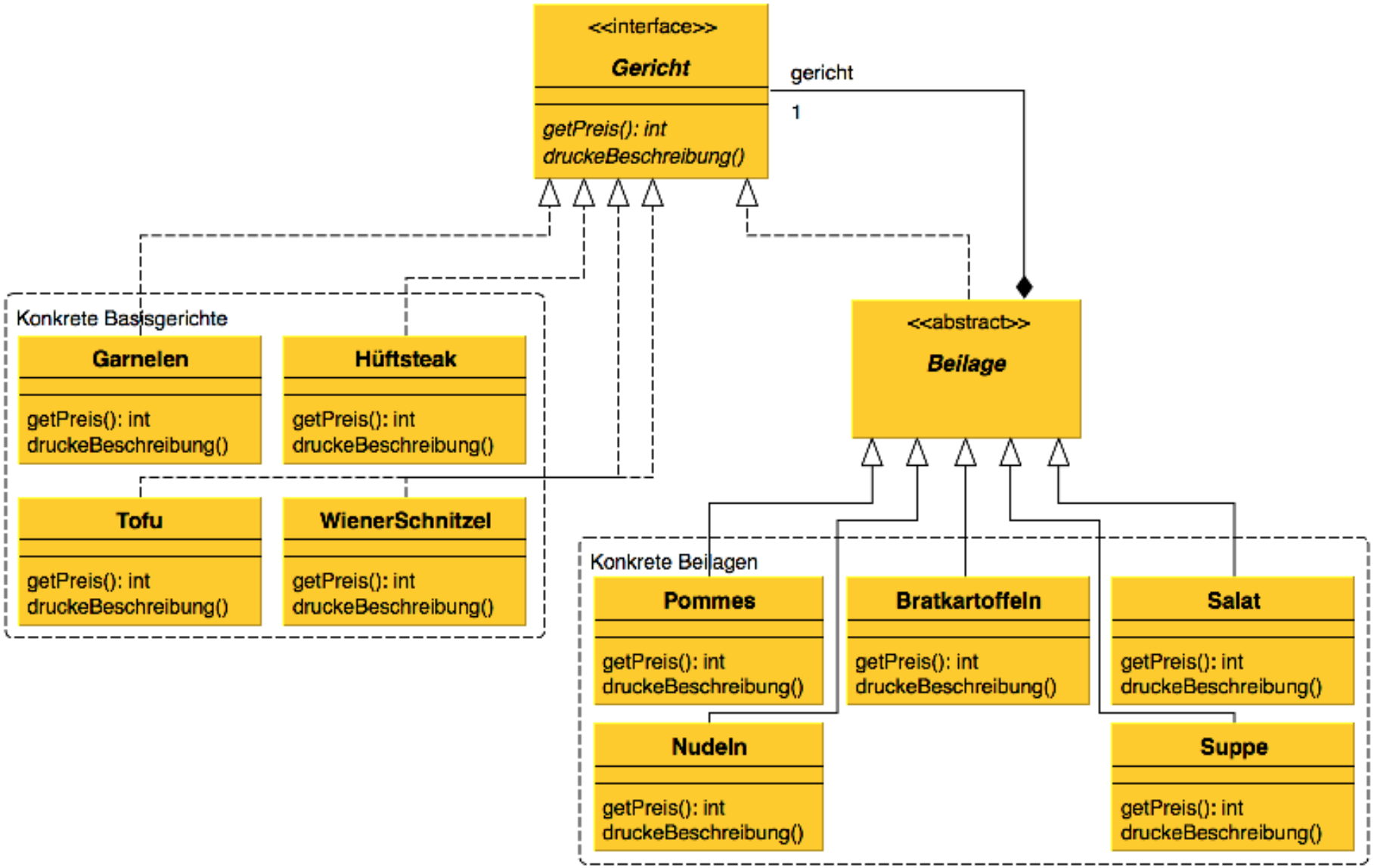
03. Design Patterns: Decorator



03. Design Patterns: Decorator



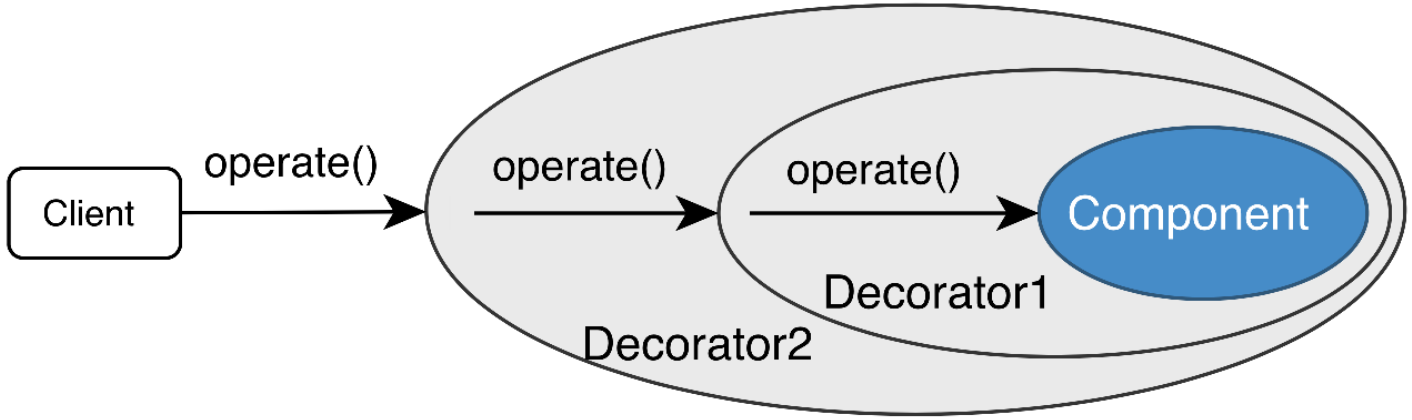
03. Design Patterns: Decorator



http://www.philipphauer.de/study/se/design-pattern/decorator.php

03. Design Patterns: Decorator

- Hinzufügen neuer Funktionalitäten zu einem existierenden Objekt (Komponente) durch andere Klassen (Dekorierer)
- Decorator ist vom selben Typ wie das zu dekorierende Objekt
- Delegiert Methodenaufrufe an seine Komponenten weiter und führt sein eigenes Verhalten davor oder danach aus
- Eine Komponente kann mit beliebig vielen Decorators dekoriert werden

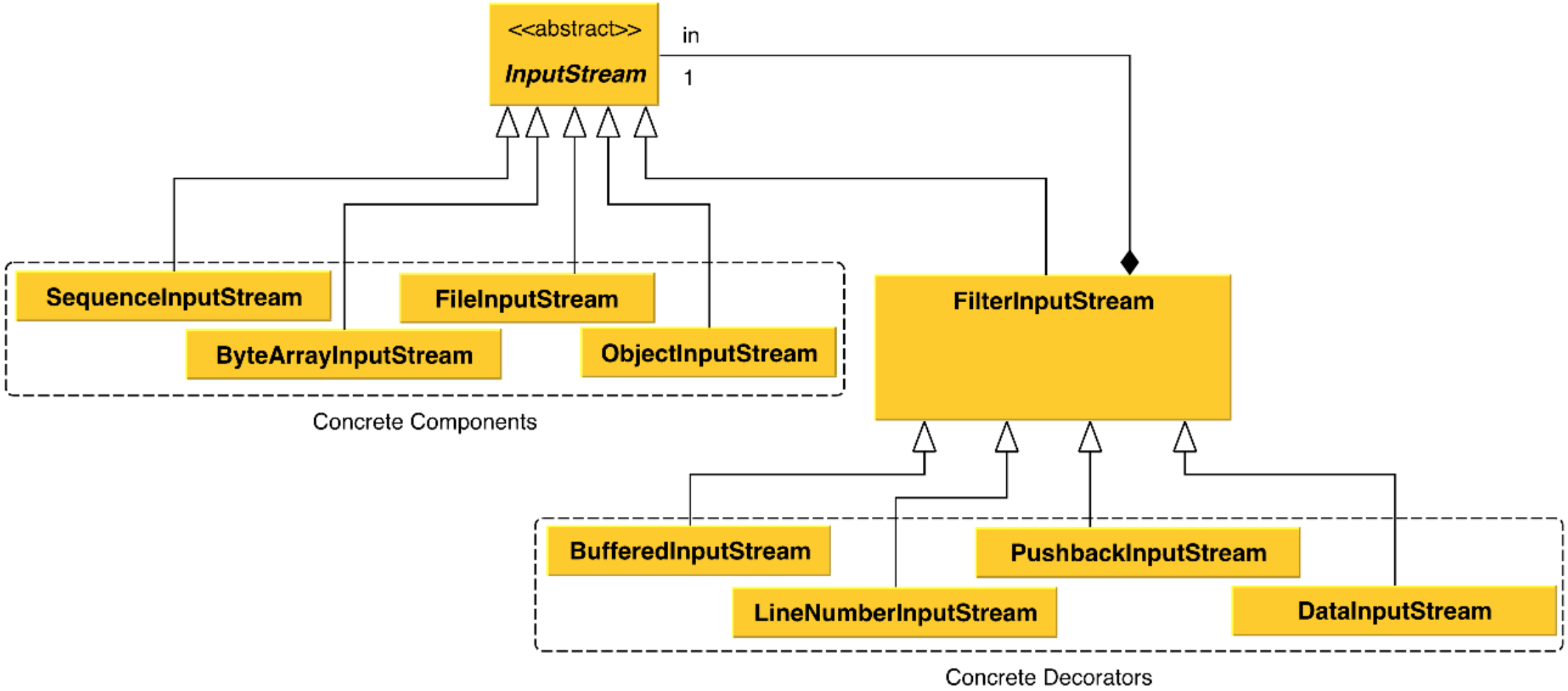


03. Design Patterns: Decorator

- **Vorteile:**
  - Klassen können ohne statische Vererbung um Verhalten erweitert werden
  - Vermeidung von langen unübersichtlichen Vererbungshierarchien
  - Funktionalitäten müssen nur dann initialisiert werden, wenn sie wirklich benötigt werden
  - Wartbar

- **Nachteile:**
  - Erschwerte Fehlerfindung
  - Hohe Objektanzahl
  - Systemkomplexität: nicht Einsteigerfreundlich (siehe Java-IO)

03. Design Patterns: Decorator




```
Inputstream is = new BufferdInputStream(new FileInputStream((new File(test.txt))))
```

03. Design Patterns: Observer

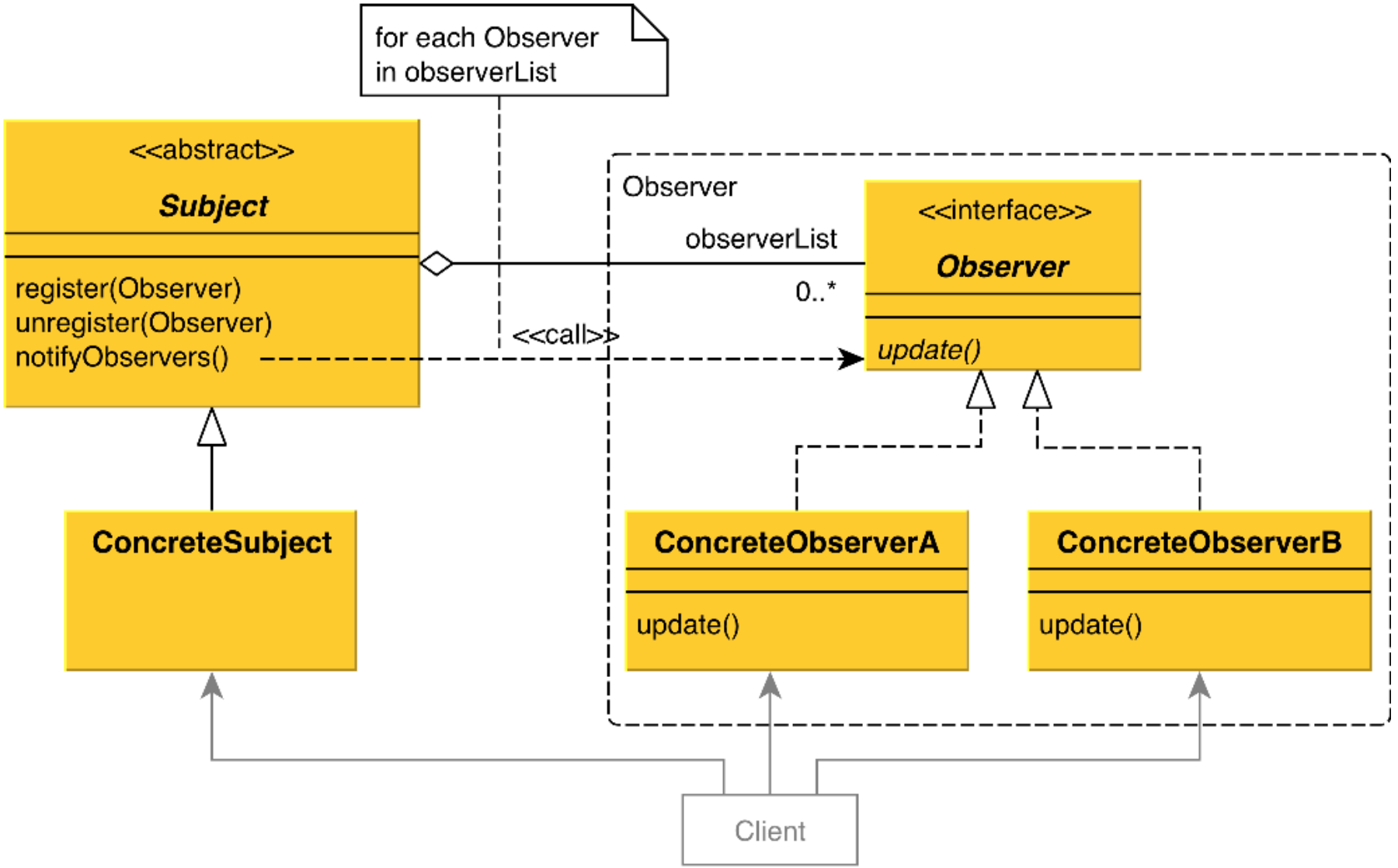
- Registrierung von Objekten (Observer) an andere Objekte (Subject) zur Informationsweitergabe bei Veränderungen

- **Analogie**: Abonnenten (Observer) können einen YT-Channel (Subject) subscriben und bekommen im Newsfeed die neusten Videos bereitgestellt.

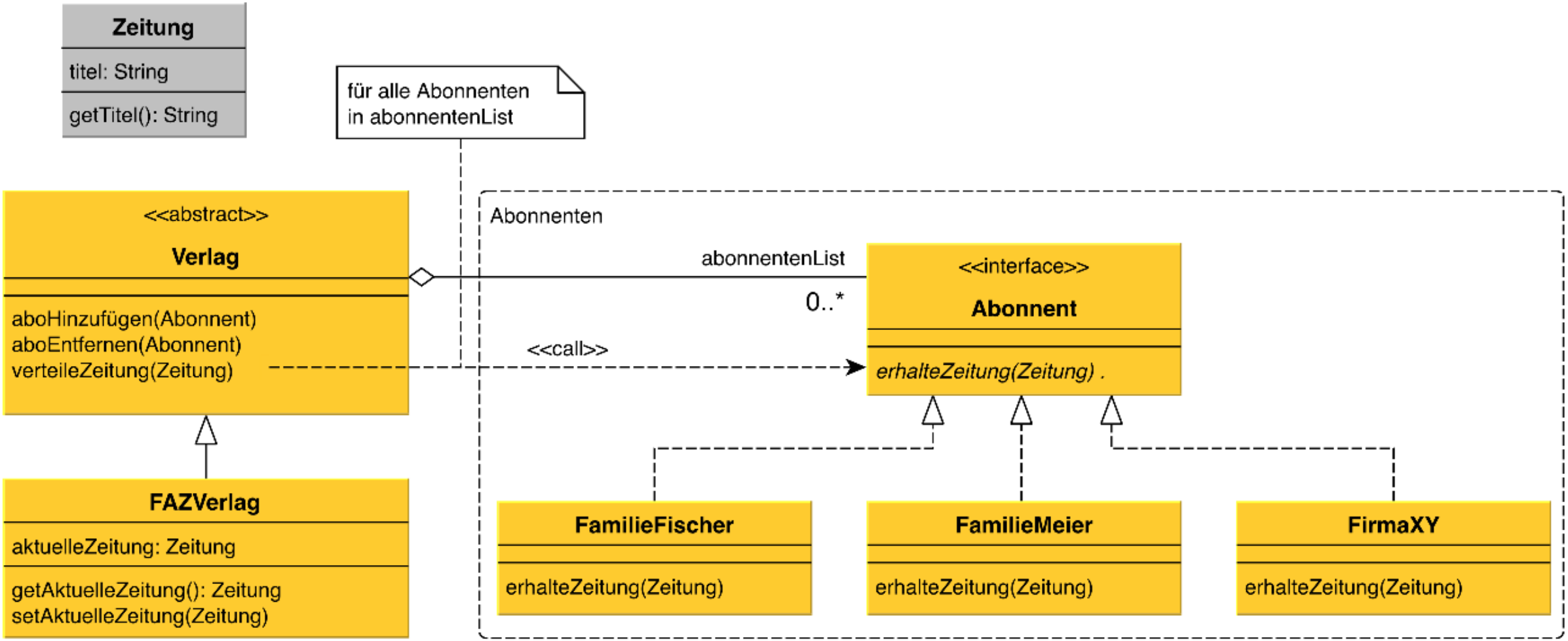
- **Observer**
  - Einheitliche Schnittstelle (Interface) mit mind. einer Aktualisierungsmethode
  - Aufruf dessen durch Subject

- **Subject**
  - Beinhaltet Administrationsmethoden (Registrieren und Löschen von Observern)
  - Ruft über `notifyObservers()` alle update-Funktionen der registrierten Observer auf

03. Design Patterns: Observer



03. Design Patterns: Observer

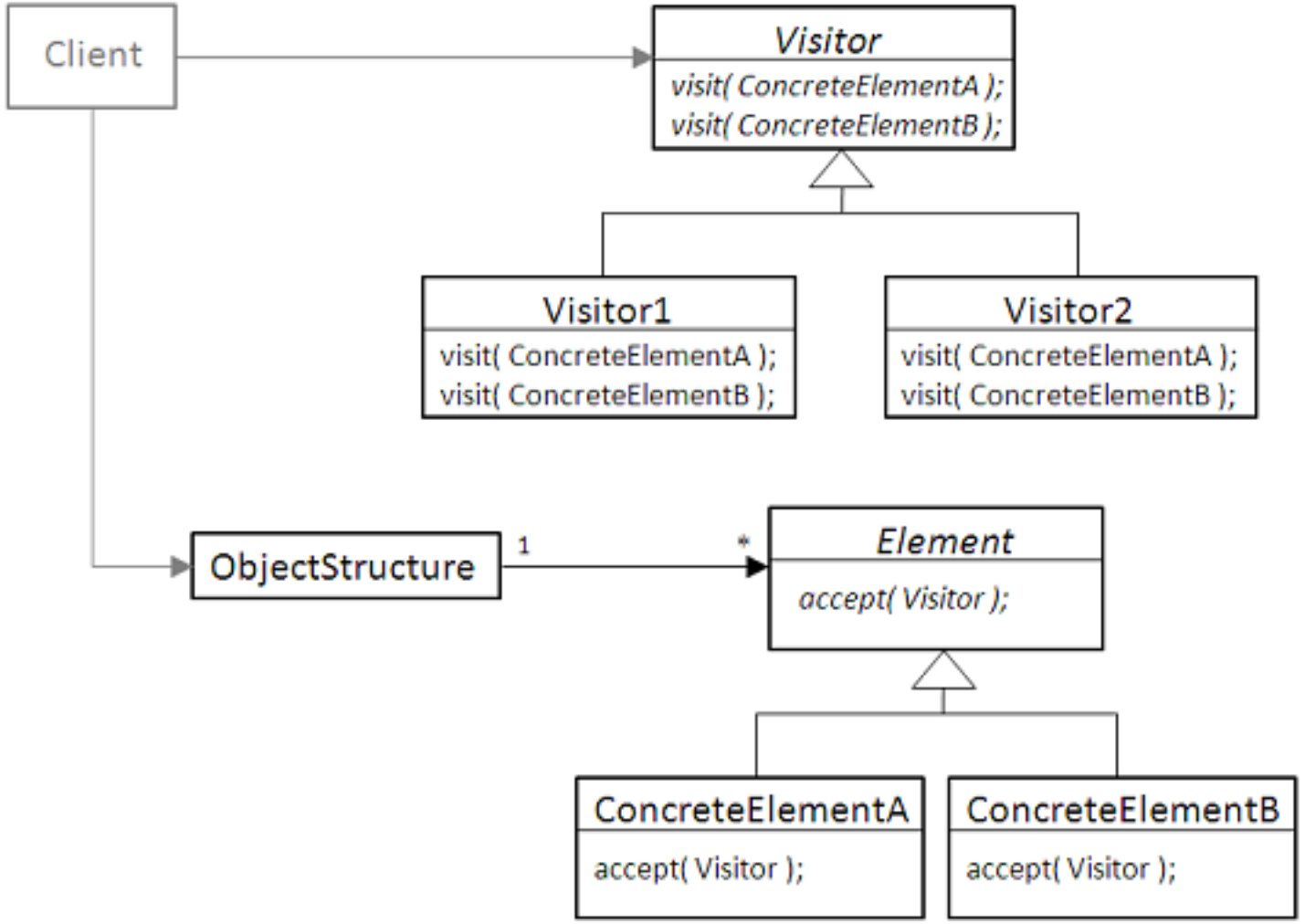


03. Design Patterns: Observer

- **Vorteile:**
  - Daten im Gesamtsystem bleiben konsistent
  - Mehrere verschiedene Observer können ein einziges Subject beobachten
  - Wiederverwendbarkeit der Klassen
  - Subject und Observer sind lose und abstrakt gekoppelt

- **Nachteile:**
  - Aktualisierungskaskaden und –zyklen
  - Observer muss sich vom Subject abmelden – kann in einem großen System schnell vergessen werden
  - „Mehrfachanmeldung“ im einfachen Fall möglich

03. Design Patterns: Visitor



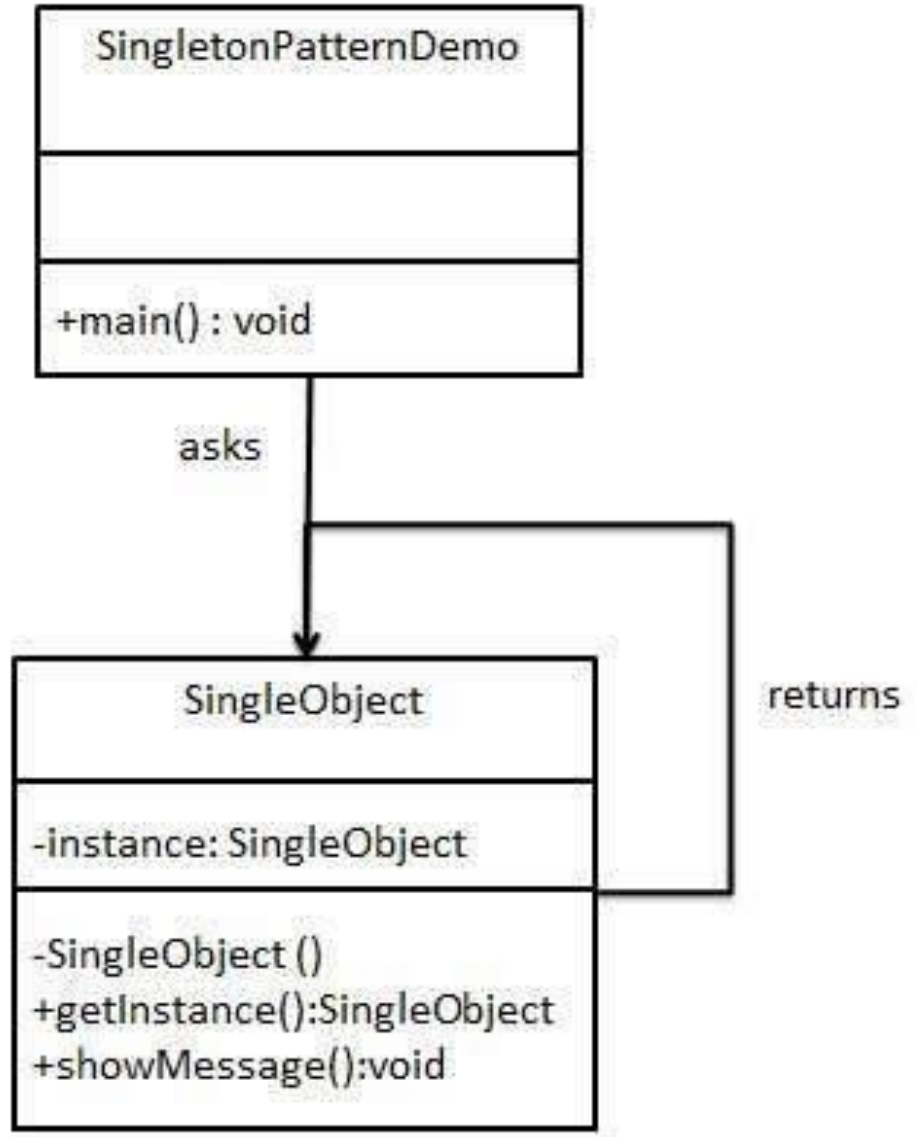

```
visit(ConcreteElementA a) {
  // DO SOMETHING WITH THE ELEMENT
}
```

```
accept(Visitor v) {
  v.visit(this);
}
```

03. Design Patterns: Singleton

- Nur eine einzige Instanz einer Klasse ist erlaubt

- Klasse regelt den Zugriff auf die Instanz über
  - privaten Konstruktor
  - statischer Variable instance
  - statischer Methode getInstance()



04. Fünf Regeln für gutes Design

- Folgen aus den 5 Kriterien der Modularität
  - Modular **Decomposability**
    - Problem kann in wenige kleinere, weniger komplexe Sub-Probleme zerlegt werden
  - Modular **Composability**
    - Softwarekomponenten können beliebig kombiniert werden
  - Modular **Understandability**
    - Entwickler kann jedes einzelne Modul verstehen, ohne die anderen zu kennen
  - Modular **Continuity**
    - Kleine Änderung der Problemspezifikation führt zu Änderung in nur einem Modul
  - Modular **Protection**
    - Abnormales Programmverhalten in einem Modul bleibt in diesem Modul

04. Fünf Regeln für gutes Design

Direct Mapping

- Wünsche des Kunden meist in seinem Problembereich = Domain (spezifisches Vokabular)
- Verstehe das Problem UND dessen Domain
- Formulierung einer Lösung auf Grundlage dessen

Few Interfaces

- Beschränke die Anzahl der Kommunikationskanäle zwischen Modulen

➢ Jedes Modul sollte mit so wenig anderen wie möglich kommunizieren

Small Interfaces

- Wenn zwei Module kommunizieren, sollen sie so wenige Informationen wie möglich austauschen

Explicit Interface

- Wenn Module A und B kommunizieren, sollte dies aus dem Programmtext von A oder B oder beiden hervorgehen.
- Funktionsparameter und globale Variablen: wer kann darauf zugreifen und wie werden sie geändert?

Information Hiding

- Nur eine Teilmenge der Eigenschaften ist nach außen hin sichtbar
- Rest ist versteckt (sowohl Inhalt (Membervariablen), als auch Implementierungen)

06. Reengineering Lebenszyklus

- Siehe VL9

ENDE

Viel Erfolg bei Euren bevorstehenden Klausuren!