universitätbonn ROOTS

Übersicht
Design Patterns

2010-2011: Eva Stöwe und Jan Nonnen

2012-2014: Jan Burmeister

24.01.2014

Disclaimer

- Diese Folien sind privat erstellt, und kein offizielles Material der Vorlesung. Insbesondere gebe ich keine Garantie, dass die Inhalte völlig fehlerfrei sind.
- Diese Folien sind **keine** Argumentationsgrundlage für Diskussionen über Klausurlösungen!
- Die Folien
  - stellen lediglich die Basis der jeweiligen Themen dar
  - erheben keinen Anspruch auf Vollständigkeit
- An mehreren Stellen wurden Sachen vereinfacht oder weggelassen.

Klausurrelevant sind die Vorlesungsfolien.

Übersicht

- Teil 1:
  - Alle Patterns mit Erklärungen zusammengefasst (→)
  - Beispiele zum Üben (→)
- Teil 2:
  - Verwendungsbeispiele: Design Patterns im JDK (→)
  - Verwendungsbeispiele: Design Patterns im Game Engine Design (→)

universitätbonn ROOTS

Design Pattern

von Eva Stöwe und Jan Nonnen
Neubearbeitung von Jan Burmeister
24.01.2014

Pattern ▸ Grundidee

- Einheitliche Sprache für Probleme und ihre Lösungen
  - Erleichtert die Kommunikation zwischen Entwicklern
- Unerfahrenen Entwicklern Lösungen für immer wiederkehrende Entwurfsprobleme bieten
- Standardkatalog der „Gang of Four“ [GoF95]:

  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

  "Design Patterns - Elements of Reusable Object-Oriented Software“
- Analog sind **Anti-Pattern** Negativ-Beispiele von Lösungen, die Hinweise geben, welche Fehler vermieden werden sollten

Pattern ▸ Überblick (Gekürzt)

Erzeugung

- Factory Method
- Abstract Factory
- Singleton

Verhalten

- Observer
- Visitor
- Command
- Strategy
- State

Struktur

- Composite
- Decorator
- Proxy
- Adapter
- Bridge
- Facade

Split Objects

Tipp: Nutzen sie die Links für schnelles Navigieren

Pattern ▸ Was macht ein Pattern aus?

- Jedes Pattern enthält als „Steckbrief“
  - Heuristik
  - Absicht
  - Motivation
  - Anwendbarkeit
  - Schema
  - Konsequenzen
  - Implementierung

- Im folgenden werden aus Platz- und Zeitgründen Punkte bei den Pattern ausgelassen

Factory Method

Factory Method ▸ Übersicht

- **Absicht**:

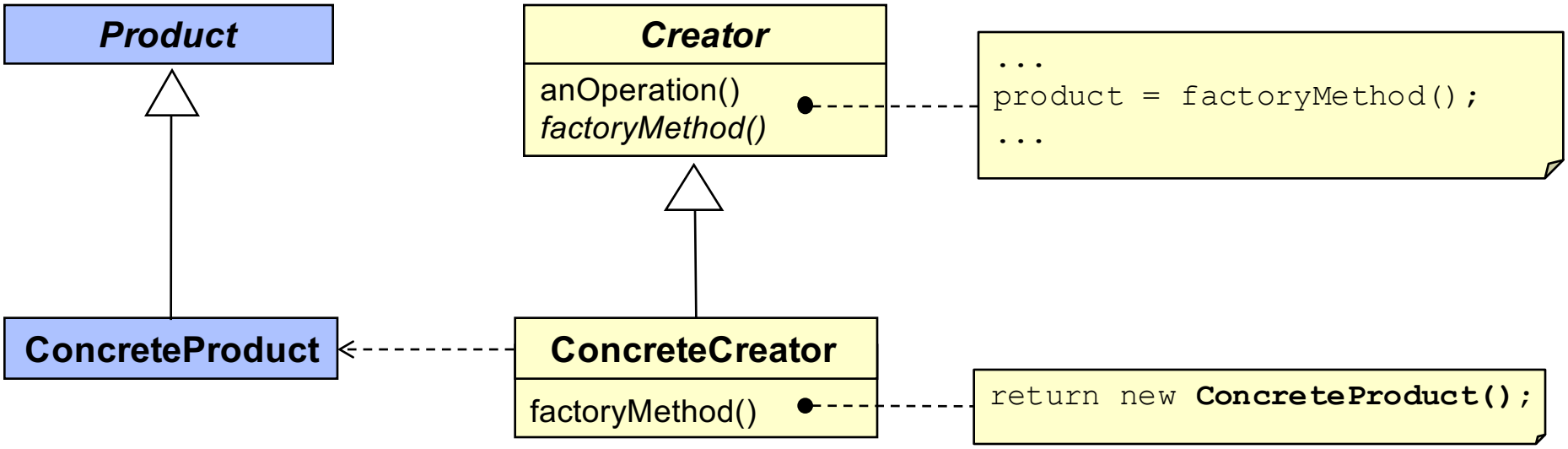
  Ermögliche es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.

- **Motivation**:
  - Entscheidung wann ein Objekt erstellt wird, wird im Klienten getroffen
  - Welche Klasse genau erzeugt wird entscheidet die Factory-Method der Unterklasse

- **Anwendbarkeit:**
  - Klient kennt die zu erzeugenden Objekte noch nicht im Voraus
  - Konstruktionsprozess der Objekte soll in Unterklassen gekapselt werden

- **Konsequenzen**:
  - Code wird abstrakter und wiederverwendbarer
  - Möglichkeit der Verbindung paralleler Klassenhierarchien
  - Einführung von Subklassen die nur der Objekterzeugung dienen

Factory Method ▸ Schema



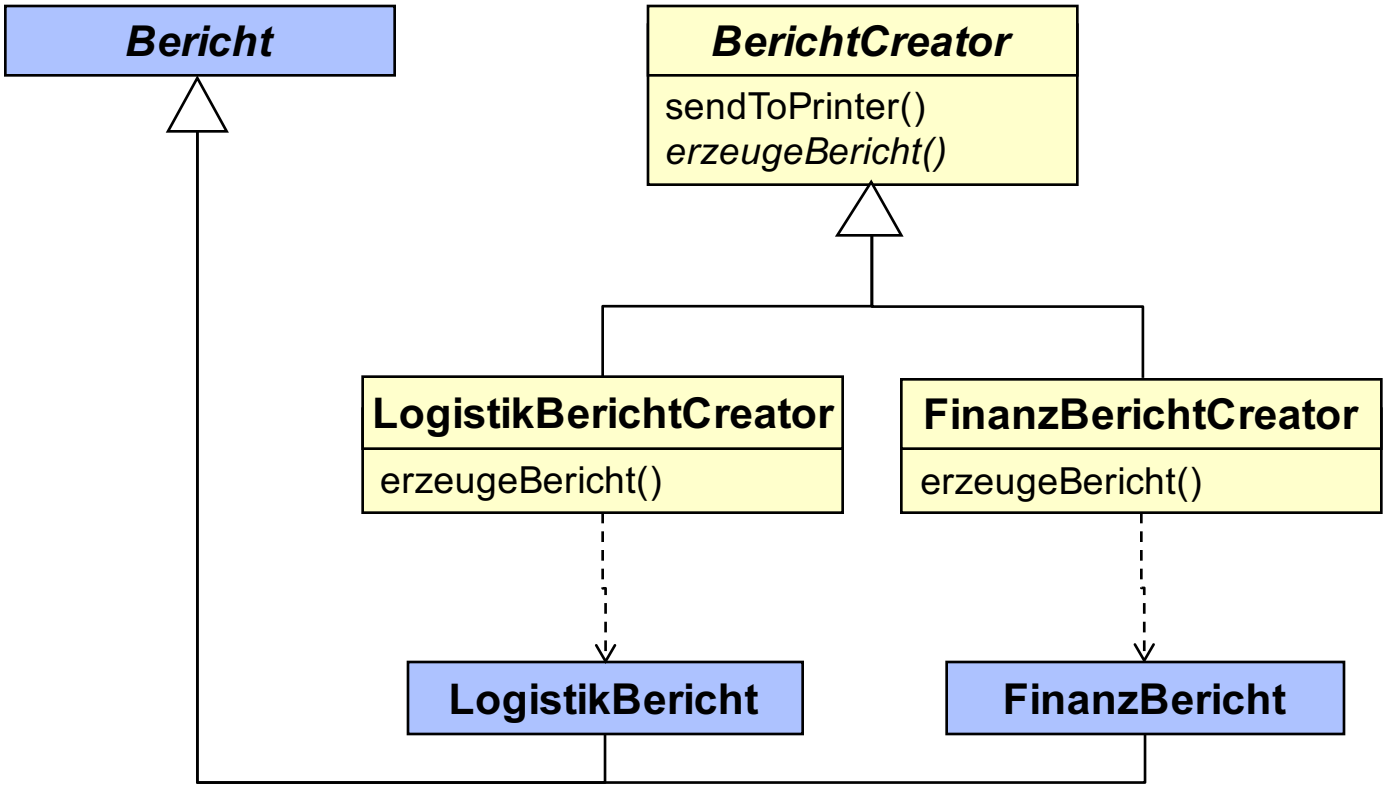
Factory Method ▸ Beispiel

- **Szenario:**
  - Eine Verwaltungssoftware für Unternehmen soll in der Lage sein die gesamte Buchhaltung, Logistik, etc… zu kontrollieren. In regelmäßigen Abständen müssen Statusberichte an die Unternehmensleitung, das Finanzamt, und andere Stellen ausgegeben werden. Jede Abteilung, und jede Behörde erwartet dabei unterschiedliche Informationen und Formatierung.
  - Sie sollen nun alle Statusberichte der unterschiedlichen Anforderungen realisieren. Dabei muss berücksichtigt werden, dass für jede Art Bericht unterschiedliche Informationen eingeholt werden müssen, und an die Formatierung angepasst werden müssen.

- **Lösung:**
  - Für jede Art Bericht wird ein eigener *ConcreteCreator* angelegt, der in der Lage ist die richtigen Informationen einzuholen, und den Bericht korrekt anzufertigen.
  - Die konkret zu erzeugende Berichtform / *ConcreteCreator* wird nach aktuellen Anforderungen durch den Benutzer ausgewählt.

- **Überlegung:**
  - Was muss getan werden um eine neue Berichtform hinzuzufügen?
  - Wie groß ist der Aufwand eine Form wieder zu entfernen?

Factory Method ▸ Beispiel



Abstract Factory

Abstract Factory ▸ Übersicht

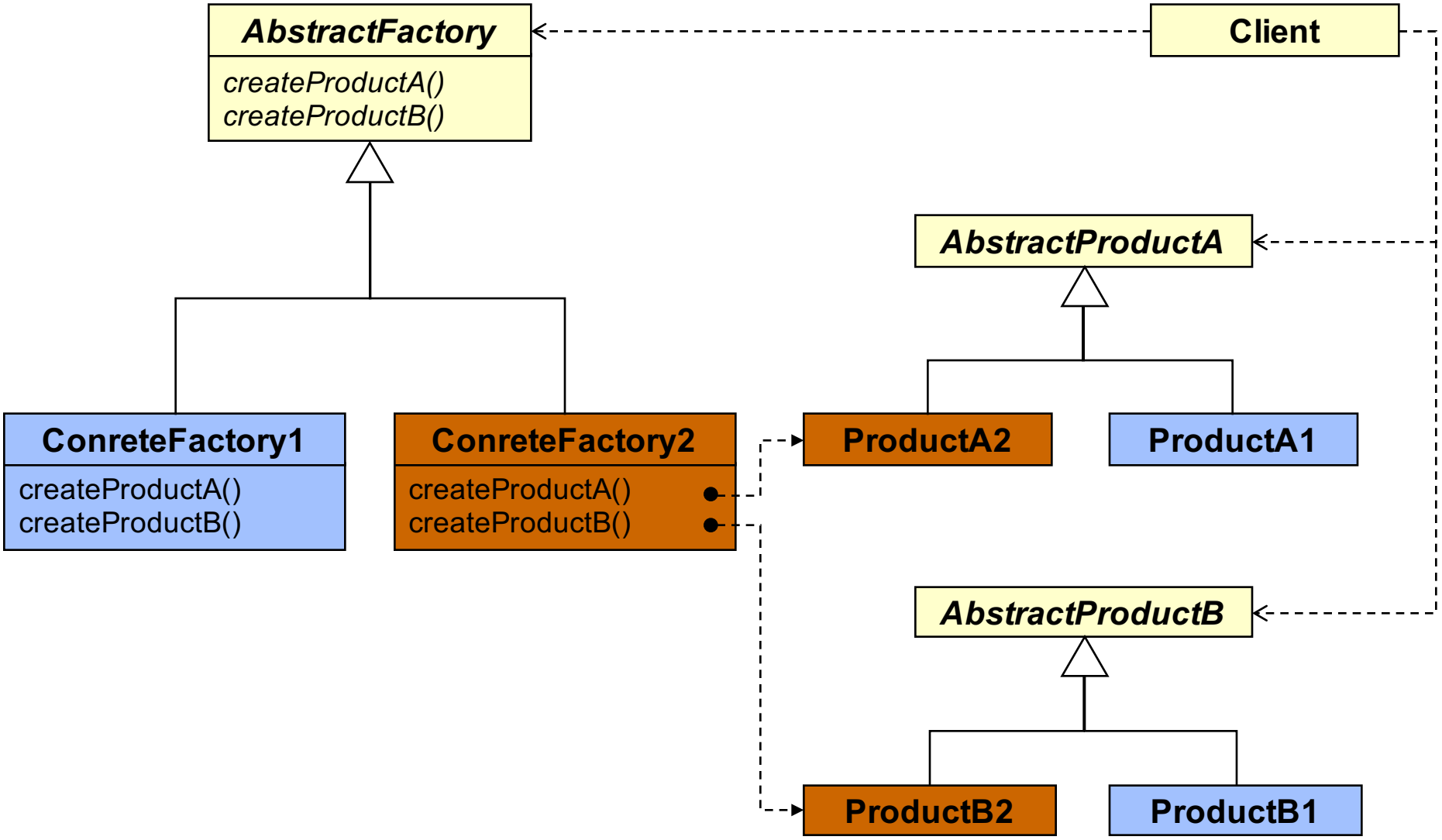
- **Absicht**:
  - Zusammengehörige Objekte verwandter Typen erzeugen
  - ohne deren Klassenzugehörigkeiten

    fest zu kodieren

- **Anwendbarkeit:**
  - Unterschiedliche Objekte müssen immer im korrekten Zusammenhang erstellt werden (Produktfamilien)

- **Konsequenzen**:
  - Einfacher Austausch von Produktfamilien
  - Konsistenzsicherung von Produktfamilien
  - Schwierige Erweiterung um neue Produktarten

Abstract Factory ▸ Schema



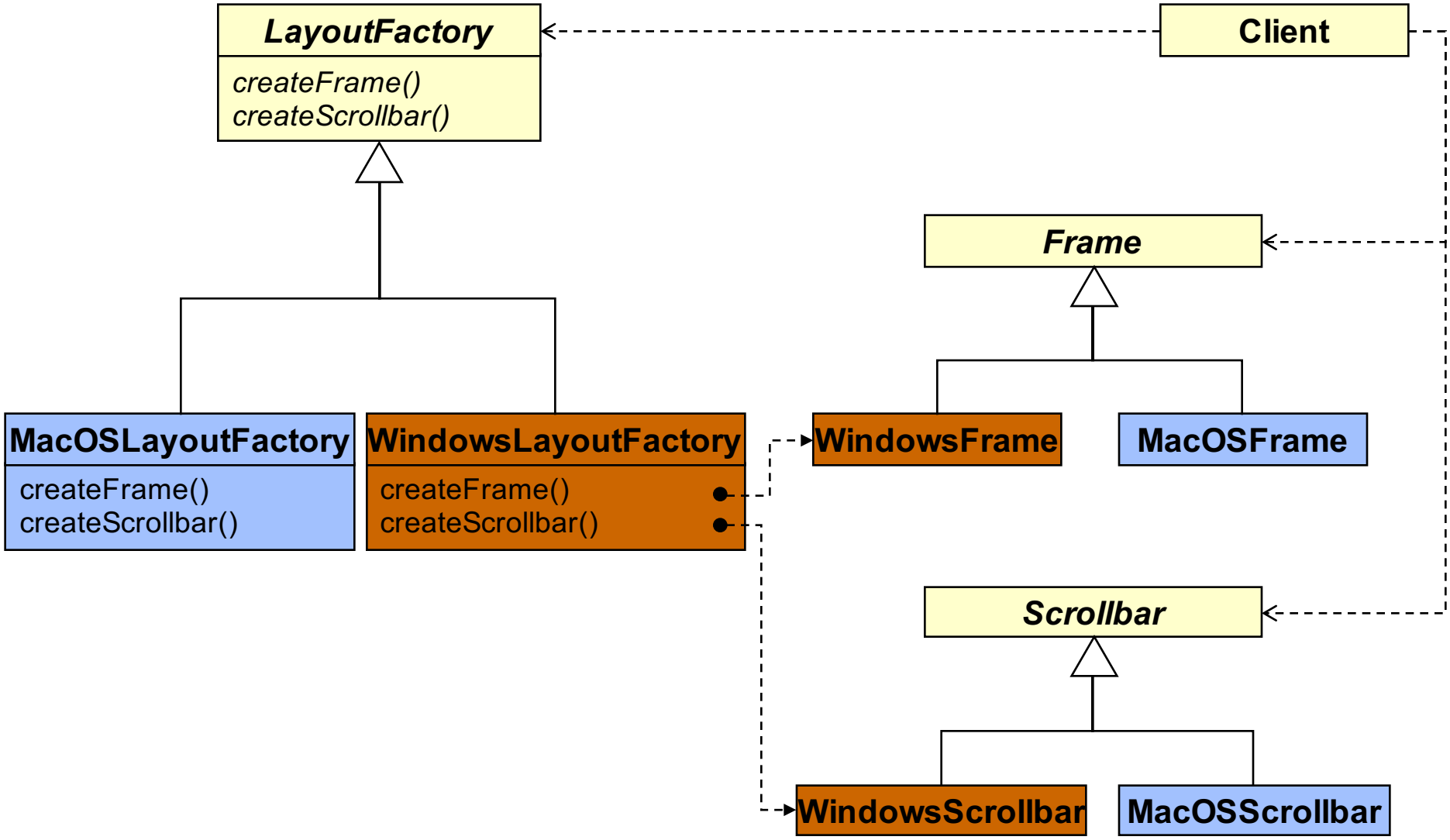
Abstract Factory ▸ Beispiel

- **Szenario:**
  - Java erlaubt es bei Fenstern (*Frames*) ein *Look&Feel* (GUI-Layout) festzulegen. Wird etwa das System-Look&Feel verwendet, werden die Fensterelemente alle im Design des laufenden Betriebssystems angezeigt. Das Java-Look&Feel verwendet hingegen plattformübergreifend das Java-Design.
  - Dabei muss sicher gestellt werden, dass bei der Berechnung eines Fensters die GUI-Elemente wirklich konsistent zueinander sind (z.B. nicht hier und da Windows-Elemente im MacOS Fenster)

- **Lösung:**
  - Die einzelnen GUI-Elemente eines Layouts werden jeweils von einer eigenen Factory erzeugt.
  - Die Factories selber sind Unterklassen einer Abstract Factory. Bei Laufzeit kann dann entschieden werden welche Factory verwendet = welche Layout-Elemente erzeugt werden.

- **Überlegung:**
  - Was muss getan werden um ein neues Fensterlayout hinzuzufügen?
  - Wie groß ist der Aufwand ein Layout wieder zu entfernen?

Abstract Factory ▸ Beispiel

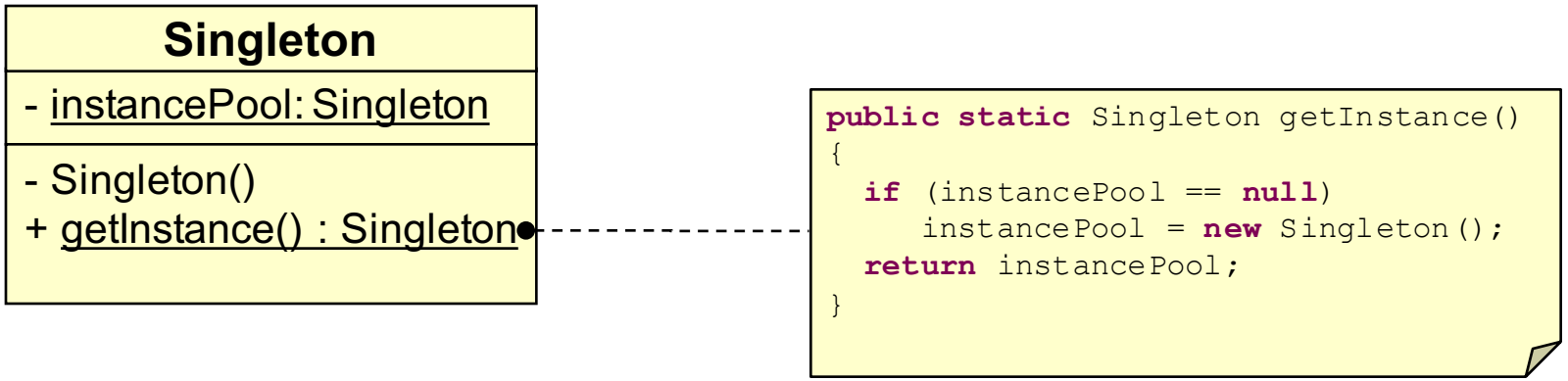


Singleton

Singleton ▸ Übersicht

- **Absicht:**
  - Sicherstellen das es nur eine bestimmte Anzahl von Instanzen einer Klasse gibt
  - Einen globalen Zugriffspunkt zur Verfügung stellen
- **Motivation:**
  - Begrenzte Ressourcen (z.B. auf mobilen Geräten)
  - Teure Objekterzeugung durch „Object Pool“ vermeiden
    - ⇨ z.B. 1000 Enterprise Java Beans vorhalten, nach Nutzung zurück in den Pool
- **Implementierung:**
  - In Java nur private Konstruktoren verwenden
  - In Java auch privaten parameterlosen Konstruktor definieren
    - ⇨ sonst erzeugt der Compiler automatisch einen public Konstruktor
  - Zugriff auf Instanzen über Klassenmethode (static)

Singleton ▸ Schema



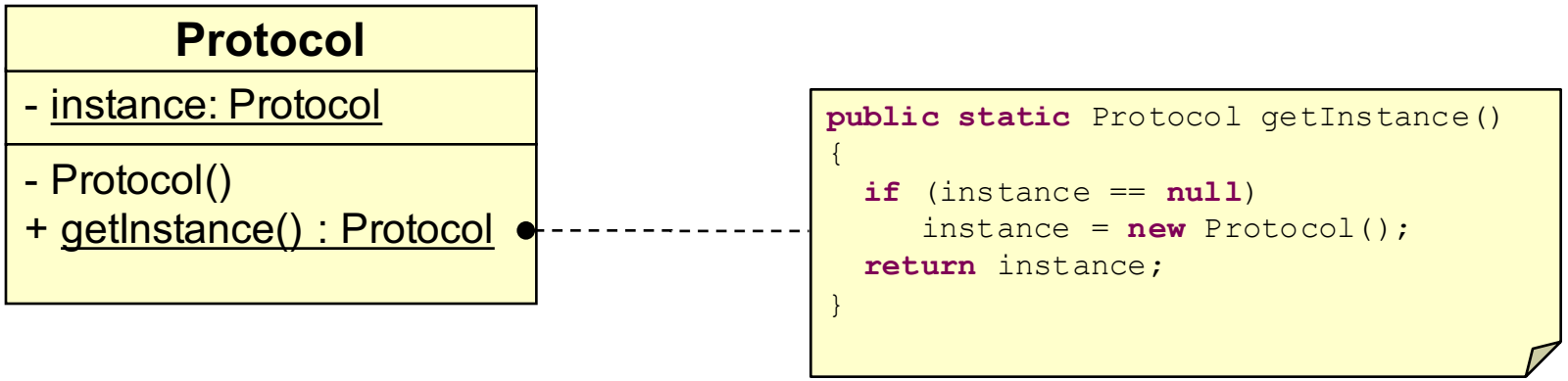
- Hier: Singleton mit genau einer Instanz (in *instancePool* vorgehalten)
- -Singleton() definiert den privaten Konstruktor
- Zugriff auf die Instanz mittels der Klassenmethode *getInstance()*

Singleton ▸ Beispiel

- **Szenario:**
  - Ihr Programm soll bestimmte Aktivitäten in eine Protokolldatei schreiben, die durch eine Klasse *Protocol* realisiert ist.
  - Es soll nun sichergestellt werden, dass alle Komponenten ihres Programms genau dieselbe Instanz dieser Klasse benutzen, um Protokolleinträge hinzuzufügen. Es muss daher sichergestellt werden, dass immer nur genau eine Instanz dieser Klasse existiert.

- **Lösung:**
  - Die Klasse Protocol wird als Singleton realisiert. Klienten, die dieses Objekt verwenden wollen, können über die Methode *getInstance()* die einzig gültige Instanz bekommen.

Singleton ▸ Beispiel



Observer

Observer ▸ Übersicht

- **Absicht:**
  - Lose 1-n-Beziehung von Objekten
  - Wenn das Eine seinen Zustand ändert, werden Abhängende aktualisiert

- **Konsequenzen:**
  - Unabhängigkeit
    - ⇨ Beliebig viele Beobachter können hinzugefügt werden, ohne andere Klassen zu verändern.
    - ⇨ Konkrete Subjekte können unabhängig voneinander und von konkreten Beobachtern variiert und wiederverwendet werden.
  - „Broadcast“-Nachrichten
    - ⇨ Subjekt benachrichtigt alle angemeldeten Beobachter
    - ⇨ Beobachter entscheiden, ob sie Nachrichten behandeln oder ignorieren
  - Unerwartete Aktualisierungen
    - ⇨ Kleine Zustandsänderungen des Subjekts können komplexe Folgen haben.
    - ⇨ Auch uninteressante Zwischenzustände können unnötige Aktualisierungen auslösen.

Observer ▸ Push –/Pull-Modell

- Pull:

  Beobachter holen sich die Informationen vom Subjekt

  \+ Geringere Kopplung zwischen Subjekt und Beobachter

  – Berechnungen werden häufiger durchgeführt

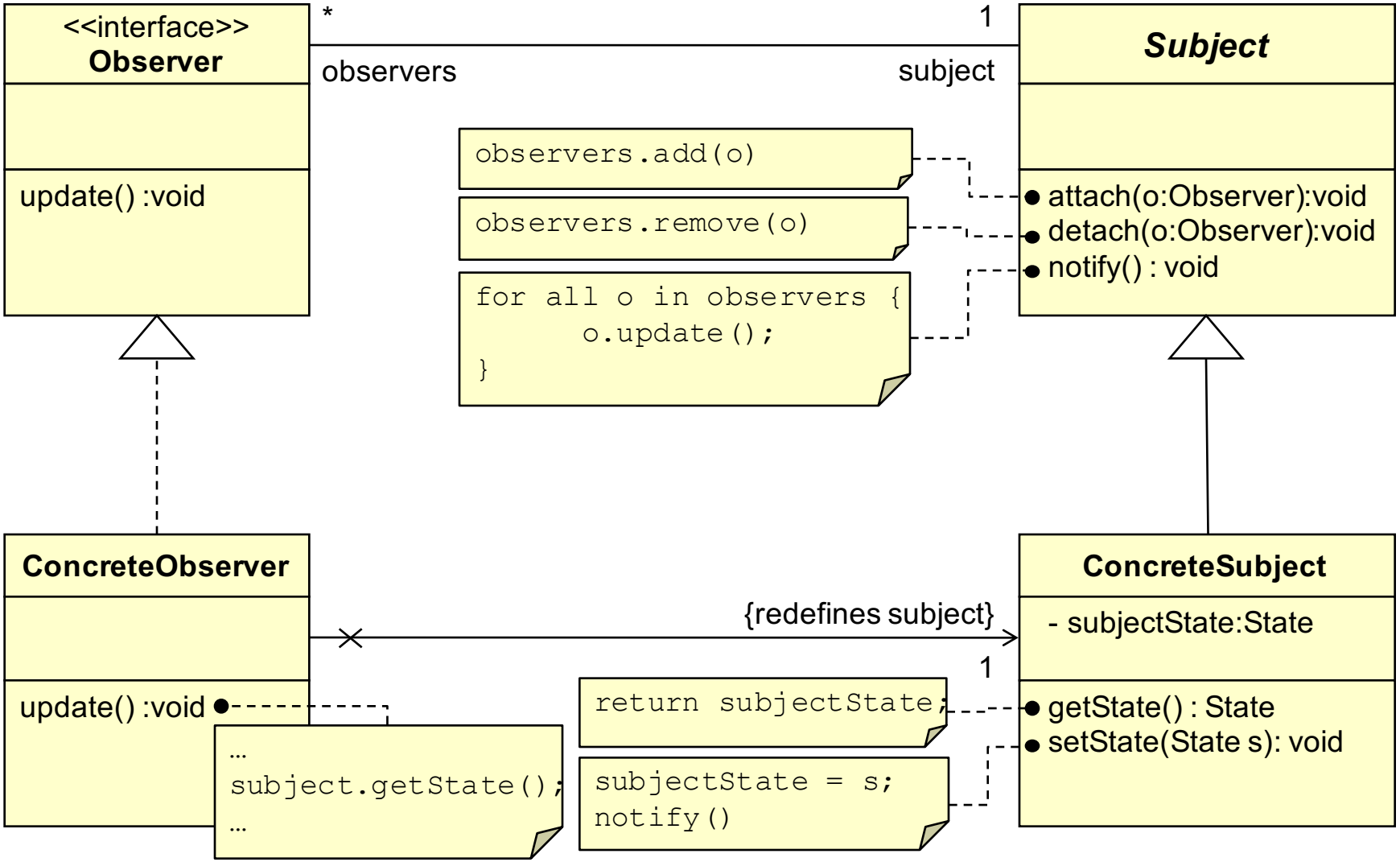
- Push:

  Subjekt übergibt per „update(State)“ Änderungen

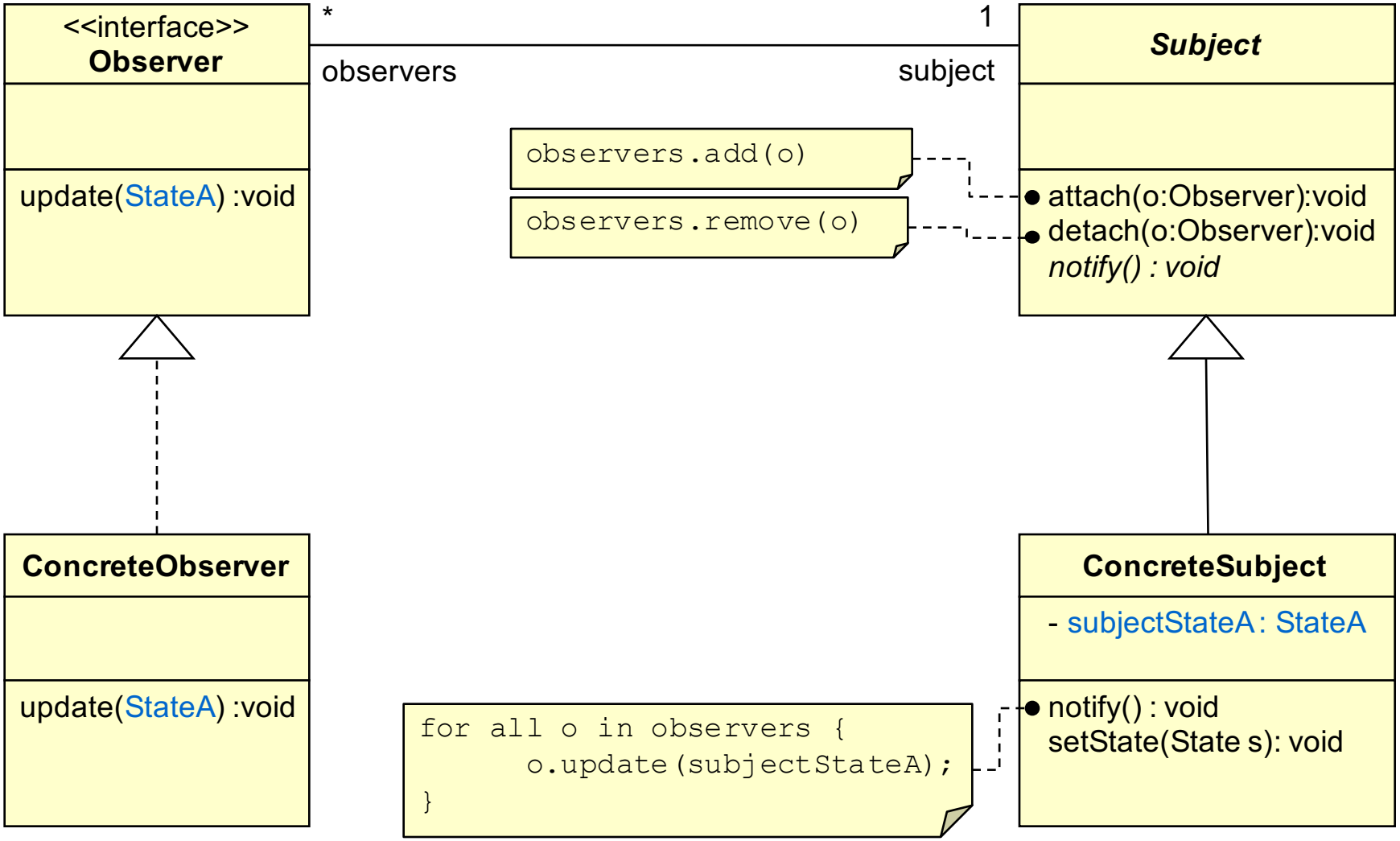
  \+ Rückaufrufe werden seltener durchgeführt

  – Beobachter sind Abhängig von Parametertypen

Observer ▸ Pull Schema



Observer ▸ Push Schema



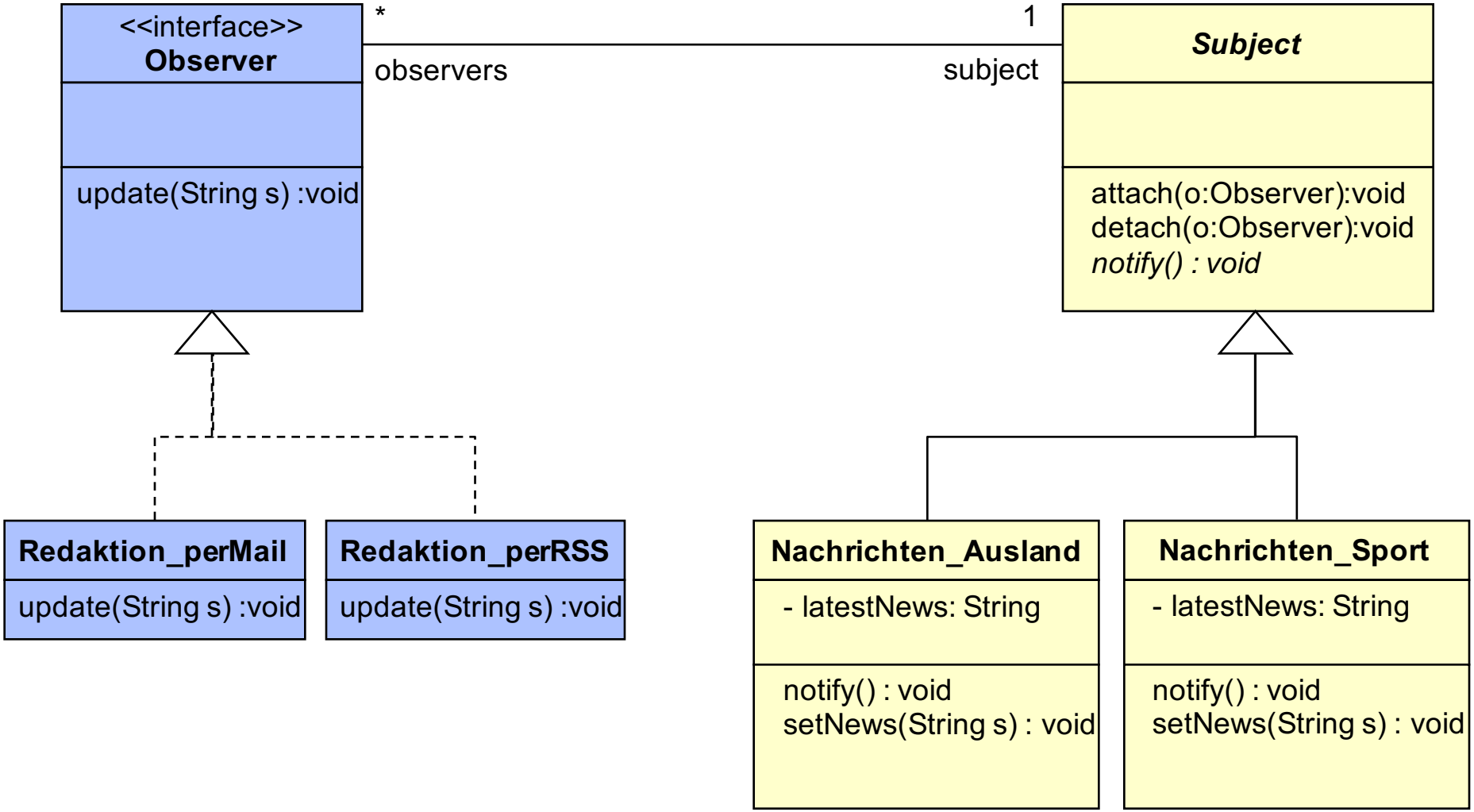
Observer ▸ Beispiel

- **Szenario:**
  - Eine Nachrichtenagentur verfügt über Schnittstellen zu diversen Zeitungen, Internetmedien und Fernsehsendern. Sobald eine neue Nachricht eingeht, soll diese umgehend an alle Redaktionen weitergeleitet werden.
  - Für jeden Interessenten gibt es ein Objekt, das die Weitergabe zu der jeweiligen Redaktion realisiert (per Mail, RSS, Twitter, etc…).
  - Nachrichten sind üblicherweise in Kategorien (Politik, Sport, …) geordnet. Die empfangenden Redaktionen wollen dabei nur Nachrichten bestimmter Kategorien bekommen.

- **Lösung:**
  - Es wird ein Interface *Observable* zur Verfügung gestellt, das die verschiedenen Nachrichtenkanäle für die einzelnen Kategorien implementieren.
  - Die Klassen, die die Kommunikation zu den Redaktionen realisieren, implementieren eine *Observer*-Schnittstelle, mit der sie beliebige Observables/Nachrichtenkanäle „abonnieren“ können.

- **Überlegung:**
  - Was muss getan werden um einen neuen Nachrichtenkanal / Empfänger hinzuzufügen?
  - Wie groß ist der Aufwand einen Nachrichtenkanal wieder „abzubestellen“?

Observer ▸ Beispiel (Push)

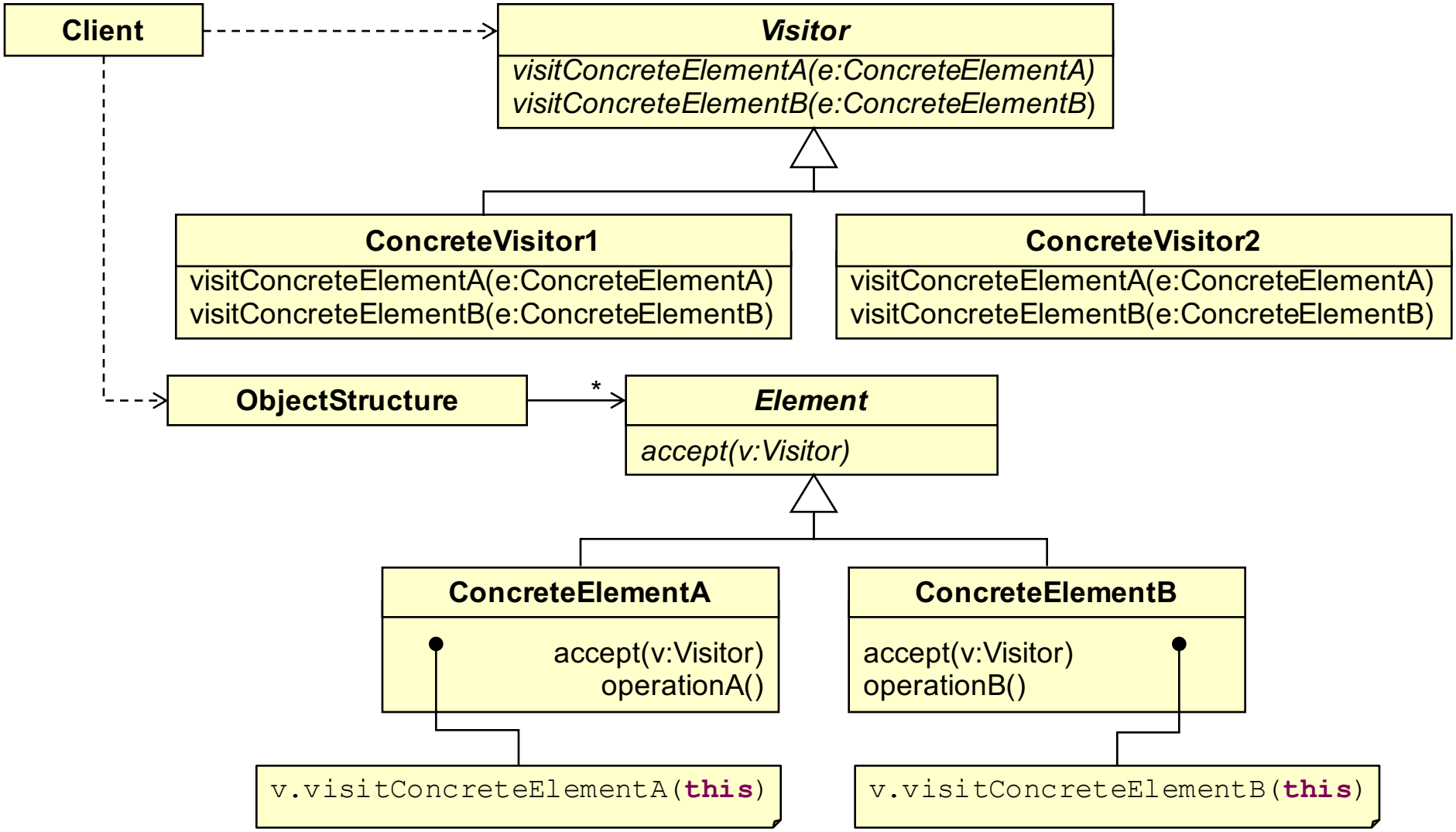


Visitor

Visitor ▸ Übersicht

- **Kontext:**
  - Eine Objektstruktur enthält Elemente unterschiedlicher Klassen
- **Absicht:**
  - Repräsentation von Operationen auf Elementen einer Objektstruktur.
  - Neue Operationen definieren, ohne die Klassen der Objektstruktur selbst zu ändern
- **Motivation:**
  - Vermeiden Code einer Operation über mehrere Klassen zu verteilen.
  - Hinzufügen oder Ändern einer Operation erfordert Änderung der gesamten Objektstruktur .
  - Neue Operationen sollen nicht die Klassen der Objektstruktur aufblähen. Die Realisierung einer Operation soll für alle dieser Klassen zusammengehalten werden.
- **Konsequenzen:**
  - Hinzufügen von Funktionalität einfach durch neue Visitor-Klasse
  - Hinzufügen neuer Elementklasse ist sehr aufwendig

Visitor ▸ Schema



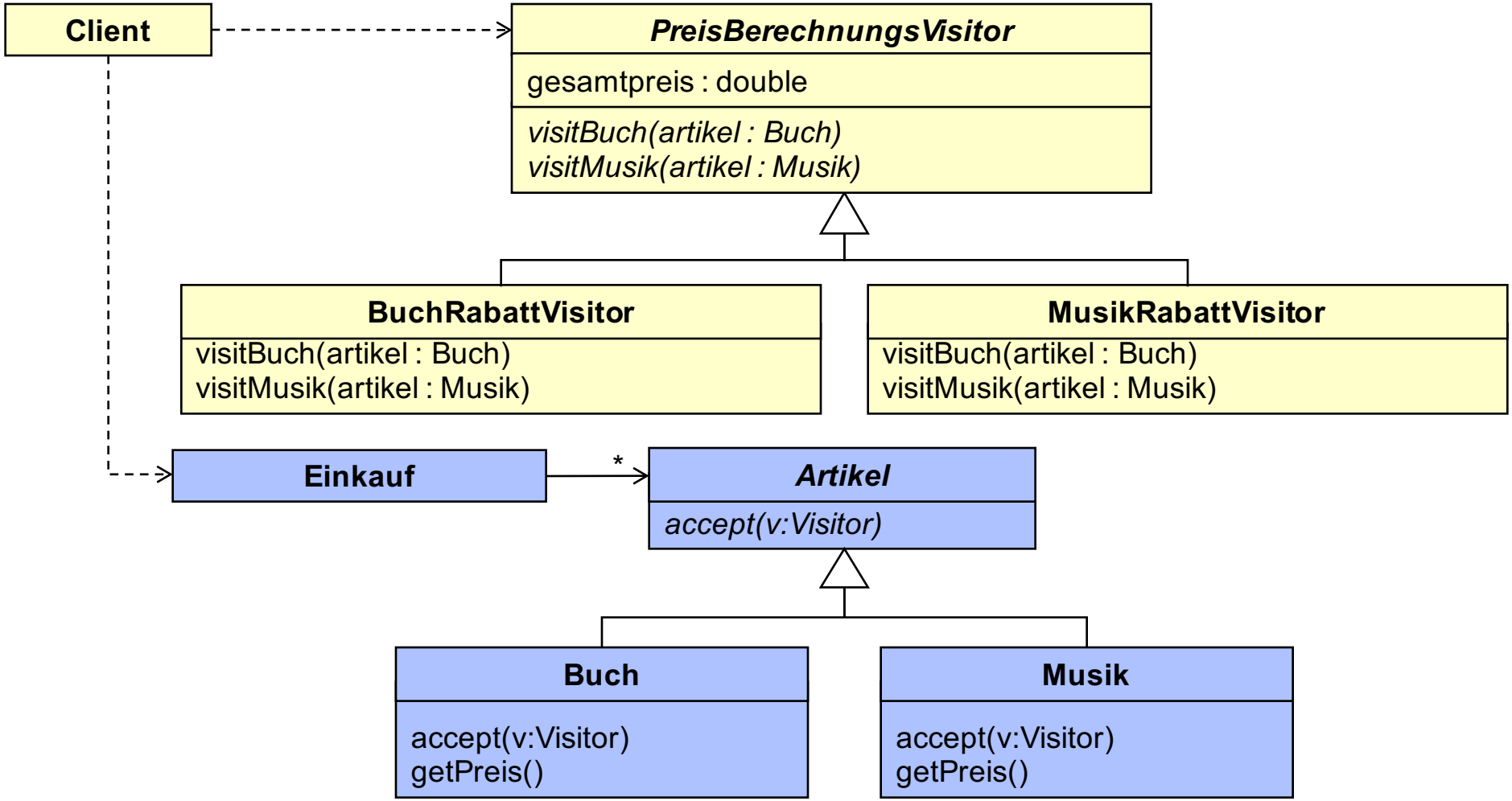
Visitor ▸ Beispiel

- **Szenario:**
  - Ein Online-Versandhändler bietet Artikel aus nahezu allen denkbaren Bereichen an (Bücher, Musik, Elektronik, ….). Um Kunden zu Einkäufen zu motivieren, werden gelegentlich Gutscheine verschenkt, die jedoch immer nur für eine bestimmte Art von Artikeln gelten (z.B. 10% auf Bücher und Filme).
  - Gibt ein Kunde beim Abschluss seines Einkaufes einen Gutscheincode ein, soll der Rabatt genau auf die betreffenden Artikel des Einkaufs angerechnet werden.
  - Die Einzelartikel sind dabei Unterklassen ihrer jeweiligen Kategorie.

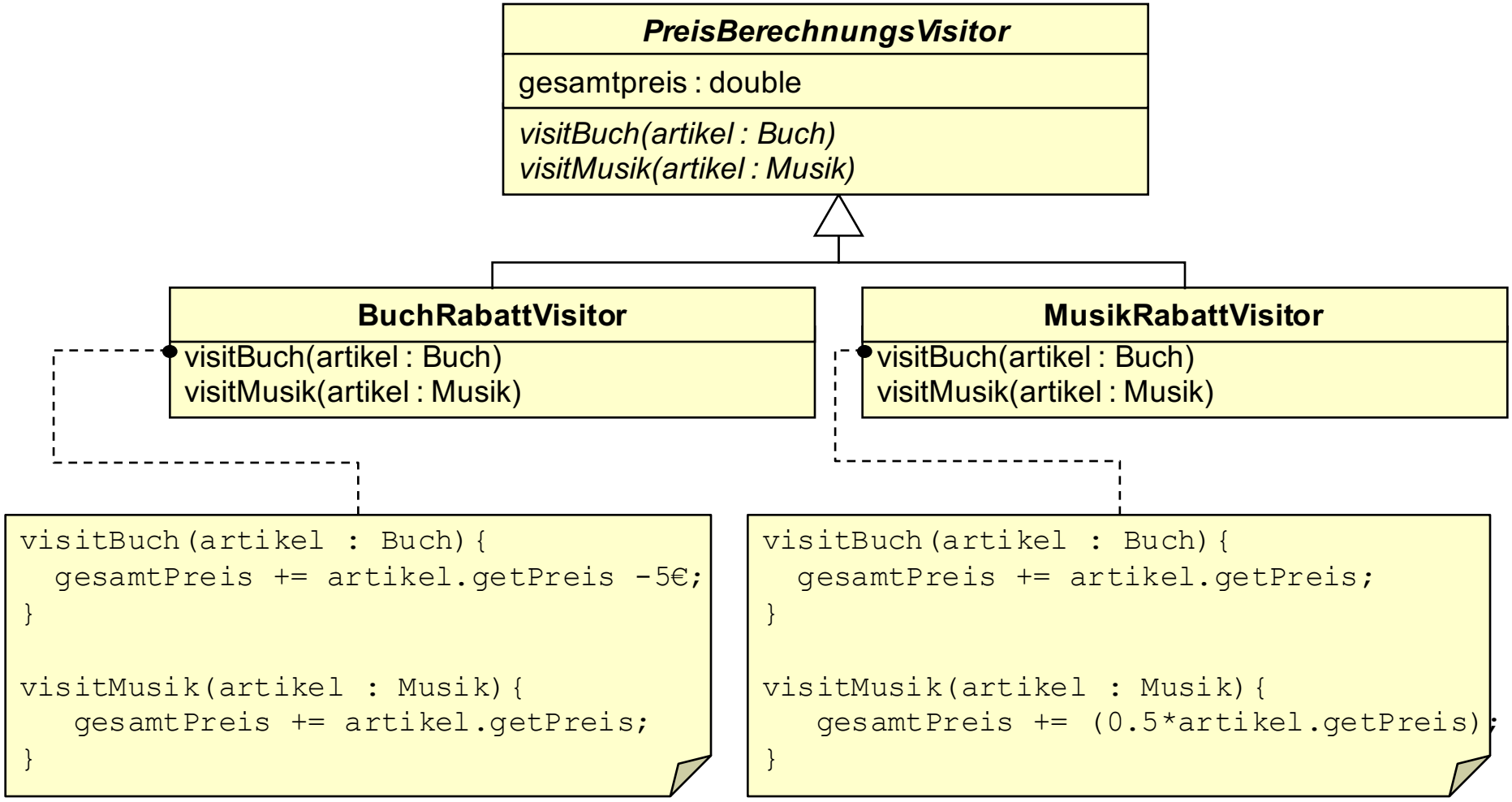
- **Lösung:**
  - Die Kategorie-Oberklassen bieten eine Methode für die Annahme von Visitors an.
  - Ein Visitor läuft über alle Artikel des Einkaufs, die Artikel rufen die korrekte accept-Methode auf, und der Visitor berechnet so den Gesamtpreis unter Berücksichtigung der Artikelkategorien.

- **Überlegung:**
  - Wie aufwendig ist es eine neue Form von Rabatt einzuführen (z.B. 5% auf Musik und 10% auf DVDs)?
  - Wie aufwendig ist es, die verschiedenen Berechnungsvisitors auszutauschen (Angebot gilt nur bis zum 1.1.Februar)?

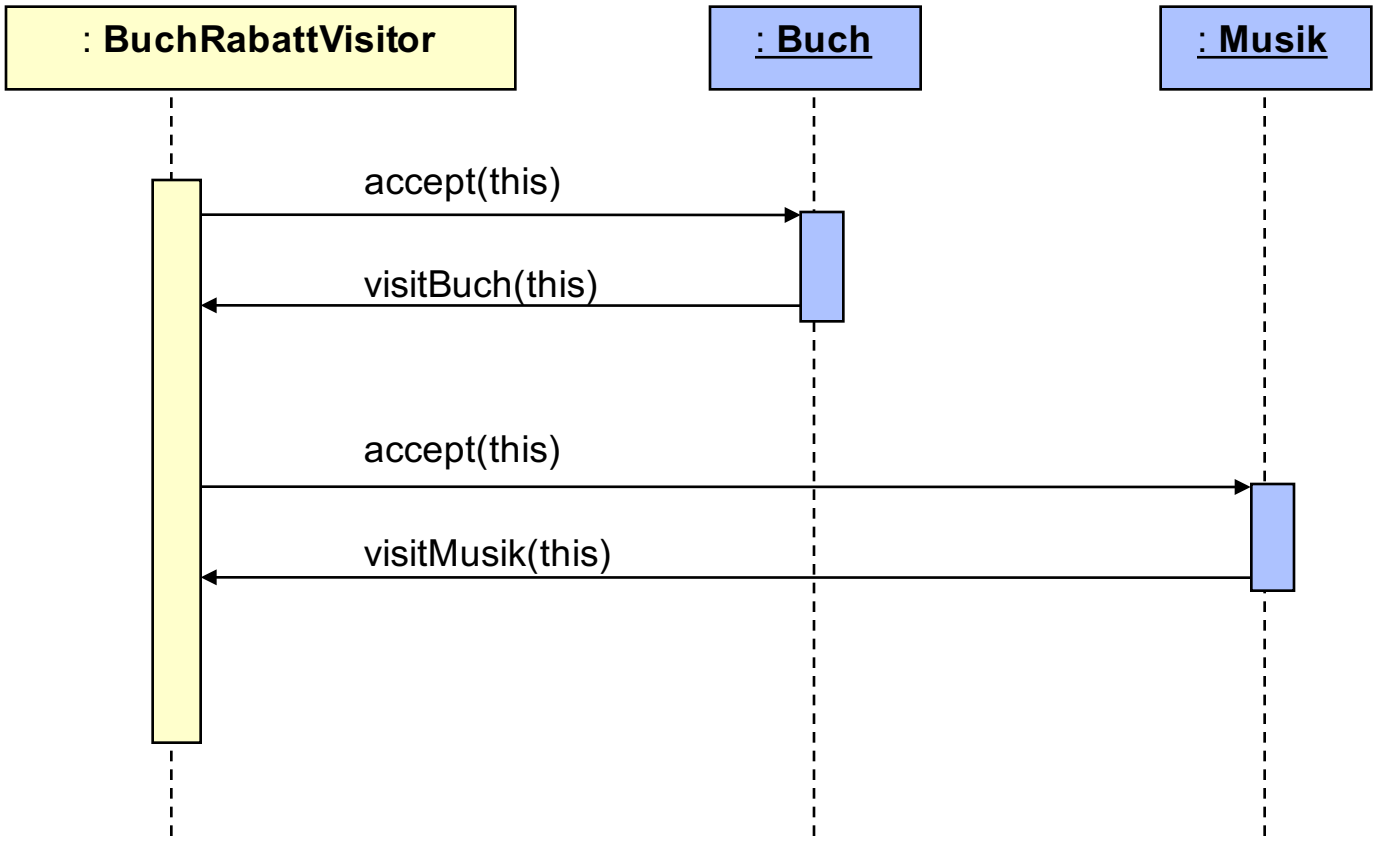
Visitor ▸ Beispiel



Visitor ▸ Beispiel



Visitor ▸ Beispiel



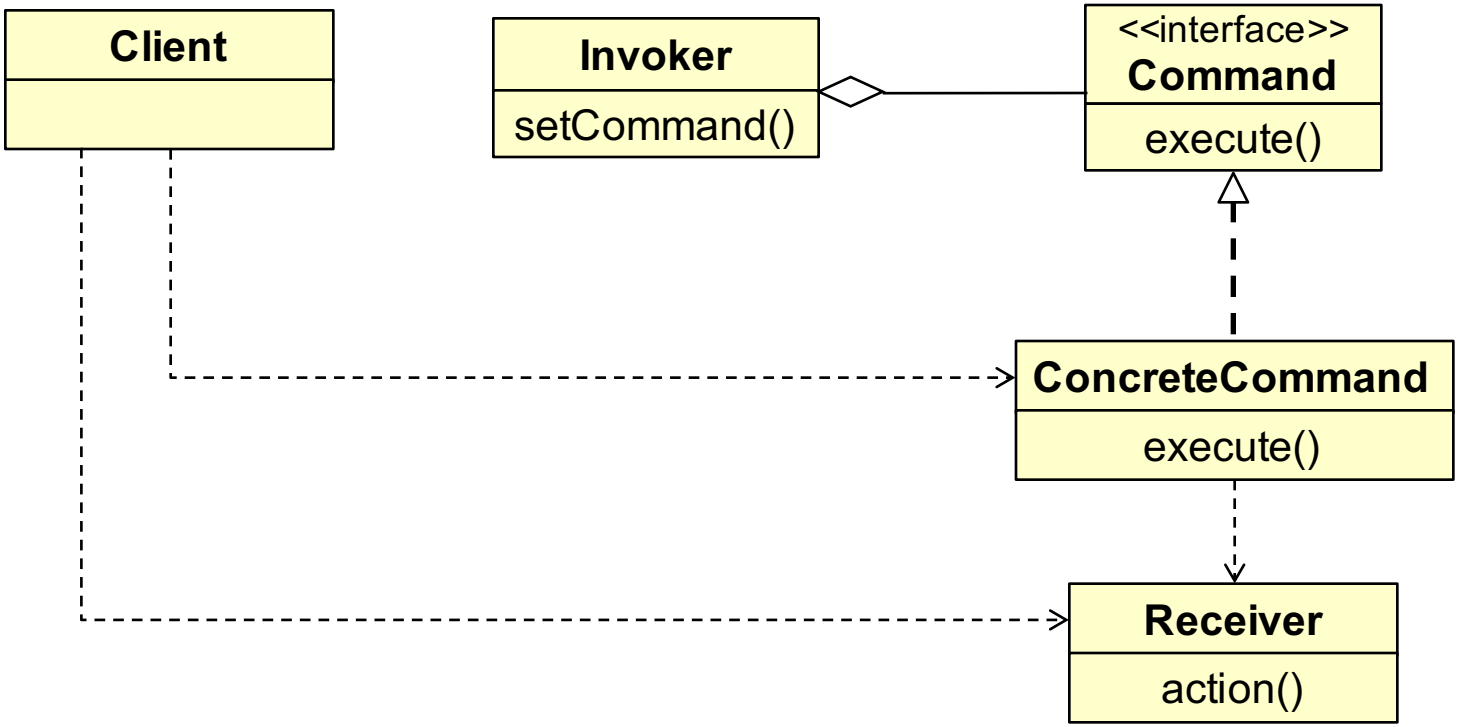
- 1. Der Visitor iteriert über die Objekte, wobei er nur die Oberklasse *Artikel* „sieht“
- 2. Die Artikel-Objekte akzeptieren den Visitor, und rufen die Methode des Visitors auf, die für sie geeignet ist.
- 3. Der Visitor führt dann z.B. in der visitBuch-Methode die Buch-spezifische Operation aus.

Command

Command ▸ Übersicht

- **Absicht:**
  - Kapselung einer Operations-Anfrage als Objekt
  - Möglichkeit Klienten mit verschiedenen Anfragen zu parametrisieren
- **Motivation:**
  - Einheitlicher Code an allen Aufrufstellen, trotzdem verschiedene Effekte
  - Aktionen sind kontext-sensitiv und können dynamisch ausgetauscht werden
- **Konsequenzen:**
  - Entkopplung von Aufrufer und Ausführer einer Operation
  - Mehrere Commands können zu weiteren Objekten zusammengefasst werden (Makros)

Command ▸ Schema



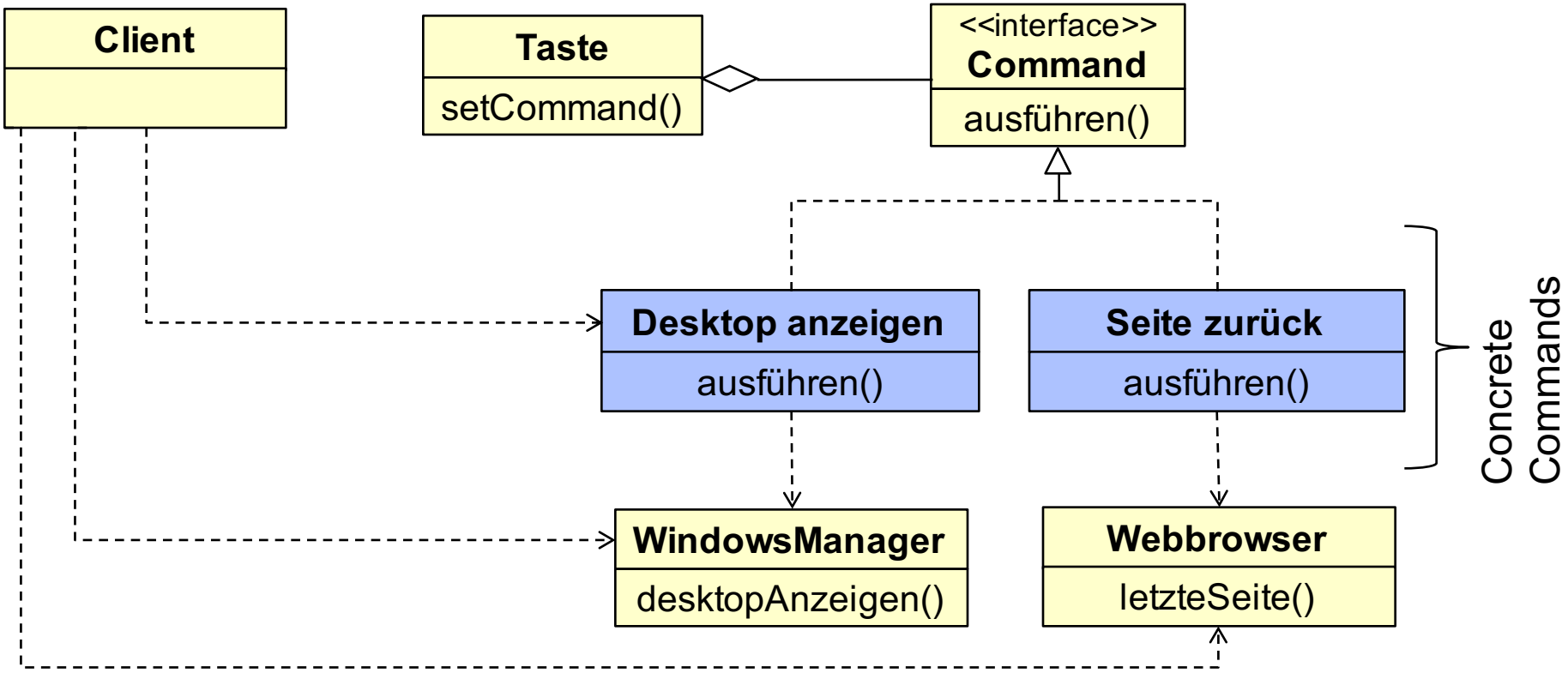
Command ▸ Beispiel

- **Szenario:**
  - Viele Tastaturen und Mäuse bieten heutzutage eine Reihe Zusatztasten an, die vom Benutzer frei mit Funktionen belegt werden können. Dazu bietet der Hersteller eine Reihe von vorgefertigten Funktionen an, mit denen die Tasten über eine Software belegt werden können.
  - Die Zuordnung der Funktionen zu den Tasten soll während der Laufzeit geändert werden können.

- **Lösung:**
  - Die Funktionen werden als *ConcreteCommands* zur Laufzeit den Tasten (*Invoker*) zugeordnet

- **Überlegung:**
  - Was muss getan werden um eine neue Funktion hinzuzufügen?
  - Wie groß ist der Aufwand eine Funktion wieder zu entfernen?

Command ▸ Beispiel

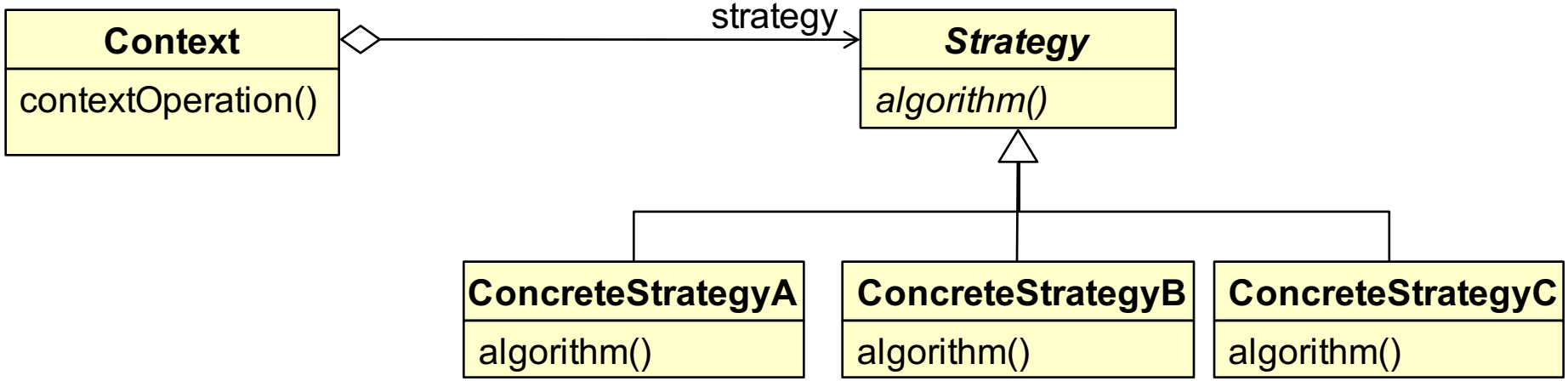


Strategy

Strategy ▸ Übersicht

- **Absicht:**
  - Kapselung einer Familie von Algorithmen mit der Möglichkeit, sie beliebig auszutauschen.
  - Ermöglicht es den Algorithmus unabhängig von nutzenden Klienten zu variieren
- **Anwendbarkeit:**
  - Verhalten ist abhängig von äußeren Randbedingungen
  - Verschiedene Varianten eines Algorithmus werden benötigt
  - Kapselung von Daten eines komplexen Algorithmus
- **Konsequenzen:**
  - Alternative führt zu einer Unterklassenbildung
    - ⇨ Erhöht Klassenanzahl
  - Strategien entfernen Bedingungsanweisungen
  - Klienten müssen Strategien kennen

Strategy ▸ Schema



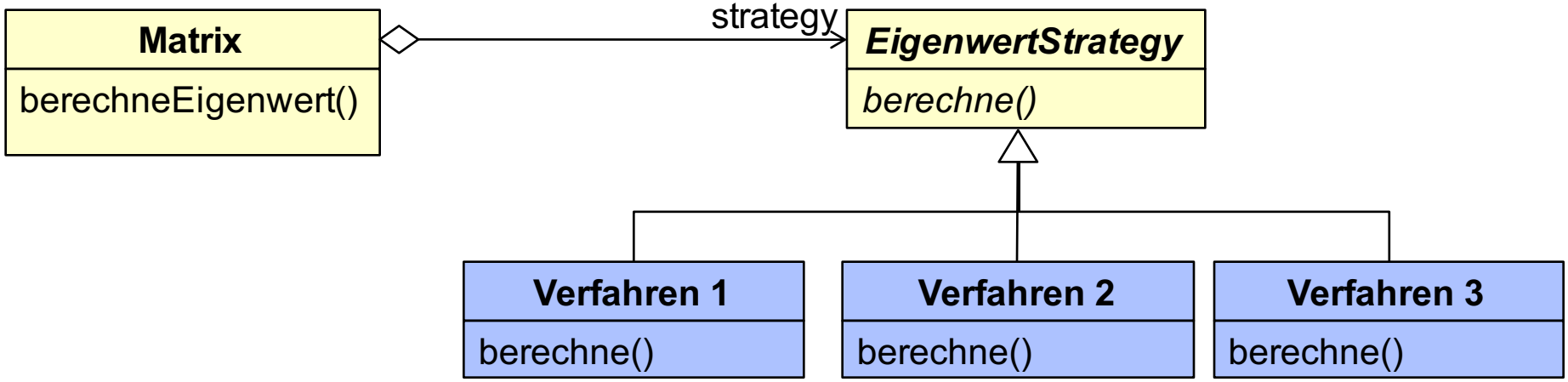
Strategy ▸ Beispiel

- **Szenario:**
  - Für die Entwicklung einer Bibliothek für mathematische Berechnungen wollen die Entwickler die Berechnung von Eigenwerten einer Matrix implementieren. Die Kollegen der Mathematik teilen ihnen mit, dass es je nach Beschaffenheit einer Matrix unterschiedliche Verfahren zur Berechnung der Eigenwerte gibt.
  - Um immer die optimale Variante verwenden zu können, soll das genaue Berechnungsverfahren erst bei Laufzeit ausgewählt werden.

- **Lösung:**
  - Die Berechnungsverfahren werden als *ConcreteStrategy* realisiert.
  - Die Auswahl der *ConcreteStrategy* erfolgt dann bei Laufzeit, nachdem der Typ der Matrix bestimmt wurde.

- **Überlegung:**
  - Was muss getan werden um ein neues Berechnungsverfahren hinzuzufügen?
  - Wie groß ist der Aufwand ein Verfahren wieder zu entfernen?

Strategy ▸ Beispiel

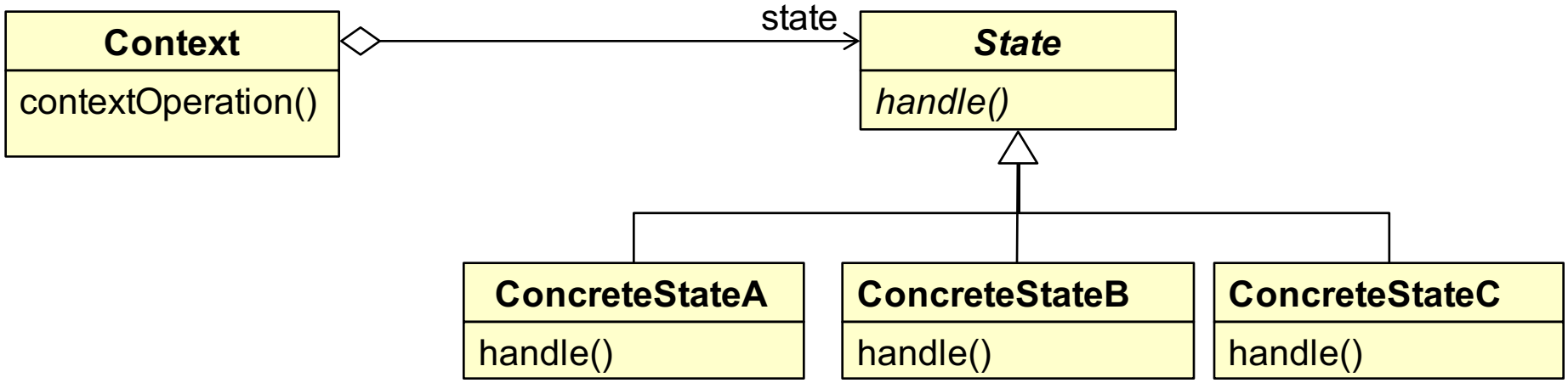


State

State ▸ Übersicht

- **Absicht:**
  - Ermögliche einem Objekt sein Verhalten zu ändern, wenn sich der interne Zustand ändert
- **Anwendbarkeit:**
  - Die Operationen der Klasse besitzen Verhalten das von dem Zustand abhängt
  - Viele Fallunterscheidungen innerhalb der Klasse verhindern, die zustandsabhängig ein Verhalten auswählen sollen
- **Konsequenzen:**
  - Zustandsabhängiges Verhalten wird in eigene Klassenhierarchie ausgelagert
  - Zusammengehörige Methoden werden nach Zuständen getrennt
  - Zustandsübergänge sind explizit
  - Neue Zustände erfordern keine / wenig Änderungen des Kontexts

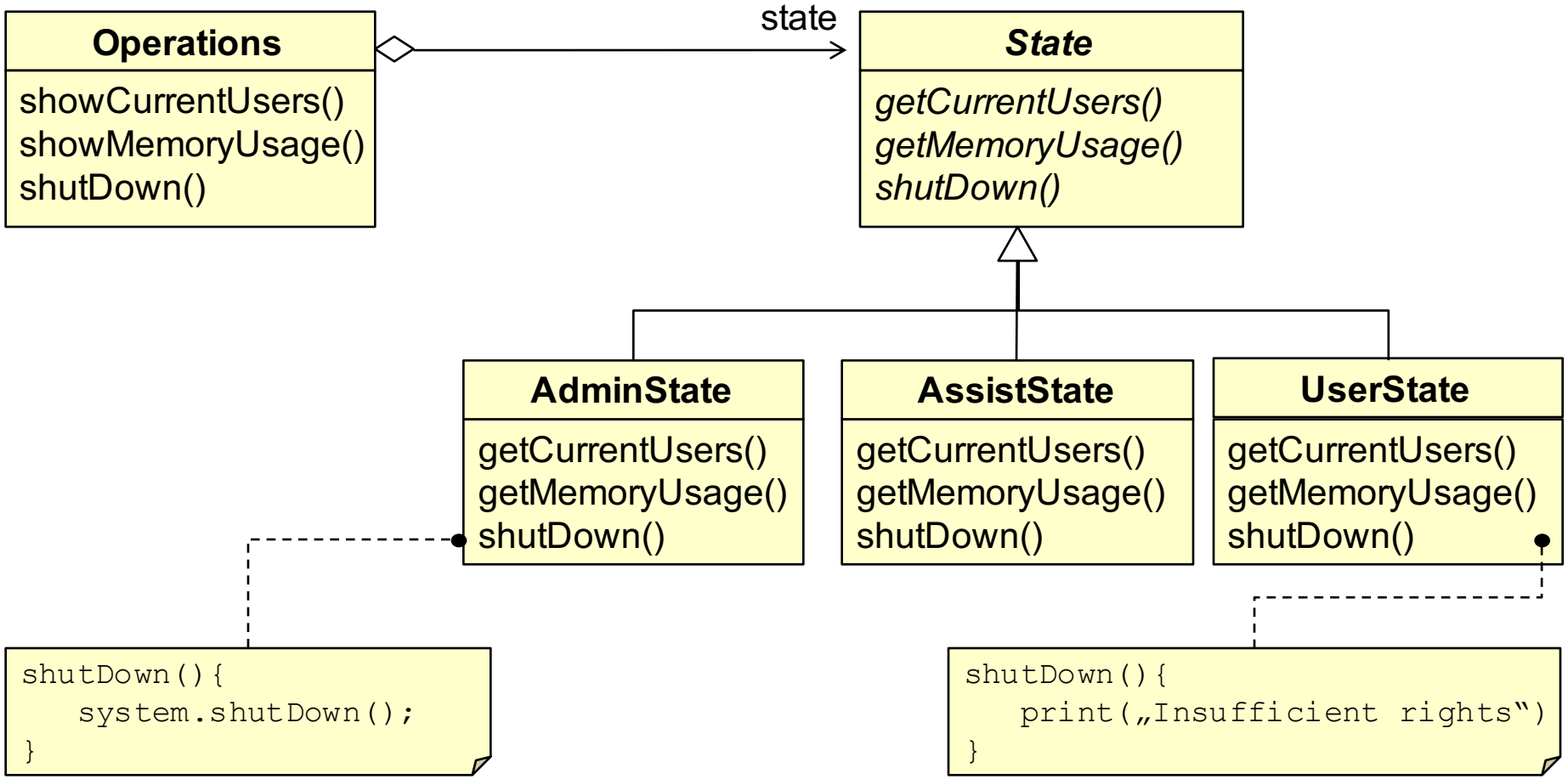
State ▸ Schema



State ▸ Beispiel

- **Szenario:**
  - Administrationssoftware wird oft von mehreren Personen mit unterschiedlichen Zugriffsrechten verwendet. In diesem Fall soll es drei Arten von Benutzern geben: Administrator, Assistenten und normale Benutzer. Je nachdem welcher Gruppe ein Benutzer angehört, erlaubt ihm die Software unterschiedliche Operationen.
  - Sobald sich ein Benutzer einloggt, sollen ihm genau die Operationen bewilligt werden, die seiner Gruppe entsprechen. Alle anderen Operationen sollen abgelehnt werden.
- **Lösung:**
  - Die Software definiert für jede Gruppe einen Zustand, die das eigentliche Verhalten der Operationen realisiert. Je nach Zustand verhalten sich einige Operationen unterschiedlich.
- **Überlegungen:**
  - Wie aufwändig ist es eine neue Benutzergruppe hinzuzufügen?
  - Angenommen eine Operation soll einer Gruppe nun doch verwehrt werden. An wie vielen Stellen muss geändert werden?

State ▸ Beispiel

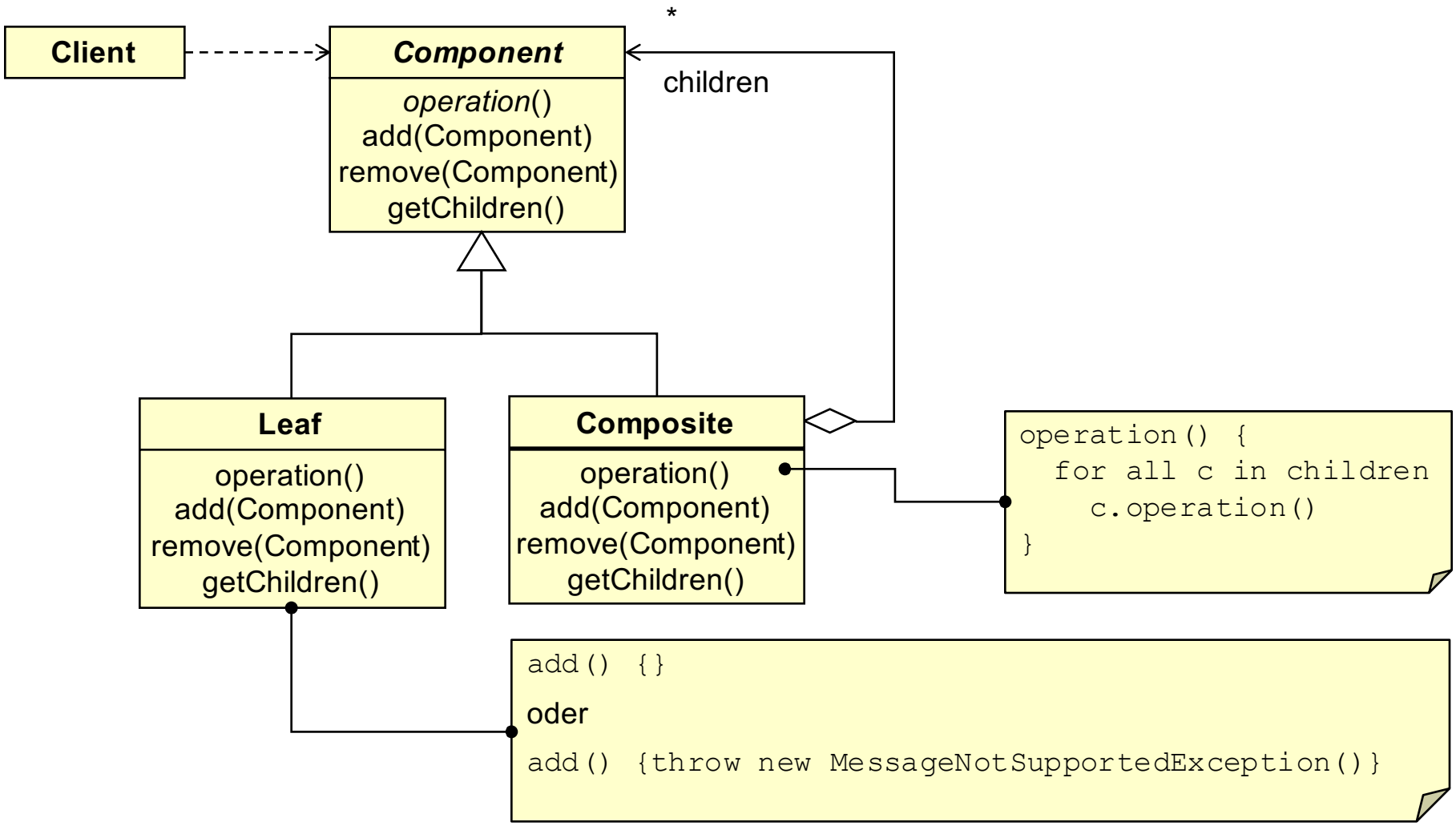


Composite

Composite ▸ Übersicht

- **Absicht:**
  - Darstellung von rekursiven Aggregations-Hierarchien (Baumstrukturen)
  - Ermögliche es Klienten Aggregat und Teile einheitlich zu behandeln
- **Anwendbarkeit:**
  - Repräsentation von rekursiven Teil-Ganzes-Hierarchien
    - ⇨ Objekte können Unterlemente besitzen, die wiederum Unterlemente besitzen,…
  - Verbergen der Unterschiede zwischen einzelnen Objekten und zusammengesetzten Objekten (Aggregaten).
- **Konsequenzen:**
  - Primitive Objekte können zu komplexeren Objekten zusammen gesetzt werden
  - Einfach neue Arten von Komponenten hinzuzufügen

Composite ▸ Schema



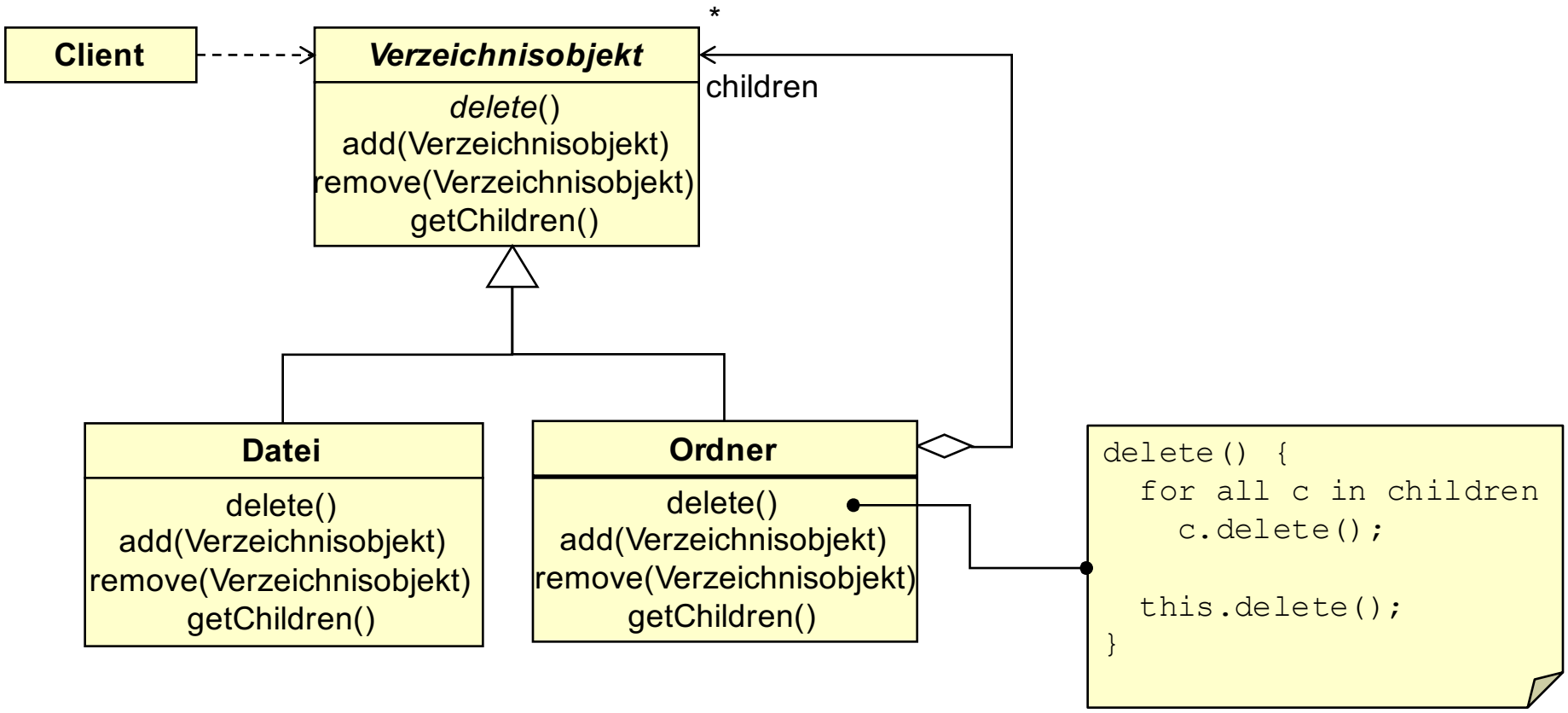
Composite ▸ Beispiel

- **Szenario:**
  - Stellen Sie sich die Verzeichnisstruktur ihres Betriebssystems vor: Vereinfacht besteht die Struktur aus Ordnern, die einzelne Dateien und wiederum Ordner enthalten können. Auf diese Weise entsteht ein hierarchischer Verzeichnisbaum.
  - Sie möchten nun eine Software schreiben, die die Verzeichnisstruktur eines Datenträgers in einer internen Struktur abbildet.

- **Lösung:**
  - Die Ordner werden als *Composites* realisiert
  - Die Dateien werden als *Leaves* realisiert.

- **Überlegungen:**
  - Angenommen, sie möchten auf allen Dateien der Struktur eine Operation aufrufen. Was müssen sie dafür tun?

Composite ▸ Beispiel

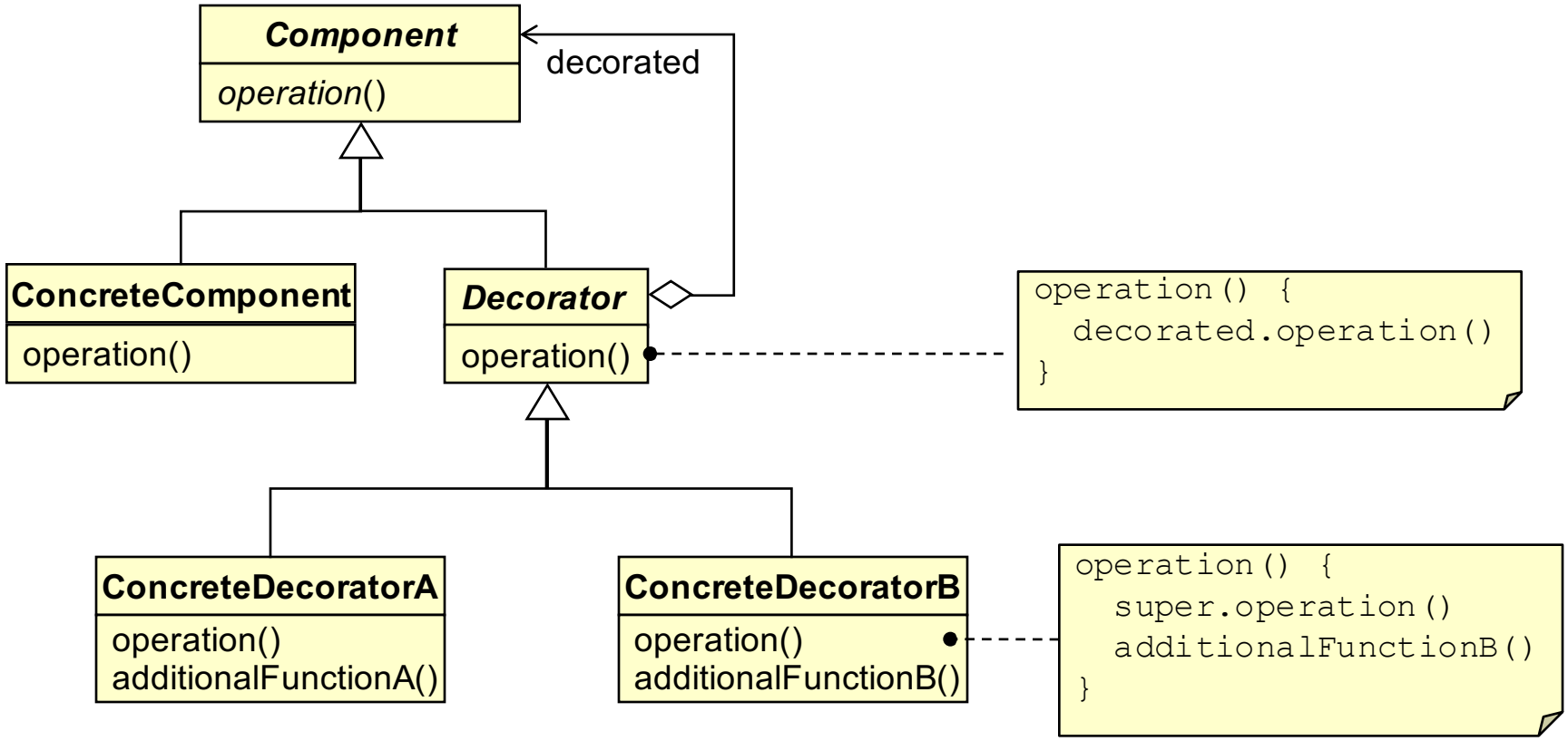


Decorator

Decorator ▸ Übersicht

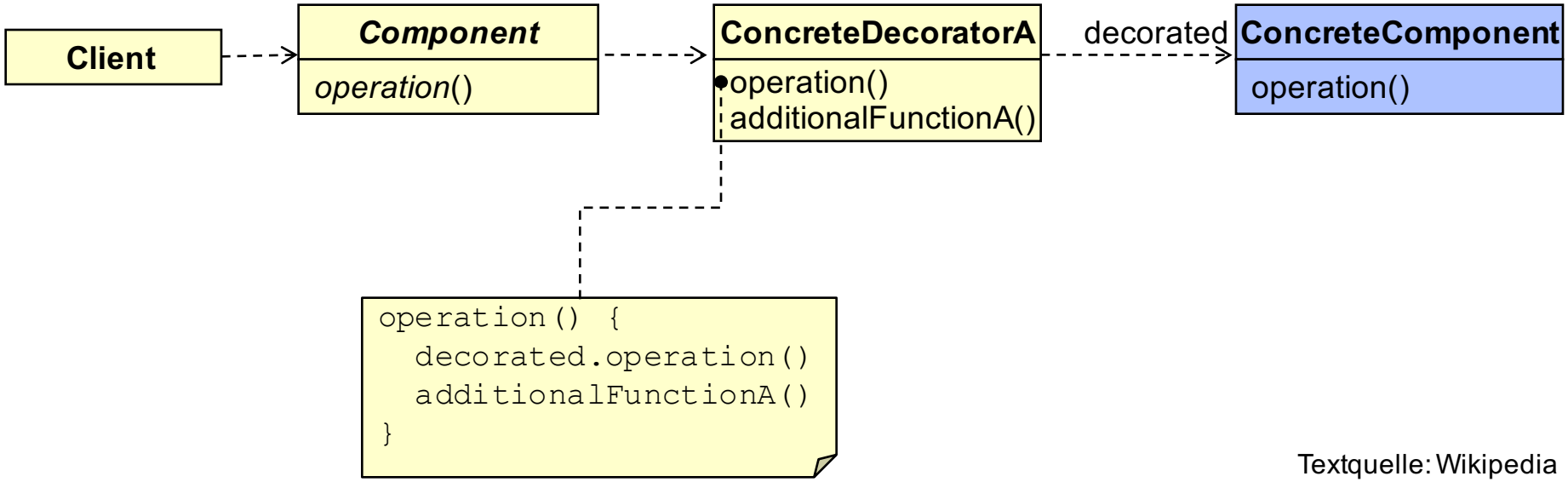
- **Absicht:**
  - Erweitert, verändert oder entfernt dynamisch Funktionalitäten eines vorhandenen Objekts, ohne das Objekt selbst zu ändern
- **Anwendbarkeit:**
  - Erweiterung mittels Unterklassenbildung nicht praktisch durchführbar
  - Dynamisch Funktionalität hinzufügen die auch entfernt werden kann
  - Dieses soll transparent geschehen (für Aufrufer nicht sichtbar)
- **Konsequenzen:**
  - Größere Flexibilität im Vergleich zu statischer Vererbung
  - Vermeidet es, weit oben in der Hierarchie stehende Klassen mit Funktionalität zu überfrachten
  - Führt meist zu vielen kleinen Objekten
  - Durch dynamische Funktionalitäten leicht zu konfigurieren / anzupassen

Decorator ▸ Schema



Decorator ▸ Funktionsweise

- Die Instanz eines Dekorierers wird vor die zu dekorierende Klasse (ConcreteComponent) geschaltet. Der Dekorierer hat die gleiche Schnittstelle wie die zu dekorierende Klasse.
- Aufrufe an den Dekorierer werden dann verändert oder unverändert weitergeleitet (Delegation), oder sie werden komplett in Eigenregie verarbeitet.
- Der Dekorierer ist dabei „unsichtbar“ (transparent), da der Aufrufende gar nicht mitbekommt, dass ein Dekorierer vorgeschaltet ist.
- Entsprechend ist es möglich mehrere Dekorierer „in Reihe“ vor die dekorierte Klasse zu schalten



Textquelle: Wikipedia

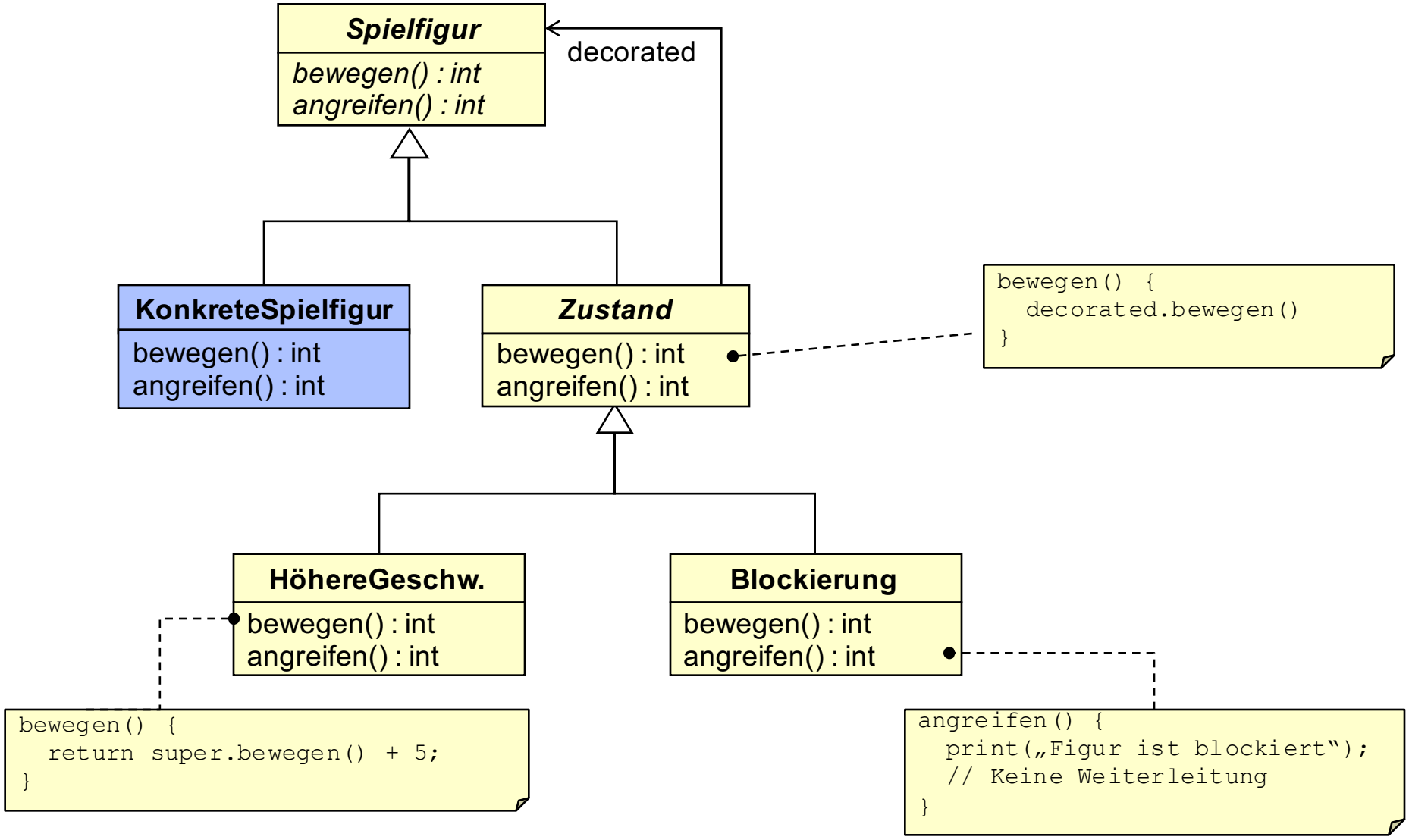
Decorator ▸ Beispiel

- **Szenario:**
  - Sie entwickeln ein neues PC-Rollenspiel. Die Spielfiguren werden als Objekte realisiert, in denen permanente Werte wie Bewegungsgeschwindigkeit, Angriffswert, etc… abgelegt werden.
  - Die Figuren sollen mit *Zuständen* belegt werden können, die es erlauben, bestimmte Eigenschaften für eine gewisse Zeit zu manipulieren (z.B. 10sec Bonus auf Angriffswert). Da viele solche Zustände gleichzeitig aktiv sein können sollen, wäre es schwierig jedesmal die Basiswerte zu verändern und wiederherzustellen.

- **Lösung:**
  - Die Figurobjekte mit den Basiswerten werden als *ConcreteComponent* realisiert
  - Die verschiedenen Zustände werden als *ConcreteDecorators* realisiert, die entsprechenden Funktionen des ConcreteComponent erweitern/überschreiben.

- **Überlegung:**
  - Wie aufwendig ist es neue Zustände hinzuzufügen?
  - Muss man die bestehenden Zustände für neue Spielfiguren anpassen?

Decorator ▸ Beispiel

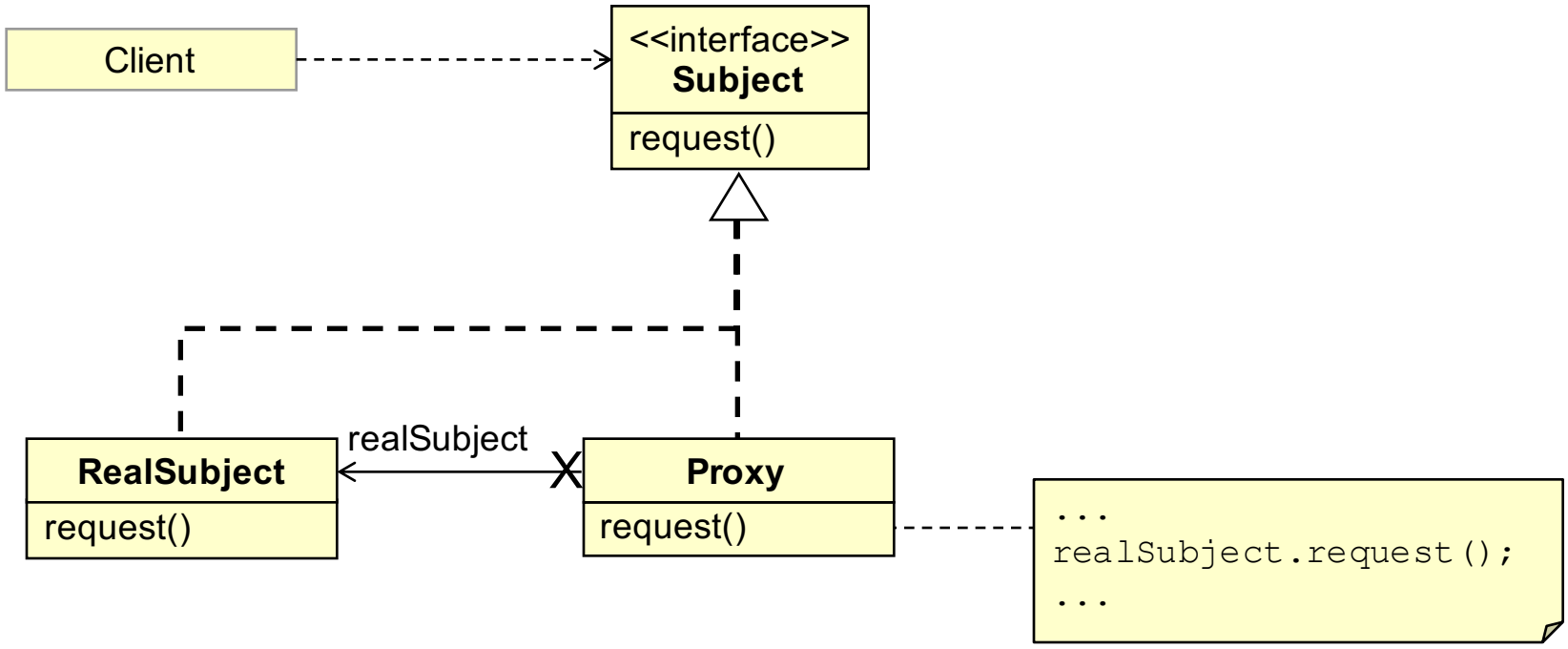


Proxy

Proxy ▸ Übersicht

- **Absicht**:
  - Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreters
- **Motivation**:
  - Verzögern der Erzeugungskosten bis zur Nutzung
    - ⇨ Platzhalter (Proxy) fungiert solange als ursprüngliches Objekt
  - Zugriff auf Objekt verwalten und/oder kontrollieren
- **Konsequenzen**:
  - Führt zusätzliche Indirektion ein
  - Je nach Proxy wird Indirektion anders verwendet

Proxy ▸ Schema



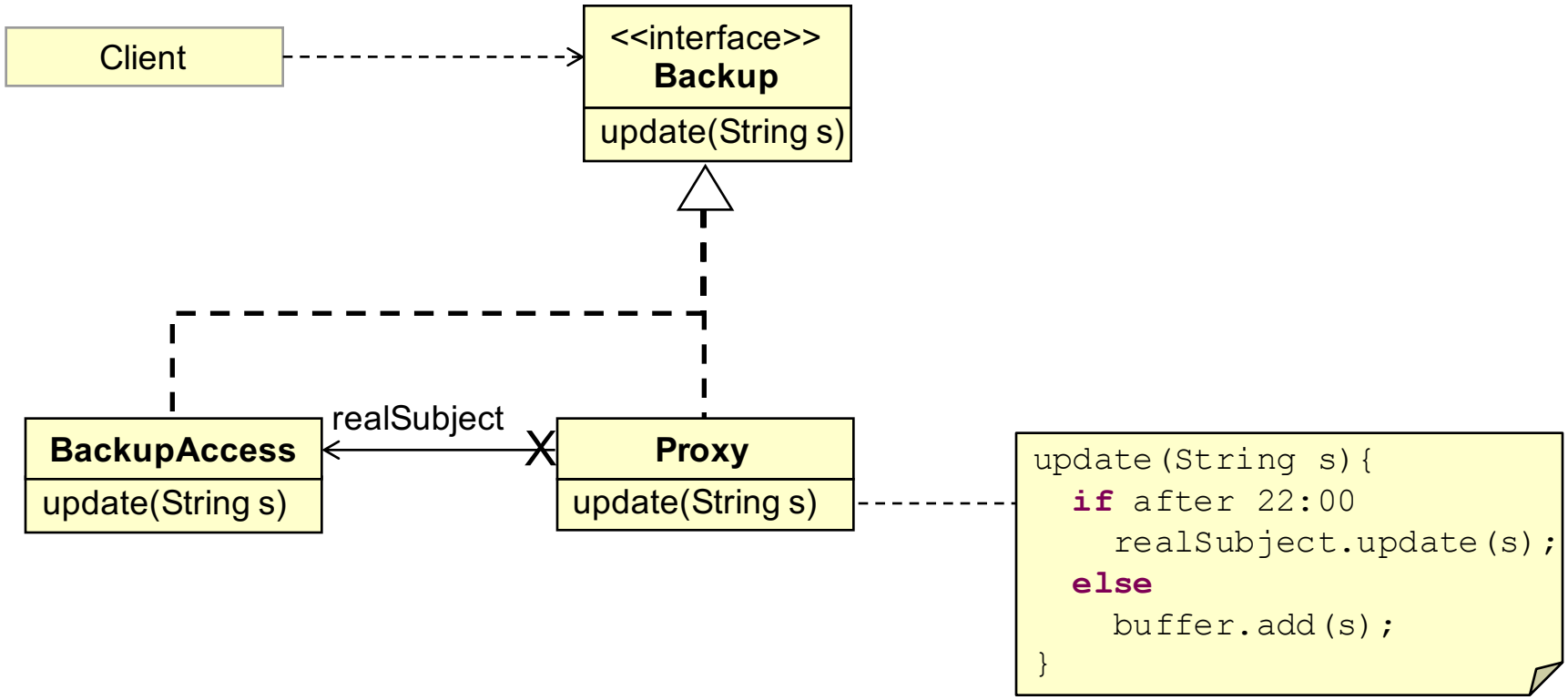
Proxy ▸ Beispiel

- **Szenario:**
  - Von einer Datenbank soll jede Nacht ein Backup auf einem entfernten System angelegt werden. Um nicht immer den kompletten Datenbestand zu übertragen, sollen jeweils nur die Änderungen auf das Backup übertragen werden.
  - Während des Tages sollen dafür alle Änderungen an die „richtige“ Datenbank in gleicher Weise auf das Backup angewendet werden. Die Änderungen sollen jedoch über den Tag gesammelt, und erst nachts an das Backup übergeben werden.

- **Lösung:**
  - Die Klasse, die die Kommunikation mit dem entfernten Backup realisiert wird als *RealSubject* realisiert.
  - Der Zugriff erfolgt über eine Proxy-Klasse, die alle Änderungen an das RealSubject annimmt, und bis zur Nacht zurückhält.

- **Überlegungen:**
  - Angenommen, die Übertragungsperioden sollen verändert werden. An wie vielen Stellen muss geändert werden?
  - Das Kommunikationsverfahren zum Backup muss geändert werden. An wie vielen Stellen muss geändert werden?

Proxy ▸ Beispiel



Adapter

Adapter ▸ Übersicht

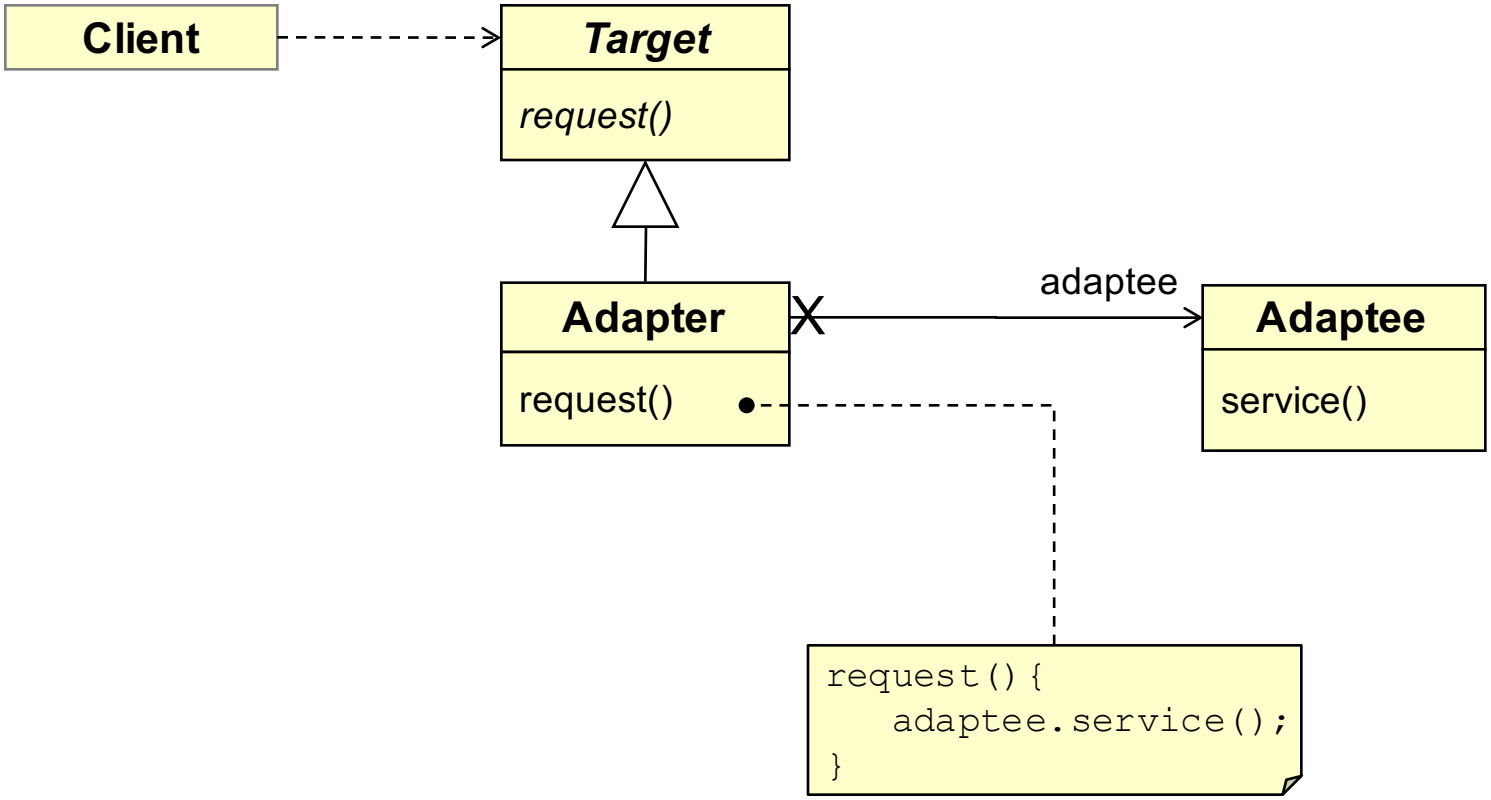
- **Absicht**:

  Schnittstelle existierender Klasse an Bedarf existierender Klienten anpassen

- **Anwendbarkeit:**
  - Eine existierende Klasse verwenden, deren Schnittstelle nicht der benötigten Schnittstelle entspricht.

- **Konsequenzen**:
  - Passt die zu adaptierende Klasse an genau eine konkrete Zielschnittstelle an
  - Funktioniert nicht für Unterklassen

Adapter ▸ Schema

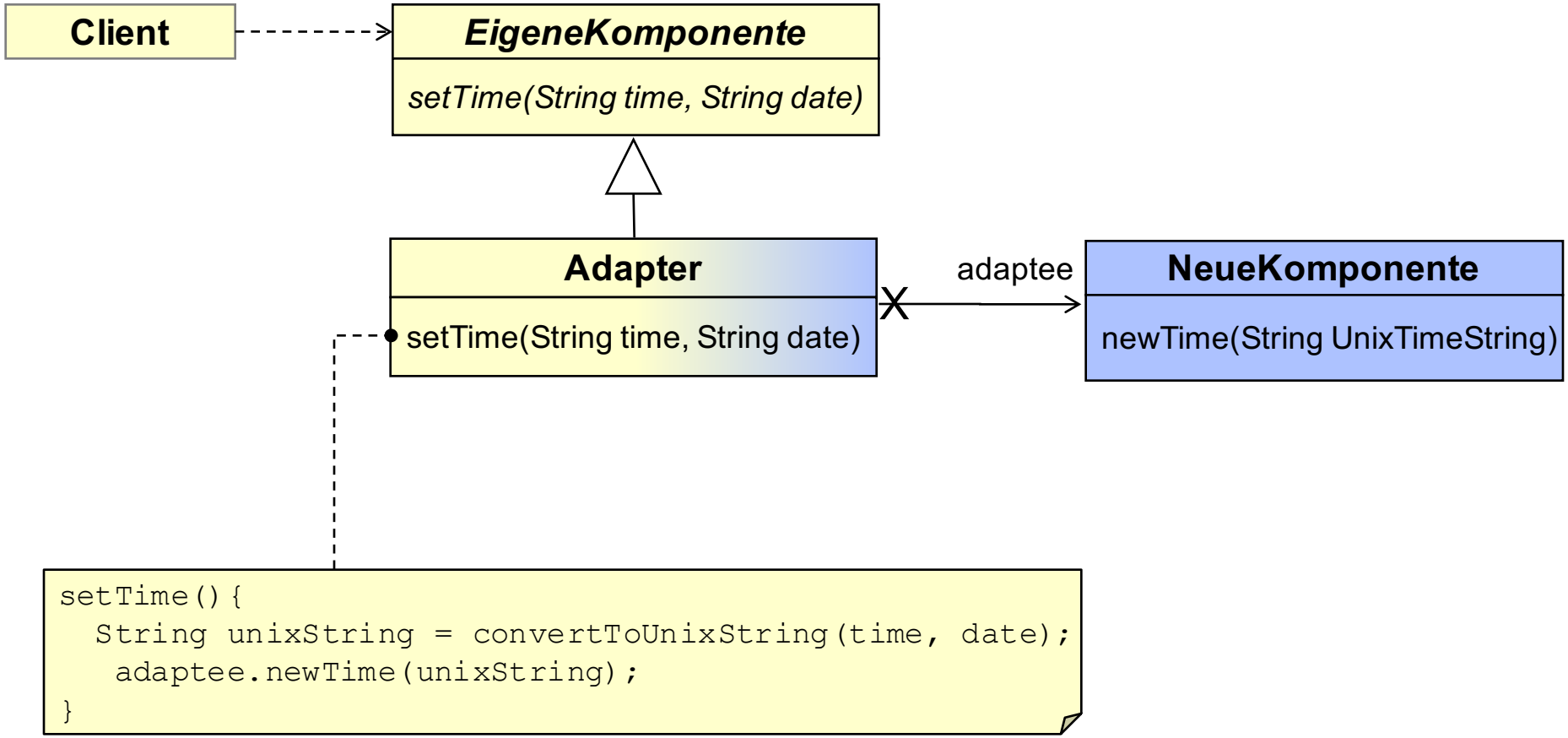


Adapter ▸ Beispiel

- **Szenario:**
  - Sie haben für Ihr aktuelles Projekt eine Software-Komponente zur Verarbeitung von Zeit- und Datumsoperationen gekauft. Eigentlich wollen sie die Komponente direkt integrieren, doch leider verwendet die gekaufte Komponente in ihren Interfaces ausschließlich Zeitformate, die zu den Interfaces ihrer Software inkompatibel sind.
  - Ihre Software verwendet das Weltzeit-Format mit einem zusätzlichem Datumsstring.
  - Die Komponente verwendet das Unix-Format, das sowohl Zeit als auch Datum direkt enthält.

- **Lösung:**
  - Zwischen Ihre eigenen Komponenten, und die neue Komponente wird ein Adapter geschaltet, der die Formate ineinander umrechnen kann.
  - Es sei hier vereinfacht ein Adapter nur für den Zugriff auf die neue Komponente benötigt.

Adapter ▸ Schema



Bridge

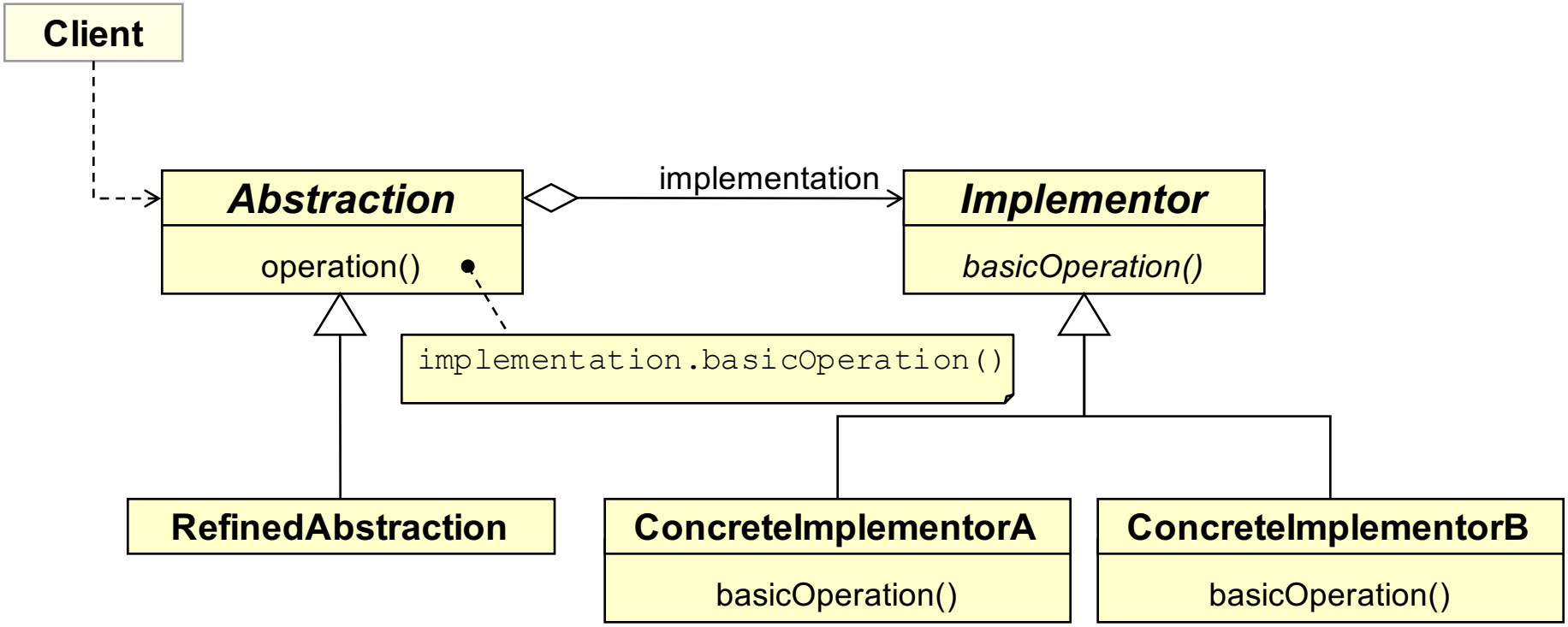
Bridge ▸ Übersicht

- **Absicht:**
  - Schnittstelle und Implementierung
    - ⇨ trennen
    - ⇨ unabhängig voneinander variieren

- **Anwendungskontext:**
  - Implementierung einer Schnittstelle soll dynamisch bei Laufzeit ausgewählt werden können
  - Schnittstelle und Implementierung sollen unabhängig voneinander variierbar sein

- **Konsequenzen:**
  - Klient wird von den abstrakten und konkreten Implementierungen abgeschirmt

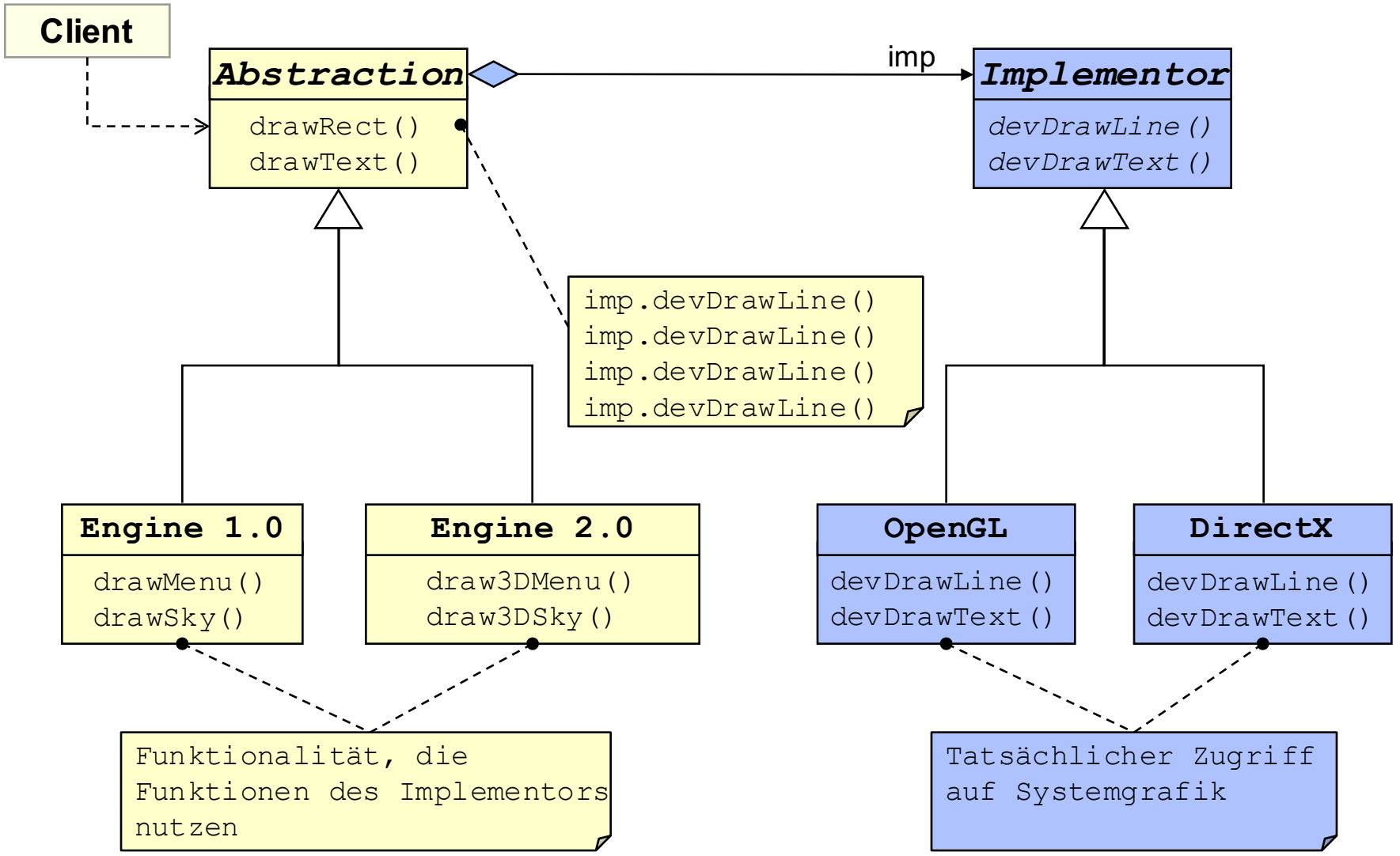
Bridge ▸ Schema



Bridge ▸ Beispiel

- **Szenario:**
  - Moderne Spiele werden heutzutage oft für mehrere Plattformen entwickelt (PC, Konsolen) um einen größeren Nutzerkreis anzusprechen. An die unterliegenden Grafik-Engines wird daher die Anforderung gestellt mit den unterschiedlichen Grafik-Schnittstellen der Systeme (DirectX, OpenGL, Konsolen) gleichermaßen umgehen zu können.
  - Der Zugriff auf die Funktionen der Engine soll unabhängig von der gerade verwendeten Plattform sein. Die Weiterentwicklung der Funktionalität soll unabhängig von der Entwicklung der Implementierung für die einzelnen Plattformen möglich sein.
- **Lösung:**
  - Die Schnittstelle zur Funktionalität wird in der *Abstraction* realisiert
  - Die Implementierung der Funktionalität als Unterklassen des *Implementors*
- **Überlegung:**
  - Was muss getan werden um eine neue Plattform zu unterstützen?
  - Was muss getan werden um neue Funktionalität hinzuzufügen?

Bridge ▸ Beispiel



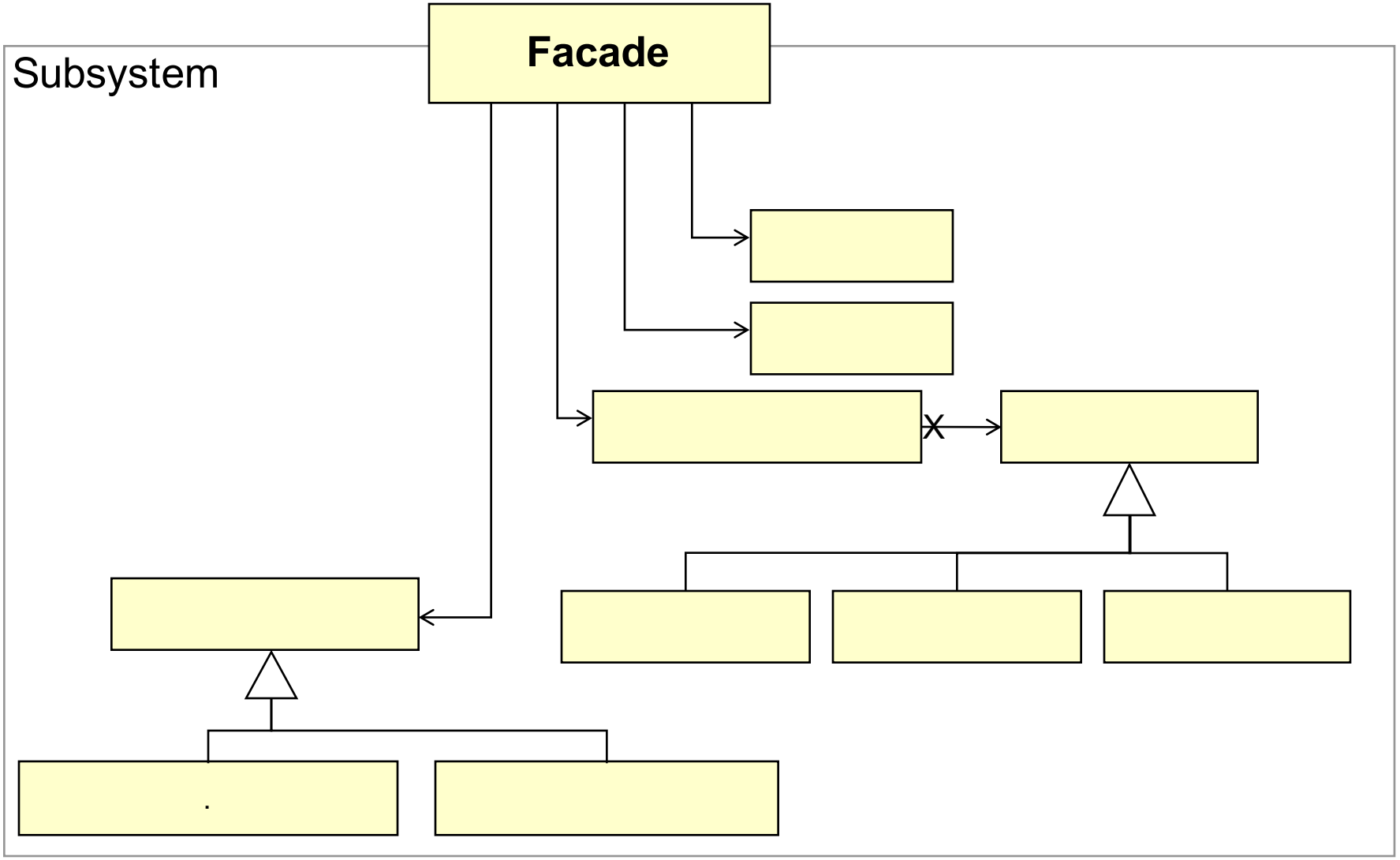
Facade

Facade ▸ Schema

- **Absicht:**
  - Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems
    - ⇨ Ermögliche einfachen Zugriff auf ein komplexes Subsystem über eine zentrale Schnittstelle
    - ⇨ Reduzierung der Abhängigkeiten der Klienten von der Struktur eines Subsystems

- **Konsequenzen:**
  - Reduziert die Anzahl von Objekten die von Klienten gehandhabt werden müssen
  - Fördert lose Kopplung zwischen dem System und seinen Klienten
  - Verhindert nicht, dass Anwendungen direkt Subsystemklassen verwenden können

Facade ▸ Schema



Facade ▸ Beispiel

- **Szenario:**
  - Sie haben eine große Komponente entwickelt, die aus hunderten Klassen und Interfaces besteht. Der überwiegende Teil ist für einen konkreten Benutzer jedoch völlig uninteressant. Der Benutzer möchte (und soll) die Komponente einfach nur nutzen, und sich nicht mit den vielen Hilfsklassen und internen Datenstrukturen befassen.

- **Lösung:**
  - Definieren Sie eine zentrale Schnittstelle, die *Facade*, die dem Benutzer erlaubt alle Funktionen ihrer Komponente zu nutzen. Im Hintergrund delegiert die Facade die Anfragen an die richtigen Klassen weiter – davon braucht der Nutzer jedoch nichts zu wissen.
  - Der Benutzer muss nur die *Facade* kennen, um die Komponente nutzen zu können. Nach diesem Konzept funktionieren viele typische Software-Bibliotheken.

Beispiele zum Üben

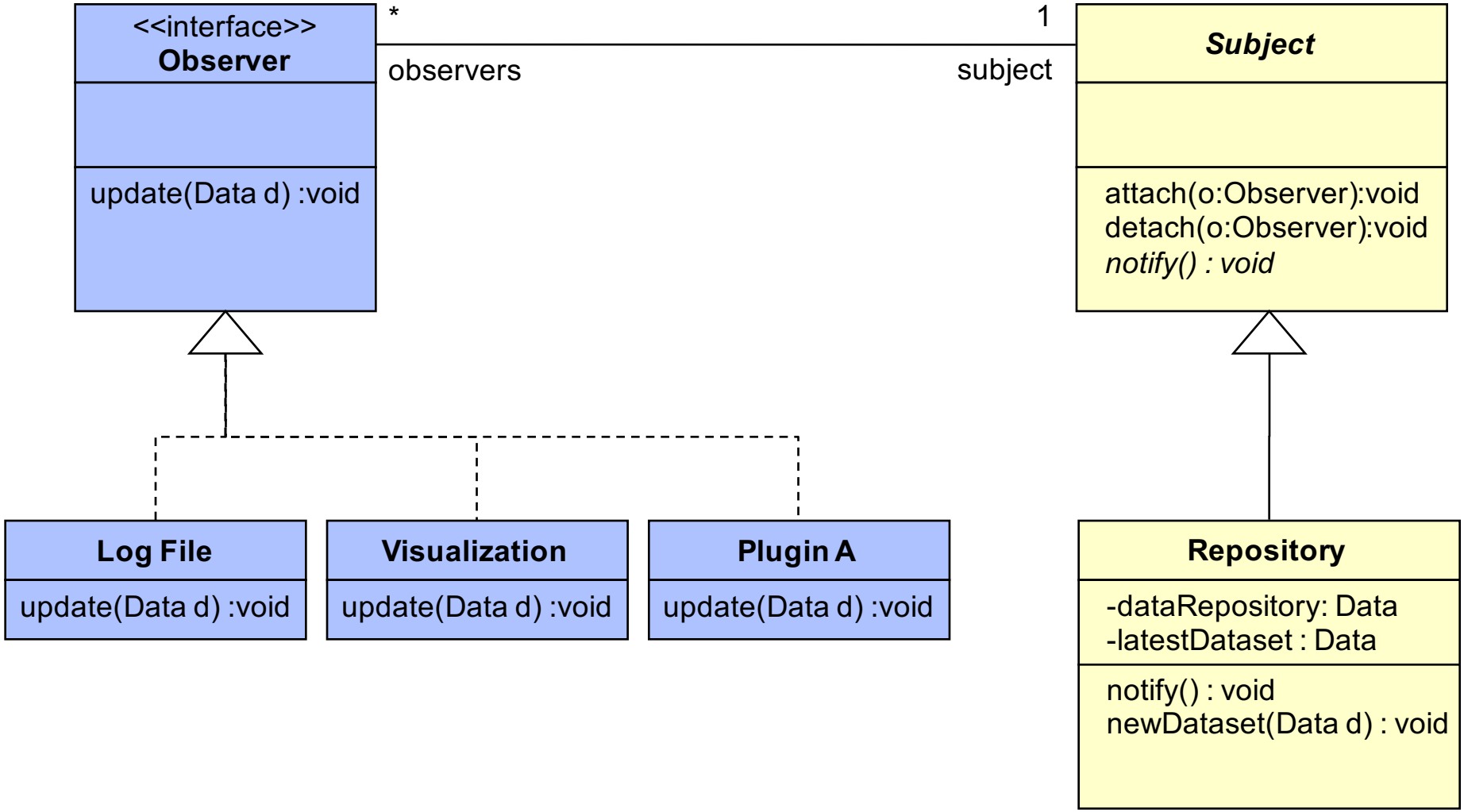
Welches Pattern würden Sie verwenden?

Beispiel 1 ▸ Szenario

- **Szenario:**
  - Eine Analyse-Software empfängt in unregelmäßigen Abständen Daten, die je nach Konfiguration unterschiedlich verarbeitet werden sollen.
  - So soll es für den Endkunden insbesondere möglich sein, eigene Plugins zu entwickeln, die bei allen neu eintreffenden Daten aufgerufen werden. Standardmäßig übergibt die Software neue Daten an:
    - ⇨ Eine Protokolldatei
    - ⇨ Eine Visualisierungsklasse
    - ⇨ Beliebig viele unbekannte Plugins

- **Aufgaben:**
  - Welches Pattern bietet sich an?
  - Was muss getan werden um neue Plugins hinzuzufügen?

Beispiel 1 ▸ Observer (Push)

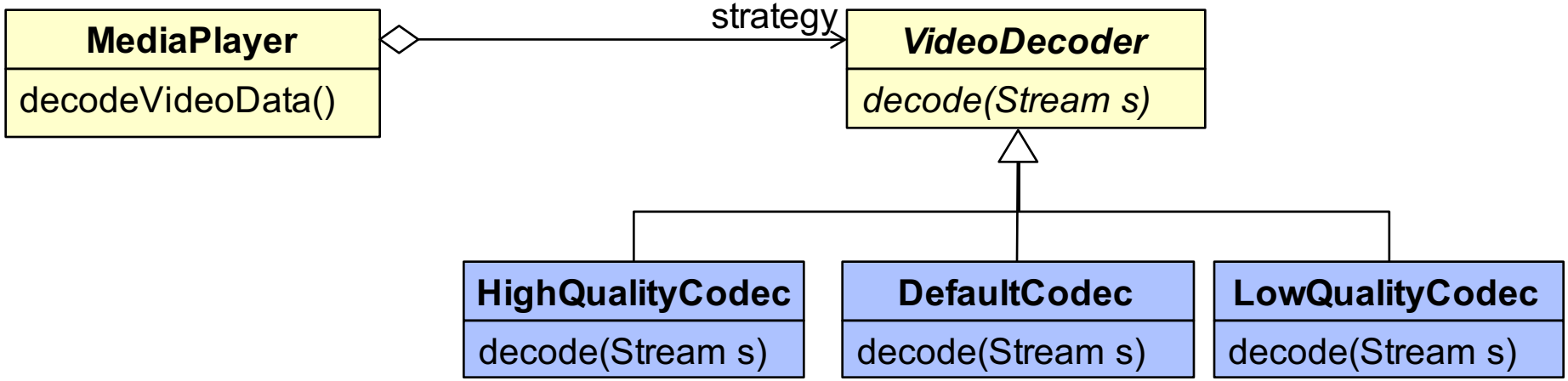


Beispiel 2 ▸ Szenario

- **Szenario:**
  - Viele MediaPlayer, die Videostreams aus dem Internet abspielen, sind in der Lage den Codec, der für die Komprimierung der Videodaten zuständig ist, dynamisch nach der verfügbaren Bandbreite anzupassen.
  - So wird etwa eine höhere Kompression gewählt, wenn nur wenig Bandbreite zur Verfügung steht. Die Qualität wird dadurch sinken, doch die Menge zu übertragender Daten nimmt ebenfalls ab.
  - Steht ausreichend Bandbreite zur Verfügung, kommt der Benutzer hingegen in den Genuss von HD-Bildern. In beiden Fällen wird so versucht, eine ruckelfreie Videowiedergabe zu erzielen.

- **Aufgaben:**
  - Welches Pattern bietet sich an dynamisch Codecs / Algorithmen auszutauschen?

Beispiel 2 ▸ Strategy

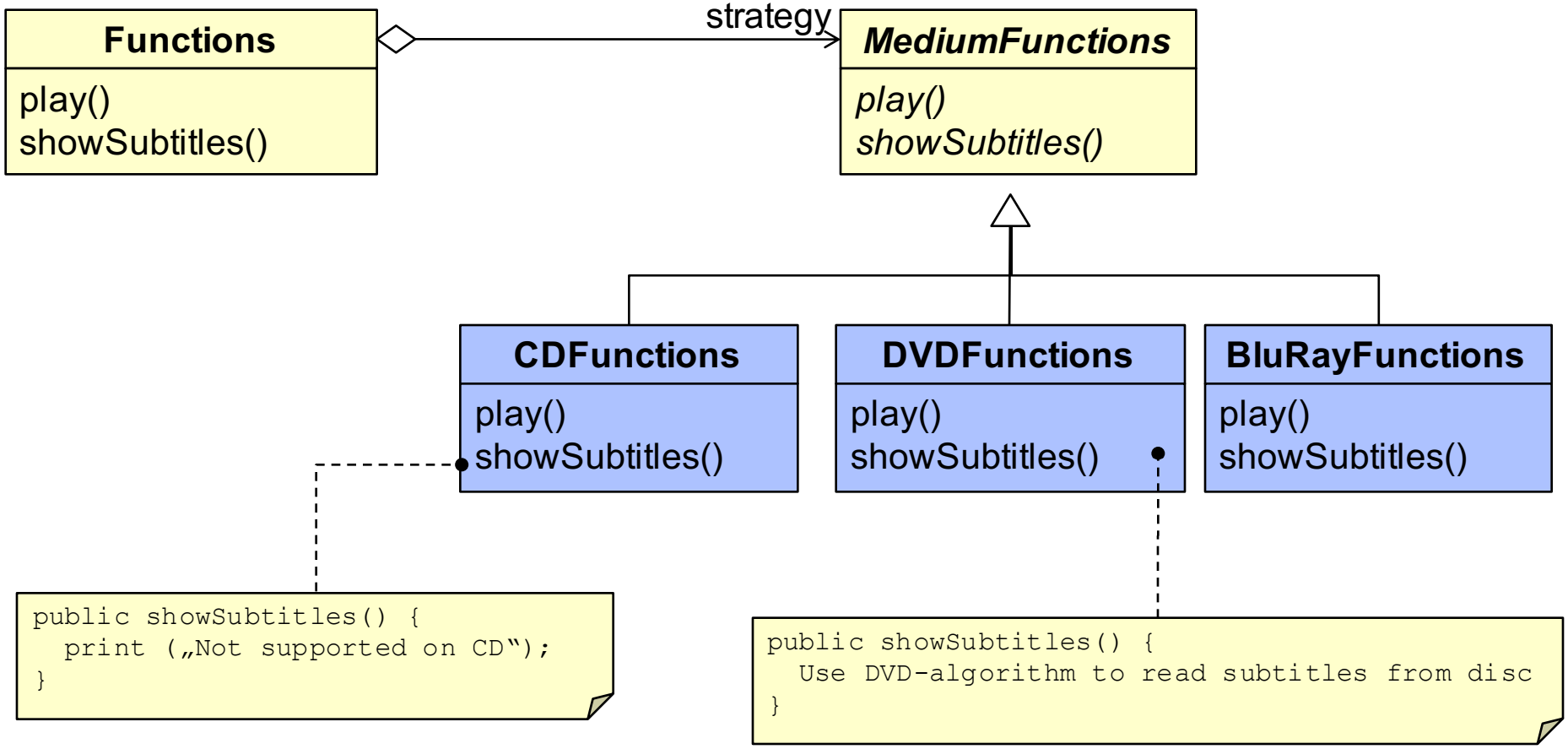


Beispiel 3 ▸ Szenario

- **Szenario:**
  - Ein MediaPlayer beherrscht das Abspielen von CDs, DVDs und Blu Ray Discs. Wird ein solcher Medientyp eingelegt, wechselt der Player in den entsprechenden Wiedergabemodus. Der Player bietet die Operationen *play()* und *showSubtitles()* an.
  - Jedes dieser Medien benutzt andere Kodierungstechniken, daher muss immer dynamisch die richtige Implementierung ausgewählt werden.
  - Es ist außerdem zu beachten, dass nicht alle Medien alle Operationen unterstützen (CDs enthalten keine Untertitel).

- **Aufgaben:**
  - Welches Pattern bietet sich an dynamisch das Verhalten bei bestimmten Zuständen (Medium = CD / DVD / Blu Ray) zu variieren?

Beispiel 3 ▸ State

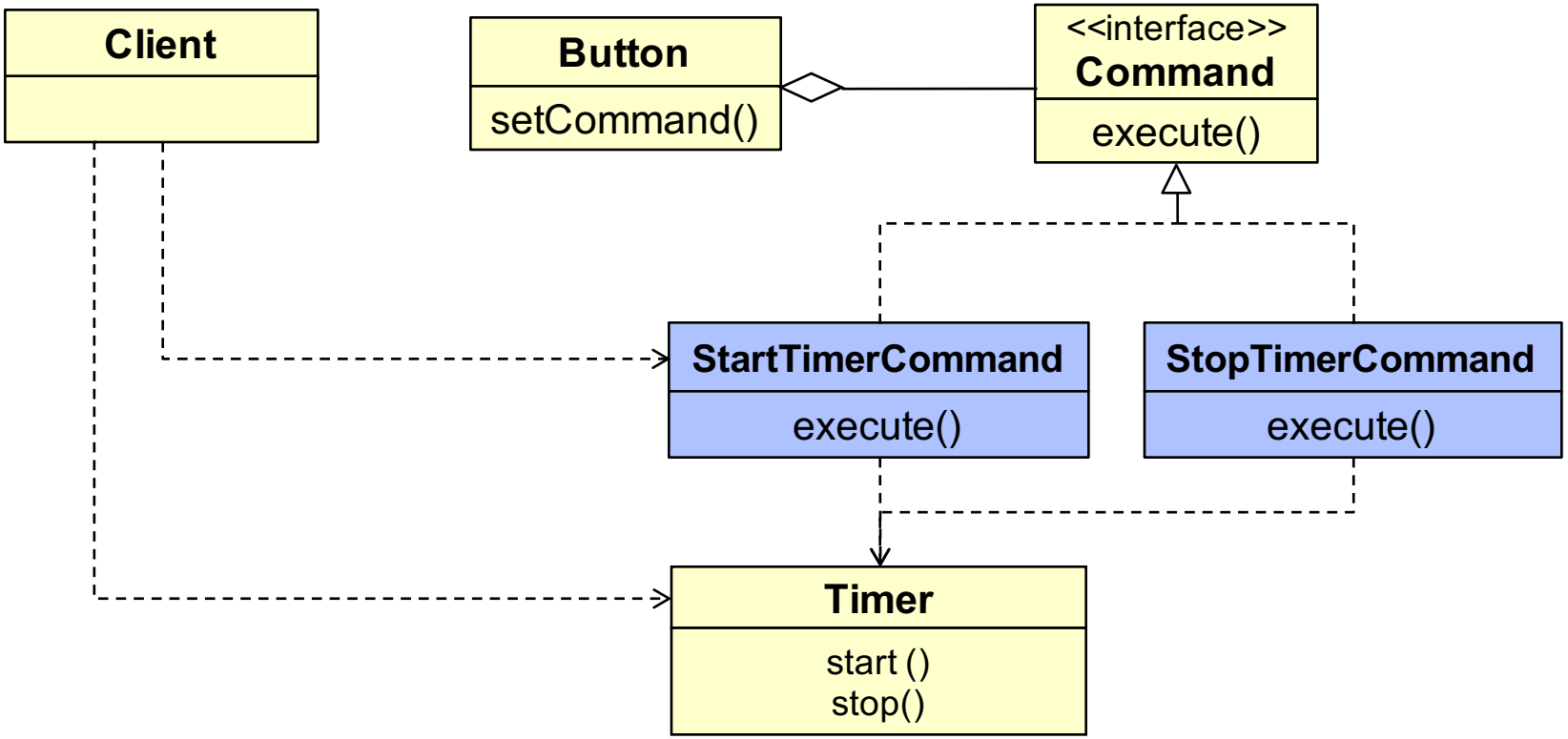


Beispiel 4 ▸ Szenario

- **Szenario:**
  - Stellen Sie sich eine typische Stoppuhr vor. In der Regel haben sie dort einen Start/Stopp-Button, mit folgender Funktionalität:
    - ⇨ Läuft die Stoppuhr, dann hält die Betätigung des Buttons die Uhr an.
    - ⇨ Steht die Stoppuhr, dann startet die Betätigung des Buttons die Uhr.
  - Die genaue Funktionalität des Buttons kann also dynamisch ausgetauscht werden.

- **Aufgaben:**
  - Welches Pattern bietet sich für dieses Problem an?
  - Wie könnte man eine dritte Funktionalität hinzufügen?

Beispiel 4 ▸ Command

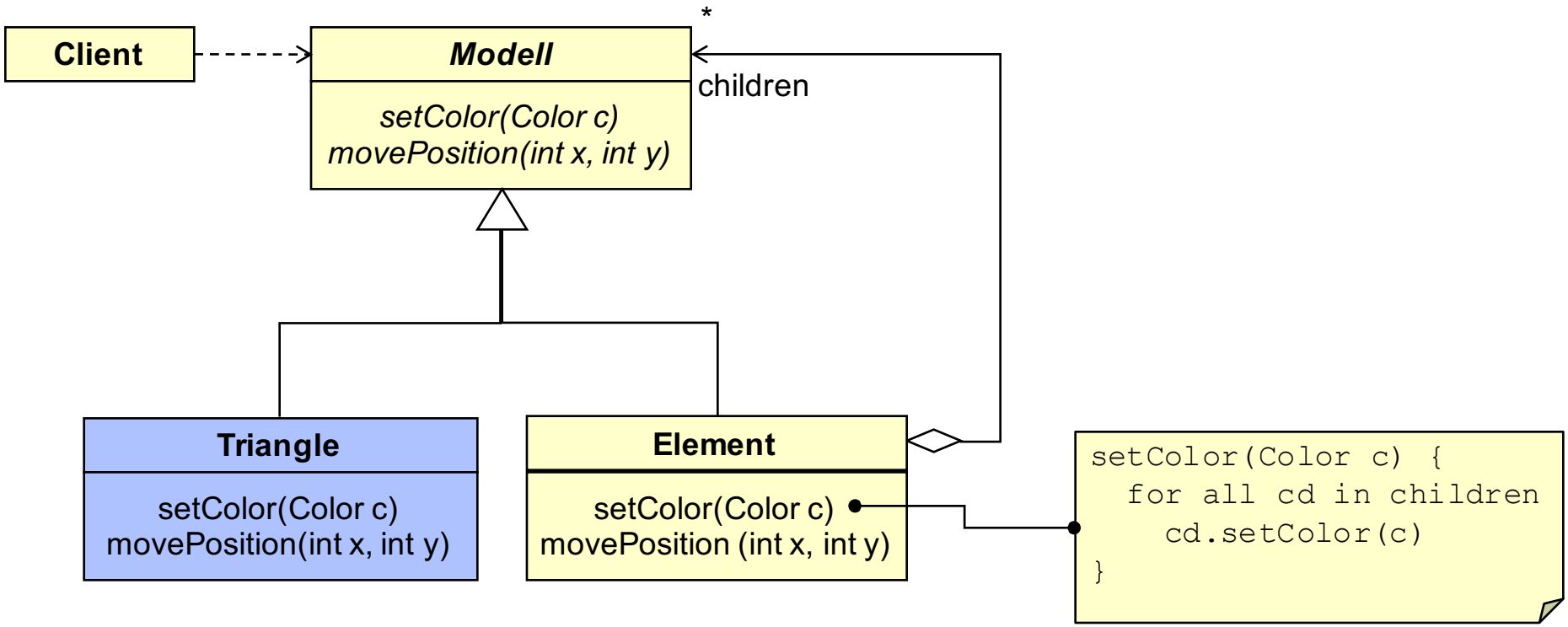


Beispiel 5 ▸ Szenario

- **Szenario:**
  - 3D-Objekte in der Computergrafik sind in der Regel aus vielen kleinen Dreiecken (Triangles) zusammengesetzt.
  - Aus diesen einfachen Elementen lassen sich komplexere Objekte wie Würfel oder Kugeln zusammensetzen.
  - Sehr komplexe Modelle, wie z.B. ganze Figuren, Gebäude, Raumschiffe, etc… sind oft aus vielen kleinen, einfacheren Elementen zusammengesetzt, die wiederum aus einfacheren Elementen bestehen, ….

- **Aufgaben:**
  - Welches Pattern bietet sich für die Darstellung eines komplexen Modells an?
  - Die Dreiecke haben eine Methode *setColor()*, mit der sie die Farbe setzen können. Wie können sie ein ganzes Modell einfärben?

Beispiel 5 ▸ Composite

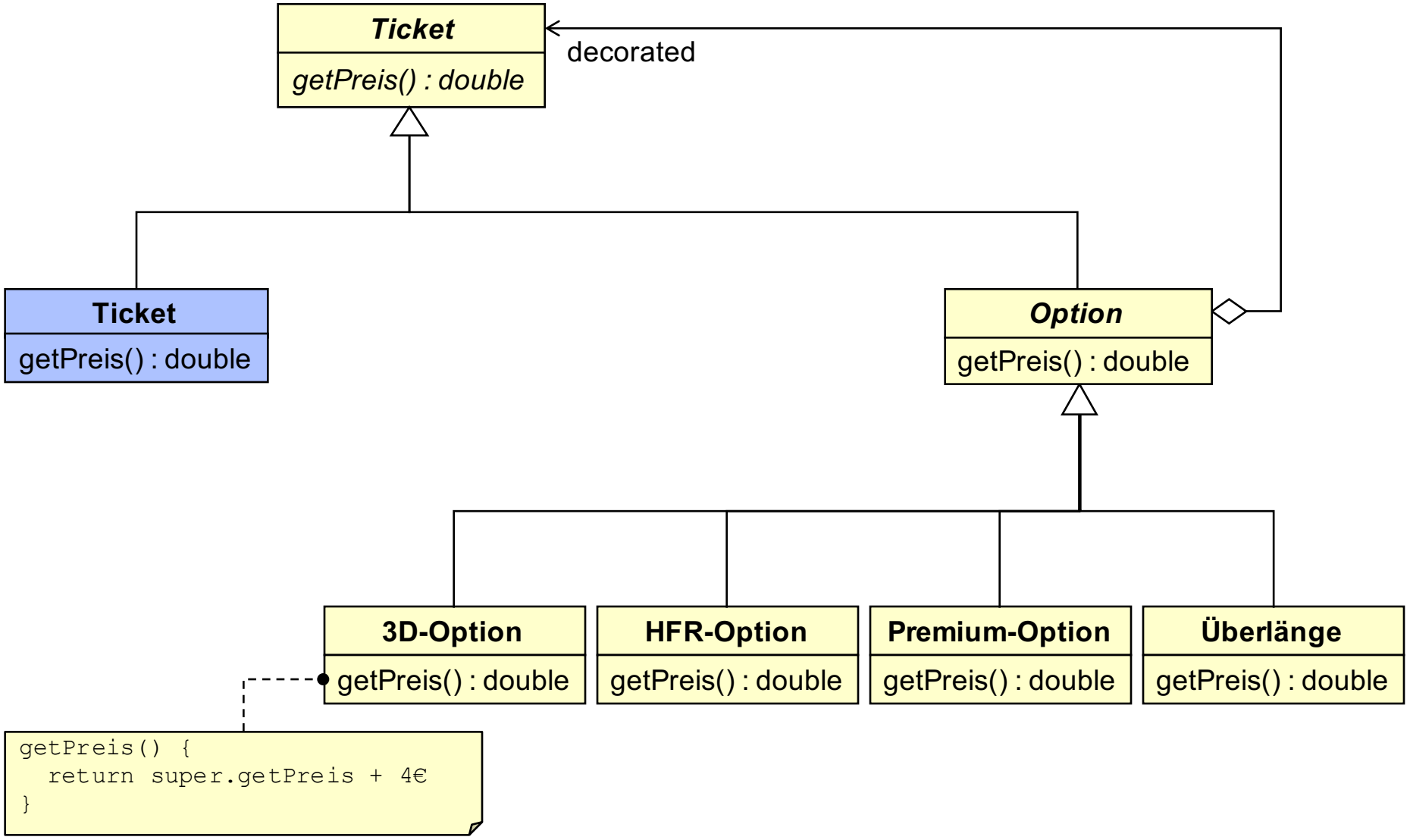


Beispiel 6 ▸ Szenario

- **Szenario:**
  - Ein Kino bietet neben dem einfachen Ticket zusätzliche Optionen an, die die Kunden je nach Wunsch individuell dazubuchen können. Jede Option erhöht entsprechend den Preis des Tickets.
  - Angeboten werden eine 3D-Option, HFR-Option, eine Premium-Option und der typische Wochenendzuschlag. Damit das System zukunftssicher bleibt, muss es möglich sein nachträglich weitere Optionen hinzuzufügen, die mit allen bisherigen kompatibel sind.

- **Aufgaben:**
  - Welches Pattern bietet sich für die Darstellung an?
  - Wie groß ist der Aufwand neue Optionen hinzuzufügen? Wie können Zuschläge für z.B. Wochenende, oder Überlänge mit einbezogen werden?

Beispiel 6 ▸ Decorator

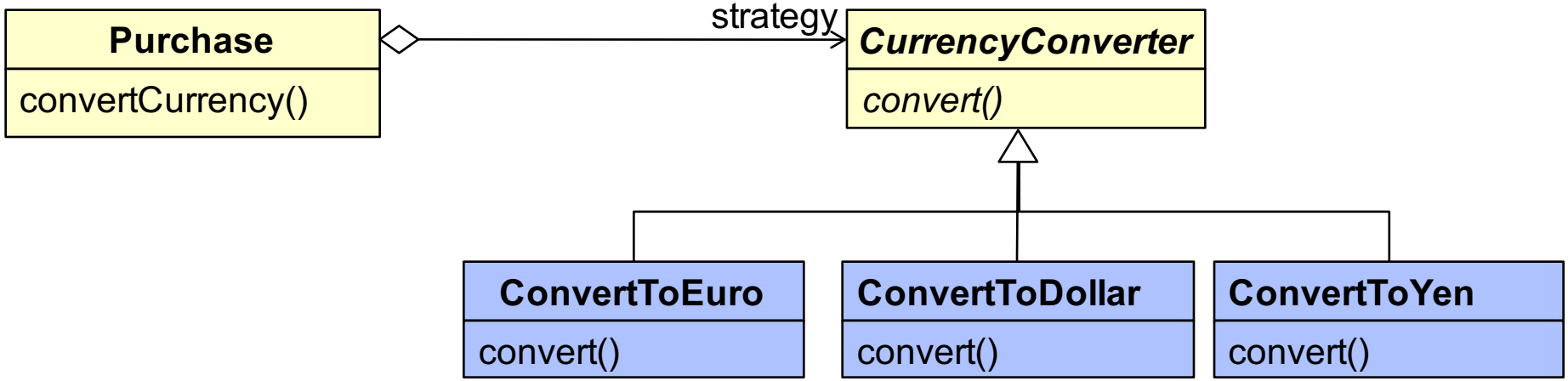


Beispiel 7 ▸ Szenario

- **Szenario:**
  - Ein internationaler Internet-Versandhandel versendet seine Produkte in nahezu alle Länder der Erde. Bei der Durchführung der Bestellung soll dem Kunden die Rechnung in der Währung seines Herkunftslandes ausgestellt werden.
  - Dazu verwaltet das Unternehmen eine Liste aller Preise. Die Wechselkurse werden stets aktuell von diversen Börsenplätzen rund um die Welt abgefragt. Die Herkunft des Kunden wird erst beim Abschluss der Bestellung bekannt. Der Berechnungsalgorithmus der Währung wird also erst bei Laufzeit ausgewählt.

- **Aufgaben:**
  - Welches Pattern bietet sich zur Berechnung des Endpreises an?
  - Was muss getan werden um eine neue Währung hinzuzufügen?

Beispiel 7 ▸ Strategy

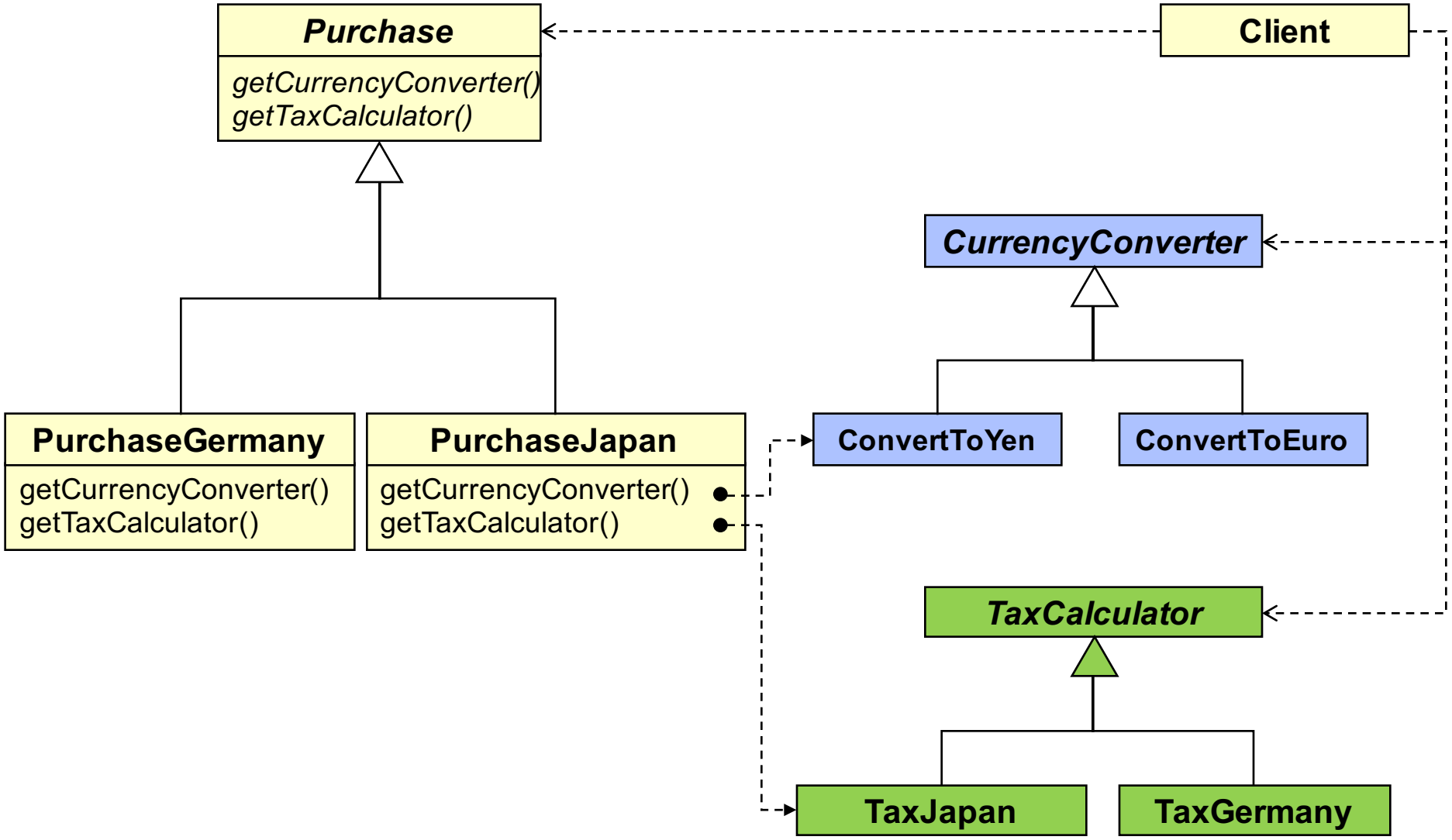


Beispiel 8 ▸ Szenario

- **Szenario:**
  - Den Entwicklern des Versandhandels ist eingefallen, dass bei der Berechnung des Endpreises natürlich auch noch die lokale Mehrwertsteuer berechnet werden muss.
  - Bei der Darstellung des Endpreises muss also sowohl die Währung, als auch die Mehrwertsteuer dynamisch nach dem Herkunftsland des Kunden bestimmt werden. Da die Steuersätze unglücklicherweise nicht an die Währung gebunden sind, müssen zwei getrennte Berechnungsalgorithmen abhängig vom Herkunftsland ausgewählt werden.

- **Aufgaben:**
  - Welches Pattern bietet sich zur Berechnung des Endpreises an?
  - Was muss getan werden um ein neues Land hinzuzufügen? Welche Klassen müssten z.B. für Frankreich eingefügt werden? Muss an der allgemeinen Struktur geändert werden?

Beispiel 8 ▸ Abstract Factory

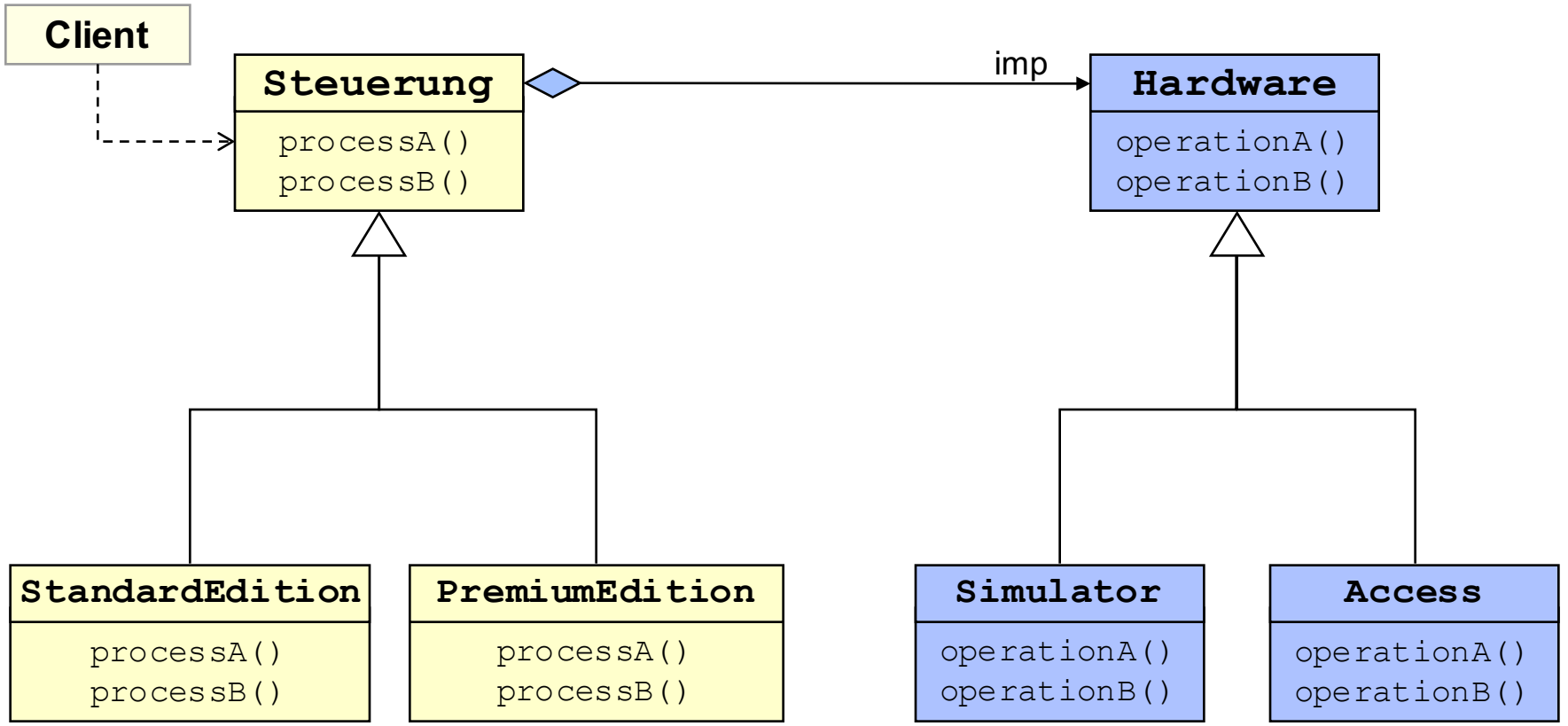


Beispiel 9 ▸ Szenario

- **Szenario:**
  - Sie sollen eine Steuerungssoftware für eine neue Maschine entwickeln. Der Zugriff auf die eigentliche Hardware erfolgt über eine Klasse *Access*. Um diese nutzen zu können, müssen sie natürlich die entsprechende Maschine vor Ort haben.
  - Dummerweise ist diese noch gar nicht komplett fertiggestellt (oder so sperrig, dass sie nicht in Ihr Büro passt), sodass sie eine Klasse *Simulator* verwenden müssen, die das Verhalten der Maschine simuliert.
  - Sie möchten nun die Steuerung fertig stellen, jedoch jederzeit problemlos zwischen der Simulation und der richtigen Implementierung wechseln können.

- **Aufgaben:**
  - Welches Pattern bietet sich für diese Problemstellung an?

Beispiel 9 ▸ Bridge

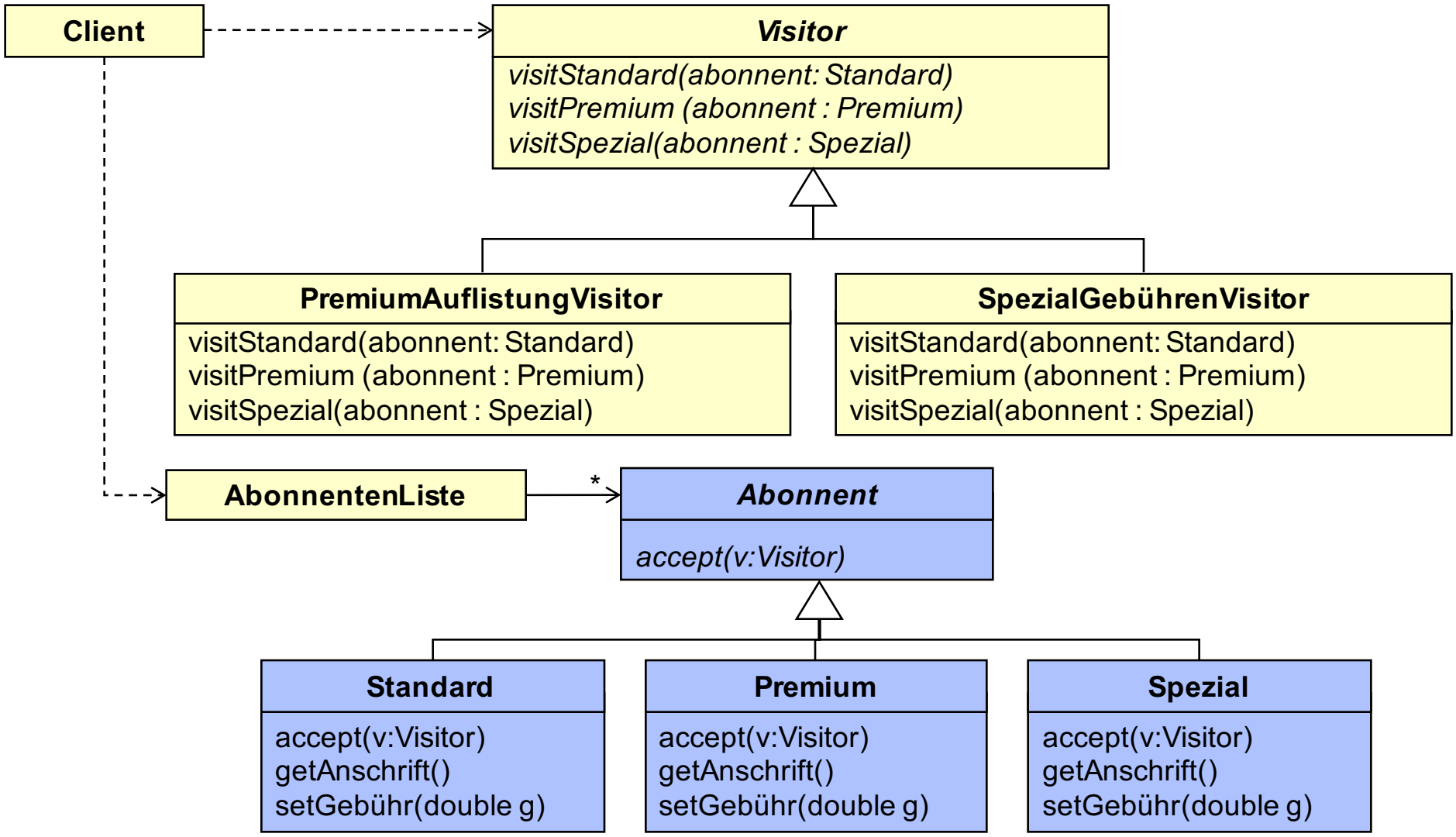


Beispiel 10 ▸ Szenario

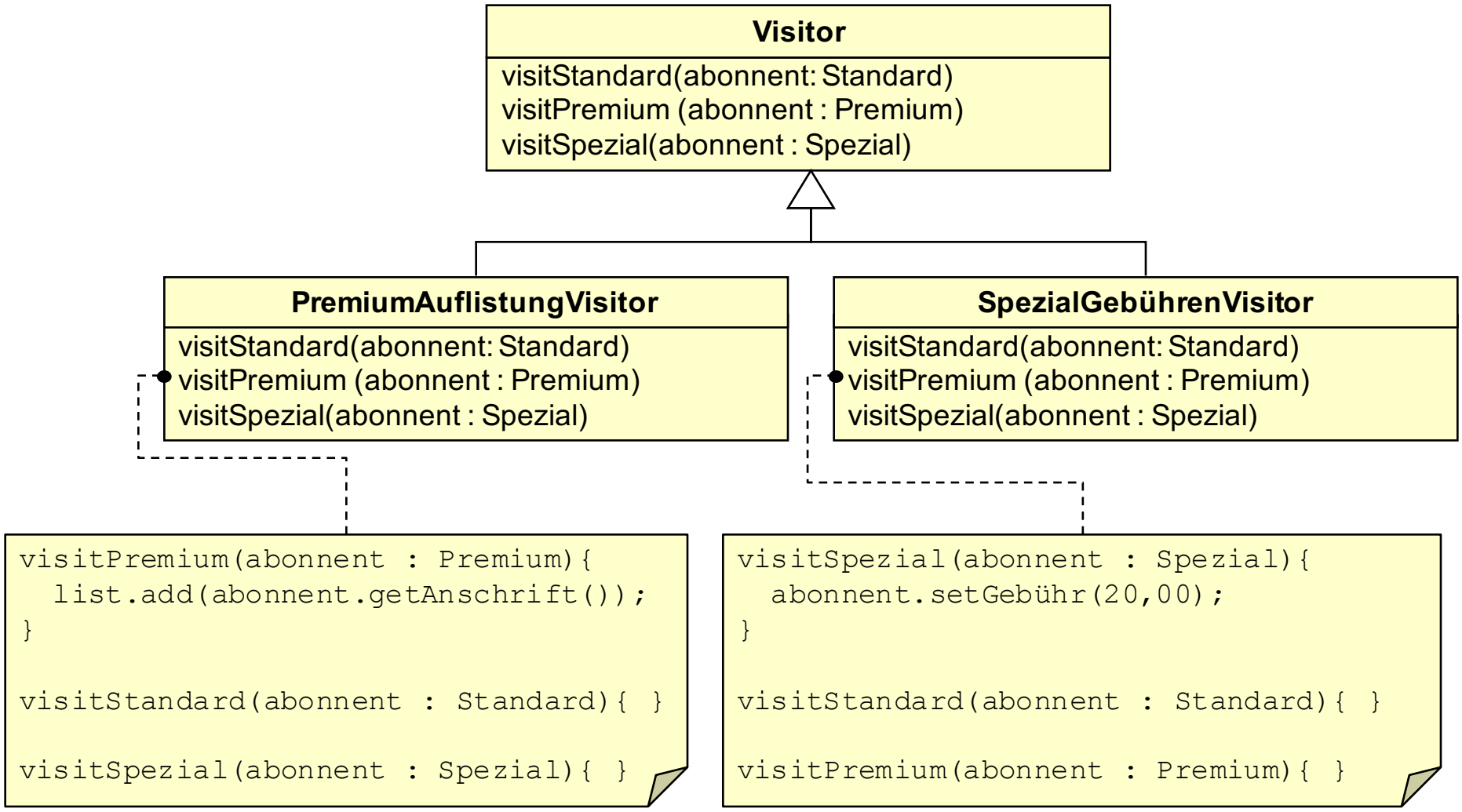
- **Szenario:**
  - Ein Zeitungsverlag verwaltet eine Datenstruktur mit allen ihren *Abonnenten*. Je nach Art des Abonnements gibt es *Standard, Premium* und *Spezial* Abonnenten.
  - Der Verlag möchte nun zu Weihnachten allen Premium Abonnenten eine Grußkarte zukommen lassen. Dafür benötigen sie die Anschriften aller Premium Kunden.
  - Unabhängig davon soll die Jahresgebühr der Spezialabonnenten auf 20€ neu gesetzt werden.

- **Aufgaben**:
  - Welches Pattern bietet sich für diese Problemstellung an?

Beispiel 10 ▸ Visitor



Beispiel 10 ▸ Visitor

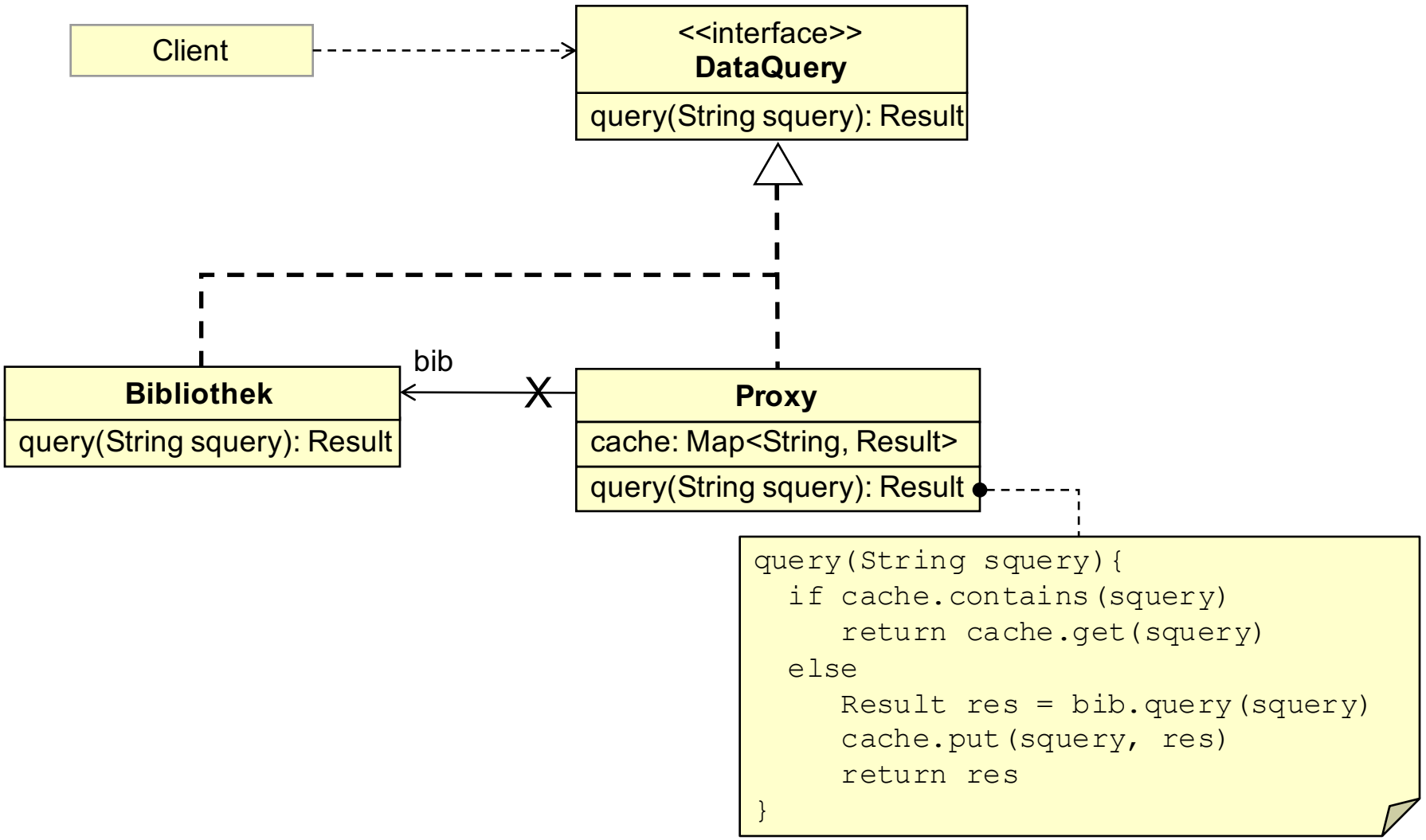


Beispiel 11 ▸ Szenario

- **Szenario:**
  - Sie verfügen über eine Bibliothek, die anhand von Parametern Anfragen an Webservices und Datenbanken zusammenstellt und die Informationen von diesen Diensten einholt.
  - Da diese Anfragen recht zeitaufwendig sind, wollen Sie die Ergebnisse zu einmal gestellten Anfragen in einem Cache speichern, damit wiederkehrende Anfragen direkt aus dem Cache beantwortet werden können. Nur neue Anfragen sollen an die Bibliothek zur Bearbeitung weitergereicht werden.

- **Aufgaben**:
  - Welches Pattern bietet sich für diese Problemstellung an?

Beispiel 11 ▸ Proxy

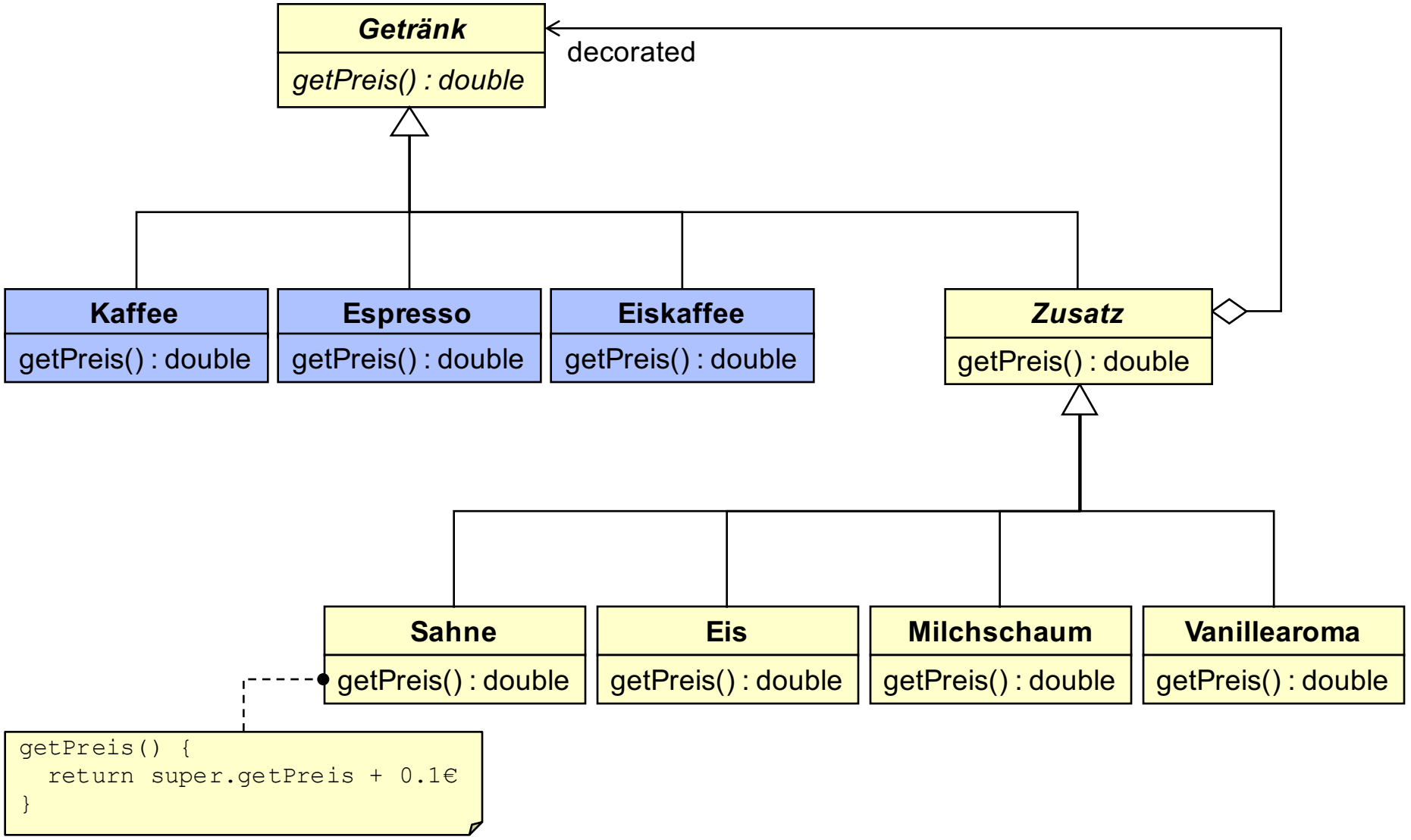


Beispiel 12 ▸ Szenario

- **Szenario:**
  - DAS Beispiel, um das man bei Design Patterns einfach nicht herumkommt:
  - Sie betreiben eine Espressobar und bieten eine Reihe von Kaffeegetränken an(Kaffee, Espresso, Eiskaffee). Zudem bieten sie diverse Zusätze an: Sahne, Eis, Milchschaum, und Vanillearoma.
  - Zu jedem Getränk kann eine beliebige Anzahl Zusätze geordert werden, wobei sich entsprechend der Preis erhöht.

- **Aufgaben:**
  - Welches Pattern bietet sich für die Darstellung aufwendiger Getränke an?
  - Wie groß ist der Aufwand neue Getränke oder Zusätze mit auf die Karte zu nehmen?

Beispiel 12 ▸ Decorator



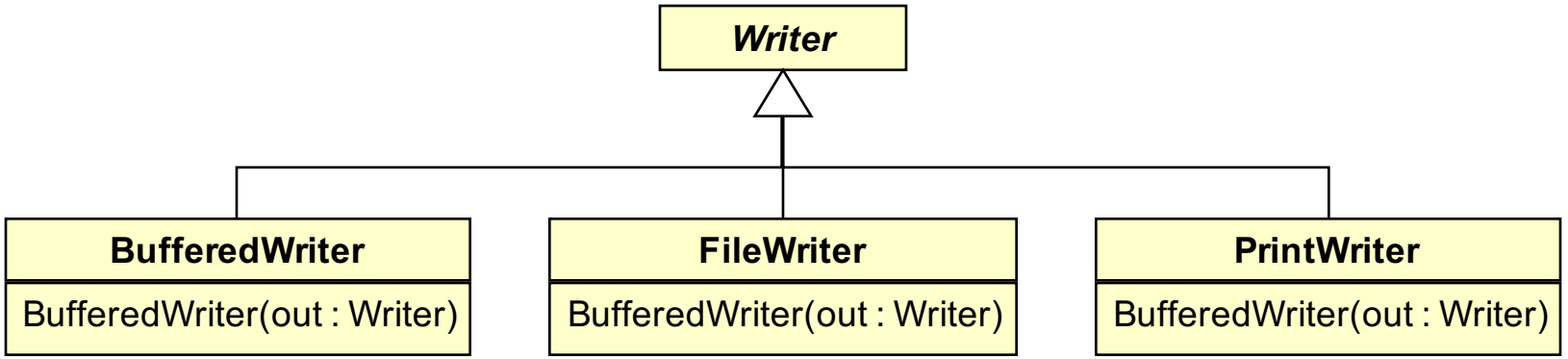
universitätbonn ROOTS

Design Patterns “in the Wild”

Einige Beispiele aus dem Java JDK

(Notation teilweise vereinfacht)

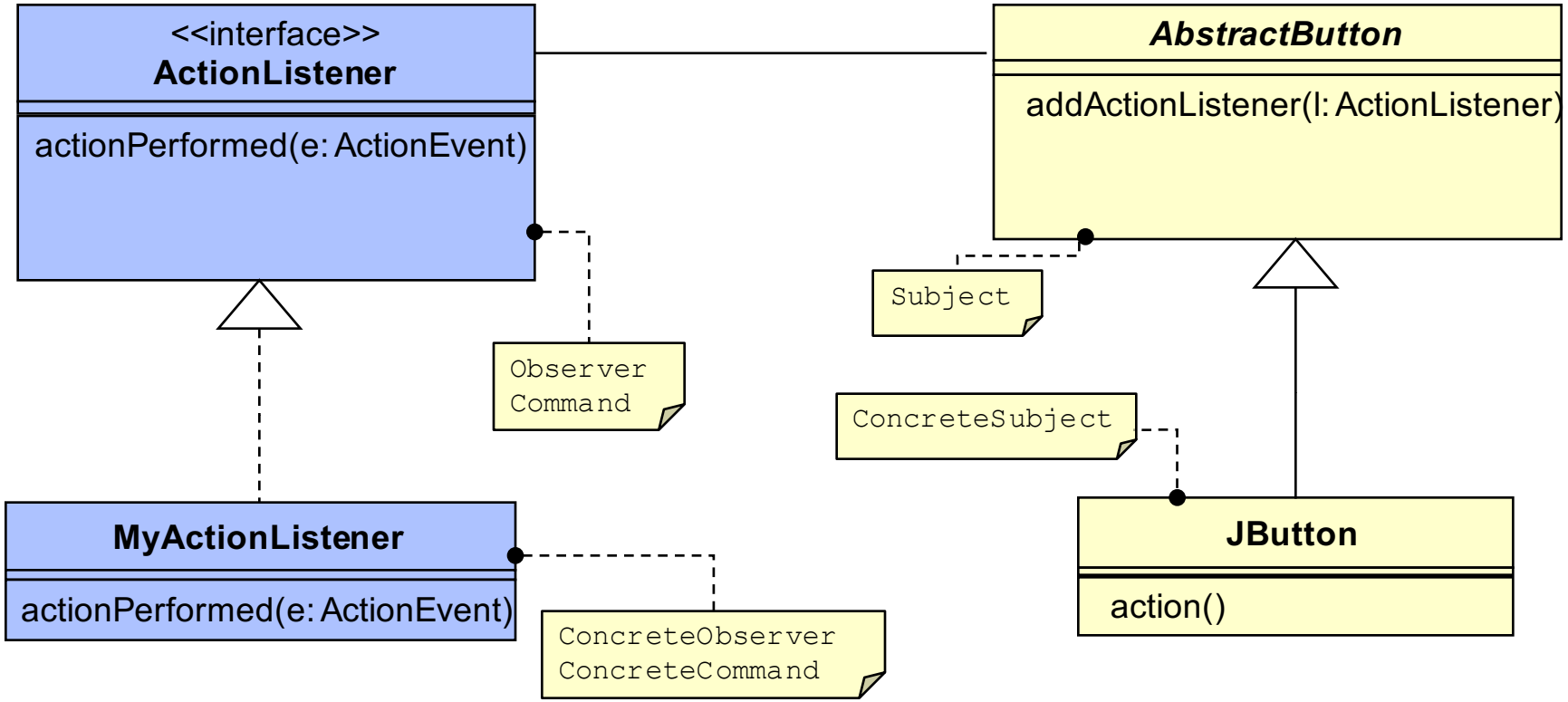
Decorator ▸ Writer Hierarchie




```
Writer writer = new PrintWriter(
                    new BufferedWriter(
                        new FileWriter(„file.txt“)));
```

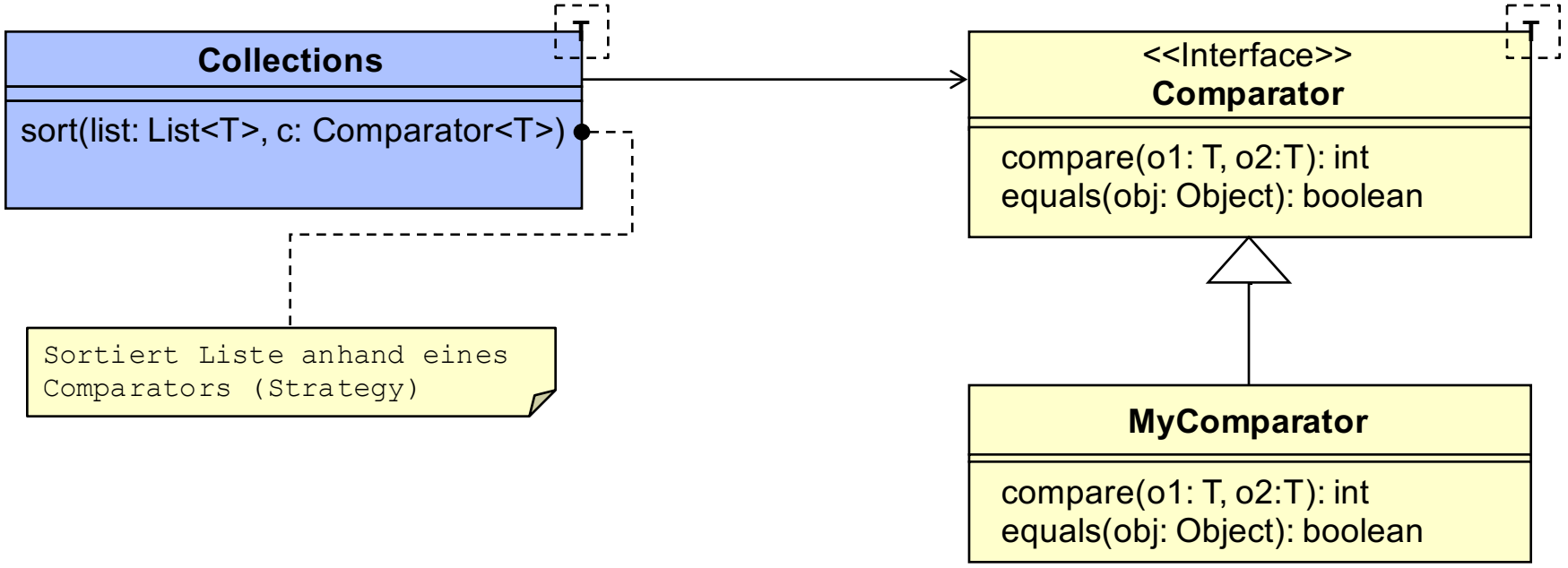
- Ziel: Wir wollen die Vorteile des PrintWriters benutzen und damit in eine Datei schreiben.
  - Der PrintWriter dekoriert einen BufferedWriter (Performance!), der wiederum einen FileWriter (Dateizugriff) dekoriert. Von außen sehen wir nur den PrintWriter.
- *Decorators kommen überall in Java IO zum Einsatz (Streams, Writer, Reader)*

Observer/Command ▸ ActionListener



- Observer Push Modell: Der Button sendet ein ActionEvent Objekt an alle seine Listener
- Command: MyActionListener ist gleichzeitig das ConcreteCommand, das an den Button geknüpft ist
  - *MyActionListener wird von Entwickler implementiert, der Rest ist Teil des JDKs*

Strategy ▸ Comparator



- Collections .sort() sortiert eine Collection anhand eines Comparators, der dazu dient zwei Elemente zu vergleichen.
  - Die konkrete Methode zum Vergleichen von Elementen kann vom Entwickler selbst implementiert werden (→ Bsp: Ordnen nach Name, Alter, etc… austauschbar)

Diverse

- **Factory:**
  - Calendar.getInstance()
    - ⇨ Gets a calendar using the default time zone and locale. The Calendar returned is based on the current time in the default time zone with the default locale.
- **Singleton:**
  - Runtime.getRuntime()
    - ⇨ Returns the runtime object associated with the current Java application.
- **Adapter:**
  - Arrays.asList(…)
    - ⇨ Returns a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.) This method acts as bridge between array-based and collection-based APIs.

Diverse

- **Proxy:**
  - java.rmi (whole API)
    - ⇨ Mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine.
- **Command:**
  - Runnable
    - ⇨ The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- Die Implementierung des JDK enthält unzählige weitere Beispiele, in denen Design Patterns verwendet werden.
  - ➢ Einige weitere Beispiele in einem Thread von Stackoverflow*

* Keine Garantie auf Korrektheit

universitätbonn ROOTS

Design Patterns in Game Engines

Verwendungsbeispiele von Design Patterns im Game Engine Design

(Mehr Details und Beispiele: Originalquelle)

Beispiel 1: State Pattern

- Wir gehen von einem ganz generischen Spiel aus, in dem der Spieler seine Spielfigur mit Tastatur und Maus durch die Welt bewegt.

- Je nach Eingabe führt die Spielfigur Aktionen aus:
  - WASD: Bewegung
  - Leertaste: Springen
  - Mausklick: Angriff

- Erste Implementierung:

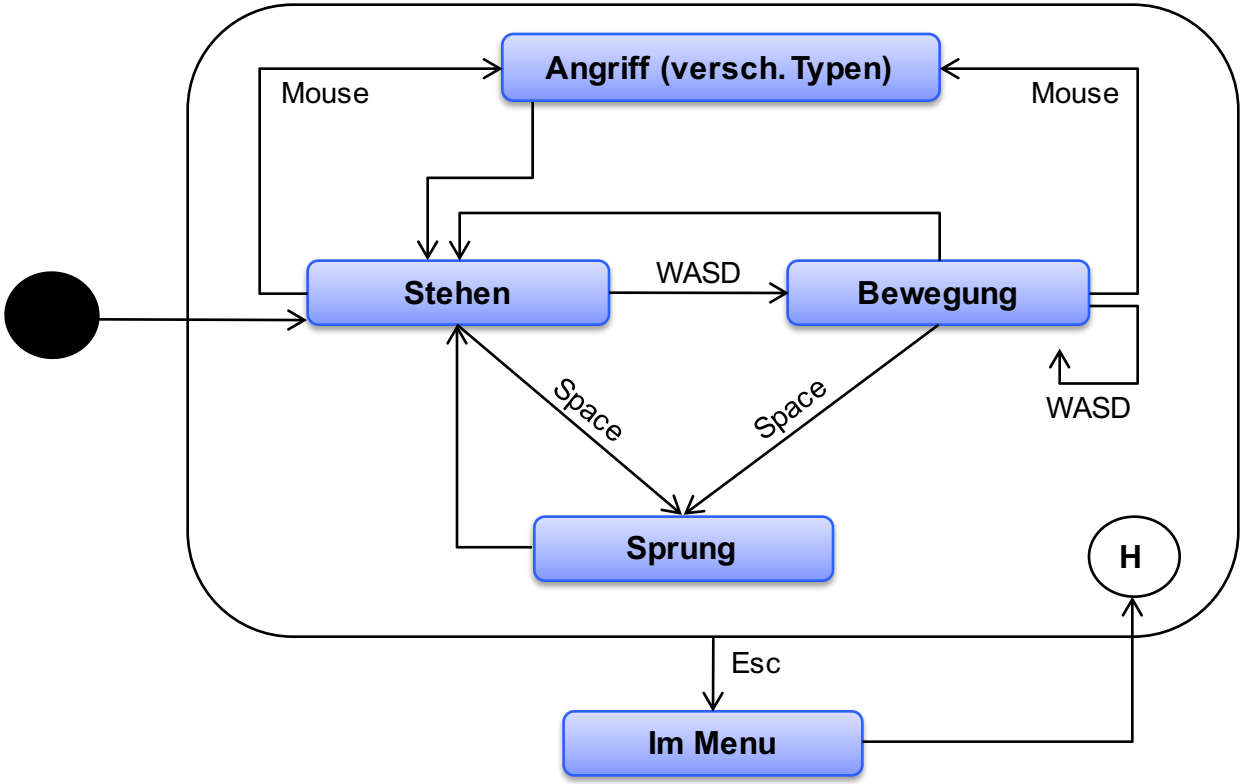
```
class Player{
   handleInput(input){
      switch(input)
         case WASD: Move one meter north/west/south/east
         case Space: Move one meter up
         case Mouse: Attack
      }
   }
```

Beispiel 1: State Pattern

- Problem:
  - Der Spieler drückt die Leertaste → Figur springt einen Meter nach oben.
  - *Während* des Sprungs drückt er nochmal Leertaste → die Figur springt jetzt auf zwei Meter, dann nochmal..., der Spieler kann praktisch fliegen.
- Lösung:
  - Wir setzen eine Variable `isJumping`, und ignorieren weitere Sprungbefehle wenn `isJumping` gesetzt ist.
- Problem: Weitere Bedingungen:
  - Im Angriff darf nicht gesprungen werden.
  - Bei bestimmten Angriffen kann sich die Figur bewegen, bei einigen nicht oder nur langsam.
  - Ist ein Menü geöffnet dürfen Mausklicks keine Angriffe auslösen.
  - …. → Komplexität explodiert, in jedem `case` komplexe if-Konstrukte!

Beispiel 1: State Pattern

- Bedingungen beschreiben Übergänge zwischen *Zuständen* der Spielfigur als endlichen Automat.
- In jedem Zustand wird die Eingabe anders behandelt (z.B. ignoriert)

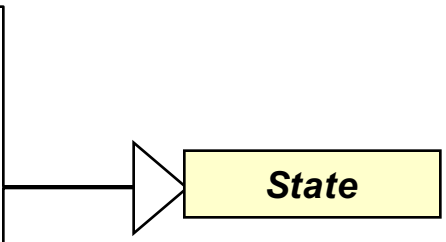


Beispiel 1: State Pattern

- Implementierung des Automaten mit State Pattern:
  - Jeder Zustand wird als *ConcreteState* realisiert, der je nach Zustand des Spielers auf die Eingaben korrekt reagiert:

```
class JumpingState extends State{
   handleInput(input){
      switch(input)
         case WASD:  Do nothing
         case Space: Do nothing
         case Mouse: Do nothing
      }
   }
```

```
class AttackState_WithMovement extends State{
   handleInput(input){
      switch(input)
         case WASD: Move slowly
         case Space: Do nothing
         case Mouse: Do nothing
      }
   }
```



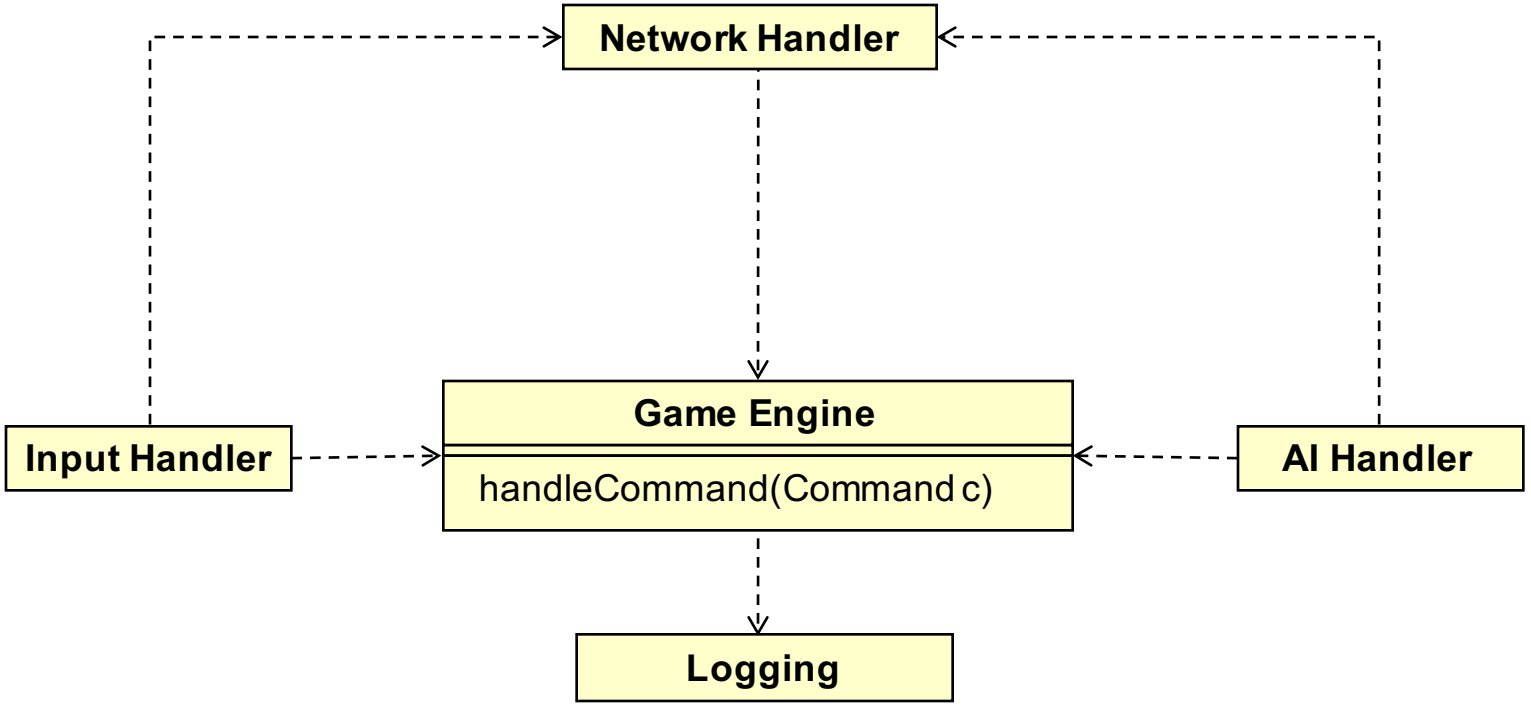
  - Benötigt Mechanismus um *ConcreteStates* auszutauschen
  - Performance: States im Voraus erzeugen und wiederverwenden

Beispiel 2: Command Pattern

- Der Spieler soll die Tasten beliebig mit Befehlen belegen können.
  - Die Lösung kennen Sie schon: Command Pattern
  - Ein InputHandler gibt uns nach einem Tastendruck ein *Command* Objekt, und leitet dieses an die Game Engine weiter, die entsprechend darauf reagiert.

- Erweiterungen:
  - Die KI Komponente, die die Aktionen von NPCs steuert erzeugt ebenfalls *Command* Objekte für die kontrollierten Figuren, und übergibt diese an die Game Engine.
  - *Commands* des Spielers können an eine Netzwerkkomponente weitergegeben werden, um Multiplayer-Sessions zu erlauben. Eingehende Nachrichten werden als Command Objekts dargestellt und an die Game Engine gegeben.
  - *Commands* können an eine Logging-Komponente gegeben werden (für Spielaufnahme oder zu Debug-Zwecken).

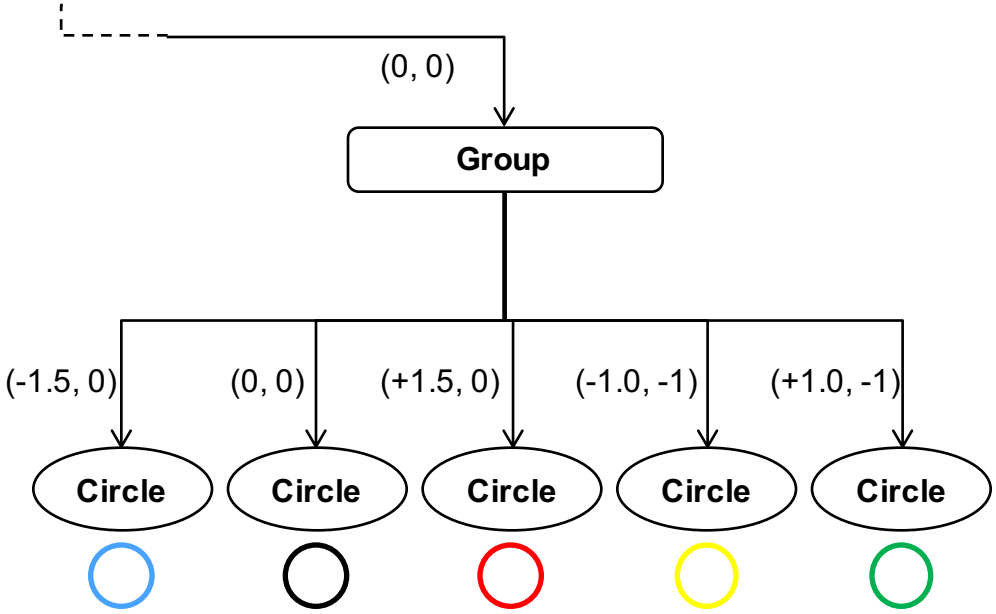
Beispiel 2: Command Pattern



- Performance: *Commands* im Voraus erzeugen und wiederverwenden

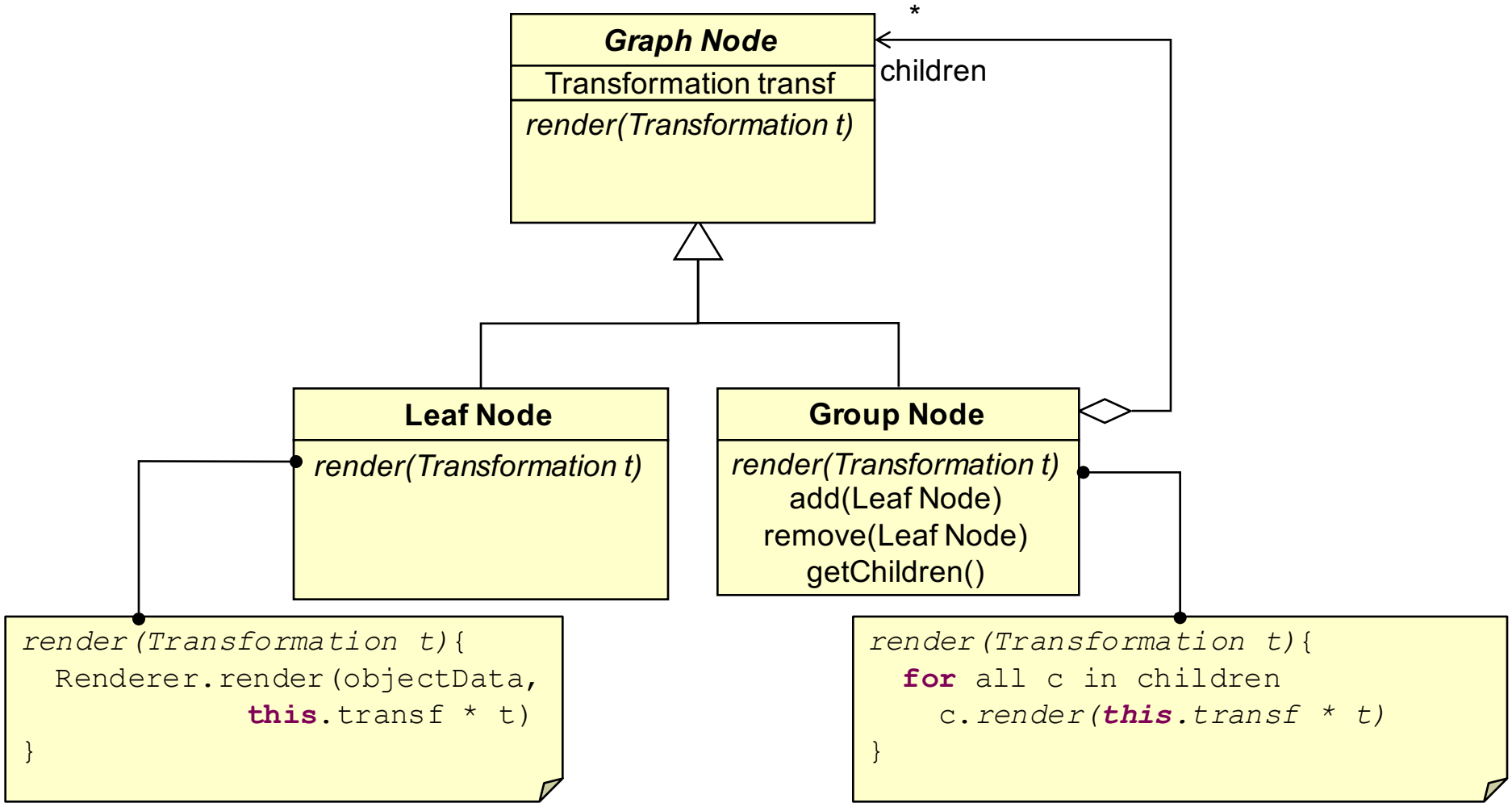
Beispiel 3: Composite Pattern

- Eine Szene wird oft in einem hierarchischem Scene Graph dargestellt.
  - Der Graph stellt sowohl logische als auch Positionsbeziehungen zwischen verschiedenen Objekten dar.
  - In einem einfachen Graph ist jede Kante zwischen einem Knoten und dem Elternknoten mit einer Transformation annotiert, die die relative Verschiebung/Rotation des Kindes zu dem Elternobjekt beschreibt:



Beispiel 3: Composite Pattern

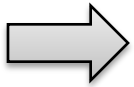
- Implementierung mit Composite Pattern:



Beispiel 4: Flyweight Pattern

Nicht in der Vorlesung, daher im Anhang

- Szenario: Eine Szene mit einem Wald aus hunderten von Bäumen.
- Naive Lösung:
  - Jeder Baum ist ein eigenes Objekt mit Vertex-, Face- und Texturdaten.
  - Resultat: Der Speicherbedarf wächst enorm.
- Lösung mit Flyweight:
  - Vorwissen: Der Wald besteht zwar aus hunderten Bäumen, die Geometrie ist jedoch für alle Bäume identisch, nur die Texturen variieren individuell.
  - Damit müssen wir die Geometrie nur exakt einmal speichern, und wirklich nur die Features, die sich unterscheiden individuell (evtl. werden auch nur 10 Baumtexturen verwendet, die sich wieder zusammenfassen lassen).

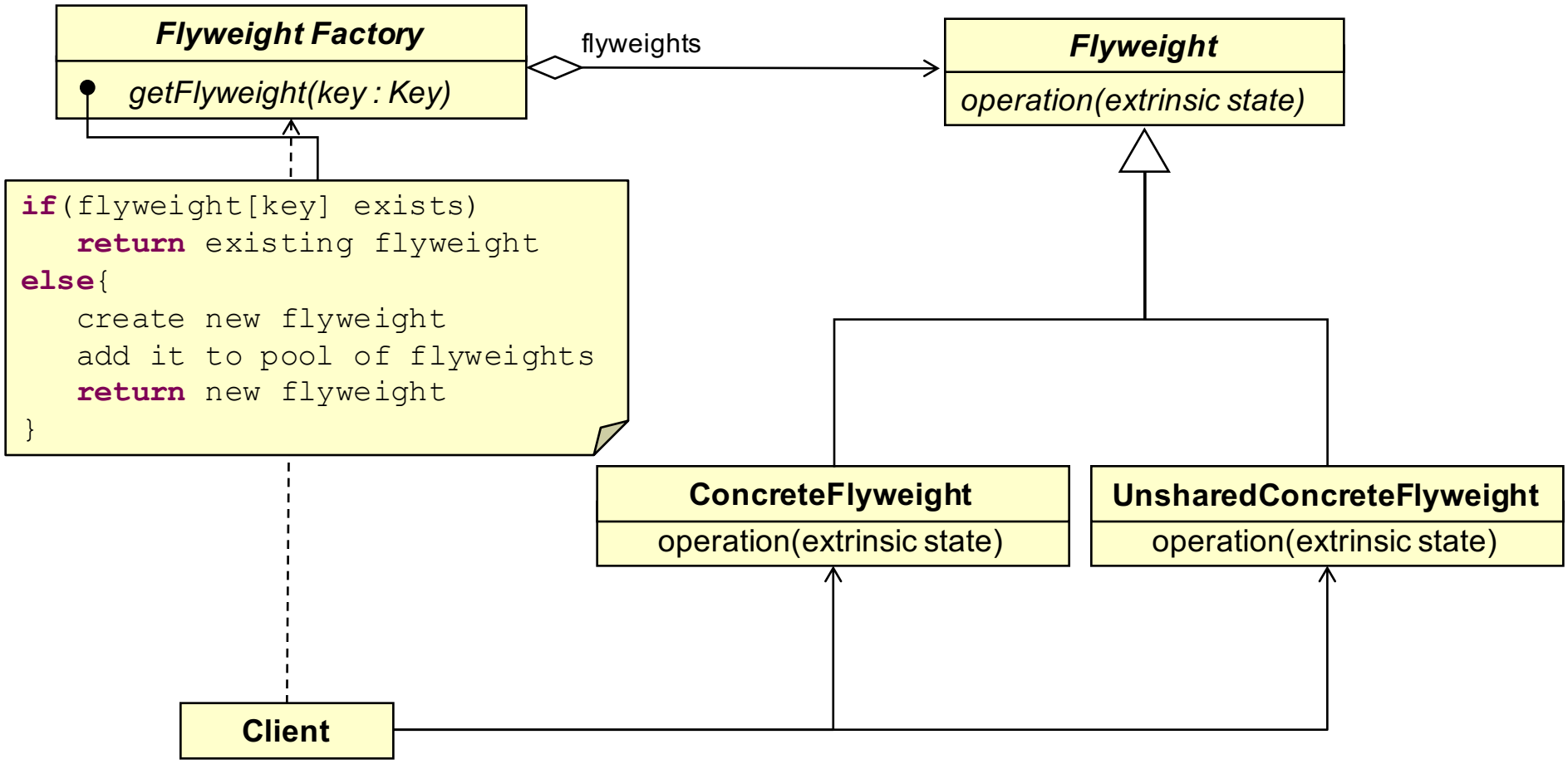


Anmerkung: OpenGL und DirectX unterstützen genau das mit *Instanced Rendering*: Ein Objekt x-mal mit unterschiedlichen Parametern (z.B. Position und Textur) rendern.

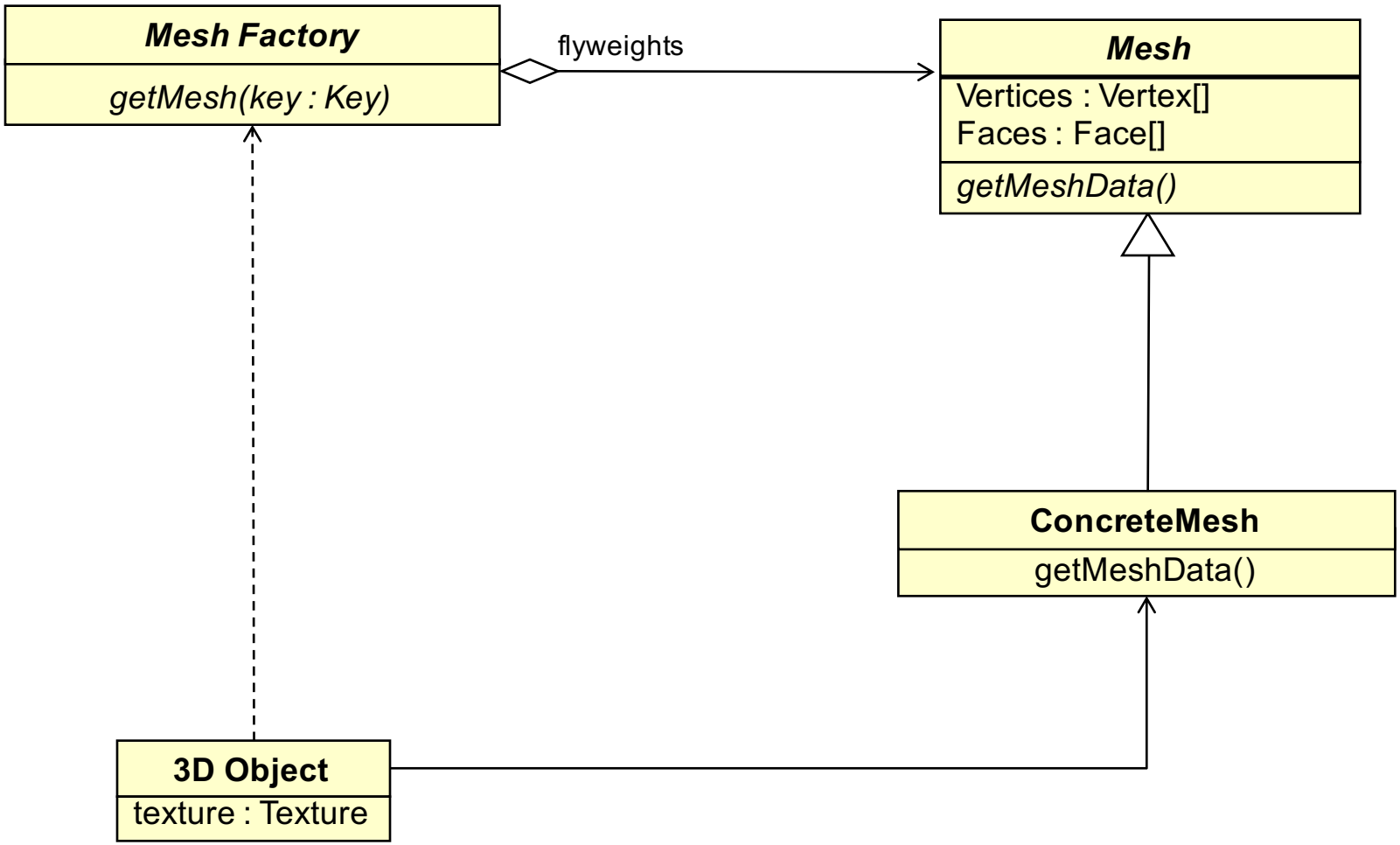
Flyweight ▸ Übersicht

- **Kontext:**
  - System muss sehr viele Objekte vorhalten, deren Inhalt teilweise identisch ist.
- **Absicht:**
  - Ermögliche eine große Menge von Objekten, die gemeinsame Teile ihres Zustands teilen.
- **Motivation:**
  - Extrinsischer Zustand: Kann ausgelagert werden, wird zwischen Objekten geteilt
  - Intrinsischer Zustand: Kann nicht ausgelagert werden, Objektspezifisch
    - ⇨ Idee: Lagere Teil des Zustands aus, und teile ihn zwischen Objekten.
- **Konsequenzen:**
  - Reduzierung von Speicherbedarf
  - Anstieg der Komplexität

Flyweight ▸ Schema



Beispiel 4: Flyweight Pattern

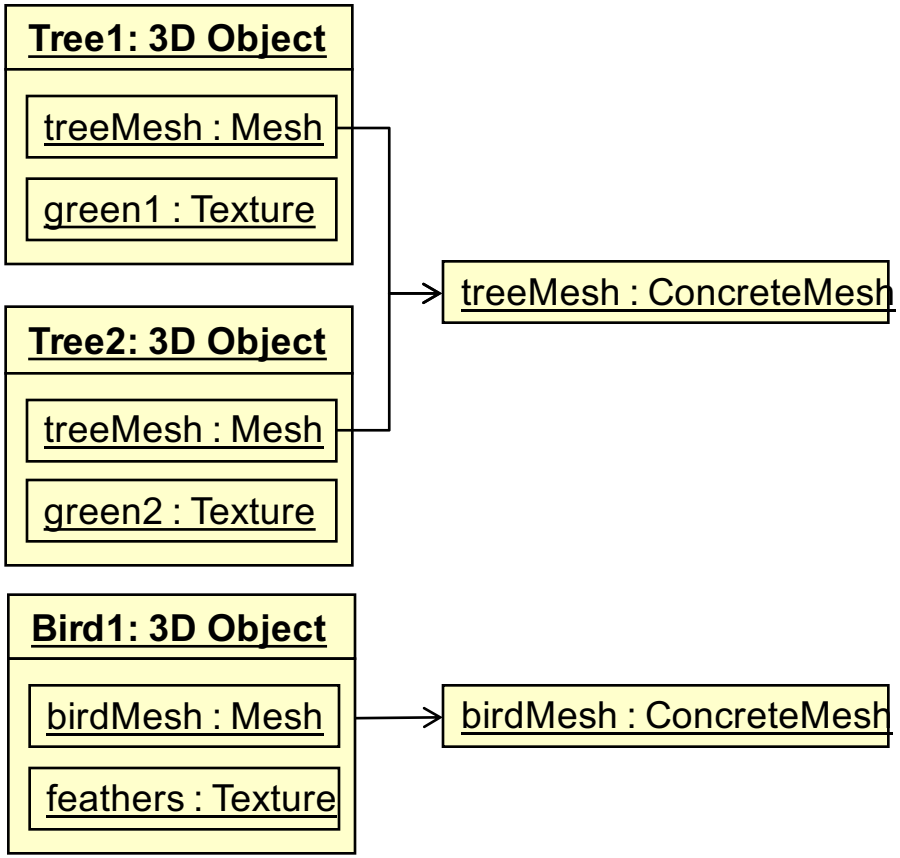


Beispiel 4: Flyweight Pattern

```
class 3DObject{

   private Mesh mesh;
   private Texture texture;

   3DObject(Key key){
      mesh = MeshFactory.getMesh(key);
   }
}
```

```
class MeshFactory{

   private Map<Key, Mesh> meshes;

   static Mesh getMesh(Key key){
      Mesh m = meshes.get(key);
      if(m == null)
         m = new Mesh(key);
         meshes.put(key, m);
      }
      return m;
   }
}
```

Selbst bei 1000 Baum-Instanzen wird das das gemeinsame Mesh nur einmal gespeichert!



Anhang

Pattern ▸ Literaturempfehlungen

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

  "Design Patterns - Elements of Reusable Object-Oriented Software“

- Bernd Brügge, Allen Durtoit

  „Object-Oriented Software Engineering“

- Robert C. Martin

  „Agile Software Development: Principles, Patterns, and Practices“

- **Vorlesungsfolien SWT 2013/2014**
  - **http://sewiki.iai.uni-bonn.de/teaching/lectures/se/2013/folien**
  - **Kapitel 7: Entwurfsmuster**
  - **Kapitel 8: Systementwurf**
  - **Kapitel 9: Objektentwurf**