

# Kapitel 3.5 – Entwurfsmuster

**SWT I – Sommersemester 2018**

**Walter F. Tichy, Sebastian Weigelt, Tobias Hey**

IPD Tichy, Fakultät für Informatik



# Definition

Ein Software-Entwurfsmuster beschreibt eine Familie von Lösungen für ein Software-Entwurfsproblem.

- Der Zweck von Entwurfsmustern ist die **Wiederverwendung** von Entwurfswissen.
- Entwurfsmuster sind für den Entwurf (oder das Programmieren im Großen) was Algorithmen für das Programmieren im Kleinen sind.

# Software-Entwurfsmuster sind dem Konzept der Entwurfsmuster aus der Architektur und Städteplanung entlehnt.

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

Quelle: Christopher Alexander et al:  
A Pattern Language,  
Oxford University Press, New York, 1977

Christopher Alexander ist ein US-amerikanischer Architekt, Architekturtheoretiker und Systemtheoretiker.

# Wozu überhaupt Entwurfsmuster?

- Muster verbessern die **Kommunikation** im Team
  - Entwurfsmuster bilden eine nützliche Terminologie, d.h. sie bieten Begriffe und Kurzformeln für die Diskussion zwischen Entwicklern über komplexe Konzepte.
- Muster erfassen wesentliche **Konzepte** und bringen sie in eine verständliche Form
  - Muster helfen Entwürfe zu verstehen
  - Muster dokumentieren Entwürfe kurz und knapp
  - Muster verhindern Architektur-Drift (d.h. Verschlechterung der ursprünglichen Architektur bei Änderungen)
  - Muster verdeutlichen Entwurfswissen

# Wozu überhaupt Entwurfsmuster?

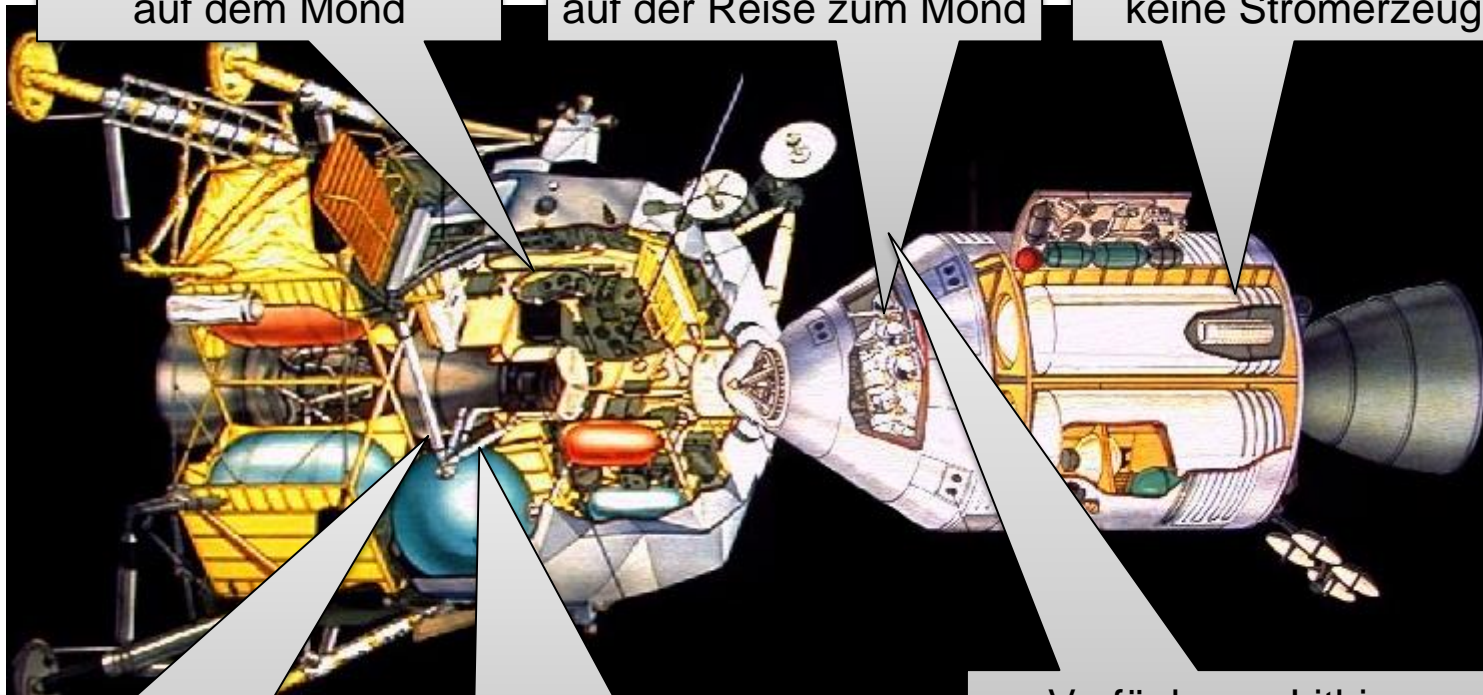
- Muster **dokumentieren** und **fördern** den Stand der Kunst
  - Muster helfen weniger erfahrenen Entwerfern
  - Muster vermeiden die Neuerfindung des Rades
  - Ein Muster ist keine feste Regel, der man blind folgt, sondern ein Vorschlag und ein Satz von Alternativen zur Lösung eines Problems. Anpassung erforderlich!
- Muster können Code-Qualität und Code-Struktur **verbessern**
  - Muster fördern gute Entwürfe und guten Code durch Angabe konstruktiver Beispiele

# Apollo 13 (1970): “Houston, we’ve had a Problem!”

Mondlandemodul (LM):  
Platz für 2 Astronauten  
auf dem Mond

Kommandomodul (CM):  
Platz für 3 Astronauten  
auf der Reise zum Mond

Servicemodul (SM):  
Sauerstofftank explodiert,  
keine Stromerzeugung



Verfügbares Lithium-  
Hydroxid: 60 Stunden für 2  
Astronauten

Benötigt wurde jedoch:  
88 Stunden für 3  
Astronauten

Verfügbares Lithium-  
Hydroxid im CM:  
„Mehr als genug.“



# Apollo 13 (1970):

## “Houston, we’ve had a Problem!”

- Das Mondlandemodul war für 60 Stunden und 2 Astronauten (2 Tage Aufenthalt auf dem Mond) ausgelegt.
- Herausforderung bei der Neuplanung:
  - Kann das Mondlandemodul für 12 Manntage als „Rettungsboot“ genutzt werden (2,5 Tage bevor zur Erde zurückgekehrt wird)?
- Vorschlag:
  - Verwende die Lithium-Hydroxid-Behälter des Kommandomoduls, um CO<sub>2</sub> aus der Atemluft zu filtern
- Problem:
  - Nicht kompatible Anschlüsse zwischen den Behältern aus dem Mondlandemodul und dem Kommandomodul.

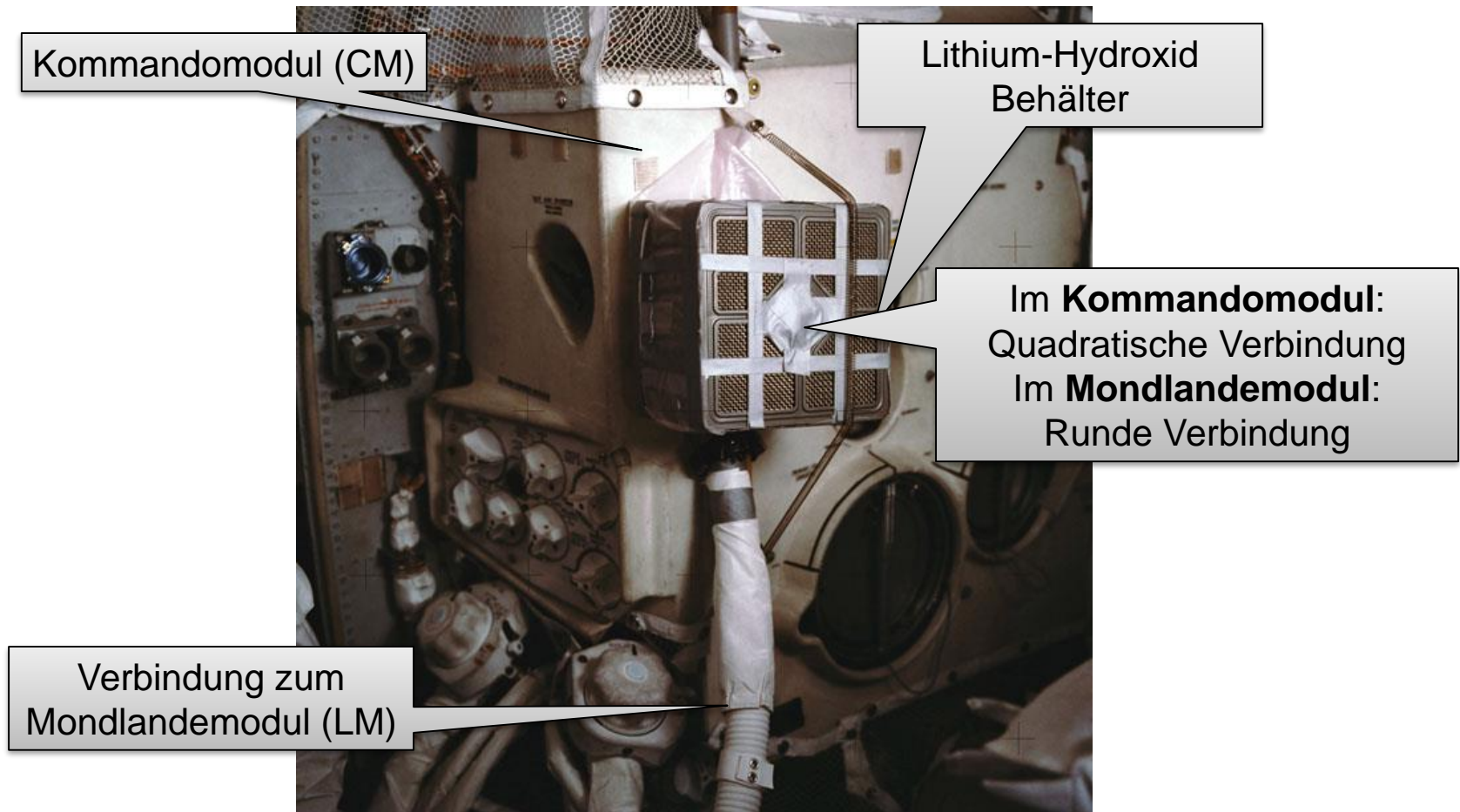
# Apollo 13:

## “Was nicht passt, wird passend gemacht.”





# Adapter: Verbinde nicht kompatible Komponenten miteinander



Quelle: <http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html>

# Entwurfsmuster: Kategorien (1)

## 1. Entkopplungs-Muster

- Entkopplungs-Muster **teilen** ein System in mehrere Einheiten, so dass einzelne Einheiten **unabhängig** voneinander erstellt, verändert, ausgetauscht und wiederverwendet werden können. Der Vorteil von Entkopplung ist, dass ein System durch **lokale Änderungen** verbessert, angepasst und erweitert werden kann, ohne das ganze System zu modifizieren.
- Mehrere der Entkopplungsmuster enthalten ein **Kopplungsglied**, das entkoppelte Einheiten über eine **Schnittstelle** kommunizieren lässt. Diese Kopplungsglieder sind auch für das Koppeln unabhängig erstellter Einheiten brauchbar.

# Entkopplungsmuster

- Adapter
- Beobachter
- Brücke
- Iterator
- Stellvertreter
- Vermittler

# Entwurfsmuster: Kategorien (2)

## 2. Varianten-Muster

- In Mustern dieser Gruppe werden **Gemeinsamkeiten** von verwandten Einheiten aus ihnen **herausgezogen** und an einer einzigen Stelle beschrieben.
- Aufgrund ihrer Gemeinsamkeiten können unterschiedliche Komponenten im gleichen Programm danach einheitlich verwendet werden, und **Wiederholungen** desselben Codes werden **vermieden**.

# Varianten-Muster

- Abstrakte Fabrik
- Besucher
- Erbauer
- Fabrikmethode
- Kompositum
- Schablonenmethode
- Strategie
- Dekorierer

Nicht prüfungsrelevant



# Entwurfsmuster: Kategorien (3)

## 3. Zustandshandhabungs-Muster

- Die Muster dieser Kategorie bearbeiten den **Zustand** von Objekten, unabhängig von deren Zweck.
- Einzelstück
- Fliegengewicht
- Memento
- Prototyp
- Zustand

# Entwurfsmuster: Kategorien (3)

## 4. Steuerungs-Muster

- Steuerungsmuster steuern den **Kontrollfluss**
  - Sie bewirken, dass zur richtigen Zeit die richtigen Methoden aufgerufen werden
- 
- Befehl
  - Master/worker

# Entwurfsmuster: Kategorien (3)

## 5. Virtuelle Maschinen

- Virtuelle Maschinen erhalten Daten und ein Programm als Eingabe und führen das Programm **selbständig** an den Daten aus.
- Virtuelle Maschinen sind in Software, nicht in Hardware implementiert.

■ Interpretierer

Nicht prüfungsrelevant

# Entwurfsmuster: Kategorien (3)

## 6. Bequemlichkeitsmuster

- Diese Muster **sparen** etwas Schreib- oder Denkarbeit.
- Bequemlichkeits-Klasse
- Bequemlichkeits-Methode
- Fassade
- Null-Objekt

# Komplexes Beispiel „Entensimulation“

- Komplexes, „praktisches“ Beispiel, das verschiedene Muster kombiniert
- Lesen, **verstehen** und die Aufgaben/Fragen bearbeiten
- Vielleicht sogar: nachbauen
- Enthält (und deswegen behandeln wir in der Vorlesung etwas straffer)
  - Adapter
  - Dekorierer
  - Abstrakte Fabrik
  - Kompositum
  - Iterator
  - Beobachter

**Nachlesen:** Head First Design Patterns, Kapitel 12



# Entkopplungsmuster

IPD Tichy, Fakultät für Informatik



# Entkopplungsmuster: Überblick

- Adapter
- Beobachter
- Brücke
- Iterator
- Stellvertreter
- Vermittler

# Adapter (engl. *adapter*)

## ■ Zweck

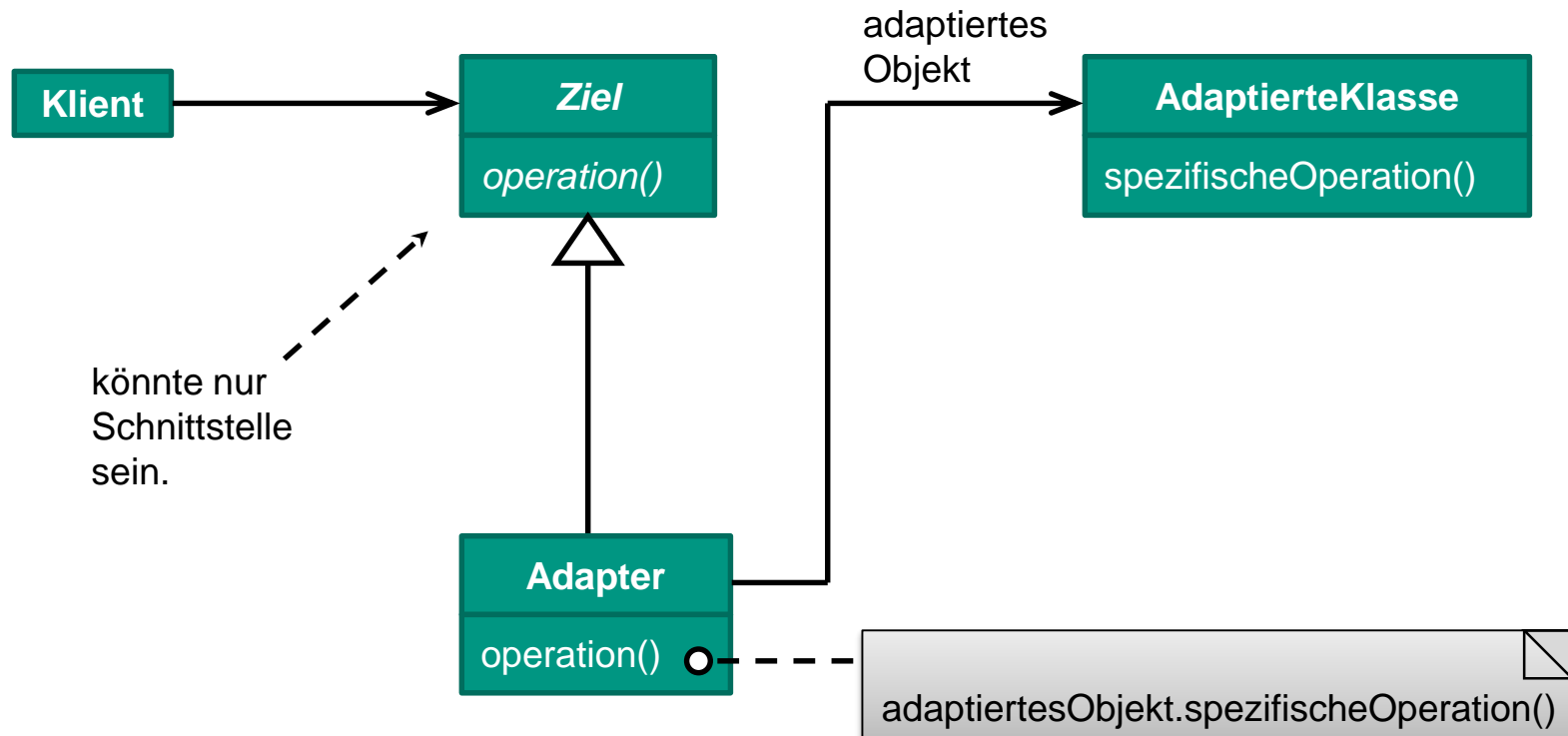
- Passe die **Schnittstelle** einer Klasse **an** eine andere von ihren Klienten erwartete Schnittstelle an
- Das Adaptermuster lässt Klassen zusammenarbeiten, die wegen **inkompatibler Schnittstellen** ansonsten dazu nicht in der Lage wären.



## ■ Synonyme: Wandler, Umwickler (engl. *wrapper*)

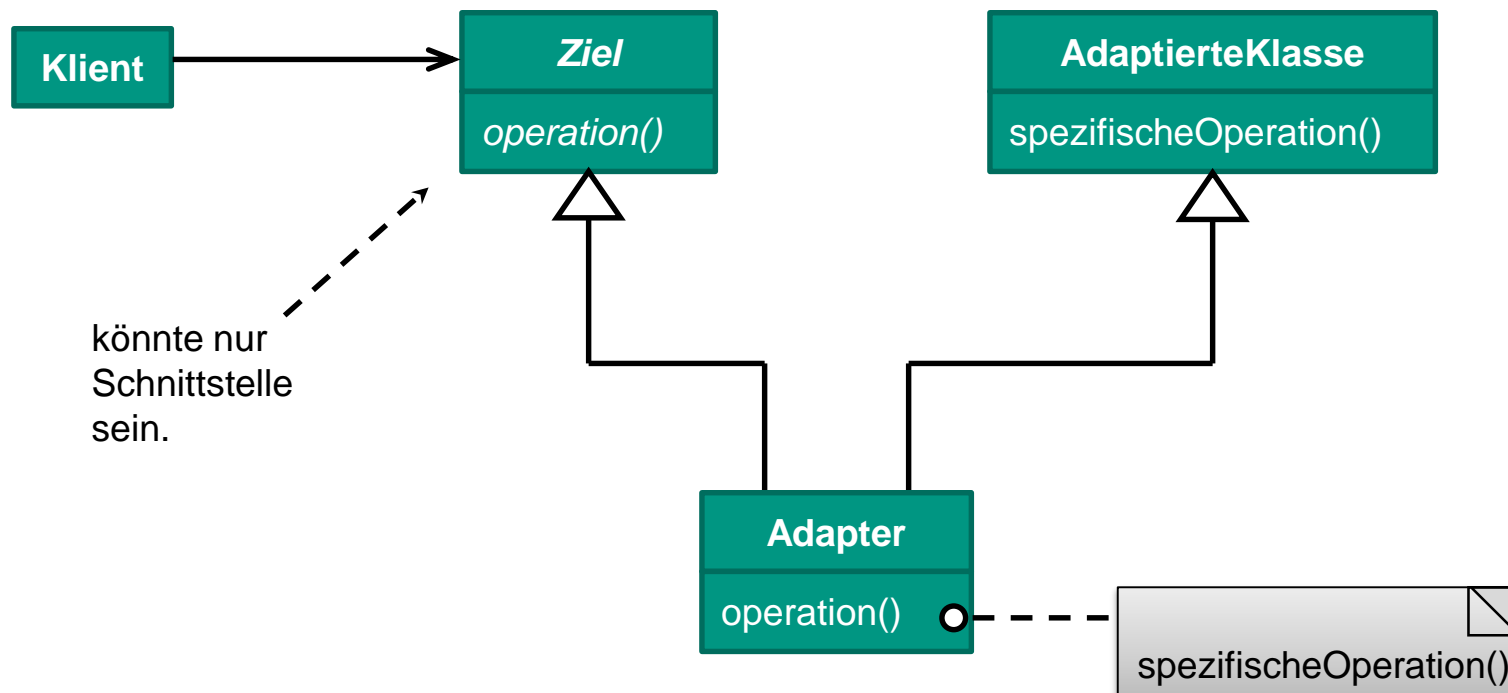
# Adapter: Struktur (1)

## ■ Ohne Mehrfachvererbung (Objektadapter)



# Adapter: Struktur (2)

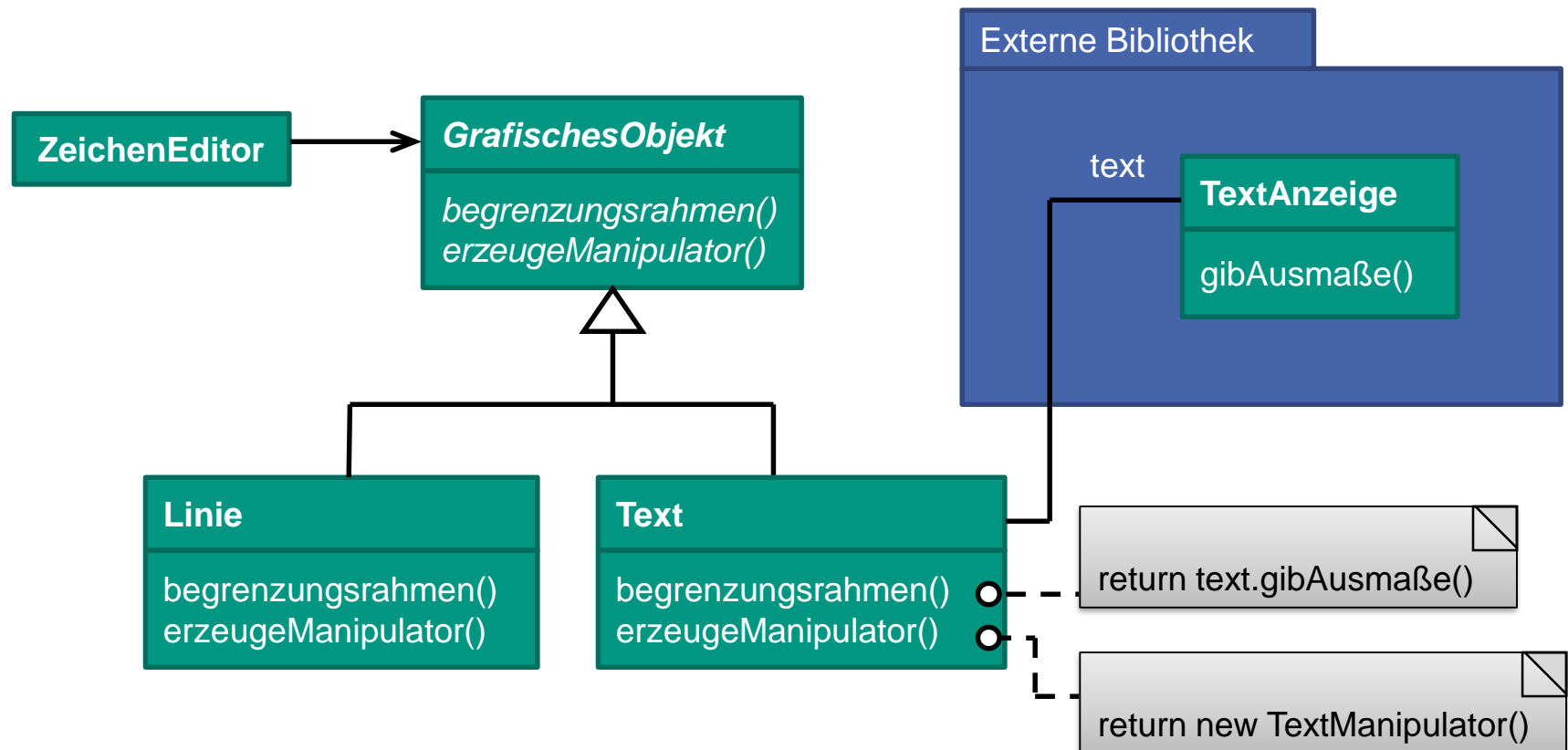
## ■ Mit Mehrfachvererbung (Klassenadapter)





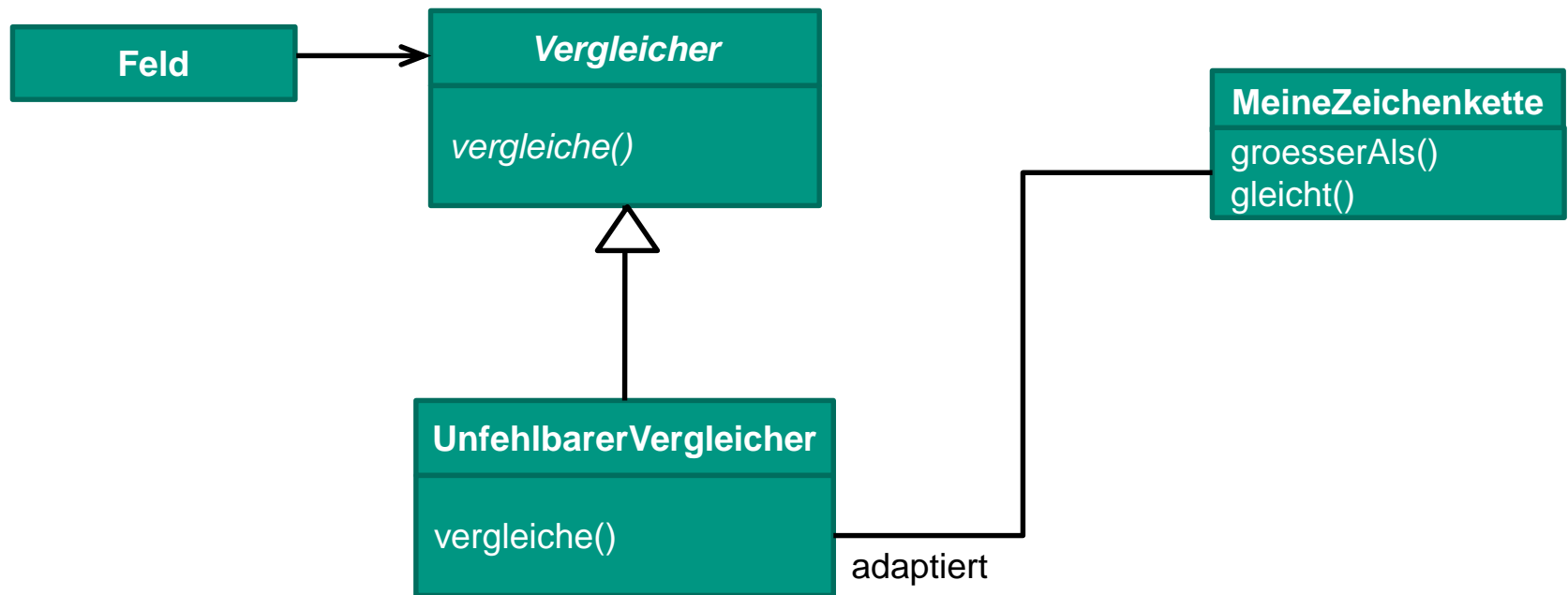
# Adapter: Beispiel (1)

- Verwendung einer externen Klassenbibliothek zur Anzeige von Texten in einem Zeicheneditor.



## Adapter: Beispiel (2)

- Sortieren von Zeichenketten in einem Feld.



## Adapter: Anwendungsbeispiel

- Apple Macintosh Computer wechselten 2006 von PowerPC-Prozessoren zu Intel-Prozessoren mit x86/Intel 64 Befehlssatz.
- MacOS X verwendet bei PowerPC-Prozessoren AltiVec Befehlssatz, bei Intel-Prozessoren die SSE Erweiterungen.
- Damit Software-Entwickler nicht selbst Quelltext sowohl für AltiVec als auch für SSE schreiben müssen, bietet Apple die Accelerate-Bibliothek an, welche, abhängig vom verbauten Prozessor, den korrekten Befehlssatz wählt.

## Adapter: Anwendbarkeit

- Wenn eine **existierende** Klasse verwendet werden soll, deren Schnittstelle aber nicht der benötigten Schnittstelle entspricht (und die benötigte Schnittstelle nicht geändert werden kann).
- Wenn eine **wieder verwendbare** Klasse erstellt werden soll, die mit unabhängigen oder nicht vorhersehbaren Klassen zusammenarbeitet, d.h. Klassen, die nicht notwendigerweise kompatible Schnittstellen besitzen.
- Wenn **verschiedene existierende** Unterklassen benutzt werden sollen, es aber unpraktisch ist, die Schnittstellen jeder einzelnen Unterklasse durch Ableiten anzupassen. Ein Objektadapter ist in der Lage, die Schnittstelle seiner Oberklasse anzupassen.

# Beobachter (engl. *observer*)

## ■ Zweck

- Definiert eine 1-zu-n Abhängigkeit zwischen Objekten, so dass die Änderung eines Zustandes eines Objektes dazu führt, dass alle abhängigen Objekte **benachrichtigt** und **automatisch aktualisiert** werden.

## ■ Synonyme

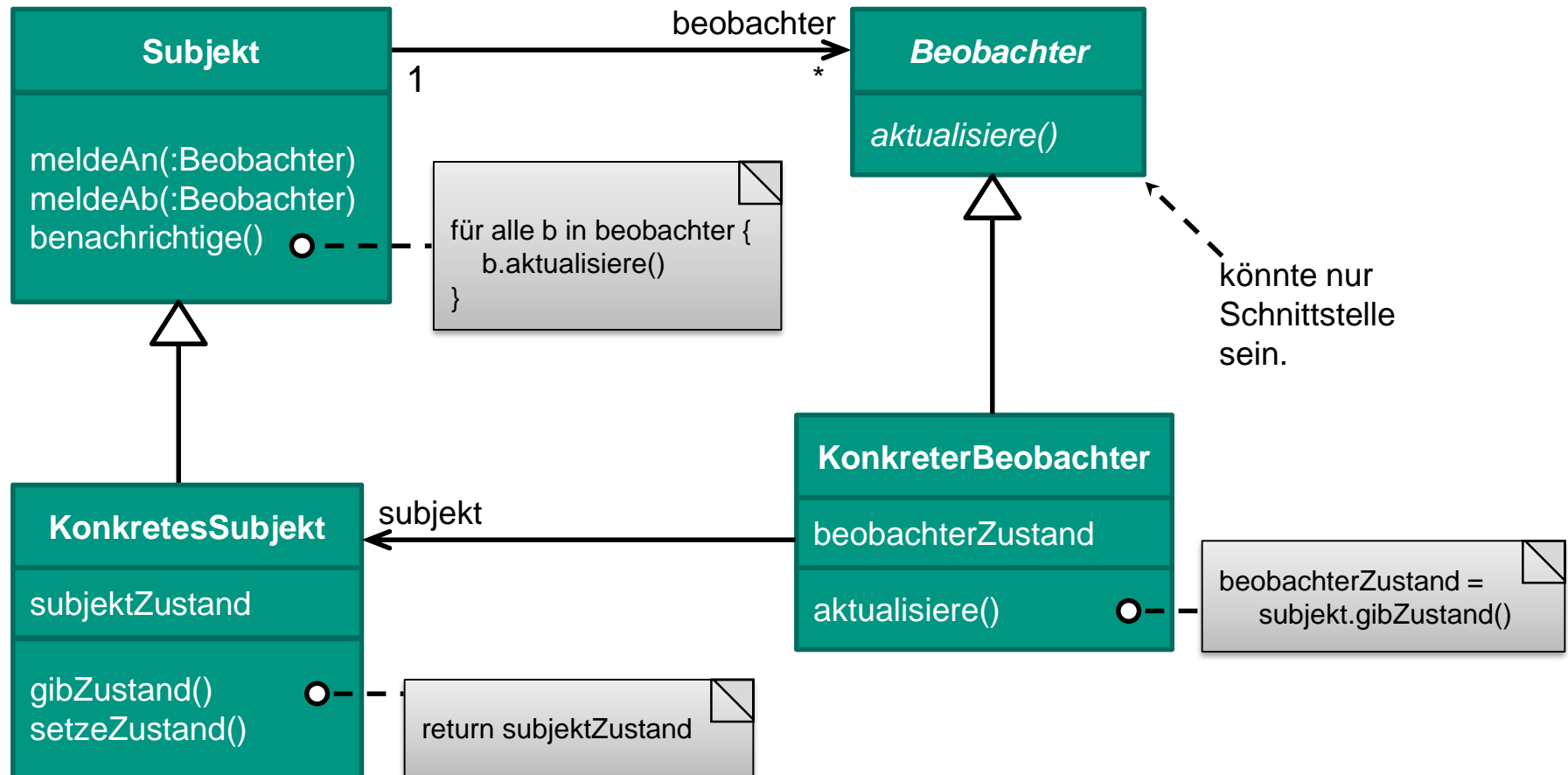
- Abhängigkeit
- Publizieren-Abonnieren (engl. *publisher-subscriber*)
- Subjekt-Beobachter



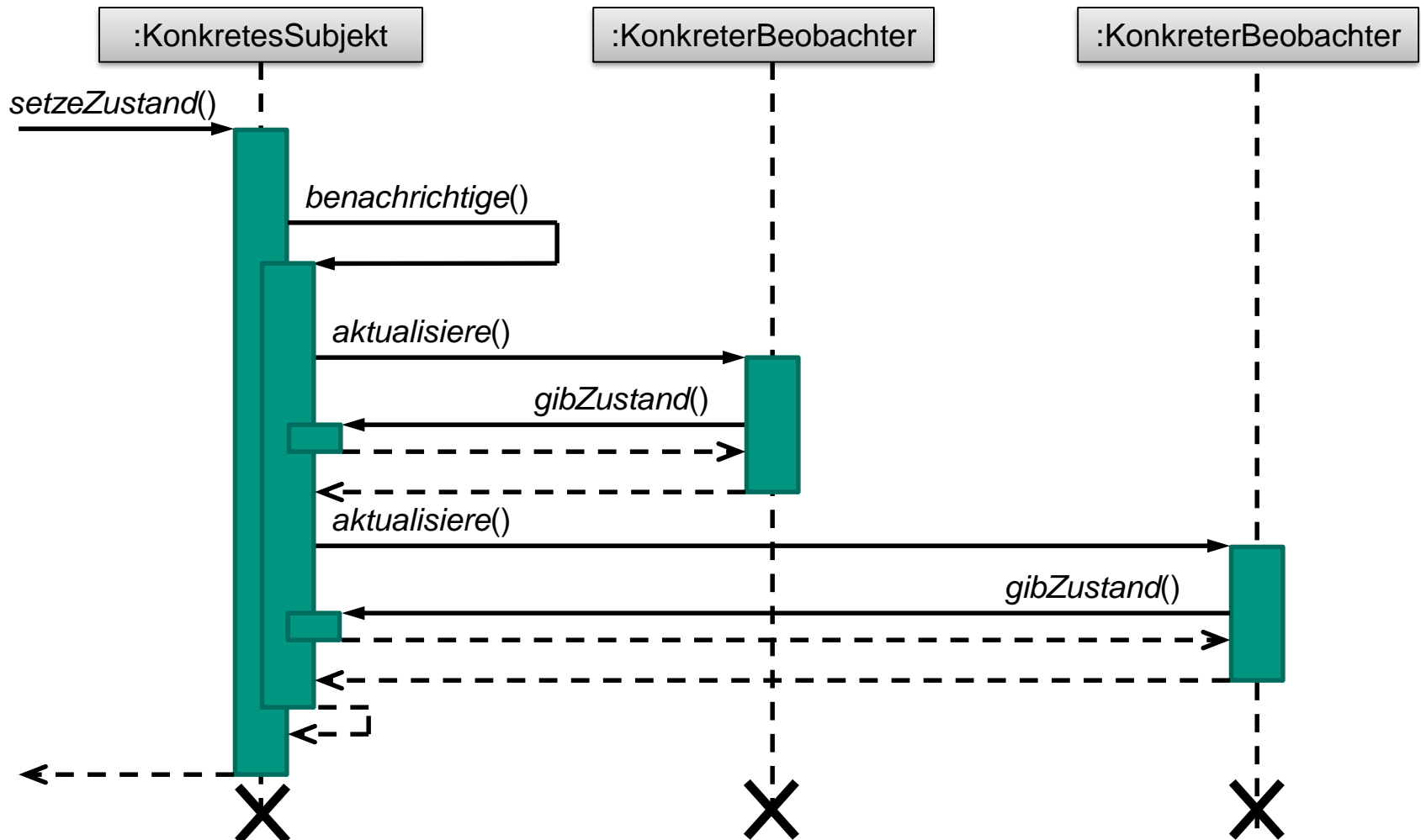
## Beobachter: Motivation

- Teilt man ein System in eine Menge von interagierenden Klassen auf, so muss die **Konsistenz** zwischen den miteinander in Beziehung stehenden Objekten aufrechterhalten werden.
- Eine **enge Kopplung** dieser Klassen ist **nicht empfehlenswert**, weil dies die individuelle Wiederverwendbarkeit einschränkt.
- Im MVC-Beispiel wissen die Tabellendarstellung und die Säulendarstellung nichts voneinander. Damit können sie unabhängig voneinander wieder verwendet werden. Trotzdem verhalten sich beide Objekte so, als ob sie einander kennen würden.

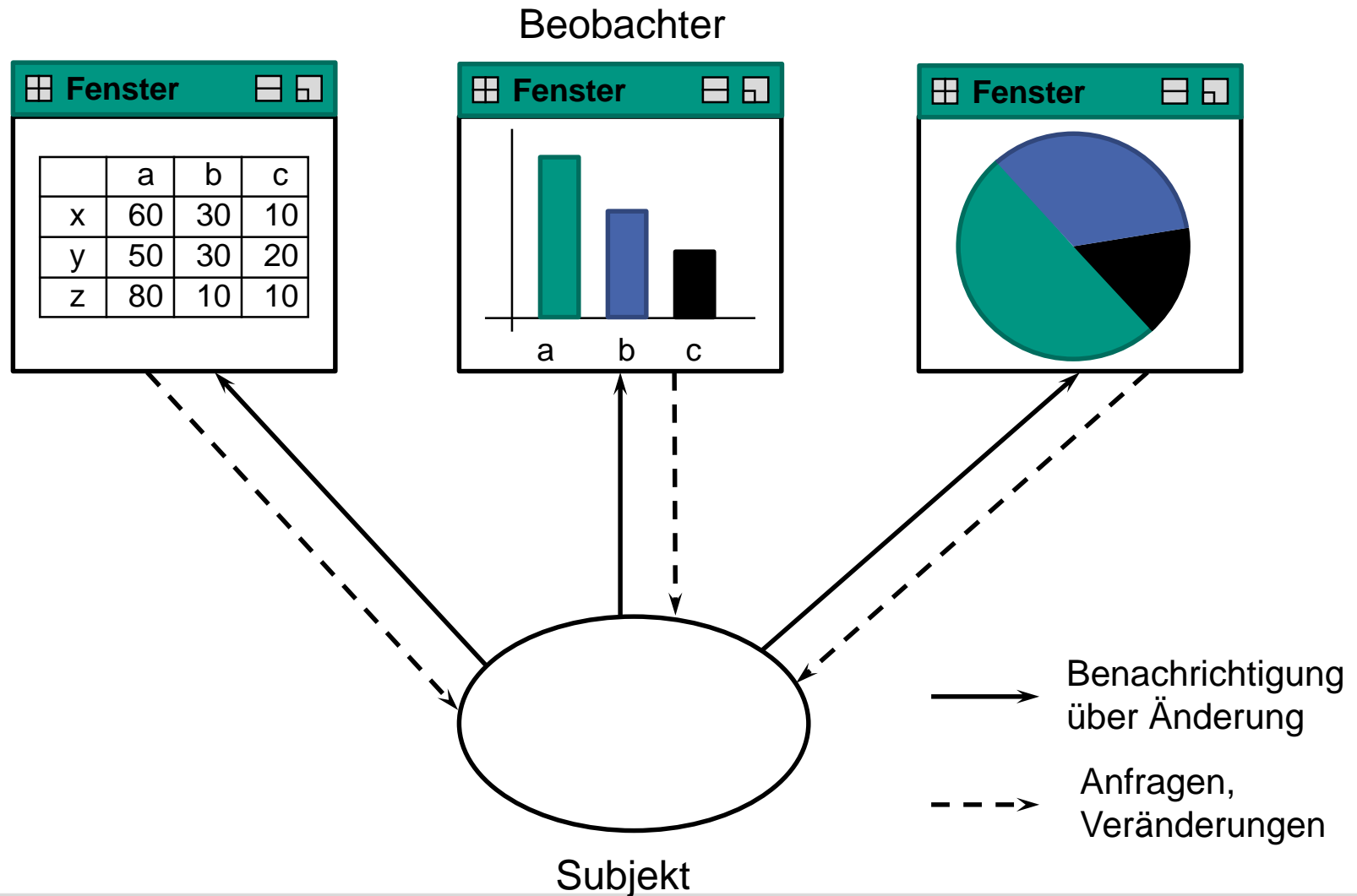
# Beobachter: Struktur



# Beobachter: Interaktionsdiagramm

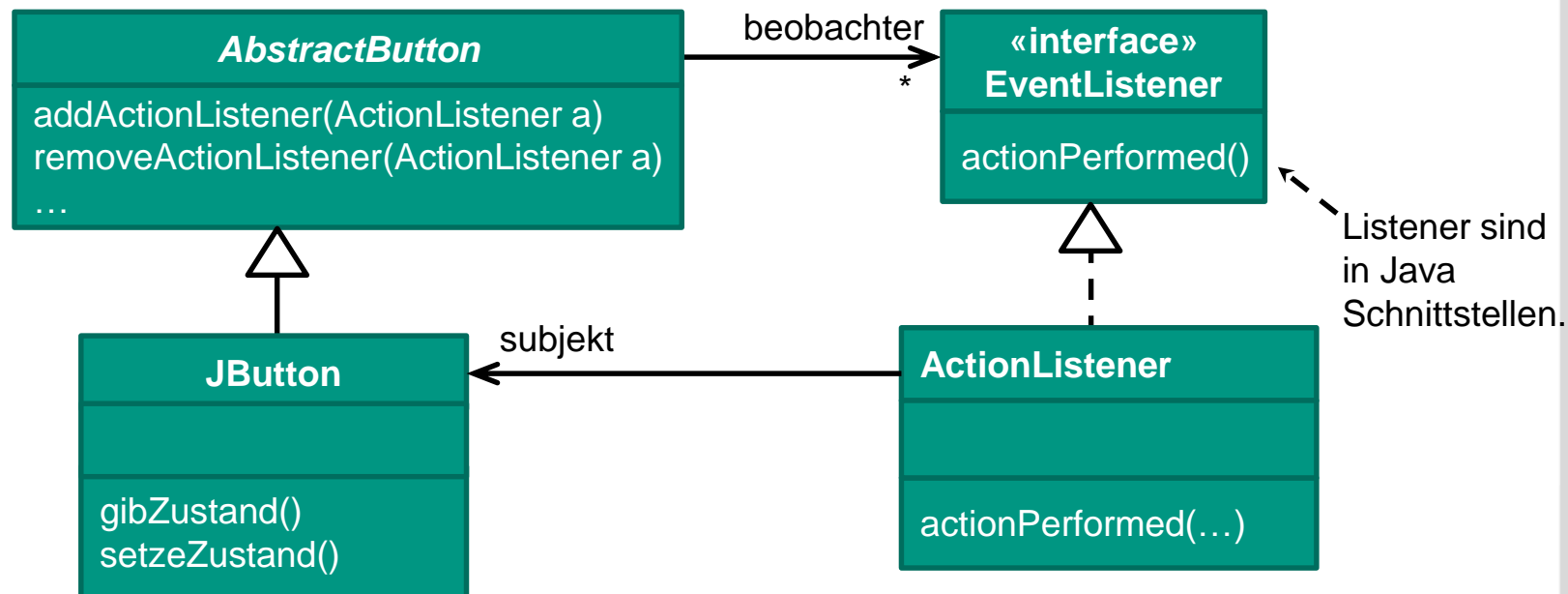


# Beobachter: Beispiel



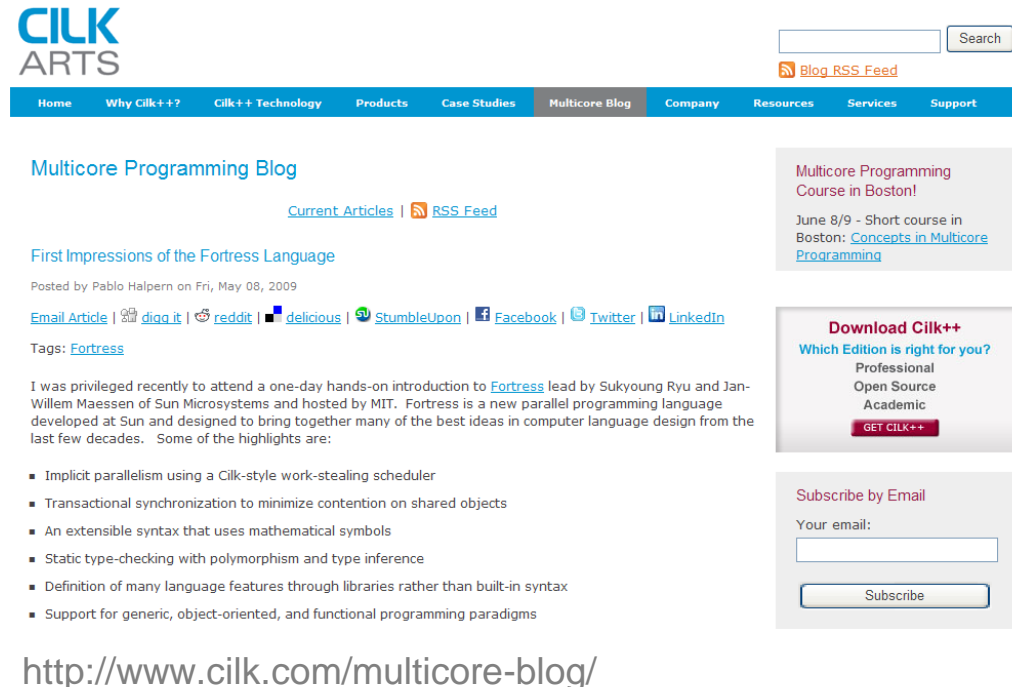
# Beobachter: Beispiel aus Java

- In Java werden bspw. **Ereignisse** mit Beobachtern behandelt.



# Beobachter: Anwendungsbeispiel

- Ein Blog wird von einem Blogger erstellt und aktualisiert.
- Wenn ein Besucher der Webseite den Blog liest und interessant findet, kann er den Blog abonnieren und wird ab sofort über neue Einträge informiert.



The screenshot shows the homepage of the Cilk Arts Multicore Programming Blog. The header includes the Cilk Arts logo, a search bar, and a navigation menu with links to Home, Why Cilk++, Cilk++ Technology, Products, Case Studies, Multicore Blog (active), Company, Resources, Services, and Support. Below the header, the main content area features the title 'Multicore Programming Blog', links to 'Current Articles' and 'RSS Feed', and a post titled 'First Impressions of the Fortress Language' by Pablo Halpern. The post includes social media links and a list of highlights for the Fortress language. On the right side, there are two sidebar widgets: 'Multicore Programming Course in Boston!' and 'Download Cilk++'.

**CILK ARTS**

Home Why Cilk++ Cilk++ Technology Products Case Studies **Multicore Blog** Company Resources Services Support

[Multicore Programming Blog](#)

[Current Articles](#) | [RSS Feed](#)

[First Impressions of the Fortress Language](#)

Posted by Pablo Halpern on Fri, May 08, 2009

[Email Article](#) | [diigo.it](#) | [reddit](#) | [delicious](#) | [StumbleUpon](#) | [Facebook](#) | [Twitter](#) | [LinkedIn](#)

Tags: [Fortress](#)

I was privileged recently to attend a one-day hands-on introduction to [Fortress](#) lead by Sukyoung Ryu and Jan-Willem Maessen of Sun Microsystems and hosted by MIT. Fortress is a new parallel programming language developed at Sun and designed to bring together many of the best ideas in computer language design from the last few decades. Some of the highlights are:

- Implicit parallelism using a Cilk-style work-stealing scheduler
- Transactional synchronization to minimize contention on shared objects
- An extensible syntax that uses mathematical symbols
- Static type-checking with polymorphism and type inference
- Definition of many language features through libraries rather than built-in syntax
- Support for generic, object-oriented, and functional programming paradigms

<http://www.cilk.com/multicore-blog/>

**Multicore Programming Course in Boston!**

June 8/9 - Short course in Boston: [Concepts in Multicore Programming](#)

**Download Cilk++**

Which Edition is right for you?

Professional  
Open Source  
Academic

[GET CILK++](#)

**Subscribe by Email**

Your email:

[Subscribe](#)

## Beobachter: Anwendbarkeit

- Wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt und man nicht weiß, wie viele und welche Objekte geändert werden müssen.
- Wenn ein Objekt andere Objekte benachrichtigen muss, ohne Annahmen über diese Objekte zu treffen.
- Wenn eine Abstraktion zwei Aspekte besitzt, wobei einer von dem anderen abhängt. Die Kapselung dieser Aspekte in separaten Objekten ermöglicht unabhängige Wiederverwendbarkeit.

## Beobachter: Konsequenzen (1)

- Subjekte und Beobachter können **unabhängig** voneinander wiederverwendet werden
- Beobachter können neu **hinzugefügt** oder **entfernt** werden, **ohne** das Subjekt oder andere Beobachter **zu ändern**
- Die abstrakte Kopplung zwischen Subjekt und Beobachter wird durch die **Benachrichtigung** erreicht
- Subjekt und Beobachter gehören verschiedenen Schichten der Benutzt-Hierarchie an, ohne dabei Zyklen zu erzeugen
  - Subjekt benutzt nicht die Beobachter, aber umgekehrt



## Beobachter: Konsequenzen (2)

- Automatischer Rundruf von Änderungen
- Beobachter **entscheiden selbst**, ob sie die Benachrichtigung ignorieren oder nicht
- Der Aufwand der Aktualisierung kann versteckt sein
  - Eine einfache Benachrichtigung kann zu einer **Kaskade** von Aktualisierungen bei den Beobachtern führen
  - Die Botschaft enthält keinen Hinweis was geändert wurde. Ein erweitertes Protokoll kann verwendet werden, um den Beobachtern die konkrete Änderung mitzuteilen.

# Beobachter: Implementierung (1)

1. Wenn mehr als ein Subjekt beobachtet wird: Verwende Subjekt als Parameter von  
`aktualisiere(s :Subjekt)`
2. Auslösen der Aktualisierung
  - `setzeZustand()` ruft `benachrichtige()`, oder
  - Klienten rufen `benachrichtige()` explizit auf
3. Löschen eines Beobachters: Als erstes beim entsprechenden Subjekt abmelden

## Beobachter: Implementierung (2)

4. Subjekte müssen vor der Benachrichtigung **konsistent** sein
  - Wenn eine Subjektunterklasse geerbte Operationen aufruft, kann **versehentlich** die Oberklasse eine Benachrichtigung auslösen, **bevor** das Unterlassenobjekt komplett konsistent ist
  - Alternative: **Schablonenmethode** für die Aktualisierung verwenden.
5. Aktualisierung: Pull- und Push-Modell
  - Pull-Modell: Beobachter holt Daten vom Subjekt (kann ineffizient sein)
  - Push-Modell: Subjekt schickt Änderungsdaten an die Beobachter in `aktualisiere()` (kann Wiederverwendbarkeit reduzieren)

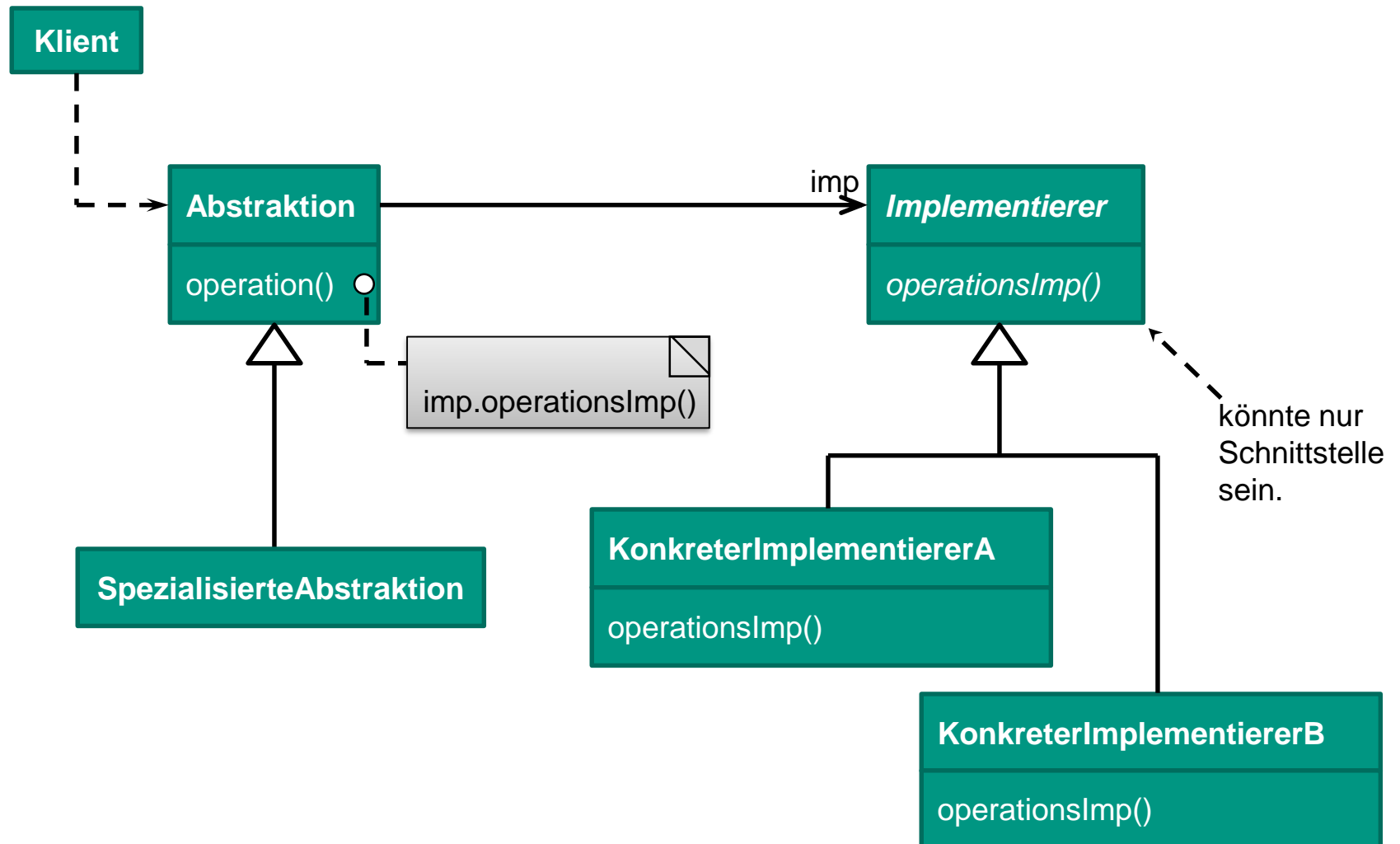
## Beobachter: Implementierung (3)

6. Änderungs-Manager zwischen Subjekt und Beobachtern
  - Bildet Subjekt auf seine Beobachter ab und bietet eine Schnittstelle zur Verwaltung dieser Abbildung
  - **Aktualisiert** bei Anforderung durch das Subjekt alle abhängigen Beobachter
  - Vermeidet **mehrfache** Aktualisierungen
  - Kapselt **komplexe** Aktualisierungssemantik

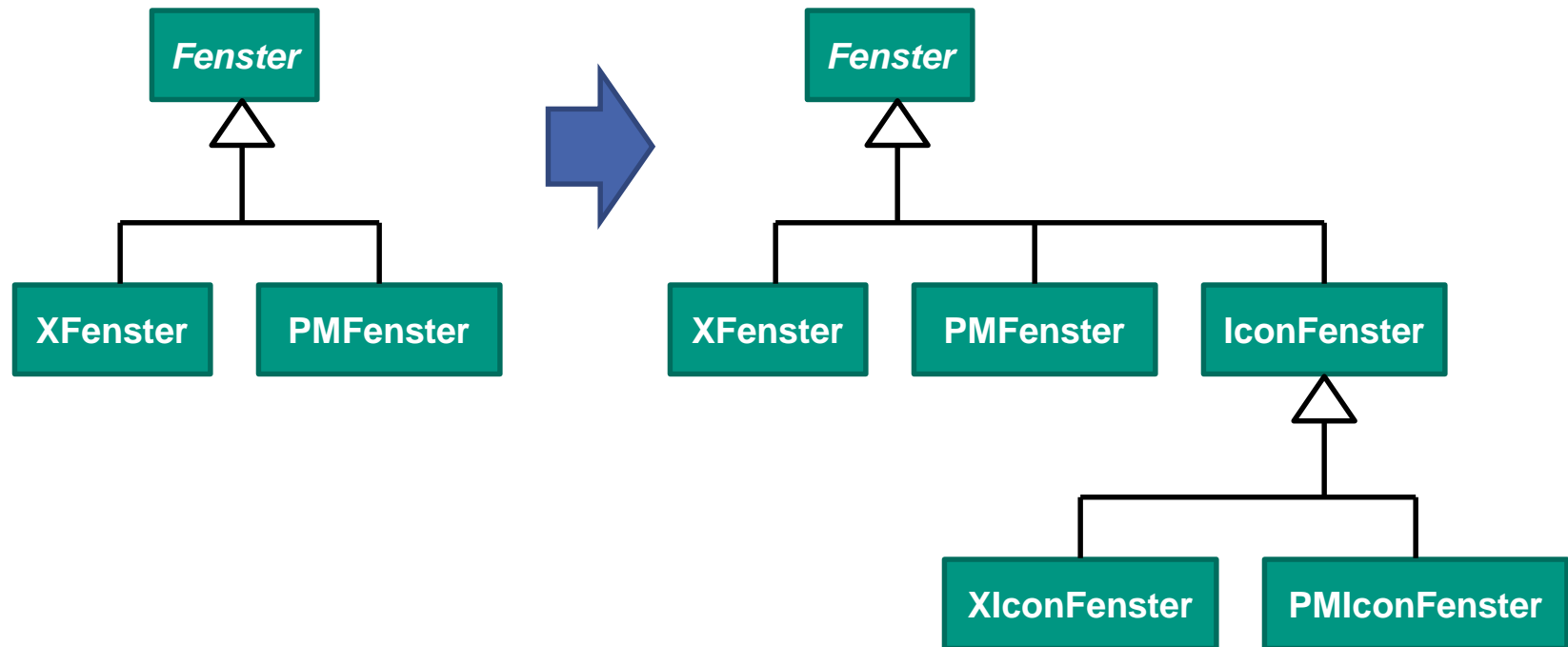
# Brücke (engl. *bridge*)

- Zweck
  - Entkopple eine **Abstraktion** von ihrer **Implementierung**, so dass beide unabhängig voneinander variiert werden können.
- Synonyme: Handle, Body

# Brücke

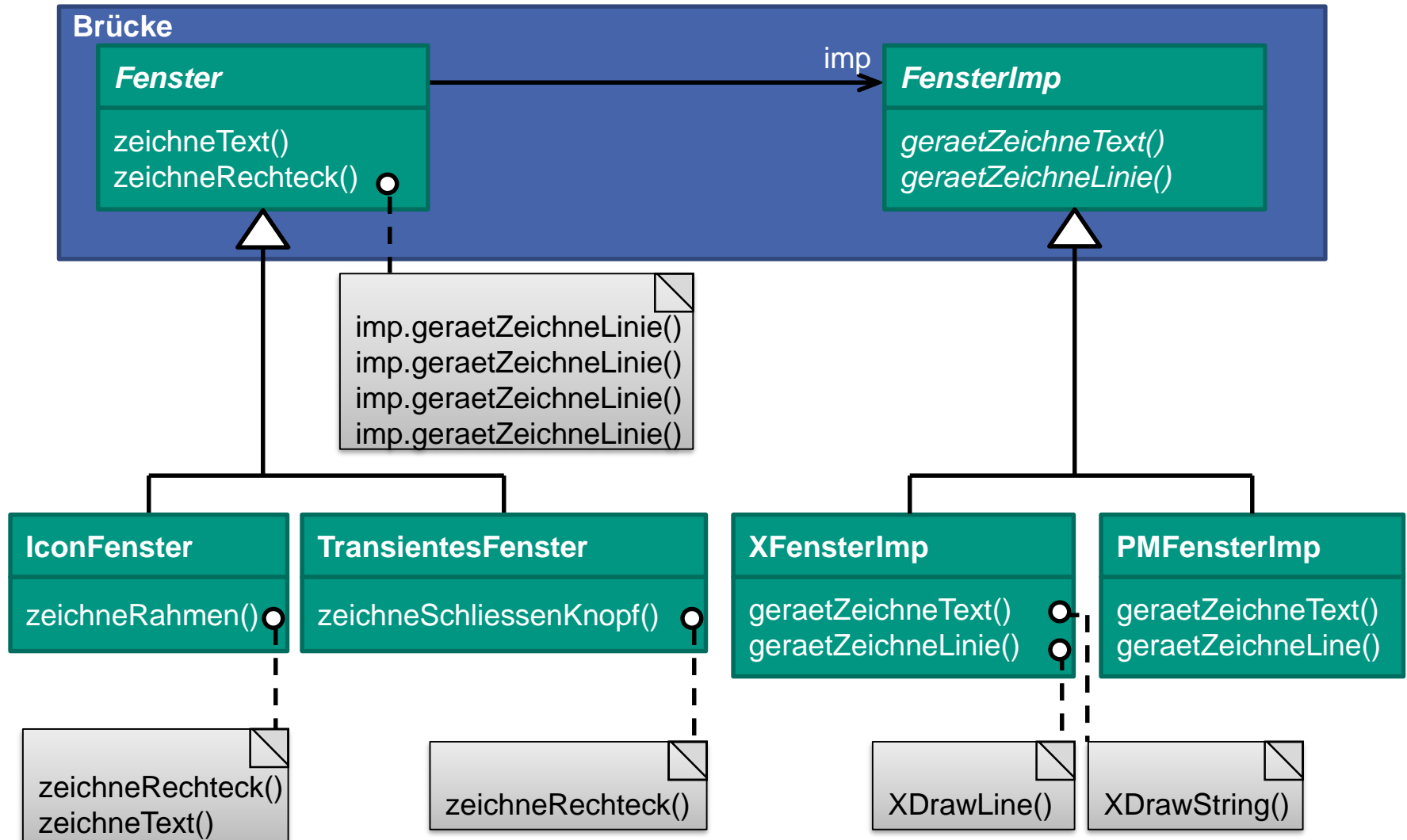


# Brücke: Gegenbeispiel



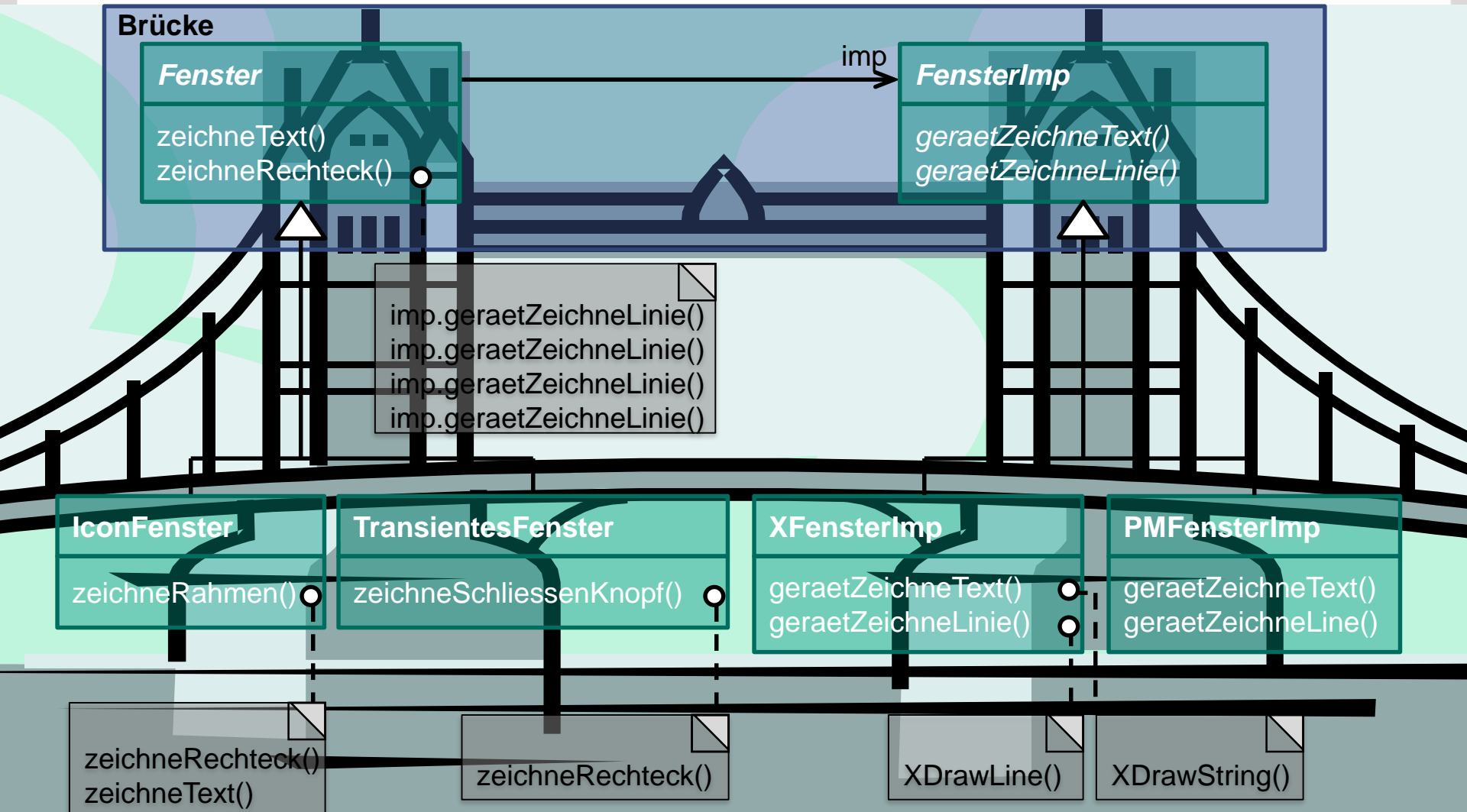
Was ist hier ungeschickt und unschön?

# Brücke: Beispiel (1)



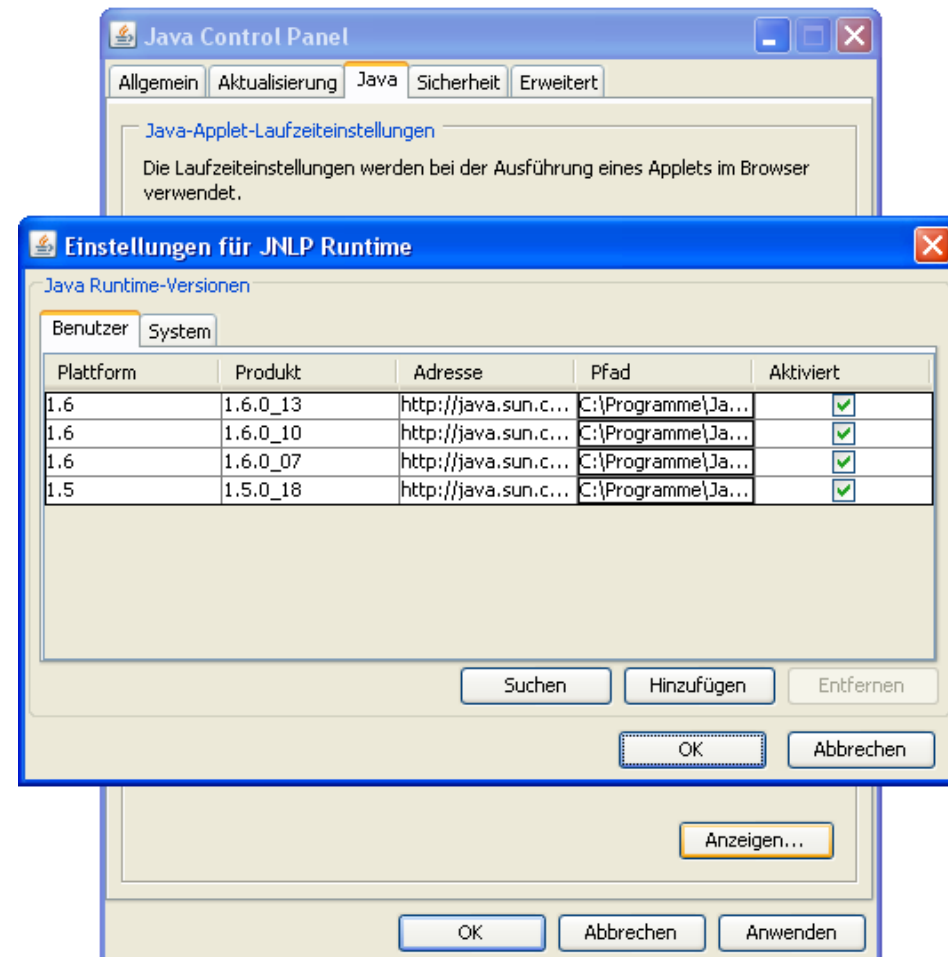


# Brücke: Beispiel (1)



# Brücke: Anwendungsbeispiel

- Installation mehrerer Java-Versionen gleichzeitig.
- Zu verwendende Java-Version kann vor Ausführung einer Anwendung gewählt werden.



## Brücke: Anwendbarkeit (1)

- Wenn eine **dauerhafte Verbindung** zwischen Abstraktion und Implementierung vermieden werden soll.
- Wenn sowohl Abstraktion als auch Implementierungen durch **Unterklassenbildung erweiterbar** sein soll.
- Wenn Änderungen in der Implementierung einer Abstraktion **keine Auswirkung** auf Klienten haben sollen.

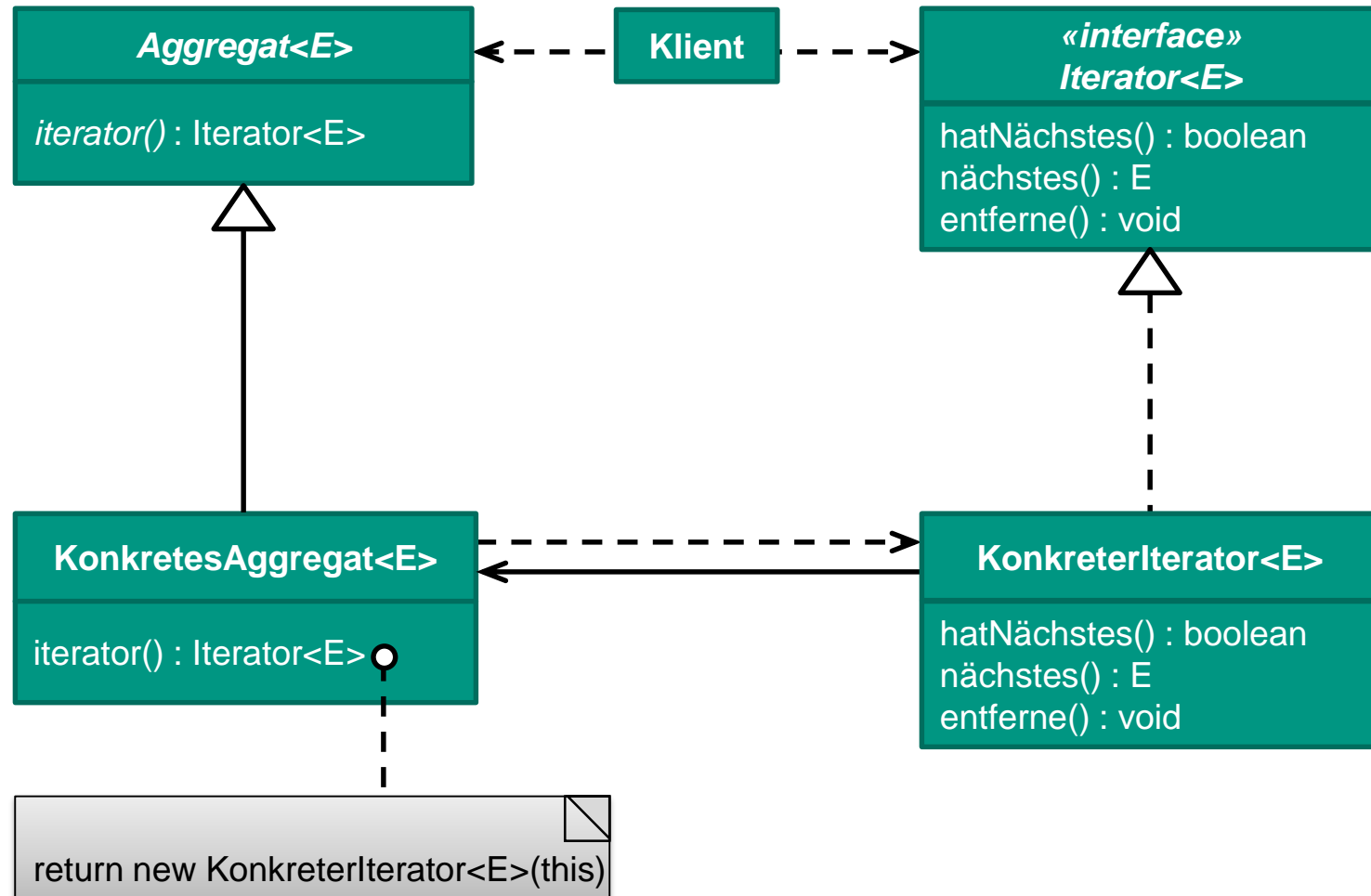
## Brücke: Anwendbarkeit (2)

- Wenn die Implementierung einer Abstraktion **vollständig** vom Klienten **versteckt** werden soll.
- Wenn eine starke **Vergrößerung** der Anzahl der Klassen **vermieden** werden soll (siehe Beispiel).
- Wenn eine Implementierung von mehreren Objekten aus **gemeinsam benutzt** werden soll.

# Iterator (engl. *iterator*)

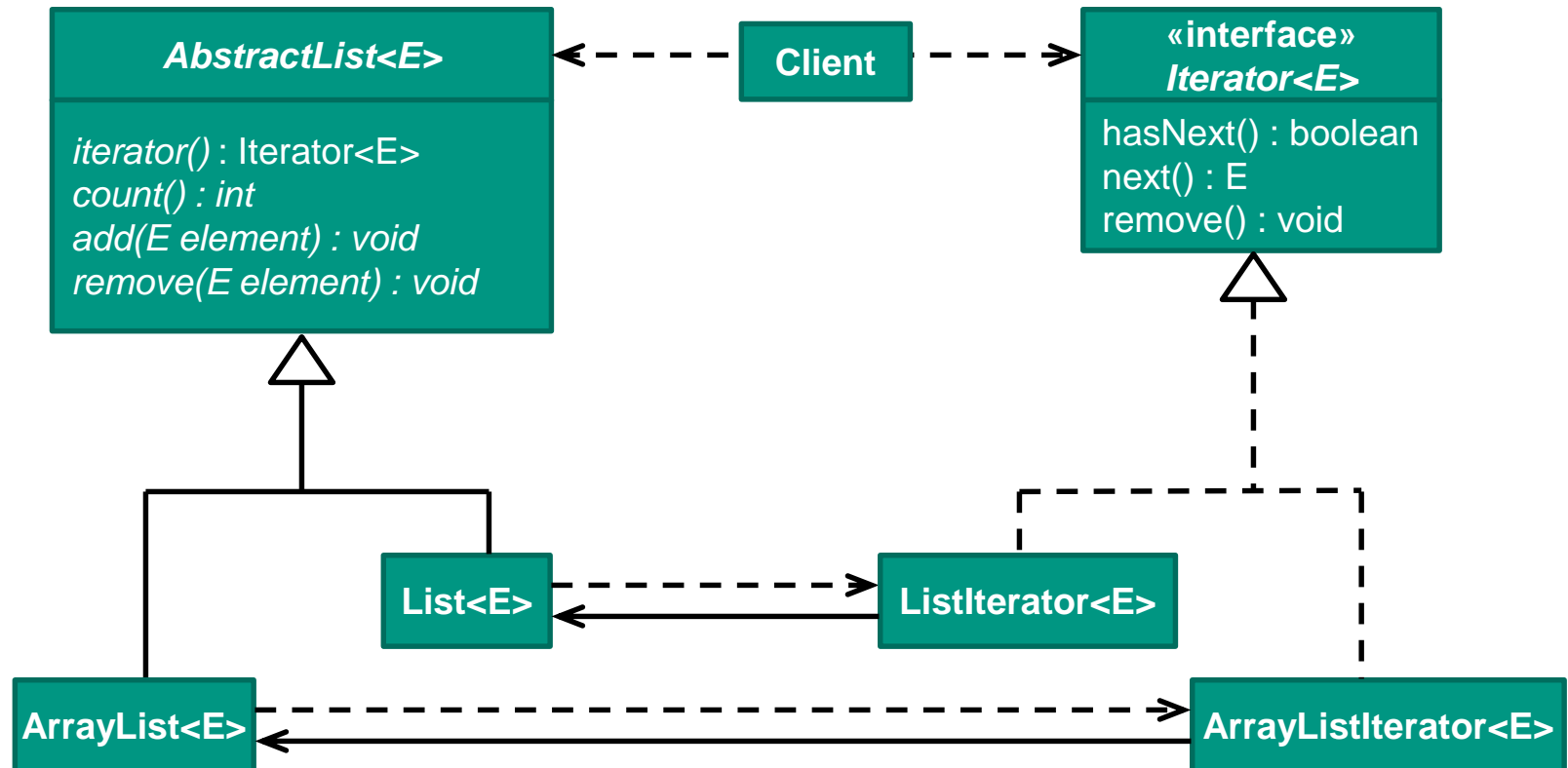
- Zweck
  - Ermögliche den **sequentiellen Zugriff** auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen.
  
- Synonyme: Enumerator, robuster Iterator

# Iterator: Struktur



# Iterator: Beispiel

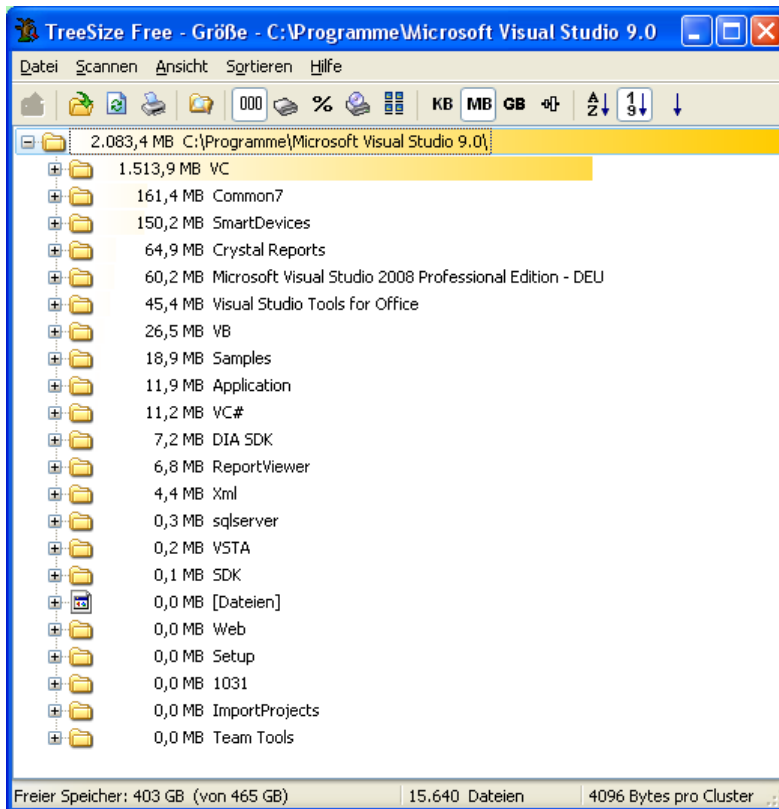
- Trennen von Listenimplementierungen und Listentraversierungs-Implementierungen



# Iterator: Anwendungsbeispiel

Speicherplatz von Verzeichnissen bestimmen

- Verwende einen Iterator, der alle Verzeichnisse sowie die darin enthaltenen Dateien kennt.
- Hole der Reihe nach alle Elemente (Dateien bzw. Verzeichnisse) mit Hilfe des Iterators, bestimme deren Größe und addiere die Werte auf.

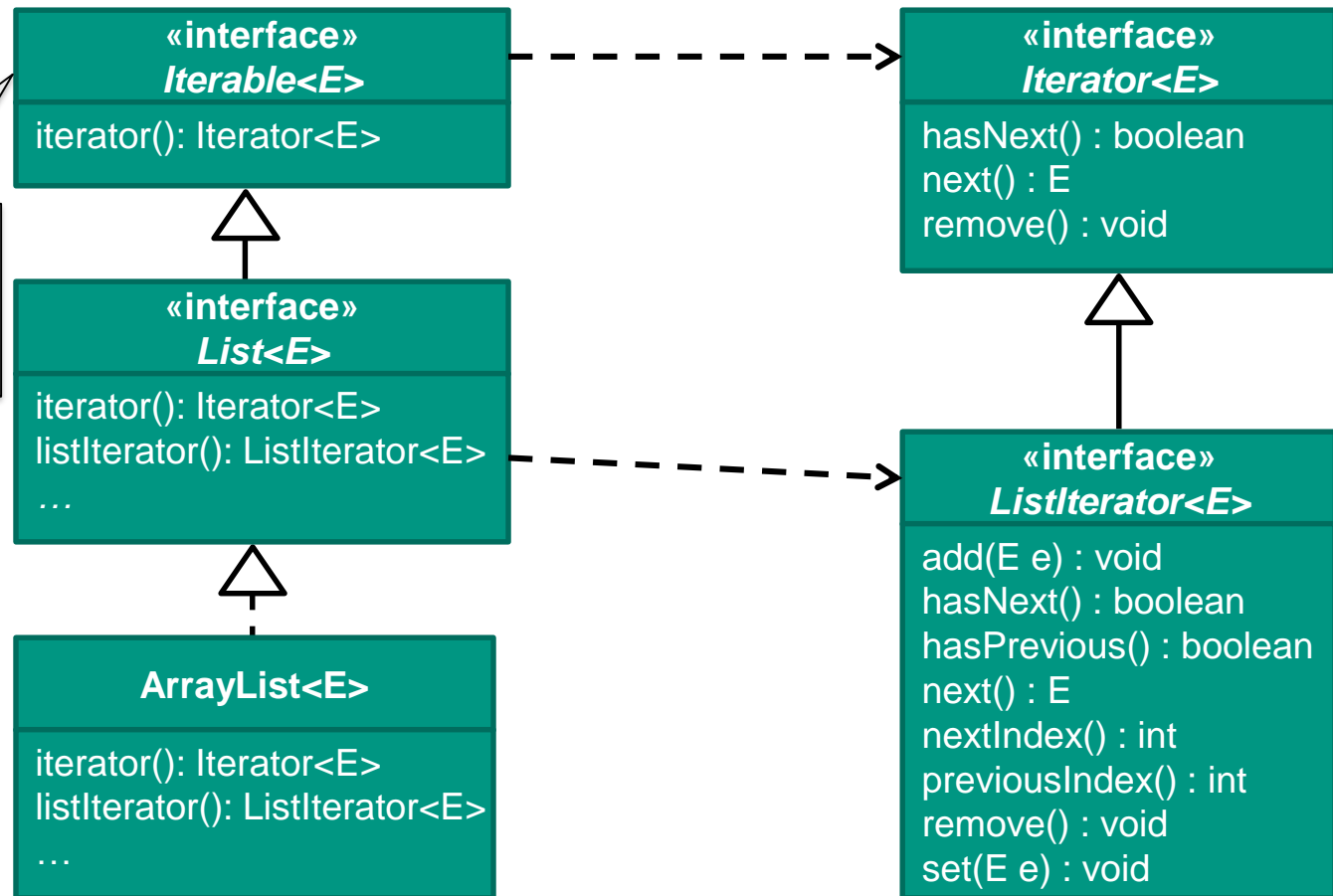




# Iterator:

## Implementierung in Java (1)

Ermöglicht  
Verwendung  
von „**foreach**“



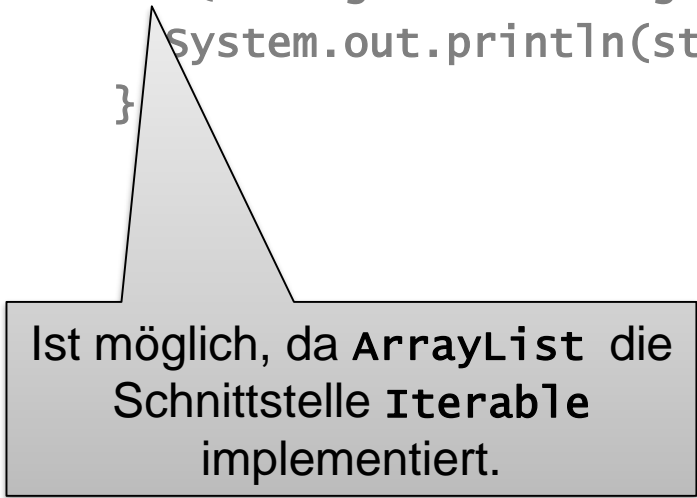
# Iterator: Implementierung in Java (2)

## Variante 1:

```
ArrayList<String>  
stringArrayList =  
    new ArrayList<String>();  
  
Iterator<String> iter =  
    stringArrayList.iterator();  
  
while(iter.hasNext()) {  
    System.out.println(  
        iter.next()  
    );  
}
```

## Variante 2:

```
ArrayList<String>  
stringArrayList =  
    new ArrayList<String>();  
  
for(String str: stringArrayList) {  
    System.out.println(str);  
}
```



Ist möglich, da **ArrayList** die Schnittstelle **Iterable** implementiert.

# Iterator: Enumerator vs. Iterator in Java

- Die Schnittstelle **Iterator** gibt es in Java erst seit Version 1.2. **Enumerator** gab es von Anfang an.
- Ein **Iterator** erlaubt das Entfernen von Elementen aus der Datenstruktur mit der **remove()**-Methode. Bei **Enumerator** ist das Entfernen von Elementen nicht vorgesehen.
- Wenn eine **remove()**-Methode nicht möglich ist, dann soll **remove()** so implementiert werden, dass es lediglich die Ausnahme **java.lang.UnsupportedOperationException** auslöst.

**Nachlesen:** Head First Design Patterns, Kapitel 9

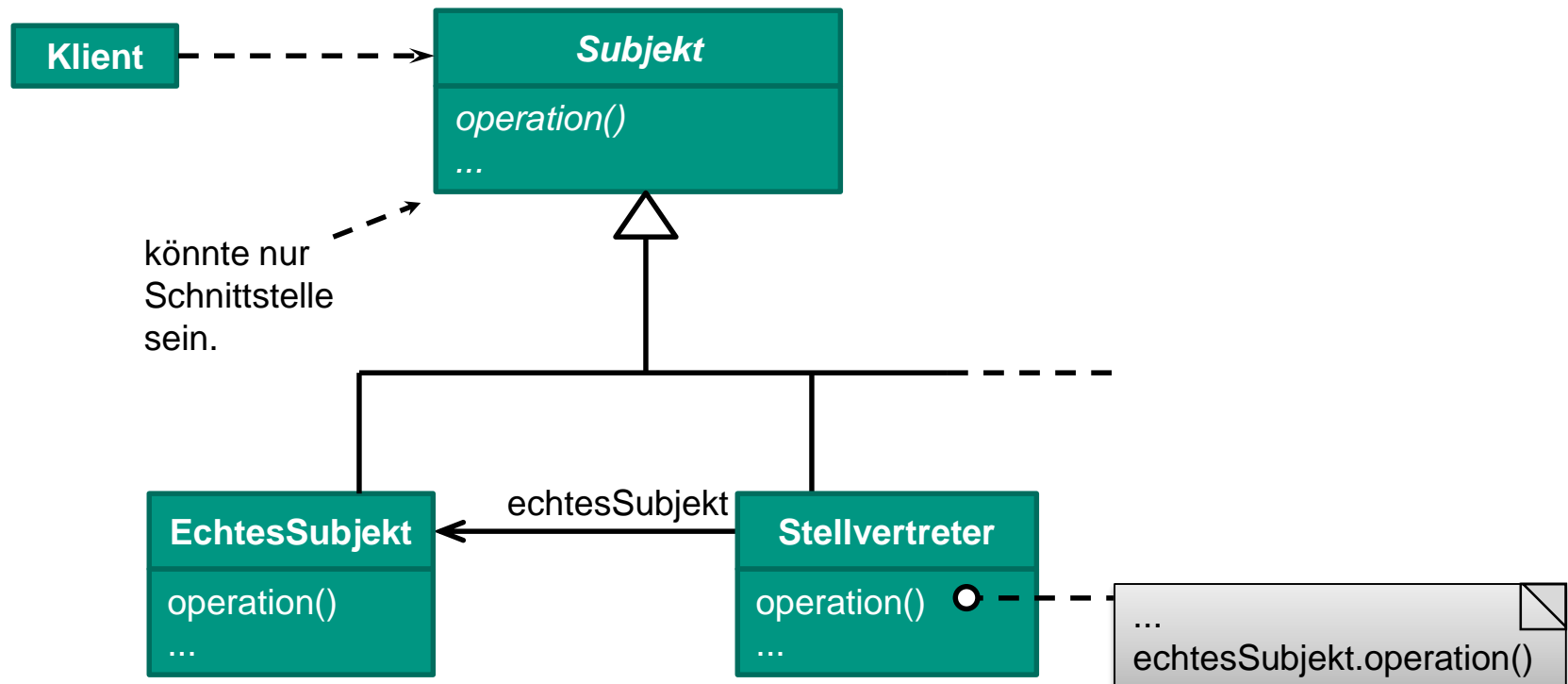
# Iterator: Anwendbarkeit

- Um den Zugriff auf den Inhalt eines zusammengesetzten Objekts zu ermöglichen, ohne dabei seine **interne Struktur** offenzulegen.
- Um eine **einheitliche Schnittstelle** zur Traversierung unterschiedlicher zusammengesetzter Strukturen anzubieten (das heißt, um polymorphe Iteration zu ermöglichen).
- **Robust** ist der Iterator, weil er gleichzeitig mehrere Traversierungen ermöglicht. Jeder Iterator enthält eine eigene „Laufvariable“.

# Stellvertreter (engl. *proxy*)

- Zweck
  - Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten **Stellvertreterobjekts**.
- Synonyme: Surrogat (engl. *surrogate*)

# Stellvertreter: Struktur



# Stellvertreter: Anwendbarkeit (1)

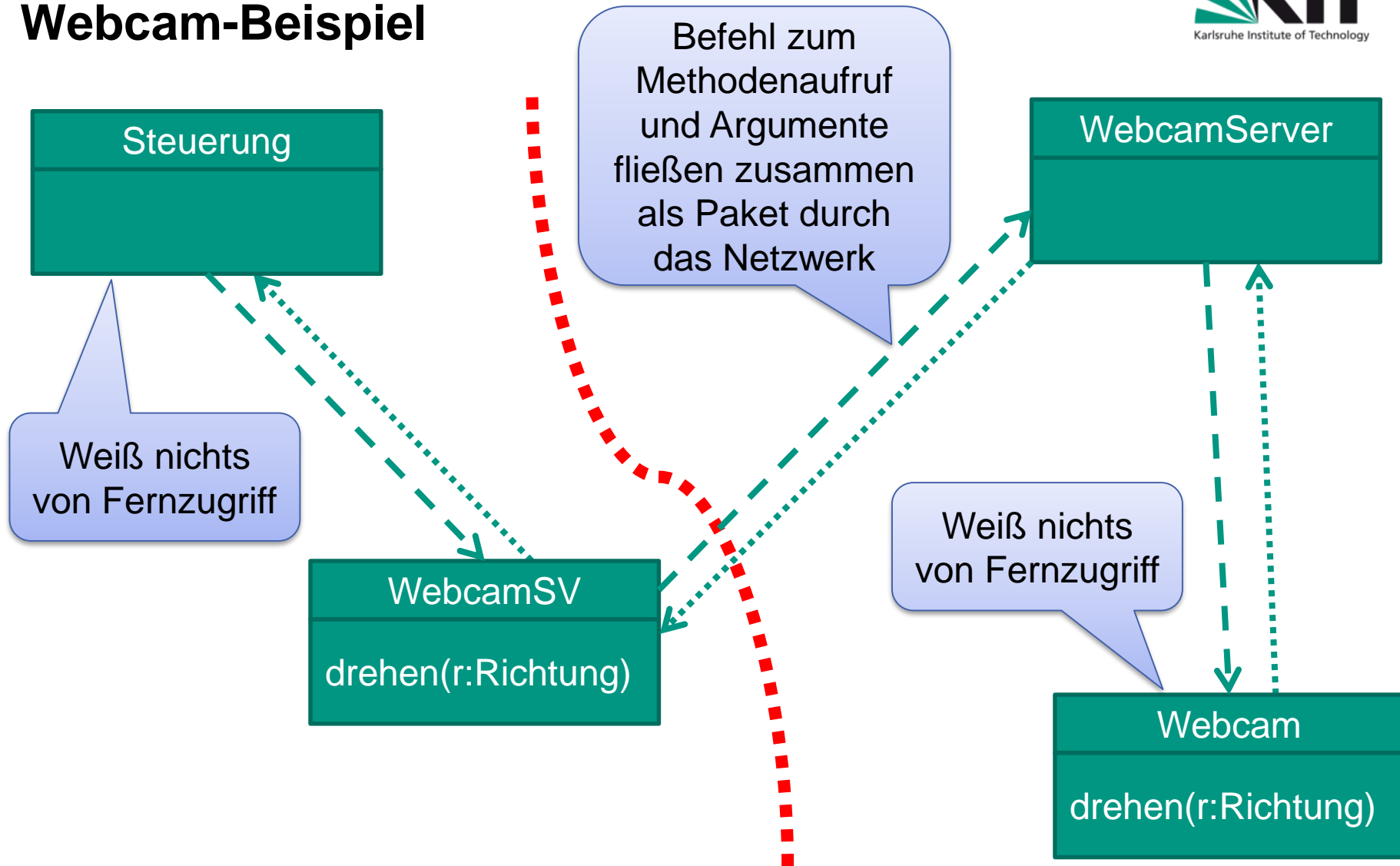
- Das Stellvertretermuster ist anwendbar, sobald es den Bedarf nach einer anpassungsfähigeren und intelligenteren Referenz auf ein Objekt als einen einfachen Zeiger gibt
- Es folgen einige verbreitete Situationen, in denen das Stellvertretermuster anwendbar ist:
  - Ein **protokollierender** Stellvertreter zählt Referenzen auf das eigentliche Objekt, so dass es automatisch freigegeben werden kann, wenn keine Referenzen mehr auf das Objekt existieren. Er kann auch andere Zugriffsinformationen protokollieren und leitet Zugriffe weiter.

## Stellvertreter: Anwendbarkeit (2)

- Ein **puffernder** Stellvertreter (engl. *caching proxy*) lädt ein persistentes Objekt erst dann in den Speicher, wenn es das erste Mal angesprochen wird. Er kann auch einen Puffer mit mehreren Objekten verwalten, die nach Bedarf zwischen Hintergrund- und Hauptspeicher bewegt werden.
- Ein **Fernzugriffsvertreter** (engl. *remote proxy*) stellt einen lokalen Stellvertreter für ein Objekt in einem anderen Adressraum dar.



# Fernzugriffsstellvertreter (*Remote Proxy*): Webcam-Beispiel



## Stellvertreter: Anwendbarkeit (3)

- Ein **Platzhalter** (engl. *virtual proxy*) erzeugt teure Objekte auf Verlangen (verzögertes Laden, verzögertes Erzeugen).
- Eine **Schutzwand oder Brandmauer** (engl. *firewall*) kontrolliert den Zugriff auf das Originalobjekt. Schutzwände sind nützlich, wenn Objekte über verschiedene Zugriffsrechte verfügen sollen.
- Ein **Synchronisierungsvertreter** (engl. *synchronization proxy*) koordiniert den Zugriff mehrerer Fäden auf ein Objekt.
- Ein **Dekorierer** fügt zusätzliche Zuständigkeiten zu einem bestehenden Objekt hinzu (möglicherweise kaskadiert).

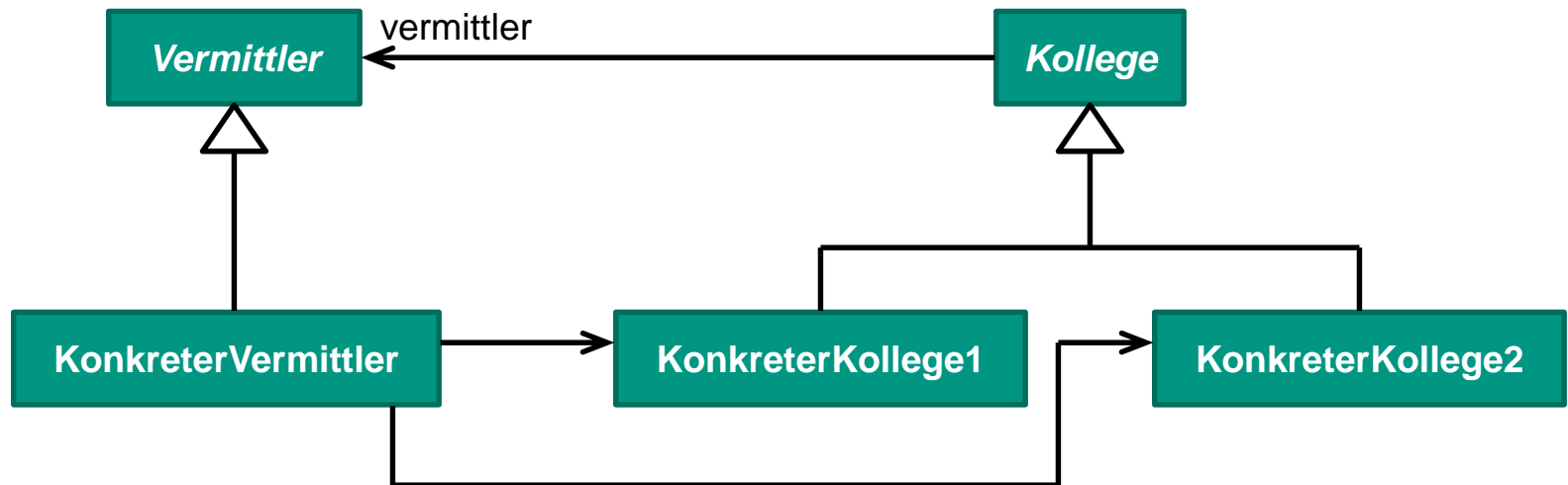
**Nachlesen:** Head First Design Patterns, Seite 460 ff.

# Vermittler (engl. *mediator*)

## ■ Zweck

- Definiere ein Objekt, welches das **Zusammenspiel** einer Menge von Objekten in sich kapselt.
- Vermittler fördern **lose Kopplung**, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es, das Zusammenspiel der Objekte unabhängig zu variieren.

# Vermittler: Struktur

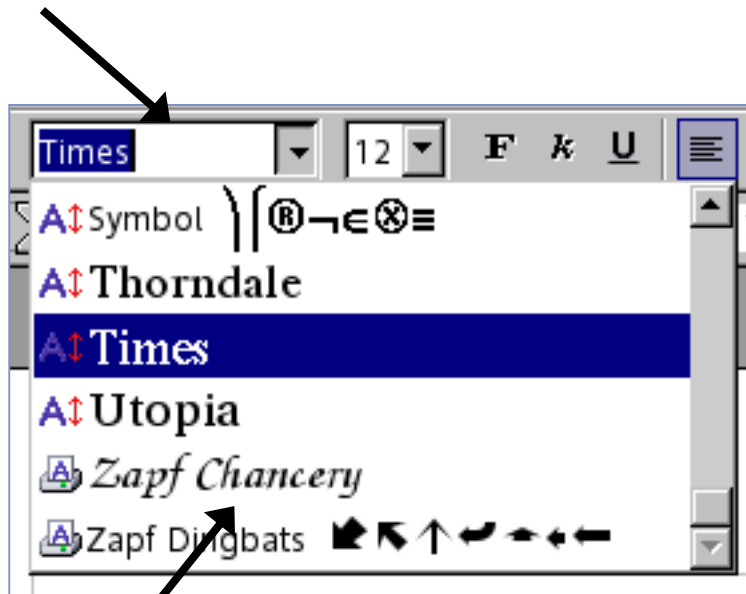


## Vermittler: Beispiel (1)

- Es gibt oft Abhängigkeiten zwischen den Elementen (Knöpfen, Menüs, Eingabefelder, etc.) einer Dialogbox. So muss z.B. ein Knopf deaktiviert sein, wenn ein bestimmtes Texteingabefeld leer ist.
  - Unterschiedliche Dialogboxen besitzen unterschiedliche Abhängigkeiten zwischen Elementen.
  - Individuelle Anpassung in Unterklassen ist mühsam und schlecht wieder verwendbar (zu viele Klassen).
- Kapseln des Gesamtverhaltens in einem Vermittlerobjekt.

## Vermittler: Beispiel (2)

Eingabefeld



Listbox

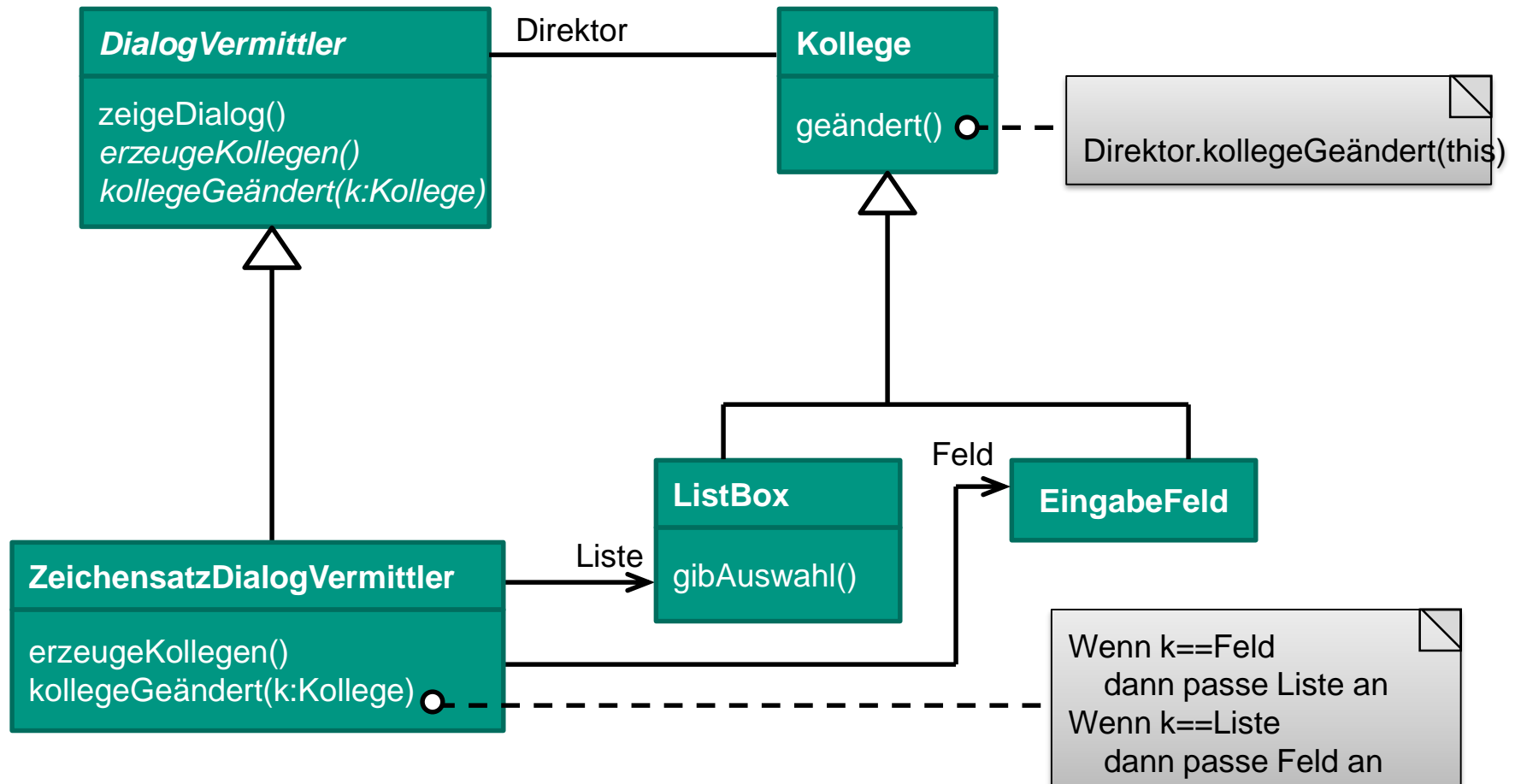
Beim Tippen rollt  
Auswahlliste auf.

Tippen eines Buchstabens  
im Eingabefeld  
zeigt ersten Eintrag mit  
gleichem Anfang.

In Listbox angewähltes  
Element  
erscheint im Eingabefeld.

# Vermittler: Beispiel (3)

## ■ Fenster mit Zeichensatz-Dialog



# Vermittler: Beispiel 2 (1)

## Wecker

```
onEvent() {
  prüfeKalender();
  prüfeRasensprenger();
  schalteKaffeemaschineAn();
}
```

## Kaffeemaschine

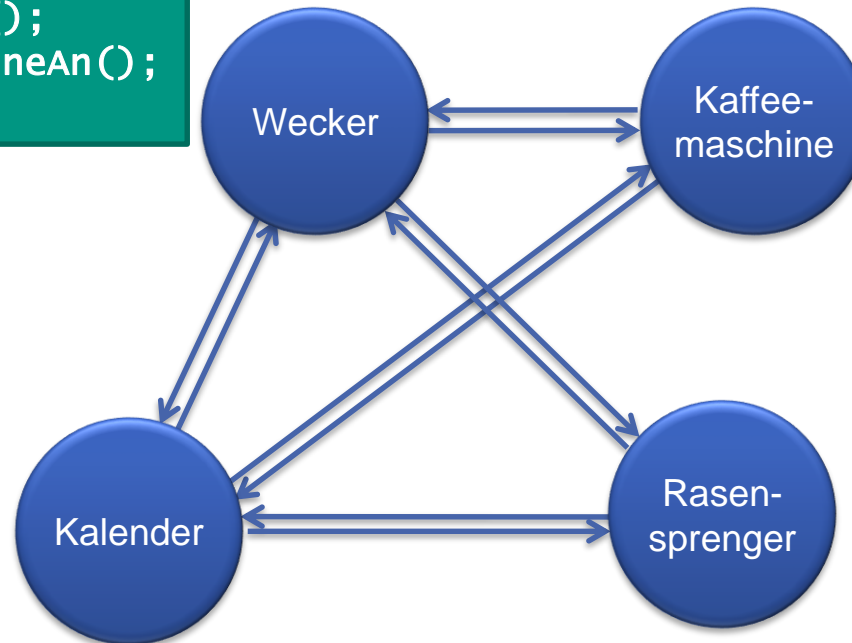
```
onEvent() {
  prüfeKalender();
  prüfewecker();
}
```

## Kalender

```
onEvent() {
  prüfewochentag();
  bewässereRasen();
  macheKaffee();
  löseWeckerAus();
}
```

## Rasensprenger

```
onEvent() {
  prüfeKalender();
  prüfeDusche();
  prüfeTemperatur();
  prüfewetter();
}
```



Jede Klasse muss  
jede kennen.

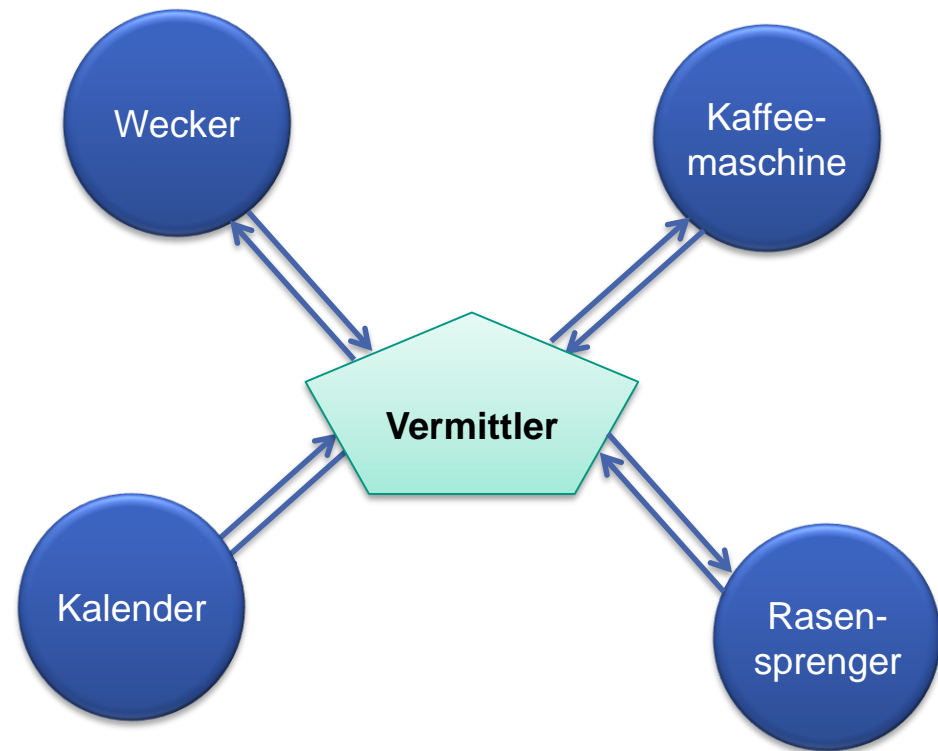
**Beispiel:** Head First Design Patterns, Seite 622 ff.



## Vermittler: Beispiel 2 (2)

### Vermittler

```
onEvent() {  
  if (weckzeit) {  
    prüfeKalender();  
    prüfeDusche();  
    prüfeTemperatur();  
  }  
  if (wochenende) {  
    prüfeWetter();  
  }  
  if (müllsammlung) {  
    weckerFrüherStellen();  
  }  
}
```



Der Vermittler koordiniert,  
übrige Klassen sind entkoppelt.

## Vermittler: Anwendbarkeit

- Wenn eine Menge von Objekten vorliegt, die in wohl-definierter, aber **komplexer** Weise zusammen arbeiten. Die sich ergebenden Abhängigkeiten sind unstrukturiert und schwer zu verstehen.
- Wenn die Wiederverwertung eines Objektes schwierig ist, weil es sich auf viele andere Objekte bezieht und mit ihnen zusammenarbeitet.
- Wenn ein auf mehrere Klassen **verteiltes Verhalten** maßgeschneidert werden soll, ohne viele Unterklassen bilden zu müssen.

**Beispiel:** Head First Design Patterns, Seite 622 ff.

# Varianten-Muster

IPD Tichy, Fakultät für Informatik



# Varianten-Muster

- Abstrakte Fabrik
- (Besucher)
- Schablonenmethode
- Fabrikmethode
- Erbauer
- Kompositum
- Strategie
- Dekorierer

Übung

Nicht prüfungsrelevant

# Abstrakte Fabrik (engl. *abstract factory*)

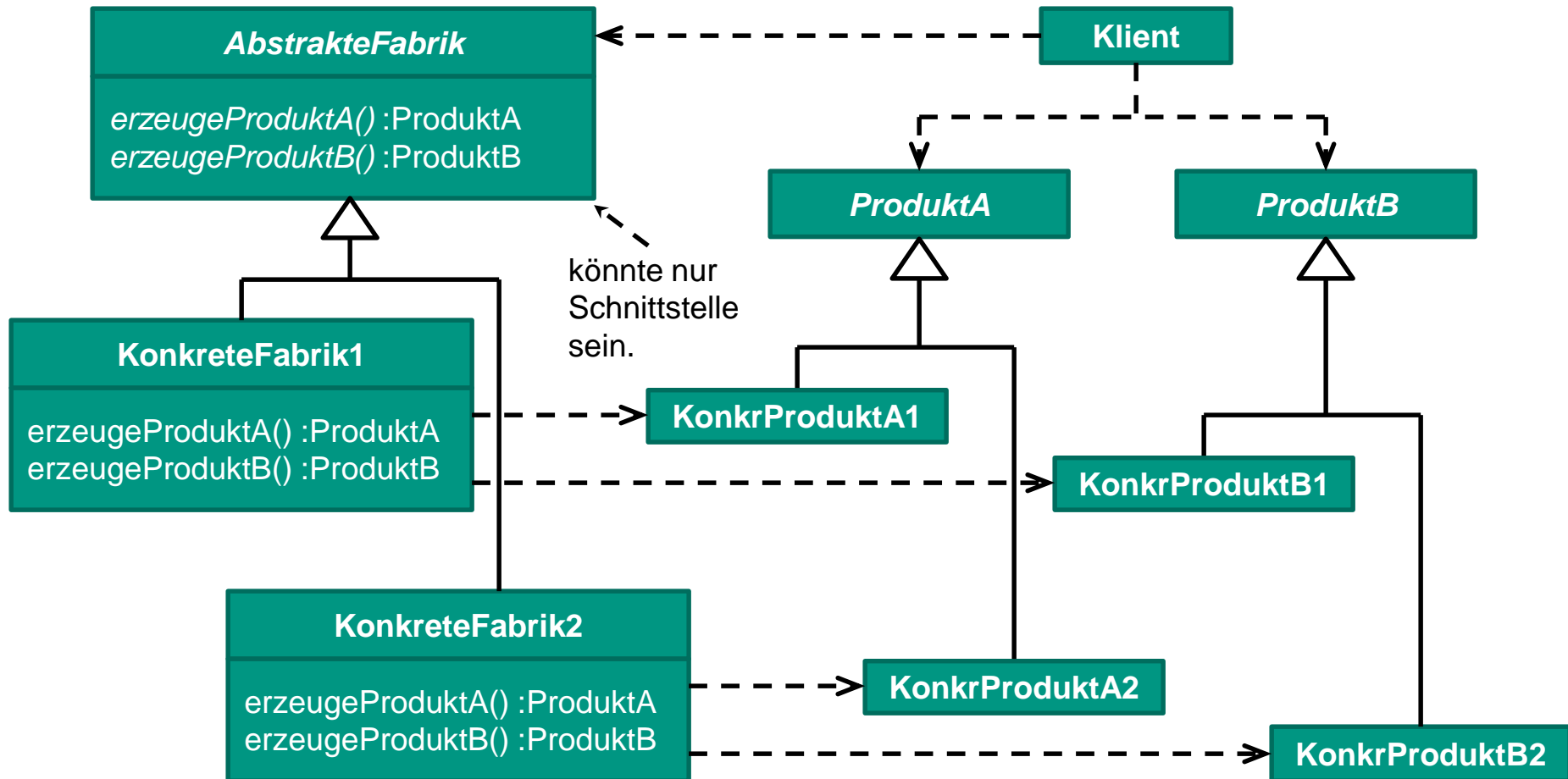
## ■ Zweck

- Bietet eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.

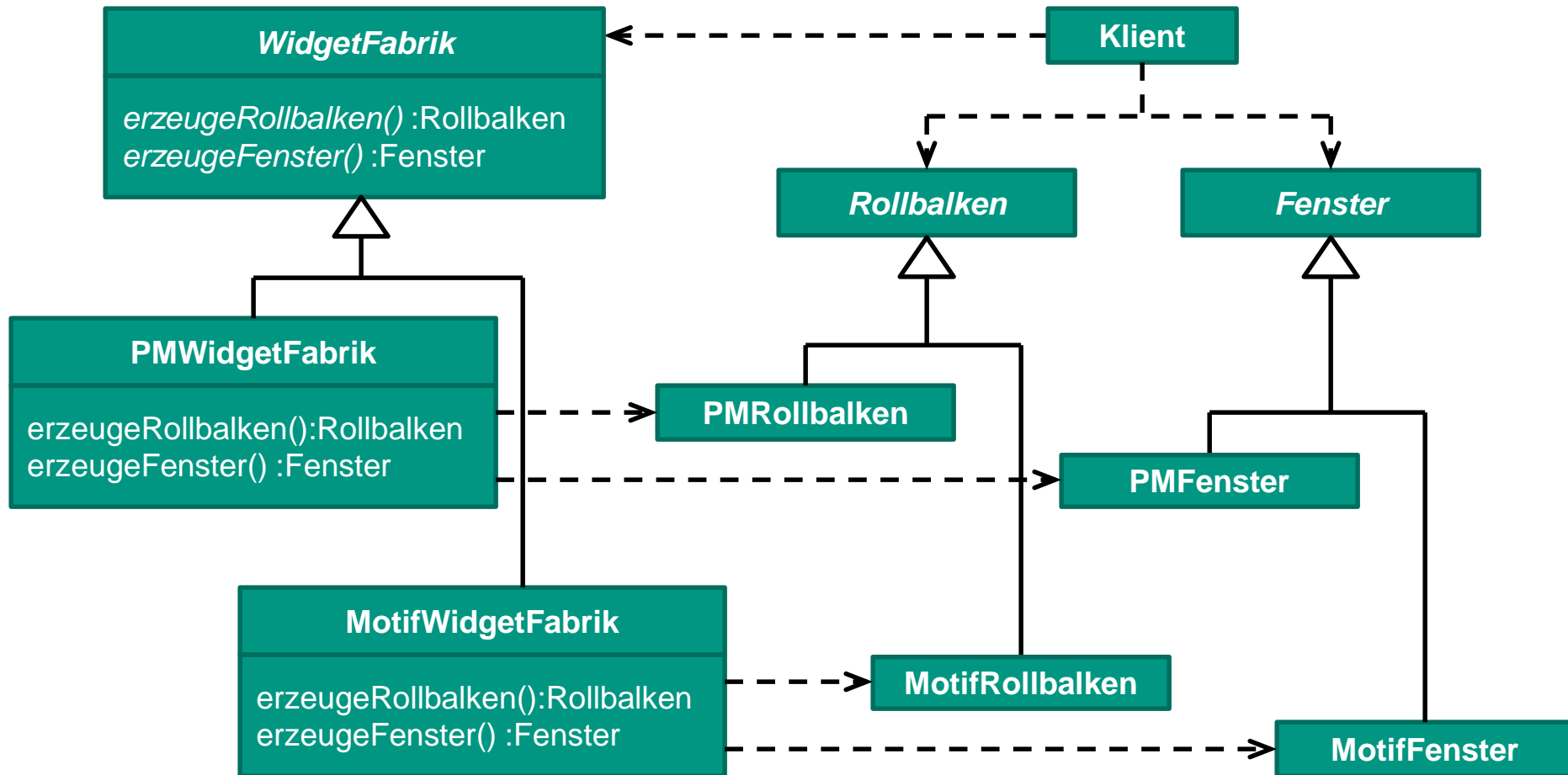
## ■ Synonyme: Bausatz (engl. *kit*)

**Nachlesen:** Head First Design Patterns, Kapitel 4

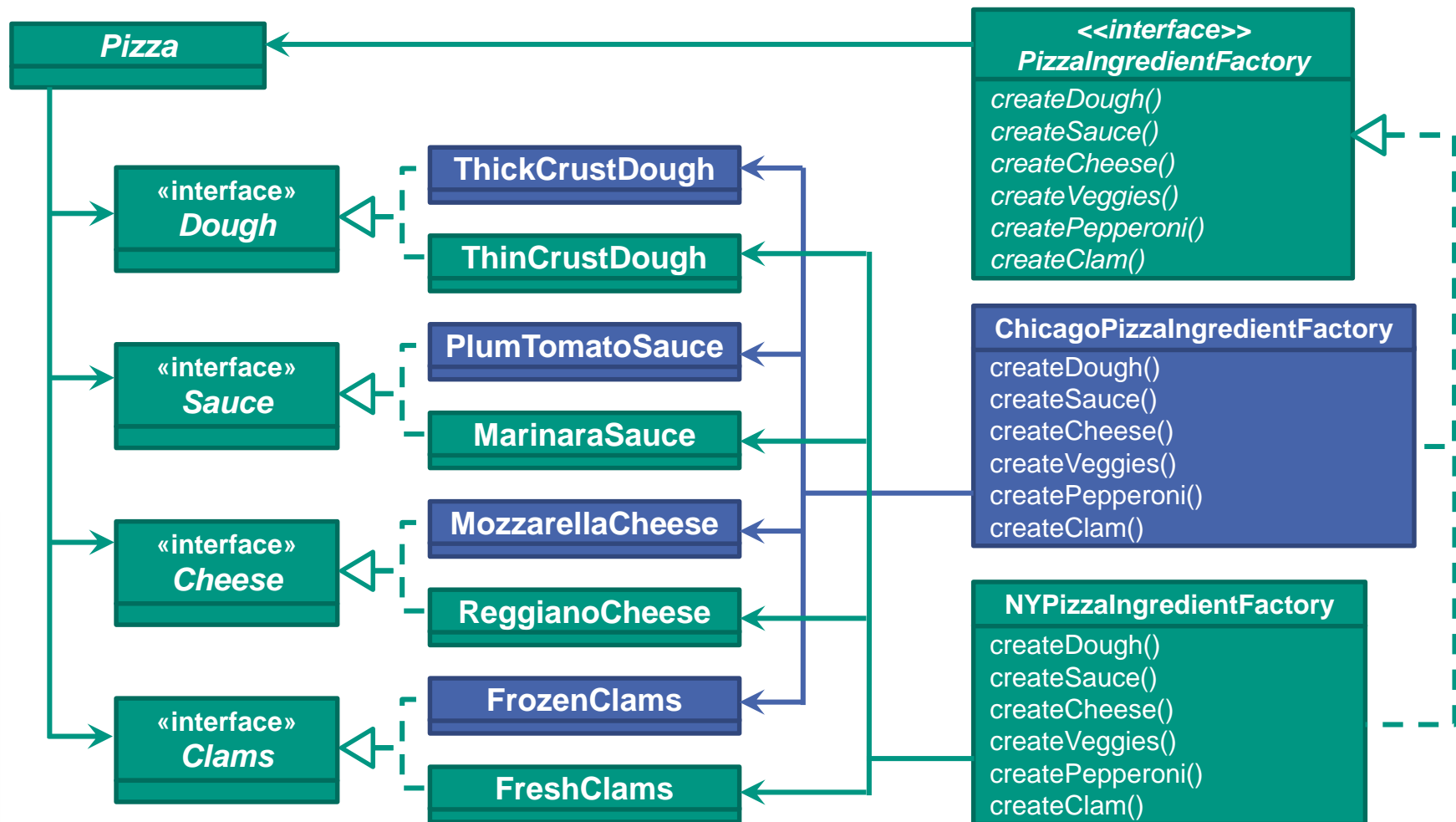
# Abstrakte Fabrik: Struktur



# Abstrakte Fabrik: Beispiel (1)



# Abstrakte Fabrik: Beispiel (2)



**Beispiel:** Head First Design Patterns, Seite 157 ff.



# Abstrakte Fabrik: Anwendbarkeit

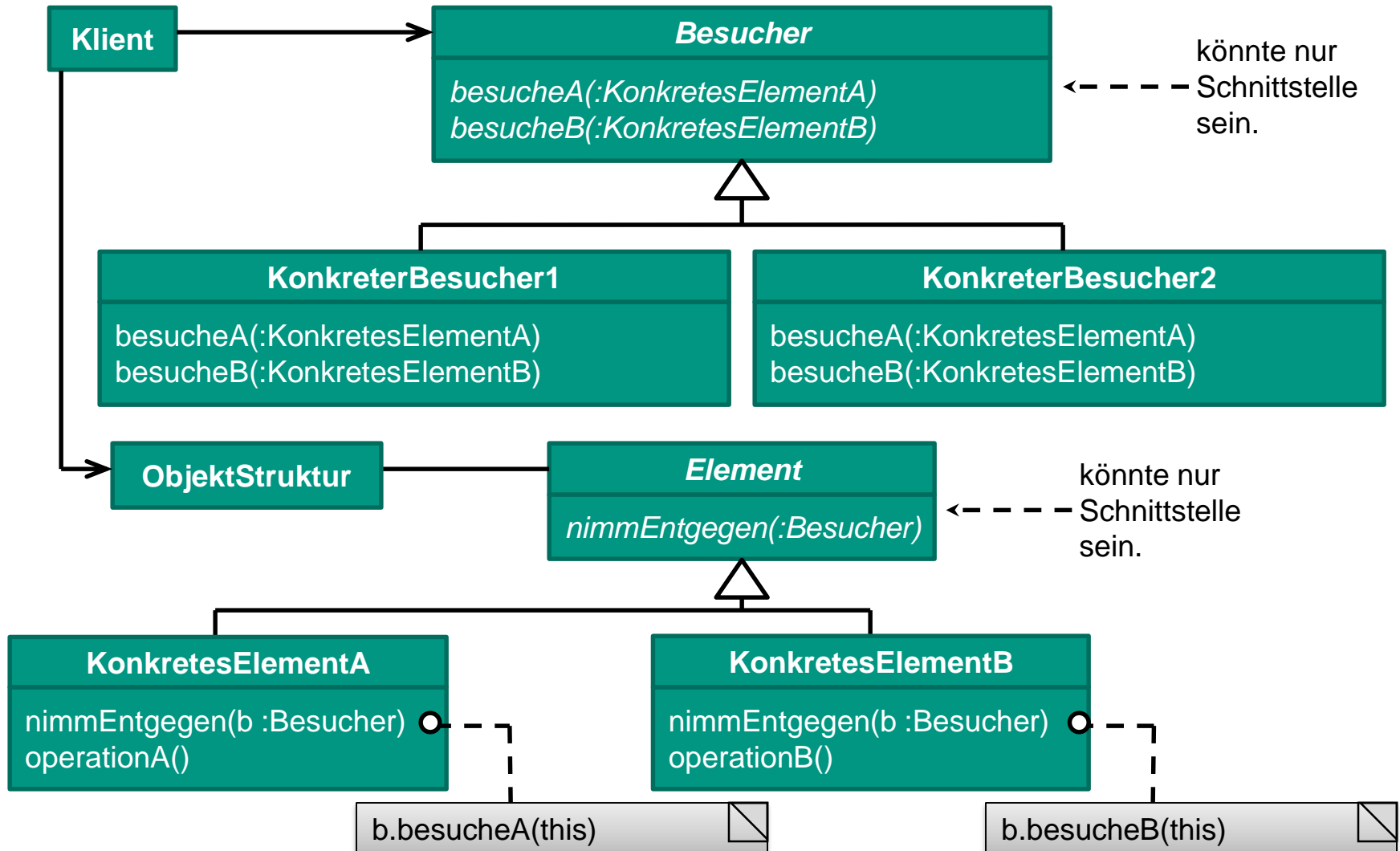
- Wenn ein System **unabhängig** davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden.
- Wenn ein System mit einer von mehreren **Produktfamilien** konfiguriert werden soll.
- Wenn eine Familie von **auf einander abgestimmten** Produktobjekten zusammen verwendet werden sollen, und dies erzwungen werden muss.
- Bei einer Klassenbibliothek, die nur die **Schnittstellen**, nicht aber die Implementierungen offen legt.

# Besucher (engl. *visitor*)

## ■ Zweck

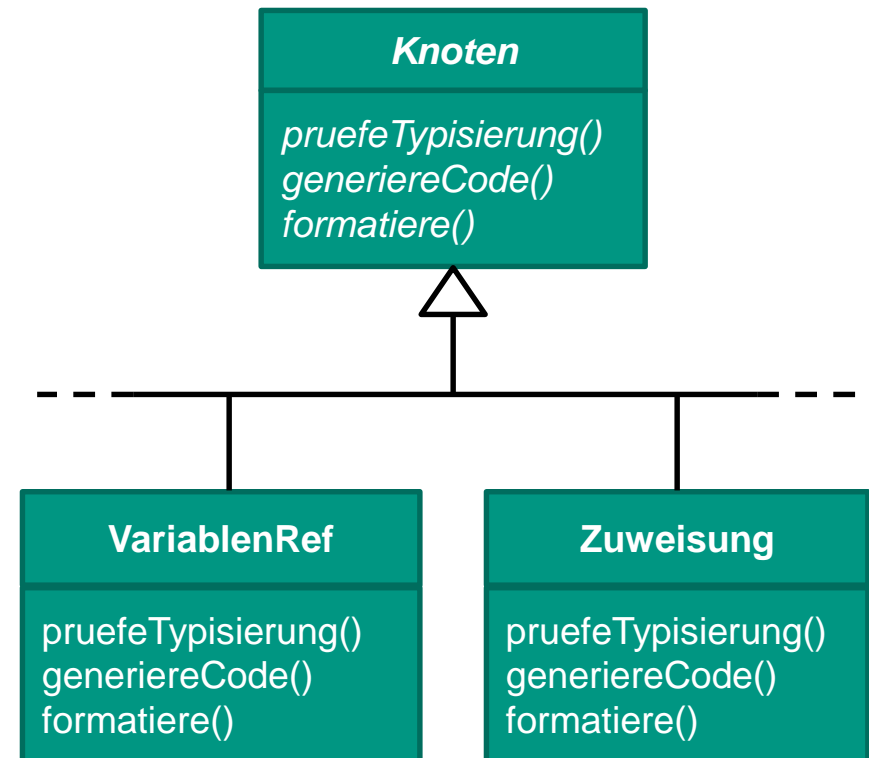
- **Kapsle** eine auf den Elementen einer Objektstruktur auszuführende **Operation** als ein Objekt.
- Das Besuchermuster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

# Besucher: Struktur

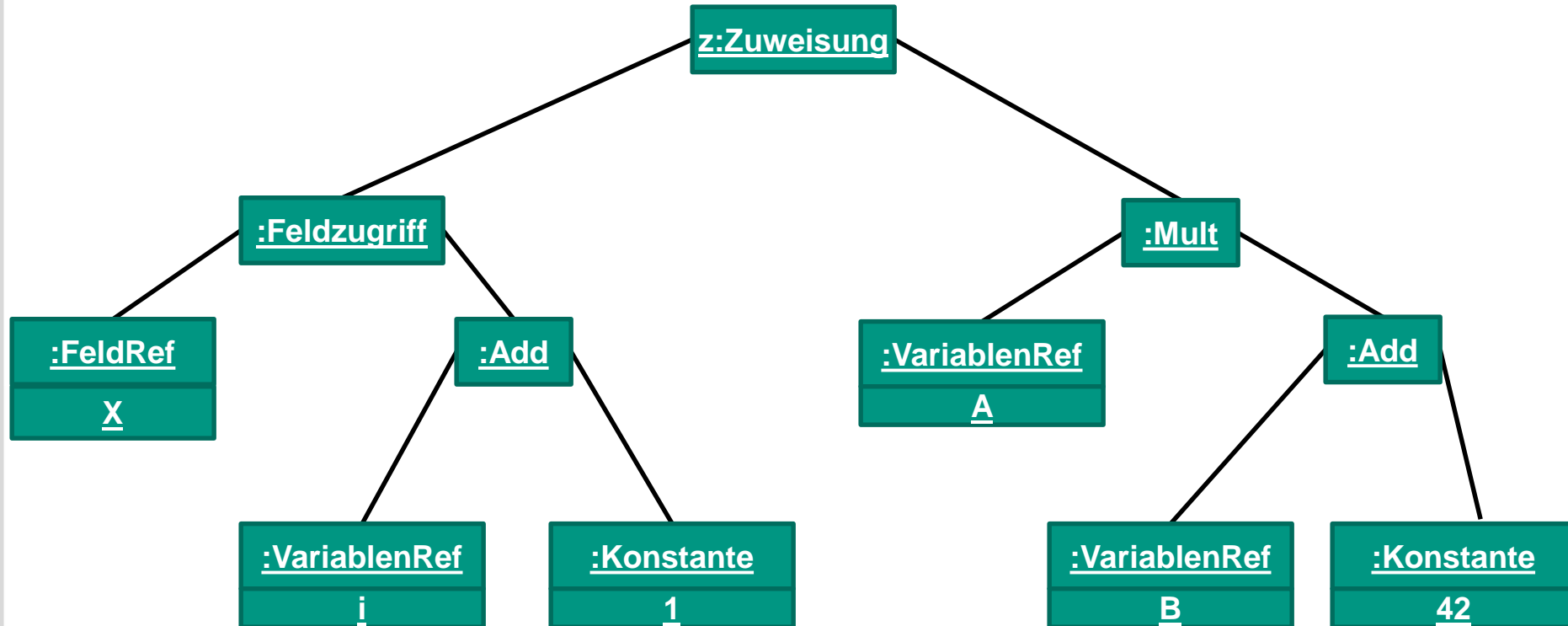


# Besucher: Beispiel (1) ohne Besucher

- Abstrakte Syntaxbäume in einem Übersetzer
- Die einzelnen Operationen sind über viele Klassen verstreut.
- Bei Einführung neuer Operationen müssen alle diese Klassen erweitert werden.

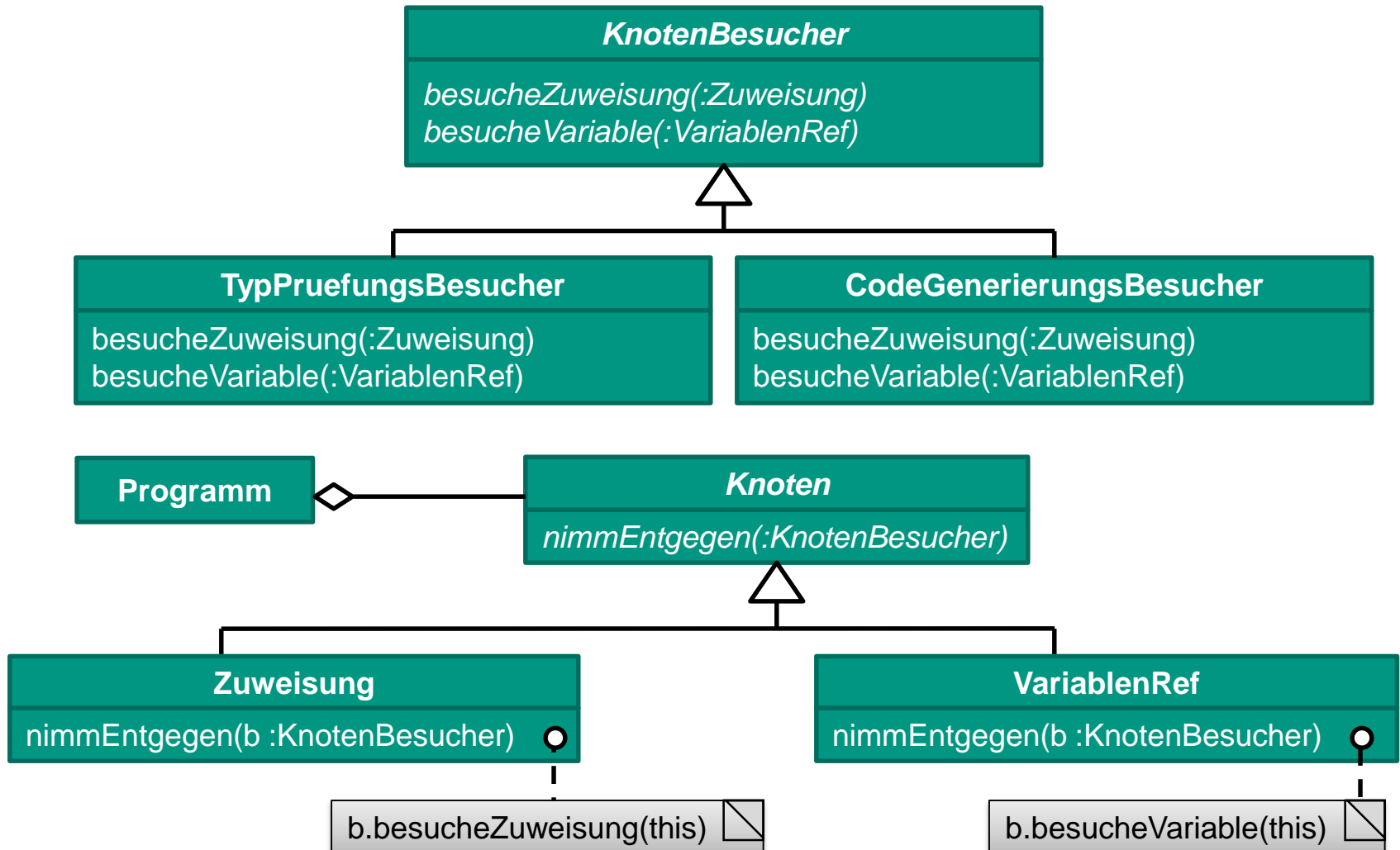


# Beispiel für einen abstrakten Syntaxbaum

$$X[i+1] = A * (B + 42)$$


Was macht `z.nimmEntgegen(new CodeGenerierungsBesucher())` ?

# Besucher: Beispiel (2) mit Besucher



## Besucher: Anwendbarkeit

- Wenn eine Objektstruktur viele Klassen von Objekten mit **unterschiedlichen Schnittstellen** enthält und Operationen auf diesen Objekten ausgeführt werden sollen, die von ihren konkreten Klassen abhängen.
- Wenn mehrere **unterschiedliche** und nicht miteinander verwandte **Operationen** auf den Objekten einer Objektstruktur ausgeführt werden müssen und diese Klassen nicht mit diesen Operation „verschmutzt“ werden sollen.
- Wenn sich die Klassen, die eine Objektstruktur definieren, praktisch nie ändern, aber häufig neue Operationen für die Struktur definiert werden.

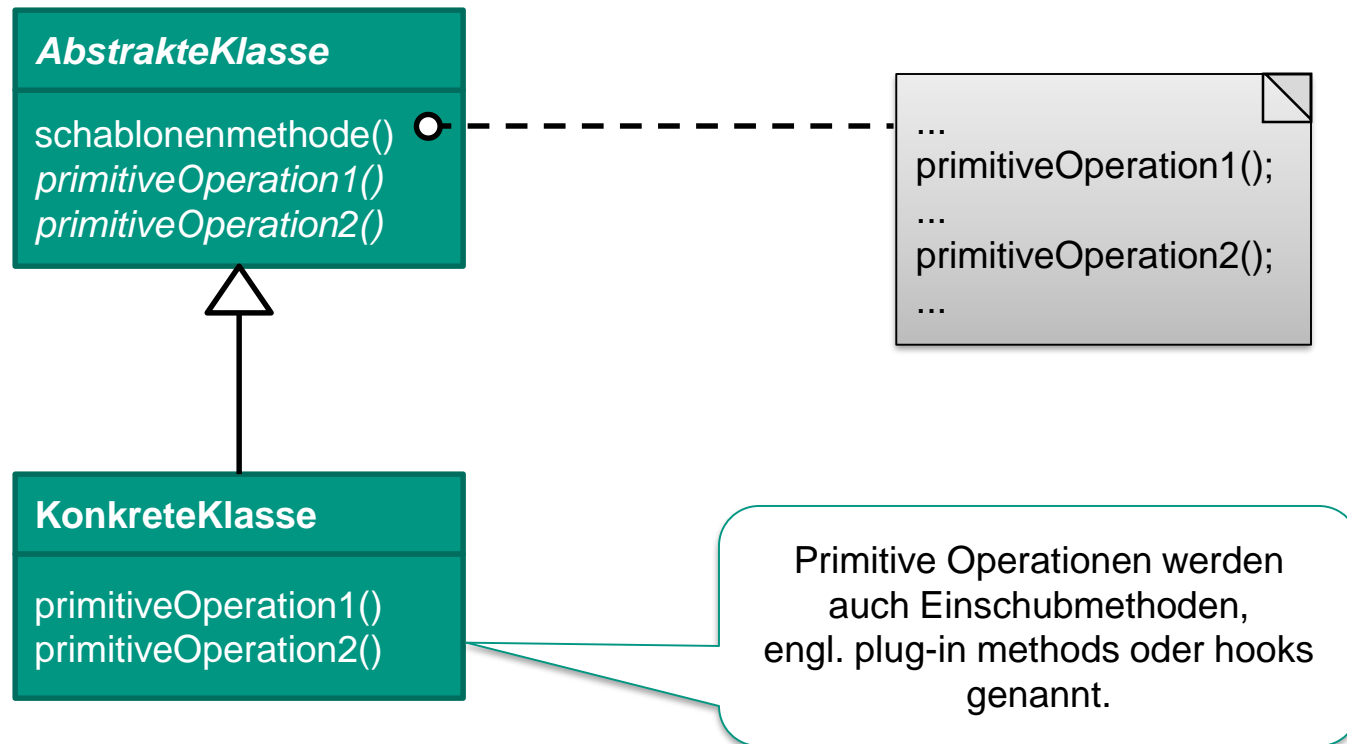
# Schablonenmethode (engl. *template method*)

## ■ Zweck

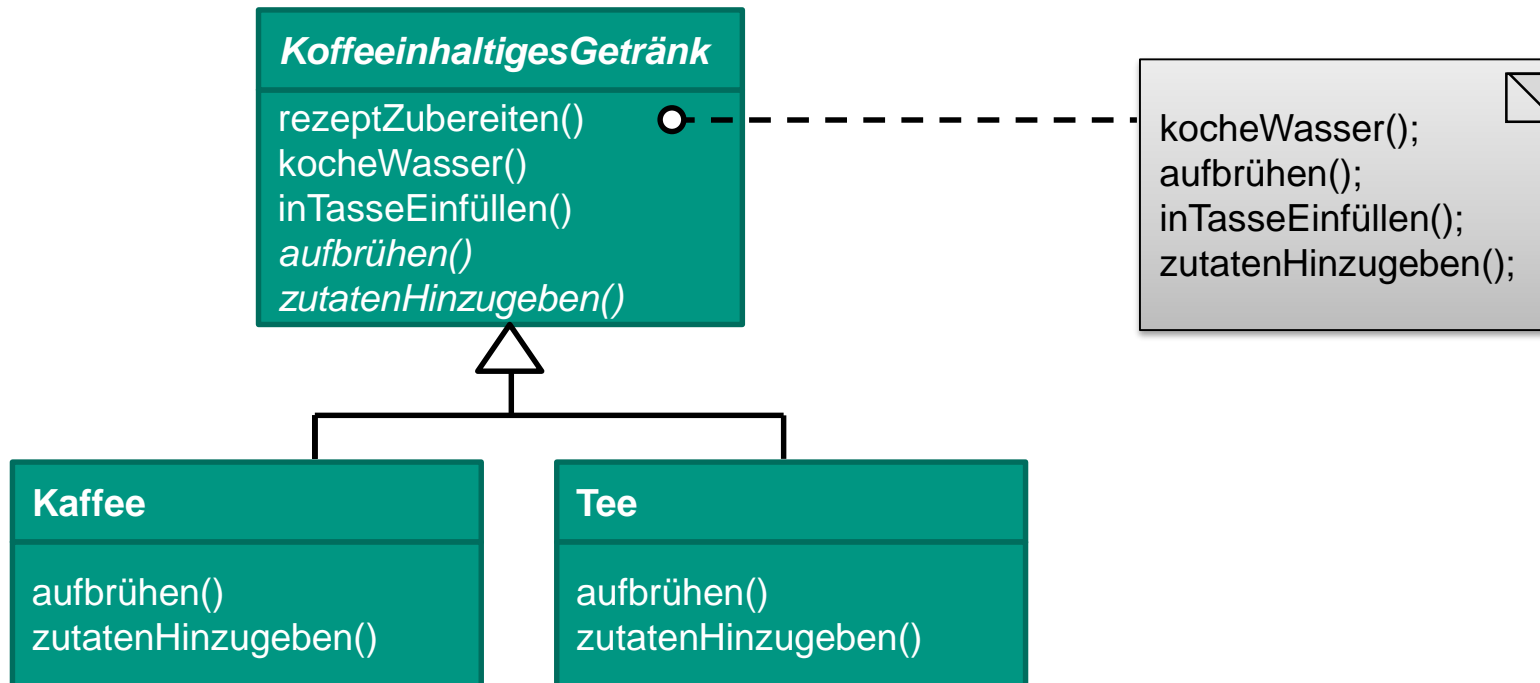
- Definiere das **Skelett eines Algorithmus** in einer Operation und delegiere einzelne Schritte an Unterklassen.
- Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, **ohne seine Struktur** zu verändern.



# Schablonenmethode: Struktur



# Schablonenmethode: Beispiel



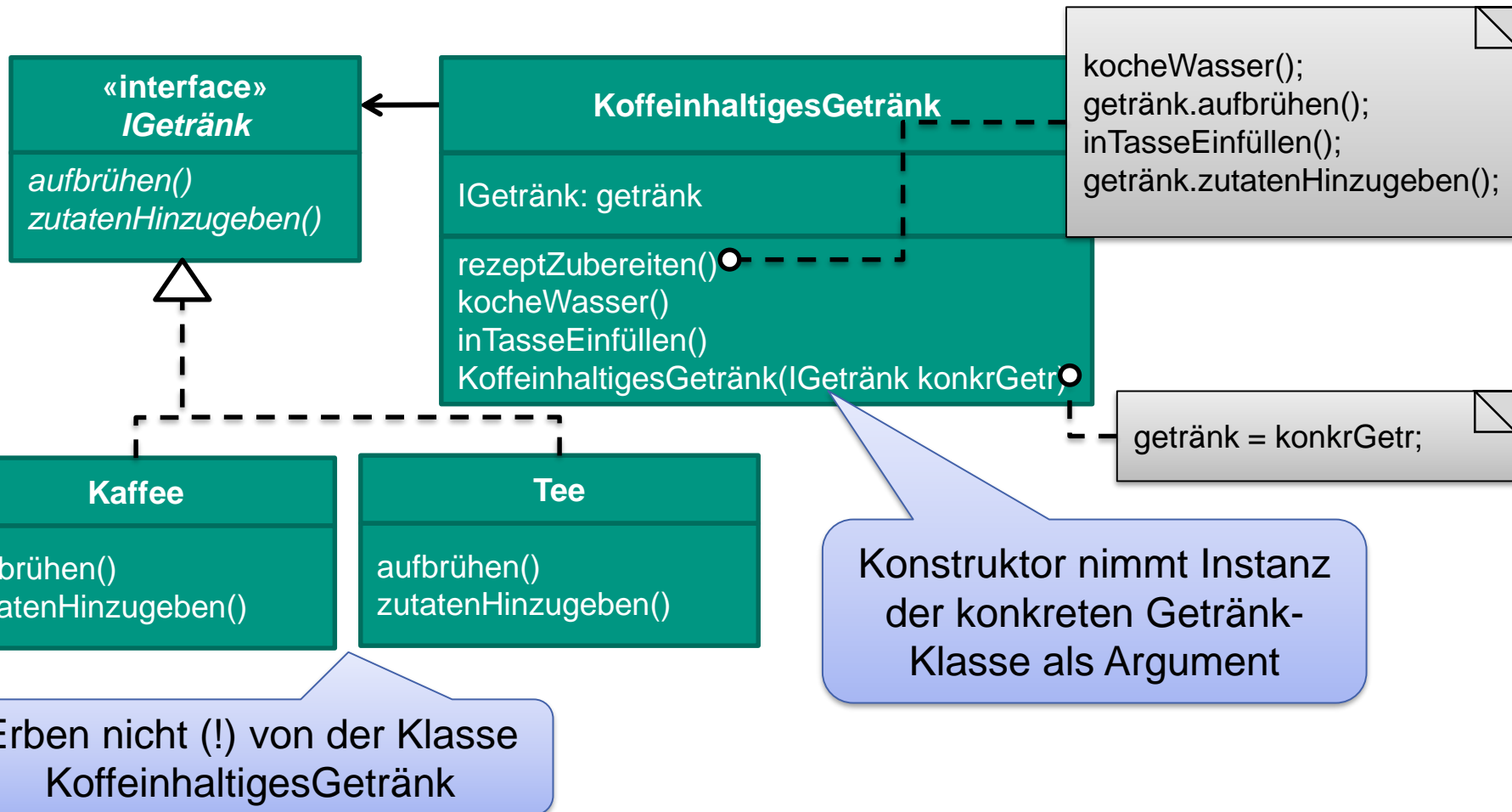
**Beispiel:** Head First Design Patterns, Seite 280 ff.

# Schablonenmethode: Anwendbarkeit

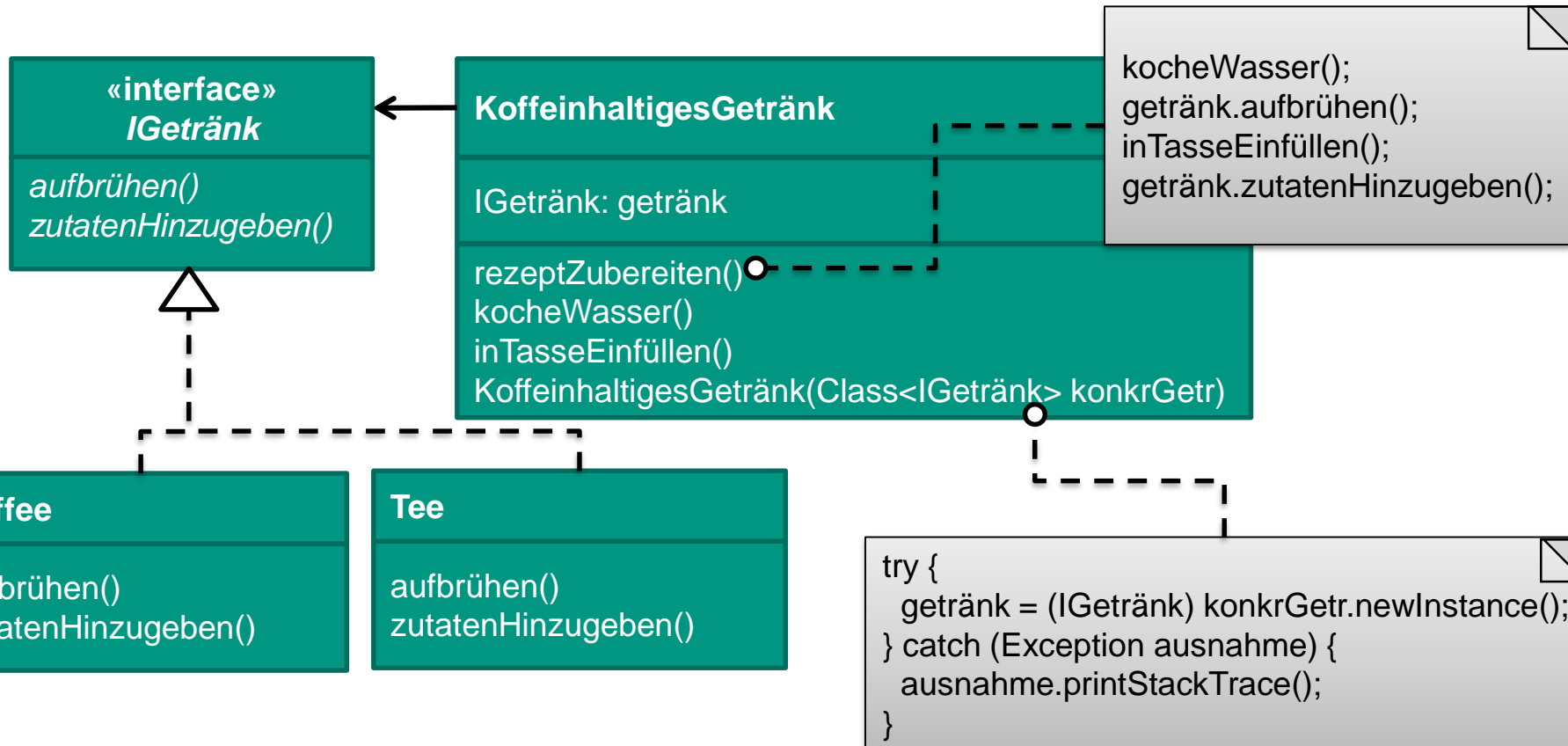
- Um die **invarianten Teile** eines Algorithmus **genau einmal** festzulegen und es dann Unterklassen zu überlassen, das variierende Verhalten zu implementieren.
- Wenn gemeinsames Verhalten aus Unterklassen **herausfaktoriert** und in einer allgemeinen Klasse platziert werden soll, um die Verdopplung von Code zu vermeiden.
- Um die Erweiterungen durch Unterklassen zu kontrollieren. Eine Schablonenmethode lässt sich so definieren, dass sie „**Einschubmethoden**“ (engl. *hooks*) an bestimmten Stellen aufruft und damit Erweiterungen nur an diesen Stellen zulässt.
- Häufig bei Rahmenarchitekturen genutzt.

**Nachlesen:** Head First Design Patterns, Kapitel 8

# Schablonenmethode – Alternative 1



# Schablonenmethode – Alternative 2: Generische Klassen



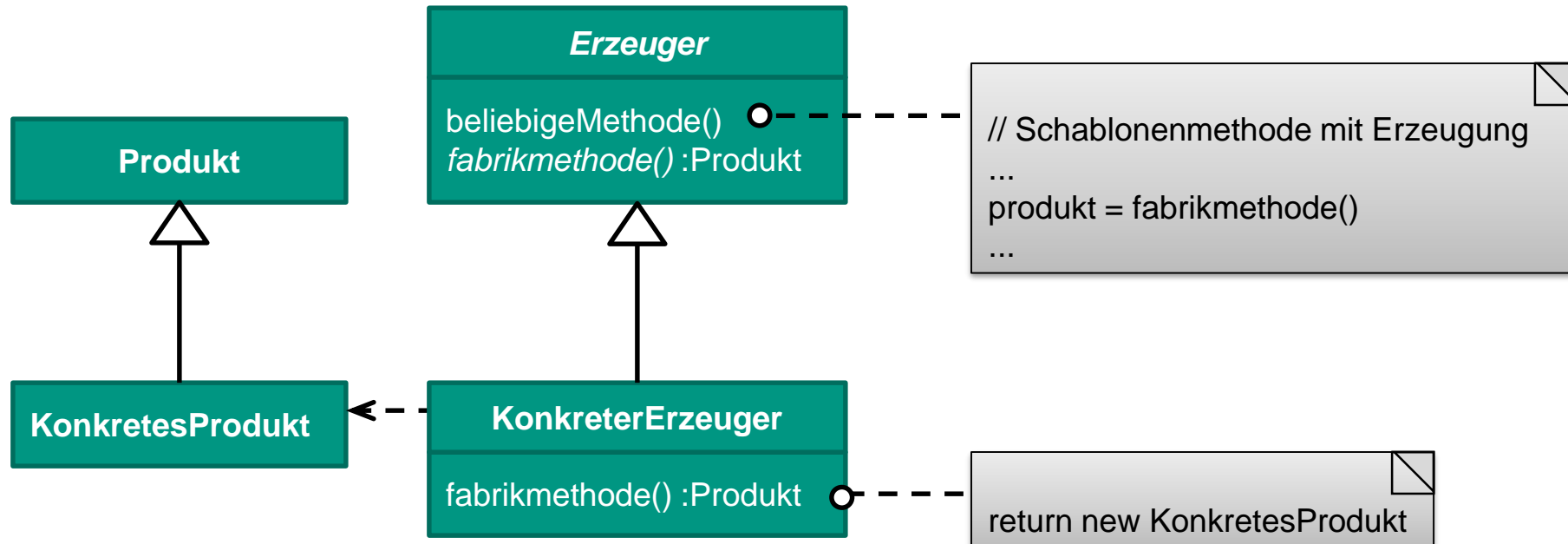
```
Class<IGetränk> t = (Class<IGetränk>)Class.forName("kit.ipd.Tee");  
KoffeinhaltigesGetränk tee = new KoffeinhaltigesGetränk(t);  
tee.rezeptZubereiten();
```

# Fabrikmethode (engl. *factory method*)

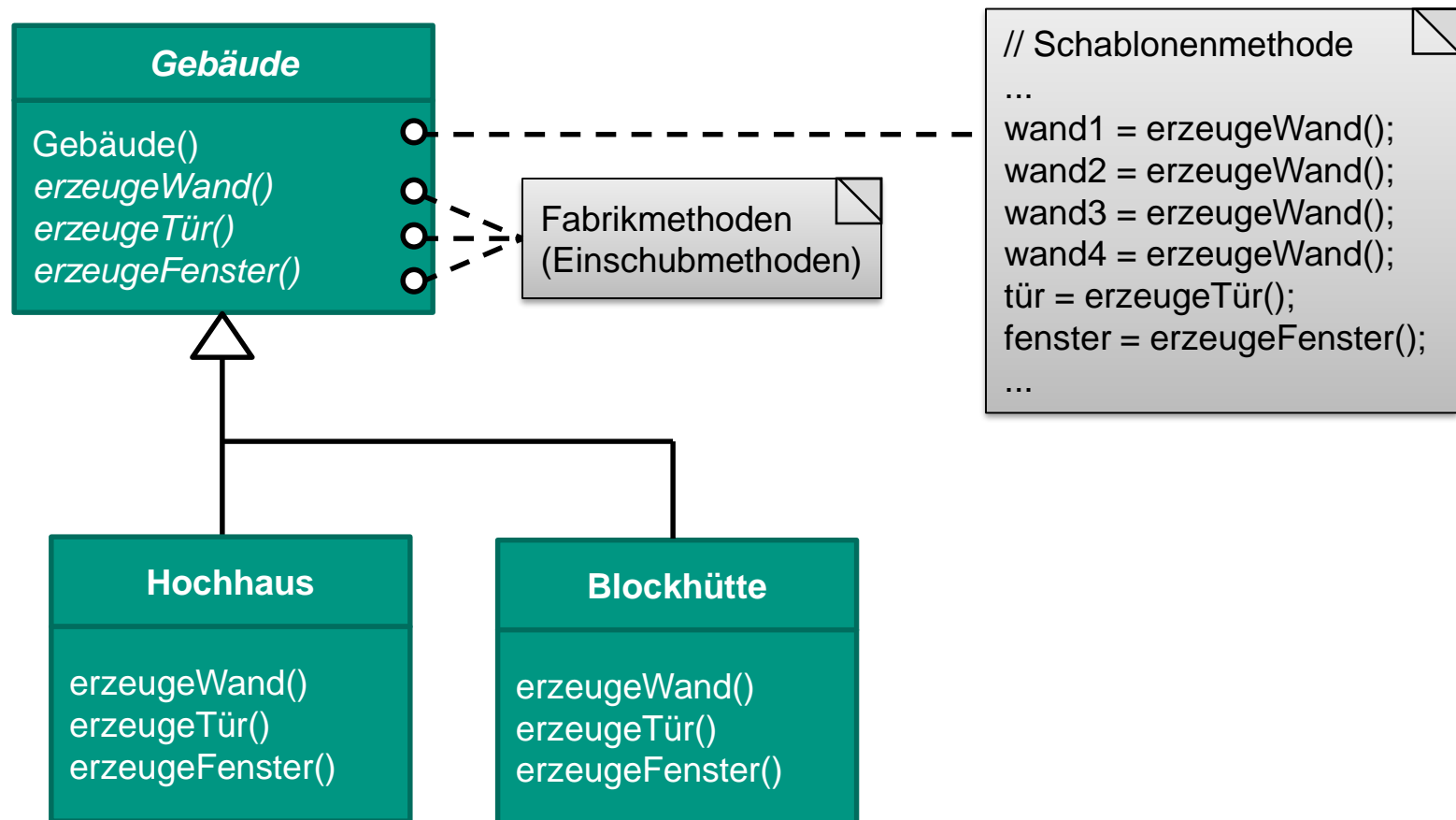
## ■ Zweck

- Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, **von welcher Klasse** das zu erzeugende Objekt ist.
- Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu **delegieren**.

# Fabrikmethode: Struktur



# Fabrikmethode: Beispiel





# Fabrikmethode: Anwendbarkeit

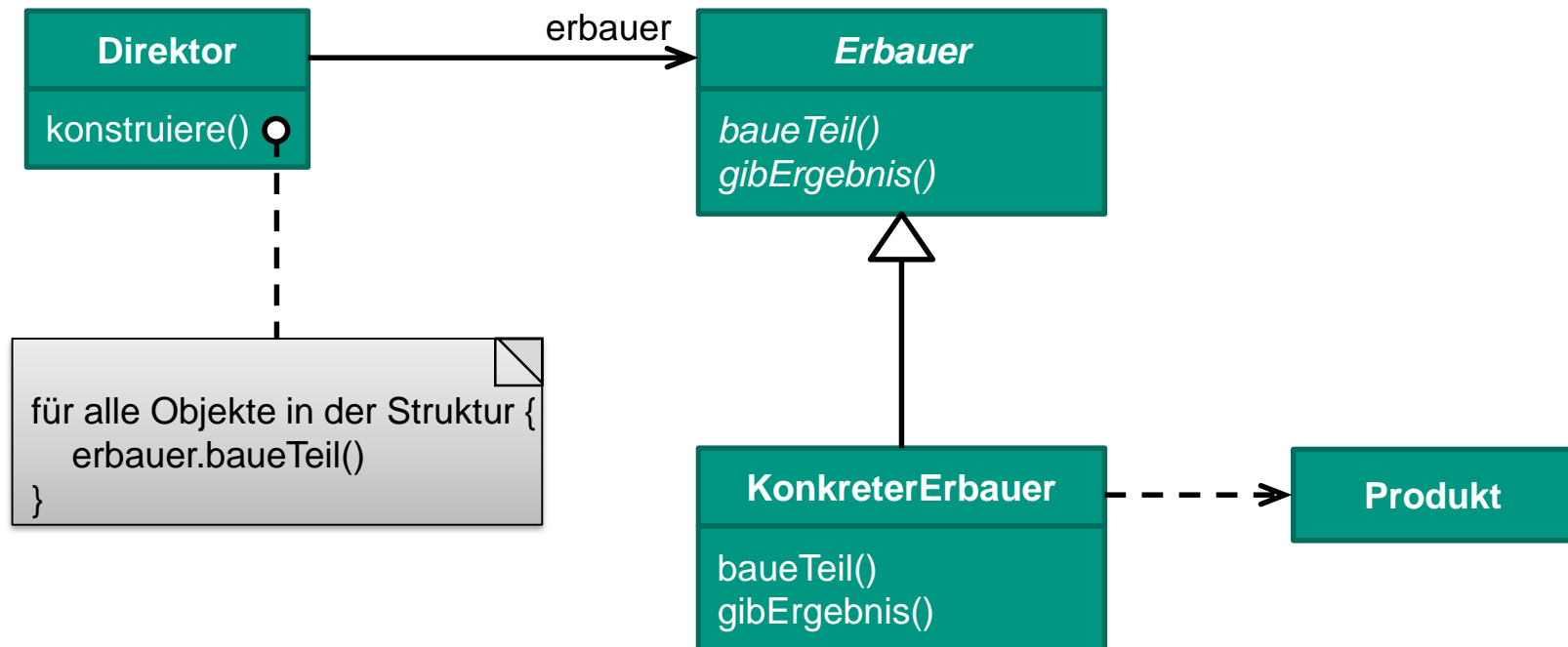
- Wenn eine Klasse die Klasse von Objekten, die sie erzeugen muss, nicht im Voraus kennen kann.
- Wenn eine Klasse möchte, dass ihre Unterklasse die von ihr zu erzeugenden Objekte festlegt.
- Wenn Klassen Zuständigkeiten an eine von mehreren Hilfsunterklassen delegieren sollen und das Wissen, an welche Hilfsunterklasse die Zuständigkeit delegiert wird, lokalisiert werden soll.
- Eine Fabrikmethode ist die Einschubmethode bei einer Schablonenmethode für Objekterzeugung.

# Erbauer (engl. *builder*) Nicht prüfungsrelevant

## ■ Zweck

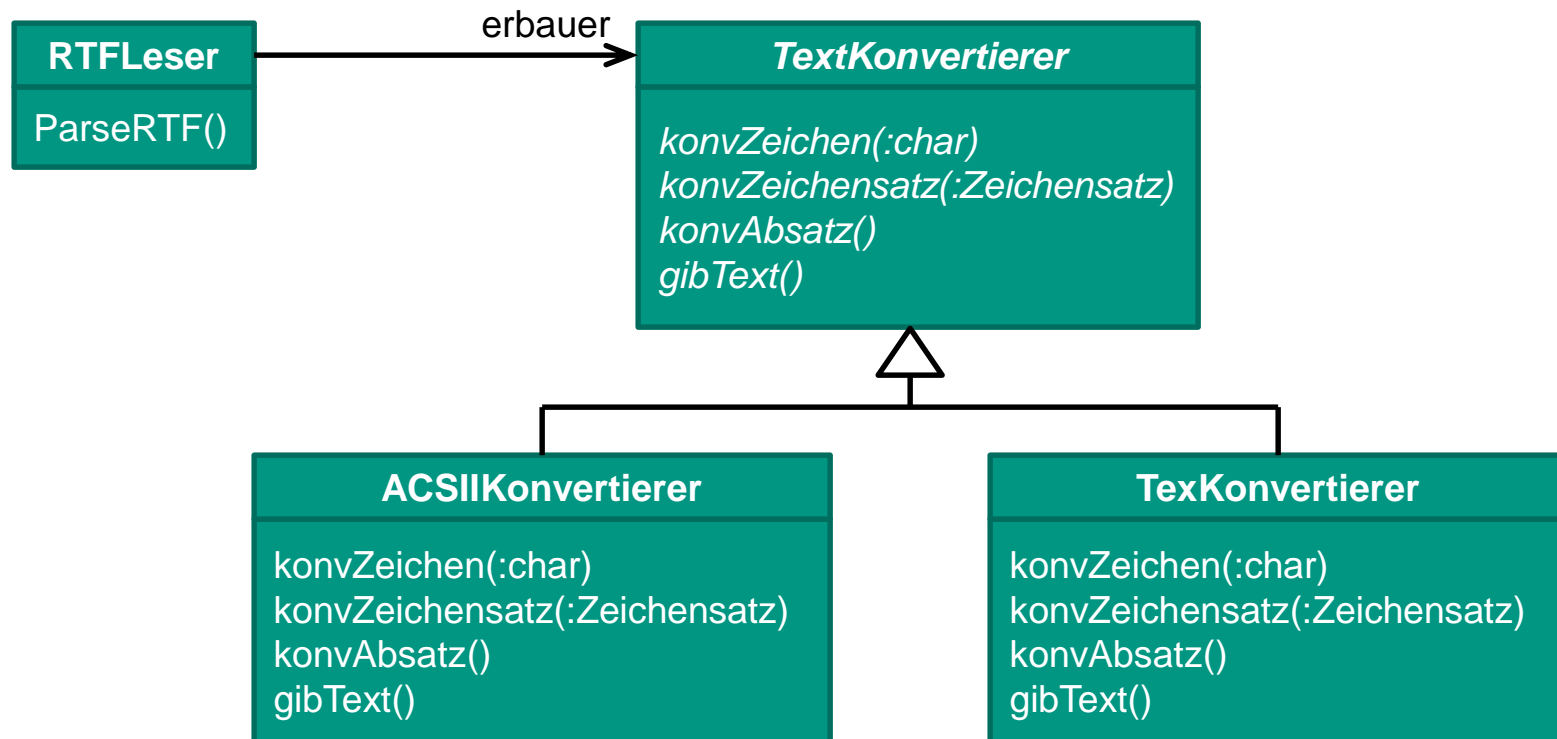
- Trenne die Konstruktion eines komplexen Objekts (bestehend aus mehreren Teilen) von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.

# Erbauer: Struktur Nicht prüfungsrelevant



# Erbauer: Beispiel Nicht prüfungsrelevant

## ■ Transformation zwischen Textformaten



## **Erbauer: Anwendbarkeit** Nicht prüfungsrelevant

- Der Algorithmus zum Erzeugen eines komplexen Objekts soll unabhängig von den Teilen sein, aus denen das Objekt besteht und wie sie zusammengesetzt werden.
- Der Konstruktionsprozess muss verschiedene Repräsentationen des zu konstruierenden Objekts erlauben.

## **Erbauer: Interaktionen** Nicht prüfungsrelevant

- Der Klient erzeugt das Direktorobjekt und konfiguriert es mit dem gewünschten Erbauerobjekt.
- Der Direktor informiert den Erbauer, wenn ein Teil des Produkts gebaut werden soll.
- Der Erbauer bearbeitet die Anfragen des Direktors und fügt Teile zum Produkt hinzu.
- Der Klient erhält das Produkt vom Erbauer.

## Fabrikmethode vs. Erbauer Nicht prüfungsrelevant

- Der Erbauer trennt den Konstruktionsalgorithmus und die Schnittstelle zum Bauen der einzelnen Teile (Direktor und Erbauerklassen). Daher ist es möglich, die Erbauerklasse und damit die Repräsentation der Einzelteile und des gesamten Produkts zu variieren (auch dynamisch).

# Kompositum (engl. *composite*)

## Zweck

Füge Objekte zu **Baumstrukturen** zusammen, um Bestands-Hierarchien zu repräsentieren. Das Muster ermöglicht es Klienten, sowohl einzelne Objekte als auch Aggregate einheitlich zu behandeln.

Synonyme: Whole-Part, Bestandshierarchie



# Kompositum

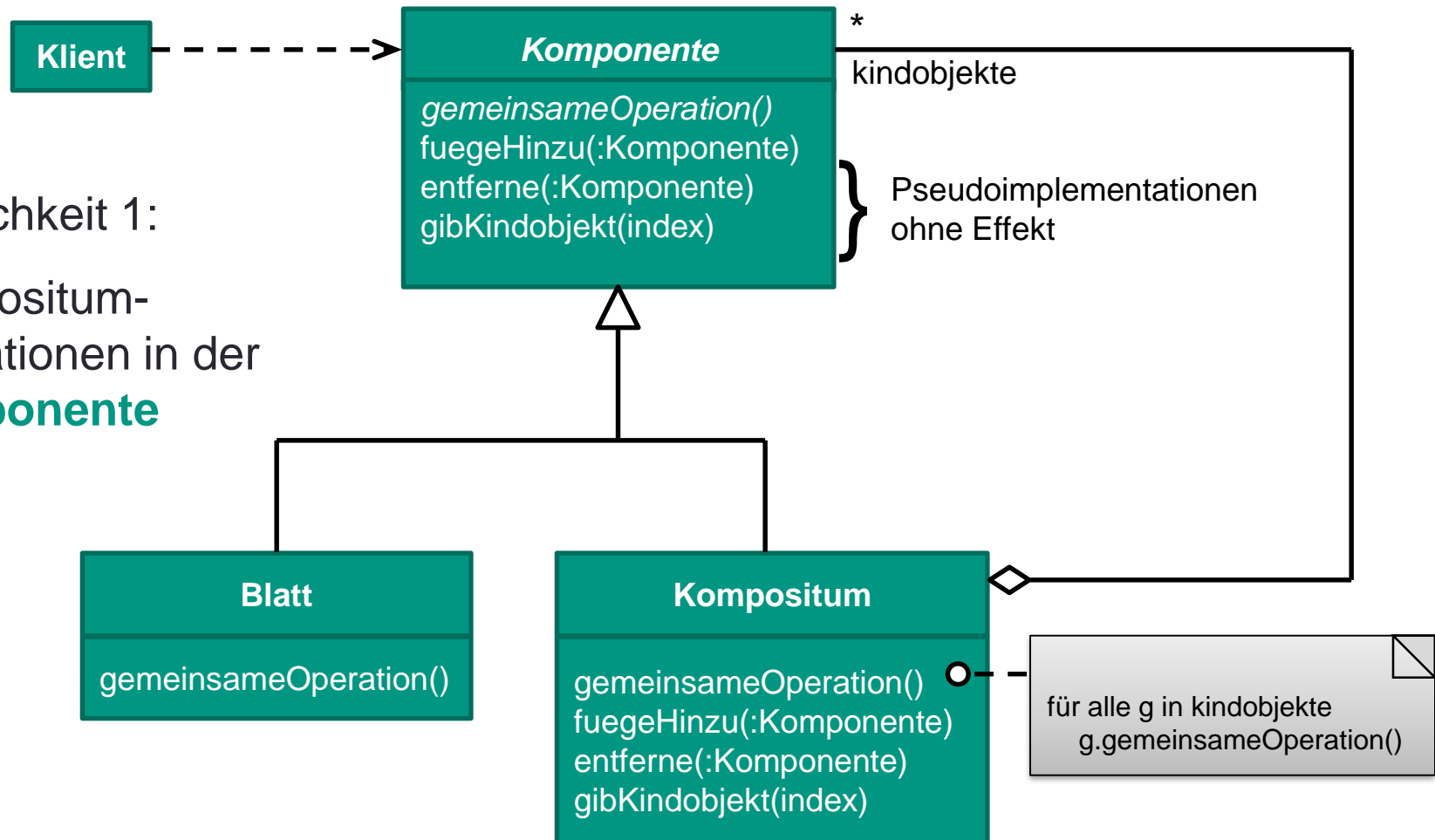
## ■ Motivation

- Bestands-Hierarchien treten überall dort auf, wo komplexe Objekte modelliert werden, wie beispielsweise Dateisysteme, graphische Anwendungen, Textverarbeitung, CAD, CIM,...
- Bei diesen Anwendungen werden einfache Objekte zu Gruppen **zusammengefasst**, welche wiederum zu **größeren Gruppen** zusammengefügt werden können.
- Häufig soll dabei die Behandlung von Objekten und Aggregaten durch das Programm **einheitlich** sein. Das Kompositum isoliert die gemeinsamen Eigenschaften von Objekt und Aggregat und bildet daraus eine Oberklasse.

# Kompositum: Struktur (1)

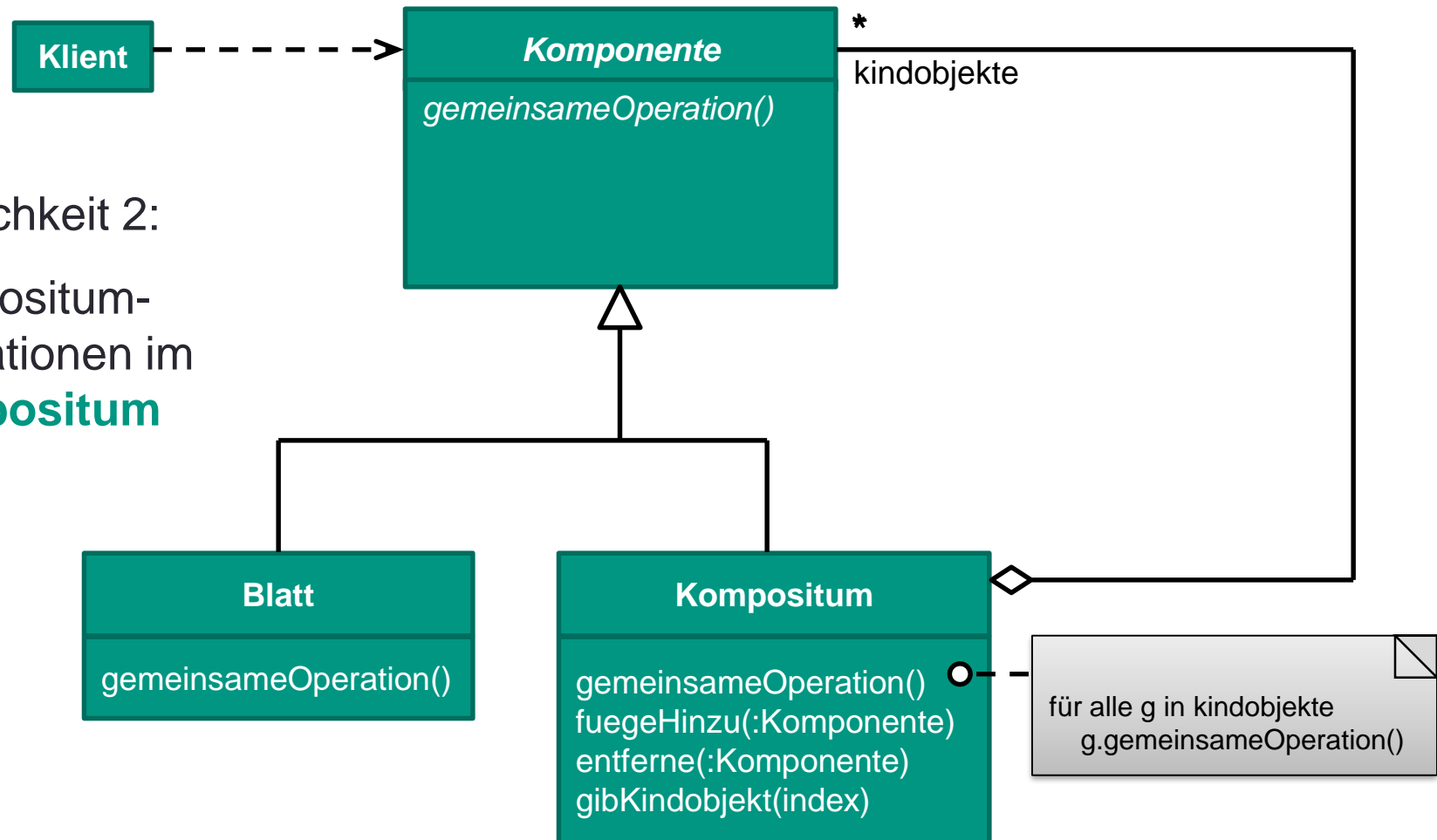
Möglichkeit 1:

Kompositum-  
Operationen in der  
**Komponente**



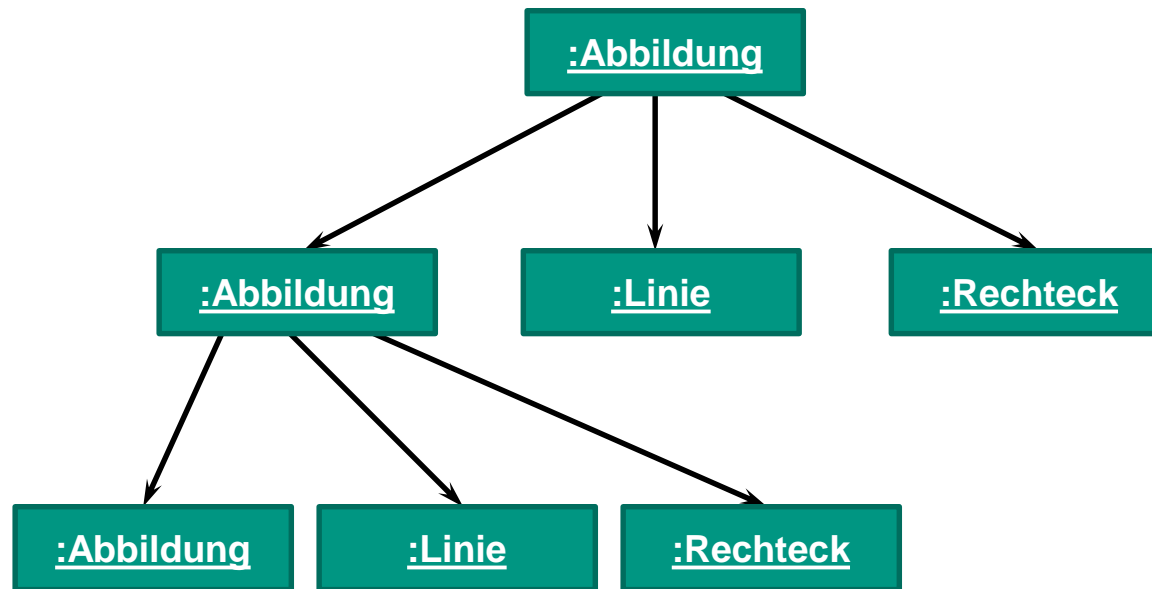
# Kompositum: Struktur (2)

Möglichkeit 2:  
Kompositum-  
Operationen im  
**Kompositum**



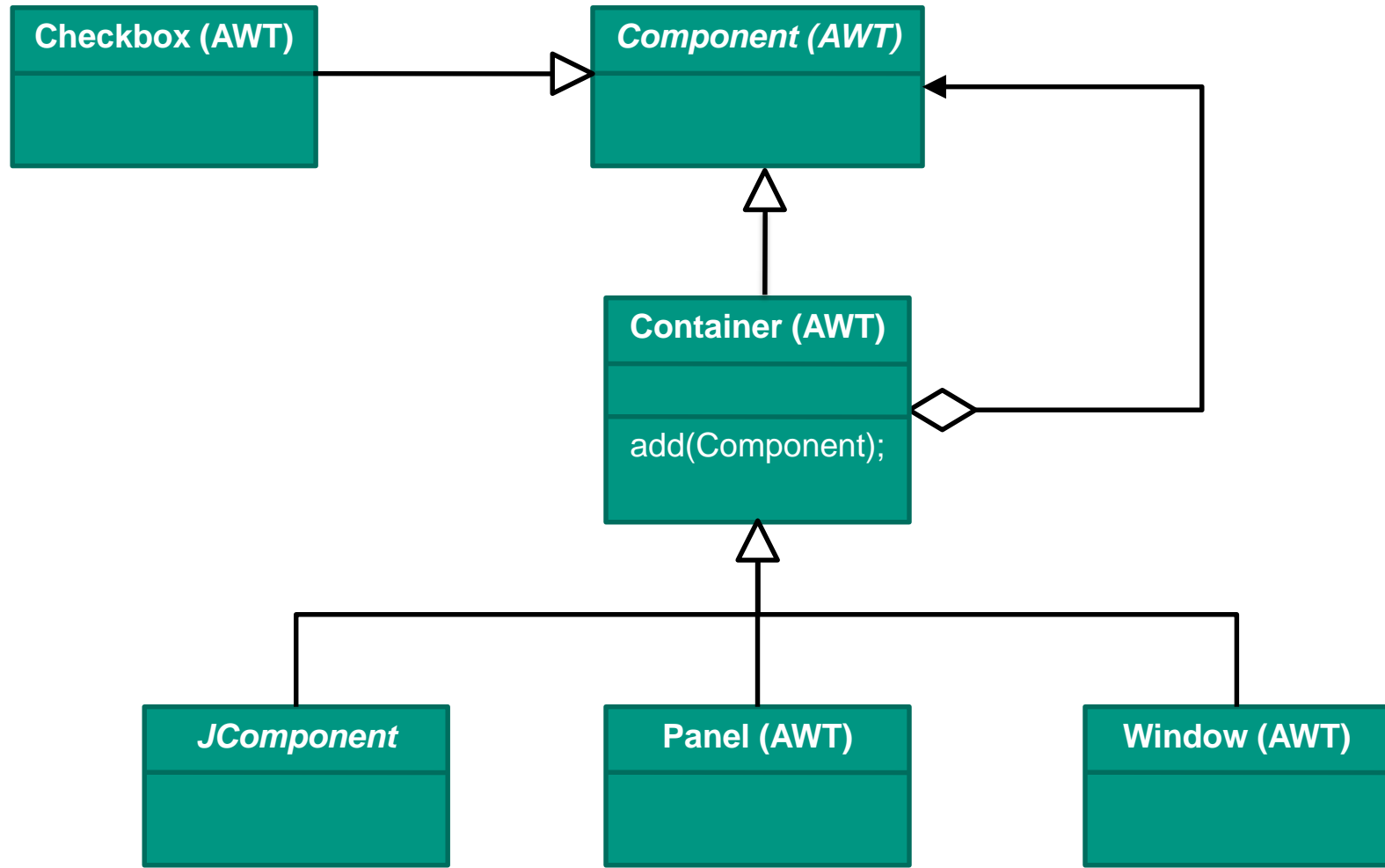
# Kompositum: Beispiel

- Zusammengefügte Graphik-Objekte

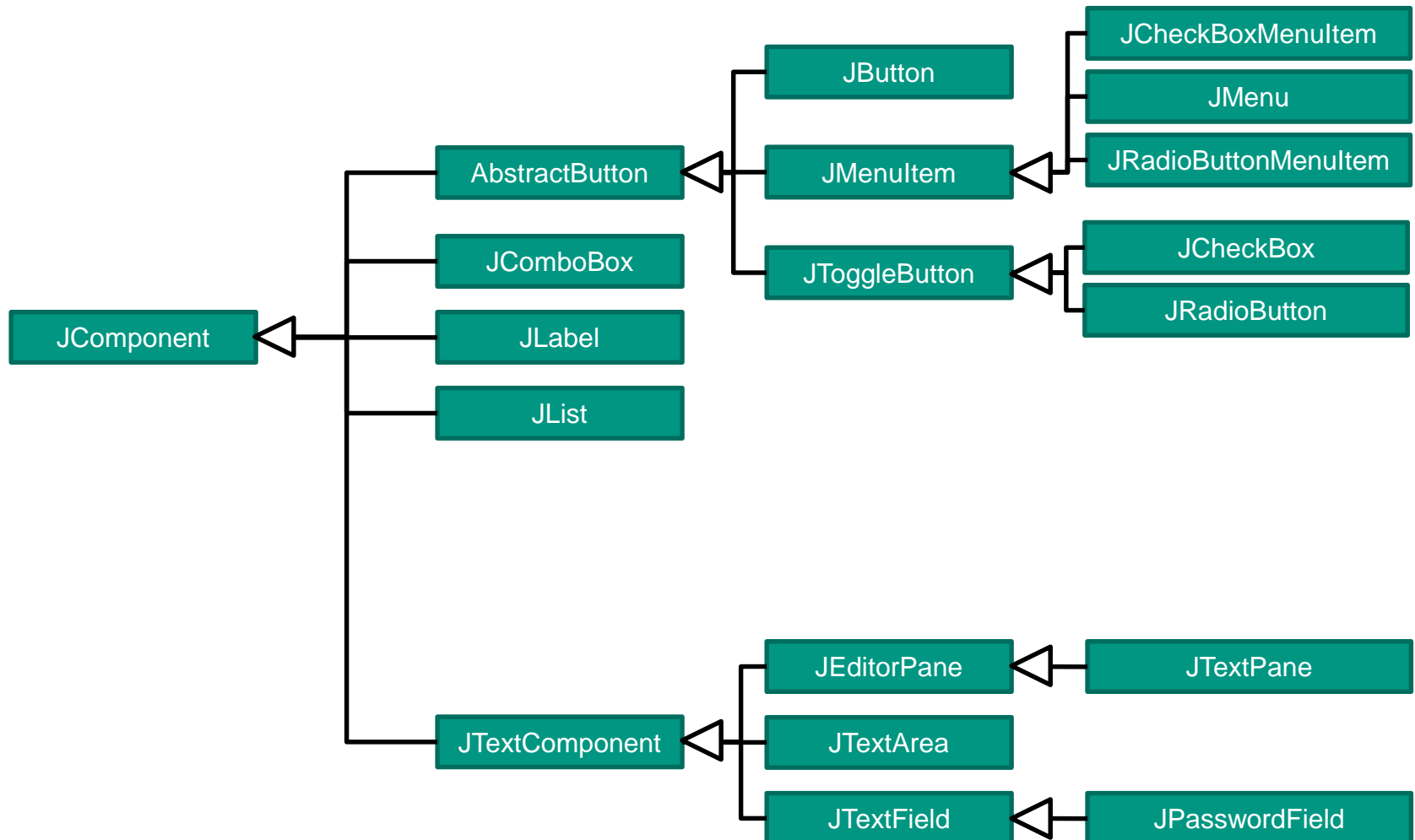


- gemeinsame Operationen: `zeichne()`, `verschiebe()`, `lösche()`, `skaliere()`

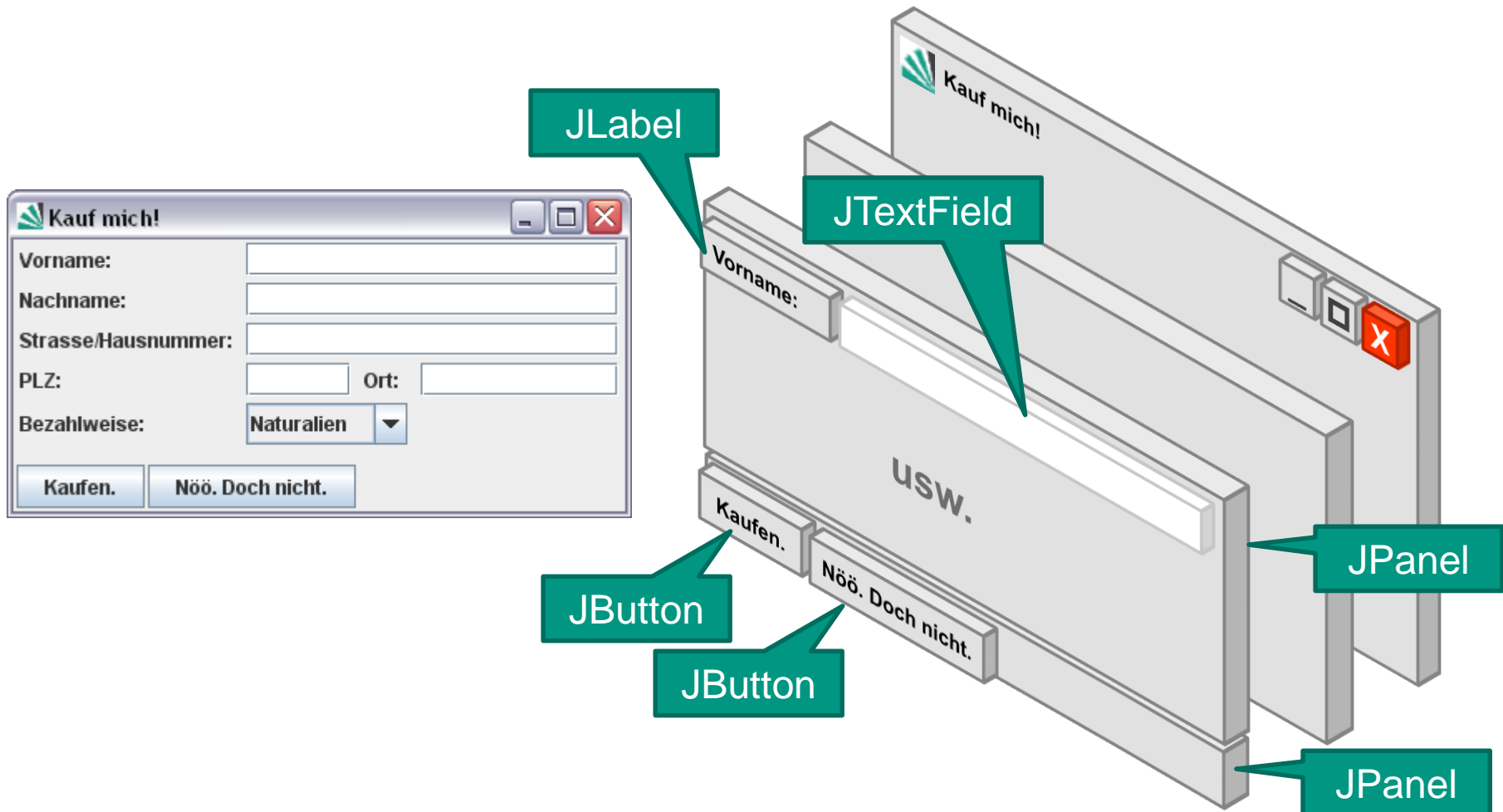
# Kompositum: Beispiel aus Java (1)



# Kompositum: Beispiel aus Java (2)



# Kompositum: Beispiel aus Java (3)



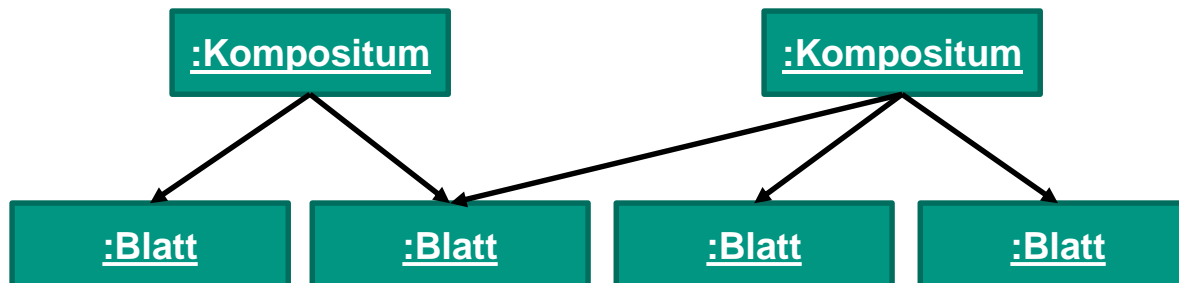
# Kompositum: Anwendbarkeit

- Die Klasse Kompositum enthält und manipuliert die Behälter-Datenstruktur, die die Komponenten speichert.
  
- Anwendbarkeit
  - Wenn Bestands-Hierarchien von Objekten repräsentiert werden sollen.
  - Wenn die Klienten in der Lage sein sollen, die Unterschiede zwischen zusammengesetzten und einzelnen Objekten zu ignorieren.



# Kompositum: Implementierung (1)

1. Es ist häufig nützlich eine Eltern-Referenz in jeder Komponente zu führen (erleichtert Traversierung).
  - Diese Referenz kann von den `fügeHinzu()`- und `entferne()`-Methoden des Kompositums gepflegt werden.
  - Das Teilen von Komponenten kann zu mehreren Eltern und damit zu Zweideutigkeiten führen.



# Kompositum: Implementierung (2)

## 2. Maximieren der Komponenten-Schnittstelle

- Die Komponenten-Schnittstelle sollte so viele gemeinsame Methoden des Kompositums und der Blätter wie möglich definieren um **Transparenz** zu garantieren.
- Wenn Methoden des Kompositums in der Komponente definiert werden, sollte `gibKindobjekt()` bei Blättern nichts zurückgeben – das kann auch durch eine entsprechende Implementierung in der Komponente erreicht werden.
- `fuegeHinzu()` und `entferne()` sollten bei Blättern fehlschlagen und einen Fehler zurückgeben oder eine Ausnahme generieren.

# Kompositum: Implementierung (3)

## 3. Speichern der Kindelemente

- Felder, Listen, Mengen oder Hash-Tabellen – abhängig von der Anwendung und benötigten Effizienz.
- Bei einer festen Anzahl Kinder verwende explizite Variablen und spezialisierte `fuegeHinzu()`/`entferne()`/`gibKindobjekt()` Operationen.
  - z.B. bei einem Binärbaum: linkes und rechtes Kind, `setzeLinks`, `setzeRechts`, `holeLinks`, `holeRechts`.
- Das Auslesen der Kindobjekte durch einen Iterator verwirklichen.
- Die Kinder müssen unter Umständen in einer bestimmten Reihenfolge gelassen werden (beispielsweise bei Anordnern (Layout Manager)).

# Strategie (engl. *strategy*)

## Zweck

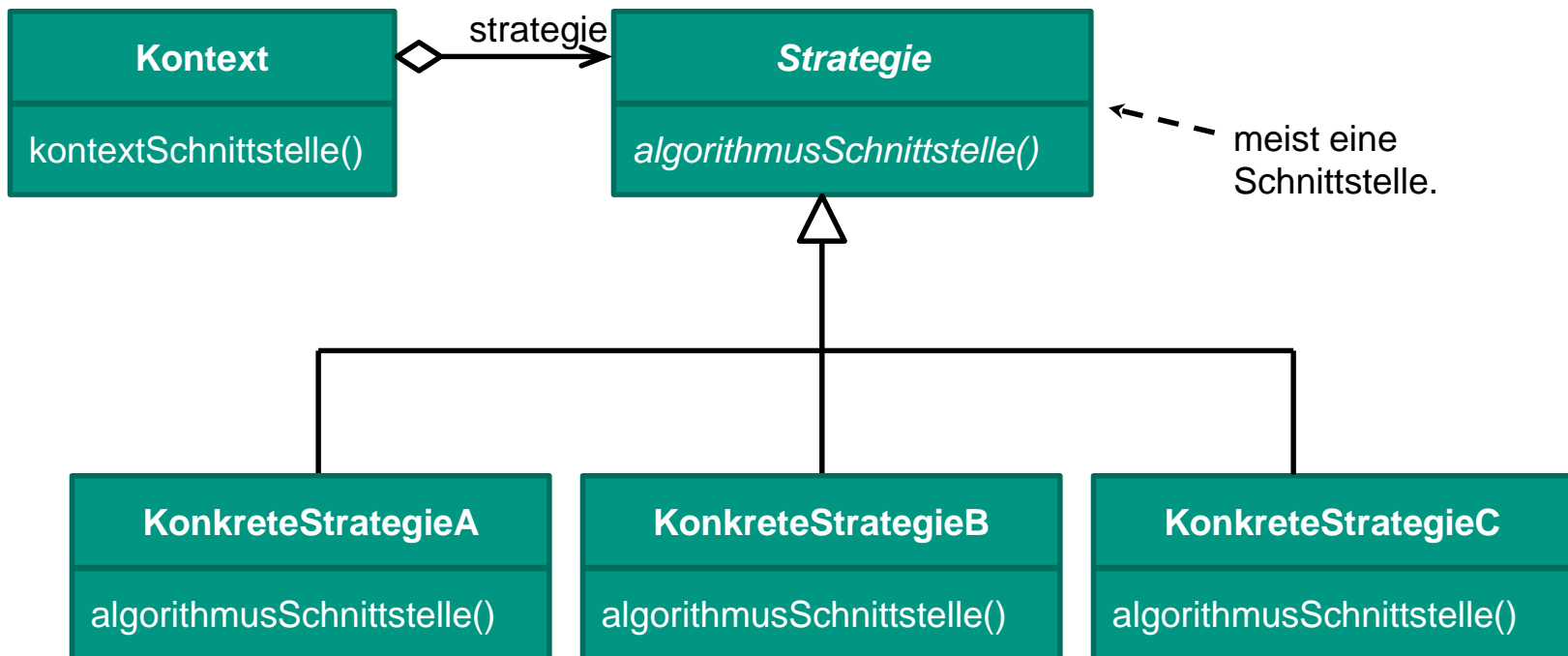
Definiere eine **Familie von Algorithmen**, kapsle sie und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von nutzenden Klienten zu variieren.

Synonyme: *Policy*

## Motivation

Manchmal müssen Algorithmen, abhängig von der notwendigen Performanz, der Menge oder des Typs der Daten, variiert werden.

# Strategie: Struktur



# Verwendung der Strategie

```
-- wähle geeignete Strategie
Strategie passendeStrategie;
switch (...) {
case 1: passendeStrategie=new KonkreteStrategieA;
        break;
case 2: passendeStrategie=new KonkreteStrategieB;
        break;
default:passendeStrategie=new KonkreteStrategieC;
        break;
}
-- verwende gewählte Strategie
passendeStrategie.algorithmusSchnittstelle();

...
```

# Beispiel 1: An assignment far, far away....



**Programmieren – Wintersemester 2014/15**  
Software-Design und Qualität (SDQ)  
<https://sdqweb.ipd.kit.edu/wiki/Programmieren>  
Prof. Dr. Ralf H. Reussner · Kiana Rostami · Philipp Merkle

---

## Übungsblatt 4

Ausgabe: 08.12.2014 13:00  
Abgabe: 22.12.2014 13:00

---

## C Warteschlangen-Simulation (12 Punkte)

### Warteschlangen- system

In dieser Aufgabe modellieren Sie ein solches Warteschlangensystem, bestehend aus *Aufträgen* (Personen, Prozesse, ...), die in *Wartebereichen* (Kassenbereich, Wartezimmer, ...) darauf warten durch *Bedieneinheiten* (Kassierer, CPU, Festplatte, ...) bedient zu werden. Darauf basierend schreiben Sie eine einfache *Simulation*, die für ein gegebenes Szenario beantwortet wie lange die einzelnen Aufträge warten mussten, um vollständig bedient zu werden.

### C.5 Aufgabenstellung

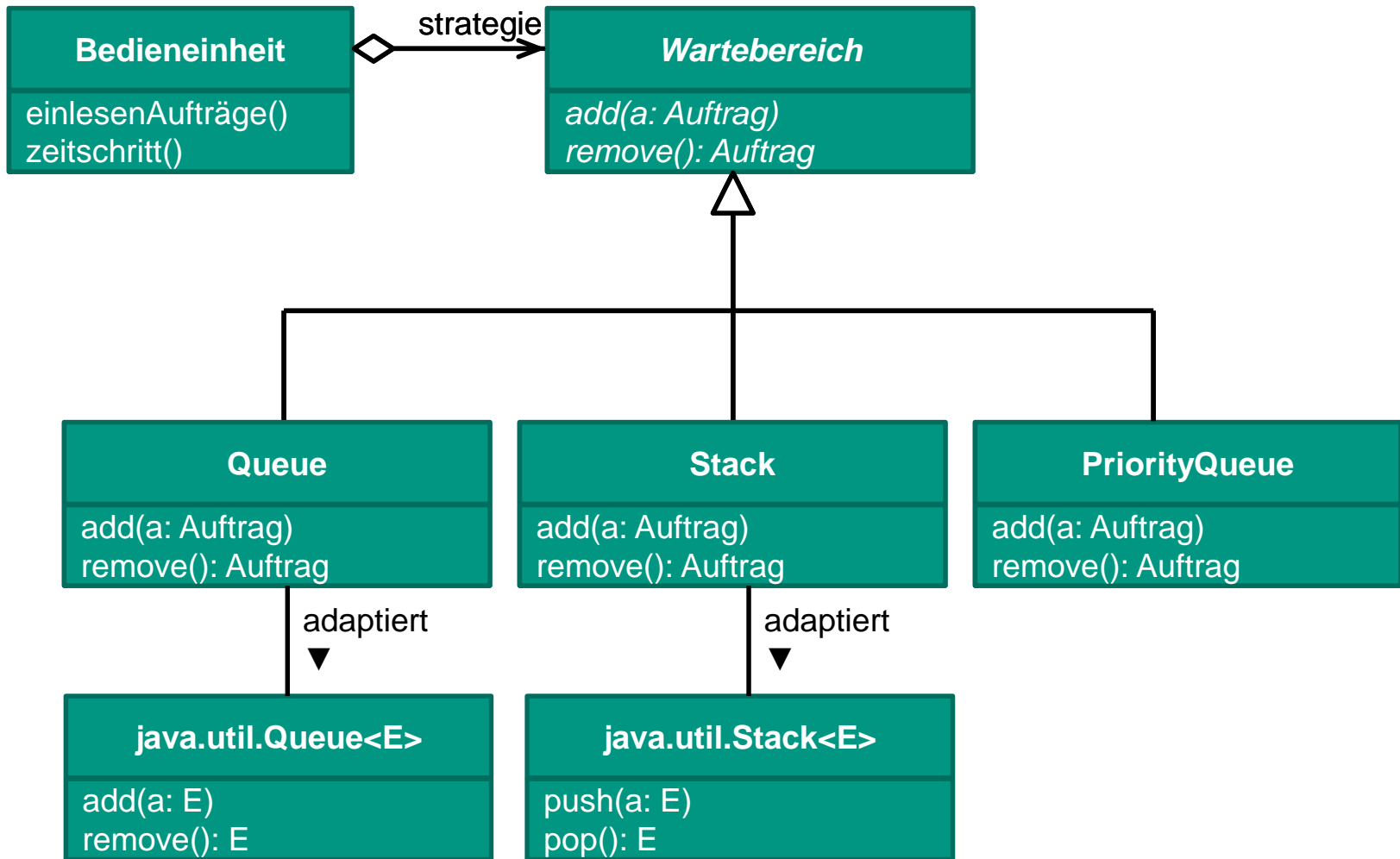
Implementieren Sie:

“Wartebereiche vom Typ Queue, Stack sowie Priority Queue”

- *Aufträge* vom Typ *SimpleJob* und *ComplexJob* wie in Abschn. C.2 beschrieben.
- *Wartebereiche* vom Typ *Queue*, *Stack* sowie *Priority Queue* wie in Abschn. C.1 bzw. Abschn. C.3 beschrieben. Beachten Sie dabei die Implementierungshinweise.
- die *Simulierte Bedieneinheit* wie in Abschn. C.4 beschrieben. Ihr Programm soll über genau eine simulierte Bedieneinheit verfügen.

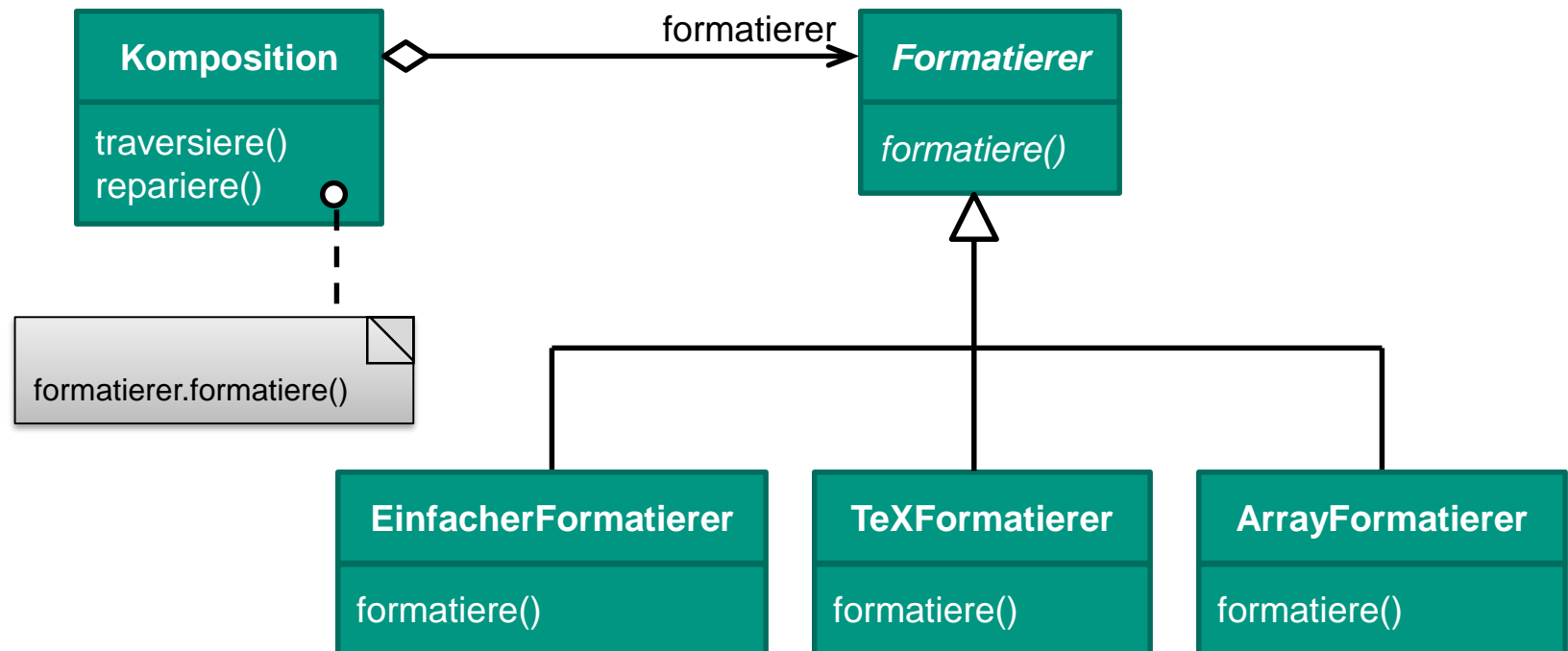


# Lösung: Man nehme das Strategiediagramm und ändere die Bezeichner



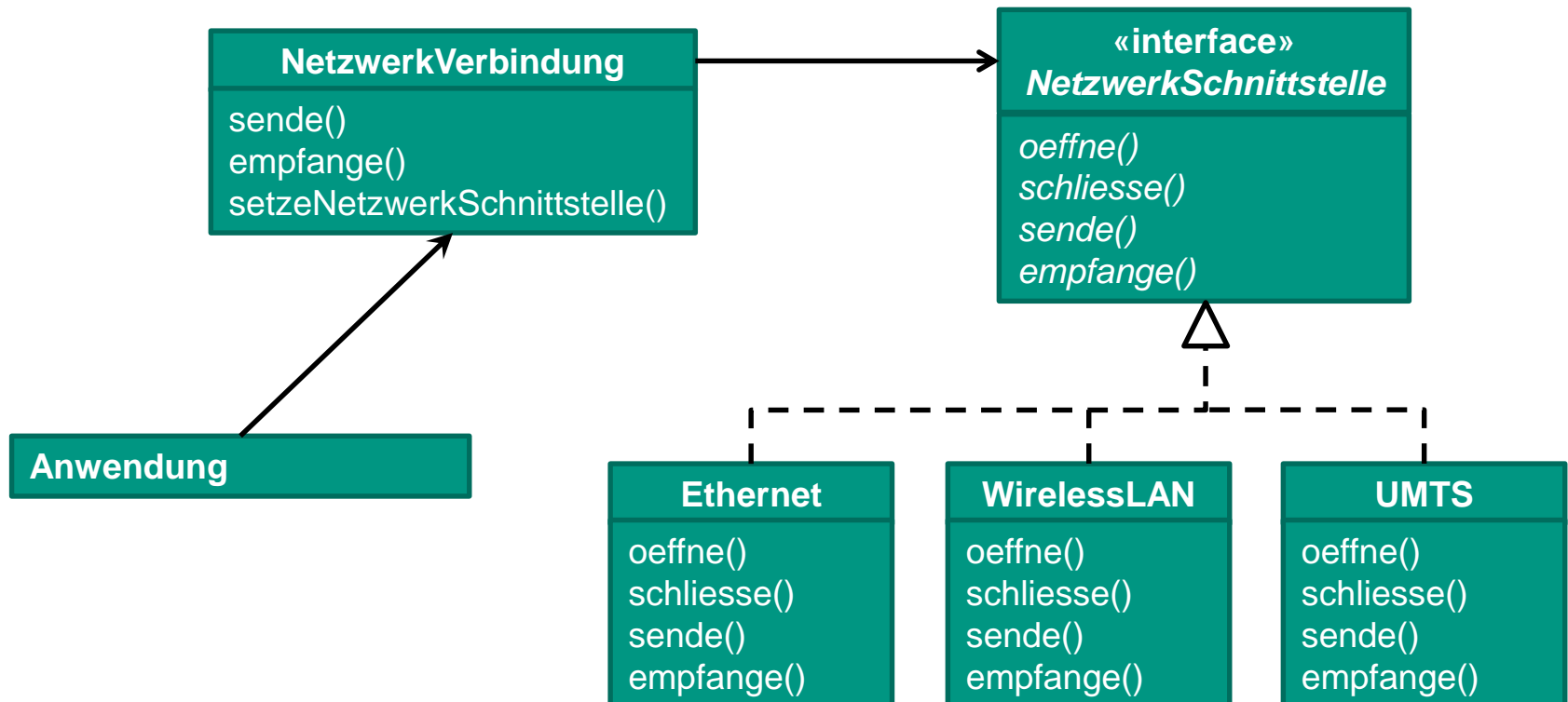
# Strategie: Beispiel 2

## Kapselung von Zeilenumbrechalgorithmen in Klassen



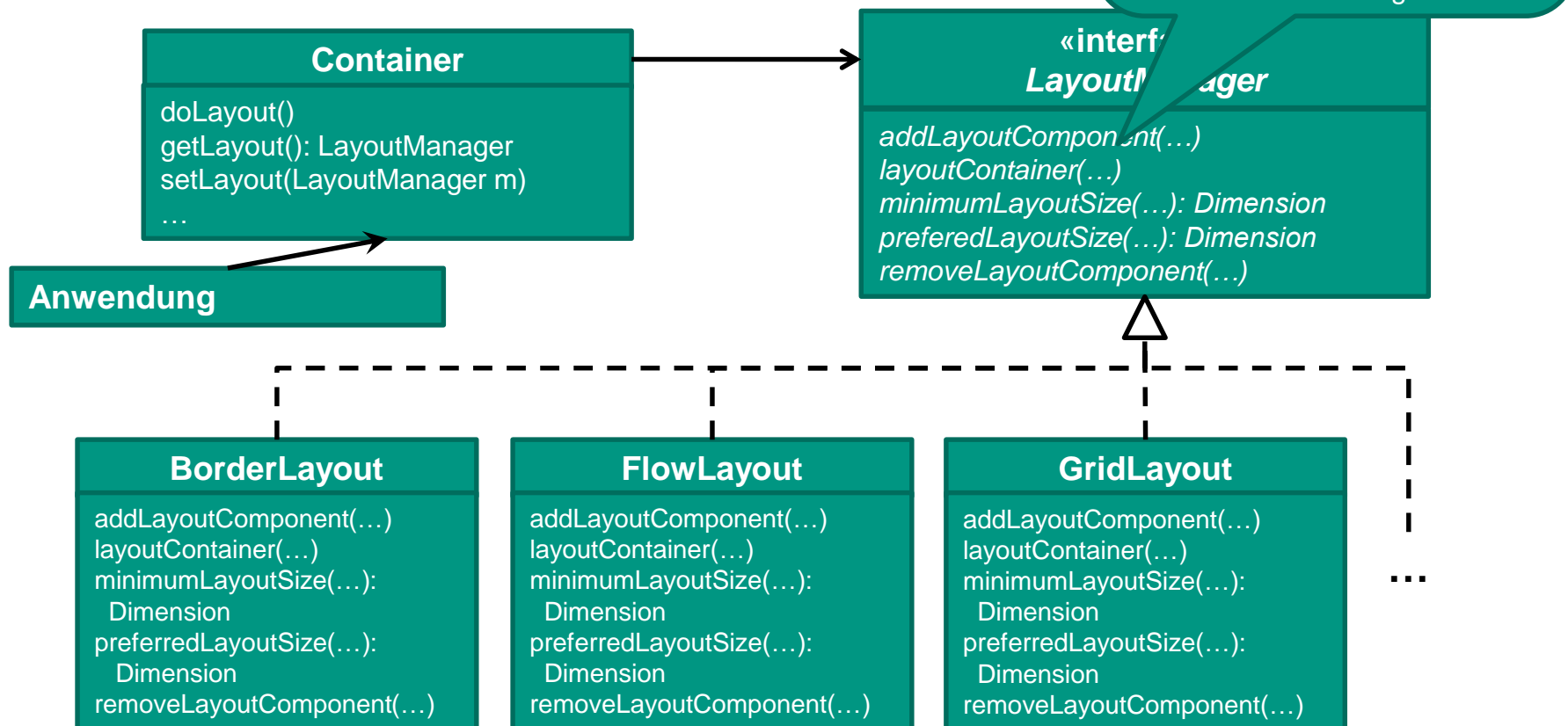
# Strategie: Beispiel 3

- Kapselung mehrerer Implementierungen einer Netzwerkschnittstelle.



# Strategie: Beispiel 4

- Implementierung der Anordner (**LayoutManager**) in AWT und Swing.



## Strategie: Anwendbarkeit (1)

- Wenn sich viele verwandte Klassen nur in ihrem Verhalten unterscheiden. Strategieobjekte bieten die Möglichkeit, eine Klasse mit einer von mehreren möglichen Verhaltensweisen zu konfigurieren.
- Wenn unterschiedliche Varianten eines Algorithmus benötigt werden.
- Wenn ein Algorithmus Datenstrukturen verwendet, die Klienten nicht bekannt sein sollen.
- Wenn eine Klasse unterschiedliche Verhaltensweisen definiert und diese als mehrfache Fallunterscheidungen in ihren Operationen erscheinen. Mit Strategie kann man diese Fallunterscheidungen vermeiden („*switch-less programming*“).

**Nachlesen:** Head First Design Patterns, Kapitel 1

# Switch-less Programming

```

m1() {
  switch(...) {
    case 1: code m1.1;
    :
    case i: code m1.i;
    :
    case n: code m1.n;
  }
}

m2() {
  switch(...) {
    case 1: code m2.1;
    :
    case i: code m2.i;
    :
    case n: code m2.n;
  }
}

m3() {
  switch(...) {
    case 1: code m3.1;
    :
    case i: code m3.i;
    :
    case n: code m3.n;
  }
}

```

- Lässt sich umwandeln in n Strategien mit Methoden m1, m2, m3

```

class Strategie_i extends Strategie {
  m1() { code m1.i }
  m2() { code m2.i }
  m3() { code m3.i }
}

```

- Kontext enthält eine switch-Anweisung:

```

Strategie passendeStrategie;
switch(...) {
  :
  case i: passendeStrategie = new Strategie_i();
  :
}

// Verwendung
passendeStrategie.m1()
:
passendeStrategie.m2()
:

```

## Strategie: Anwendbarkeit (2)

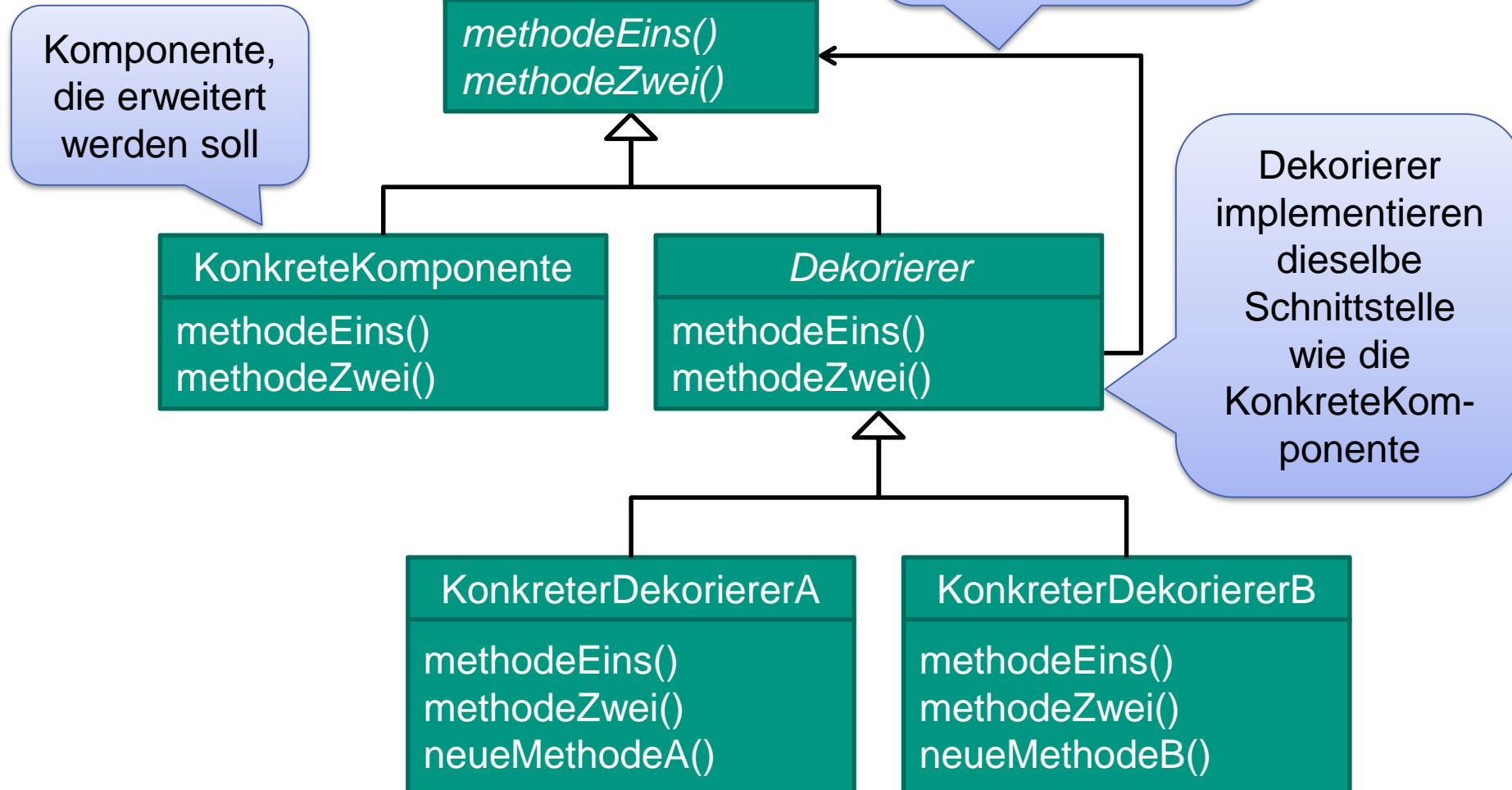
- Alternativ zur Ableitung der Klasse **Strategie** könnte man auch die Klasse **Kontext** ableiten, um verschiedene Verhaltensmuster zu implementieren.
- Das Ergebnis sind viele Klassen, die sich nur im Verhalten unterscheiden, welches für jede Klasse fest ist.
- Das Strategie-Muster erlaubt demgegenüber auch eine dynamische Veränderung des Verhaltens.

# Dekorierer (engl. *Decorator*)

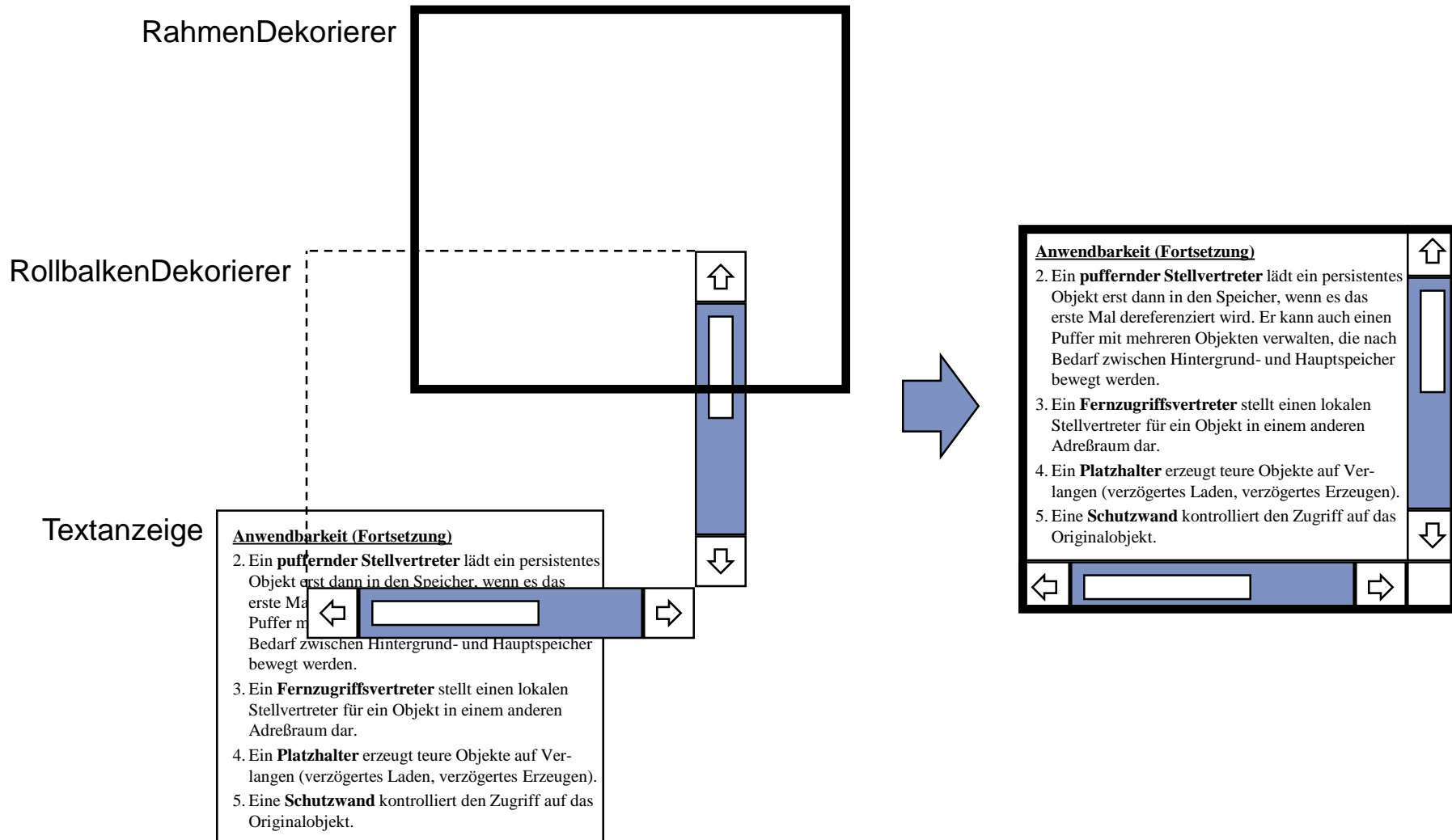
- Zweck
  - Fügt dynamisch neue Funktionalität zu einem Objekt hinzu.
- Kann alternativ zu Vererbung verwendet werden
- Achtung: Nicht mit dem Stellvertreter verwechseln!



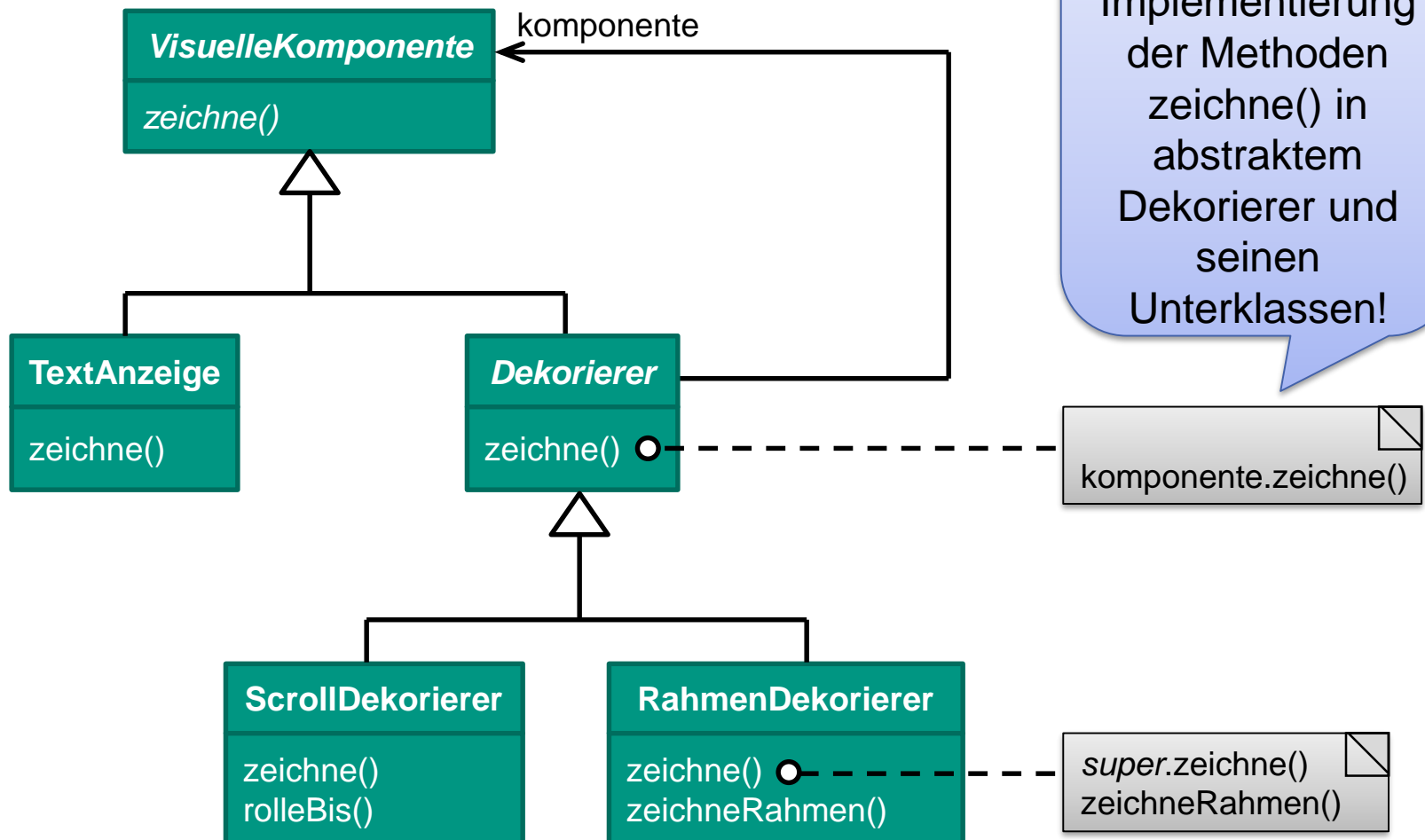
# Dekorierer: Struktur



# Dekorierer: Beispiel (1)



# Dekorierer: Beispiel (2)



# Dekorierer vs. Stellvertreter

## Dekorierer

- Fügt Objektfunktionalität hinzu, ohne Subjekt zu ändern
- Kann Subjektschnittstelle erweitern

## Stellvertreter

- Zugriffssteuerung
- Kann genauso wie das Subjekt verwendet werden
- Kann Latenz verstecken
- Kann Methoden verstecken
- Eigenes Objekt mit Subjekt “im Hintergrund”

**Nachlesen:** Head First Design Patterns, Seite 472 ff.

# Zustandshandhabungsmuster

IPD Tichy, Fakultät für Informatik



# Zustandshandhabungsmuster

- Einzelstück
- Fliegengewicht
- Memento
- Prototyp

# Einzelstück (engl. *singleton*)

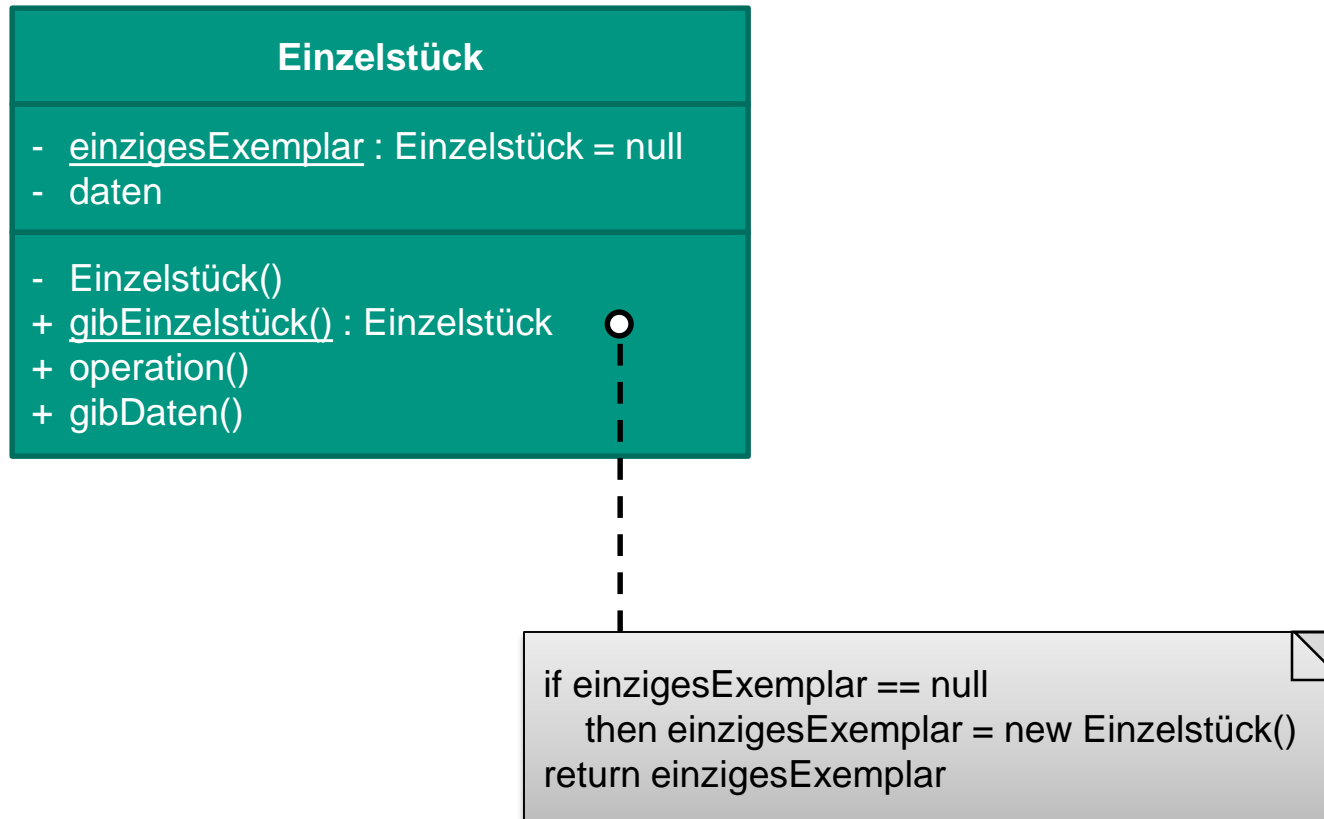
## ■ Zweck

Sichere zu, dass eine Klasse **genau ein Exemplar** besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.

## ■ Motivation

Die Klasse ist selbst für die Verwaltung ihres einzigen Exemplars zuständig. Die Klasse kann durch Abfangen von Befehlen zur Erzeugung neuer Objekte sicherstellen, dass kein weiteres Exemplar erzeugt wird.

# Einzelstück: Struktur





## Einzelstück: Anwendbarkeit

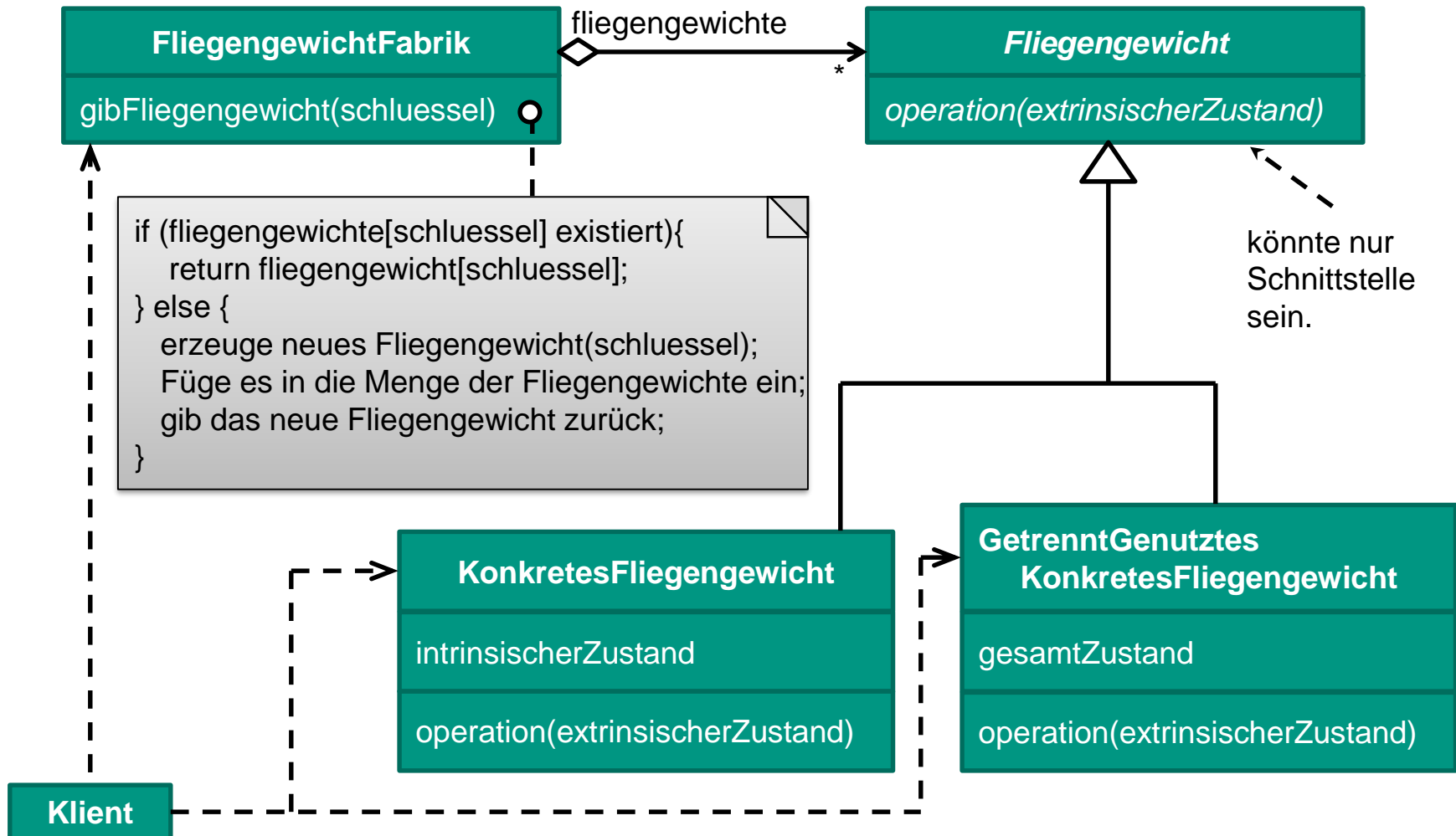
- Wenn es von einer Klasse **nur eine einzige Instanz** geben darf und diese Instanz den Klienten an einer bekannten Stelle zugänglich gemacht werden soll.
- Wenn es schwierig oder unmöglich ist, festzustellen, welcher Teil der Anwendung die erste Instanz erzeugt.
- Wenn die einzige Instanz durch Unterklassenbildung erweiterbar sein soll und die Klienten ohne Veränderung ihres Quelltextes diese nutzen sollen.

# Fliegengewicht (engl. *flyweight*)

## ■ Zweck

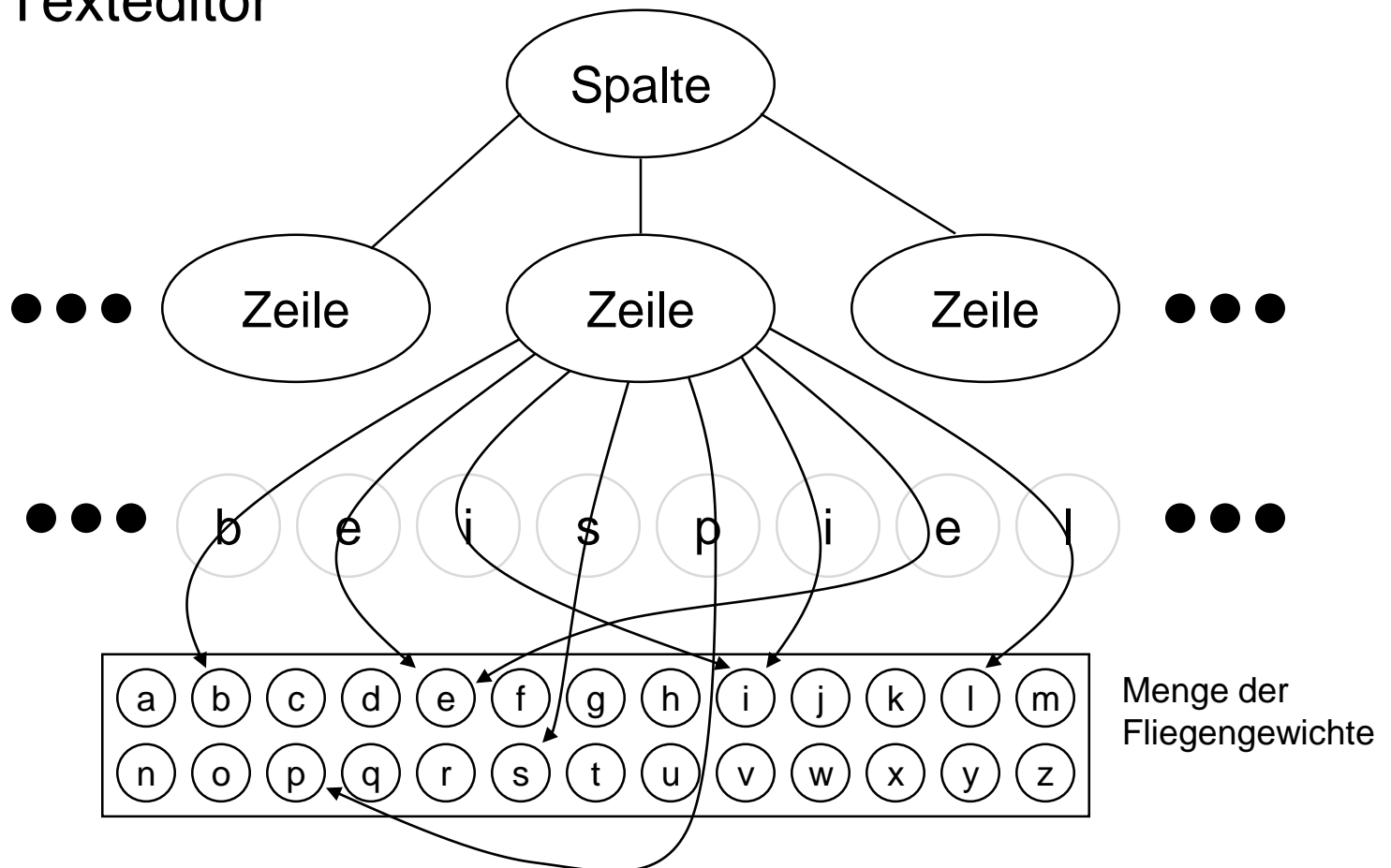
- Nutze Objekte kleinster Granularität **gemeinsam**, um große Mengen von ihnen **effizient speichern** zu können.

# Fliegengewicht: Struktur



# Fliegengewicht: Beispiel (1)

- Objektmodellierung bis hinunter zu einzelnen Zeichen in einem Texteditor



## Fliegengewicht: Beispiel (2)

- Die einzelnen Zeichen können durch einen Code repräsentiert werden (innerer, „intrinsischer“ Zustand).
- Die Informationen über Schriftart, Größe und Position können externalisiert werden (äußerer, „extrinsischer“ Zustand) und in dem Zeilen- oder Spaltenobjekt oder auch in Teilfolgen von Zeichen gespeichert werden.

# Fliegengewicht: Anwendbarkeit

- Wenn die Anwendung eine **große Menge von Objekten** verwendet, und
- wenn **Speicherkosten** allein aufgrund der Anzahl von Objekten hoch sind, und
- wenn ein Großteil des Objektzustands in den Kontext verlagert und damit extrinsisch gemacht werden kann, und
- viele Gruppen von Objekten durch relativ wenige gemeinsam genutzte Objekte ersetzt werden, sobald einmal der extrinsische Zustand entfernt wurde, und
- die Anwendung nicht von der Identität der Objekte abhängt (hier: Identität trotz möglicherweise konzeptuellem Unterschied).

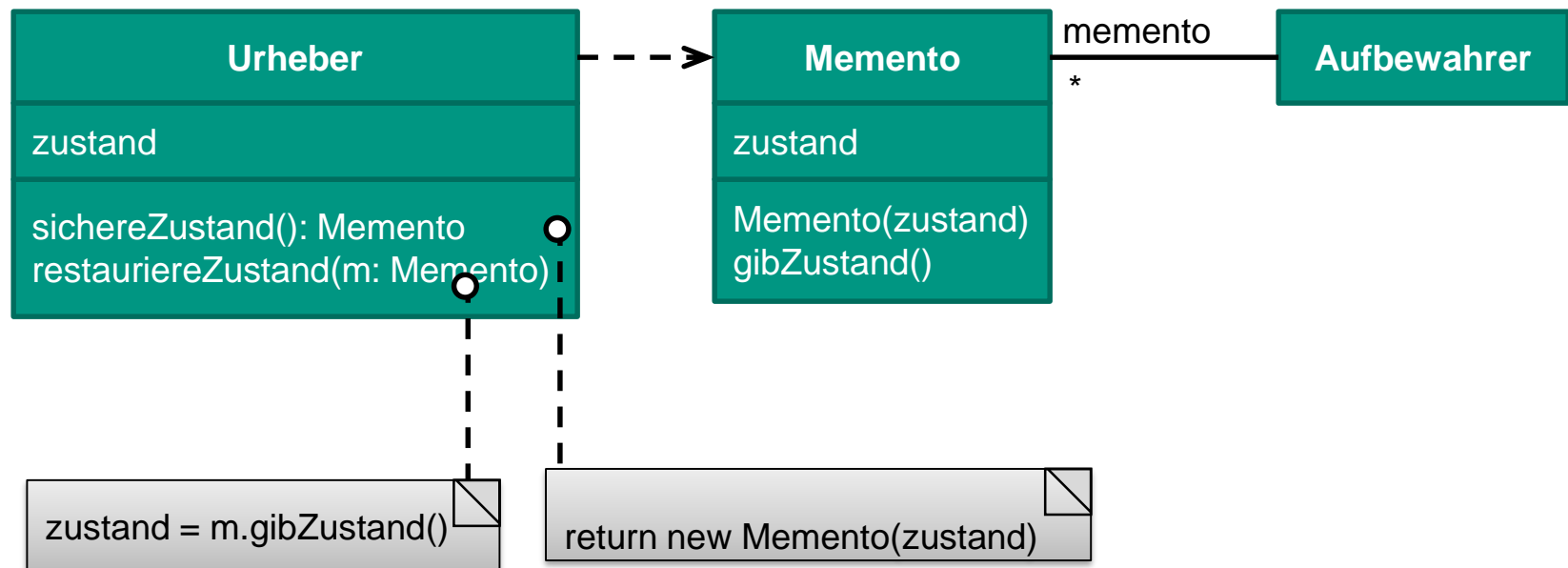
# Memento (engl. *memento*)

## ■ Zweck

- Erfasse und externalisiere den internen Zustand eines Objekts, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.

## ■ Synonyme: Token

# Memento: Struktur



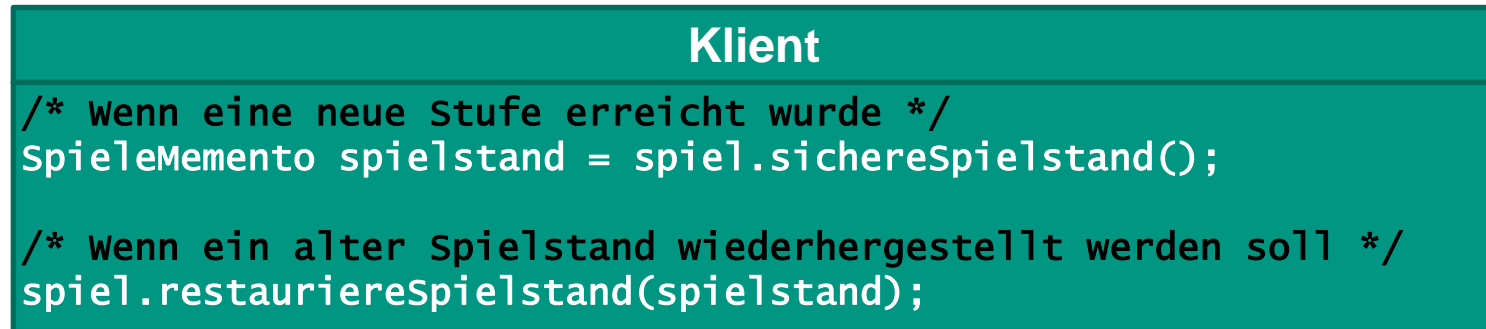


# Memento: Motivierendes Beispiel (1)

## ■ Szenario:

Dein interaktives Rollenspiel ist extrem erfolgreich und alle Spieler versuchen, die sagenumwobene Stufe 13 zu erreichen. Je näher die Spieler Stufe 13 kommen, desto mehr von ihnen scheitern am hohen Schwierigkeitsgrad der Stufen und müssen von vorne beginnen. Selbst Spieler der ersten Stunde sind überfordert. Die Spieler flehen sie an, endlich die lang angekündigte Funktion zum Speichern und Laden des Spielstandes einzubauen.

# Memento: Motivierendes Beispiel (2)



spielstand \*

## SpielMemento

```

spielstand: int

SpielMemento(int spielstand) {
    this.spielstand = spielstand;
}

int gibspielstand () {
    return this.spielstand;
}

```

## Spiel

```

spielstand: int

SpielMemento sichereSpielstand () {
    return new SpielMemento(spielstand);
}

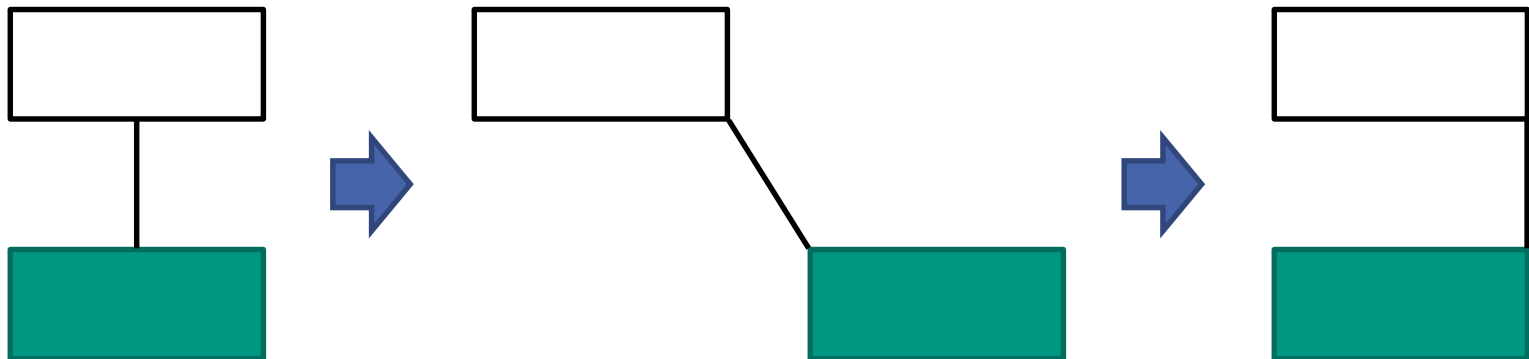
void restauriereSpielstand(
    SpielMemento s) {
    /* Spielstand wiederherstellen*/
    spielstand = s.gibSpielstand();
}

```

**Achtung:** Variablen mit Referenz-  
typen müssen kopiert werden!

## Memento: Weiteres Beispiel

- Komplexe Haltepunkte und Undo-Mechanismen wie z.B. in einem grafischen Editor



Rechtecke bleiben verbunden, wenn ein Rechteck bewegt wird.

Mögliches falsches Ergebnis nach einem Undo, falls nur Entfernung zum Ursprung des Rechtecks gespeichert wurde.

## Memento: Anwendbarkeit

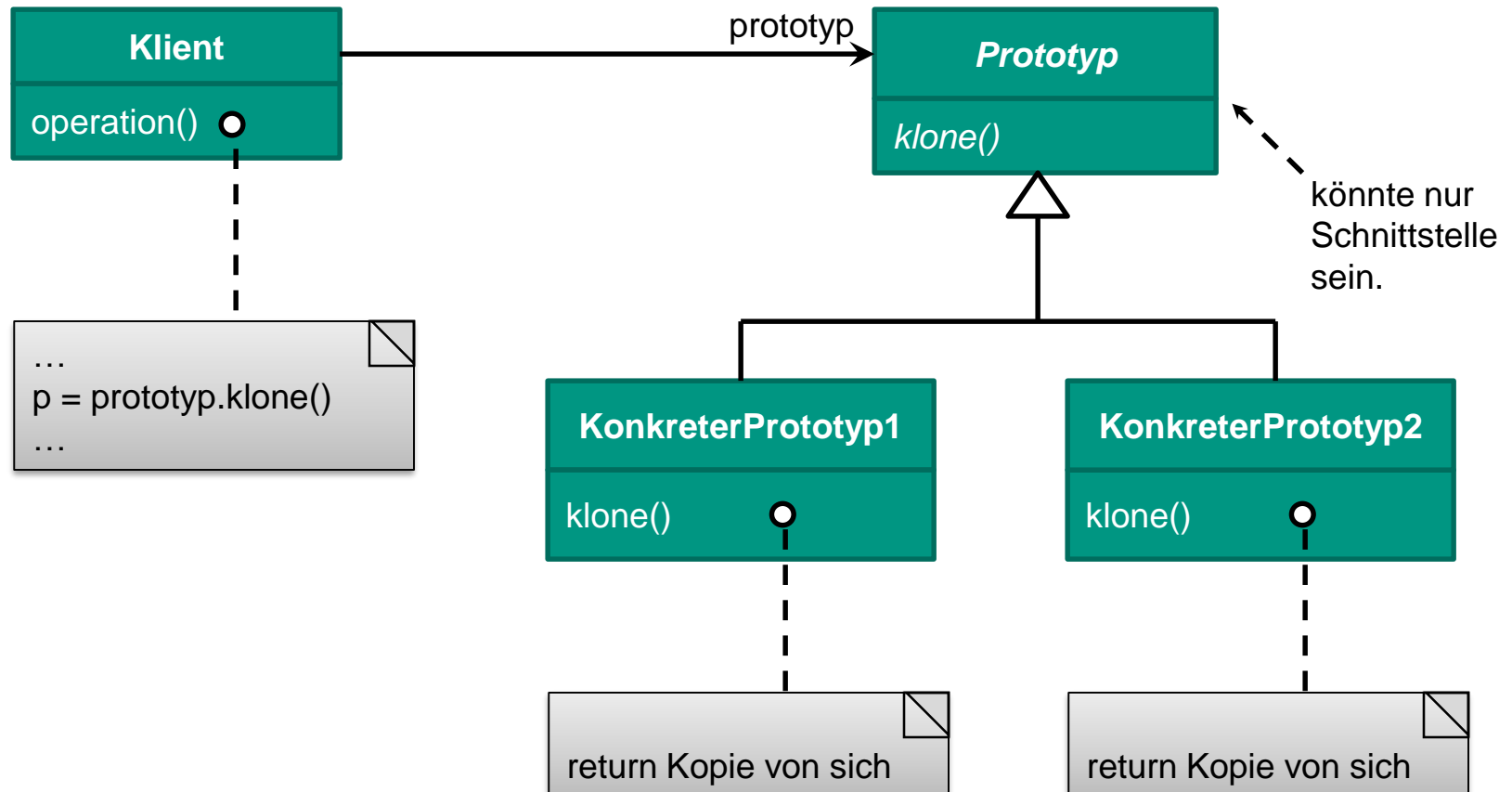
- Wenn eine **Momentaufnahme** (eines Teils) des Zustands eines Objekts **zwischengespeichert** werden muss, so dass es zu einem späteren Zeitpunkt in diesen Zustand zurückversetzt werden kann, und
- wenn eine **direkte Schnittstelle** zum Ermitteln des Zustands die Implementierungsdetails offenlegen und die Kapselung des Objekts aufbrechen würde.

# Prototyp (engl. *prototype*)

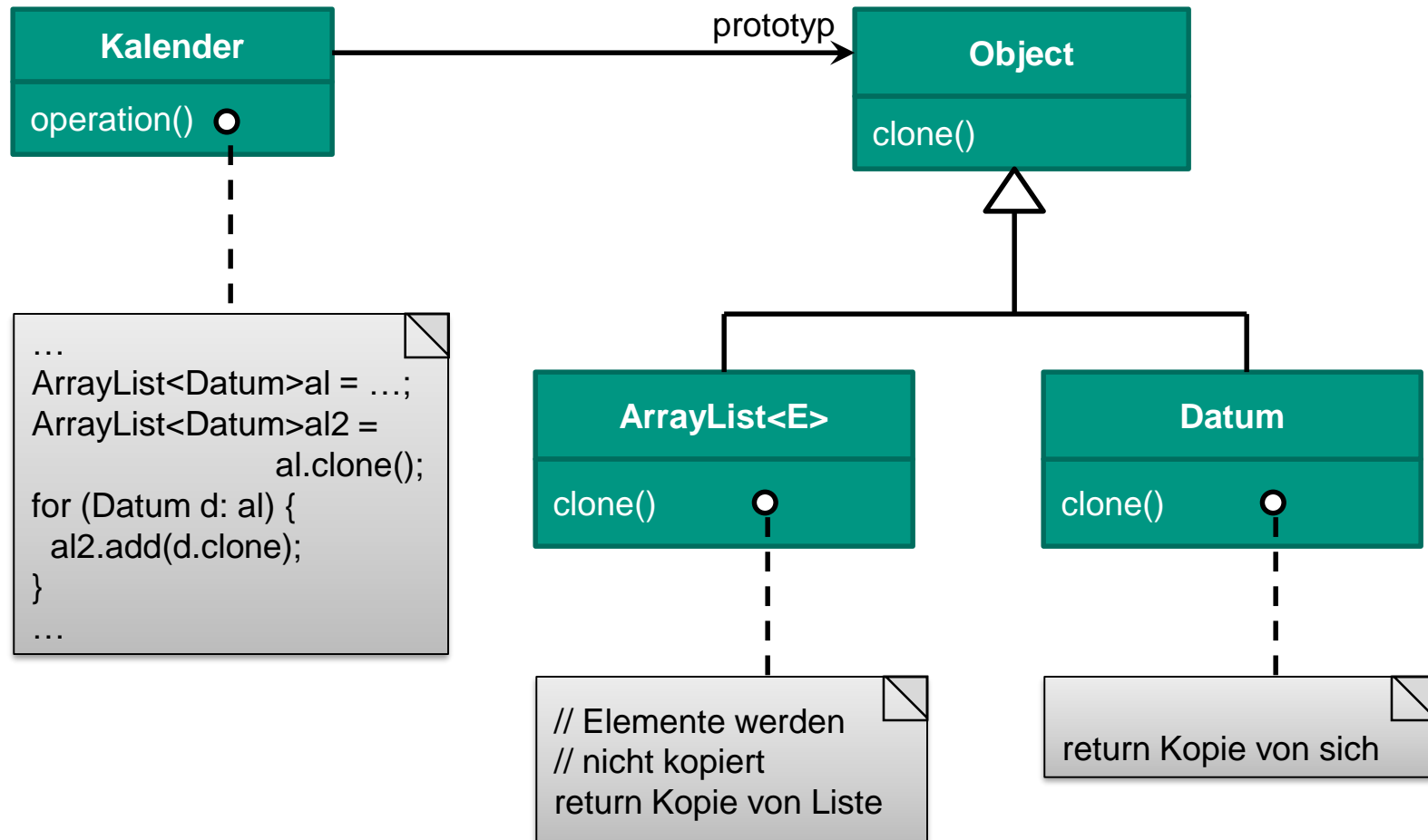
## ■ Zweck

- Bestimme die **Arten zu erzeugender Objekte** durch die Verwendung eines typischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototyps.

# Prototyp: Struktur



# Prototyp: Beispiel in Java (1)



## Prototyp: Beispiel in Java (2)

- Die Methode `clone()` der Klasse `Object` erstellt in Java eine **seichte** Kopie (engl. shallow copy) des Objektes.
- Bei einer seichten Kopie werden alle Attribute kopiert, einschließlich der Referenzen auf andere Objekte. Die referenzierten Objekte selbst werden **nicht** kopiert. (Die Gleichheit welcher Stufe haben wir damit?)
- Im Fall der `ArrayList` müssen daher alle Elemente „von Hand“ in die neue Liste kopiert werden.



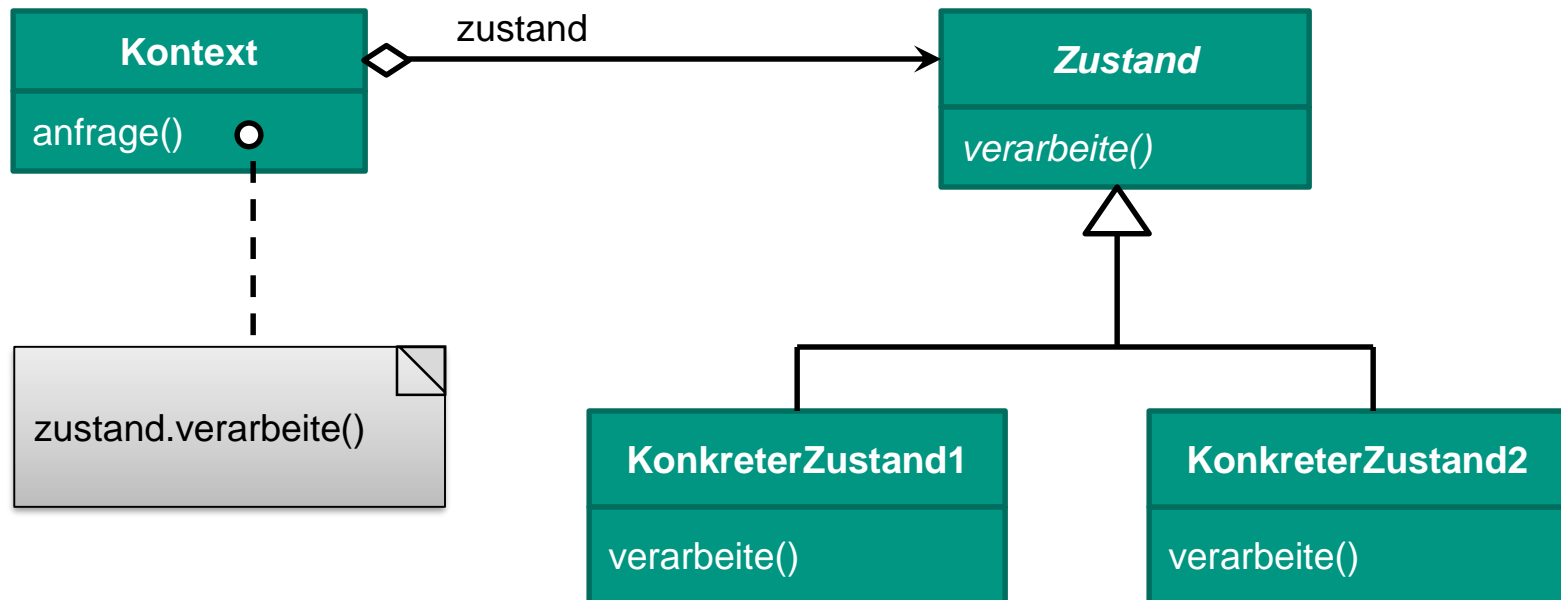
# Prototyp: Anwendbarkeit

- Das Prototypmuster wird verwendet, wenn ein System **unabhängig** davon sein soll, **wie** seine Produkte erzeugt, zusammengesetzt und repräsentiert werden, und
- falls der **Aufbau** eines Objekts wesentlich **mehr Zeit** erfordert als eine **Kopie** anzulegen, oder
- wenn die Klassen zu erzeugender Objekte erst zur Laufzeit spezifiziert werden, z.B. durch dynamisches Laden, oder
- um eine **Klassenhierarchie** von Fabriken zu **vermeiden**, die parallel zur Klassenhierarchie der Produkte verläuft, oder
- wenn Exemplare einer Klasse nur wenige Zustandskombinationen haben können. Es ist möglicherweise bequemer, eine entsprechende Anzahl von Prototypen einzurichten und sie zu klonieren statt die Objekte einer Klasse jedes mal von Hand mit dem richtigen Zustand zu erzeugen.

# Zustand (engl. *state*)

- Zweck
  - Ändere das Verhalten des Objektes, wenn sich dessen interner Zustand ändert.
  
- Auf das Zustandsmuster wird in Kapitel 4.1.2 genauer eingegangen.

# Zustand: Struktur



## Zustand: Anwendbarkeit

- Das Zustandsmuster wird verwendet, wenn das Verhalten des Objektes von dessen Zustand abhängt und das Objekt sein Verhalten während der Laufzeit, abhängig vom aktuellen Zustand, ändern muss.

# Steuerungsmuster

IPD Tichy, Fakultät für Informatik



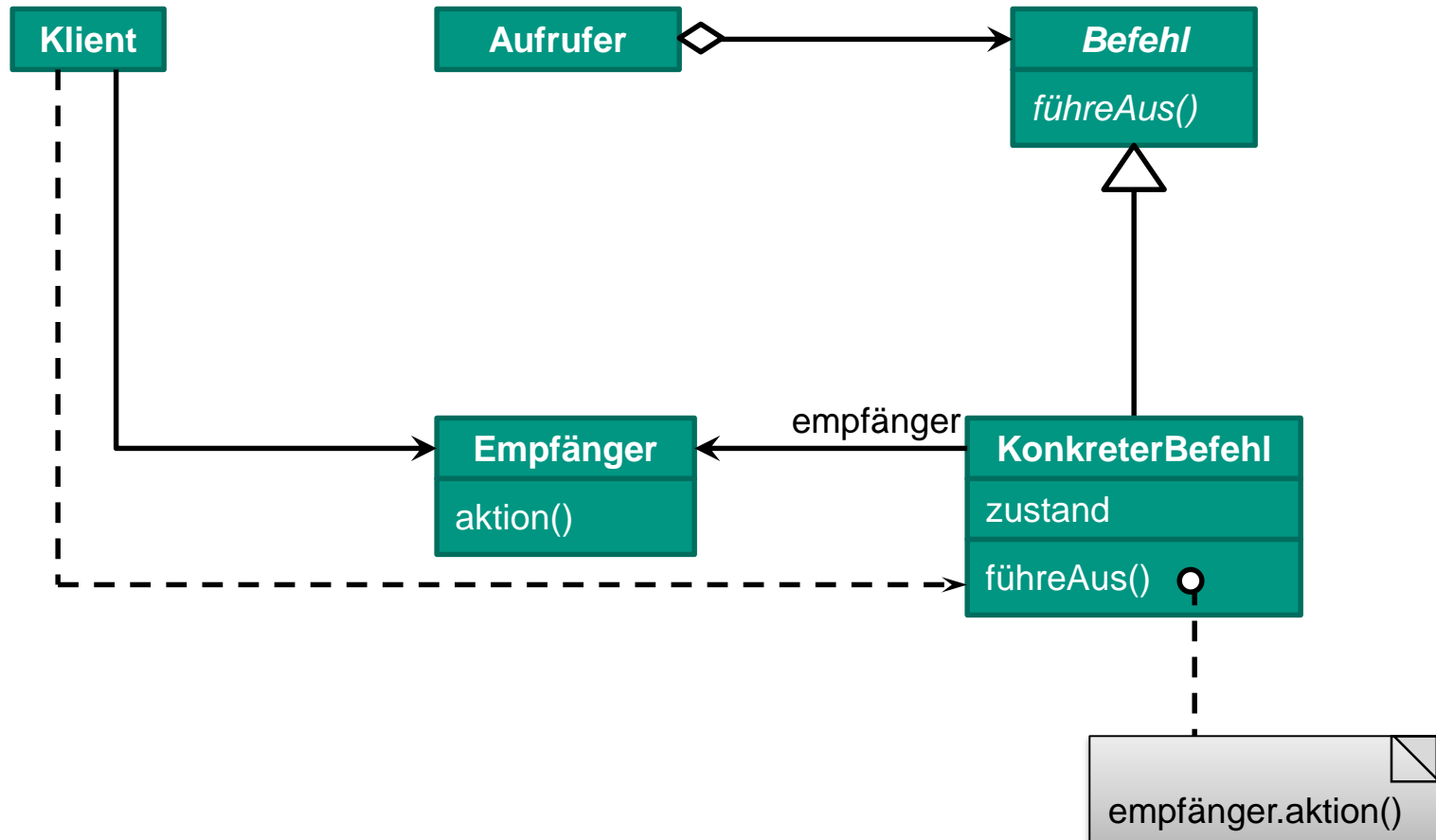
# Steuerungsmuster

- Befehl (*command*)
- Auftraggeber/-nehmer (*master/worker*)

# Befehl (engl. *command*)

- Zweck
  - Kapsle einen **Befehl als** ein **Objekt**. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Warteschlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.
- Synonyme: Kommando, Aktion, Transaktion

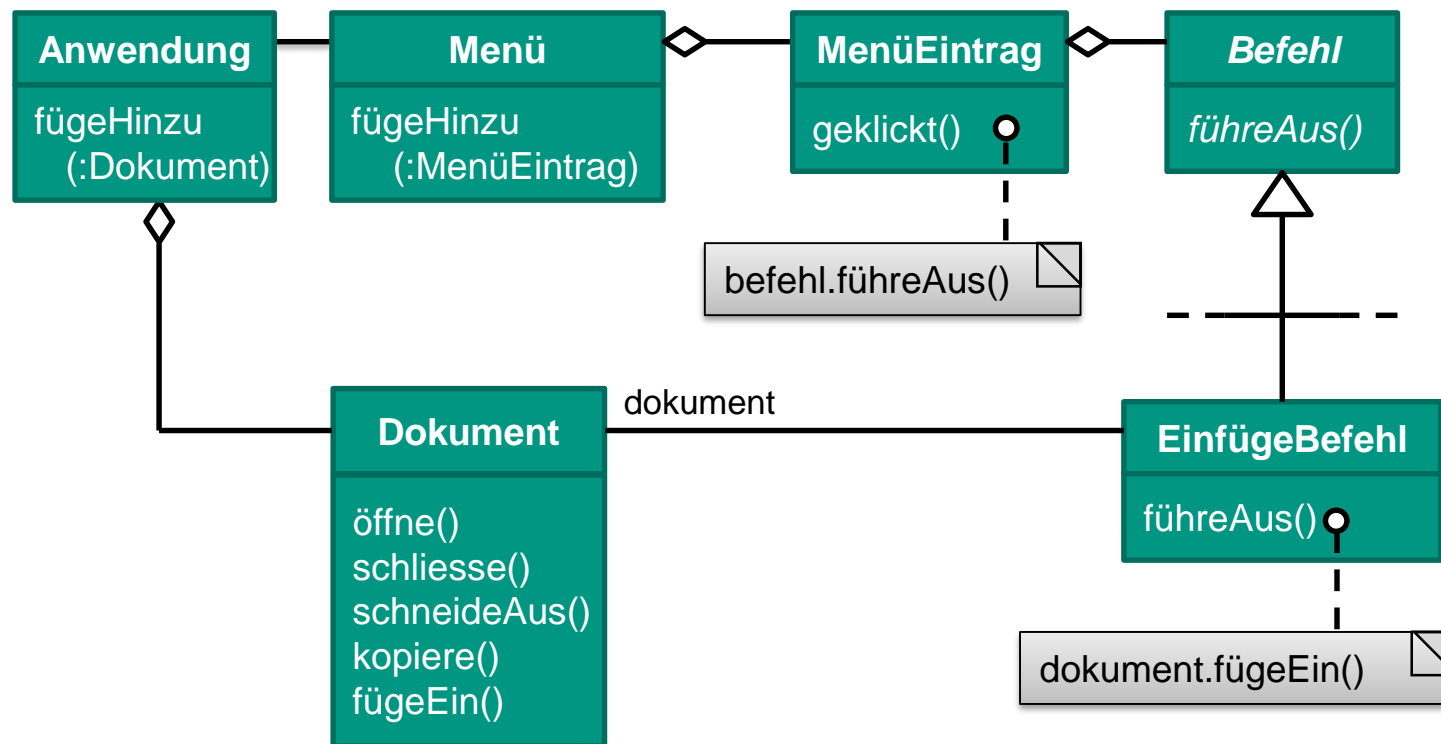
# Befehl: Struktur





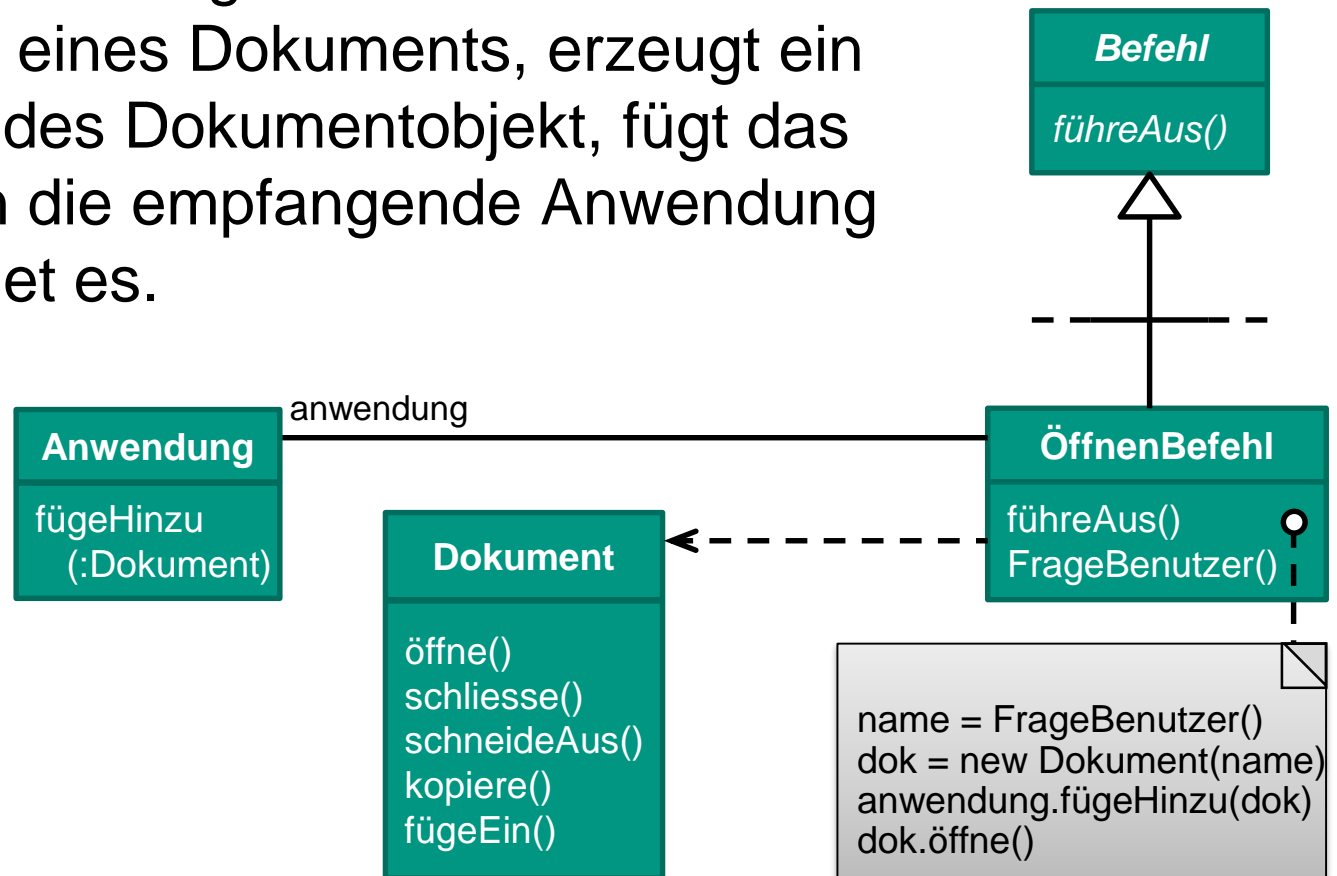
# Befehl: Beispiel (1)

- Ein Befehl enthält eine Methode `führeAus()` und speichert das Objekt, an dem diese Operation durchgeführt werden soll.



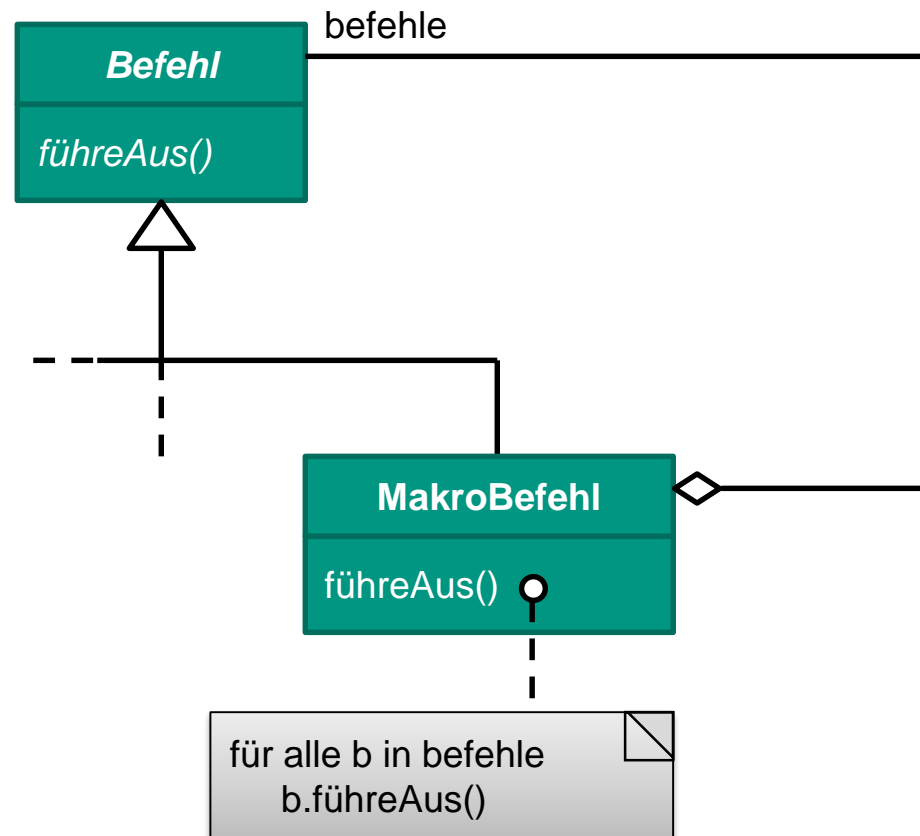
## Befehl: Beispiel (2)

- Die `führeAus`-Operation der Klasse `ÖffnenBefehl` fragt den Benutzer nach dem Namen eines Dokuments, erzeugt ein entsprechendes Dokumentobjekt, fügt das Dokument in die empfangende Anwendung ein, und öffnet es.



# Befehl: Abfolge von Befehlen

- siehe Kompositum



## Befehl: Anwendbarkeit

- Wenn Objekte mit einer auszuführenden Aktion **parametrisiert** werden sollen (wie bei den MenüEintrag-Objekten).
- Wenn Anfragen zu unterschiedlichen Zeiten spezifiziert, aufgereiht und ausgeführt werden sollen.
- Wenn ein **Rückgängigmachen** von Operation (Undo) unterstützt werden soll.
- Wenn das **Mitprotokollieren** von Änderungen unterstützt werden soll (um System nach Absturz wiederherzustellen).
- Wenn ein System mittels komplexer Operationen strukturiert werden soll, die aus primitiven Operationen aufgebaut werden (Makrobefehle).

# Auftraggeber/-nehmer (engl. *master/worker*)

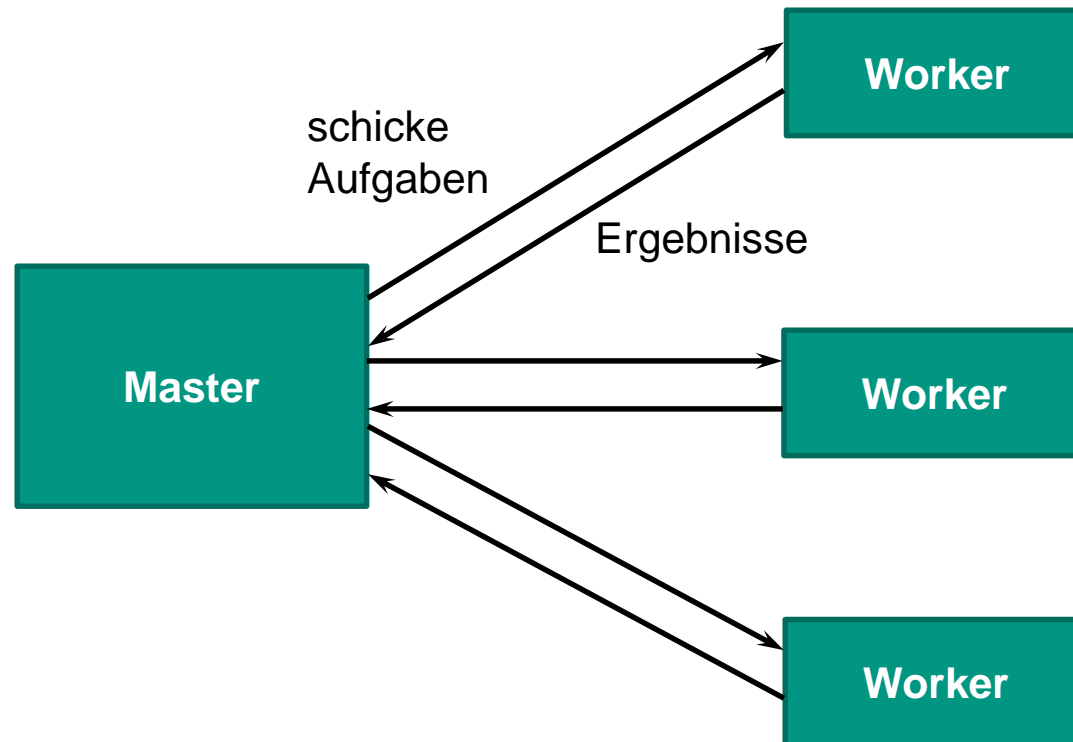
## ■ Zweck

- Auftraggeber/-nehmer bietet **fehlertolerante** und **parallele Berechnung**. Ein Auftraggeber verteilt die Arbeit an identische Arbeiter (Auftragnehmer) und berechnet das Endergebnis aus den Teilergebnissen, welche die Arbeiter zurückliefern.

## ■ Synonyme: Master/Slave

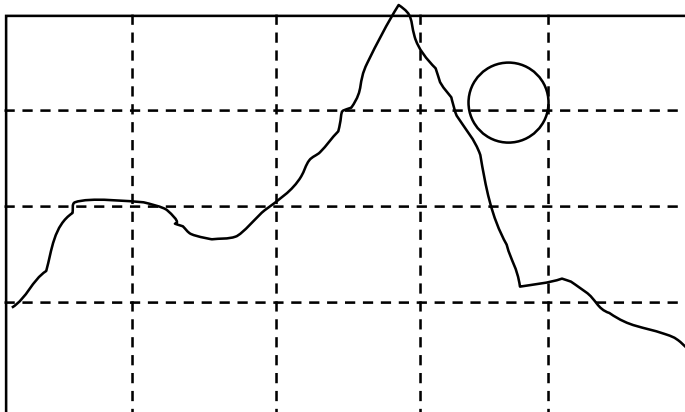
# Master/Worker: Struktur

- Master und alle Arbeiter laufen in eigenen Prozessen parallel.



# Master/Worker: Beispiel

## ■ Parallele Berechnung eines 3D Bildes



Der Master gibt rechteckige Teile des Bildes zur Berechnung an seine Arbeiter weiter und setzt das gesamte Bild aus den einzelnen Teilen zusammen.

Weitere Beispiele: Seti@home, Folding@home

## Master/Worker: Anwendbarkeit

- Wenn es **mehrere Aufgaben** gibt, die **unabhängig** voneinander bearbeitet werden können.
- Wenn mehrere **Prozessoren** zur parallelen Verarbeitung zur Verfügung stehen.
- Wenn die Belastung der Arbeiter **ausgeglichen** werden soll.



# Virtuelle Maschinen

IPD Tichy, Fakultät für Informatik



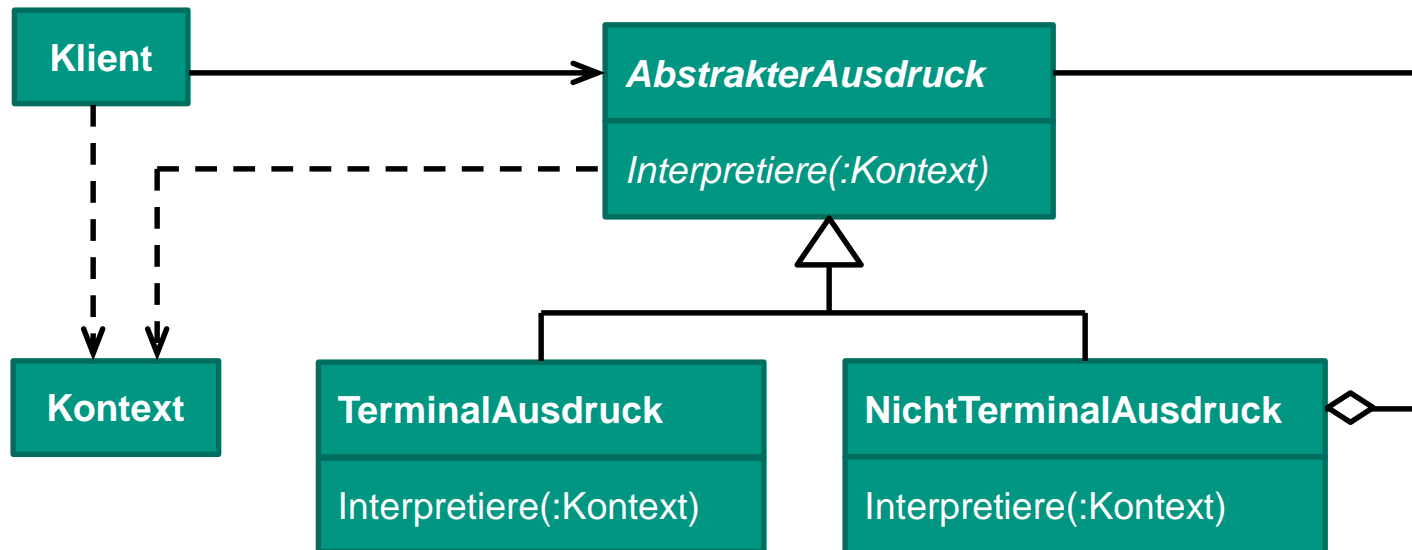
# Interpretierer (engl. interpreter)

Nicht prüfungsrelevant

## ■ Zweck

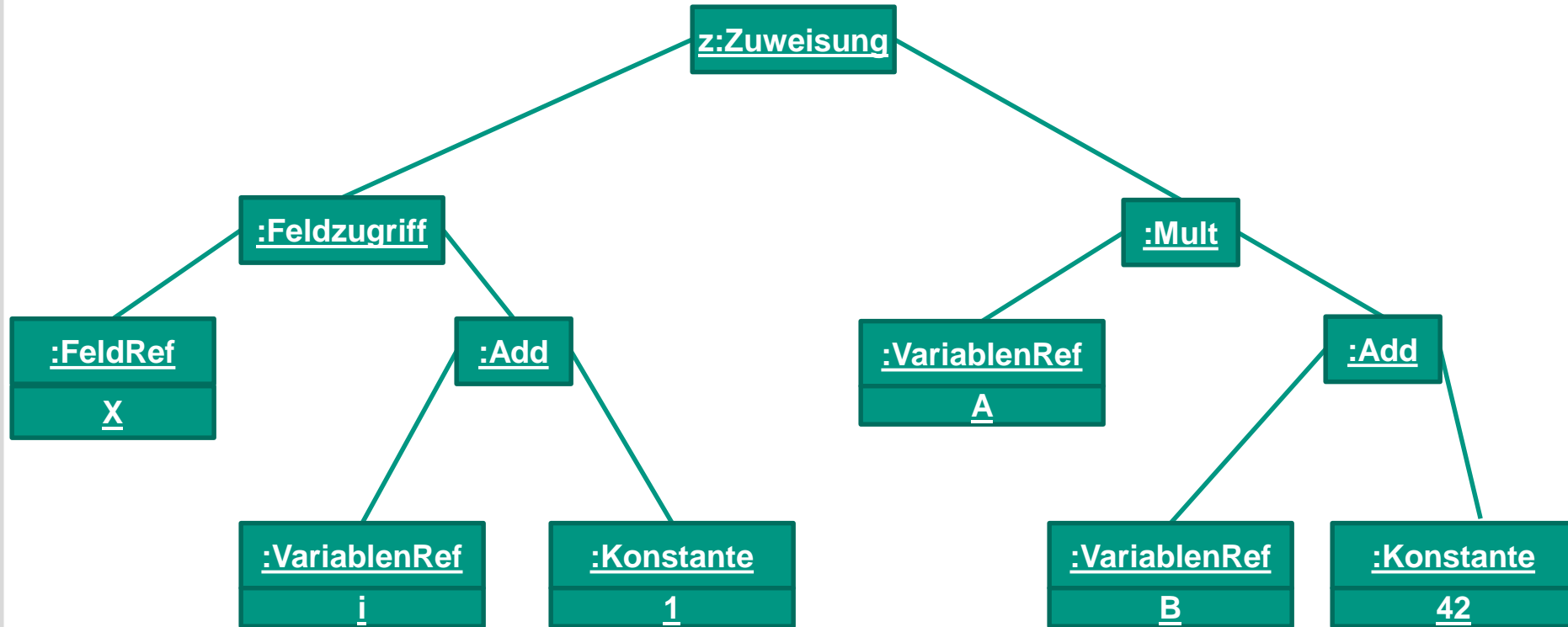
- Definiere für eine gegebene Sprache eine **Repräsentation** der **Grammatik** sowie einen Interpretierer, der die Repräsentation nutzt, um Sätze in der Sprache zu interpretieren.

# Interpretierer: Struktur Nicht prüfungsrelevant



# Beispiel für einen abstrakten Syntaxbaum

Nicht prüfungsrelevant



- Was macht `z.interpretiere(symbolTabelle)`?

# Interpreterer: Anwendbarkeit Nicht prüfungsrelevant

- Wenn eine Sprache interpretiert werden muss und Ausdrücke der Sprache als abstrakte Syntaxbäume darstellbar sind. Das Interpreterermuster funktioniert am besten, wenn
  - die **Grammatik einfach** ist. Bei komplexen Grammatiken wird die Klassenhierarchie zu groß und nicht mehr handhabbar. In diesem Falle stellen Werkzeuge wie Parsergeneratoren eine bessere Alternative dar.
  - die **Effizienz unproblematisch** ist. Effiziente Interpreterer werden üblicherweise nicht durch Interpretation von Syntaxbäumen implementiert; sie transformieren die Bäume stattdessen in eine andere Form, z.B. Zwischencode.

# Interpreter vs. Kompositum vs. Besucher

Nicht prüfungsrelevant

- Der Interpreter und das Kompositum haben die **gleiche Struktur**. Man spricht von einem Interpreter, wenn Sätze einer Sprache repräsentiert und ausgewertet werden. Der Interpreter kann als **Spezialfall** des Kompositums gesehen werden.
- Ein Besucher kann dazu verwendet werden, das Verhalten eines jeden Knotens im abstrakten Syntaxbaum in einer einzigen Klasse zu kapseln.



# Bequemlichkeitsmuster

IPD Tichy, Fakultät für Informatik



# Bequemlichkeitsmuster (engl. *convenience patterns*)

- Bequemlichkeits-Klasse
- Bequemlichkeits-Methode
- Fassade
- Null-Objekt



# Bequemlichkeits-Klasse (engl. *convenience class*)

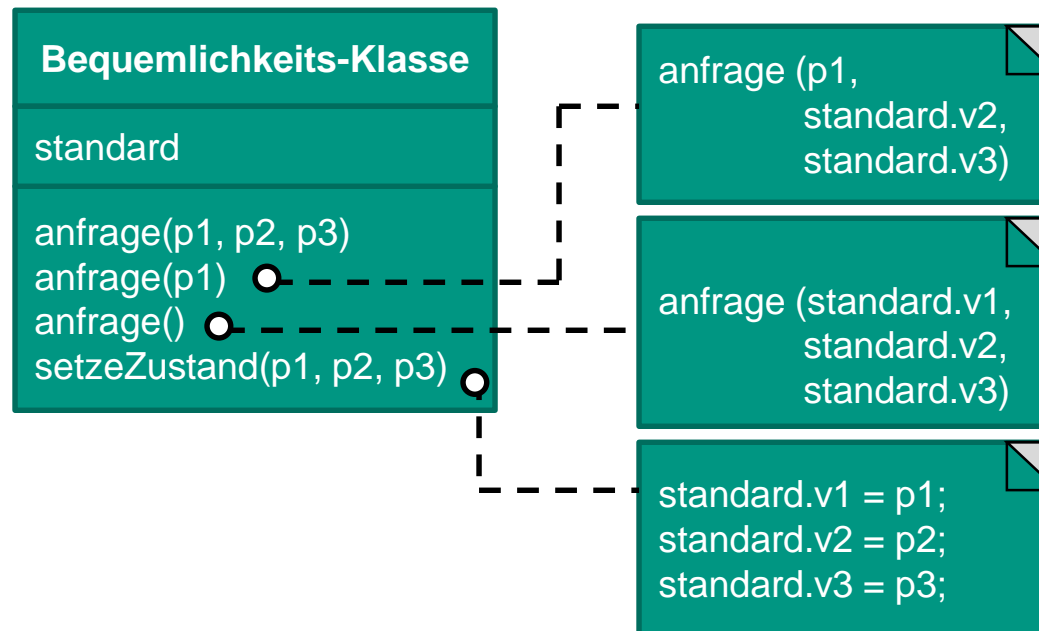
## ■ Zweck

- Vereinfache den **Methodenaufruf** durch Bereithaltung der Parameter in einer **speziellen Klasse**.

## ■ Anwendbarkeit

- Wenn Methoden häufig mit den gleichen Parametern aufgerufen werden, die sich nur selten ändern.

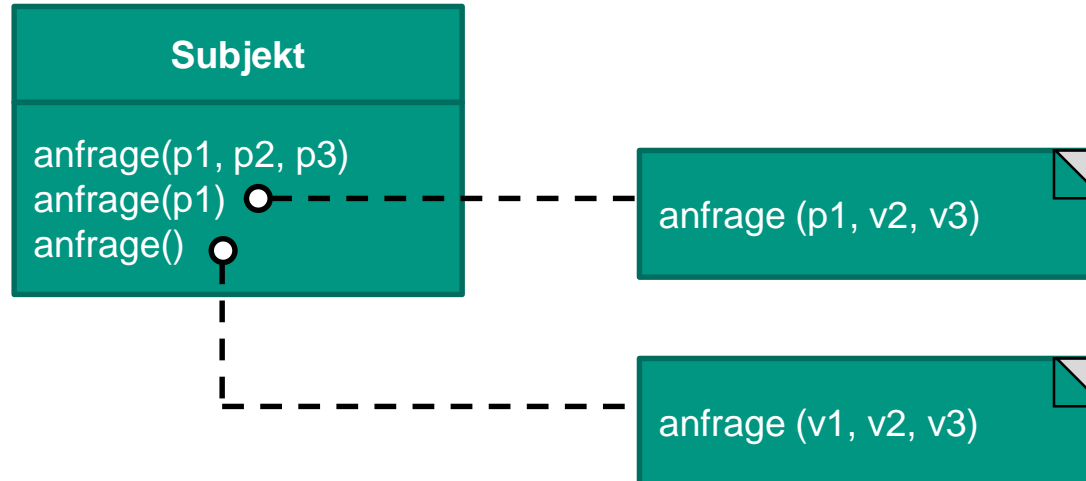
# Bequemlichkeits-Klasse: Struktur



# Bequemlichkeits-Methode (engl. *convenience method*)

- Zweck
  - Vereinfachen des **Methodenaufrufs** durch die Bereitstellung häufig genutzter Parameterkombinationen in **zusätzlichen Methoden** (Überladen).
- Synonyme: vorbelegte Parameter, default parameters
- Anwendbarkeit
  - Wenn Methodenaufrufe häufig mit den gleichen Parametern auftreten.

# Bequemlichkeits-Methode: Struktur



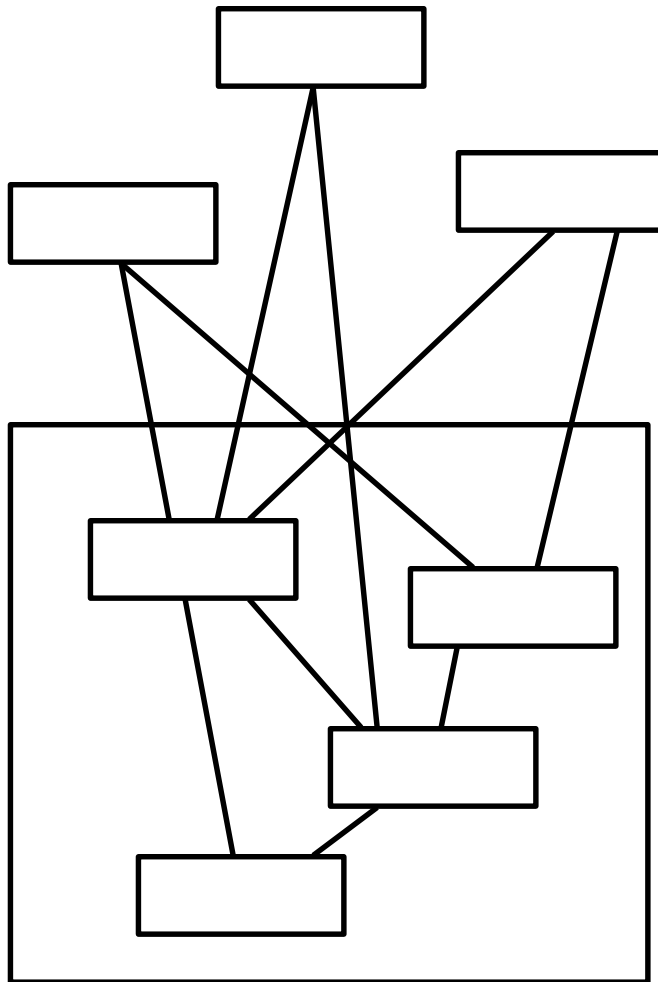
# Fassade (engl. *facade*/*façade*)

## ■ Zweck

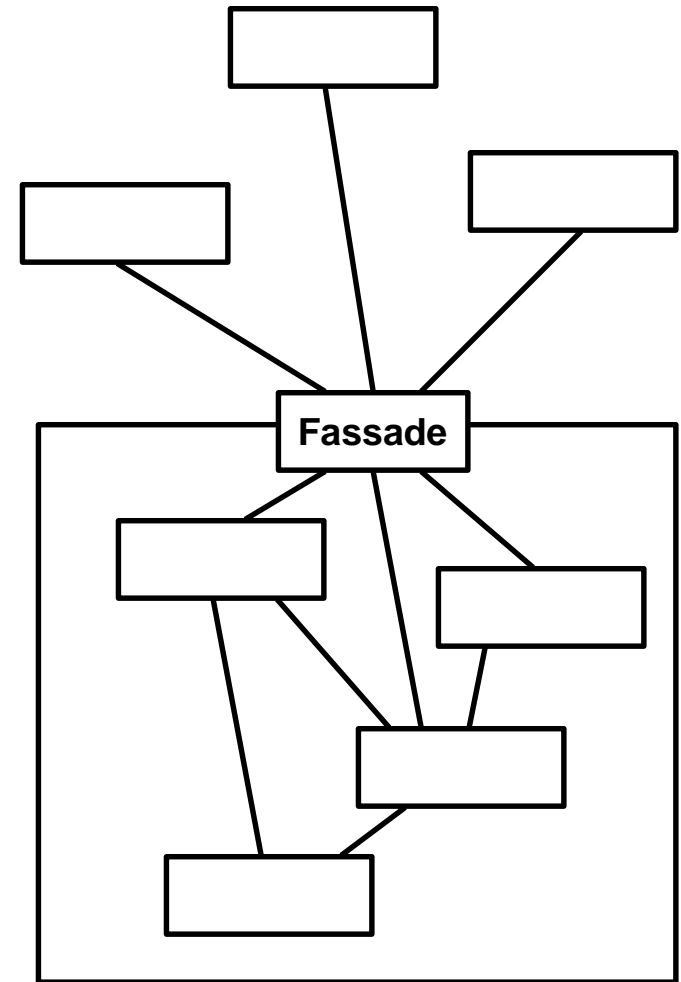
- Biete eine **einheitliche Schnittstelle** zu einer Menge von Schnittstellen eines Subsystems.
- Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.

# Fassade: Beispiel

Klientenklassen



Subsystemklassen



# Fassade: Anwendungsbeispiel

## Das 1-Click-Bestellsystem von Amazon.de



Menge: 1 ▼

 In den Einkaufswagen  
oder jetzt kaufen per

 1-Click® Premiumversand GRATIS

Versenden an:  
David Meder ▼

☐ Geschenkpapier /  
Grußbotschaft hinzufügen

- Normalerweise muss der Kunde bei jedem Bestellvorgang seine Daten (Liefer-/Rechnungsadresse, Bankverbindung, ...) angeben.
- Hat der Kunde seine Daten bereits bei Amazon hinterlegt und das 1-Click System aktiviert, kann der Kunde den gewählten Artikel mit einem Klick bestellen.

# Fassade: Anwendbarkeit

- Wenn eine **einfache Schnittstelle** zu einem **komplexen Subsystem** angeboten werden soll. Eine Fassade kann eine einfache voreingestellte Sicht auf das Subsystem bieten, die den meisten Klienten genügt.
- Wenn es **viele Abhängigkeiten** zwischen den Klienten und den Implementierungsklassen einer Abstraktion gibt. Die Einführung einer Fassade entkoppelt die Subsysteme von Klienten und anderen Subsystemen.
- Wenn Subsysteme in Schichten aufgeteilt werden sollen. Man verwendet eine Fassade, um einen Eintrittspunkt zu jeder Subsystemschicht zu definieren.



# Null-Objekt (engl. *null object*)

## ■ Zweck

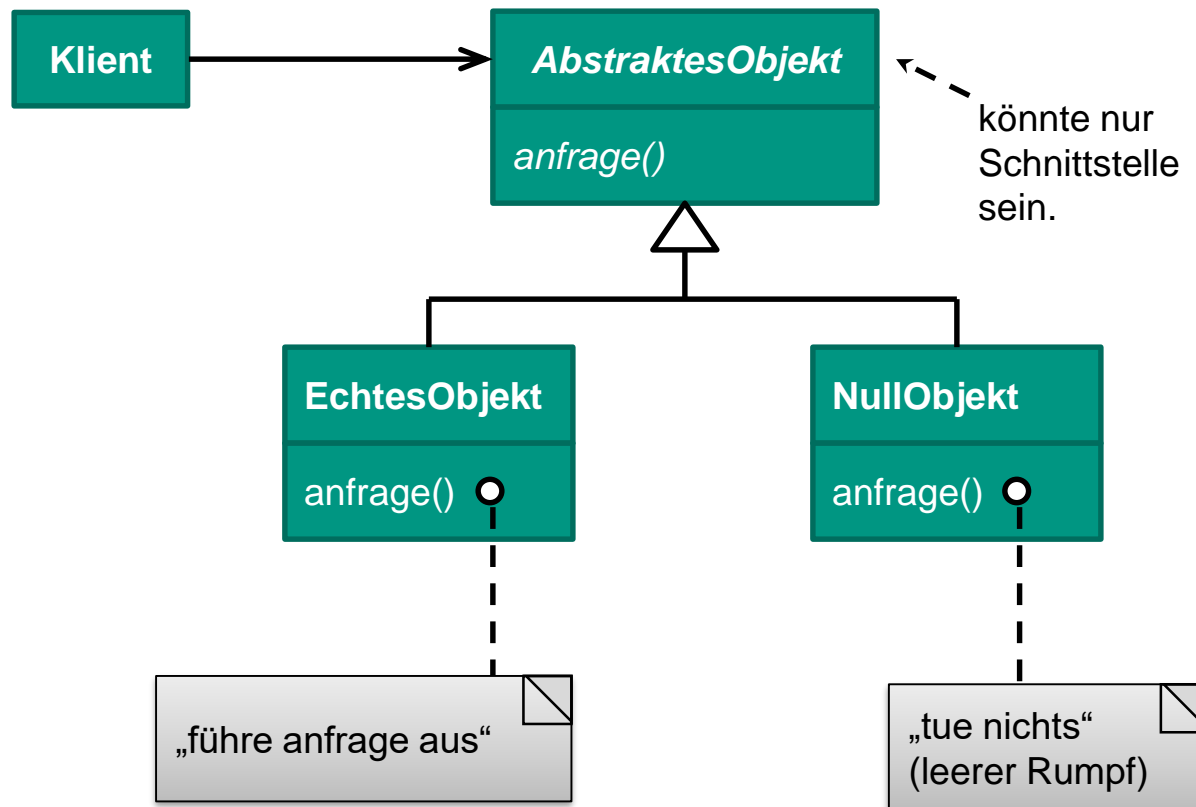
- Stelle einen **Stellvertreter** zur Verfügung, der die gleiche Schnittstelle bietet, aber nichts tut. Das Null-Objekt kapselt die Implementierungs-Entscheidungen (wie genau es „nichts tut“) und versteckt diese Details vor seinen Mitarbeitern.

## ■ Motivation

- Es wird verhindert, dass der Code mit Tests gegen Null-Werte verschmutzt wird, wie:

```
if (thisCall.callingParty != null)
    thisCall.callingParty.action();
```

# Null-Objekt: Struktur



# Und wie geht das jetzt?

```
Class NullAction implements Action {  
    public void action(){ /* leer */ }  
}  
Class TrueAction implements Action {  
    public void action () {  
        /* echter Code */  
    }  
}  
...  
thisCall.callingParty = new TrueAction();  
...  
thisCall.callingParty = new NullAction();  
....  
thisCall.callingParty.action(); // kein Test auf null
```

# Null-Objekt: Anwendbarkeit

- Wenn ein Objekt Mitarbeiter benötigt und einer oder mehrere von ihnen nichts tun sollen.
- Wenn Klienten sich nicht um den Unterschied zwischen einem echten Mitarbeiter und einem der nichts tut kümmern sollen.
- Wenn das „tue nichts“-Verhalten von verschiedenen Klienten wiederverwendet werden soll.
- Beispiel in Swing: Adapterklassen (mit leeren Methodenrumpfen).

# Wo finde ich mehr über Entwurfsmuster?

- “Design Patterns”, Gamma et al, Addison Wesley, 1995.  
Deutsche Ausgabe: “Entwurfsmuster”, Riehle.
- „Head First Design Patterns“, Freeman&Freeman, O'Reilly, 2004 (ausgezeichnet!). Deutsche Ausgabe: „Entwurfsmuster von Kopf bis Fuß“.
- “Pattern Languages of Program Design”, conference proceedings, Addison-Wesley, 1995, 1996,...
- “Pattern Hatching”, John Vlissides, Addison-Wesley, 1998.  
Deutsch: “Entwurfsmuster anwenden”, 1999.
- “Patterns in Java”, Mark Grand, Wiley, 1998.
- “Pattern-Oriented Software Architecture”, Schmidt et al, Wiley, 2000.

# Zusammenfassung

## ■ Entwurfsmuster

- bieten ein Vokabular zur effizienten Kommunikation zwischen Teammitgliedern
  - erfassen den „Stand der Kunst“
  - dokumentieren Entwürfe
  - machen aus Anfängern gute Entwerfer
  - machen gute Entwerfer noch besser
  - verbessern Qualität und Produktivität
  - sind schwierig zu finden und zu beschreiben
- Mehrere kontrollierte Experimente haben bestätigt, dass Entwurfsmuster SW-Qualität, Programmier-Produktivität und Team-Kommunikation tatsächlich verbessern.