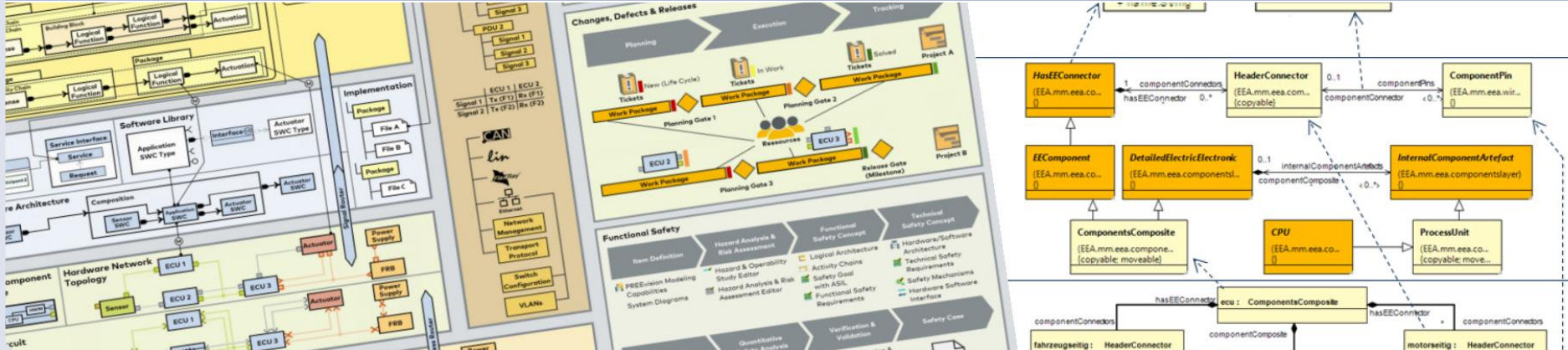


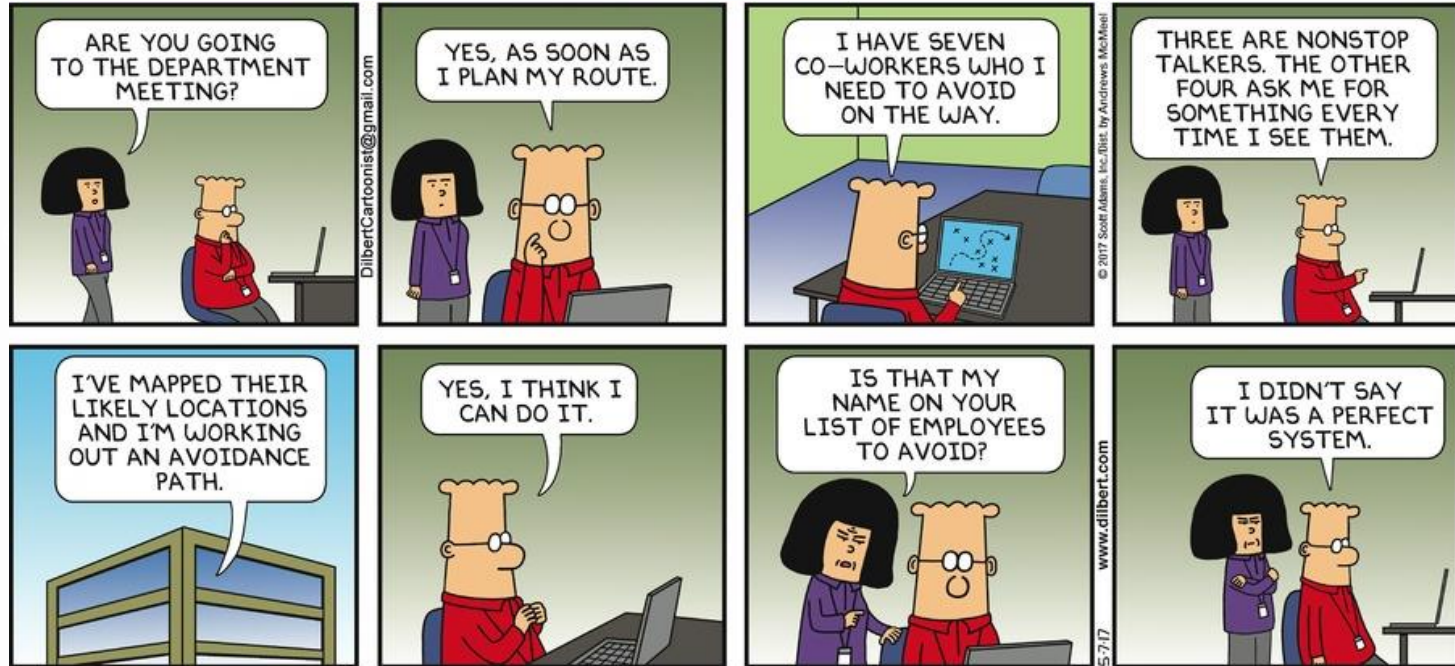
Vorlesung Software Engineering (SE)

Wintersemester 2017/2018

Kapitel 6 – Entwurfsmuster



6. Entwurfsmuster



6.1 Einführung und Begriffe

6.2 Vor- und Nachteile

6.3 Klassifizierung

6.3.1 Entkopplungs-Muster

6.3.2 Varianten-Muster

6.3.3 Zustandshandhabungs-Muster

6.3.4 Steuerungs-Muster

6.3.5 Virtuelle Maschinen

6.3.6 Bequemlichkeits-Muster

6.4 Beziehungen zwischen Entwurfsmustern

6.5 Zusammengesetzte Muster



Bücher

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
Entwurfsmuster, Addison-Wesley, 2. Auflage 2001, ISBN 3-8273-1862-9

B. Brügge, A.H. Dutoit:
Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java, Pearson 2004

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal: ***Patternorientierte Softwarearchitektur***, Addison-Wesley 1998

James W.Cooper: ***The Design Patterns***, Addison-Wesley 1998

Internet

SourceMaking: sourcemaking.com/design_patterns

Brad Appleton: bradapp.com/docs/patterns-nutshell.html

JavaWorld: javaworld.com/javaworld/jw-05-2003/jw-0530-designpatterns.html

Oracle: java.sun.com/blueprints/patterns



Die Cheops Pyramide

Erbaut: 2551 – 2528 v.Chr



Das Reichstagsgebäude

Erbaut: 1884 – 1894

Die Glaskuppel

Erbaut: 1995 – 1999



Der Schiefe Turm zu Pisa

Erbaut: 12. Jahrhundert

Der Untergrund erwies sich als
nicht tragfähig genug für den Turm.



Die Knickpyramide

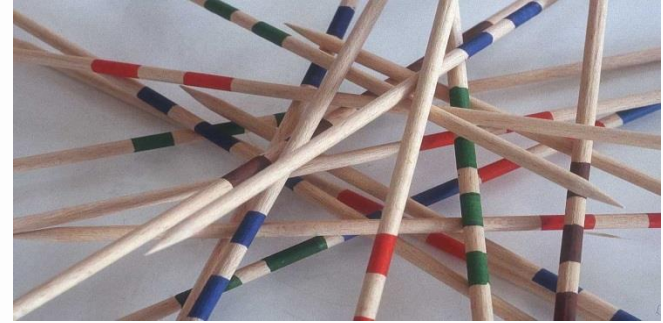
Erbaut: 2570 – 2480 v.Ch

Die Steigung der unteren Pyramide
erwies sich als **zu steil**.

Um ein Abrutschen zu verhindern, wurde
der obere Teil deutlich flacher gebaut.

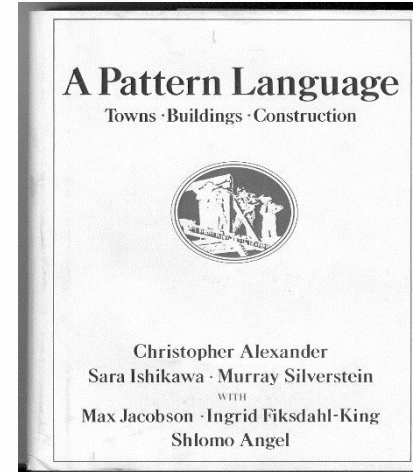


Ein *Software-Entwurfsmuster* beschreibt eine **Familie von Lösungen** für ein Software-Entwurfsproblem. Das Ziel eines Entwurfsmusters ist die **Wiederverwendbarkeit** von Entwurfs**wissen**.

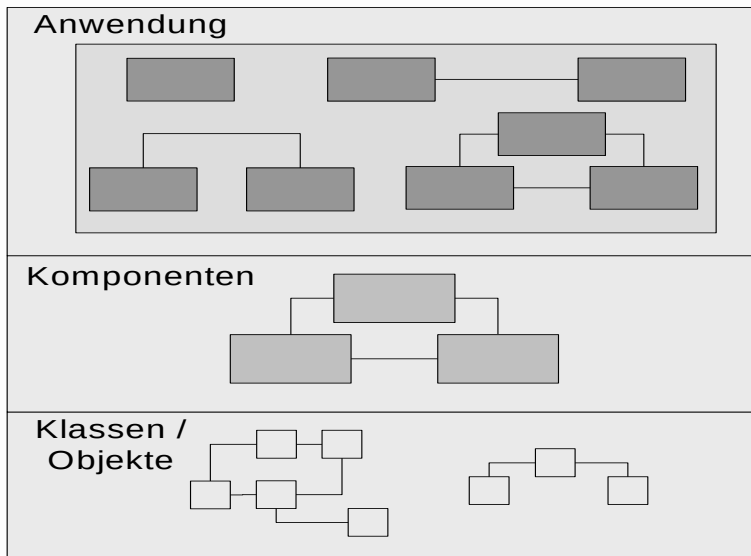


Entwurfsmuster sind für das Programmieren **im Großen** (also den Entwurf),
was **Algorithmen** für das Programmieren **im Kleinen** sind.

- » Ein *Entwurfsmuster* beschreibt ein Problem, das immer wieder in unserer Umwelt vorkommt; und beschreibt dann den Kern einer Lösung des Problems, sodass die Lösung für Probleme dieses Typs millionenfach verwendet werden kann, ohne zweimal genau dasselbe zu tun.
- frei nach C. Alexander, Gamma et. al. (S.2)



Entwurfsmuster sind **abstrakte Anweisungen**, wie Software implementiert werden kann, damit bestimmte **Eigenschaften** eines Entwurfs **erreicht** werden.
(Z.B. Änderbarkeit, Wartbarkeit, Wiederverwendbarkeit...)



Musterarchitekturen

**Architektur
Pattern**



Architekturmuster

J2EE-Patterns



Entwurfsmuster

GoF-Patterns



Entwurfsmuster

eine **Zwischenlösung** zwischen Frameworks und persönlicher Erfahrung

Persönliche Erfahrung

Schwierigster Weg:

- nicht dokumentiert, daher **schlecht nachvollziehbar**
- von Person zu Person unterschiedlich

Wichtigster Weg:

- Software-Entwurf ist **keine mechanische Tätigkeit**, sondern ein **kreatives Abwägen von Alternativen**, dass auf den Menschen angewiesen ist und sich (noch?) nicht automatisieren lässt.

Was sind Entwurfsmuster NICHT?

Keine Rezepte

Rezepte können **blind angewandt** werden.

Entwurfsmuster müssen erst **verstanden werden**, da sie wegen der breiten Anwendbarkeit sehr **abstrakt** sind.

Keine Frameworks

Frameworks sind komplette **Speziallösungen**.

Keine Entwurfs-Regeln

Entwurfsmuster sind **keine starren Vorgaben**, sondern **flexible, anpassbare Vorschläge**.

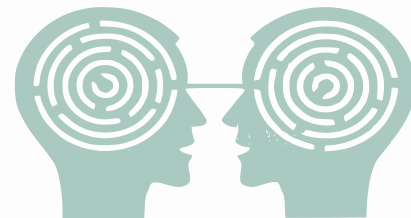
Keine Idiome

Idiome (Programmierstile) sind **sprachspezifische Konstrukte**.



Mustername

- **Stichwort** zur Benennung von Mustern
- wichtig für den **Austausch** mit Kollegen
- wichtig zum **klarereren Denken** in Entwurfsmustern



Problemabschnitt

- **Wann** ist das Muster anzuwenden?
- Welches Problem wird **adressiert**?
- Was ist der **Kontext**?



Lösungsabschnitt

- Aus welchen **Elementen** besteht die Lösung?
- Welche **Beziehungen** bestehen zwischen den Elementen?
- Wofür sind die Elemente **zuständig**?
- **Wie** arbeiten die Elemente zusammen?



Konsequenzabschnitt

- Welche **Vorteile** hat die Anwendung dieses Musters?
- Welche **Nachteile** hat die Anwendung dieses Musters?

Anwendung

1. Anhand der Beschreibungen der Entwurfsmuster (Konsequenzen!) **ein passendes auswählen**
2. **Übertragen** des Entwurfsmusters in den **konkreten** Anwendungsfall

Die **Wiederverwendung** einer Klasse **wird** oft dadurch **verhindert**, dass diese zwar im Wesentlichen die erforderliche Funktion realisiert, jedoch bestimmte **Details nicht ausreichend** realisiert sind.

- Die Klasse wird erneut programmiert, wobei **nur kleine Änderungen** vorgenommen werden.
- Die **Wiederverwendbarkeit** der Klasse **erhöht sich** bei jedem **Neuentwurf**.

Je nach Erfahrung des Programmierers wird nach einer kleineren oder größeren Anzahl von Entwürfen eine Klasse programmiert, die sich **wieder verwenden lässt ohne wesentliche Änderungen**.

Dabei stellt sich heraus, dass bei einem **guten Entwurf** weniger das konkret gelöste Problem im Vordergrund steht.

Vielmehr ist ein **allgemeines Prinzip** realisiert worden, das sich auf viele konkrete Gegebenheiten anwenden lässt.

Dieses allgemeine Prinzip lässt sich als Entwurfsmuster formulieren.

In guten Entwurfsmustern stecken die **Erfahrungen guter Programmierer**.

Durch die Kenntnis von Entwurfsmustern **gelingt** es (auch weniger erfahrenen Programmierern)...

- gut strukturierte und **wiederverwendbare** Programme selbst zu **entwickeln**
- **von anderen entwickelte** Programme zu **verstehen** und nachzunutzen.

Warum Entwurfsmuster? (III)

Entwurfsmuster sind nicht für spezielle Programmiersprachen entwickelt. Sie beschreiben vielmehr **allgemeine** Programmier**probleme** und deren **prinzipielle Lösung**. Sie lassen sich **in allen Sprachen** in konkrete Programme umsetzen.

Entwurfsmuster **helfen** dem Programmierer, sein Problem zu **verstehen**, zu **strukturieren** und mit Hilfe eines Programmes ggf. unter Verwendung vorhandener Bausteine zu **lösen**.

Die **Entwicklung** von Entwurfsmustern
beeinflusst die **Entwickler** von Programmiersprachen.

Nachteile

- manchmal **zu kompliziert**
- **noch keine** umfassenden, praktikablen **Kataloge**
- **nur** für eine **bestimmte Ebene** des Entwurfs sinnvoll
- Lohnen sich nur, **wenn Flexibilität und Erweiterbarkeit notwendig** sind
- bei machen Programmen **komplexer**
- Lösungen oftmals **umständlich**
- Verlängern die **Laufzeit**
- Sind ein **Kostenfaktor**

Finden **passender SW-Objekte**

- die während der Analyse-Phase gefundenen Objekte **reichen oft nicht**

Festlegen der **Objektgranularitäten**

- komplexe Objekte dürfen **nicht beliebig groß** werden

Festlegen der **Schnittstellen** von Objekten/Methoden

- mit den **richtigen Schnittstellen** steht und fällt ein Entwurf

Festlegen der **Objektimplementierung**

- der **richtige Einsatz von Vererbung** kann den Aufwand vermindern

Wiederverwendungsmechanismen aktiv und passiv nutzen / vorsehen

- Vererbung ist bei weitem nicht die einzige Möglichkeit

Implementierung der Part-of-Relation?

- als Objekt, Zeiger, Liste, Array, ...

Virtuelle Methoden oder parametrisierte Typen?

- **Flexibilität** zur Laufzeit oder nur zur Compilezeit
- weniger oder mehr **Effizienz** bei der Ausführung des Programms

Vererbung oder Delegation?

- **Vermeiden von Mehrfachvererbung** auf Kosten eines komplexeren Entwurfs

Behandlung von mehrdimensionalen Klassifizierungshierarchien?

- durch Mehrfachvererbung oder mit parametrisierten Typen?

→ Erfahrung und Problemkenntnisse nötig

Any Questions?

Begriffe



Wieder-
verwendung

Zweck/Arbeit	Erzeugende	Strukturelle	Verhaltensabhängige
Klasse	Fabrik	Adapter	Interpreter
Objekt	Abstrakte Fabrik Prototyp Einzelstück Builder	Adapter Brücke Fassade Stellvertreter Composite Decorator	Kommando Strategie Iterator Visitor Chain of Responsibility Mediator Memento Flyweight Observer State

- A) Entkopplungs-Muster
- B) Varianten-Muster
- C) Zustandshandhabungs-Muster
- D) Steuerungs-Muster
- E) Virtuelle Maschinen
- F) Bequemlichkeits-Muster

A) Entkopplungs-Muster (I)

 *Entkopplungs-Muster* teilen ein System in mehrere Einheiten, sodass **einzelne Einheiten unabhängig** voneinander erstellt, verändert, ausgetauscht und wiederverwendet werden können.

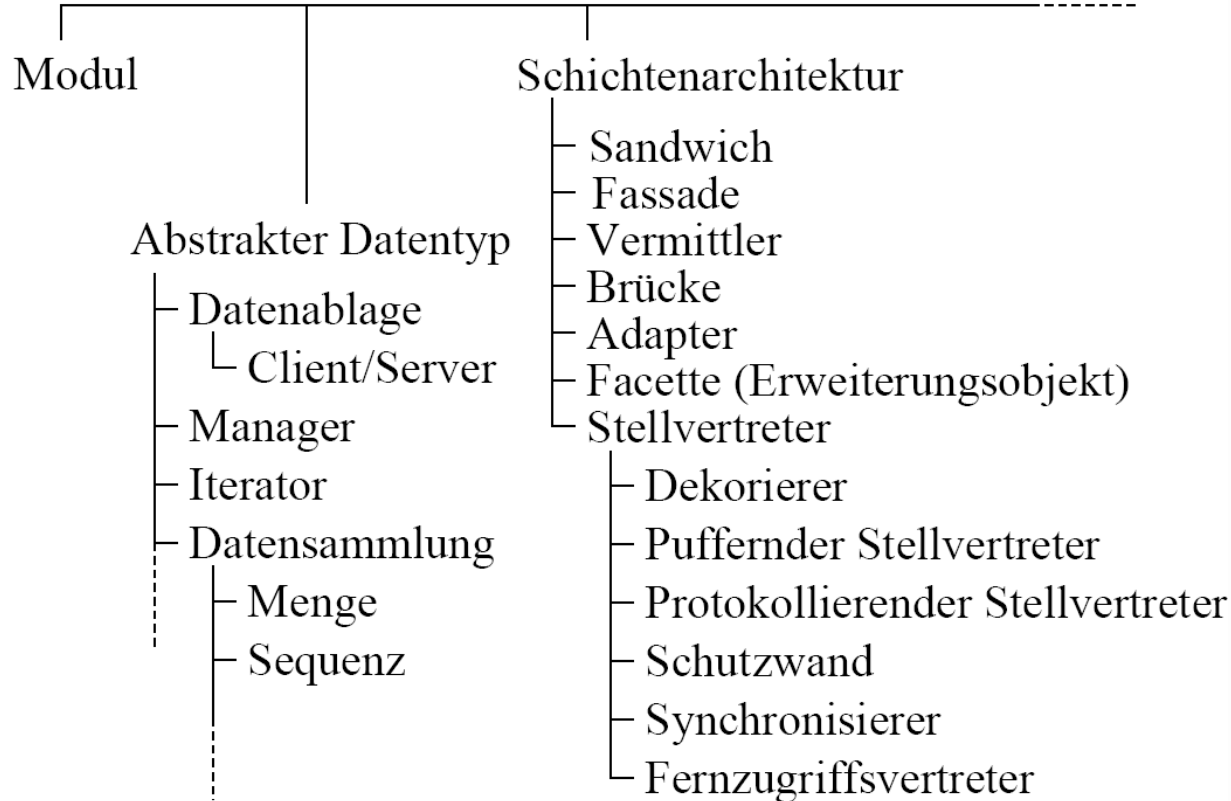
Vorteil

Das System kann durch **lokale Änderungen** verbessert, angepasst und erweitert werden, **ohne das ganze System** zu modifizieren.

Mehrere der Entkopplungs-Muster enthalten ein **Kopplungsglied**, das entkoppelte Einheiten über **eine Schnittstelle** kommunizieren lässt.

- Kopplungsglieder sind auch für das Koppeln *unabhängig* erstellter Einheiten brauchbar.

A) Entkopplungs-Muster (II)



A) Entkopplungs-Muster (III)



B) Varianten-Muster (I)

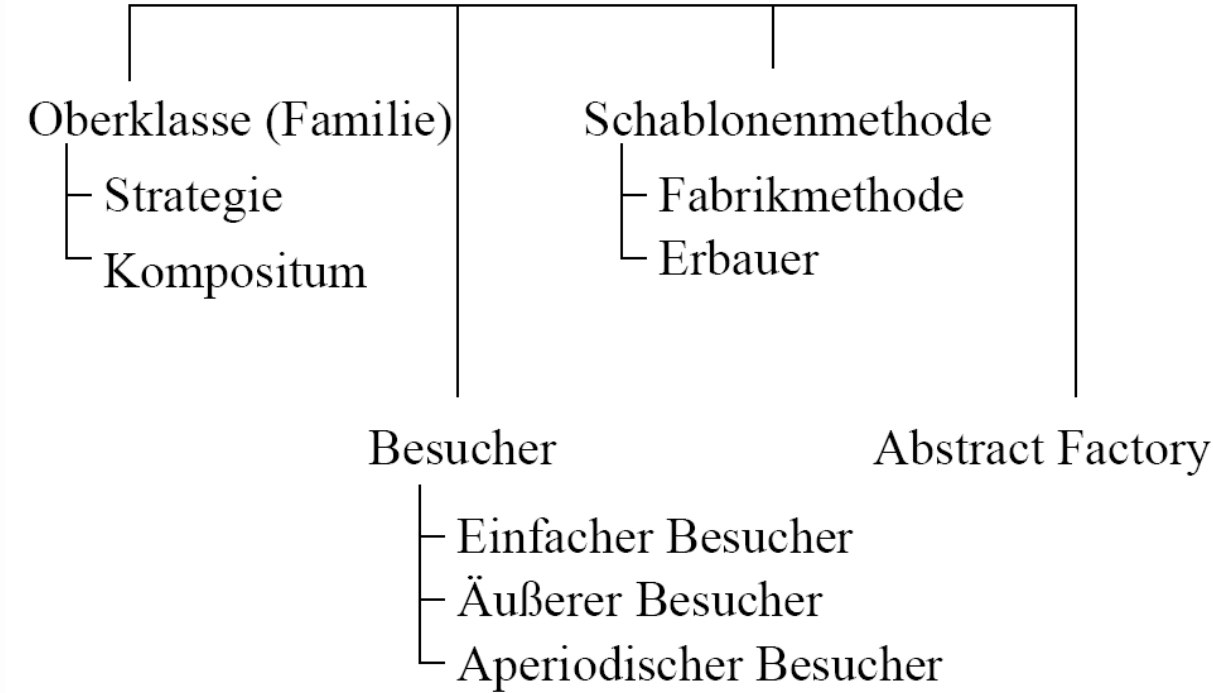
 In Mustern dieser Gruppe werden **Gemeinsamkeiten** von verwandten Einheiten aus ihnen **herausgezogen** und an einer einzigen Stelle beschrieben.

Vorteil

Aufgrund ihrer Gemeinsamkeiten können unterschiedliche Komponenten im gleichen Programm danach **einheitlich verwendet** werden.

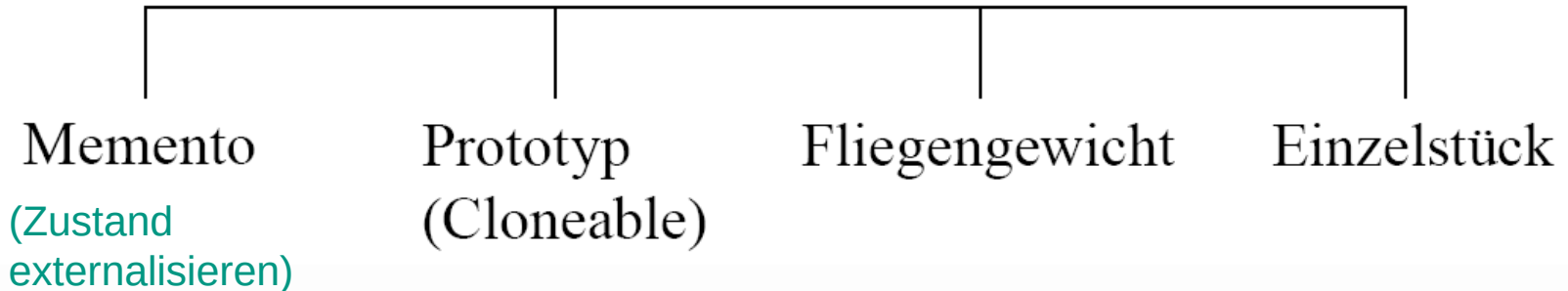
➤ **Vermeiden von Wiederholungen** desselben Codes.

B) Varianten-Muster (II)



C) Zustandshandhabungs-Muster

 Die Muster dieser Kategorie bearbeiten den **Zustand von Objekten**, **unabhängig** von deren **Zweck**.



D) Steuerungs-Muster (I)

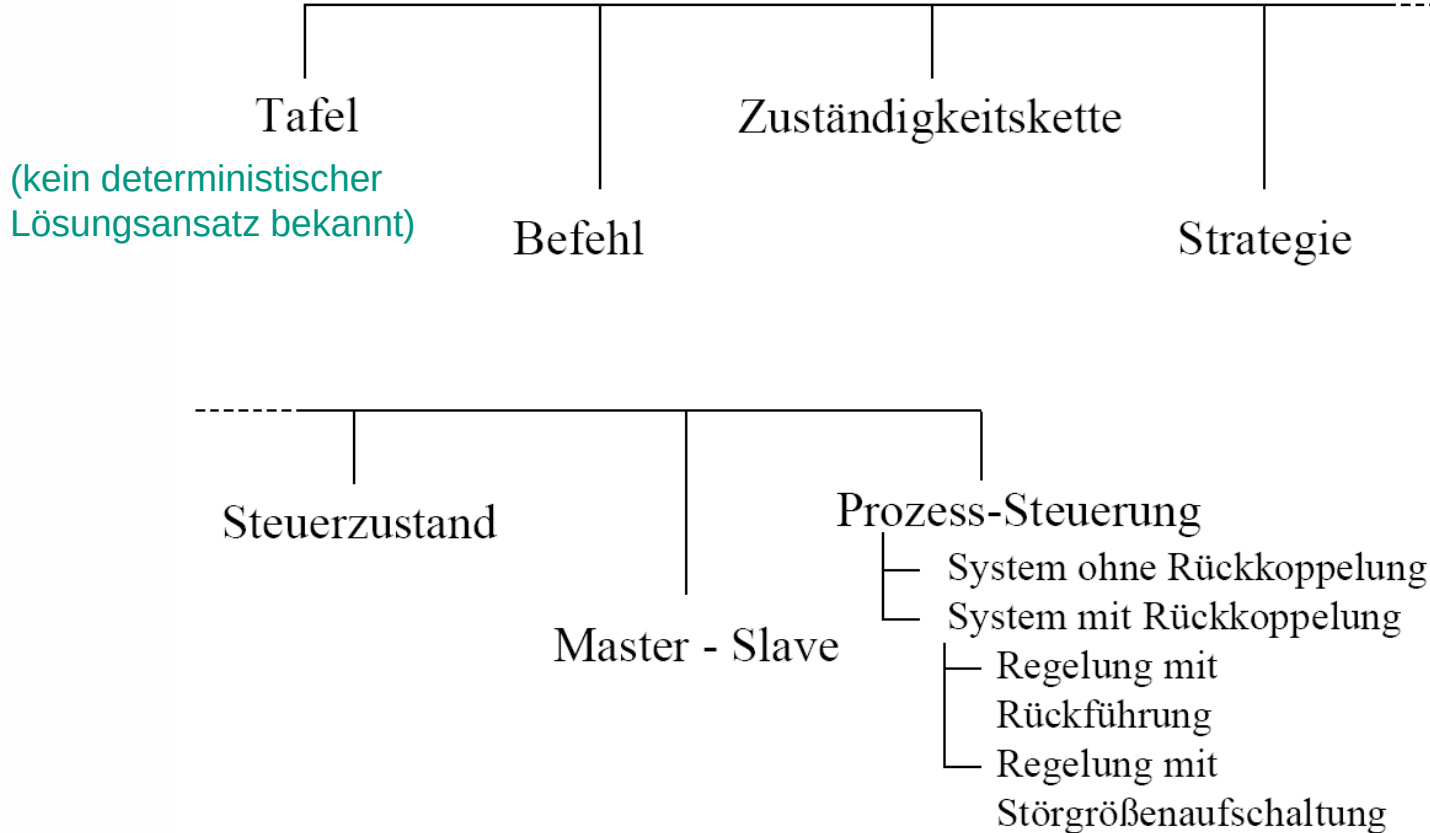


Steuerungs-Muster steuern den **Kontrollfluss**.

Vorteil

Es werden **zur richtigen Zeit die richtigen Methoden** aufgerufen.

D) Steuerungs-Muster (II)



E) Virtuelle Maschinen

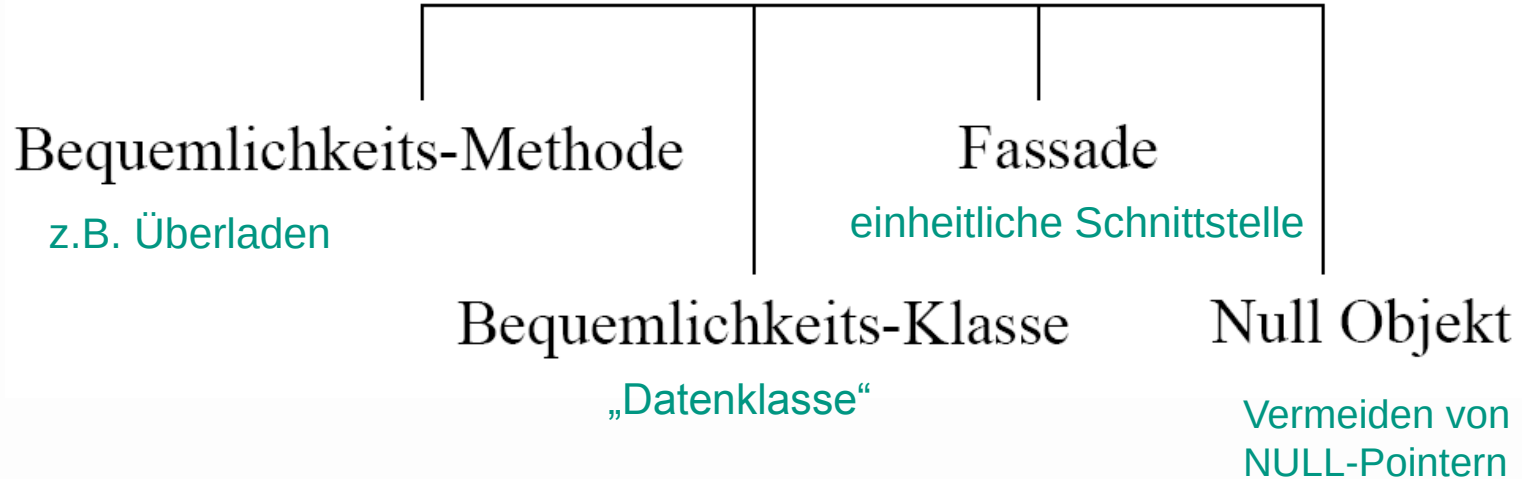


Virtuelle Maschinen erhalten **Daten** und ein **Programm als Eingabe** und führen das Programm **selbständig** an den Daten aus.

Virtuelle Maschinen sind in **Software**, nicht in Hardware implementiert.



F) Bequemlichkeits-Muster



Beispiele für häufige Fehler

Problem	Lösung
Ein Objekt direkt aus einer Klasse erzeugen	Abstrakte Fabrik, Prototyp
Abhängigkeit von einer bestimmten Methode	Kommando
Abhängigkeit von Hard- /Software	Abstrakte Fabrik, Brücke
Abhängigkeit von Implementierungen	Abstrakte Fabrik, Stellvertreter
Abhängigkeit von Algorithmen	Strategie, Besucher
Zusammenhängende Objekte	Fassade, Mediator
Unmöglichkeit, Klassen zu ändern	Adapter, Dekorieren

Fragen?



Inhalt

- Abstrakter Datentyp (ADT)
- Modul
- Datenablage (Repository)
- Client/Server
- Iterator
- Datenablage (Collections)
- Schichtenarchitektur
- Brücke
- Vermittler (Mediator)
- Adapter
- Stellvertreter (Proxy)
- Fließband
- Ereigniskanal
- Rahmenprogramm (Framework)

Abstrakter Datentyp (I)

ADT (Abstract Data Type)



Ein *abstrakter Datentyp* (**ADT**) definiert einen neuen Datentyp zusammen mit geeigneten Operationen. Die Implementierung dieses Datentyps ist wie beim Modul hinter einer (änderungs-unempfindlichen) **abstrakten Schnittstelle** verborgen. → OO-Klasse

ClassName
Operation1() Type Operation2() ...
instanceVariable1 Type instanceVariable2 ...

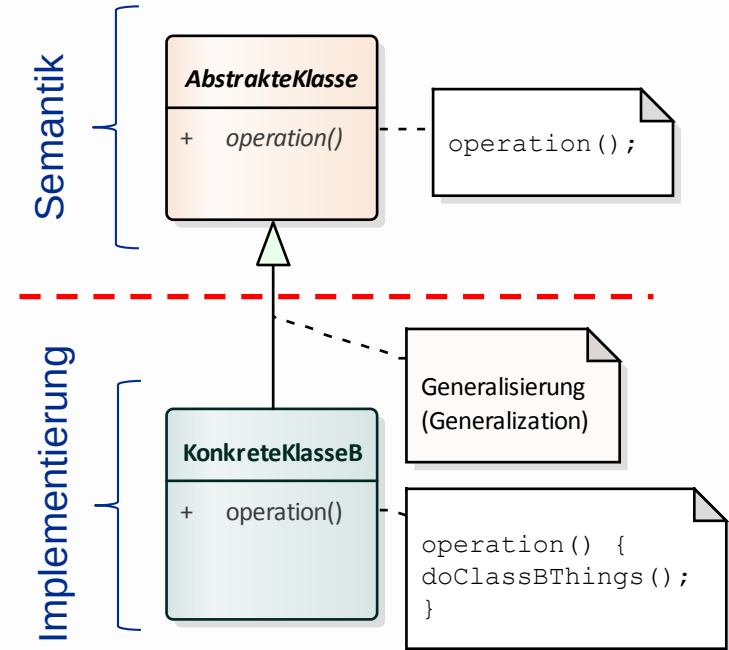
Unterschiede zum Modul

- Modul ist i.d.R. **eine größere Einheit**.
 - Kann z.B. mehrere voneinander abhängige ADTs zusammenfassen.
Beispiel: eine Aggregatsklasse mit zugehörigem Iterator.
- Von einem Modul gibt es in jedem Programm **nur ein Exemplar**.
Von einem ADT kann man beliebig viele Exemplare anlegen.

Prinzip

Konkrete Klasse hat die **Verpflichtung**, die Operation zu implementieren

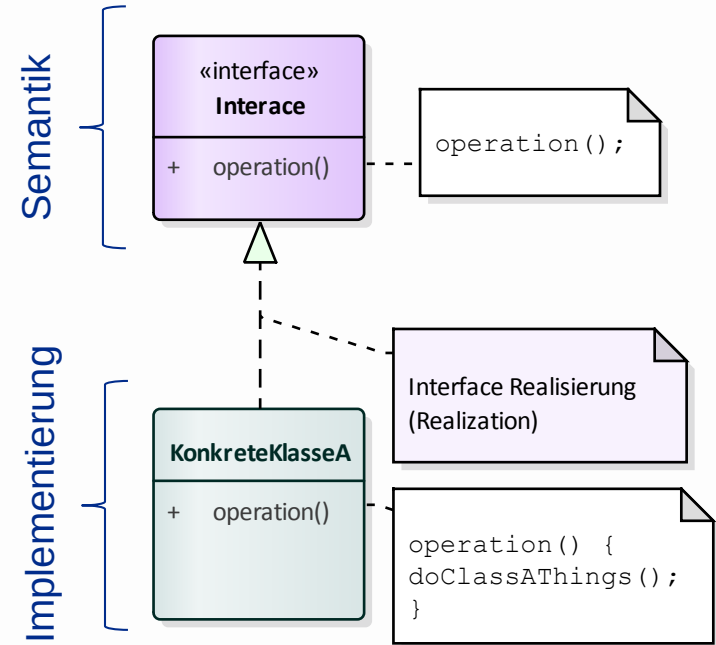
- **Trennung** von Semantik und Implementierung
- Als **Abstrakte Klasse** oder Interface



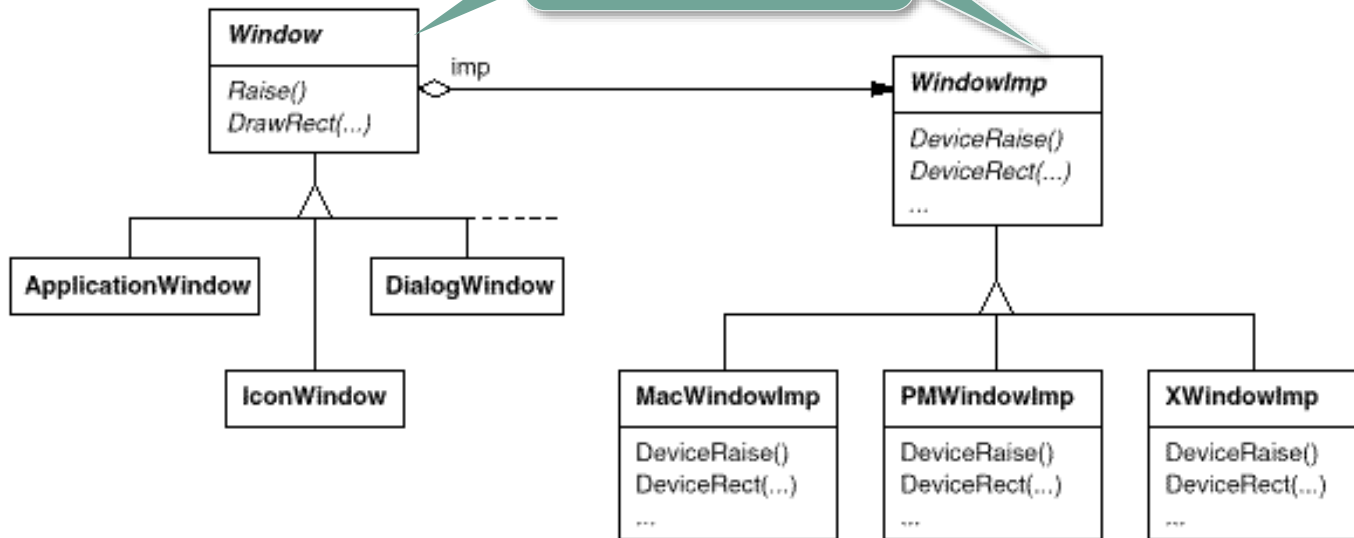
Prinzip

Konkrete Klasse hat die **Verpflichtung**, die Operation zu implementieren

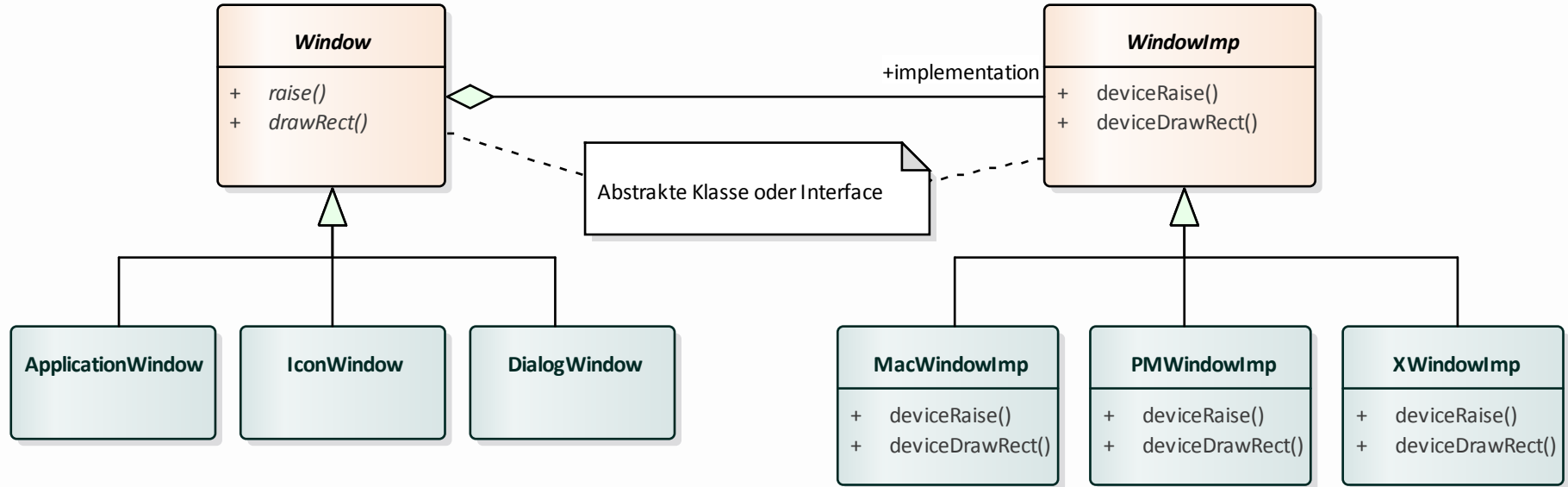
- **Trennung** von Semantik und Implementierung
- Als Abstrakte Klasse oder **Interface**



Abstrakte Klasse
oder Interface

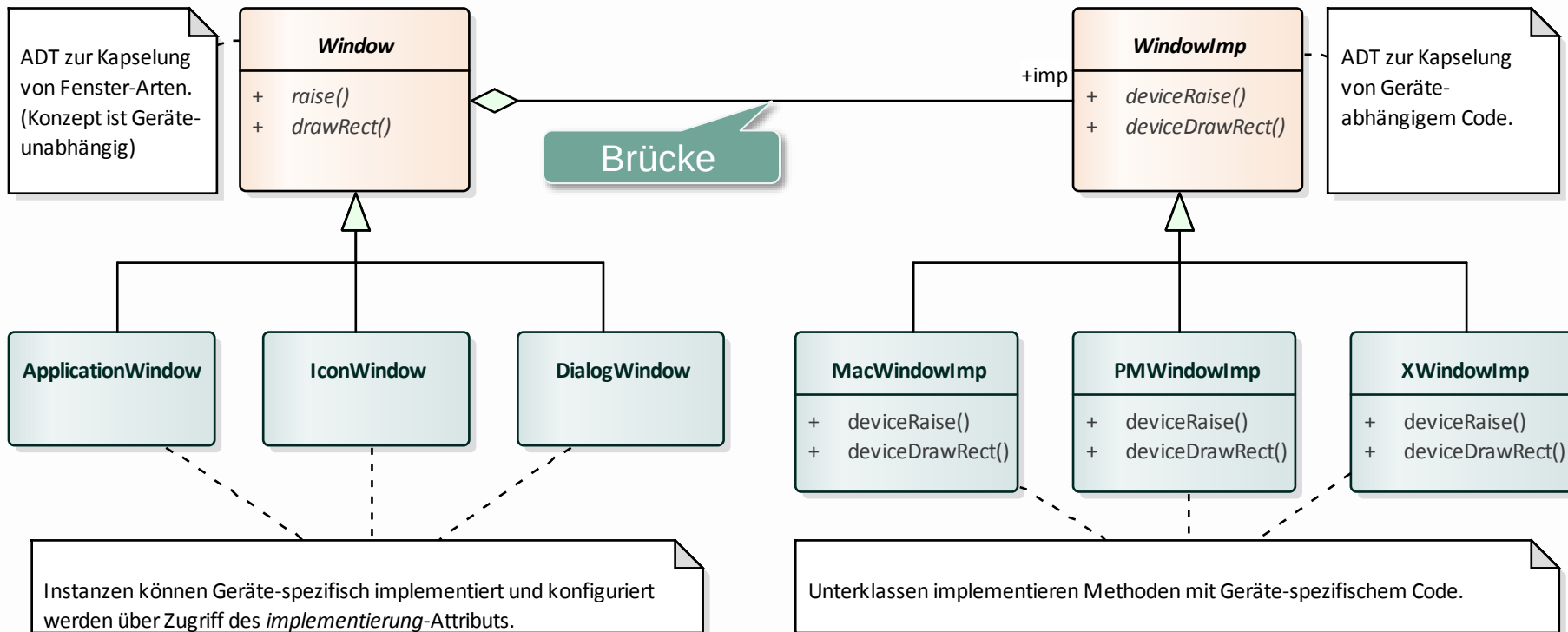


Beispiel: Abstrakter Datentyp – Fenster



PM: Presentation Manager

Beispiel: Abstrakter Datentyp – Fenster



PM: Presentation Manager

Beispiel: Abstrakter Datentyp – Fenster (Codeausschnitt)

```
public abstract class Window {  
    protected WindowImp windowImp;    // Instance of desired device  
    public Window(WindowImp imp) {  
        this.windowImp = imp;  
    }  
    public void DrawRect() {  
        this.windowImp.DeviceRect(); // Use device-specific code  
    };  
}  
  
public class IconWindow extends Window {  
    public IconWindow(WindowImp imp) {  
        super(imp);  
    }  
}
```

Modul



Ein *Modul* ist eine **Menge von Programmkomponenten**, die gemeinsam entworfen und verändert werden. Diese Komponenten werden hinter einer änderungs-unempfindlichen Schnittstelle verborgen (**“Geheimnisprinzip”**).

Ziel der Modularisierung: **Entkopplung**

- modul-interne Komponenten können **verändert oder ersetzt** werden, ohne die Benutzer des Moduls anpassen zu müssen.

Um effektiv zu sein, muss man die möglichen **Änderungen voraussehen** und in die Modulstruktur und Schnittstellen **hineinplanen**.

Kandidaten für Veränderung/Verbergung

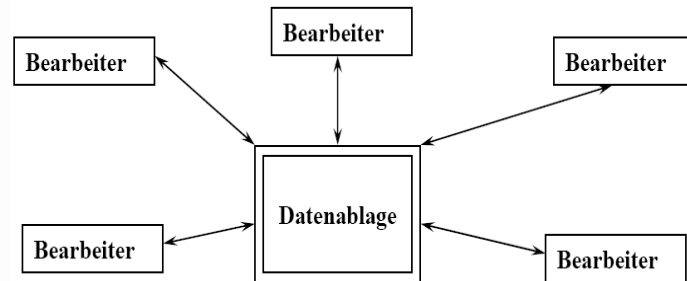
- Datenstrukturen und Operationen, Größe der Datenstrukturen, Optimierungen
- maschinennahe Details
- betriebsystemnahe Details
- Ein-/Ausgabeformate
- Benutzerschnittstellen
- Texte für Dialoge und Fehlermeldungen
- Maßeinheiten (Internationalisierung)
- Reihenfolge der Verarbeitung, Vorverarbeitung, inkrementelle Verarbeitung
- Zwischenpufferung

Datenablage (Repository)

Zweck



Eine Menge unabhängiger **Komponenten kommunizieren** über eine **zentrale Ablage**, indem sie Elemente in dieser Datenstruktur ablegen oder aus ihr herausholen.



Beispiele für große Datenablagen

Datenbanken, Hypertextsysteme.

Zusätzliche Mechanismen gegenüber einfachen Datenablagen:
Persistenz, Zugriffskontrolle, Transaktionsverwaltung

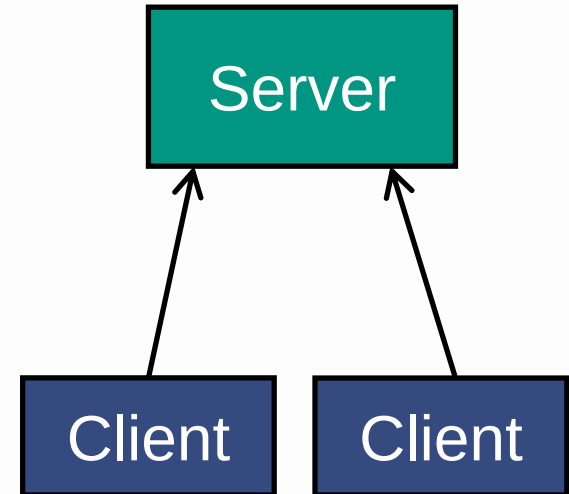
Client/Server



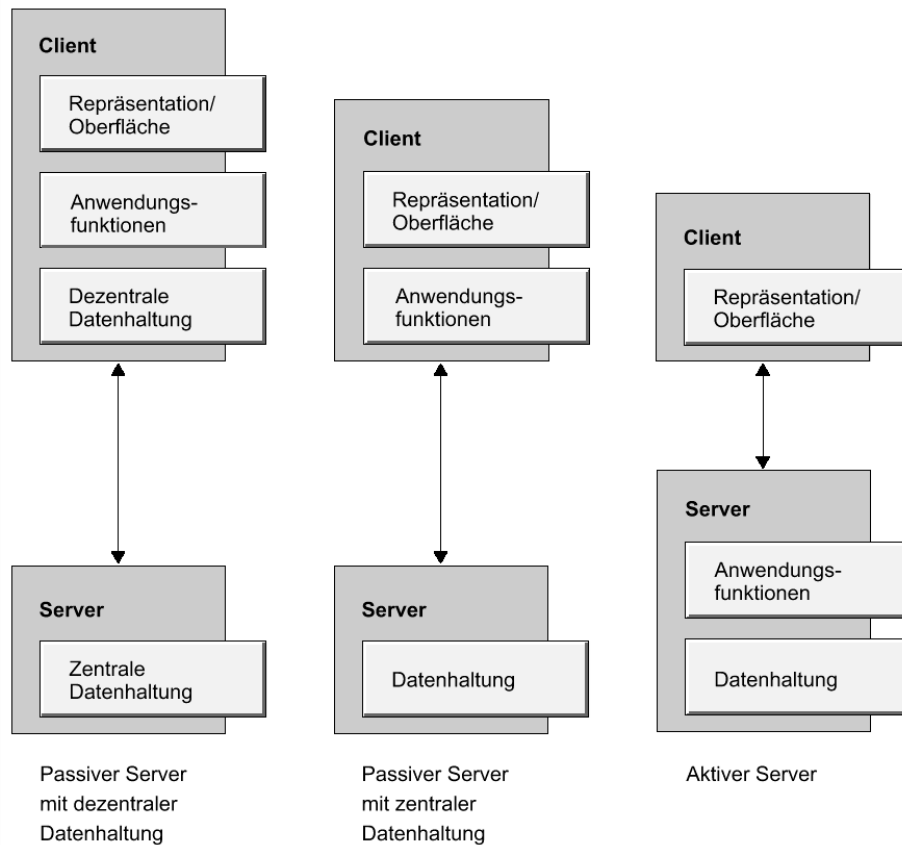
Zweck

- Verarbeitung **im Parallelen**
- Aufgabenteilung
- **Kapselung** von Programmeinheiten
- Ein Server, mehrere Klienten

Kunde/Dienstleister
(Client/Server)



Client/Server



Abstrakter
Datentyp

Client/Server



Modul

Iterator (Verhaltensmuster)

Iterator

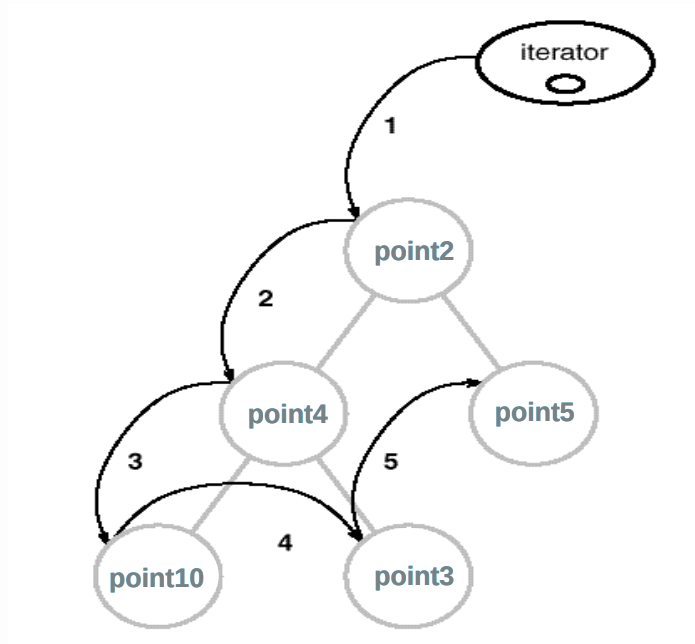
*auch bekannt als:
Enumerator, Cursor*



Zweck

- **Zugriff** auf den **Inhalt eines zusammengesetzten Objekts**, ohne seine interne Struktur offenzulegen
- mehrfache, gleichzeitige **Traversierungen** auf zusammengesetzte Objekte
- Einheitliche Schnittstelle zur Traversierung unterschiedlich zusammengesetzter Strukturen (**polymorphe Iteration**)

Iterator (Verhaltensmuster)

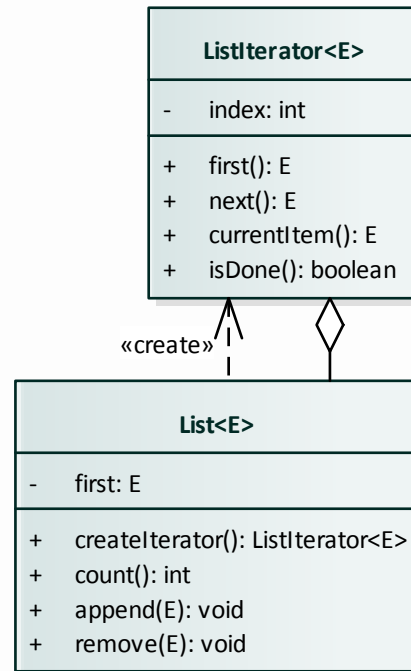
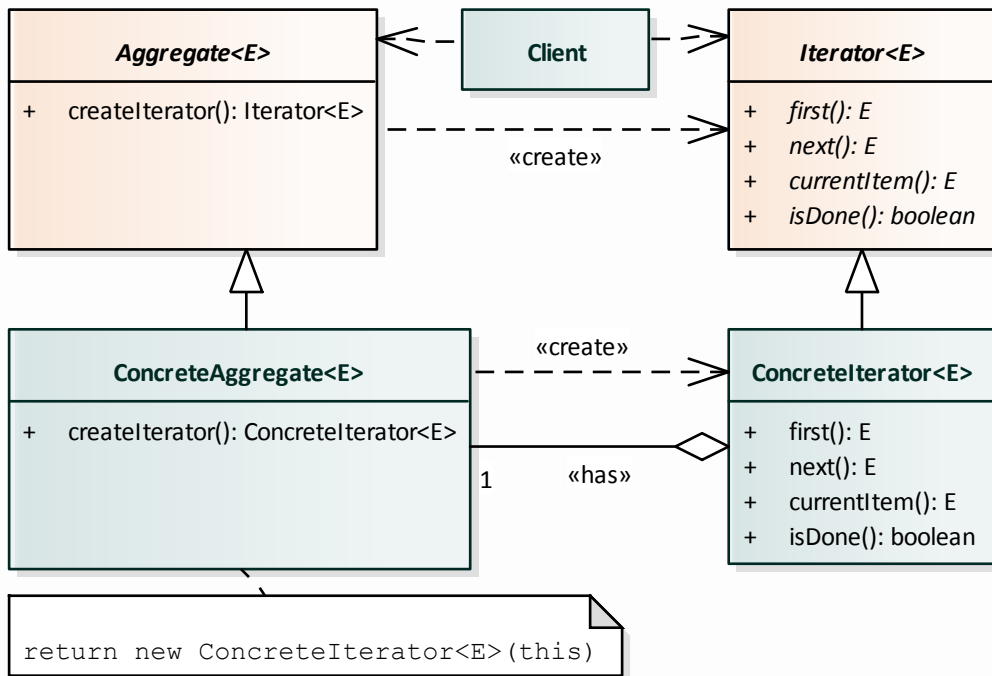


Iterator

Collection

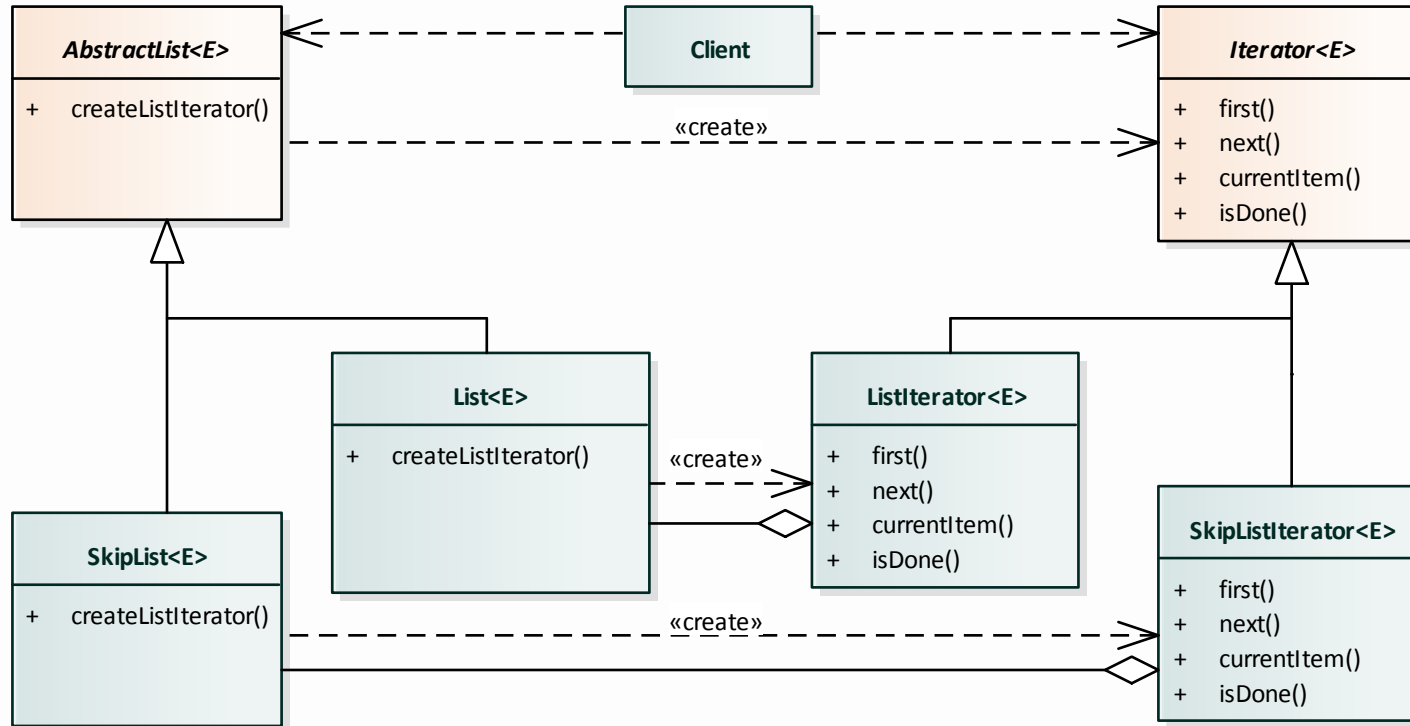
>	point2
>	point4
>	point10
>	point3
>	point5
	-
	-

Iterator – Klassendiagramm (I)

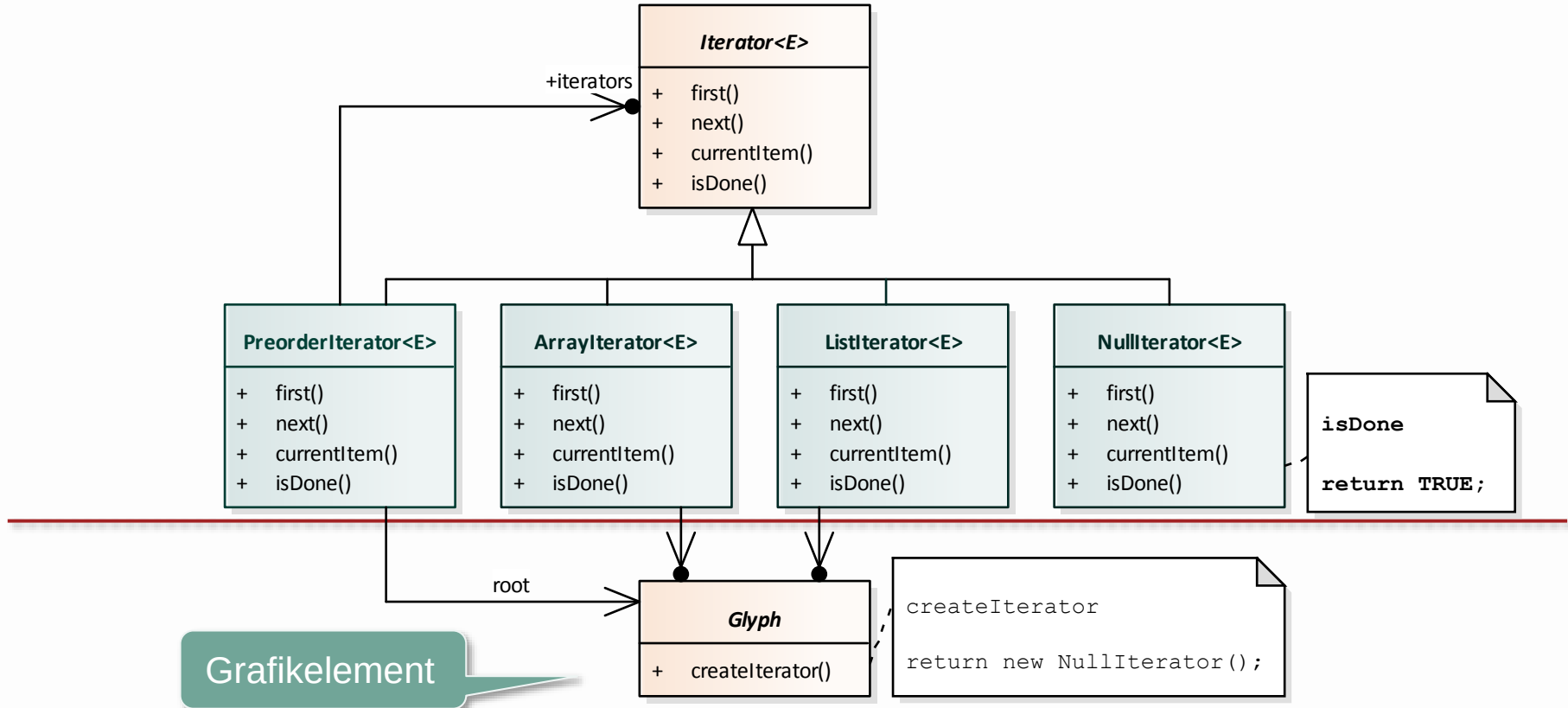


Beispiel eines Aggregats:
die Datenstruktur ‚Liste‘

Iterator – Klassendiagramm (II)



Iterator – Klassendiagramm (III)



Grafikelement

Iterator-Konzept (I)

Aufzählen der in einem ‚Behälter‘ befindlichen Elemente

- Keine Aussage über die Reihenfolge!
- Interface: **java.util.Iterator**

```
interface Iterator {  
    public abstract boolean hasNext();  
    public abstract Object next();  
    public void remove();  
}
```

Verwendungs-
beispiel:

```
Iterator i = ...;  
while (i.hasNext()) {  
    doSomething(i.next());  
}
```

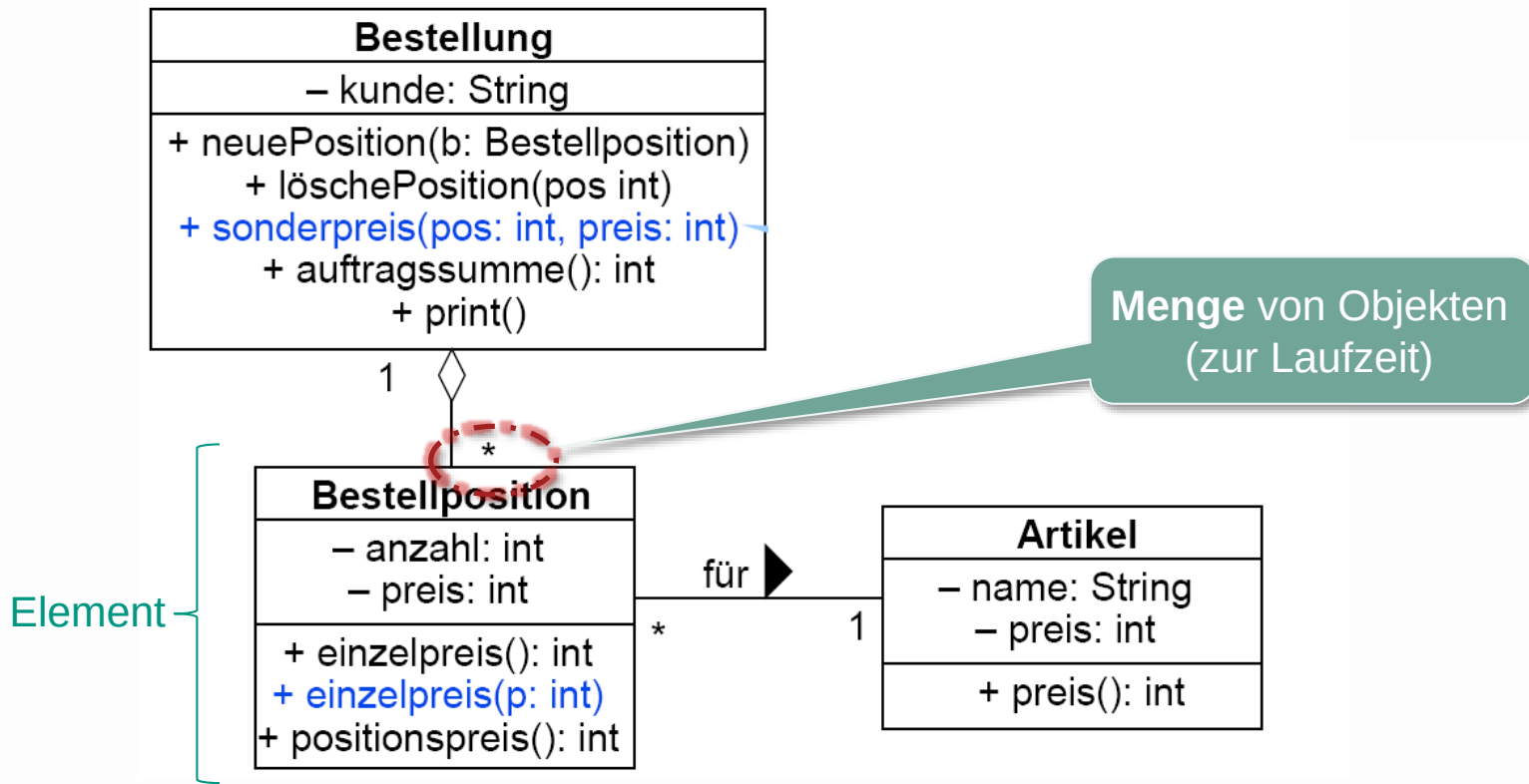
Iterator-Konzept (II)

Erzeugung eines Iterators für eine beliebige Kollektion
(deklariert in **java.util.Collection**)

```
public Iterator iterator();
```

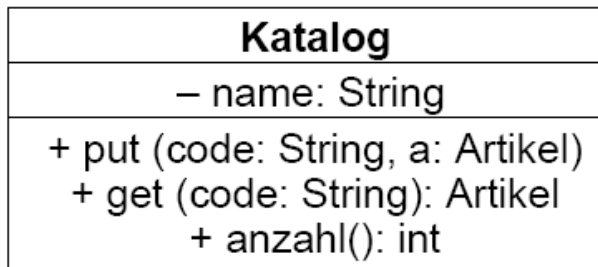
→ Datenablage: Collections

Modellierung: * ← Realisierung?



→ Datenablage: Collections

Qualified Assoziationen



Qualifier

Map
Ordnet jedem Code seinen Artikel zu

Qualifier

Schlüssel	Wert
„007“	James Bond
„16:50 ab Paddington“	Miss Marple
„10 Gebote“	Die Bibel
...	...

→ Datenablage: Map

Collection (Kollektion)

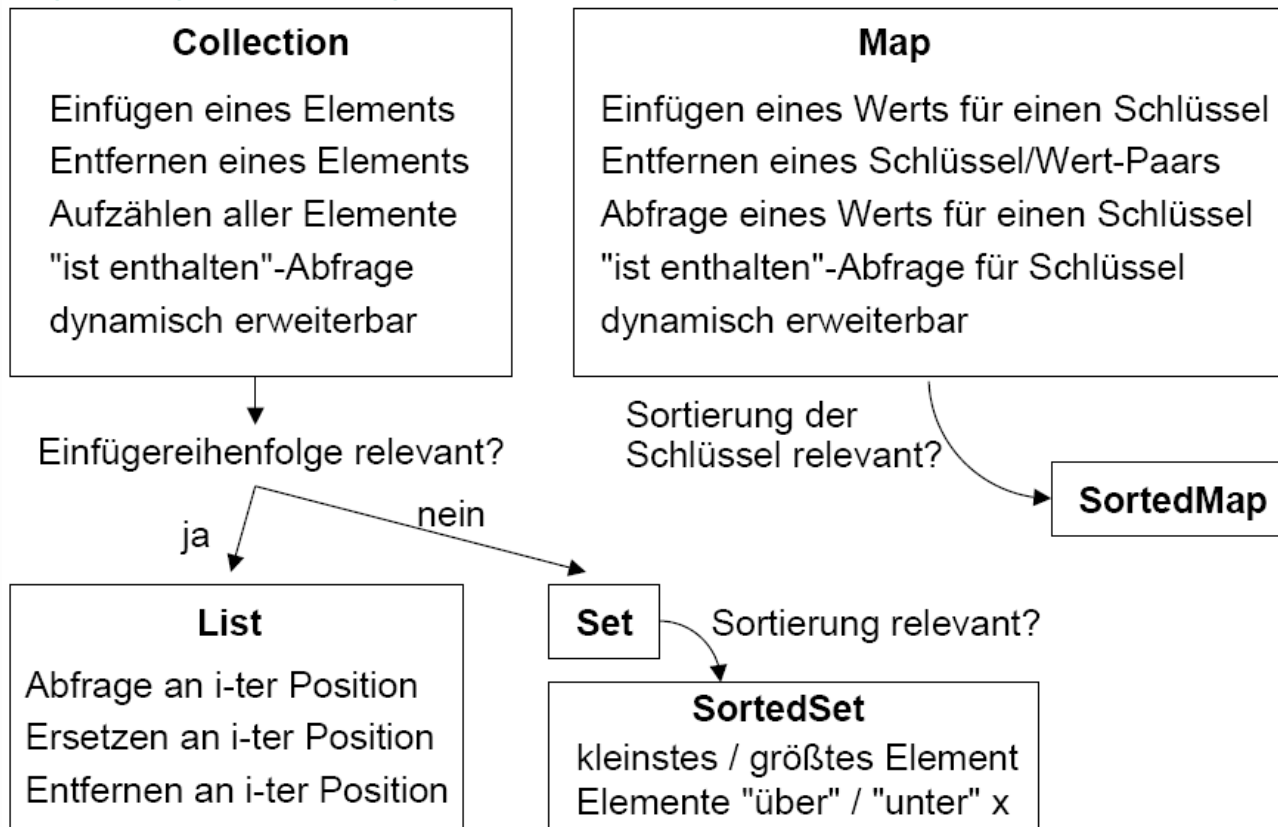
- **Ansammlung** von Datenelementen
- **Hinzufügen**, **Entfernen**, **Suchen**, **Durchlaufen**

Set (Menge)

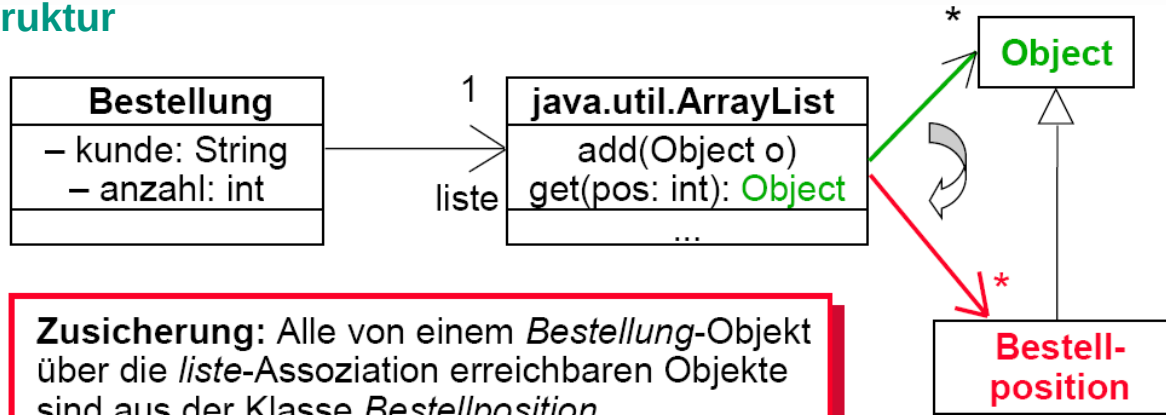
- **Mehrfachvorkommen** spielen keine Rolle
- Reihenfolge des Einfügens spielt keine Rolle
- **SortedSet** (geordnete Menge):
Ordnung auf den Elementen

Klassifikation von Java-Datenstrukturen (II)

- **List** (Liste)
 - Mehrfachvorkommen separat abgelegt
 - Reihenfolge des Einfügens bleibt erhalten
- **Map** (Abbildung)
 - Zuordnung von Schlüsselwerten auf Eintragswerte
 - Mehrfachvorkommen bei Schlüsseln verboten, bei Einträgen erlaubt
 - **SortedMap** (geordnete Abbildung):
Ordnung auf den Schlüsseln



Beispielstruktur



Zusicherung: Alle von einem *Bestellung*-Objekt über die *liste*-Assoziation erreichbaren Objekte sind aus der Klasse *Bestellposition*.

Typanpassung (cast):

Explizite Typanpassung (*dynamic cast*) erforderlich, wenn Operationen der Unterklasse auf Objekte anzuwenden sind, die sich in Variablen einer Oberklasse befinden:

(Typ) Variable

hier: *(Bestellposition) liste.get(i)*

java.util.Collection (Auszug aus Protokoll)

```
public interface Collection {  
    public boolean add (Object o);  
    public boolean remove (Object o);  
    public void clear();  
    public boolean isEmpty();  
    public boolean contains (Object o);  
    public int size();  
    ...  
    public Iterator iterator();  
}
```

Iterator Pattern

java.util.List (Auszug)

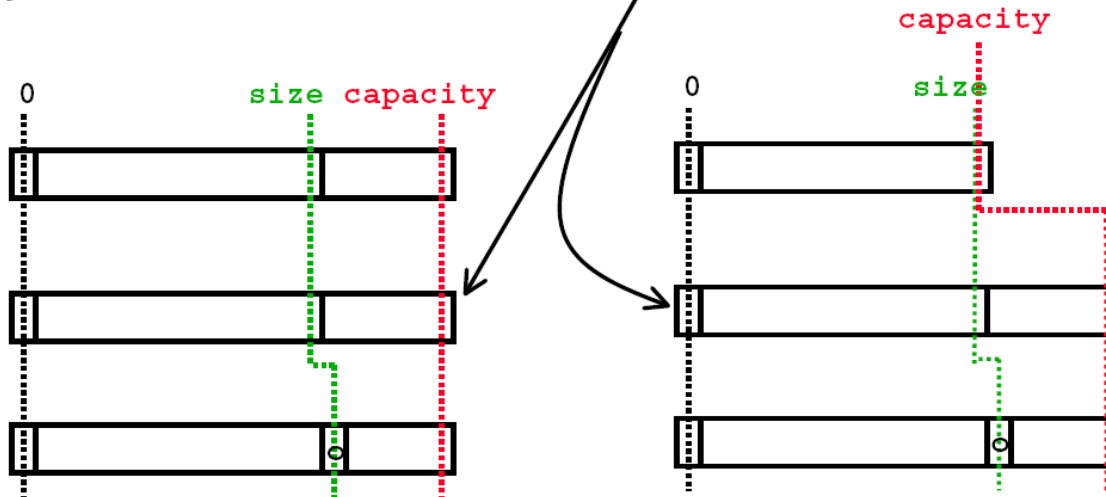
```
public interface List extends Collection {  
    public boolean add (Object o);  
    public boolean remove (Object o);  
    public void clear();  
    public boolean isEmpty();  
    public boolean contains (Object o);  
    public int size();  
    ...  
    public Object get (int index);  
    public Object set (int index, Object element);  
    public boolean remove (int index);  
    public int indexOf (Object o);  
}
```



Collection

Anfügen an ArrayList

```
public boolean add(Object o) {  
    ensureCapacity(size + 1);  
    elementData[size++] = o;  
    return true;  
}
```



implementiert als **doppelt verkettete Liste**



ArrayList oder LinkedList?

Gemessener relativer **Aufwand für** Operationen auf **Listen**:

Typ	Lesen	Iteration	Einfügen	Entfernen
ArrayList	110	490	3790	8730
LinkedList	1980	220	110	110

aus: Eckel, *Thinking in Java*

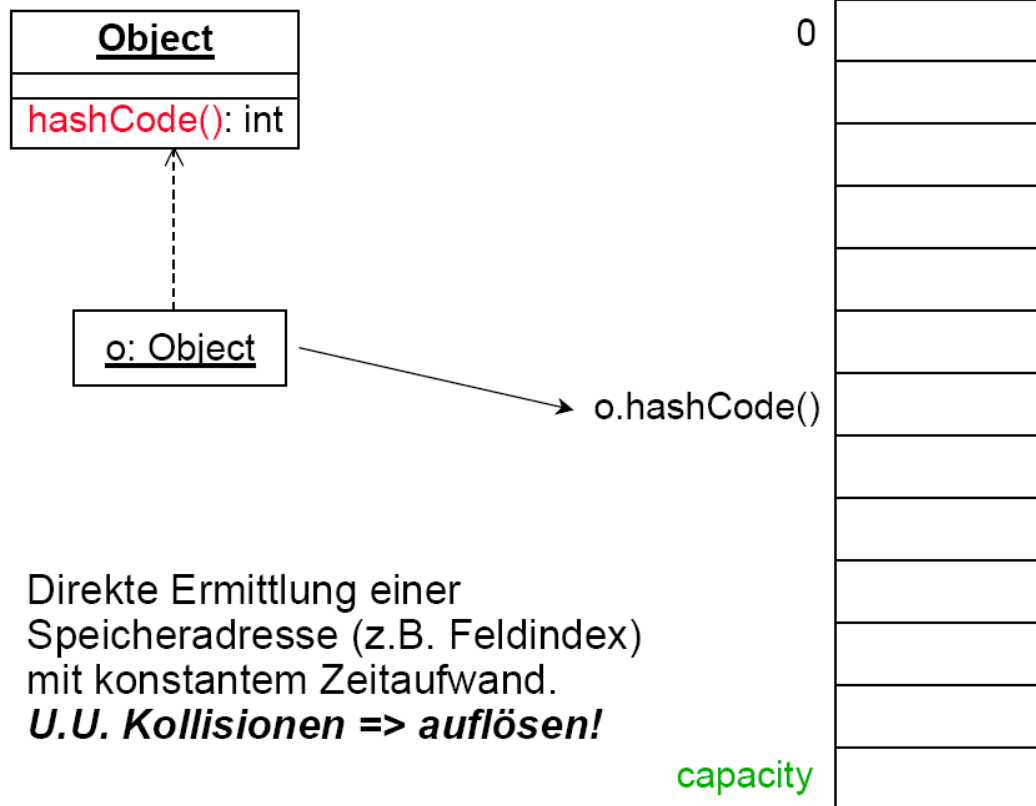
Stärken von **ArrayList**

- Iteration
- **wahlfreier Zugriff**

Stärken von **LinkedList**

- Iteration
- **Einfügen und Entfernen irgendwo** in der Liste

Prinzip einer Hashfunktion



HashSet oder TreeSet?

Gemessener relativer Aufwand für Operationen auf Mengen:

Typ	Einfügen	Enthalten	Iteration
HashSet	7,4	6,6	9,5
TreeSet	31,1	18,7	11,8

aus: Eckel, *Thinking in Java*

Stärken von HashSet

- unsortierte Mengen
- Verhalten von Iteration abhängig von reservierter Größe der Tabelle

Stärken von TreeSet

- sortierte Mengen
- Verhalten von Iteration abhängig von Anzahl der Elemente

Fragen?



Schichtenarchitektur (I)

Schichtenarchitektur (Multitier Architecture)



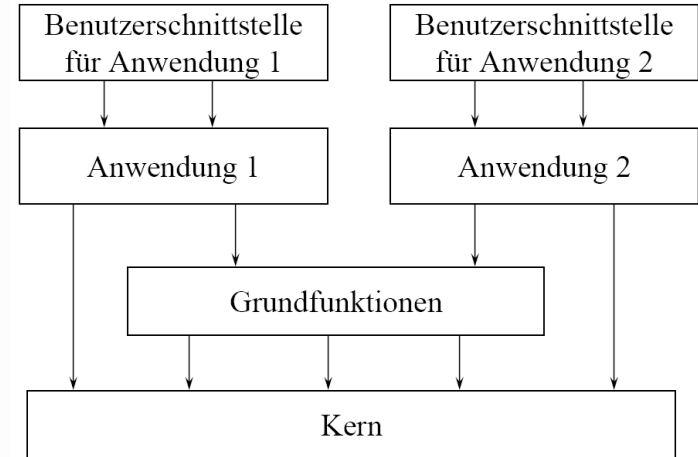
Zweck

Gliedere ein System in eine **hierarchisch geordnete Menge** von Schichten.



Eine **Schicht** besteht aus einer **Menge von Software-Komponenten** mit einer wohl definierten Schnittstelle.

Sie nutzt die darunter liegenden Schichten als Klient, und stellt seine Dienste an darüber liegende Schichten zur Verfügung.



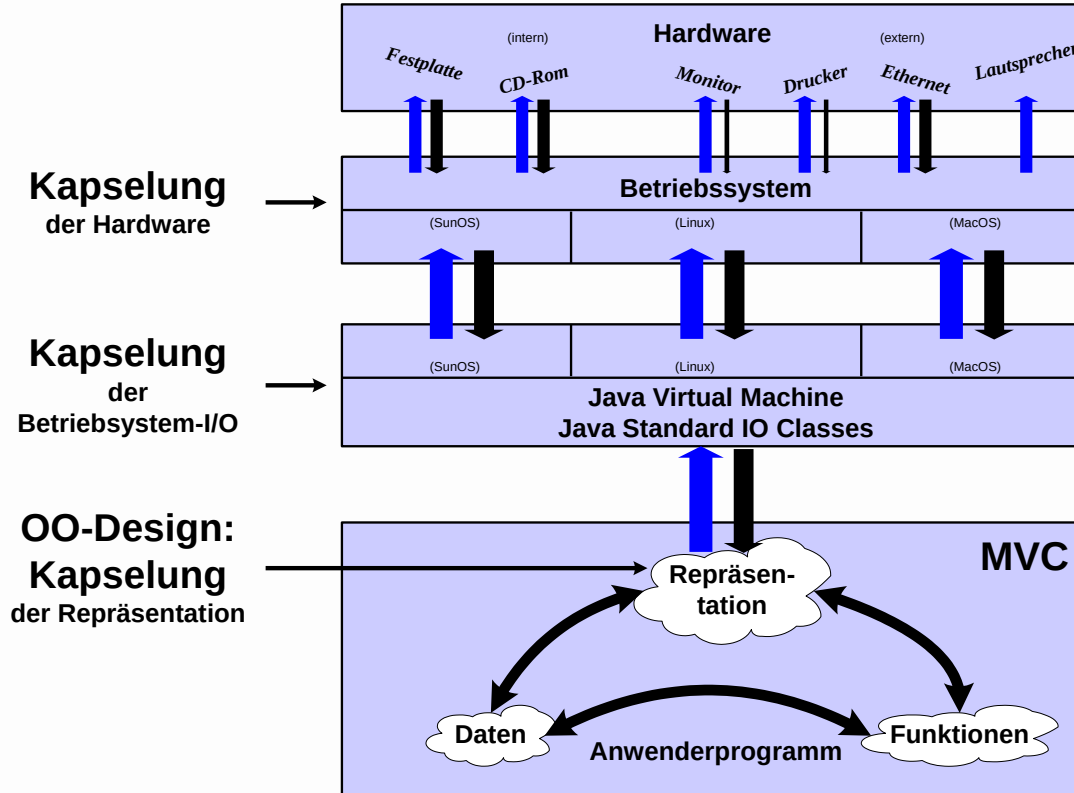
Beispiele

- Mikrokerne (Betriebssysteme)
- Protokolltürme bei der Datenfernübertragung
- Informationssysteme (auf Datenbanken aufbauend)

In manchen Systemen sind Benutzung nur zwischen **aufeinander liegenden** Schichten erlaubt.

Eine weitere Variante ist, dass jede Schicht neben den eigenen Komponenten nur eine **sorgfältig bestimmte Untermenge** der Komponenten der darunter liegenden Schicht weiterexportiert.

Beispiel Schichtenarchitektur



- **Unabhängige Entwicklung** und Korrektur, Austausch von Schichten
- Schrittweiser Aufbau und **schrittweise** Testen
- **Wiederverwendung** von tieferen Schichten in anderen Konfigurationen.

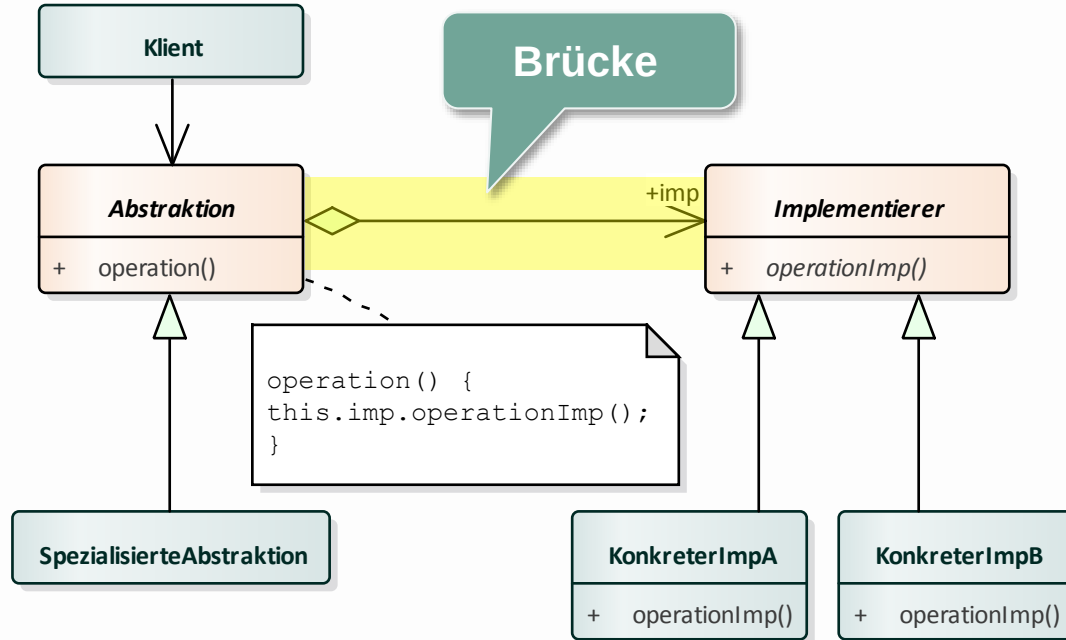
Brücke (Bridge)

Brücke (Bridge)
auch bekannt als:
Handle/Body

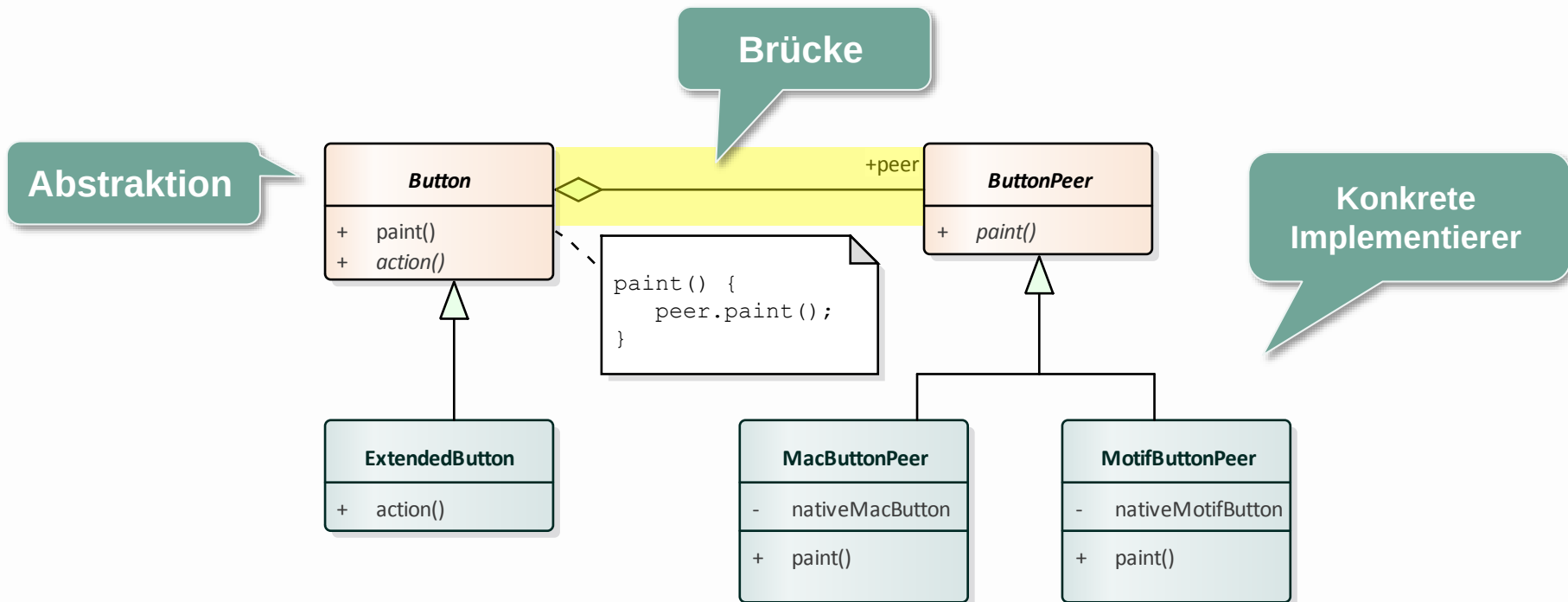
Zweck



Entkopple eine **Abstraktion** von ihrer **Implementierung**, sodass beide unabhängig voneinander variiert werden können.



- **imp**-Attribut wird die Instanz einer Implementierer-Klasse zugewiesen.
- **Abstraktion** wird mit den dort spezifizierten Methoden implementiert.



- 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.
- 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.





Zweck

Erzeuge ein Objekt, welches das **Zusammenspiel einer Menge** von Objekten in sich **kapselt**.

Vermittler erreichen, das Zusammenspiel der Objekte **von ihnen unabhängig zu variieren**.

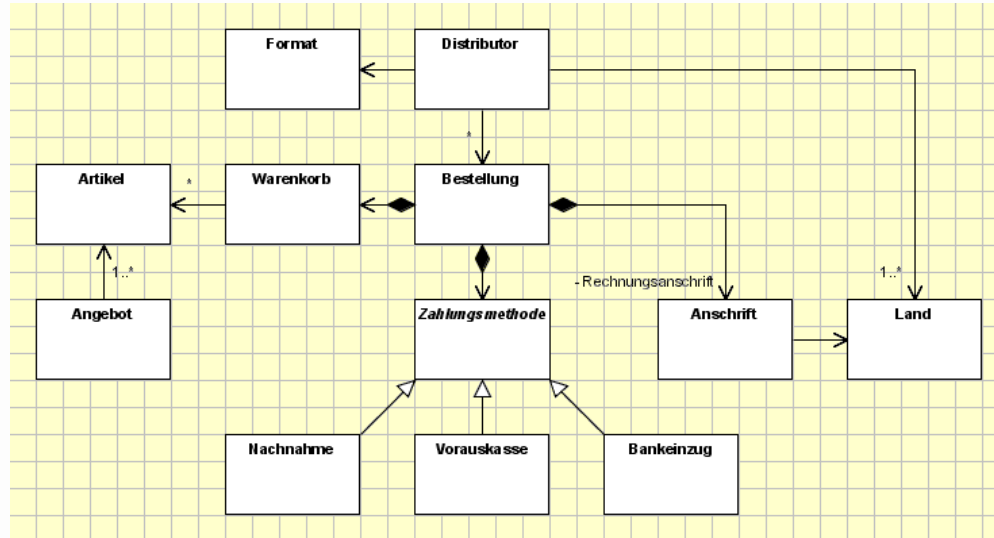
Vermittler (Mediator-Pattern)

Problem

Viele Verbindungen zwischen Objekten.

Schwierigkeit, das Verhalten des Systems auf bedeutsame Weise zu ändern, da Verhalten über so viele Objekte verstreut ist

→ viele Unterklassen

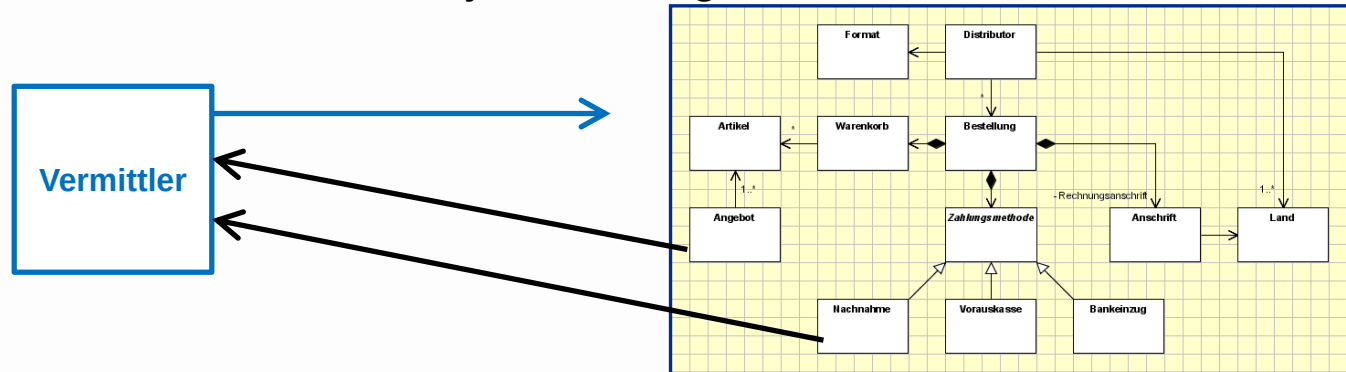


Vermittler (Mediator-Pattern)

Lösung

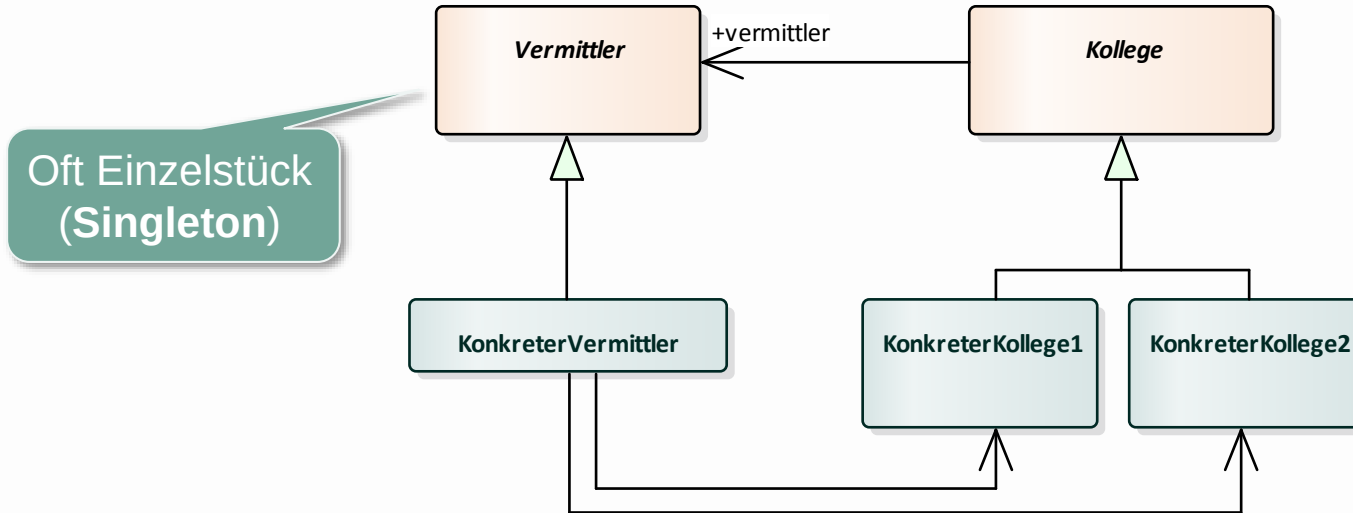
Separates Vermittlerobjekt kapseln.

- Vermittler ist für **Kontrolle** und **Koordination** der Interaktion innerhalb einer Gruppe von Objekten zuständig.
- **Objekte kennen Vermittler** und reduzieren dadurch Anzahl ihrer Verbindungen.
- Vermittler verhindert, dass Objekte Bezug zueinander nehmen.



Anwendung, falls...

- ... Eine Menge von Objekten vorliegt, die **in komplexer Weise** miteinander zusammenarbeiten.
- ... Die **Wiederverwendung** eines Objektes **schwierig** ist, da es mit vielen anderen Objekten zusammenarbeitet.
- ... Ein auf mehrere Klassen **verteiltes Verhalten** angepasst werden soll, **ohne viele Unterklassen** bilden zu müssen.



Mediator (Vermittler)

- **Schnittstelle** für die Interaktion mit Kollegen-Objekten.

ConcreteMediator

- implementiert **Gesamtverhalten** durch Koordination der Kollegen-Objekte.

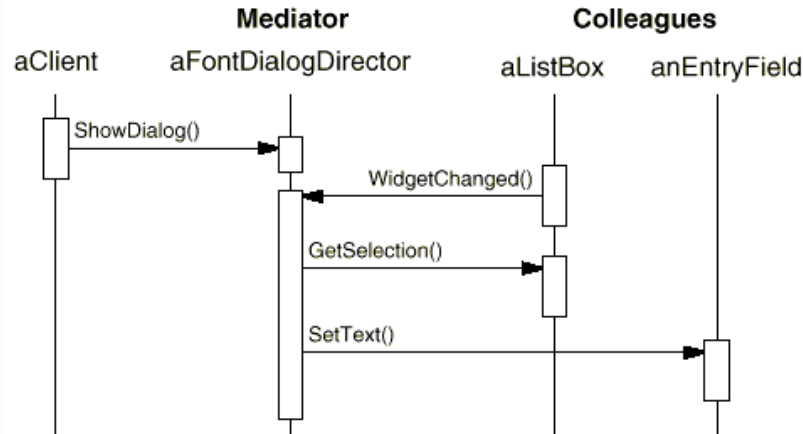
Kollege

- jede Kollegen-Klasse **kennt ihre Vermittler-Klasse**.
- jedes Kollegen-Objekt arbeitet mit Vermittler zusammen, statt mit Kollegen-Objekten.

Struktur (III)

Interaktion

- **Kollegen**-Objekte senden und empfangen **Anfragen** von einem Vermittler-Objekt.
- Der **Vermittler** implementiert das Gesamtverhalten durch das **Weiterleiten der Anfragen** zwischen den richtigen Kollegenobjekten.



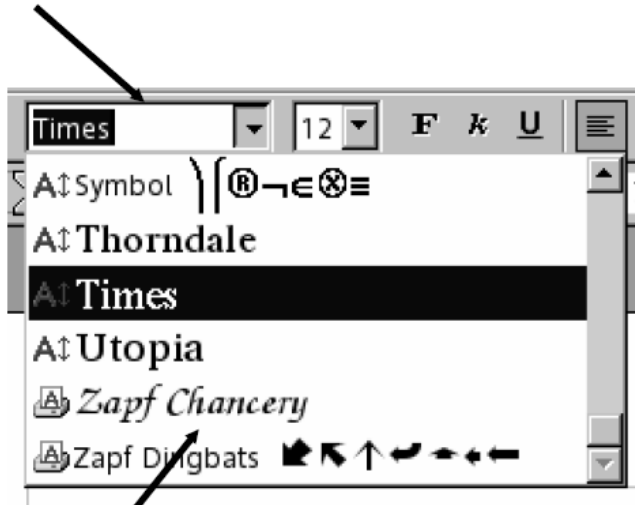
Beispiel: Dialogbox (I)

- Es gibt oft **Abhängigkeiten zwischen den Elementen** (*Knöpfe, Menüs, Eingabefelder, etc.*) einer Dialogbox.
Bsp: Ein Knopf muss 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.

Beispiel: Dialogbox (II)

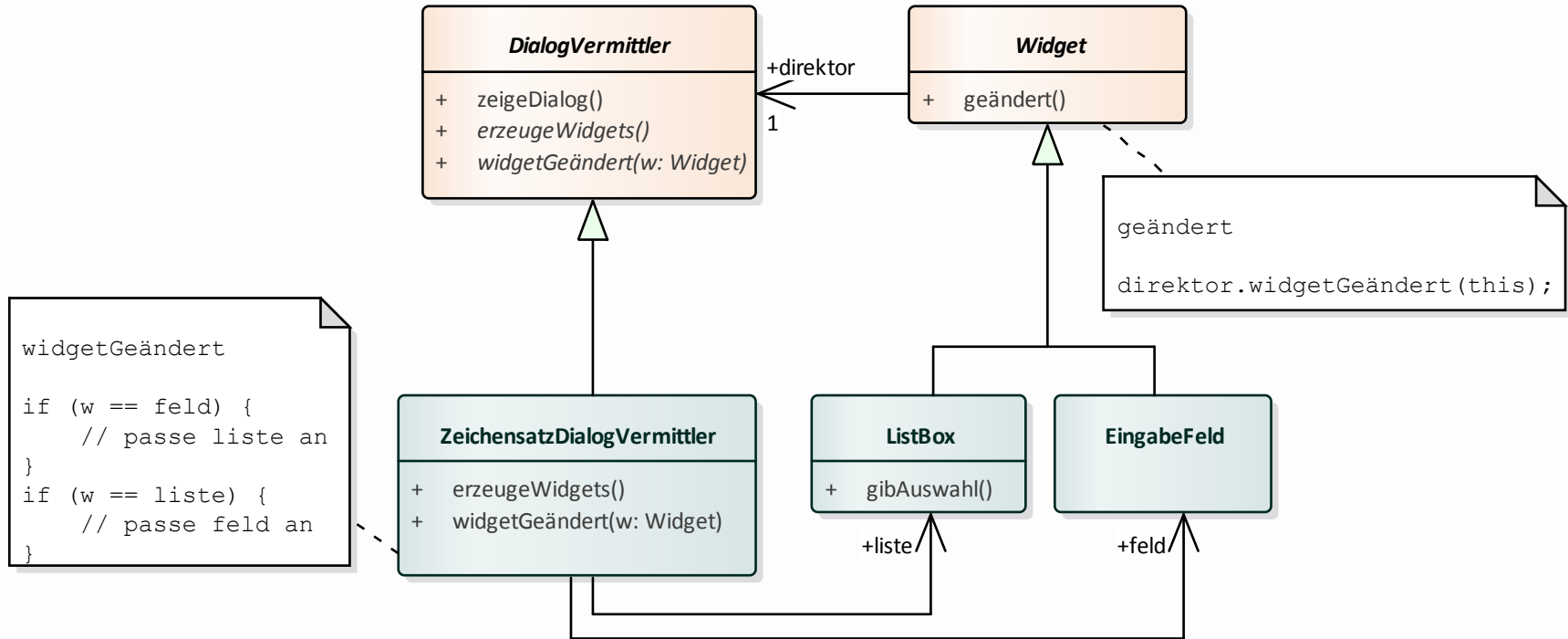
Eingabefeld



ListBox

- Beim Tippen rollt **Auswahlliste** auf
- **Eingabefeld** zeigt ersten Eintrag mit gleichem Anfang.
- Ausgewähltes Element erscheint im Eingabefeld.

Beispiel: Fenster mit Zeichensatz-Dialog



- Wenn eine **Menge** von Objekten vorliegt, die **in wohl definierter, aber komplexer Weise** miteinander zusammenarbeiten.
(Die sich ergebenden Abhängigkeiten sind sonst unstrukturiert und schwer zu verstehen.)
- Wenn die Wiederverwertung eines Objektes schwierig ist, weil es einen **Bezug auf viele andere Objekte** hat und mit ihnen zusammenarbeitet.
- Wenn man ein auf mehrere Klassen **verteiltes Verhalten maßschneidern** soll, ohne viele Unterklassen bilden zu müssen.

Fragen?



Adapter-Pattern



Zweck

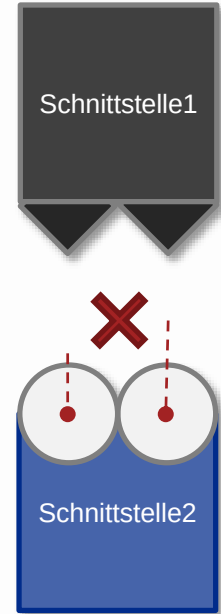
Anpassung von Schnittstellen einer Klasse, sodass Zusammenarbeit möglich wird.

(Für zwei oder mehr Klassen mit inkompatiblen Schnittstellen.)

Adapter

auch:

Umwickler (Wrapper)



Anwendung

- unterschiedliche **Schnittstellen** von Klassen
- **Wiederverwendbarkeit**

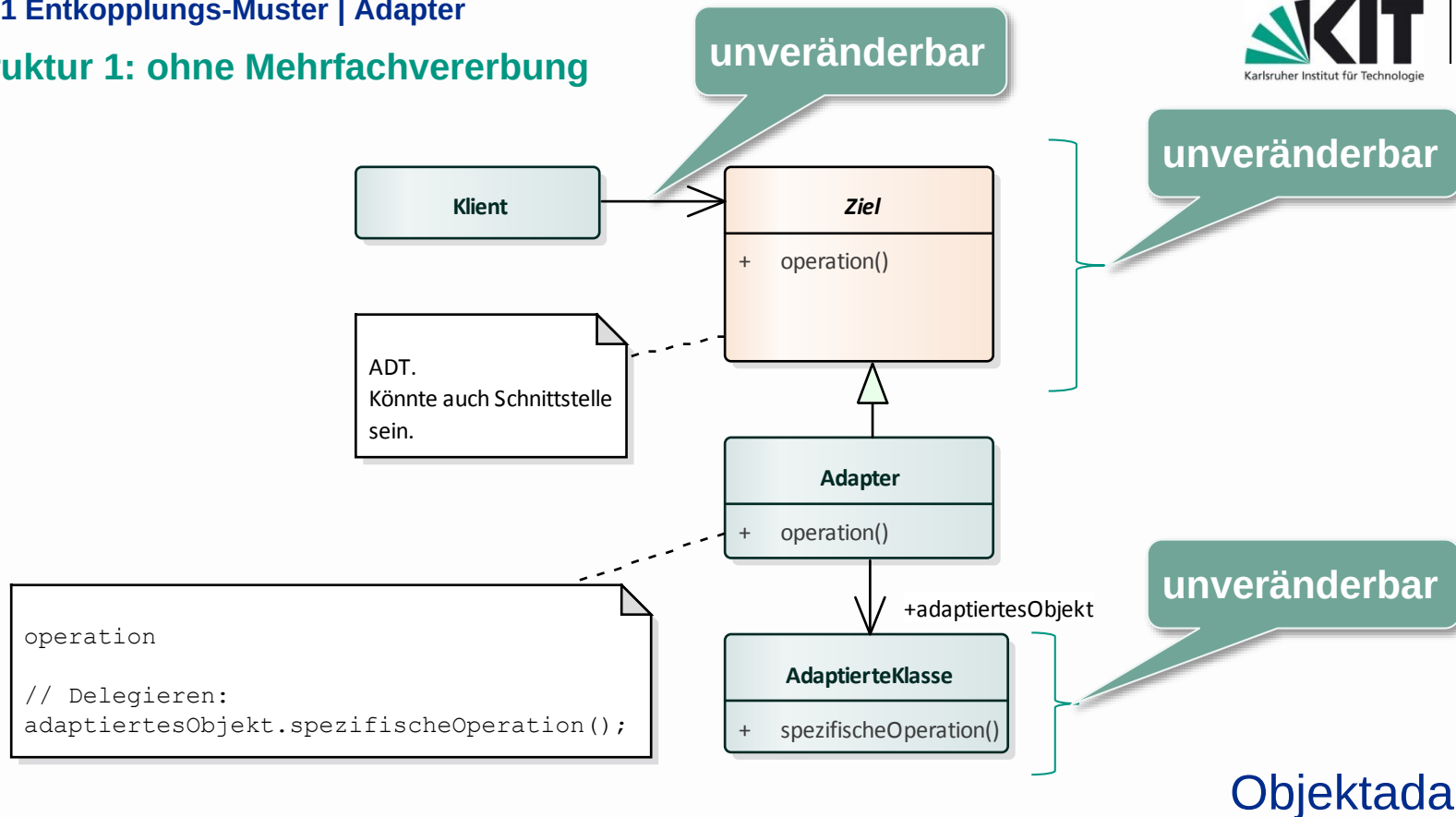
Beispiel

Grafikeditor – Klassen für...

- einfache grafische Formen (Kreis)
- anzupassende ("externe") Klasse für Textausgabe

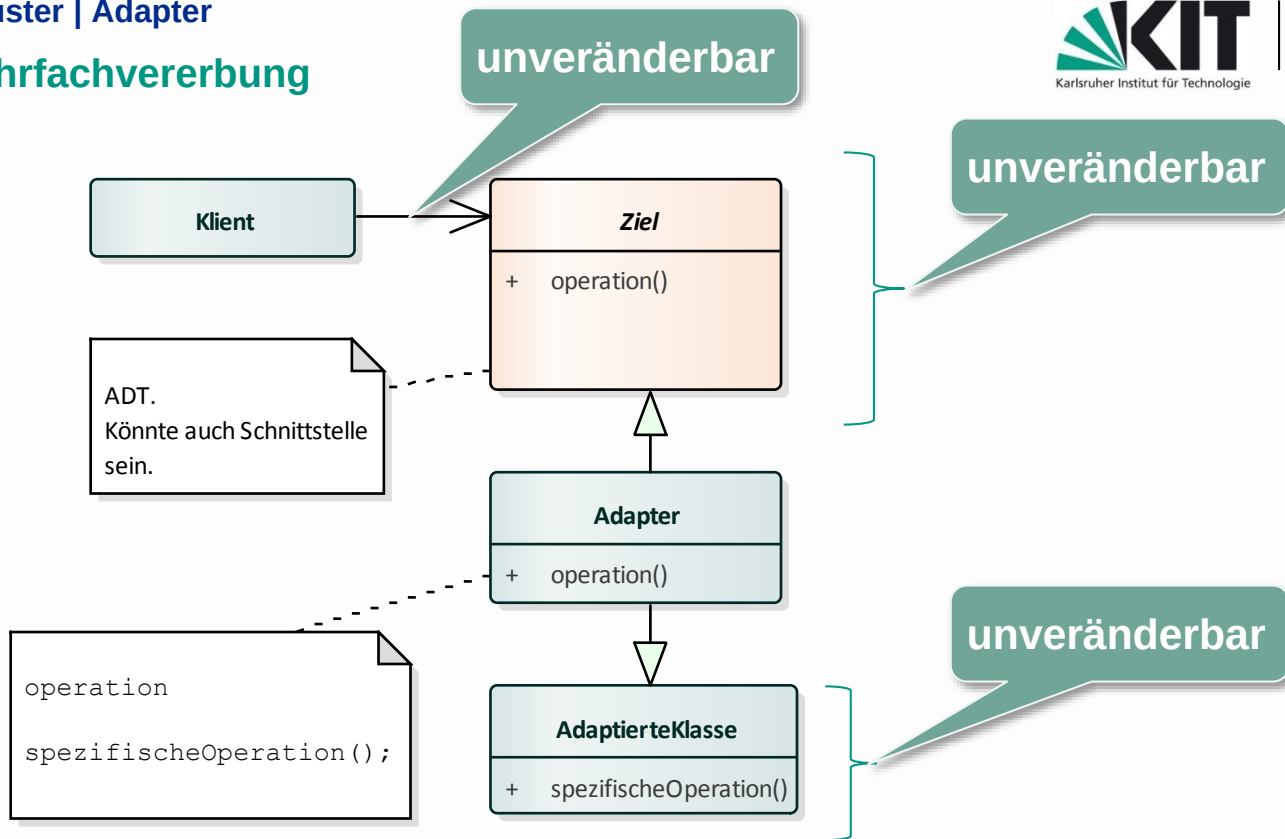


Struktur 1: ohne Mehrfachvererbung



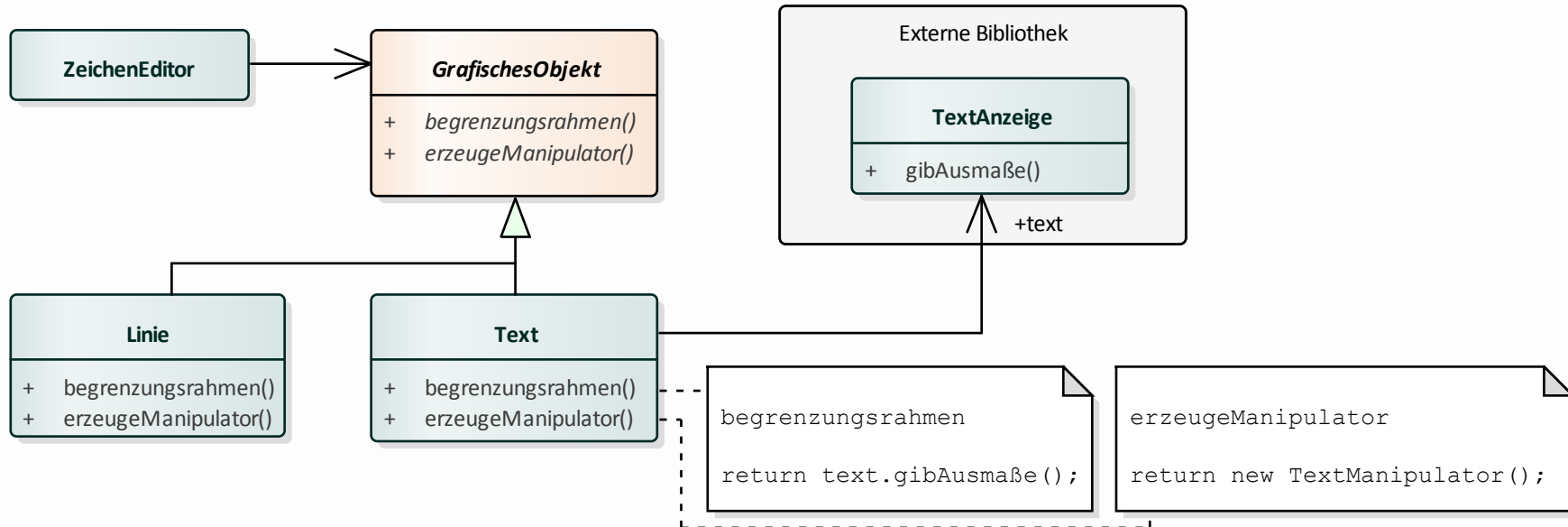
Objektadapter

Struktur 2: mit Mehrfachvererbung



Klassenadapter

Beispiel: Textanzeige



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

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

Fragen?



Stellvertreter (Proxy-Pattern)

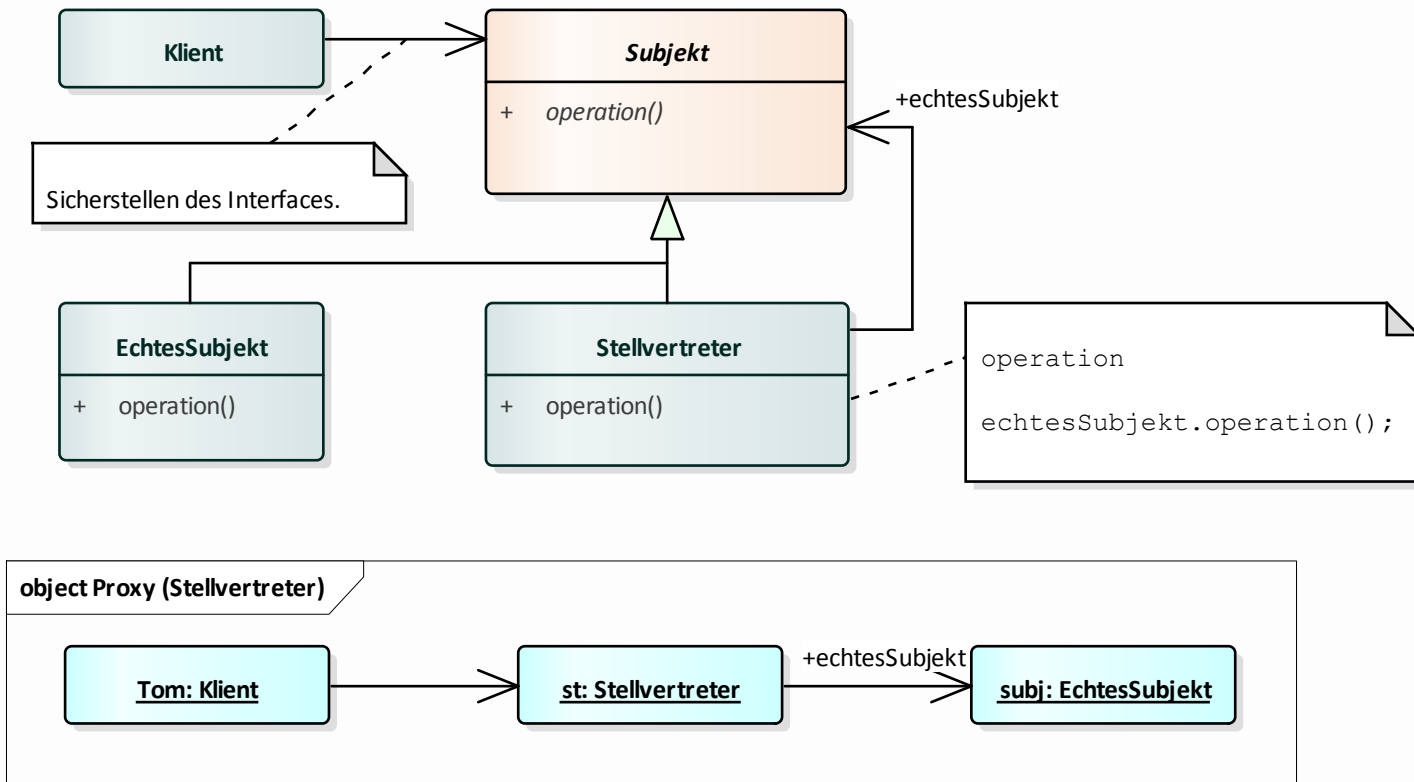
Stellvertreter (Proxy)

auch: Surrogat



Zweck

Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten *Stellvertreter-Objekts*.



Anwendbarkeit (I)

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 Anwendungssituationen:

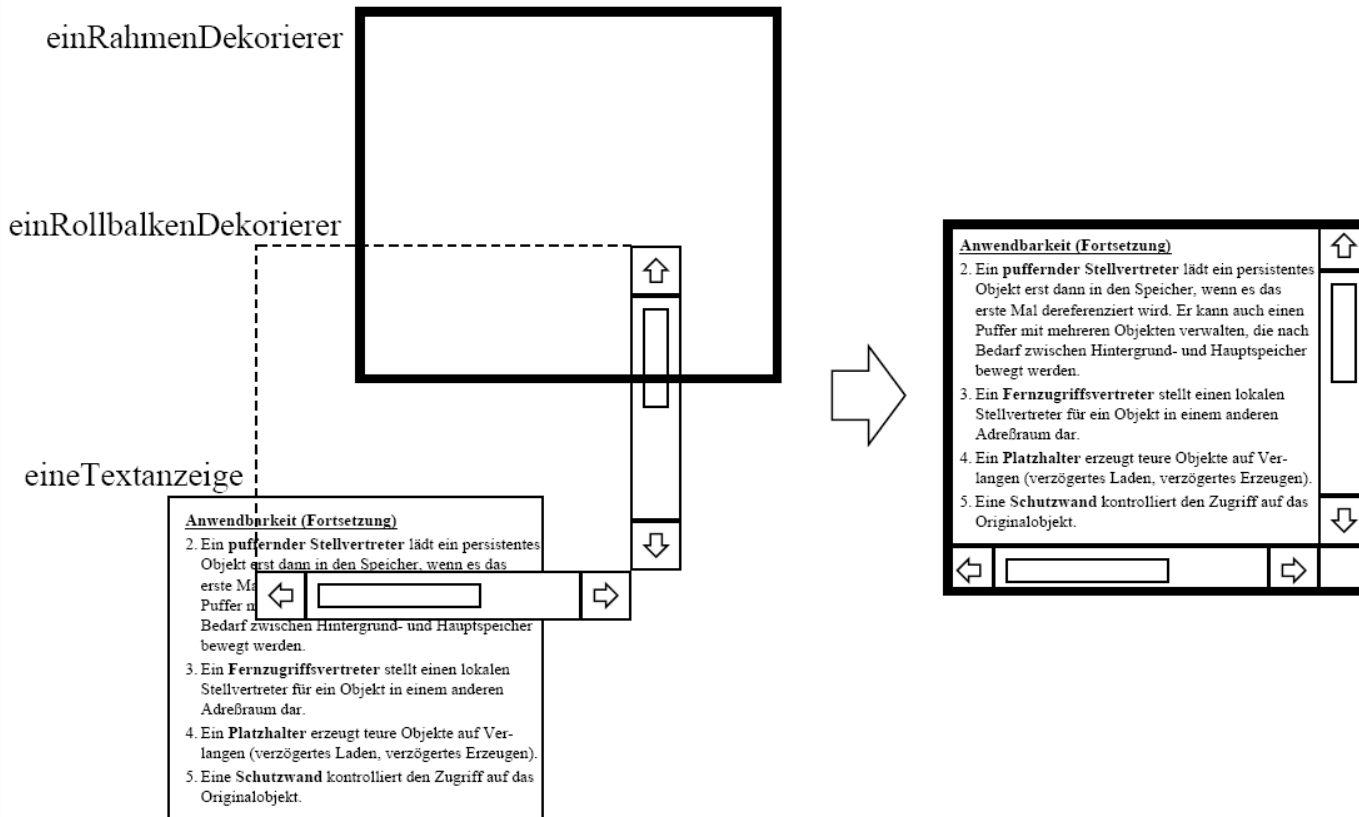
1. Ein **protokollierender Stellvertreter** zählt Referenzen auf das eigentliche Objekt, sodass es automatisch freigegeben werden kann, wenn keine Referenzen mehr auf das Objekt existieren. Er kann auch andere Zugriffsinformationen protokollieren und leitet Zugriffe weiter.
2. Ein **puffernder Stellvertreter** lädt ein persistentes Objekt erst dann in den Speicher, wenn es das erste Mal referenziert wird. Er kann auch einen Puffer mit mehreren Objekten verwalten, die nach Bedarf zwischen Hintergrund- und Hauptspeicher bewegt werden.

Anwendbarkeit (II)

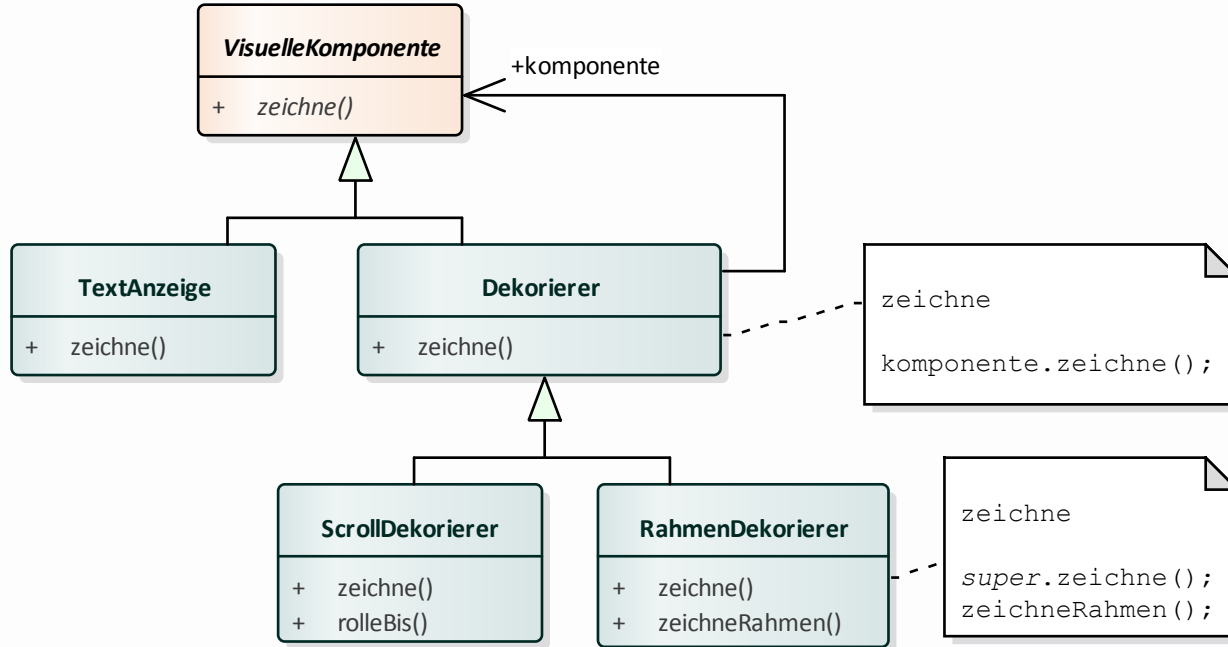
Das Stellvertretermuster ist anwendbar, sobald es den Bedarf nach einer **anpassungsfähigeren und intelligenteren Referenz** auf ein Objekt als einen *einfachen Zeiger* gibt. Es folgen einige verbreitete Anwendungssituationen:

3. Ein **Fernzugriffsvertreter** stellt einen lokalen Stellvertreter für ein Objekt in einem anderen Adressraum dar.
4. Ein **Platzhalter** erzeugt **teure Objekte** auf Verlangen (verzögertes Laden, verzögertes Erzeugen).
5. Eine **Schutzwand** kontrolliert den Zugriff auf das Originalobjekt. Schutzwände sind nützlich, wenn Objekte über verschiedene Zugriffsrechte verfügen sollen.
6. Ein **Dekorierer** fügt zusätzliche **Zuständigkeiten** zu einem bestehenden Objekt hinzu (möglicherweise kaskadiert).

Beispiel: Dekorierer (I)



Beispiel: Dekorierer (II)





Fließband (Pipes and Filters)

Fließband (Pipe and Filters)

auch:

Datenstrom (Data Flow),
Kanäle und Filter



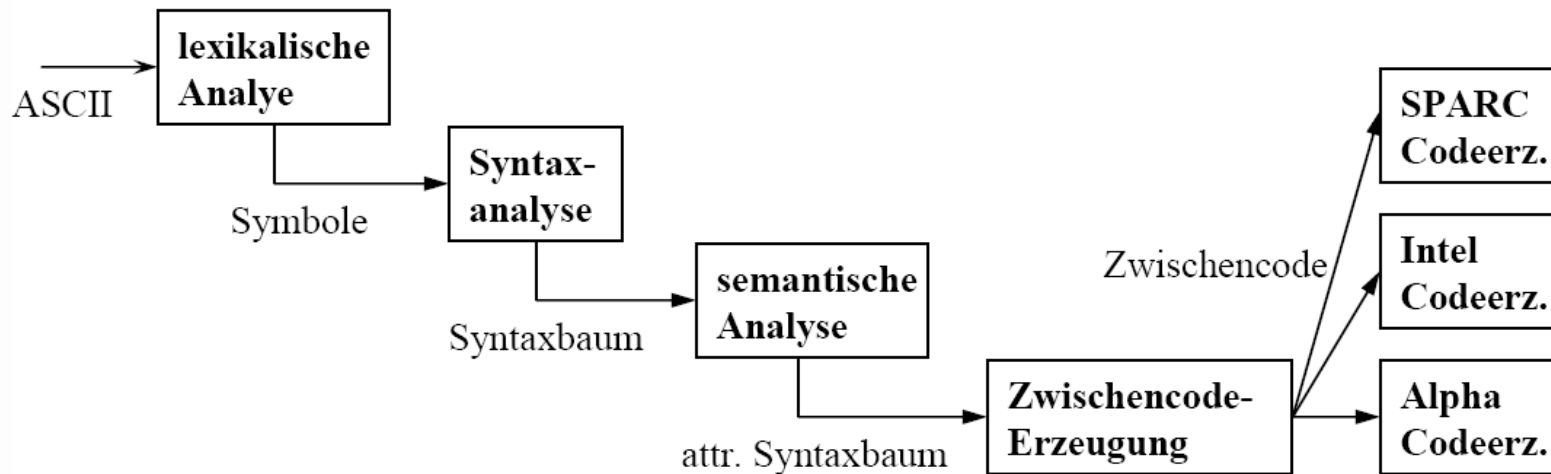
Zweck

Biete eine **Struktur für** Systeme, die **Datenströme** bearbeiten. Jeder Bearbeitungsschritt ist in einer **Filter**-Komponente gekapselt.

Daten werden über **Kanäle** von einem Filter zu einem anderen weitergegeben. Eine Neukombination von Filtern ermöglicht es, **Familien von Systemen** zu erstellen.



Beispiel: Übersetzer



Anwendbarkeit

- Wenn ein System **Datenströme** bearbeiten oder transformieren muss; und ein System bestehend aus nur **einer Komponente unhandhabbar** ist.
- Wenn in zukünftigen Entwicklungen einzelne Komponenten ersetzt oder **Umordnung** der Arbeitsschritte vorgenommen werden sollen.
- Wenn kleinere Komponenten einfacher in anderen Zusammenhängen zur **Wiederverwendung** dienen können.
- Wenn einzelne Komponenten parallel oder quasi-**parallel ablaufen** sollen.
- **Nachteil:** nicht geeignet für **interaktive Systeme**

Beispiel: Reimwörterbuch

Erzeugen eines Reimwörterbuches:

Wörter sind von hinten her sortiert: Wörter, die gleich enden, stehen hintereinander

reversiere < **wörterbuch** | **sort** | **reversiere** > **reimwörterbuch**

wörterbuch	Ein vorhandenes Wörterbuch; ein Wort pro Zeile
sort	Systemsortierungsprogramm von Unix
reversiere	Einfaches Programm zum Umkehren von Einzelzeilen
	Kanal
<	Eingabe von Datei
>	Ausgabe von Datei

Fragen?



Ereigniskanal (Event Channel)

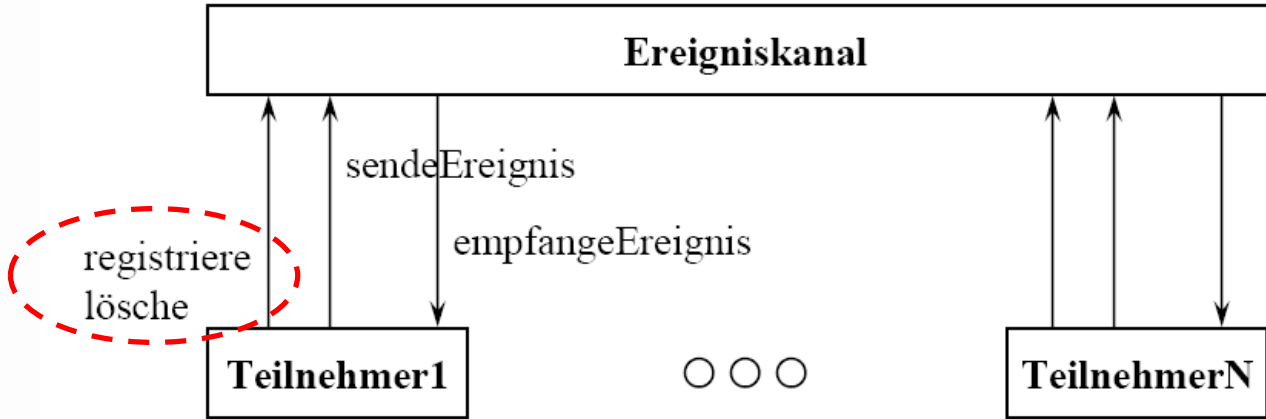


Zweck

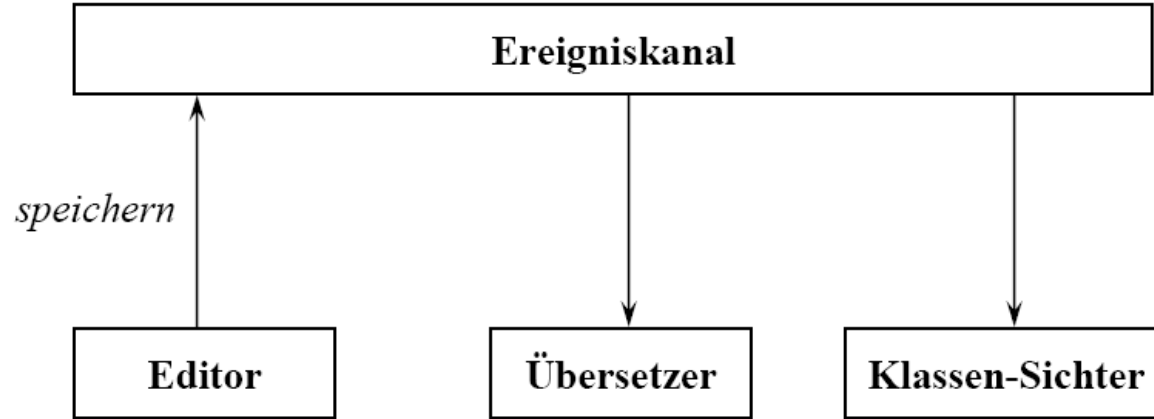
Entkopple **Teilnehmer** an einem Gesamtsystem vollständig voneinander, sodass sie völlig **eigenständig arbeiten** können und über die Existenz oder Anzahl anderer Teilnehmer nichts wissen.

Interaktionen erfolgen über **Ereignisse**.

Ereigniskanal (Event Channel)
auch bekannt als:
Ereignissteuerung (Flow Control)



- Teilnehmer **registrieren sich** am Ereigniskanal, indem sie angeben, bei welchen Ereignissen sie benachrichtigt werden sollen.
- Wenn ein Teilnehmer ein Ereignis (*evtl. mit Daten*) an den **Ereigniskanal** sendet, tätigt dieser die **Weiterleitung** an die dafür registrierten Teilnehmer.



eine Programmierungsumgebung

Wenn der **Editor** eine Datei speichert, wird ein Ereignis (*speichern*) generiert, welches den **Übersetzer** startet und den **Klassen-Sichter** dazu bringt, seine Anzeige zu aktualisieren.

Fragen?



Rahmenarchitektur (Framework)

Rahmenarchitektur (Framework)



Zweck

Biete ein (nahezu) vollständiges Programm, das durch Einfüllen geplanter „**Lücken**“ erweitert werden kann.

Es enthält die **vollständige Anwendungslogik**, meistens sogar ein komplettes Hauptprogramm.

Von einigen der Klassen in dem Programm können Benutzer Unterklassen bilden und dabei Methoden überschreiben oder vordefinierte abstrakte Methoden implementieren.

Das Rahmenprogramm sieht vor, dass die vom Benutzer **gelieferten Erweiterungen richtig aufgerufen** werden.

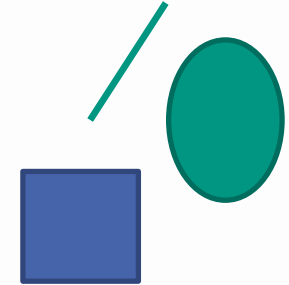
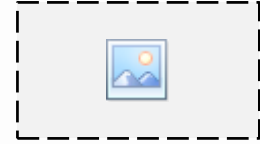


Beispiel: Rahmenprogramm

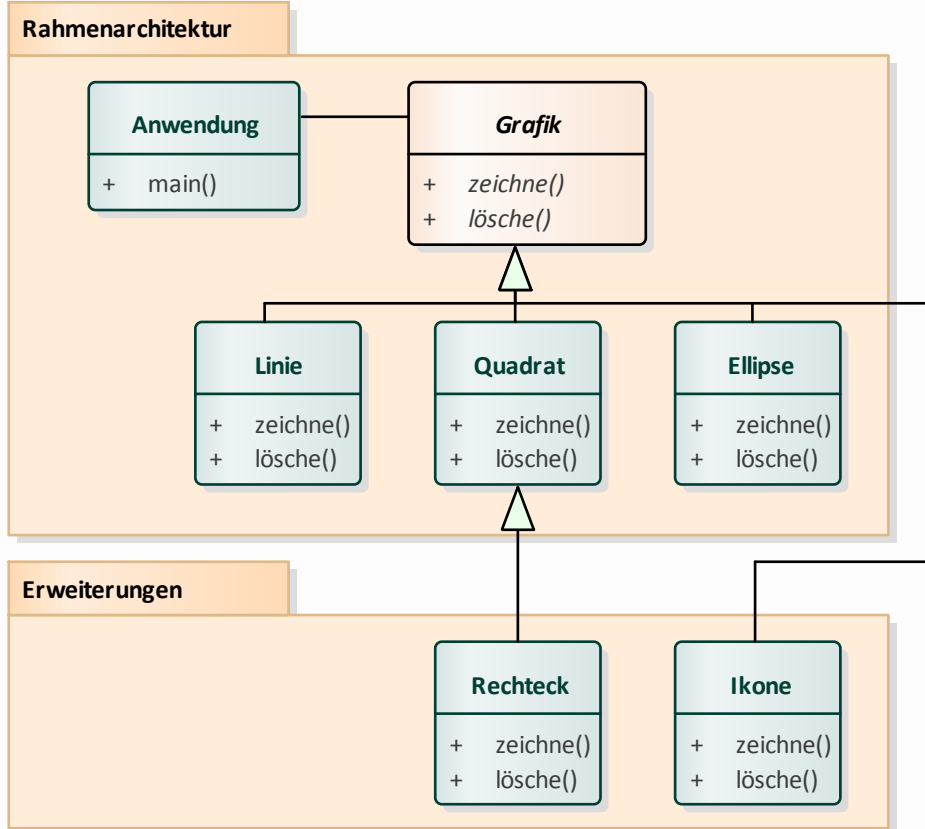
Ein **Zeichen-Rahmenprogramm** sieht eine Klasse Grafik mit einigen Unterklassen vor (z.B. *Linie*, *Quadrat*, *Ellipse*, usw.)

Der Benutzer darf **neue Unterklassen** von Grafik und seinen Unterklassen bilden (z.B. *Rechteck* oder *Ikone*), muss dazu aber eine Methode „**zeichne**“ bereitstellen.

Das Rahmenprogramm sorgt dann dafür, dass Objekte der neuen Klassen **richtig erzeugt**, auf dem Zeichenbrett positioniert, verschoben, gesichert, usw. werden können.

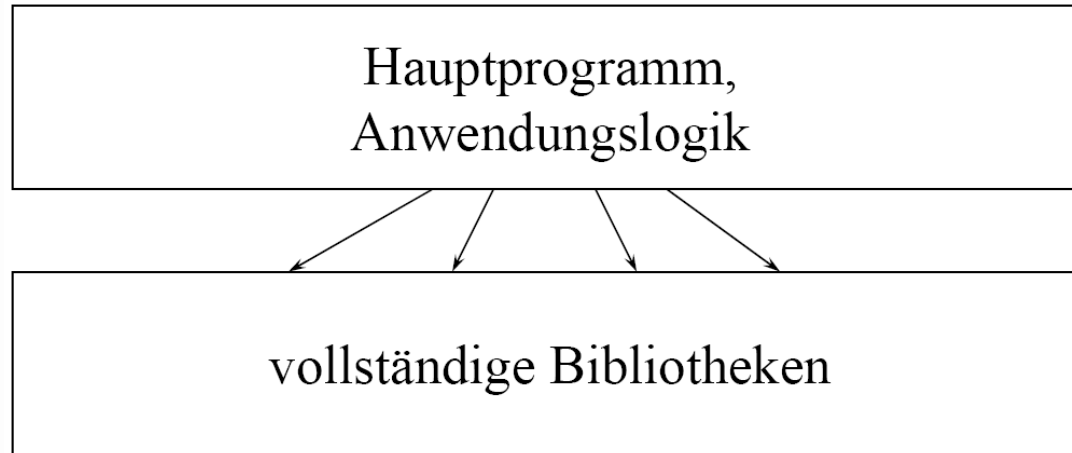


Beispiel: Zeichen-Rahmenarchitektur



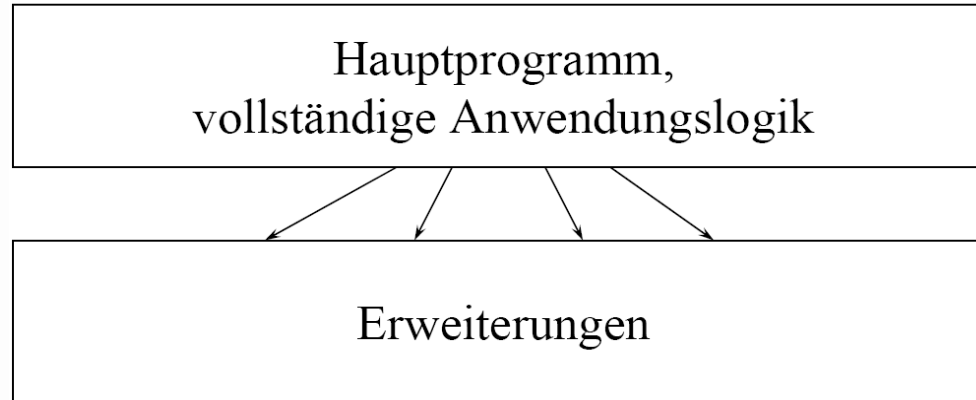
Struktur: Herkömmliche Systemstruktur

- Hersteller liefert Bibliotheken
- Benutzer schreibt Hauptprogramm und Anwendungslogik



Struktur: Hollywood-Prinzip

Ein Rahmenprogramm befolgt das „Hollywood-Prinzip“:
»Don't call us – we'll call you«



→ Das **Hauptprogramm besteht bereits**;
und ruft die Erweiterungen der Benutzer auf.

Anwendbarkeit

- Wenn eine **Grundversion** der Anwendung schon **funktionsfähig** sein soll.
- Wenn Erweiterungen möglich sein sollen, die sich **konsistent verhalten** (Anwendungslogik im Rahmenprogramm).
- Wenn **komplexe Anwendungslogik** nicht neu programmiert werden soll.

Hinweis

Die Muster *Fabrikmethode*, *abstrakte Fabrik* und *Schablonenmethode* werden häufig in Rahmenprogrammen benötigt.

6.3.2. Varianten-Muster (B)

Inhalt

- Oberklasse
- Kompositum (Composite)
- Strategie
- Besucher
- Schablonenmethode
- Fabrikmethode
- Erbauer
- Abstrakte Fabrik

1. **Gemeinsamkeiten** von verwandten Einheiten **werden** aus ihnen herausgezogen und an einer einzigen Stelle **beschrieben**.
2. Aufgrund ihrer Gemeinsamkeiten können unterschiedliche Komponenten im gleichen Programm **einheitlich verwendet** werden, und **Code-Wiederholungen werden vermieden**.

Oberklasse



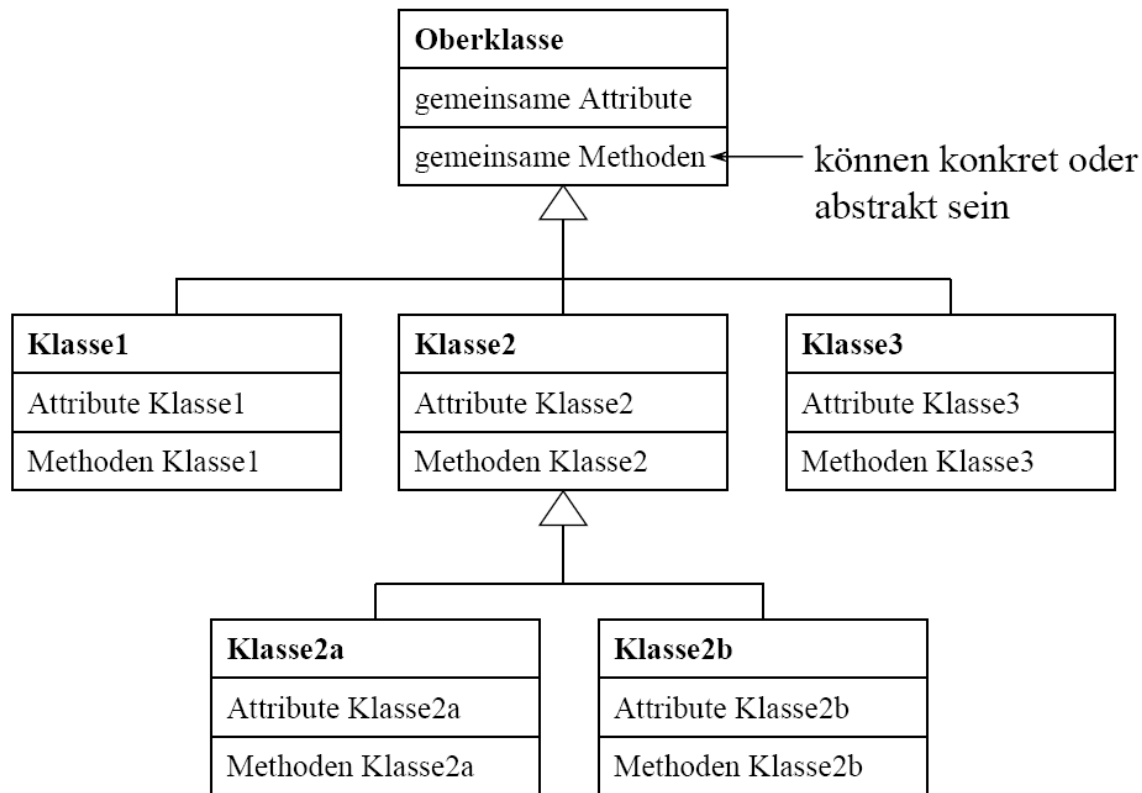
Zweck

Einheitliche Behandlung von Objekten, die unterschiedlichen Klassen angehören, aber gemeinsame Attribute oder Methoden besitzen.

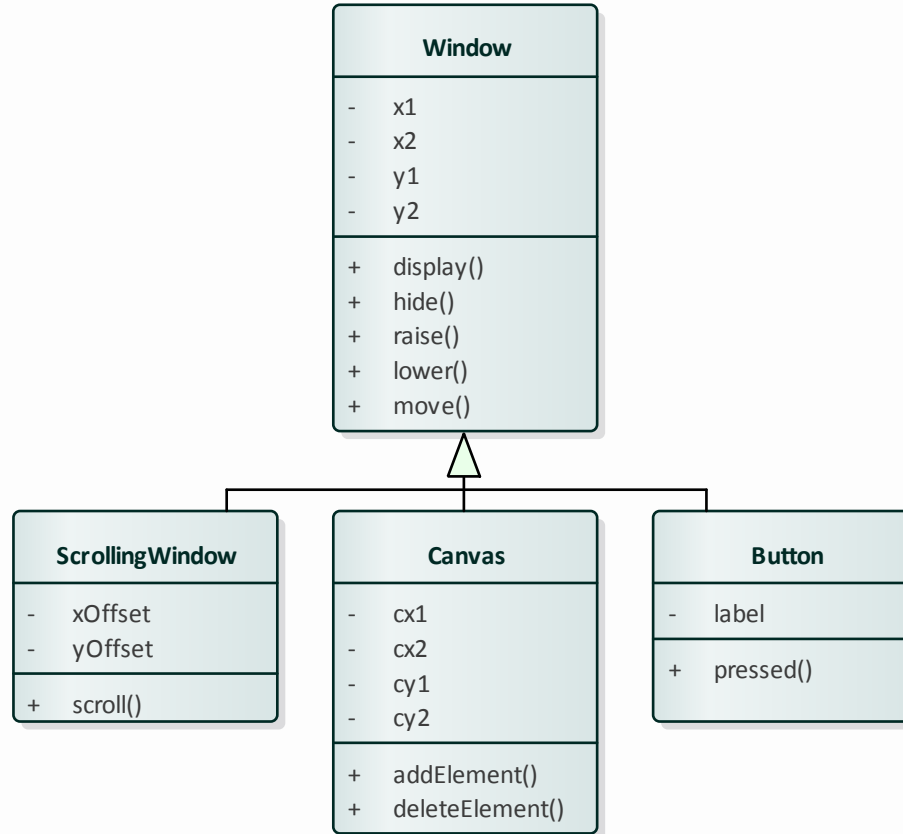
Oberklasse

auch:

Superklasse (Superclass),
Basisklasse (Base Class),
Elternklasse (Parent Class)



Beispiel einer Oberklasse



```
Window w;  
Button b;  
Canvas c;  
w = c;  
w.display();
```

- Wenn Objekte verschiedener Klassen **gemeinsame Attribute, Methoden oder Schnittstellen** haben.
- Wenn Objekte verschiedener Klassen **einheitlich** in einem Programm behandelt werden sollen.
- Wenn es möglich sein soll, weitere **Klassen hinzuzufügen**, ohne den bestehenden Quelltext zu verändern.

Kompositum (Composite)

Kompositum (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**.

Kompositum (Composite)

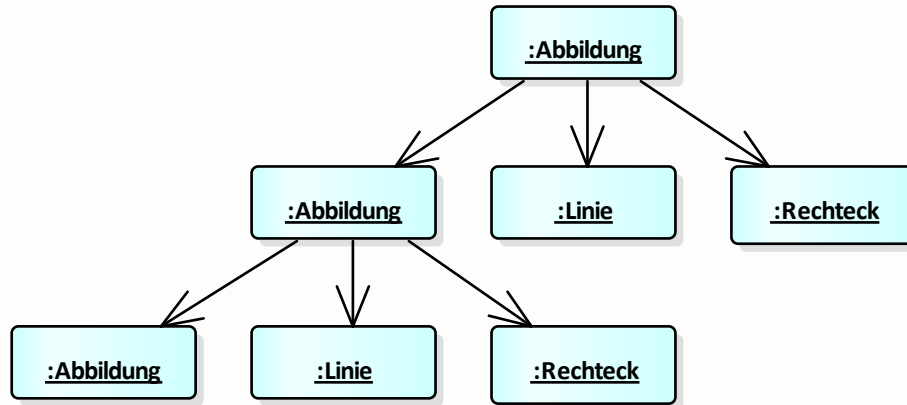
Motivation

- Bestands-Hierarchien treten überall dort auf, wo **komplexe Objekte** modelliert werden, wie beispielsweise Datei-Systeme, 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**.

Beispiel: Grafik-Objekte

Zusammengefügte Grafik-Objekte



gemeinsame Operationen

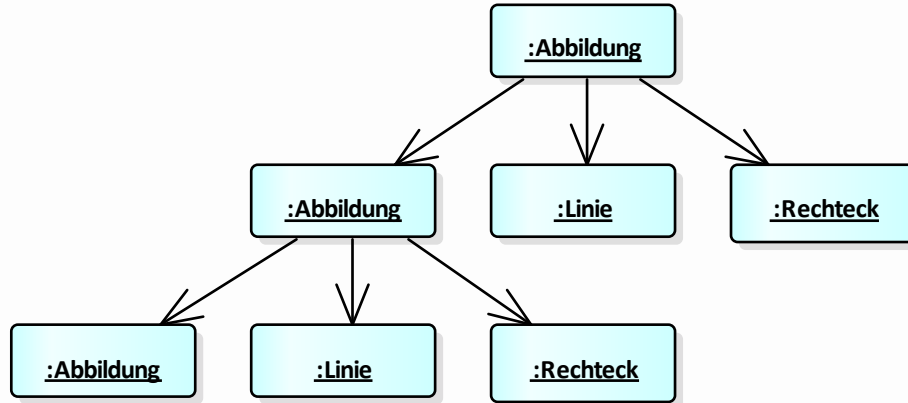
zeichne()

verschiebe()

lösche()

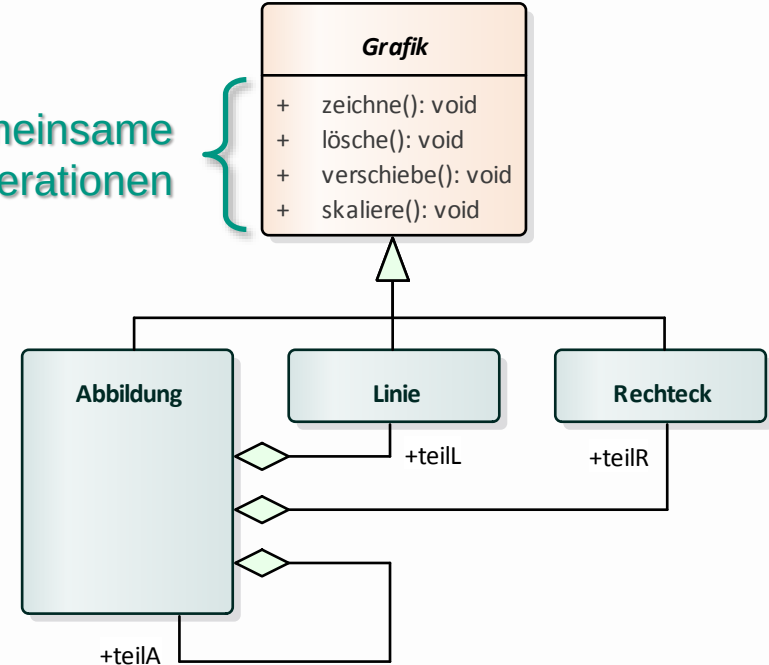
skaliere()

Beispiel: Grafik-Objekte



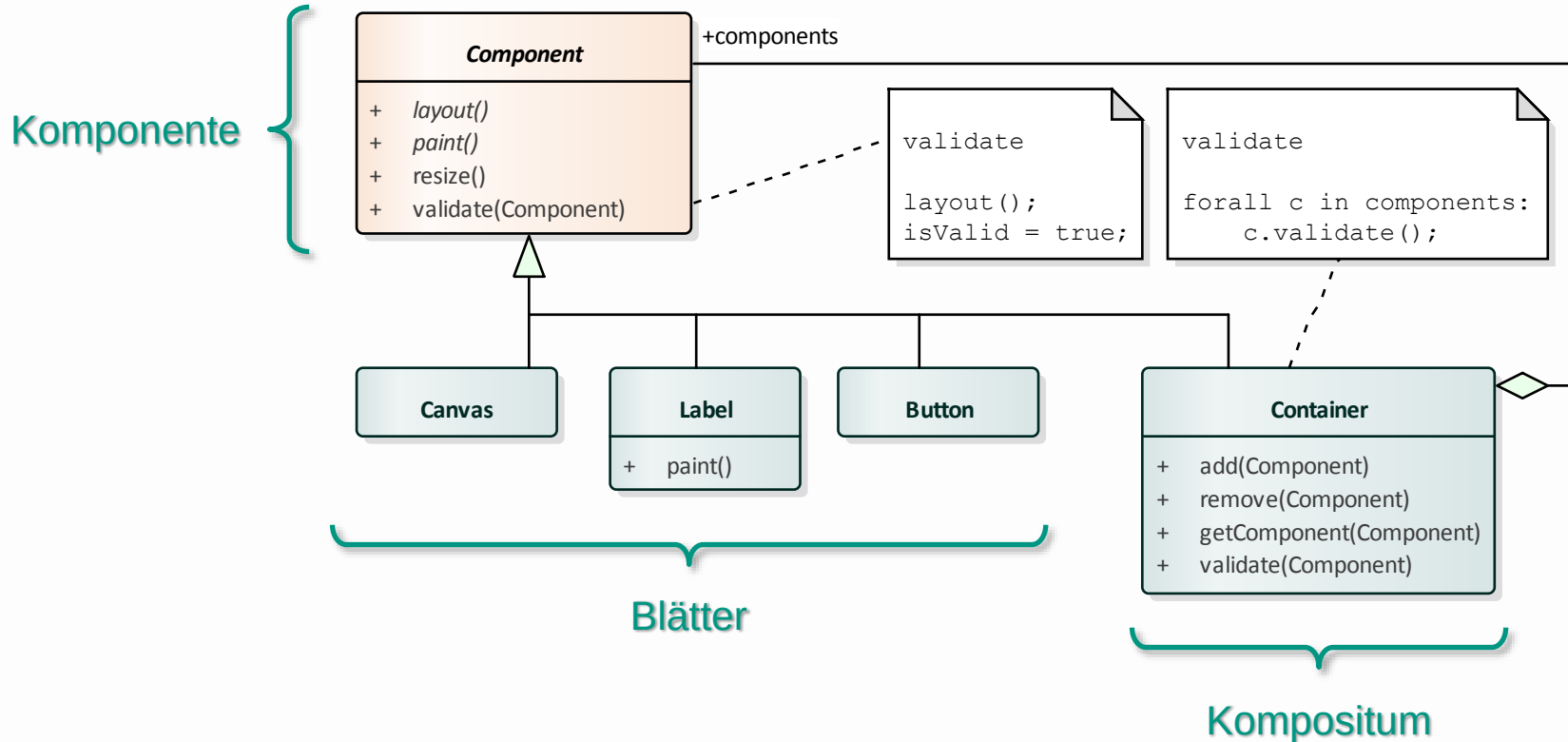
Zusammengefügte Grafik-Objekte

Gemeinsame
Operationen

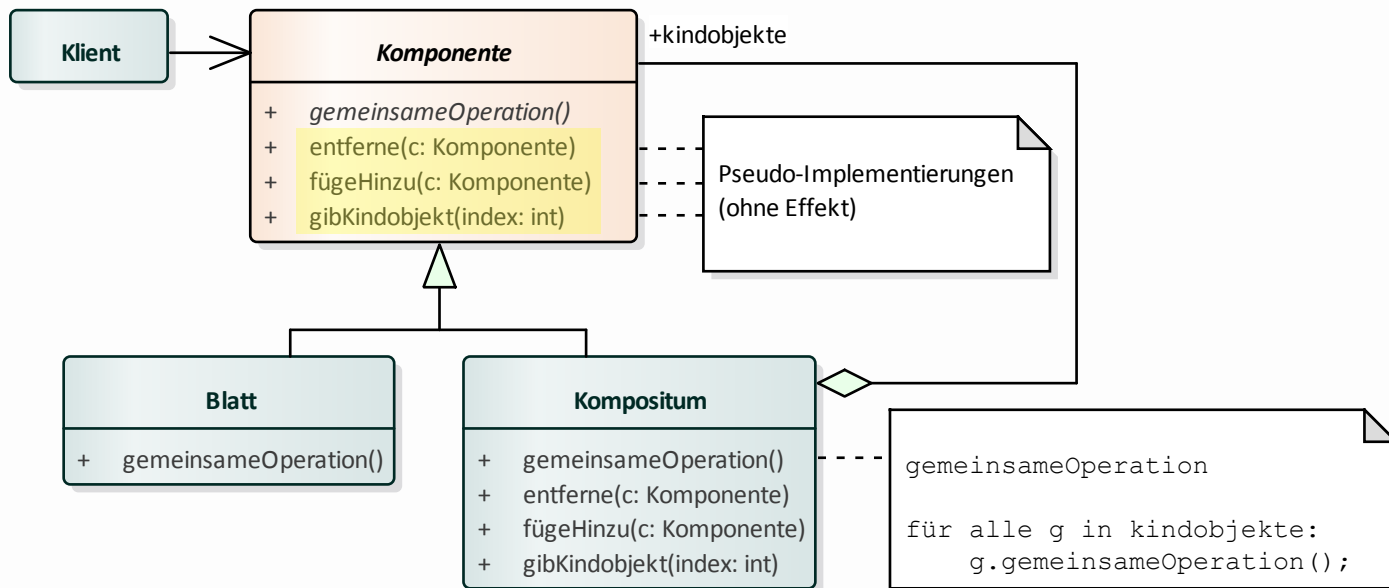


Grafik Klassehierarchie

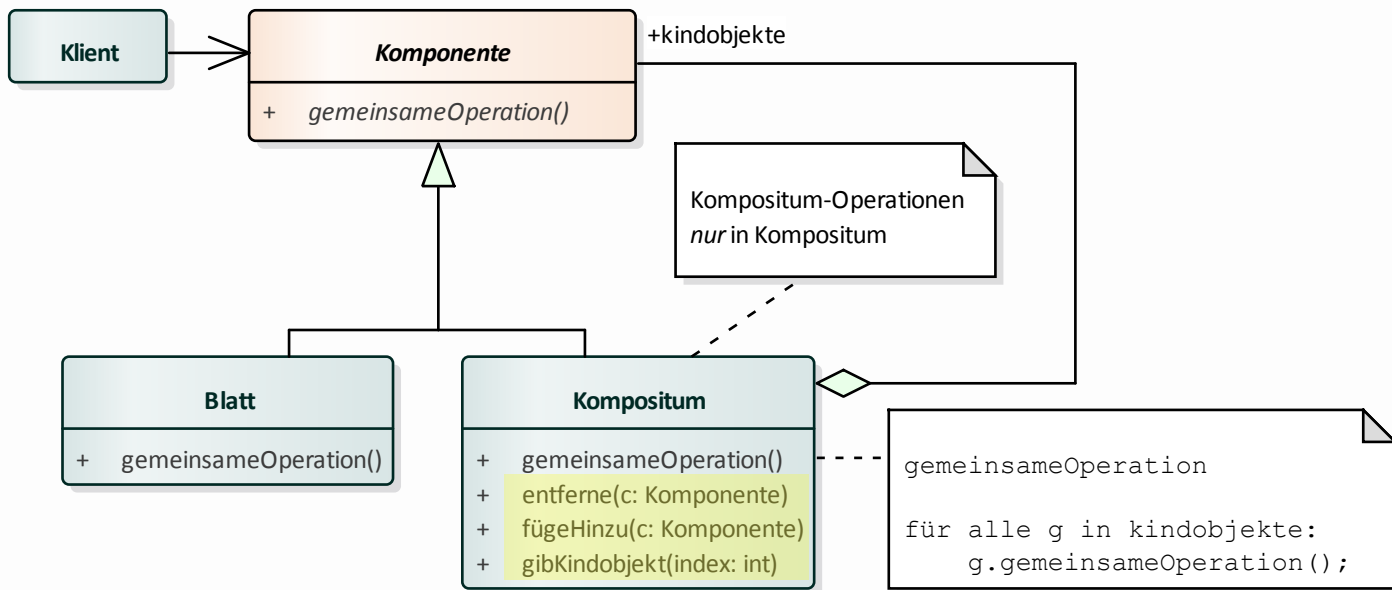
Beispiel: JAVA Abstract Window Toolkit (AWT)



Kompositum-Operationen in der Komponente

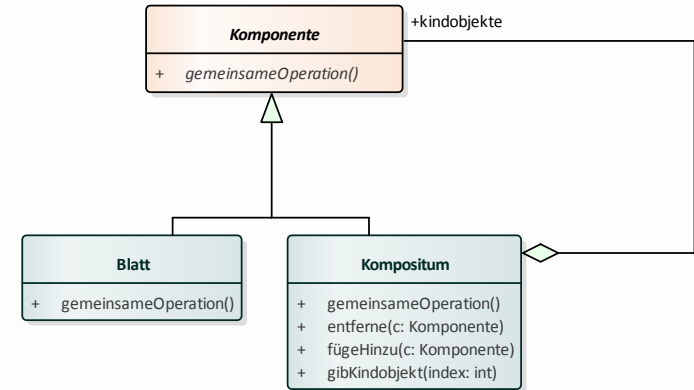
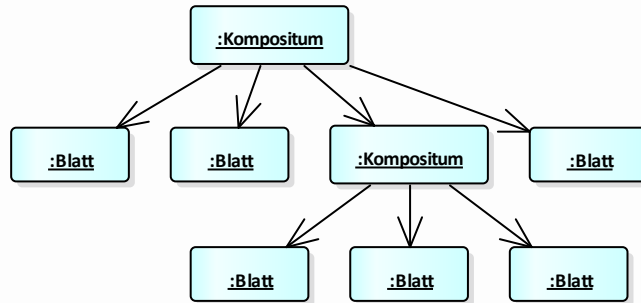


Kompositum-Operationen im Kompositum



Struktur, 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**.

Strategie (Strategy-Pattern)



Zweck

Definiere eine **Familie von Algorithmen**, kapsle sie und mache sie austauschbar.

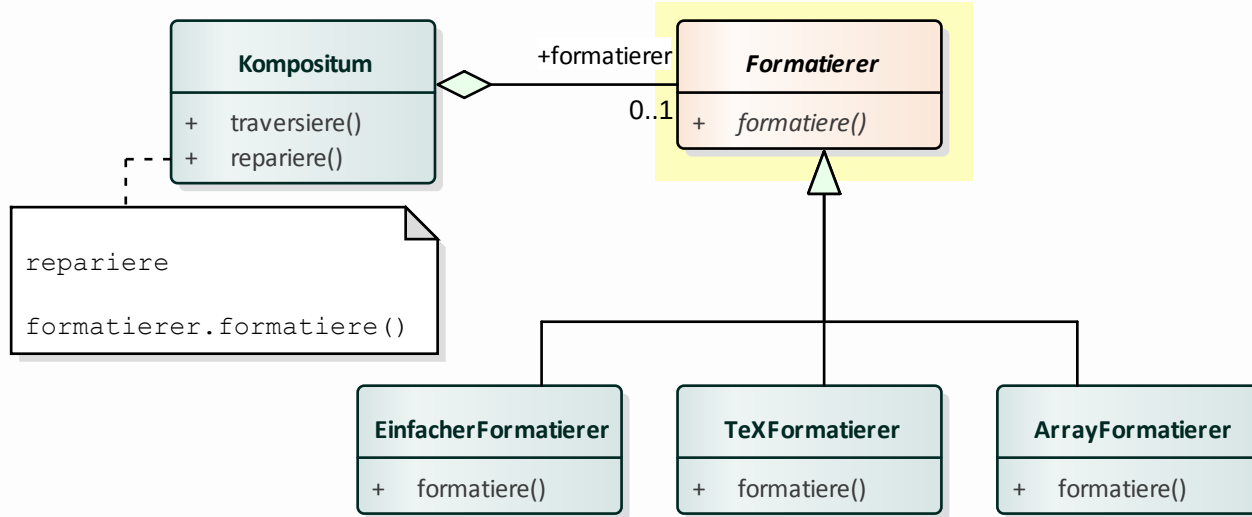
Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von nutzenden Klienten zu variieren.

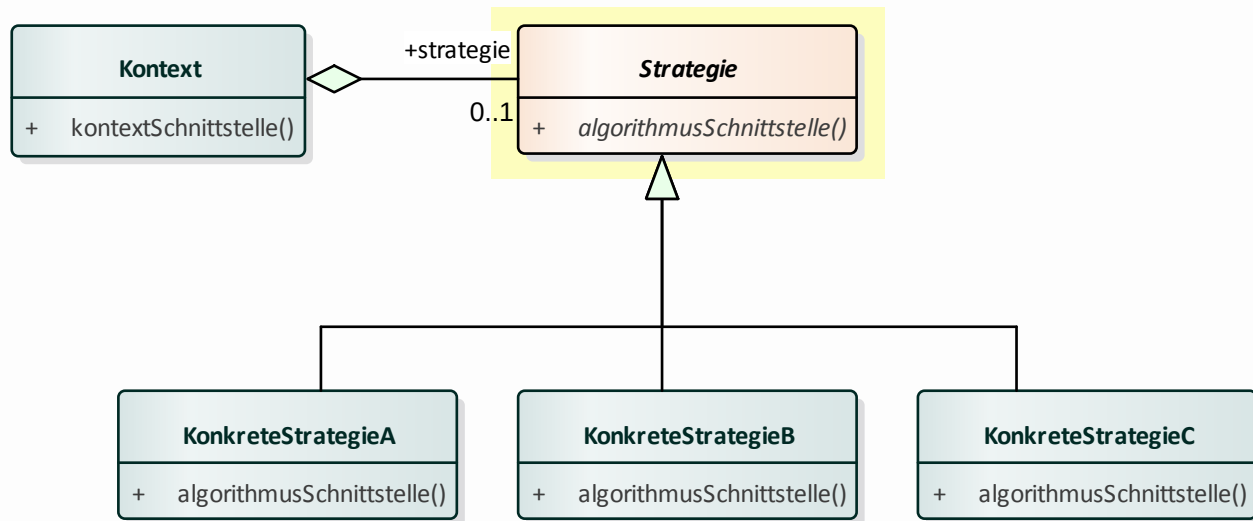
Motivation

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

Strategie (Strategy)
auch bekannt als:
Policy

Kapselung von Zeilenumbruch-Algorithmen in Klassen



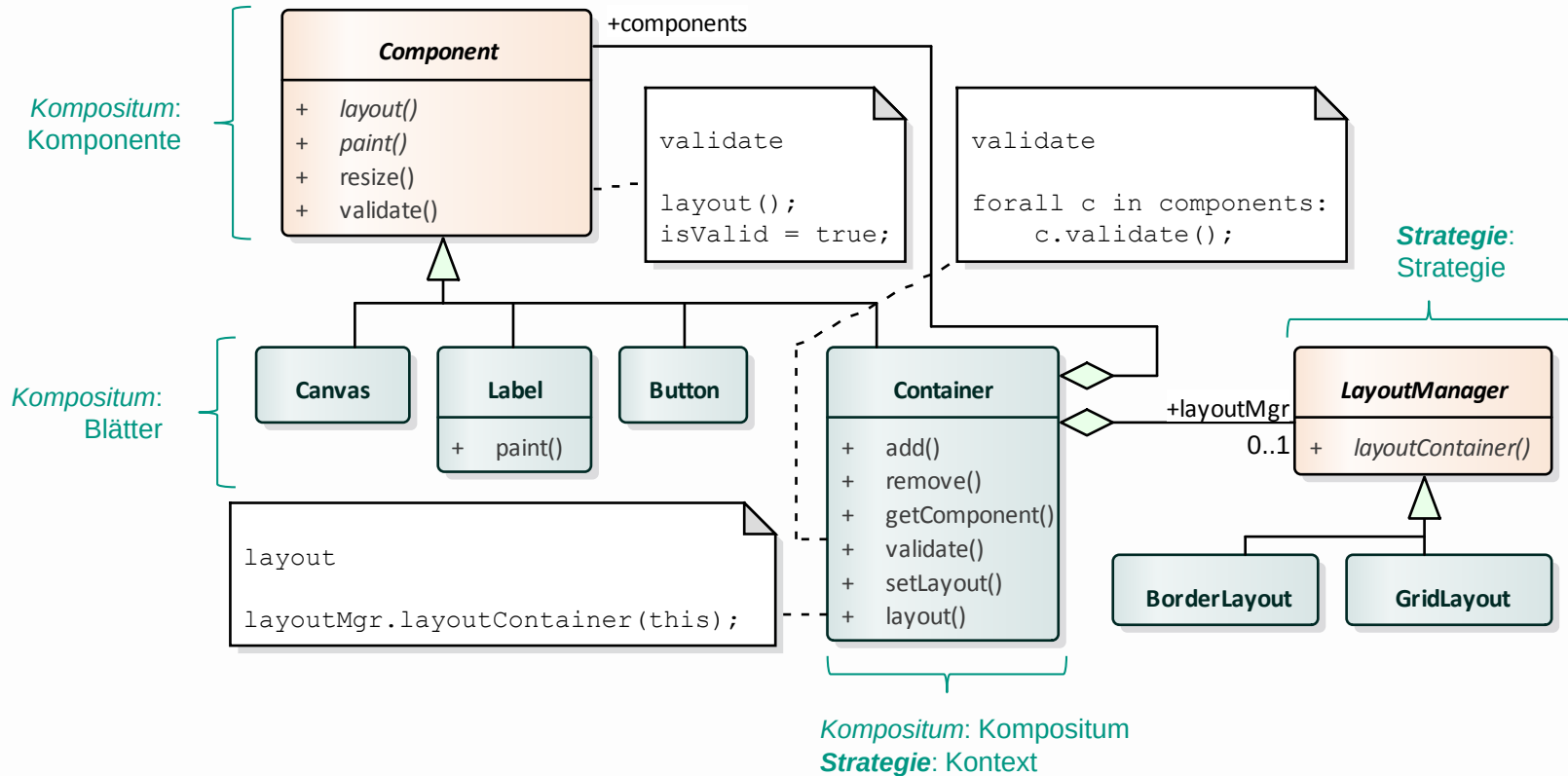


Anwendbarkeit (I)

- 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 Bedingungsanweisungen** in ihren Operationen erscheinen.
- Alternativ zur Ableitung der Klasse Strategie kann 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.

Beispiel: aus JAVA AWT



Strategie vs Brücke

	Brücke	Strategie
Gemeinsamkeit	Vererbung nutzen für mehr Flexibilität: Container-Klasse erhält einen Verweis auf eine andere Klasse mit gewisser Grundstruktur.	
Typ	Struktur (Aufbau)	Verhalten (einmalige Bindung)
Ziel	Entkopplung: Die Abstraktion und <i>ihre</i> Implementierung sollen je unabhängig voneinander veränderbar sein.	Interface: Der Abstraktion sollen verschiedene Implementierungen geboten werden.
Zweck	Code-Wartung: Zukünftige Änderungen (etwa für Features) werden erleichtert. „Die Klassen sind jenseits der Brücke getrennt.“	Auswahl: Anstatt switch -Anweisung. Das Aufgabenziel (was) bleibt gleich, die Strategie (wie) ist variabel.
Container	Dient als Adapter für das Interface.	Soll sich je nach Klient die geeignetste Implementierung aussuchen.
Implementierung	Eine bestimmte Idee innerhalb einer eigenen Vererbungshierarchie	Ein Kandidat aus einer Familie von Algorithmen, der je nach Klient ausgewählt wird.

Besucher (Visitor)

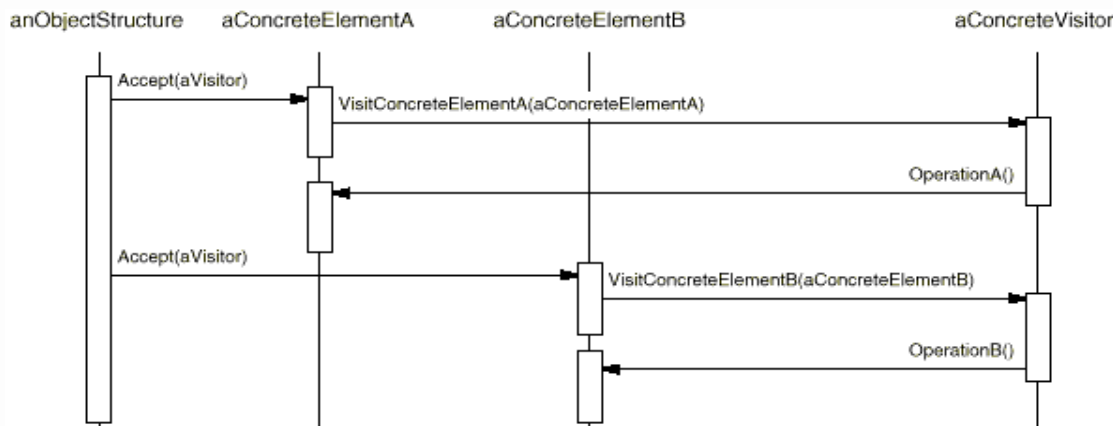
Besucher (Visitor)



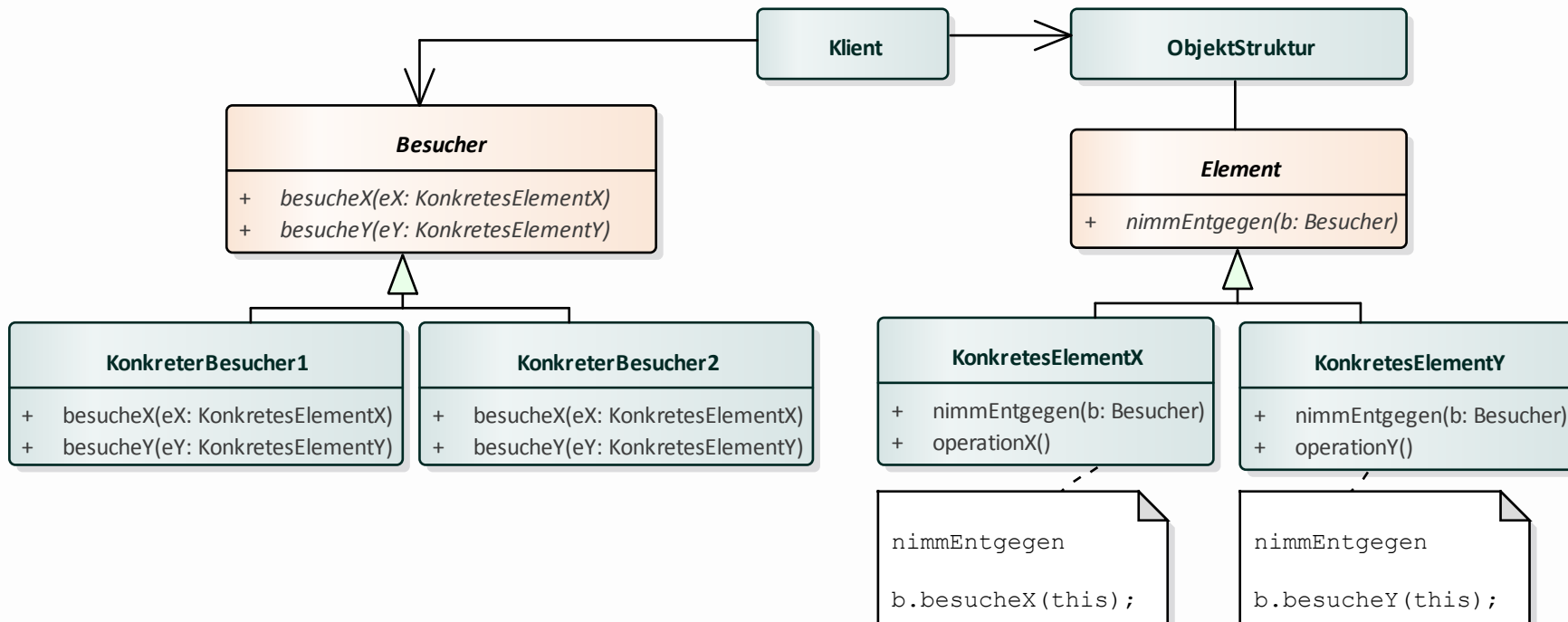
Zweck

Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt.

Das *Besuchermuster* ermöglicht die Definition einer neuer Operation, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.



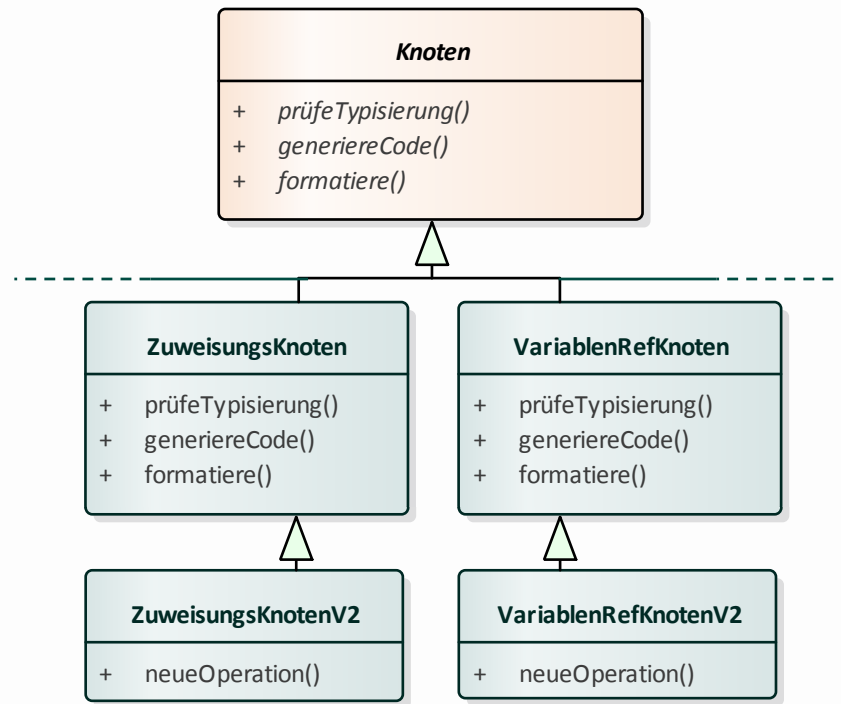
Struktur



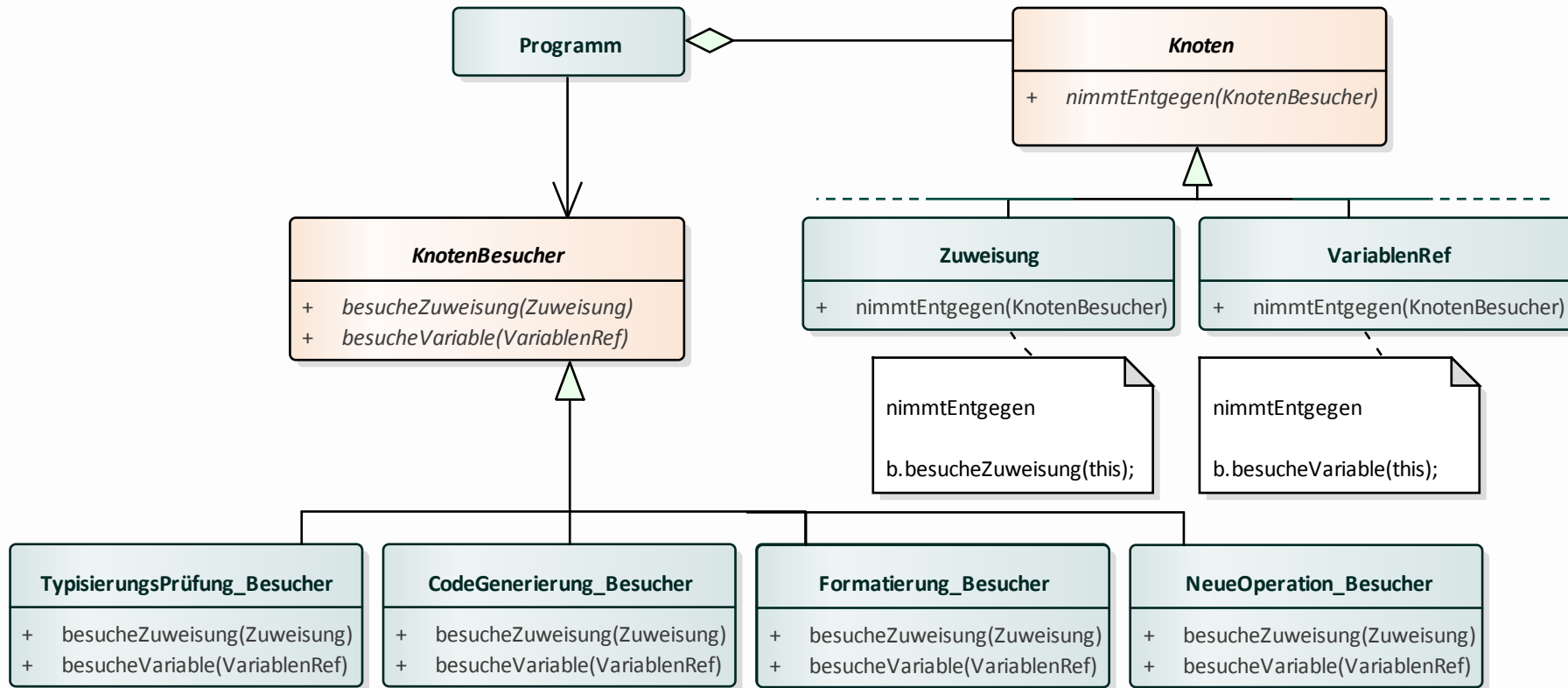
Beispiel: *ohne* Besucher

Abstrakte Syntaxbäume in einem Übersetzer

- Die einzelnen Operationen sind **über viele Klassen verstreut**.
- Bei Einführung neuer Operationen müssen diese **Klassen allesamt erweitert** werden.



Beispiel: mit Besucher



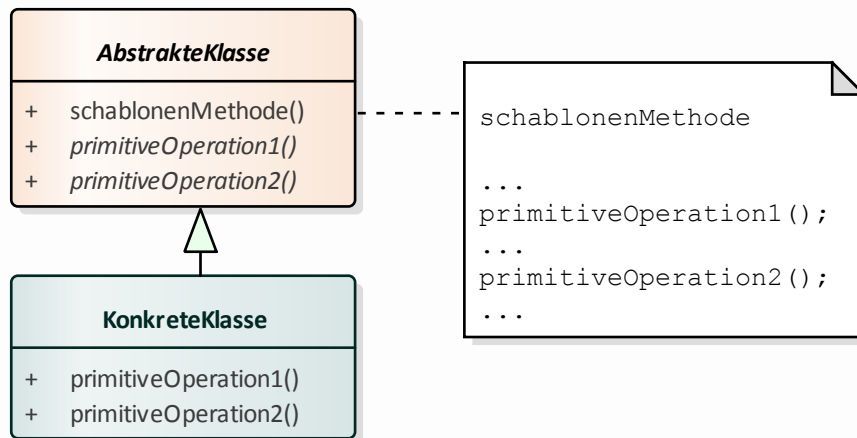
- Wenn eine Objektstruktur viele Klassen von Objekten mit **unterschiedlichen Schnittstellen** enthält und Operationen auf diesen Objekten ausgeführt werden sollen, die von ihren konkreten Klassen abhängen.
- Wenn viele unterschiedliche und **nicht miteinander verwandte Operationen** auf den Objekten 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.



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.



Anwendbarkeit

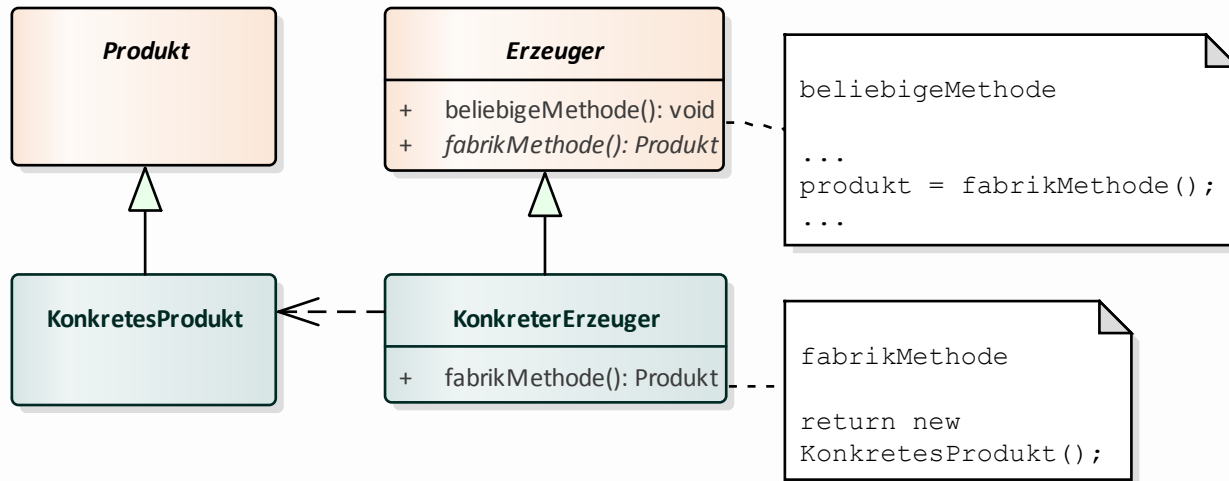
- Um die **invarianten Teile** eines Algorithmus genau einmal festzulegen und es dann Unterklassen zu überlassen, das variierende Verhalten zu implementieren.
- Wenn gemeinsames Verhalten aus Unterklassen herausfaktoriert und in einer allgemeinen Klasse platziert werden soll, um die **Verdopplung von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



Zweck

Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber **lasse Unterklassen entscheiden**, von welcher Klasse das zu erzeugende Objekt ist.

*Fabrikmethode*n ermöglichen es einer Klasse, die **Erzeugung von Objekten** an Unterklassen zu **delegieren**.



Beispiel: "Archi" (I)

„Archi“ – ein Rahmenprogramm zur Gebäudeplanung

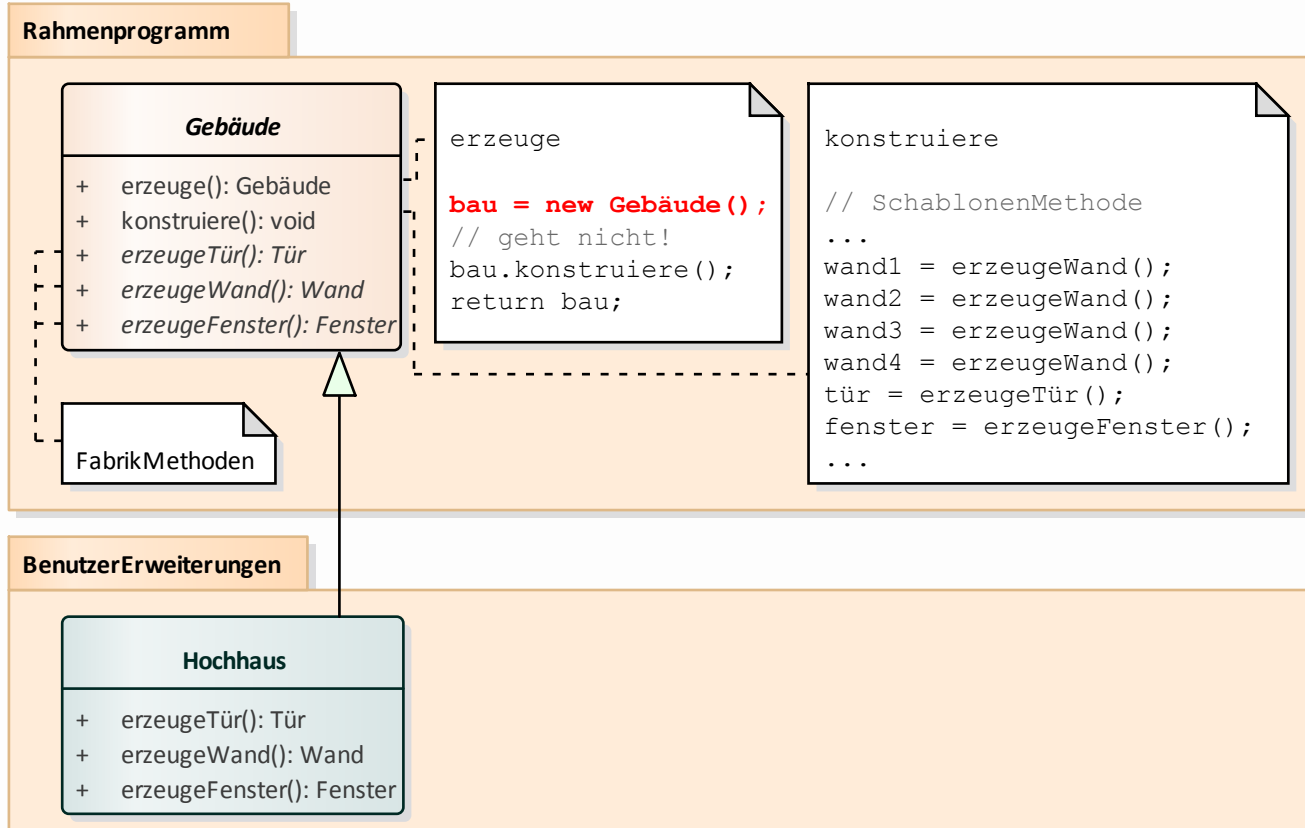
Archi enthält die Klasse **Gebäude**, die eine Methode **konstruiere** enthält.

- Diese **Schablonenmethode** zum Konfigurieren eines Gebäudeobjektes delegiert das Erzeugen der benötigten Komponenten an die Fabrikmethoden.
- Dem Benutzer des Gebäudes ist es gestattet, **Unterklassen** von Gebäude anzulegen (z.B. **Hochhaus**).
- Diese Unterklassen müssen Implementierungen für die abstrakten Methoden **erzeugeWand**, **erzeugeTür** und **erzeugeFenster** enthalten.

Problem

Wie kann der Benutzer dem Rahmenprogramm mitteilen,
wie seine Unterklasse heißt?

Beispiel: "Archi" (II)



Beispiel: "Archi" (III)

Lösung 1: Übergebe den Klassennamen **Hochhaus** als Zeichenkette und wandle ihn in die entsprechende Klasse um.

in Java: `Class.forName(String name)`

```
public static Gebäude erzeuge() {  
    Gebäude bau;  
    String KlassenName = holeKlassenname();  
    bau = (Gebäude) Class.forName(KlassenName).newInstance();  
    bau.konstruiere();  
    return bau;  
}
```

Definition
zur Laufzeit

liest z.B. eine
Konfigurationsdatei
oder nutzt einen
Aufrufparameter

Beispiel: "Archi" (IV)

Lösung 2: Gib Unterklasse von Gebäude im Rahmenprogramm vor, und parametrisiere **konstruiere()** entsprechend.

```
public static Gebäude erzeuge (Gebäudetyp t) {  
    Gebäude bau;  
    switch (t) {  
        case HOCHHAUS: bau = new Hochhaus(); break;  
        case BANK: bau = new Bank(); break;  
        case WOHNHAUS: bau = new Wohnhaus(); break;  
    }  
    bau.konstruiere();  
    return bau;  
}
```

Nachteil:
Unterklassen fixiert

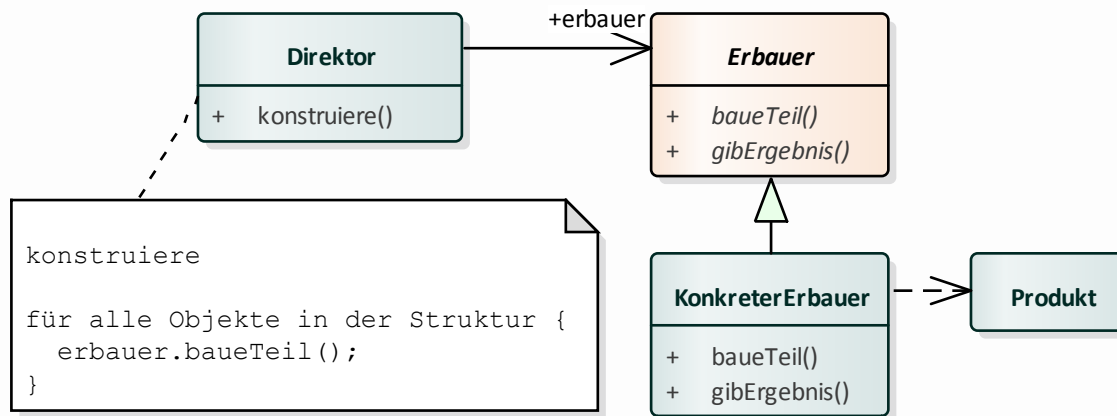
- Wenn eine Klasse die Klasse von Objekten, die sie erzeugen muss, **nicht im Voraus kennen kann**.
- Wenn eine Klasse möchte, dass ihre Unterklassen die von ihr zu erzeugenden **Objekte festlegen**.
- Wenn Klassen Zuständigkeiten an eine von mehreren Hilfsunterklassen delegieren sollen; und dabei das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



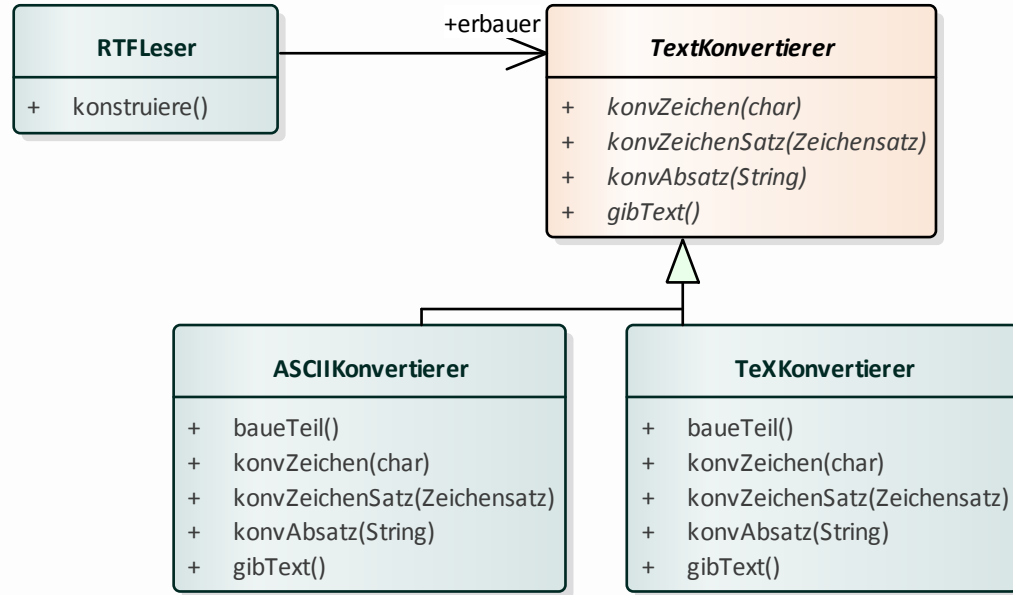
Zweck

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

Struktur des Erbauers



Beispiel eines Erbauers

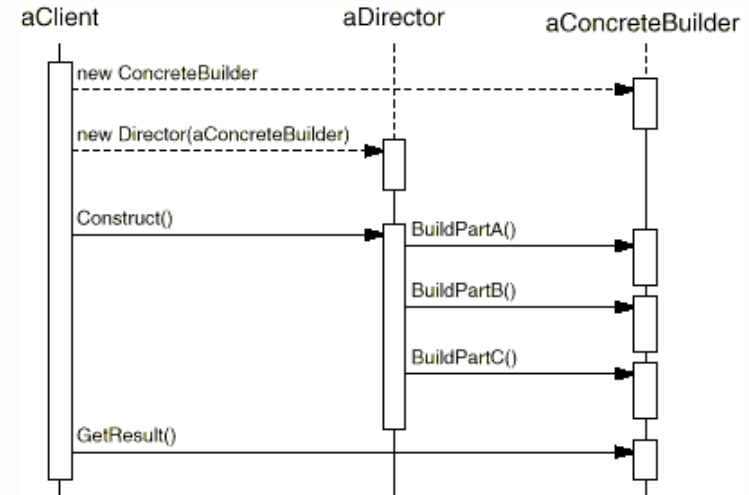


Transformation zwischen Textformaten

- 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.

Interaktionen

- Der **Klient erzeugt** das Direktor-Objekt und konfiguriert es mit dem gewünschten Erbauer-Objekt.
- 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.



Der Erbauer trennt den Konstruktionsalgorithmus von der 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).

Abstrakte Fabrik (Abstract Factory)

Abstrakte Fabrik (Abstract Factory)

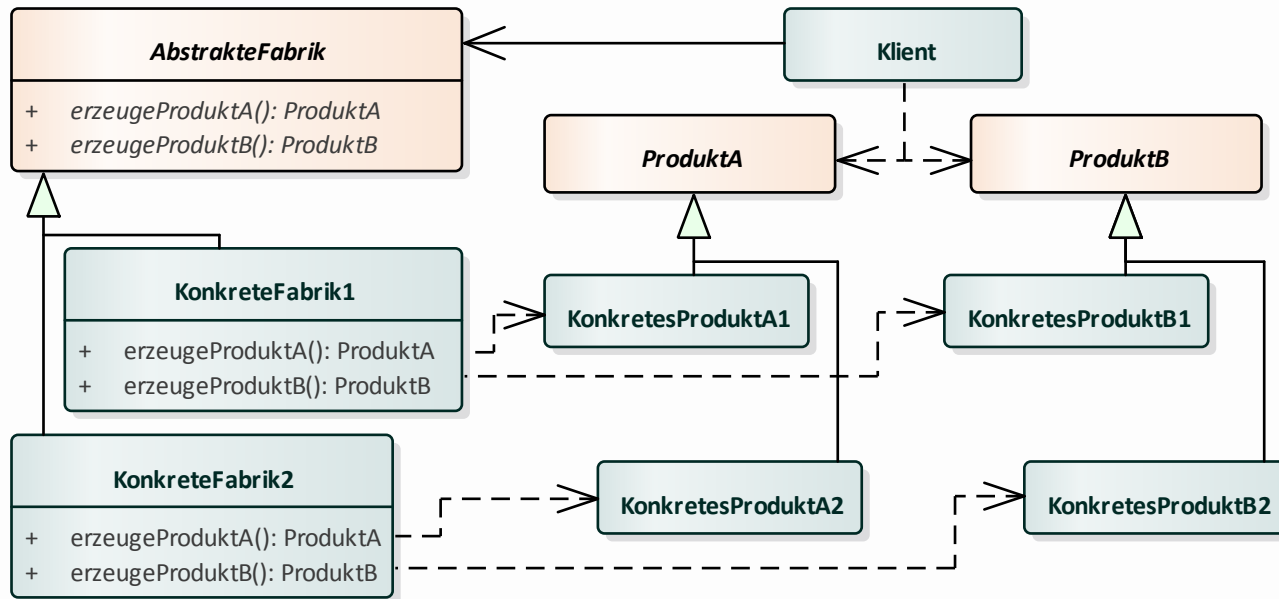
auch bekannt als:

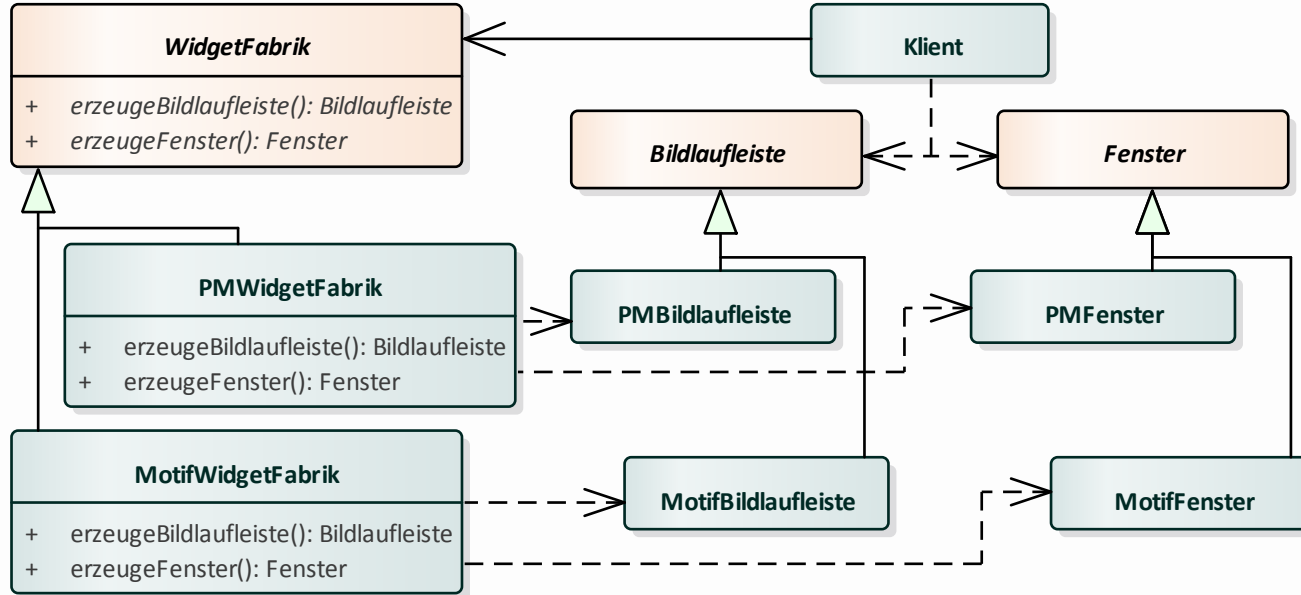
Kit



Zweck

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





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 verwandten Produktobjekten **zusammen verwendet** werden solle, und dies erzwungen werden muss.
- Bei einer Klassenbibliothek, die nur die Schnittstellen, aber **nicht die Implementierungen offen** legt.

Erbauer vs. Abstrakte Fabrik

Das Abstrakte-Fabrik-Muster ist dem Erbauermuster in der Hinsicht ähnlich, dass es ebenfalls komplexe Objekte konstruieren kann.

Der Hauptunterschied ist, dass das **Erbauermuster** sich auf den *schrittweisen Konstruktionsprozess* eines komplexen Objekts konzentriert.

Die Betonung der **abstrakten Fabrik** liegt auf **Familien von Produktobjekten** (ob nun einfach oder komplex).

Der **Erbauer** gibt das Produkt als **letzten Schritt** zurück, während die **Abstrakte Fabrik** das Produkt **unmittelbar** zurückgibt.

Inhalt

- Memento
- Prototyp
- Fliegengewicht
- Einzelstück

Memento

Memento (Memento)
auch bekannt als:
Token



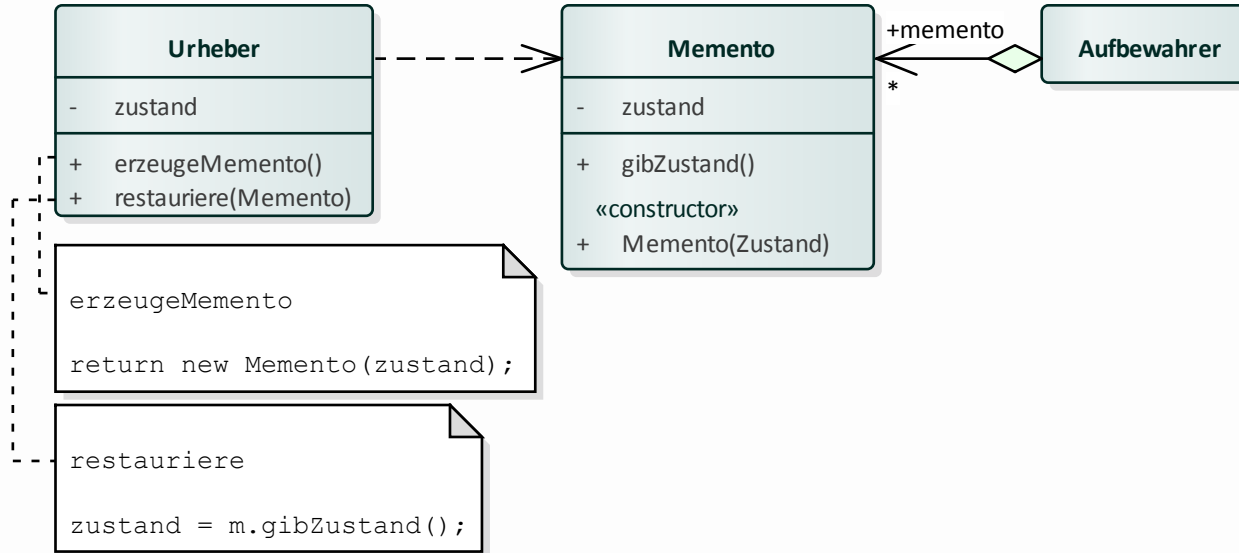
Zweck

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

- Separates Objekt als **Kontrollpunkt** für Undo- und Rollback-Operationen.

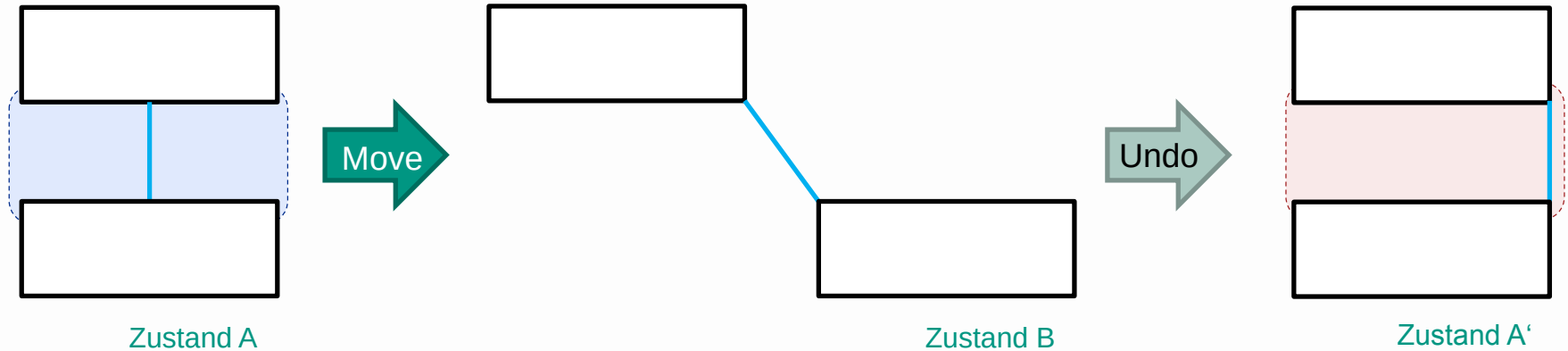


Struktur des Memento



Komplexe Haltepunkte und Undo-Mechanismen

Rechtecke bleiben verbunden, wenn ein Rechteck bewegt wird:



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

- Wenn eine **Momentaufnahme** (eines Teils) des Zustands eines Objekts zwischengespeichert werden muss, sodass 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 (Prototype)

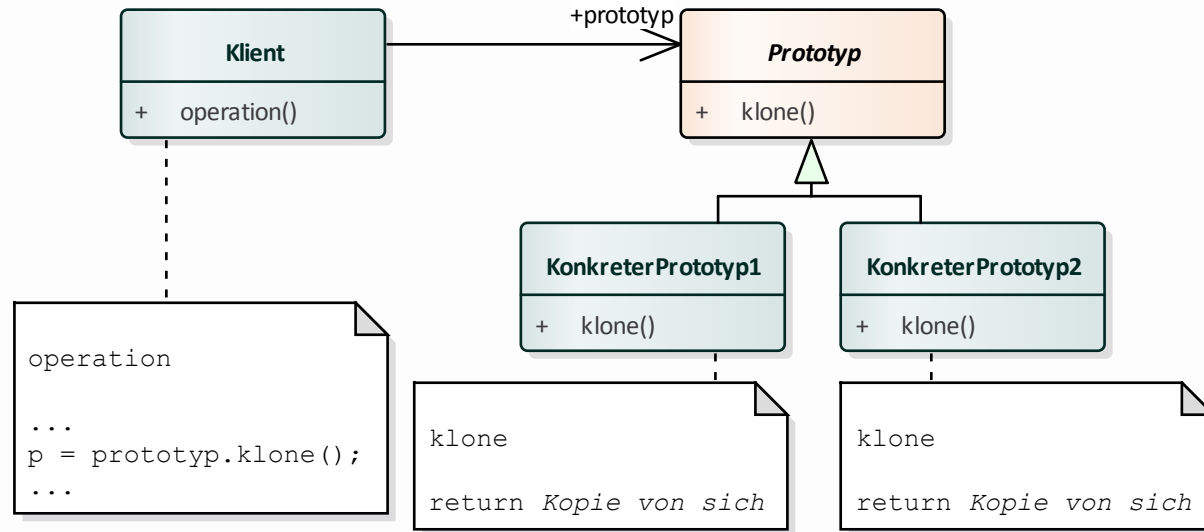
Prototyp (Prototype)



Zweck

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

Struktur des Prototyps



Das Prototypmuster wird verwendet, wenn ein **System unabhängig davon** sein soll, **wie** seine Produkte erzeugt, zusammengesetzt und repräsentiert werden, und...

- ... 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 klonen, statt die Objekte einer Klasse jedes mal von Hand mit dem richtigen Zustand zu erzeugen.

Prototyp vs. Fabrikmethode

Benutze Prototypen,
falls der **Aufbau** eines Objekts wesentlich
mehr Zeit erfordert **als eine Kopie** anzulegen.

Benutze Prototypen *nicht*,
wenn die **Kopien so groß** werden,
dass es zu **Speicherengpässen** kommt.

Fliegengewicht (Flyweight)

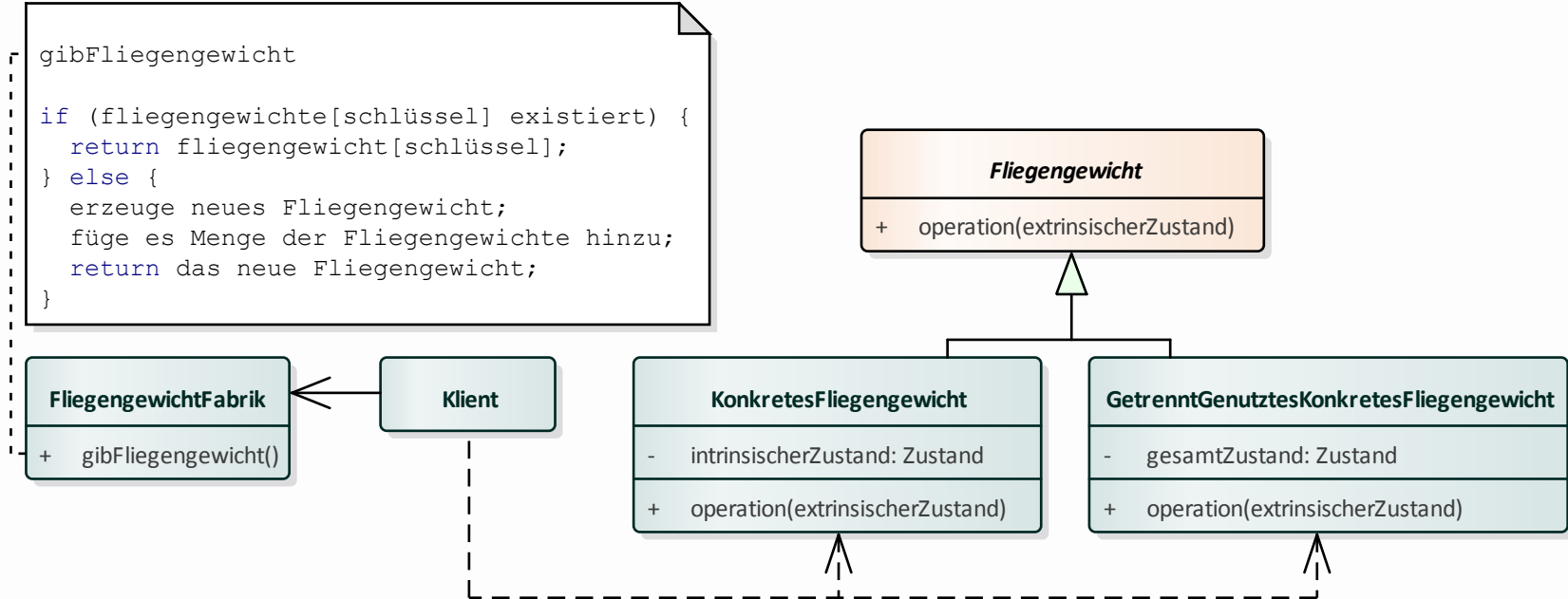
Fliegengewicht (Flyweight)



Zweck

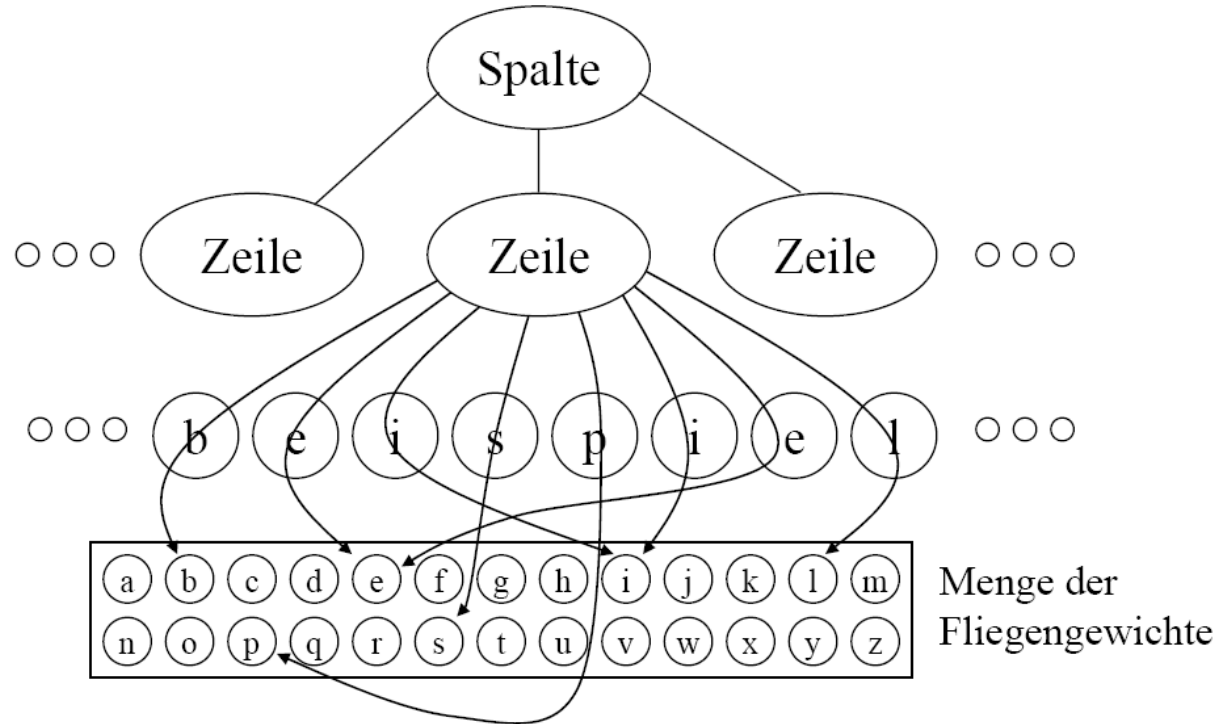
Nutze Objekte **kleinster Granularität** gemeinsam, um große Mengen von ihnen **effizient speichern** zu können.

Struktur des Fliegengewichts



Beispiel: Texteditor (I)

Objektmodellierung bis hinunter zu einzelnen Zeichen



Beispiel: Texteditor (II)

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

z.B. eine Konstante:

```
public final String UNDEFINED = „undef“;
```

Anwendbarkeit

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

Einzelstück (Singleton)

Einzelstück (Singleton)



Zweck

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



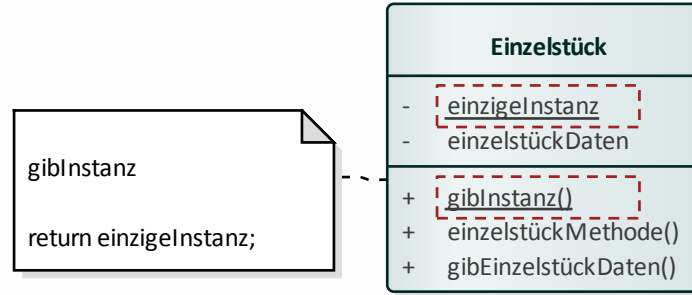
Motivation

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.

Oft bei Fassade
und Memento

Struktur Einzelstück

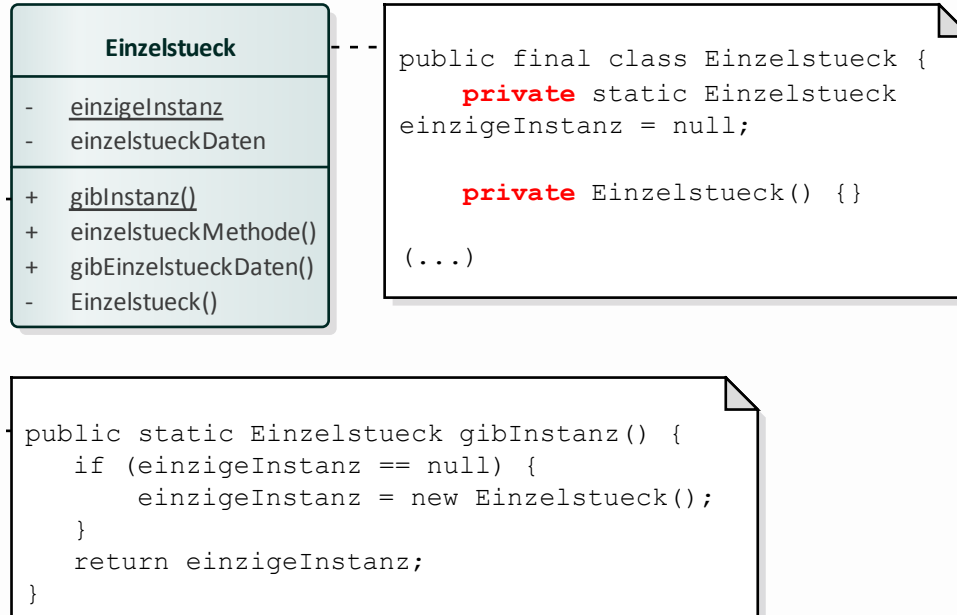


Instanz und sein Getter
sind static !

Anwendbarkeit

- Wenn es von einer Klasse **nur eine Instanz** geben darf und die Klienten **globalen Zugriff** darauf über eine Instanz-Operation bekommen sollen.
- Wenn die einzige Instanz durch Unterklassenbildung **erweiterbar** sein soll, und die Klienten ohne Veränderung ihres Quelltextes diese nutzen sollen.
- Wenn es schwierig oder unmöglich ist, festzustellen, welcher Teil der Anwendung die **erste Instanz** erzeugt.

Codebeispiel (Lazy Initialization)



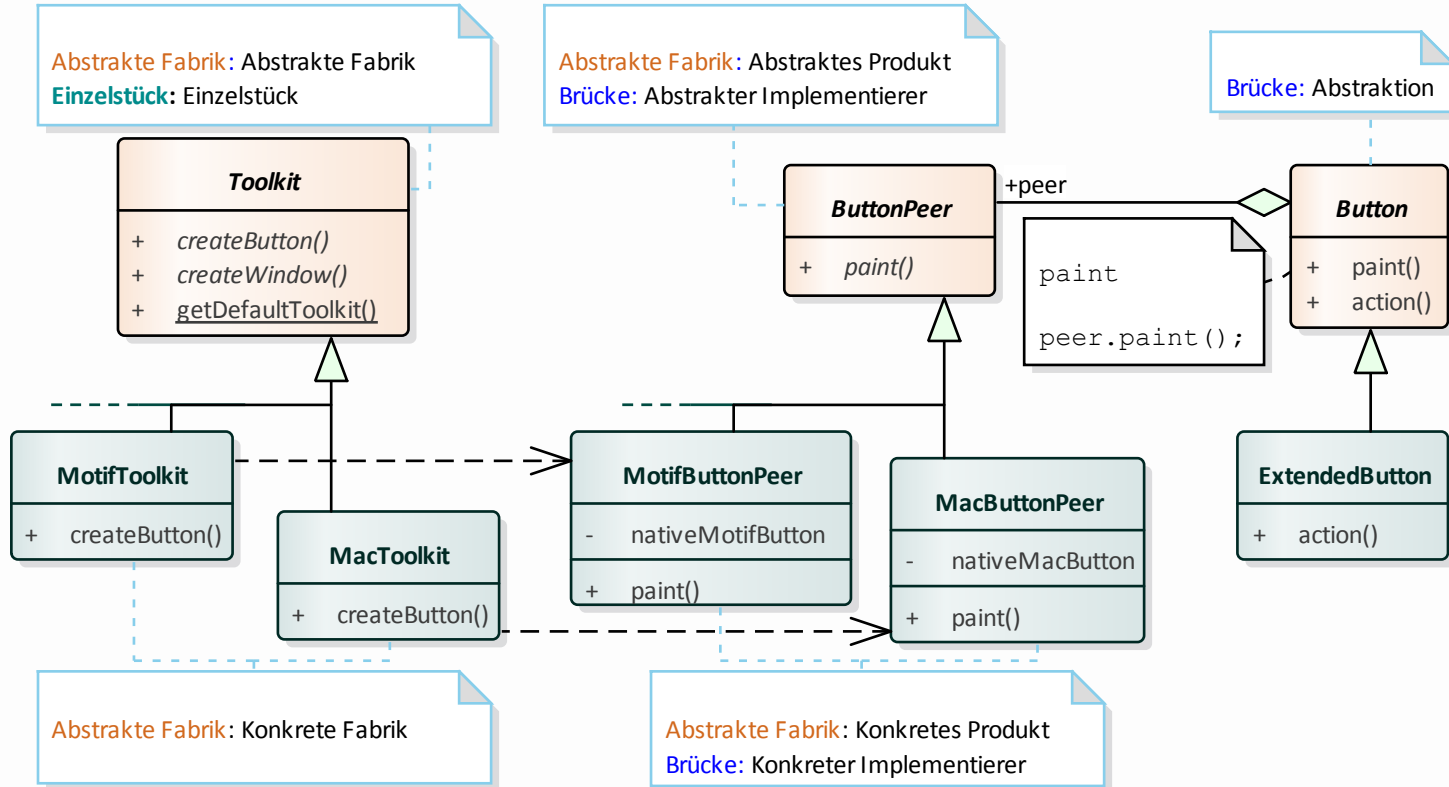
Checkliste: Einzelstück

1. Einzelstück-Klasse erfüllt Instanz-Modifizierungen
 - nur **private** (oder **protected**) Konstruktoren
(Kein Konstruktor darf **public** sein!)
 - **private static** Instanz-Attribut
 - **public static** Instanz-Zugriffsoperation
2. Die Zugriffsoperation erfüllt ihre Aufgaben
 - **Lazy Initialization**: Sie sorgt dafür, dass das Klassenattribut im Falle der ersten Nutzung initialisiert wird.
 - Klienten können (**nur**) mit ihr die Instanz erhalten und verändern



Einzelstueck	
-	<u>einzigelInstanz</u>
-	einzelstueckDaten
+	<u>gibInstanz()</u>
+	einzelstueckMethode()
+	gibEinzelstueckDaten()
-	Einzelstueck()

Beispiel: Brücke, Einzelstück und Abstrakte Fabrik



Inhalt

- Tafel
- Befehl
- Zuständigkeitskette
- Observer (Beobachter)
- Master/Slave
- Prozesssteuerung
 - System ohne Rückkoppelung
 - System mit Rückkoppelung
 - mit Rückführung
 - mit Störgrößenaufschaltung

Tafel (Blackboard)

Tafel (Blackboard)

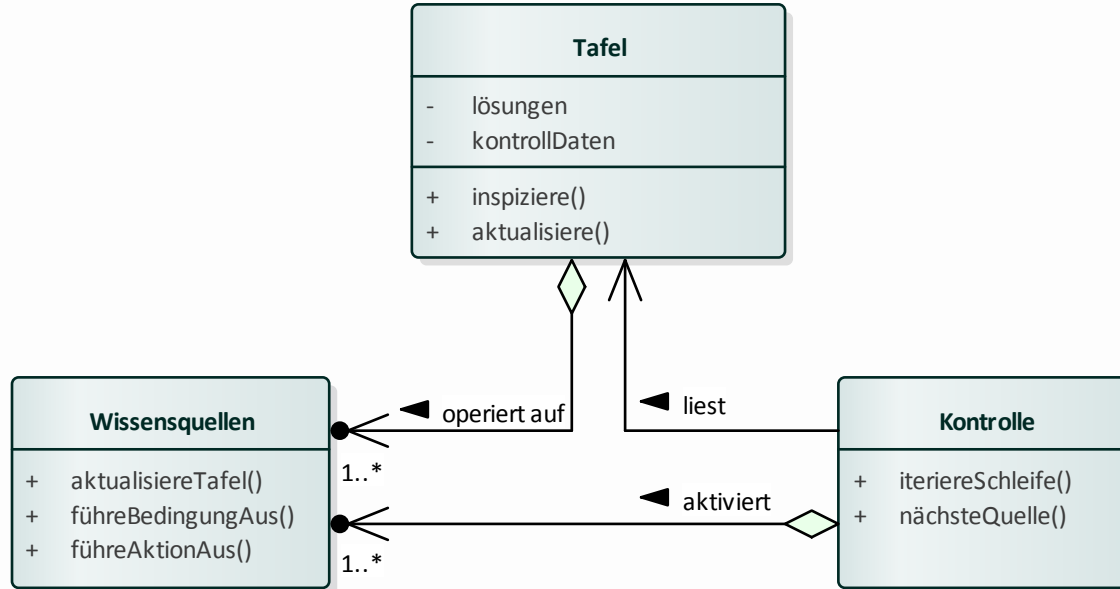


Zweck

Das Tafelmuster ist nützlich bei Problemen, für die **keine deterministischen Lösungsstrategien** bekannt sind.

Mehrere spezialisierte **Subsysteme vereinigen ihr Wissen** auf der Tafel, um eine eventuell partielle oder **approximative Lösung** zu finden.

Struktur der Tafel

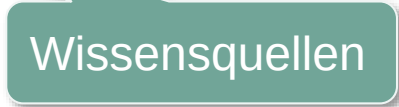


Einfache Form

Kontrolle aktiviert alle Wissensquellen **der Reihe nach**.

Komplexere Form

Bestimmt eine Folge anwendbarer Wissensquellen (über **bedingung()**), wählt davon aus, und führt diese dann aus.



- Wenn mehrere Transformationen („Wissensquellen“) auf einer **gemeinsamen Datenstruktur** („Tafel“) operieren.
- Wenn das Auslösen der Transformationen **vom Inhalt** der Datenstruktur **gesteuert** wird.
- Wenn die **Auswahl** der anwendbaren Transformationen gesteuert werden soll.

Befehl

Befehl (Command)

auch bekannt als:

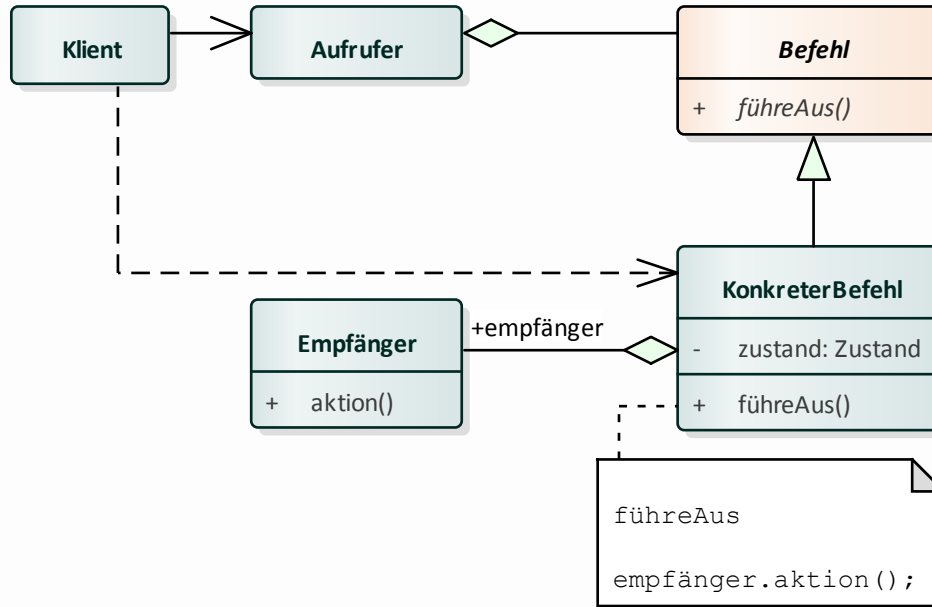
Kommando, Aktion, Transaktion



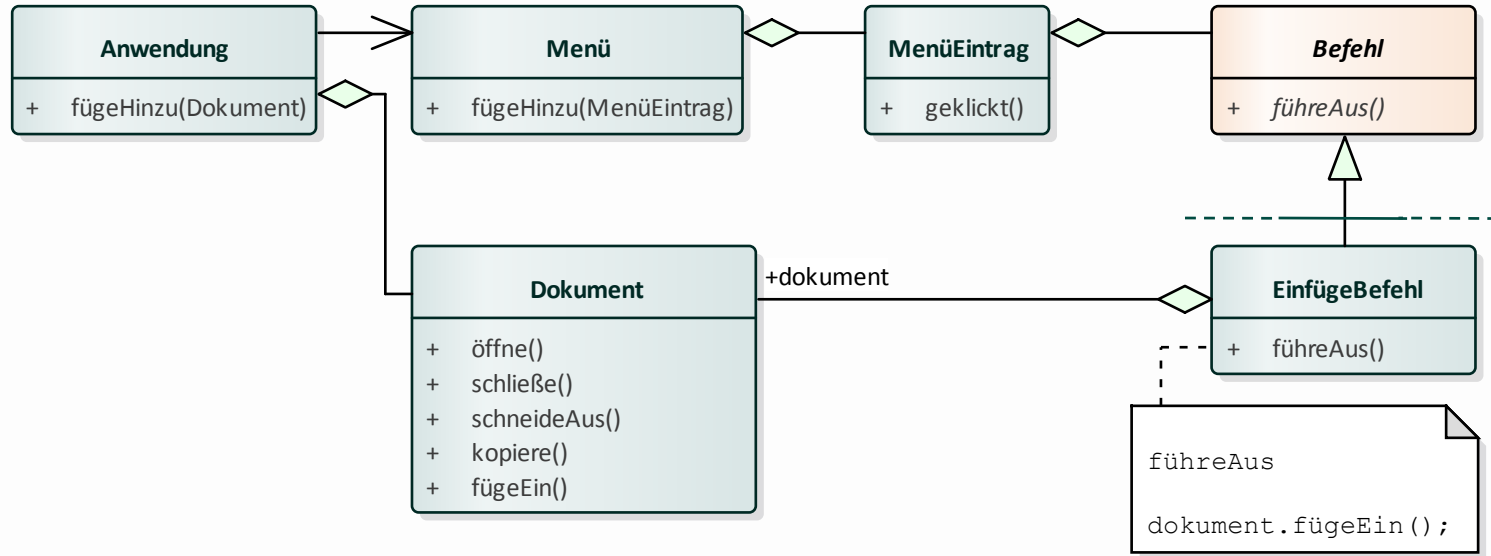
Zweck

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.

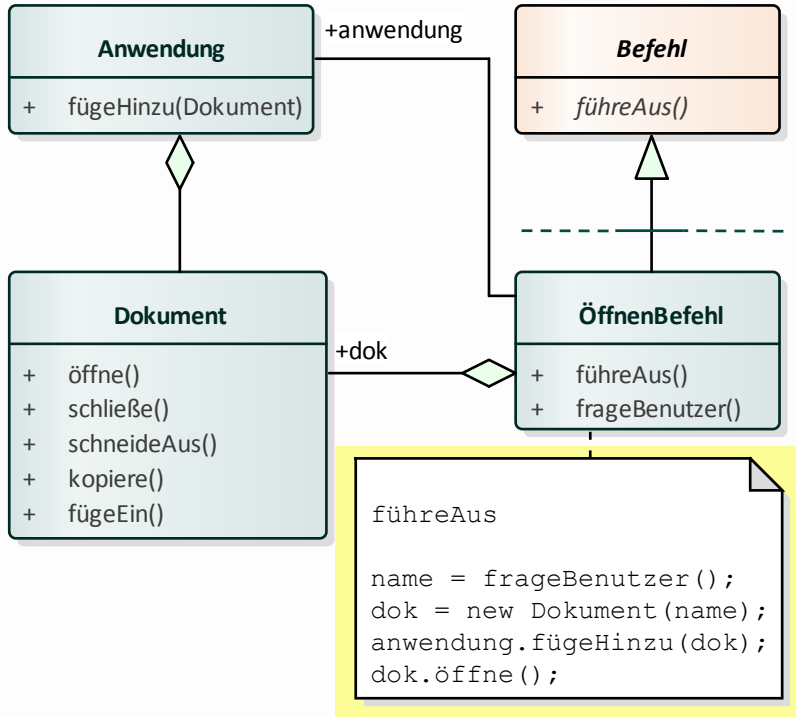


Beispiel (1): Menüs bei Benutzerschnittstellen



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

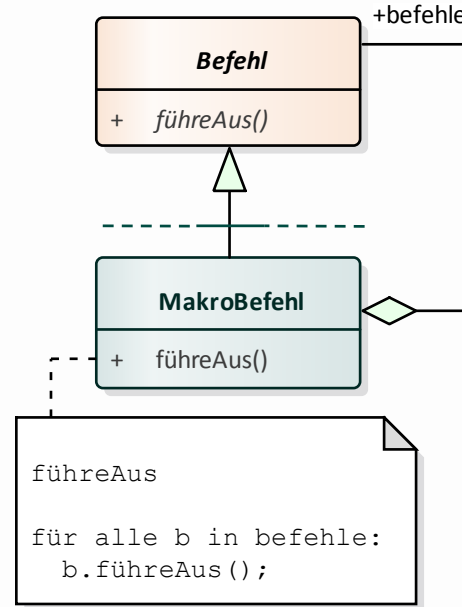
Beispiel (2): Befehl zum Öffnen eines Dokuments



Die **führeAus()**-Operation der Klasse **ÖffnenBefehl**

- fragt den Benutzer nach dem Namen eines Dokuments,
- erzeugt ein entsprechendes **Dokument**-Objekt,
- fügt das Dokument in die empfangende **Anwendung** ein,
- und öffnet es.

Abfolge von Befehlen (Makrobefehl)

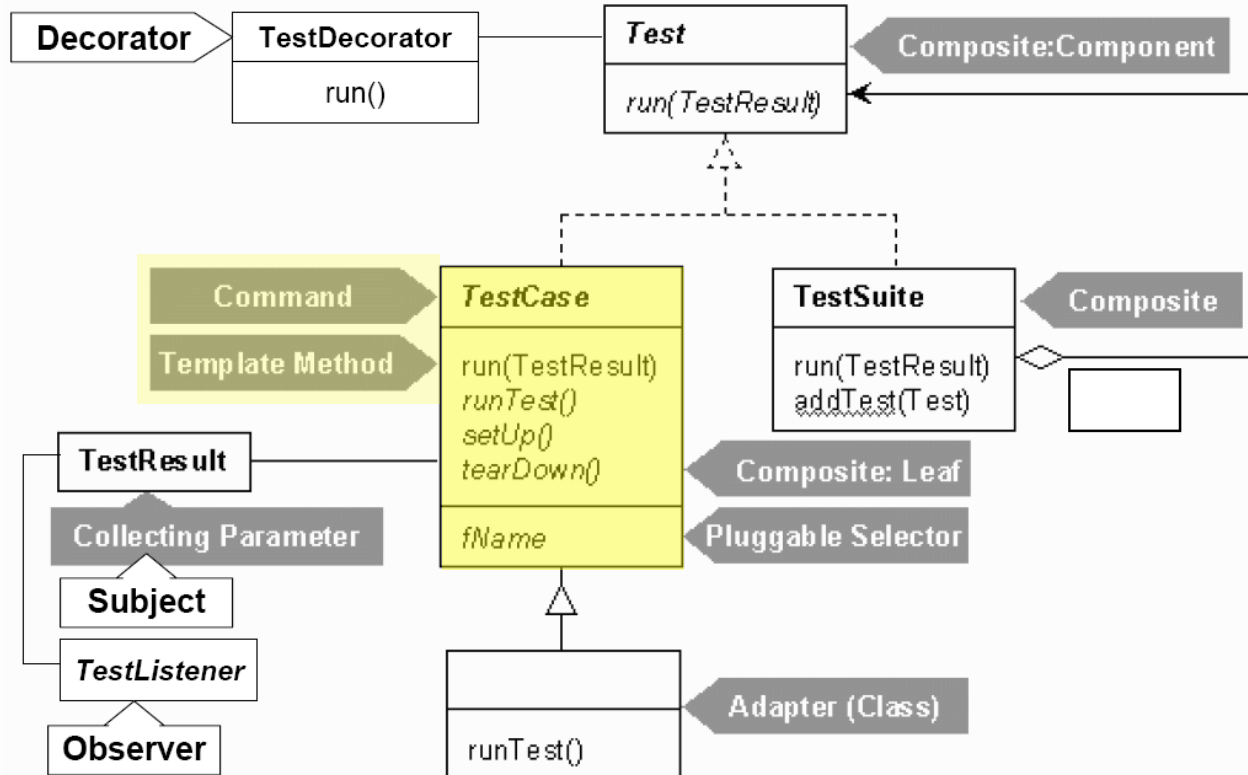


Kompositum

Anwendbarkeit

- Wenn Objekte mit einer auszuführenden Aktion **parametrisiert** werden sollen (wie bei den MenüEintrag-Objekten).
- Wenn die auszuführende Aktion **von verschiedenen Stellen** im Programm angestoßen wird (**Menü, Kontextmenü, Button, ...**)
- Wenn Anfragen zu **unterschiedlichen Zeiten** spezifiziert, aufgereiht und ausgeführt werden sollen.
- Wenn ein **Rückgängigmachen** von Operationen (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**).

Beispiel: JUnit - ein Regressionstest-Framework



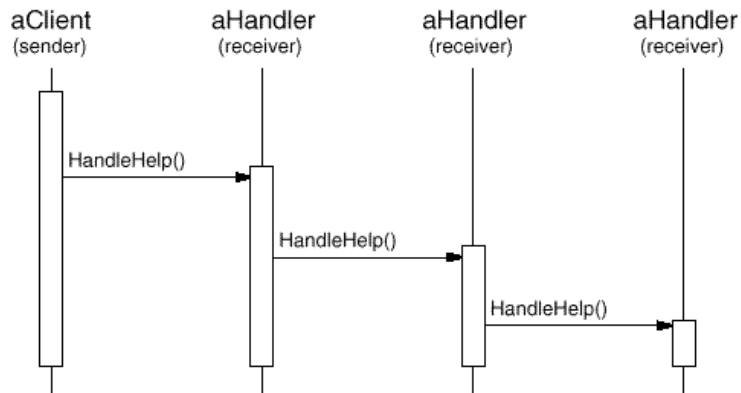
Zuständigkeitskette (Chain of Responsibility)

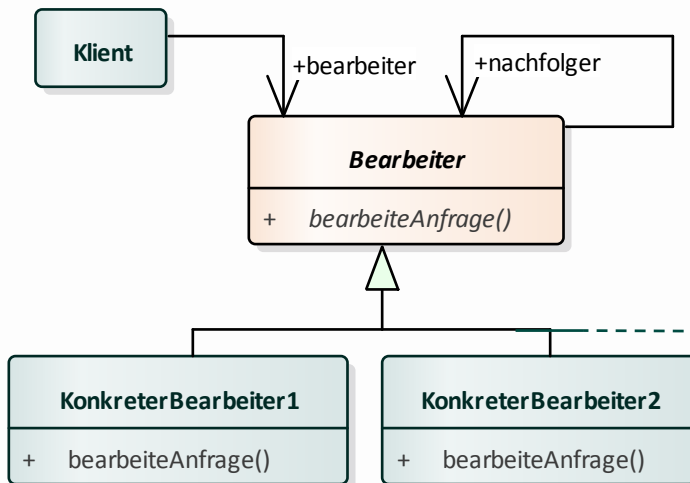
Zuständigkeitskette (Chain of Responsibility)



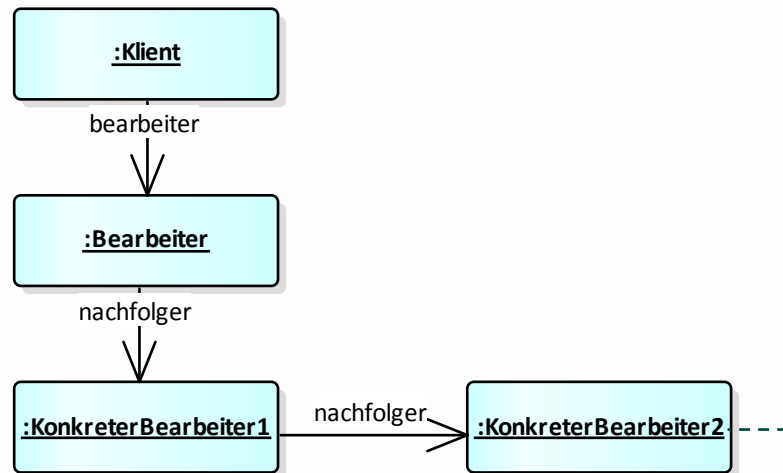
Zweck

Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Anfrage zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.



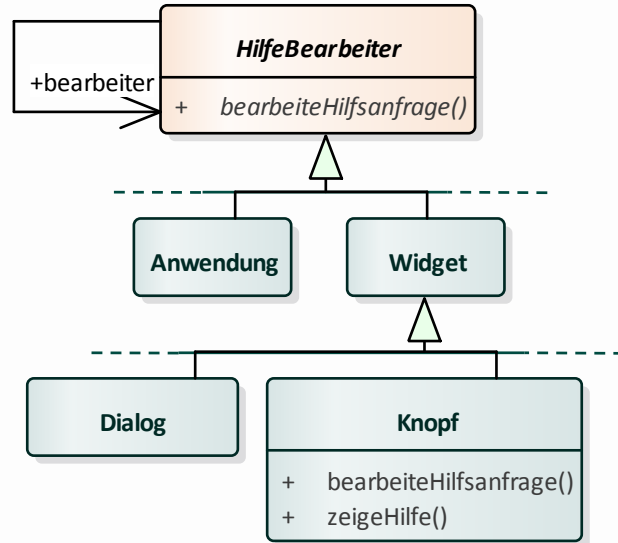


Klassendiagramm

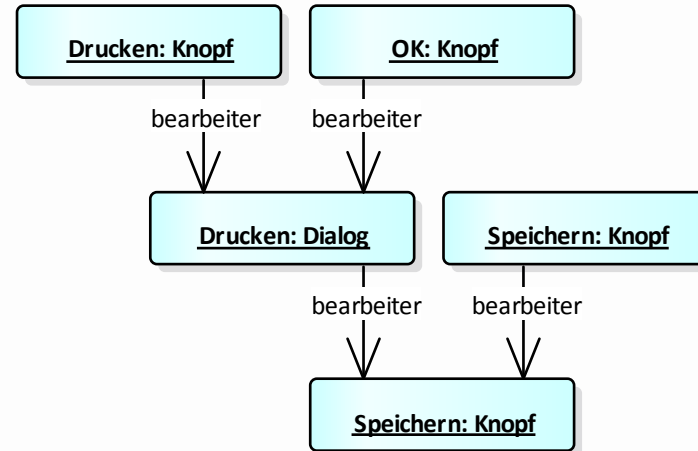


Objektdiagramm

Beispiel: Hilfesystem für grafische Benutzerschnittstelle



Klassendiagramm



Objektdiagramm

- Wenn **mehr als ein Objekt eine Anfrage bearbeiten** können soll und dasjenige Objekt, das dies tut, nicht von vornherein bekannt ist. (Dieses Objekt muss zur Laufzeit automatisch bestimmt werden.)
- Wenn eine Anfrage an eines von mehreren Objekten gerichtet werden soll, **ohne den Empfänger explizit anzugeben**.
- Wenn eine **Menge** Objekte, welche eine Anfrage bearbeiten sollen, **dynamisch festgelegt** wird.



Zweck

Definiere eine **1-zu-n Abhängigkeit** zwischen Objekten, sodass die Änderung des Zustandes eines Objektes zu einer Aktualisierung aller abhängigen Objekte führt.

Beobachter (Observer-Pattern) (II)

Problem

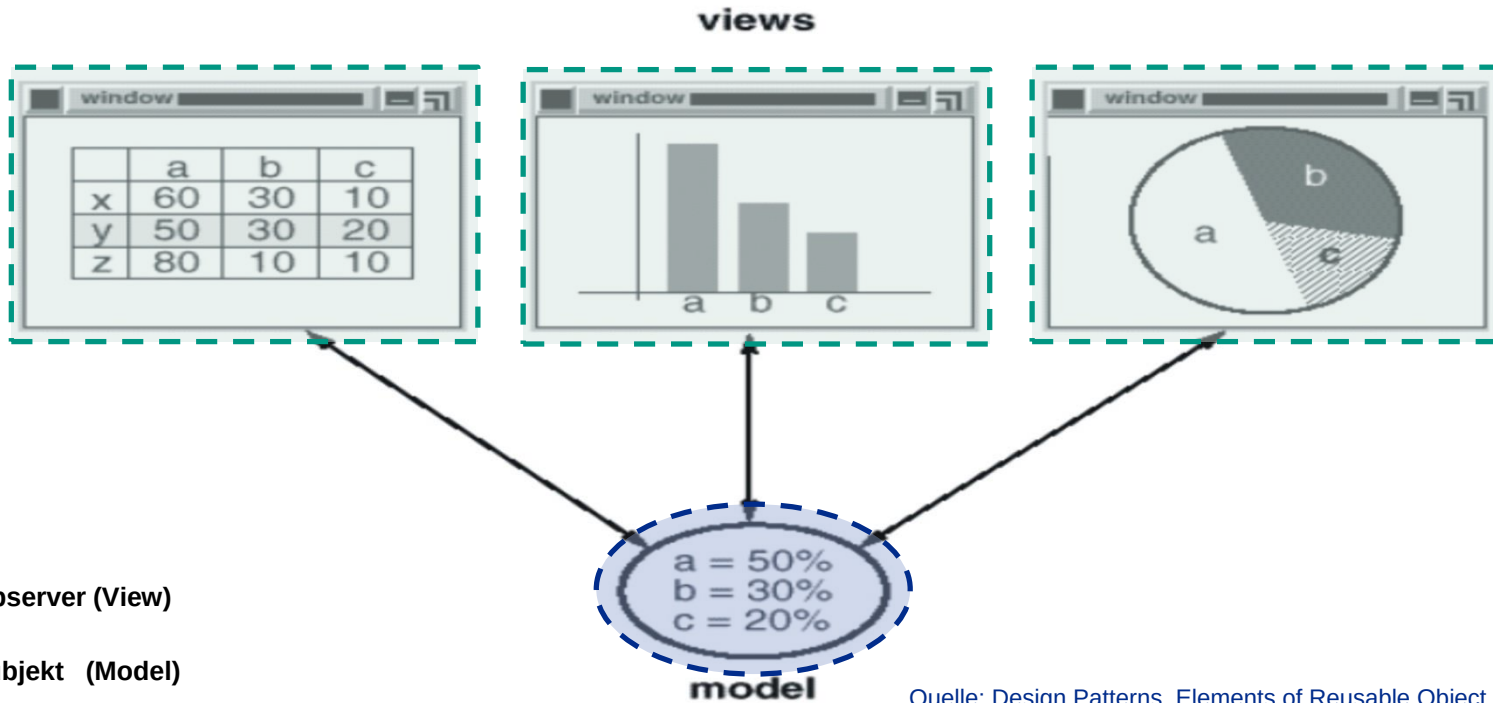
Das unabhängige Objekt soll die abhängigen Objekte bzgl. Veränderungen benachrichtigen, ohne Annahmen zu treffen, wer und wie viele dies sind (**lose gekoppelt**).

Lösung

Ein oder mehrere Objekte („Observer“) möchten bei der Veränderung eines Objekts („Subject“) eine Methode ausführen, die auf den veränderten Zustand zugreifen kann. Die „Observer“ registrieren sich beim „Subject“.

Observer-Pattern Abbildung

Model-Objekt und drei View-Objekte



Quelle: Design Patterns, Elements of Reusable Object Oriented Software

Abstraktion mit zwei voneinander abhängigen Aspekten

- wenn ein Aspekt von dem anderen **abhängt**
- **Kapselung** der Aspekte in unterschiedlichen Objekten unabhängig wiederverwendbar

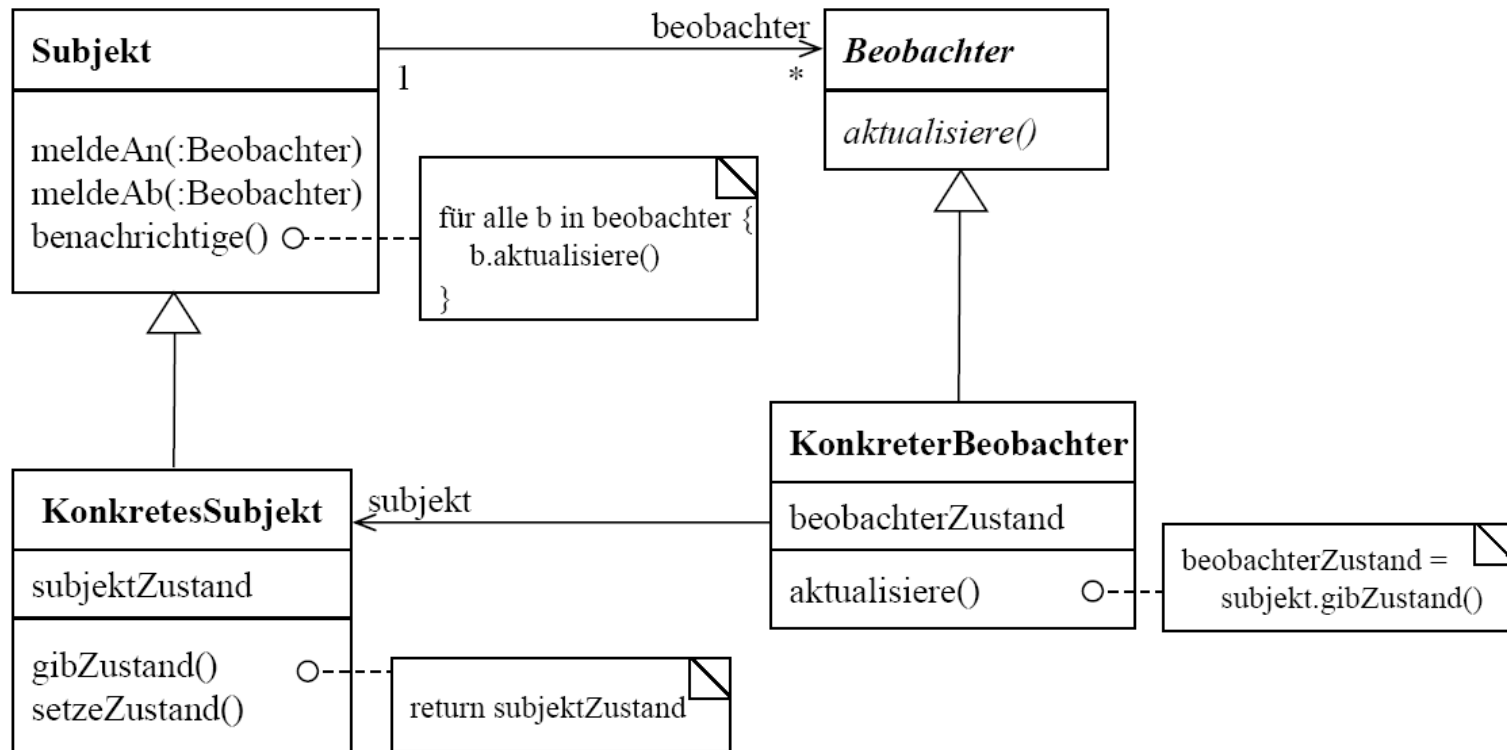
Kaskadierende Änderung

- wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt (**Wirkkette**)
- Bei **Ungewissheit** über Anzahl der Objekte, die geändert werden müssen.

Benachrichtigung unbekannter Objekte

- **ohne Annahmen** über die zu benachrichtigenden Objekte
- Vermeidung von eng gekoppelten Objekten.

Observer-Pattern Struktur



Subject

kennt seine Observer

Observer

definiert `update ()` - Schnittstelle

ConcreteObserver

verwaltet Referenz auf das Subjekt

- speichert Zustand, der mit dem Subjekt zusammenpassen soll.
- Implementiert `update ()`, um Zustand mit dem des Subjekts konsistent zu halten.

Observer-Pattern Interaktion

Beobachter werden **durch konkretes Subjekt** benachrichtigt, sobald sich der Zustand (Subjekt) ändert.

Mit Hilfe von Anfragen an das Subjekt wird der jeweilige Zustand ermittelt.

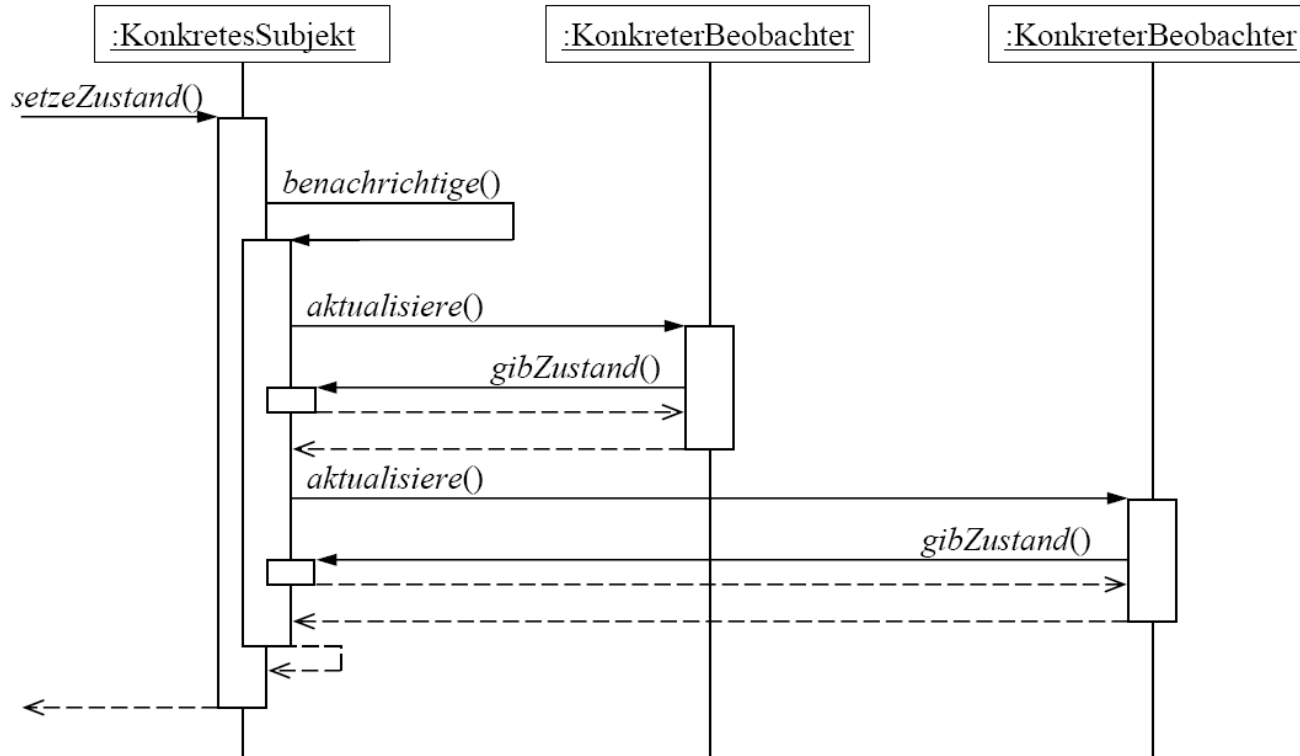
Subject

- Zustand ändert sich
- benachrichtigt alle konkreten Beobachter

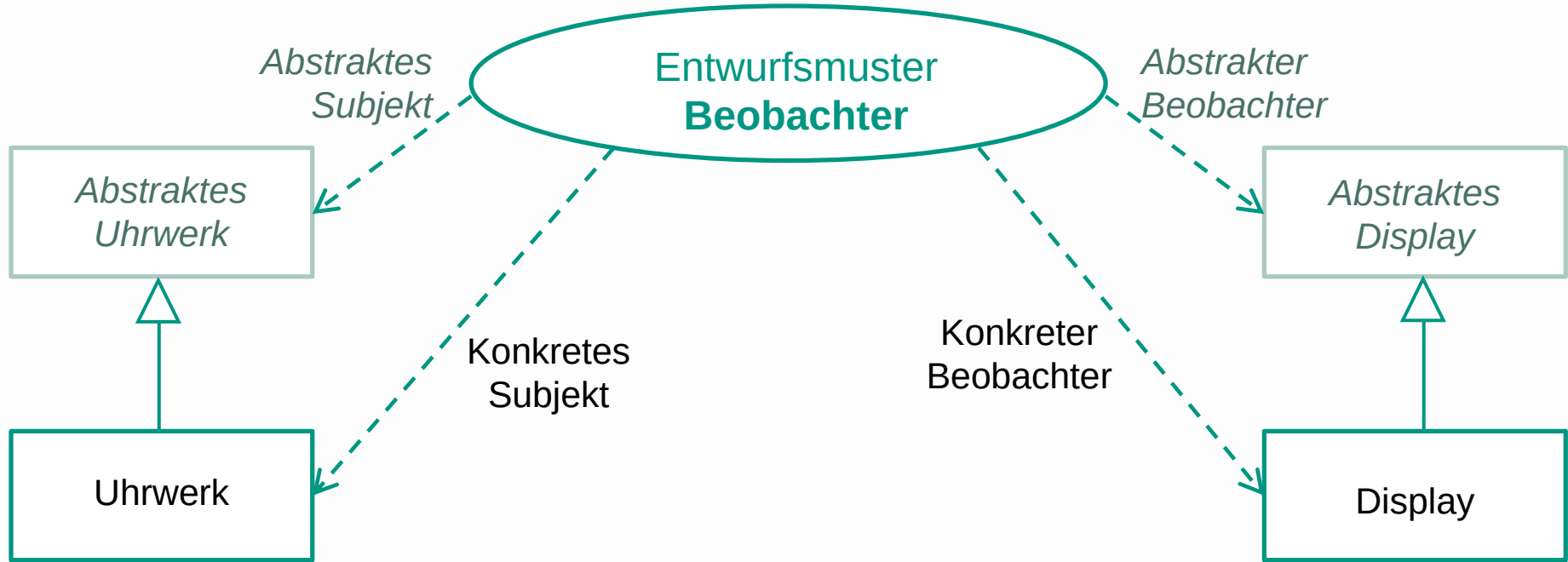
ConcreteObserver

- befragt Subjekt nach Informationen
- verwendet Informationen, um seinen Zustand mit dem des Subjekts abzugleichen
- **update ()** (**aktualisiere ()**)

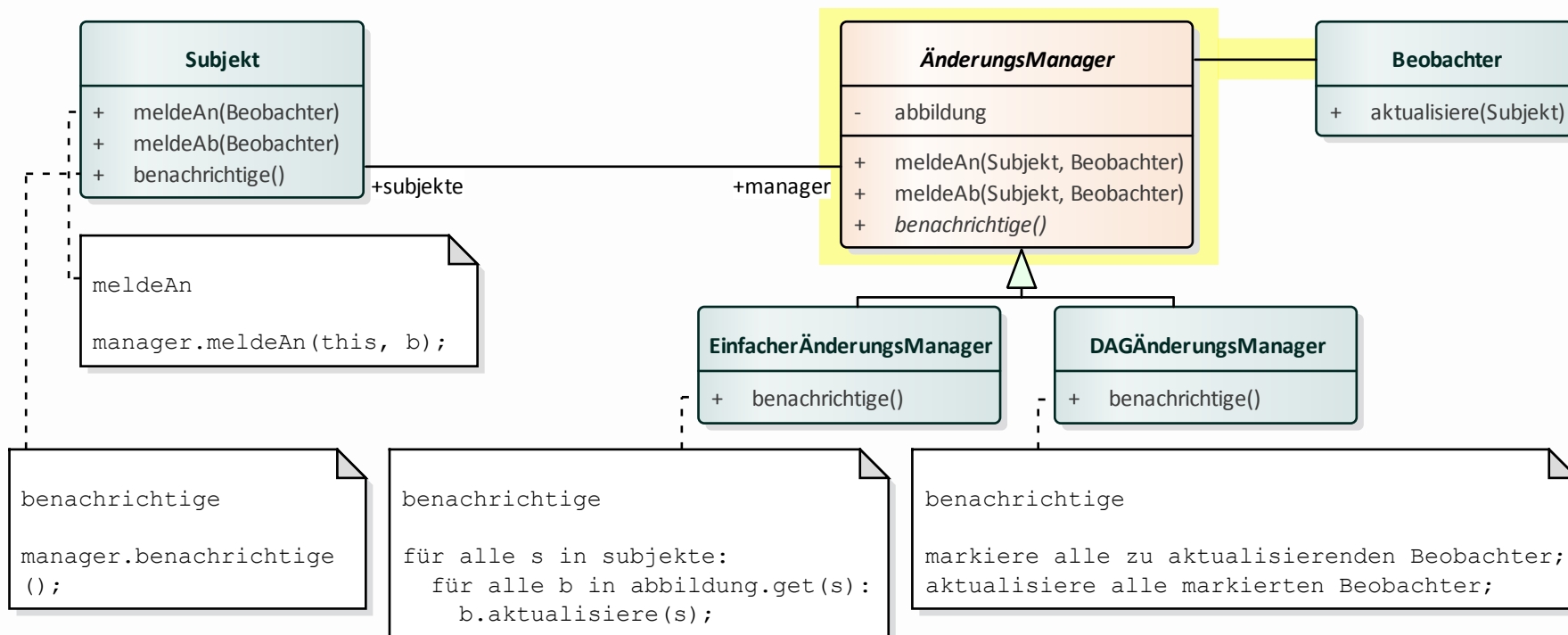
Interaktionsdiagramm Beobachter



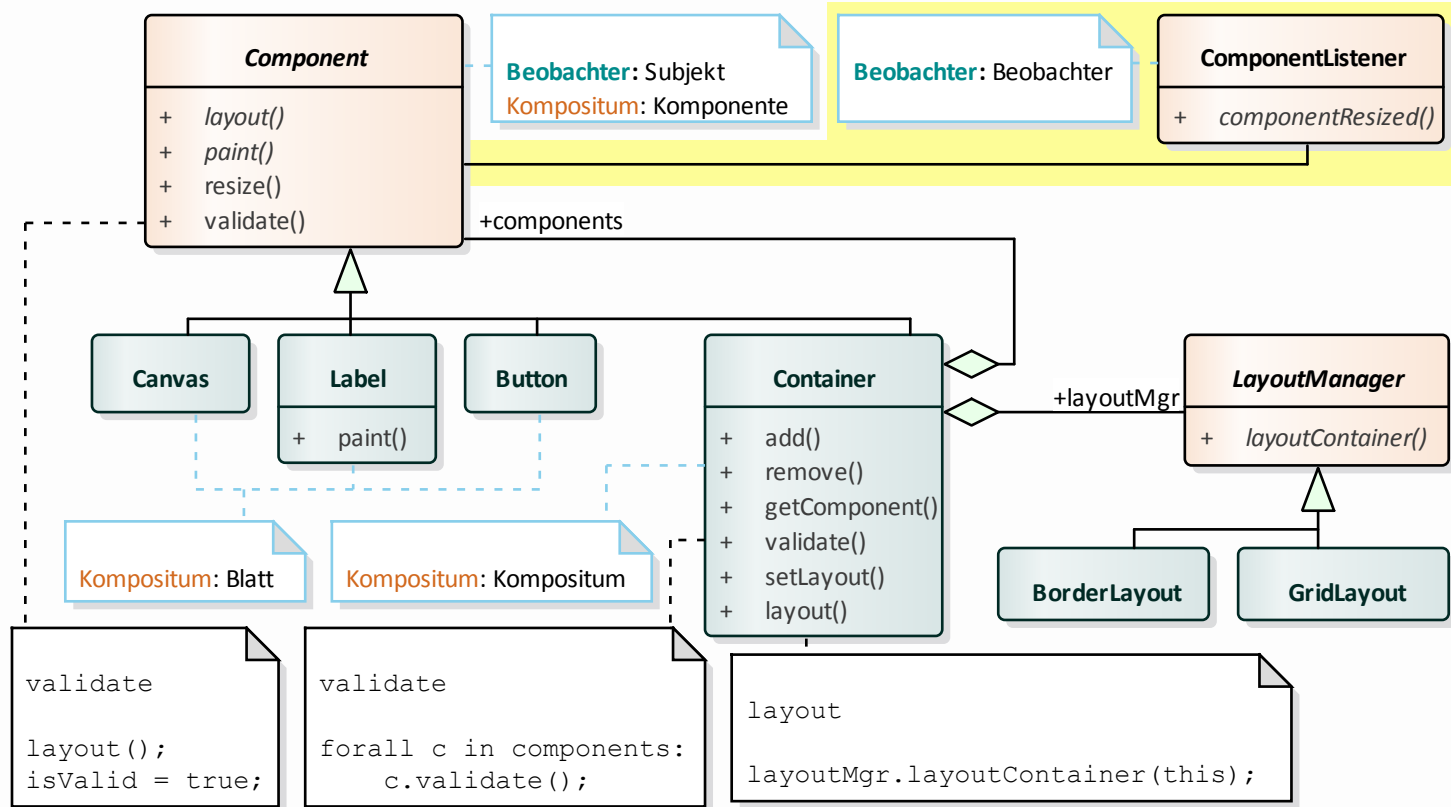
Beispiel



Beobachter mit ÄnderungsManager



Beispiel aus JAVA AWT (Beobachter)



Master/Slave

Auftraggeber/Arbeiter (**Master/Slave**)
auch bekannt als:

Auftraggeber/Auftragnehmer,
Master/Workers

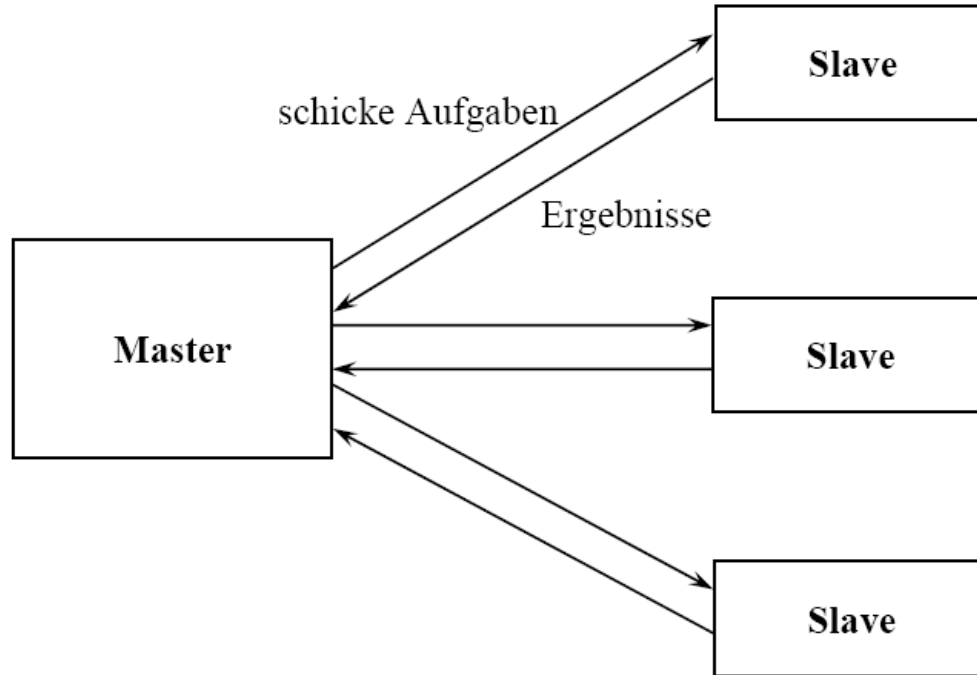


Zweck

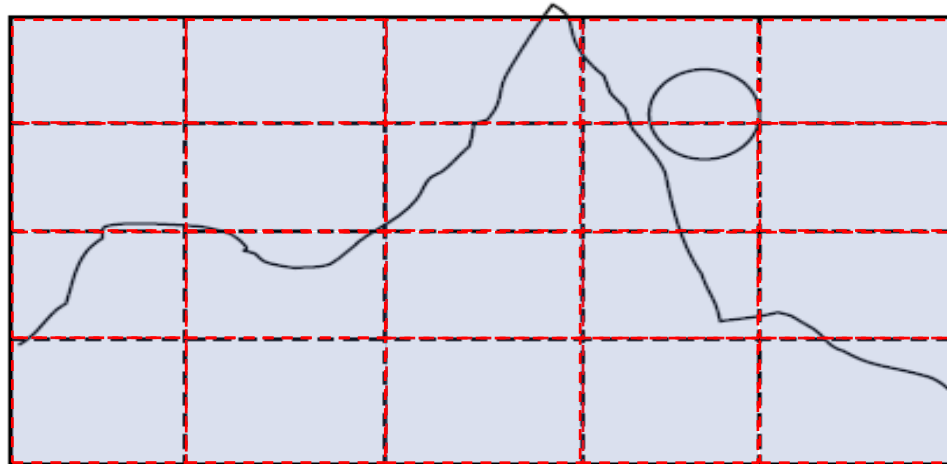
Das *Master/Slave-Muster* unterstützt **Fehlertoleranz** und **parallele Berechnung**.

Eine Master-Komponente (Auftraggeber) **verteilt die Arbeit** an identische Slave-Komponenten (Arbeiter); und **berechnet das Endergebnis** aus den Teilergebnissen, welche die Arbeiter zurückliefern.

Struktur von Master/Slave



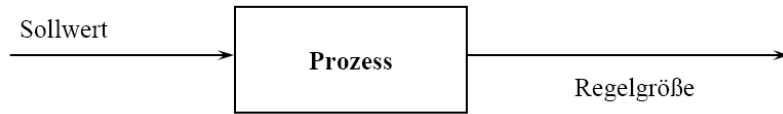
Dann setzt er das gesamte Bild aus den einzelnen Teilen zusammen.



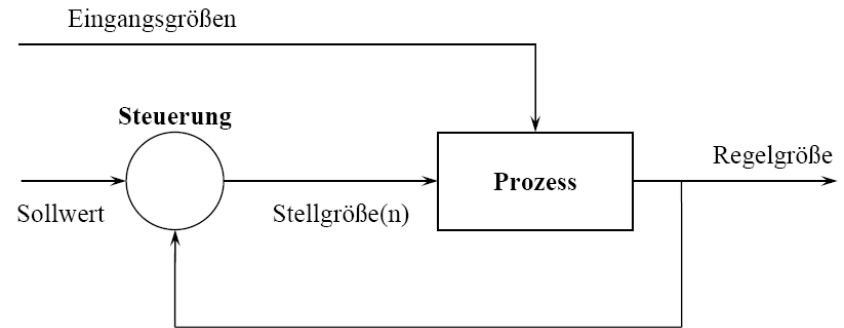
- Wenn es **mehrere Aufgaben** gibt, die **unabhängig** voneinander bearbeitet werden können.
- Wenn **mehrere Prozessoren** zur parallelen Verarbeitung zur Verfügung stehen.
- Wenn die **Belastung der Arbeiter ausgeglichen** werden soll.

Prozess-Steuerung (process control)

 **Zweck**
Regulierung eines physikalischen (kontinuierlichen) Prozesses.



System *ohne* Rückkoppelung



System mit Rückkoppelung

Regelung mit Rückführung

Regelung mit Störgrößenaufschaltung

Prozessvariablen

Eigenschaften des Prozesses, die gemessen werden können; folgende spezielle Arten werden häufig unterschieden.

Eingangsrößen

Prozessvariablen, die als Eingabewerte für den Prozess dienen.

Stellgrößen

Prozessvariablen, deren Wert von der Steuerung **verändert** werden kann.

Regelgrößen

Prozessvariablen, deren Wert das **System kontrollieren** möchte.

Sollwert

Der **angestrebte Wert** einer Regelgröße.

Inhalt

- Interpretierer
- Regelbasierter Interpretierer

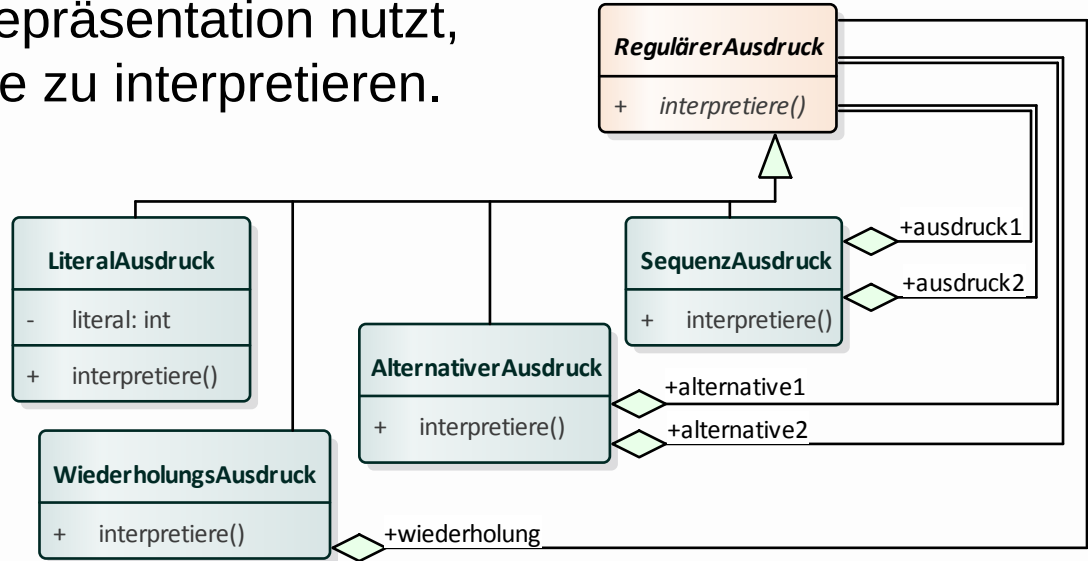
Interpretierer



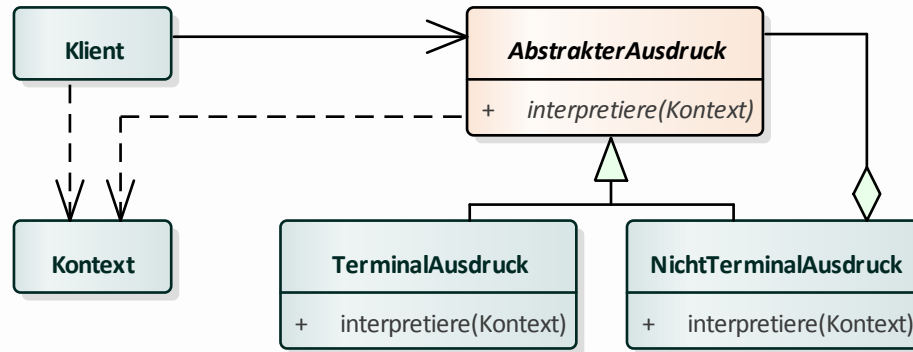
Zweck

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

Interpretierer (Interpreter)



Struktur des Interpretierers



Wenn eine Sprache interpretiert werden muss, und Ausdrücke der Sprache als **abstrakte Syntaxbäume** darstellbar sind.

Das Interpretierer-Muster 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 Parser-Generatoren eine bessere Alternative dar.

...die **Effizienz unproblematisch** ist.

Effiziente Interpretierer werden üblicherweise nicht durch Interpretation von Parse-Bäumen implementiert; sie transformieren die Bäume stattdessen in eine andere Form.

Interpretierer vs. Kompositum vs. Besucher

Der *Interpretierer* und das *Kompositum* haben die gleiche Struktur.

Man spricht von einem *Interpretierer*, wenn *Sätze einer Sprache* repräsentiert und ausgewertet werden.

- Der Interpretierer 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.

Regelbasierter Interpretierer



Zweck

Löse ein Problem durch **Anwendung** einer Menge **von Regeln**.

Regelbasierter Interpretierer
(Rule-Based System)

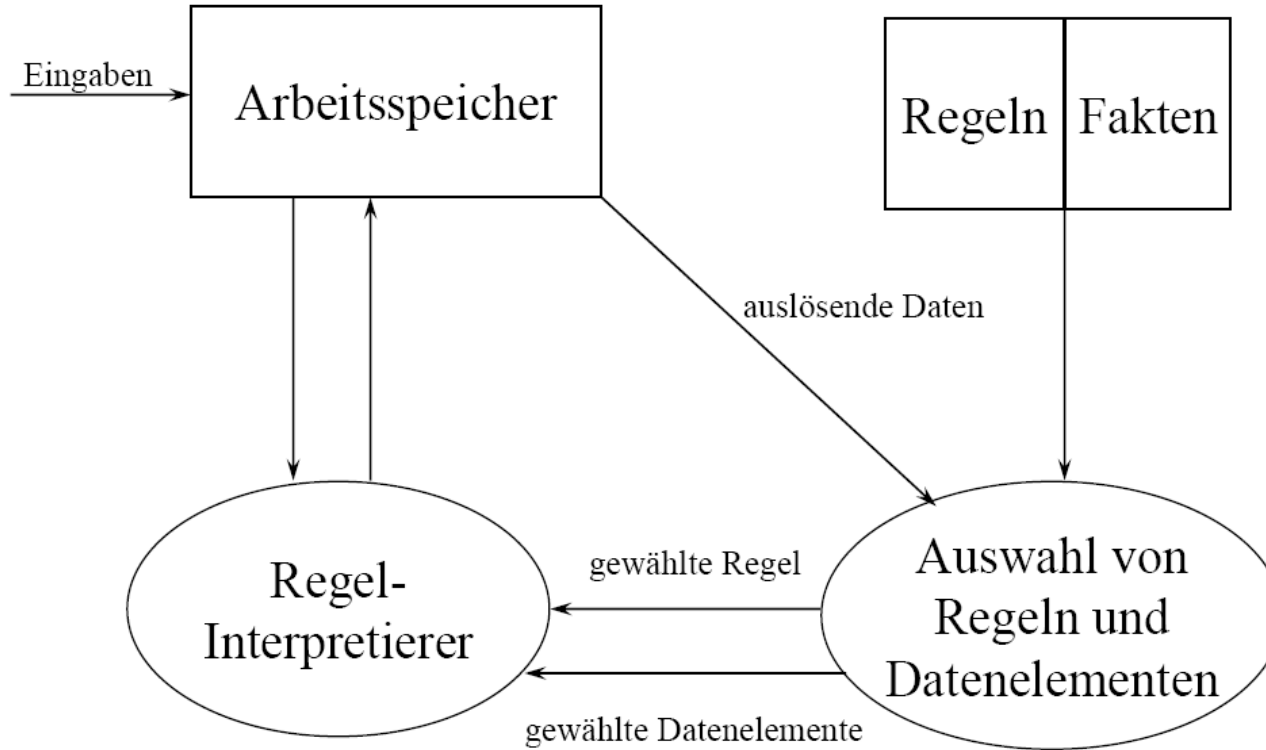
auch bekannt als:

Regelbasiertes Expertensystem

Eine **Regel** besteht aus Bedingungsteil und Aktionsteil.

- Falls der **Bedingungsteil**, angewandt auf eine Menge von Datenelementen im Arbeitsspeicher, wahr ergibt, dann kann der Aktionsteil ausgeführt werden.
- Der **Aktionsteil** ändert, ersetzt oder löscht Datenelemente, die im Bedingungsteil ausgewählt wurden, oder fügt neue Datenelemente hinzu.

Regelbasierter Interpretierer



Regelauswahl

Oft ist mehr als eine Regel anwendbar.

Dann wird eine **Auswahl** nach folgenden Schritten getroffen:

1. Eine gegebene Regel darf auf ein Datenelement im Arbeitsspeicher **höchstens einmal** angewandt werden.
2. Wenn mehrere *Regel-Datenelement-Kombinationen* anwendbar sind, dann werden die Regeln ausgewählt, die **mit jüngsten Datenelementen** operieren.
3. Wähle die Regeln mit der **höchsten „Spezifität“**, d.h. diejenigen, dessen Bedingungen die meisten Elemente im Arbeitsspeicher benötigen.
4. Wähle die Regel a) nach ihrer **Ordnung** b) **zufällig**

Anwendbarkeit

- Wenn eine Problemlösung am besten als eine Menge von **Bedingungs-Aktions-Regeln** formuliert werden kann.

Beispiel:

- **Diagnoseaufgaben**
 - **Konfigurierungsaufgaben**
- Wenn der Aktionsteil **nur einfache Operationen** an den Datenelementen erfordert:
 - keine Schleifen
 - keine Rekursion
 - keine Prozeduraufrufe, um Arbeitsspeicher zu modifizieren.

Musterklasse F:
**Bequemlichkeits-
Muster**

Inhalt

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

Bequemlichkeits-Methode

**Bequemlichkeits-
Methode**
(Convenience Method)

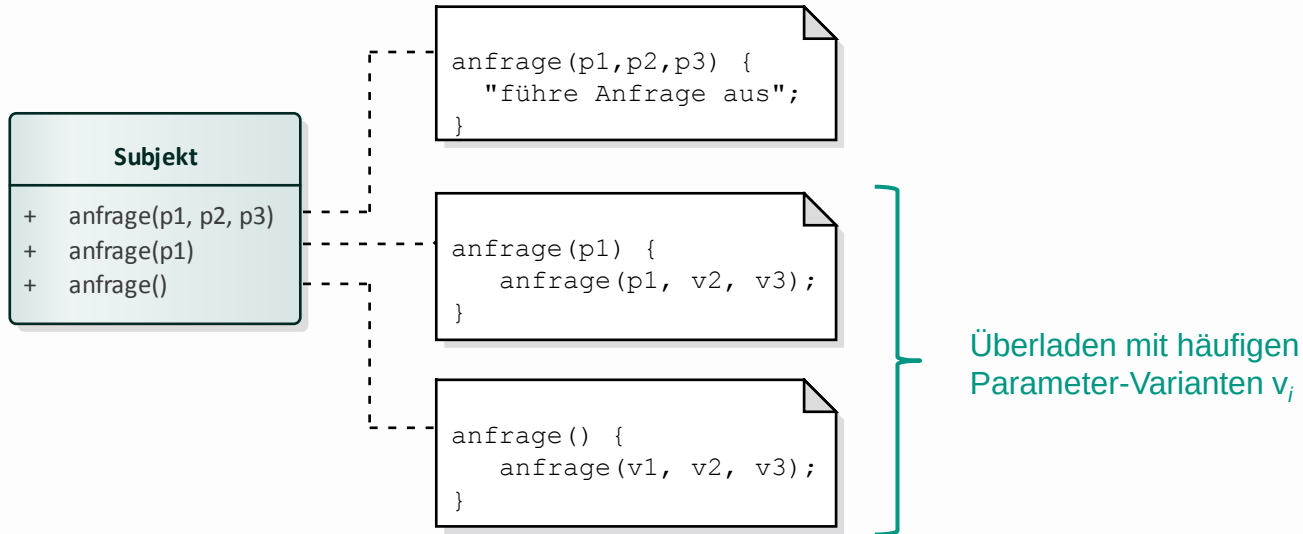


Zweck

Vereinfachen des **Methodenaufrufs** durch die Bereitstellung häufig genutzter Parameterkombinationen durch zusätzliche Methoden (**Überladen**).

Anwendbarkeit

Wenn Methodenaufrufe häufig mit den gleichen Parametern auftreten.



Bequemlichkeits-Klasse

**Bequemlichkeits-
Klasse**
(Convenience Class)



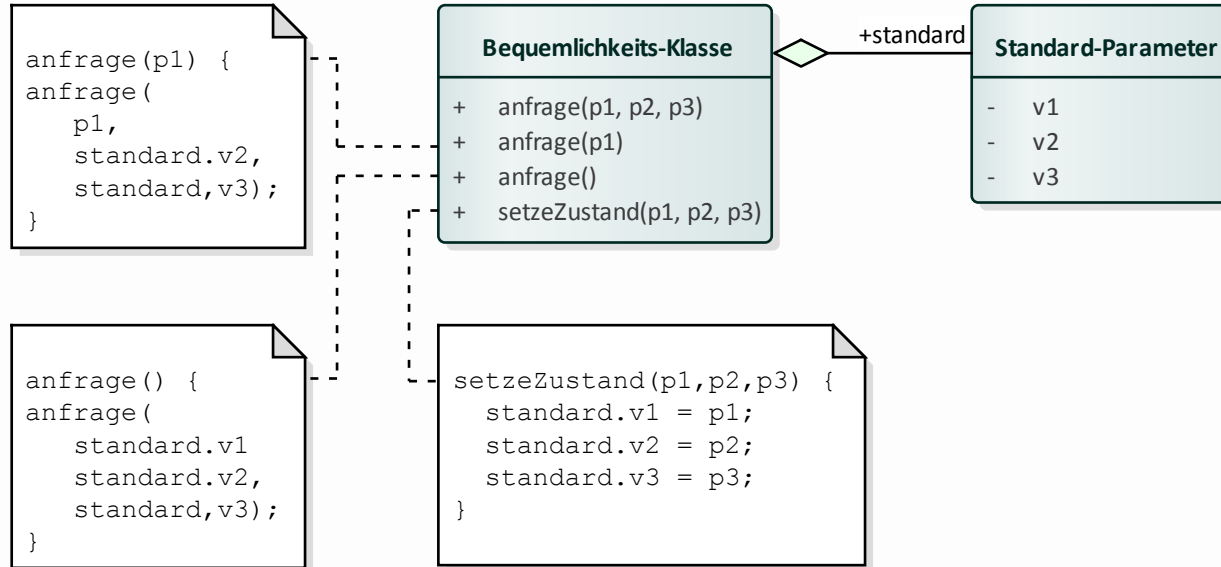
Zweck

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



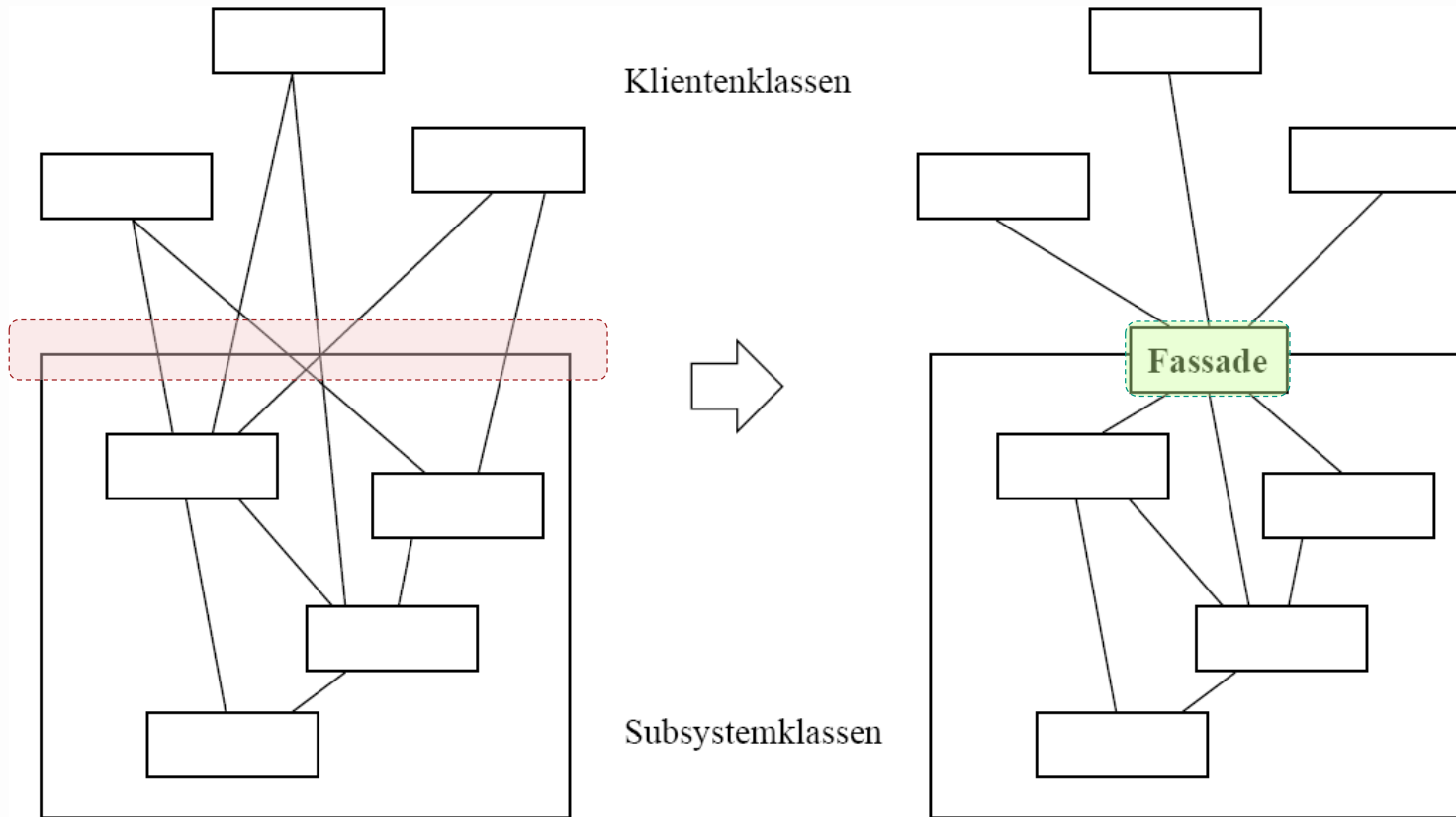


Zweck

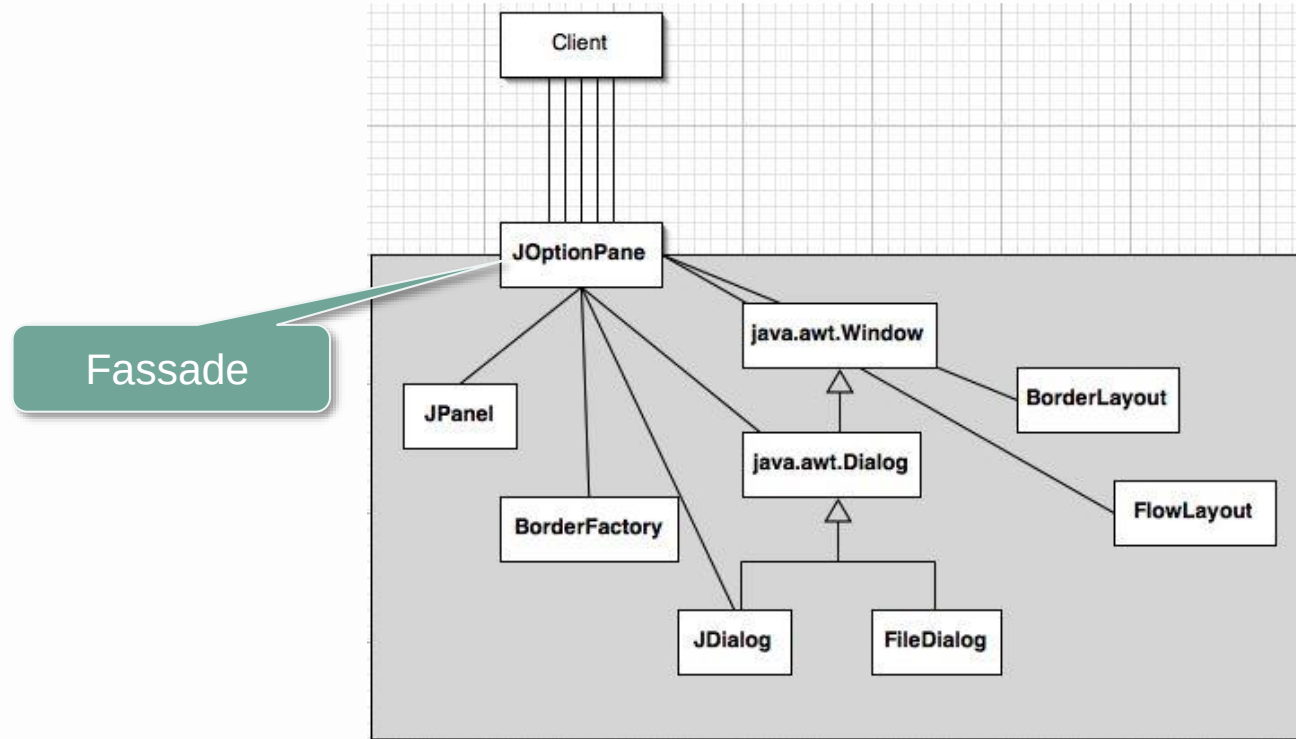
Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems.

Die Fassadenklasse definiert eine **abstrakte Schnittstelle**, welche die Benutzung des Subsystems vereinfacht.

Fassade



Beispiel (Java AWT)



Fassade

Anwendung

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

Null-Objekt (Null-Object)



Zweck

Stelle einen **Stellvertreter** zur Verfügung, der die gleiche Schnittstelle bietet, aber **nichts tut**.

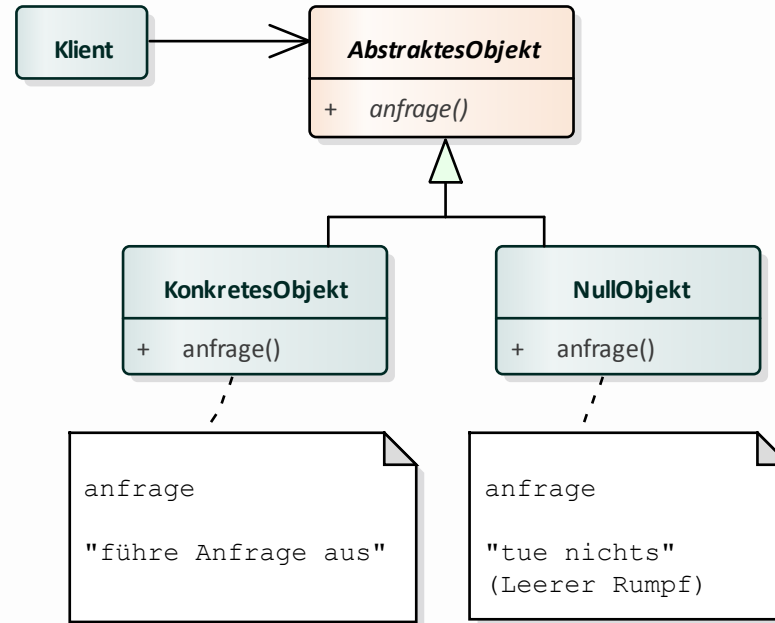
Das Null-Objekt kapselt die Implementierungs-Entscheidungen (wie genau es „nichts tut“) und versteckt diese Details vor seinen Mitarbeitern.

Motivation

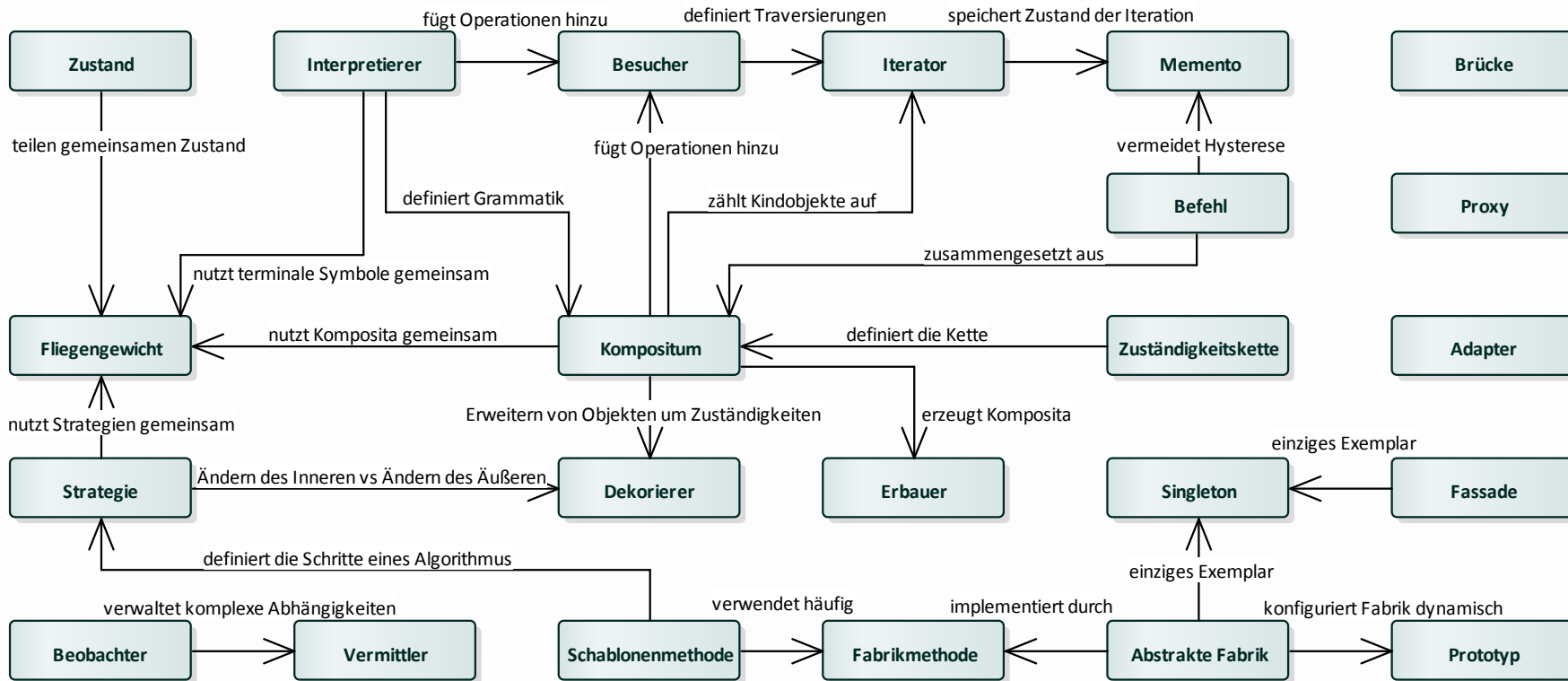
Es wird verhindert, dass der Code mit Tests gegen Null-Werte verschmutzt wird →

```
if (thisCall.callingParty != null) {  
    thisCall.callingParty.action()  
}
```

Notwendige Abfrage ohne Null-Objekt



- Wenn ein Objekt Mitarbeiter benötigt, und einer oder mehrere von ihnen **nichts tun** sollen.
- Wenn Klienten sich **nicht um den Unterschied kümmern** sollen zwischen einem echten Mitarbeiter und einem, der nichts tut.
- Wenn das „tue nichts“-Verhalten von verschiedenen Klienten **wieder verwendet** werden soll.



Quelle: GoF-Buch

Am Muster: Model-View-Controller (MVC)

Model-View-Controller-Muster (MVC Pattern)

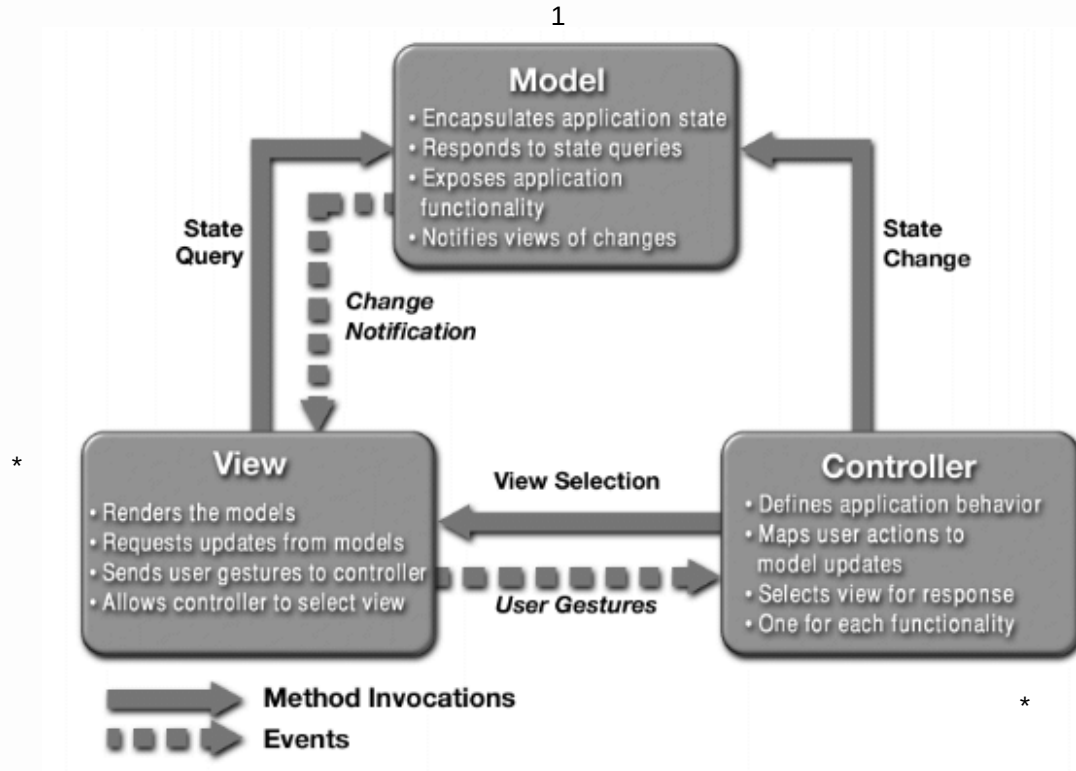


Das *Model-View-Controller-Muster* (**MVC**) unterteilt eine interaktive Anwendung in drei Komponenten:

Model	– (Modellschicht) Sie <i>enthält</i> die dauerhaften Daten einer Anwendung – verwaltet Daten unabhängig von der Präsentation
View	– (Darstellungsschicht) Sie <i>präsentiert</i> die Daten. – beinhaltet die visuellen Elemente (Fenster, Buttons, HTML, etc.)
Controller	– (Steuerungsschicht) Sie <i>realisiert</i> die <i>eigentliche Arbeit</i> – steuert den Ablauf — ändert Modelldaten – entscheidet, welche View aufgerufen wird

- MVC ist ein Entwurfsmuster, das die drei Entwurfsmuster **Beobachter**, **Kompositum** und **Strategie** kombiniert.

MVC Muster Abbildung

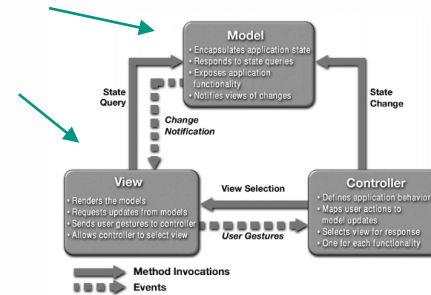


Quelle: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>

Muster 1 – Beobachter

Beispiel für Beobachter-Muster: Beziehung zwischen Modell und Sichten

- Es gibt eine Registrierungs- und Benachrichtigungs-Interaktion zwischen dem Modell (dem „Subjekt“) und den Sichten (den „Beobachtern“).
- Wenn Daten im Modell sich ändern, werden die Sichten benachrichtigt. Daraufhin aktualisiert jede Sicht sich selbst durch Zugriff auf das Modell.
- Die Sichten *wissen nichts voneinander*:
Das Modell weiß nicht, in welcher Weise die Sichten die Daten des Modells verwenden (**Entkopplung**).



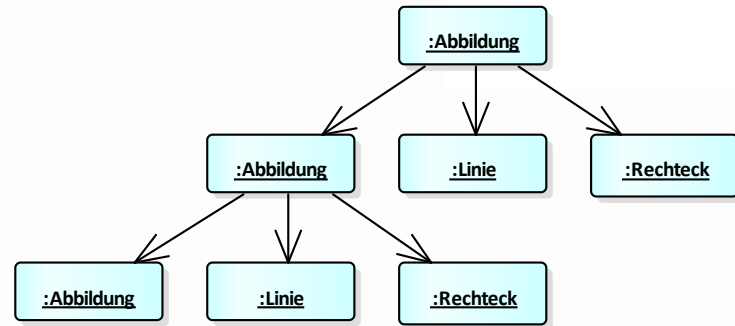
Muster 2 – Kompositum

Sichten können **geschachtelt** sein.

Beispiel für Kompositum-Muster:

Eine **zusammengesetzte Sicht**

- Z.B. kann ein Objektinspektor aus geschachtelten Sichten bestehen und kann selbst in der Benutzerschnittstelle eines Debuggers enthalten sein.
- Die Klasse **ZusammengesetzteSicht** ist eine Unterklasse von **Sicht**. Ein Exemplar von **ZusammengesetzteSicht** kann genauso wie ein Exemplar von **Sicht** benutzt werden. Es enthält und verwaltet geschachtelte Sichten.

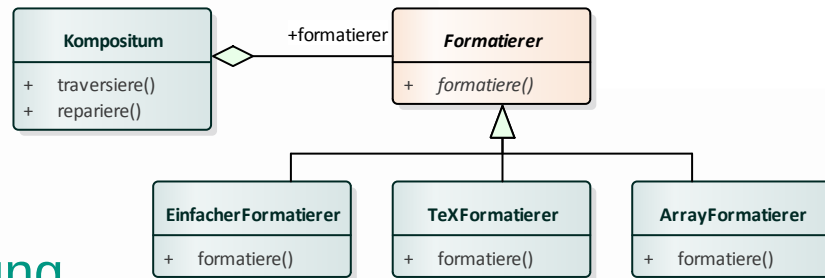


Muster 3 – Strategie

Beispiel für Strategie-Muster:

Beziehung zwischen Sicht und Steuerung

- Es gibt mehrere **Unterklassen von Steuerung**, die verschiedene **Antwortstrategien** implementieren.
Zum Beispiel werden Tastatureingaben unterschiedlich behandelt, oder ein Menü wird anstelle von Tastaturkommandos benutzt.
- Eine **Sicht wählt** eine gewisse Unterklasse aus, die die entsprechende Antwortstrategie implementiert. Diese kann auch dynamisch ausgewählt werden (zum Beispiel um ausgeschaltete Zustände zu ignorieren).



MVC-Pattern – Anwendung

Probleme bei der Anwendung

- Dieselbe Information hat in unterschiedlichsten Fenstern verschiedene Darstellungsweisen, z.B. Balken- oder Kuchendiagramme.
 - Grafische Darstellung und Verhalten der Anwendung müssen Datenänderungen sofort widerspiegeln.
- Unterstützung verschiedener Standards für Erscheinungsbild und Bedienmerkmale, ohne dass dabei der funktionale Kern geändert werden muss.
- Änderungen an der Benutzeroberfläche auch zur Laufzeit.

MVC-Pattern – Interaktion

Wenn ein Anwender das Model mit Hilfe der Steuerungskomponente einer Ansicht ändert, sollten alle Views des Models diese **Änderung anzeigen**:

1. Model benachrichtigt seine Views, sobald sich **Daten ändern**.
2. Views entnehmen die neuen Daten und **passen** die angezeigte **Information an**.

MVC-Pattern – Verwendung

Entwicklung von Benutzeroberflächen für **interaktive Anwendungen**

Beispiele

- Smalltalk-80-Umgebung
- Erstellung von Windows-Anwendungen in der Entwicklungsumgebung von Visual C++

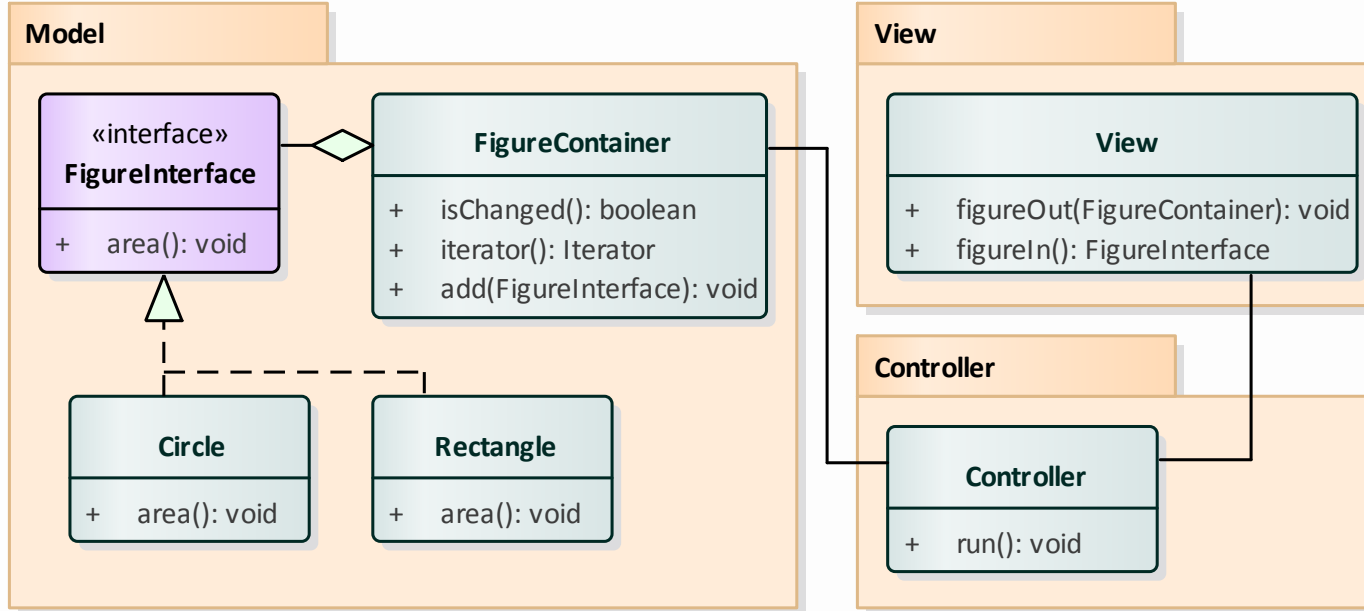
Vorteile

- **mehrere Views** desselben Modells
- **austauschbare Views**
- Potential für **Frameworks**

Nachteile

- größere **Komplexität**
- **ineffizienter Datenzugriff** innerhalb der Views
- etc.

Java Beispiel: Figure



Code: Model

```
class FigureContainer{
    List figures_;
    boolean isChanged_;
    FigureContainer(){
        figures_ = new ArrayList();
        isChanged_ = false;
    }
    void add(FigureInterface fig){
        figures_.add(fig);
        isChanged_ = true;
    }
    boolean isChanged(){
        return isChanged_;
    }
    Iterator iterator(){
        return figures_.iterator();
    }
}
```

Code: View

```
class View{
    View() {}

    FigureInterface figureIn(){
        return new Circle(10.0D * Math.random());
    }

    void figureOut(FigureContainer cont){
        Iterator itr = cont.iterator();
        System.out.println("FIGURE_CONTAINER:");
        while (itr.hasNext()){
            FigureInterface fig = (FigureInterface)itr.next();
            System.out.println(fig.getClass().getName());
        }
        System.out.println("END");
    }
}
```


Code: Controller

```
class Controller{
    View view_;
    FigureContainer container_;

    Controller(){
        view_ = new View();
        container_ = new FigureContainer();
    }
    void run(){
        for (int i = 1; i < 5; i++){
            container_.add(view_.figureIn());
        }
        if (container_.isChanged()){
            view_.figureOut(container_);
        }
    }
}
```

Code: Figure Beispiel

```
public class FigureViewExample {  
  
    public static void main(String[] args) {  
        Controller controller = new Controller();  
        controller.run();  
  
        System.out.println("finish");  
    }  
  
}
```

Fragen?

