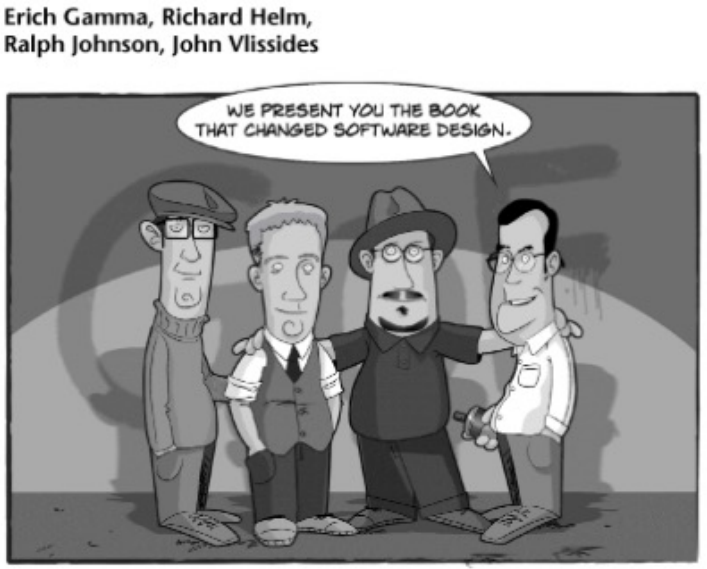
Softwaretechnik 2

Entwurfsmuster



Fachhochschule
Dortmund
University of Applied Sciences and Arts

Struktur

1. Einführung
2. Architekturmodellierung / Grobentwurf
3. Architekturstile/–muster I
4. Architekturstile II
5. Architekturstile III
6. OOD der Businesslogik / Fachlogik
7. Entwurfsmuster

Struktur

7. Entwurfsmuster

7.1. Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2. Beispiele für Muster

- 7.2.1. Fabrikmethode-Muster (klassenbasiertes Erzeugungsmuster)
- 7.2.2. Singleton-Muster (objektbasiertes Erzeugungsmuster)
- 7.2.3. Kompositum-Muster (objektbasiertes Strukturmuster)
- 7.2.4. Proxy-Muster (objektbasiertes Strukturmuster)
- 7.2.5. Fassaden-Muster (objektbasiertes Strukturmuster)
- 7.2.6. Beobachter-Muster (objektbasiertes Verhaltensmuster)
- 7.2.7. Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)
- 7.2.8. Zustands-Muster (objektbasiertes Verhaltensmuster)
- 7.2.9. ….

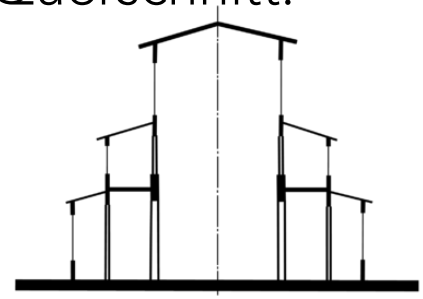
Entwurfsmuster (design patterns)

- Bewährte, generische Lösung für ein immer wiederkehrendes Entwurfsproblem
- Zusammengefasst in einem Standardwerk von 1994:
  Design Patterns. Elements of Reusable Object-Oriented Software
- Autoren (GoF, Gang of Four):

Von Links nach rechts: Ralph Johnson, Erich Gamma, Richard Helm und John Vlissides († 2005)

Kölner Dom – Architekturstil/-muster

- Seit 1996 UNESCO Weltkulturerbe
- Erbaut aus Londorfer Basalt, einem Lava-gestein aus dem Vogelsberg-Massiv
- Der Dom weist eine Außenlänge von 144,58 Metern und eine Höhe von 157,38 Meter auf.
- Gotische Kathedrale mit dem Grundriss fünfschiffige Basilika mit einem ausladenden Querhaus.
- Bauschema einer klassischen fünfschiffigen Basilika im Querschnitt:



Das mittlere Hauptschiff ist höher als die niedrigen Seitenschiffe.

Nordportal des Kölner Doms

Entwurfsmuster in der Architektur

- Hauptportal im nördlichen Querhaus des Kölner Doms
- Michaelsportal (Mitte)

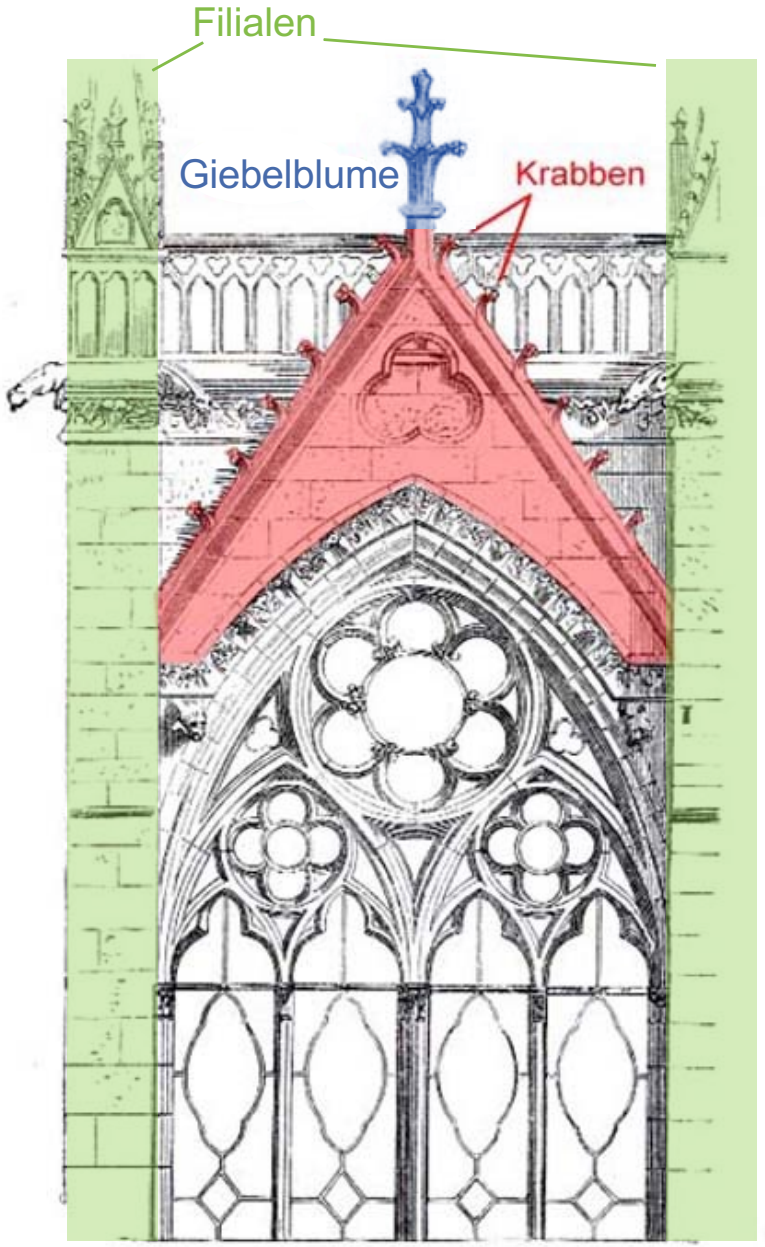
- Wie würden Sie dieses Portal beschreiben?

Entwurfsmuster in der Architektur

- der Fachmann (Architekt) sagt:
  - Dombaumeister Zwirner entwarf die 1843-1855 errichtete neugotische Nordfassade
  - über dem Michelsportal ragt ein hoher **Wimperg** mit fünf Statuen auf
    - Ambrosius
    - Gregor der Große
    - Auferstandener Christus
    - Augustinus
    - Hieronymus
  - der Wimperg wird von **Filialen** flankiert und besitzt deutlich ausgearbeitete **Krabben**, sowie eine Kreuzblume als **Giebelblume**

Entwurfsmuster in der Architektur

- Der Wimperg (auch Wimberg) ist ein giebelförmiges Bauteil der Gotik zur Bekrönung von Portalen und Fenstern.
- Die Schrägen des Wimpergs sind mit Krabben und
- oft von zwei kleinen Türmchen, den Fialen geschmückt.
- Die Spitze wird häufig mit einer Giebelblume abgeschlossen.



[www.architektur-lexikon.de]

Entwurfsmuster in der Softwaretechnik (design patterns)

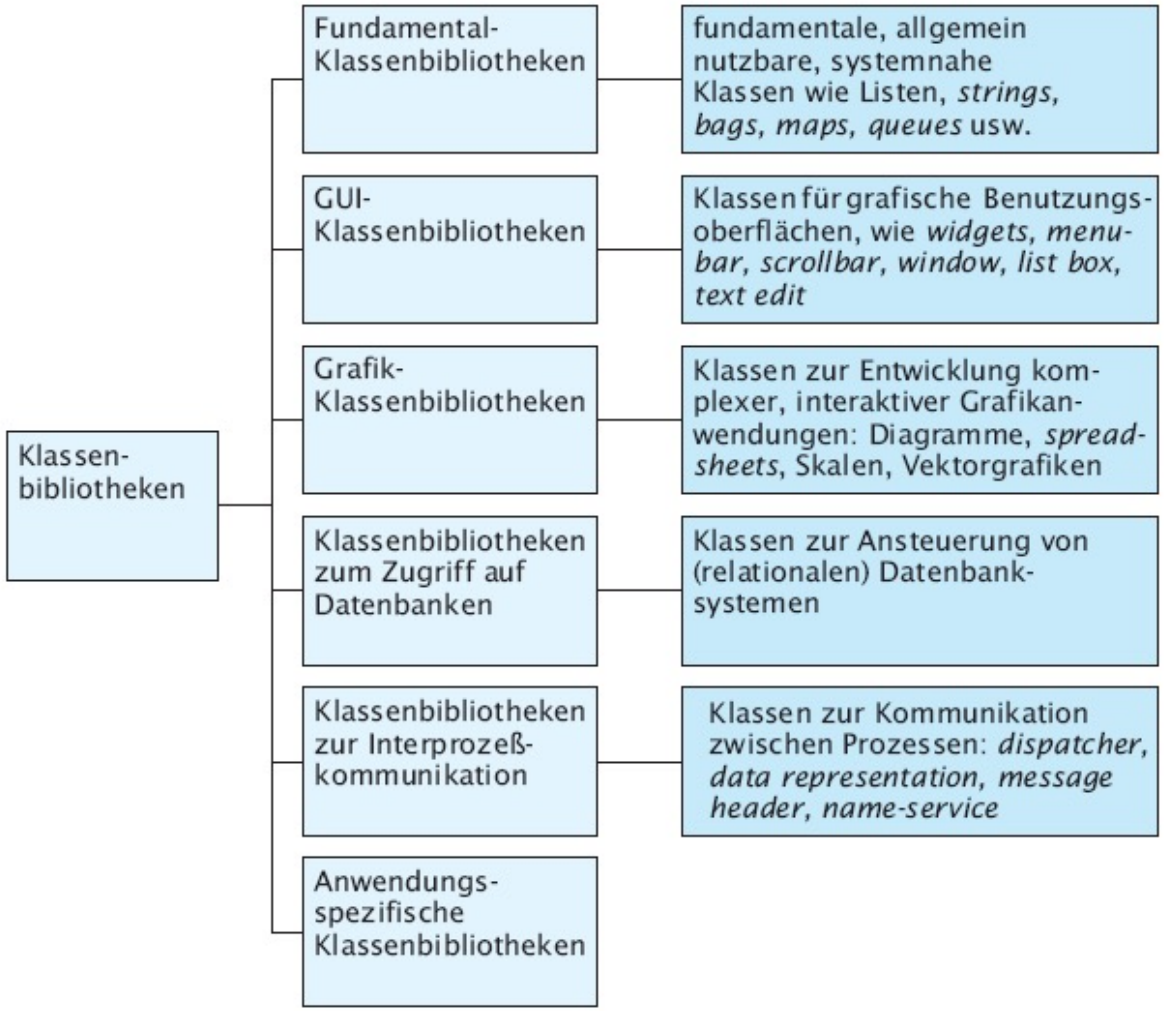
- **Beschreibung eines Musters**
  - Name
    - Beschreibt ein Entwurfsproblem, seine Lösung und Konsequenzen mit einem oder zwei Wörtern
  - Problembeschreibung
    - Gibt an, wann das Muster anwendbar ist
    - Problem und Kontext
    - Auch Erklärung spezifischer Entwurfsprobleme
  - Lösungsbeschreibung
    - Kein konkreter Entwurf und keine Implementierung
    - Abstrakte Beschreibung des Entwurfsproblems
    - Beschreibt allgemeine Anordnung der Klassen bzw. Objekte
  - Konsequenzen
    - Zeit- und Speichereffizienz
    - Sprach- und Implementierungseigenschaften
    - Auswirkungen auf Flexibilität, Erweiterbarkeit und Portierbarkeit

Entwurfsmuster (design patterns)

- Klassifikation der Muster
  - Erzeugungsmuster (creational patterns)
    Helfen, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden
  - Strukturmuster (structural patterns)
    Befassen sich mit der Zusammensetzung von Klassen und Objekten zu größeren Strukturen
  - Verhaltensmuster (behavioral patterns)
    - Befassen sich mit der Interaktion zwischen Objekten und Klassen
    - Beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachvollziehbar sind

- Weitere Klassifikation der Muster
  - Klassenbasierte Muster
    -Behandeln Beziehungen zwischen Klassen
    - Ausgedrückt durch Generalisierungsstrukturen
    - Festgelegt zur Übersetzungszeit
  - Objektbasierte Muster
    - Beschreiben Beziehungen zwischen Objekten, die zur Laufzeit geändert werden können
    - Benutzen auch bis zu einem gewissen Grad die Generalisierung

Klassenbiliothek



Klassenbibliothek

Beispiel für eine fundamentale Klassenbibliothek

Die C++-Standardbibliothek bietet beispielsweise:

- Container
- Iteratoren
- Algorithmen
- Funktionsobjekte
- Zeichenketten
- Eingabe und Ausgabe
- Lokalisierung
- Numerik
- Ausnahmen
- RunTime Type Information

Sequenzielle Container

Name	Klassenname	Beschreibung
Felder dynamischer Größe, Vector	`std::vector`	Einfügen und Löschen am Ende ist in $\mathcal{O}(1)$ und für anderen Elemente in $\mathcal{O}(n)$ möglich. Der Container unterstützt wahlfreien Zugriff (Random Access) in $\mathcal{O}(1)$.
Felder fester Größe	`std::tr1::array`	Erst seit *technical review 1* verfügbar.
doppelt verkettete Listen	`std::list`	Einfügen und Löschen ist in $\mathcal{O}(1)$ möglich. Wahlfreier Zugriff ist nicht möglich.
Warteschlangen	`std::queue`	Der Container unterstützt keine Iteratoren.
Warteschlangen mit zwei Enden	`std::deque`	Der Datentyp verhält sich wie der Vector, kann jedoch Elemente am Anfang und Ende in $\mathcal{O}(1)$ einfügen.
Warteschlangen mit Prioritäten	`std::priority_queue`	Die Struktur garantiert wie der Heap, dass immer das Element mit der höchsten Priorität am Beginn steht.
Stapel	`std::stack`	Der Container unterstützt keine Iteratoren.

etc.

Framework (1/2)

- Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren
- Besteht aus konkreten und – insbesondere – aus abstrakten Klassen, die Schnittstellen definieren
- Definition von Unterklassen zur Verwendung und Anpassung des Frameworks
  - Selbstdefinierte Unterklassen empfangen Botschaften von vordefinierten Framework-Klassen
  - Hollywood-Prinzip: »Don't call us, we'll call you«.
- Ist immer spezifisch auf einen Anwendungsbereich ausgelegt
  - Beispiele
    - Erstellung grafischer Editoren
    - Erstellung von Finanzsoftware
- Spezialisierung für eine konkrete Anwendung durch Ableiten von Unterklassen aus den abstrakten Framework-Klassen
- Realisierung der Frameworks mittels Programmiersprachen
  Frameworks können also ausgeführt und direkt wiederverwendet werden

Framework (2/2)

Framework

- Ermöglicht hohe Wiederverwendung
- Bestimmt die Architektur der Anwendung
- Definiert die Struktur der Klassen und Objekte und deren Verantwortlichkeiten
- Legt fest, wie Klassen und Objekte zusammenarbeiten
- Legt fest, wie der Kontrollfluss aussieht
- Anwendungsprogrammierer kann sich auf die Details der Anwendung konzentrieren

Muster vs. Framework

- Entwurfsmuster sind abstrakter als Frameworks
  - Werden nur beispielhaft durch Programmcode repräsentiert
  - Anwendung von Entwurfsmustern mit einer neuen Implementierung verbunden
- Entwurfsmuster sind kleiner als Frameworks
  - Ein typisches Framework enthält mehrere Entwurfsmuster
- Entwurfsmuster sind weniger spezialisiert als Frameworks
  - Keine Beschränkung auf einen bestimmten Anwendungsbereich

Entwurfsmuster

Die klassischen Entwurfsmuster

Erzeugende Muster	Strukturelle Muster	Verhaltensmuster
Singleton (Einzelstück)	**Facade(Fassade)**	Mediator (Vermittler)
Prototype (Prototyp)	Decorator (Dekorierer)	Iterator
Factory Method (Fabrikmethode)	Bridge (Brücke)	Interpreter
Builder (Erbauer)	**Composite (Kompositum)**	Command (Kommando)
Abstract Factory (Abstrakte Fabrik)	Adapter	Chain of Responsibility (Zustandigkeitskette)
	Flyweight (Fliegengewicht)	Memento
	Proxy (Stellvertreter)	**Observer (Beobachter)**
		State (Zustand)
		Strategy (Strategie)
		Template Method (Schablonenmethode)
		Visitor (Besucher)

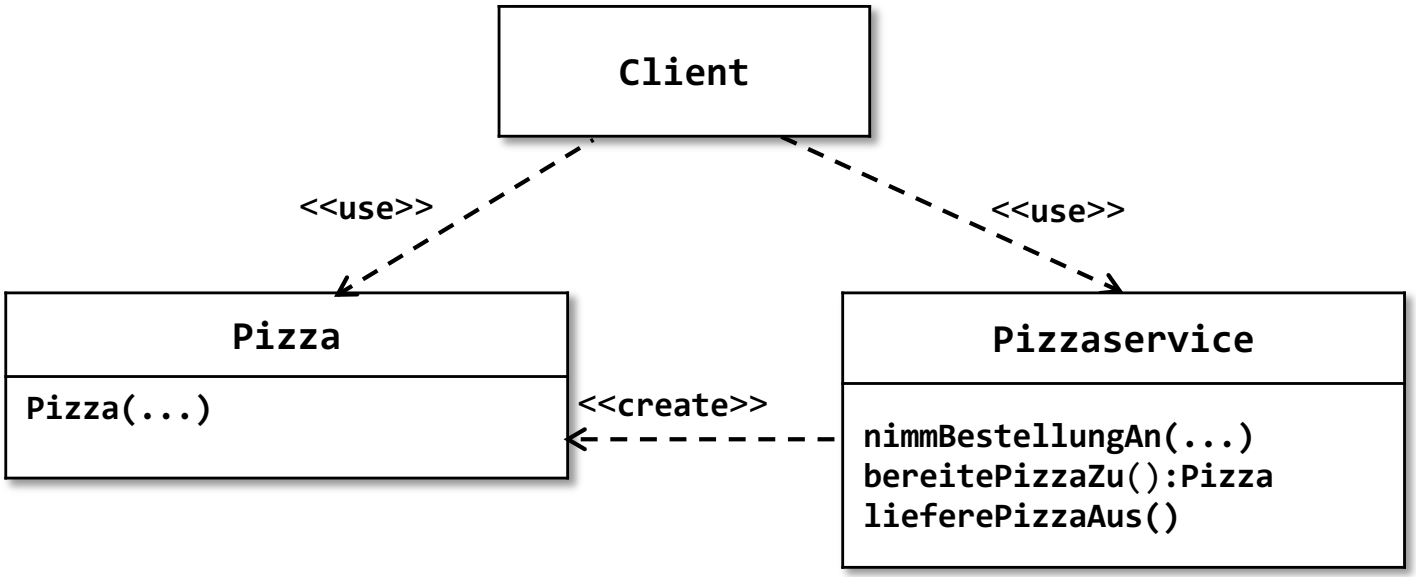
Struktur

7. Entwurfsmuster

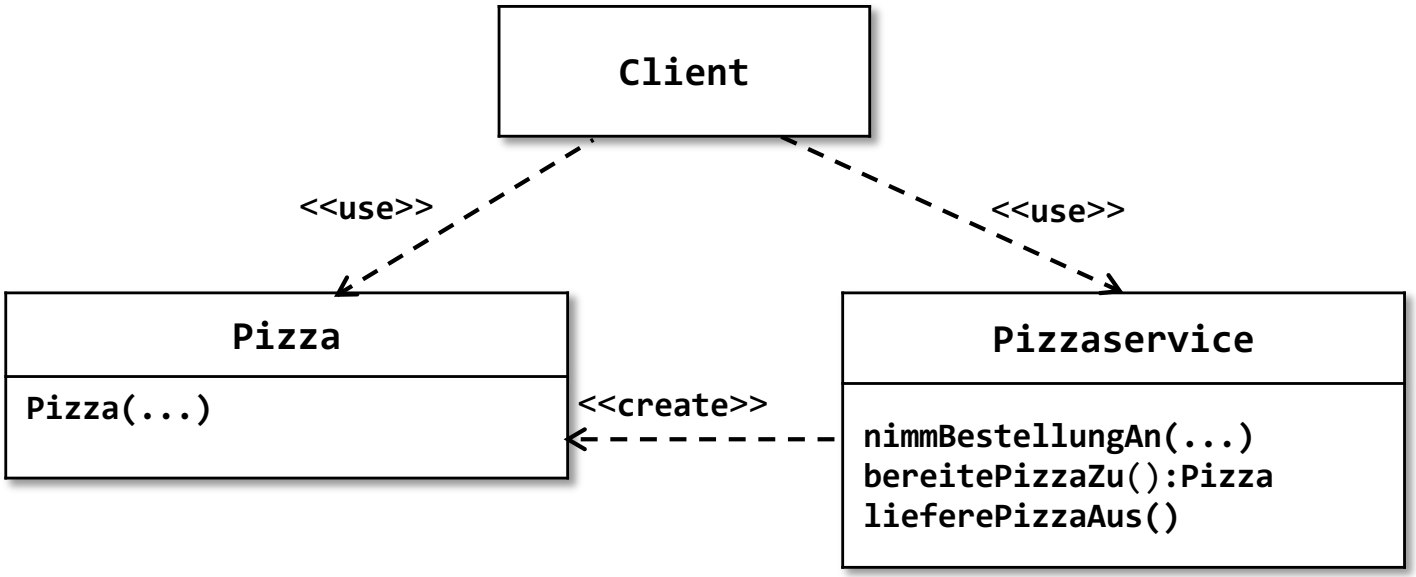
7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2. Beispiele für Muster

Fabrikmethode (klassenbasiertes Erzeugungsmuster)

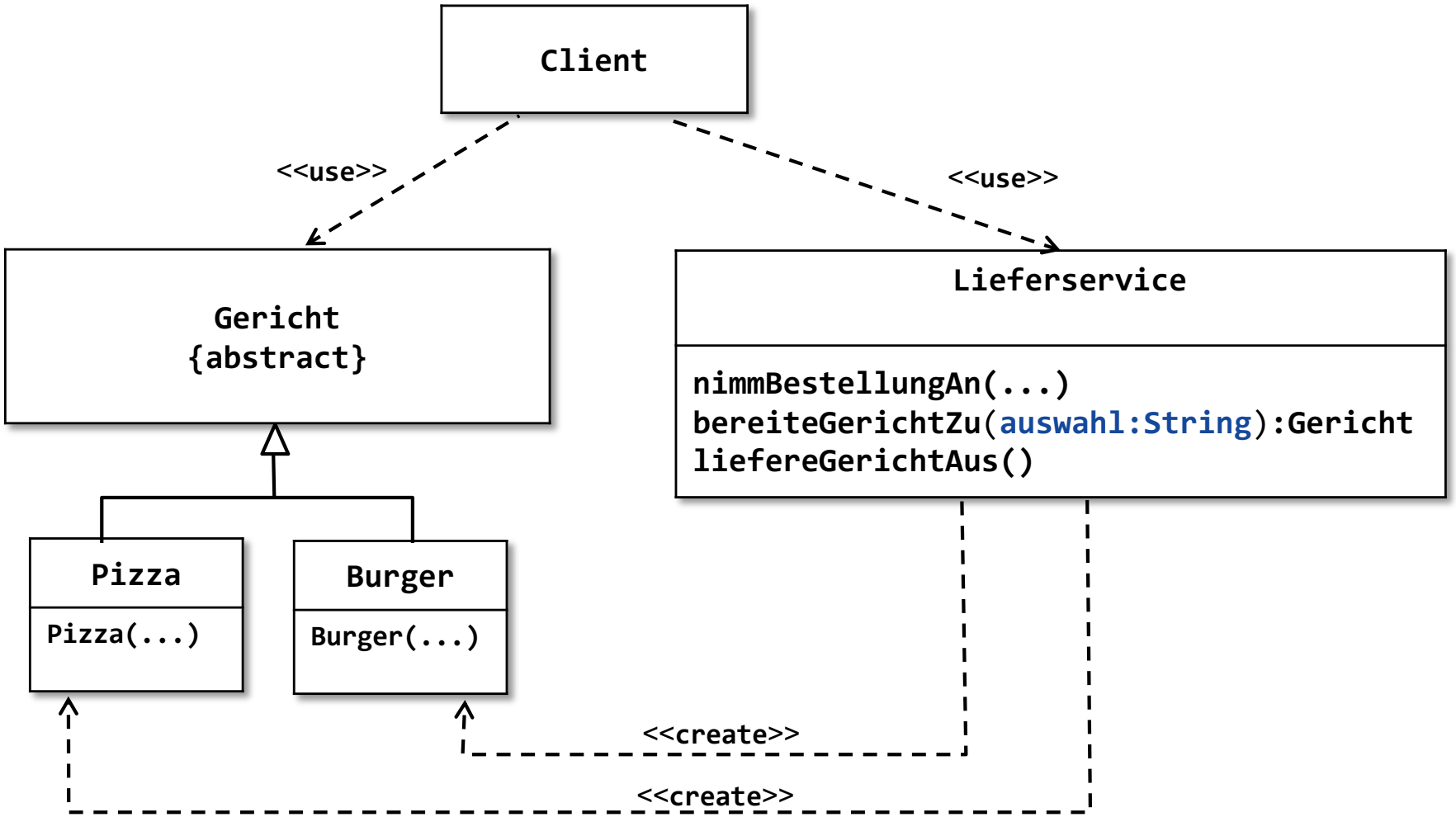


Fabrikmethode (klassenbasiertes Erzeugungsmuster)

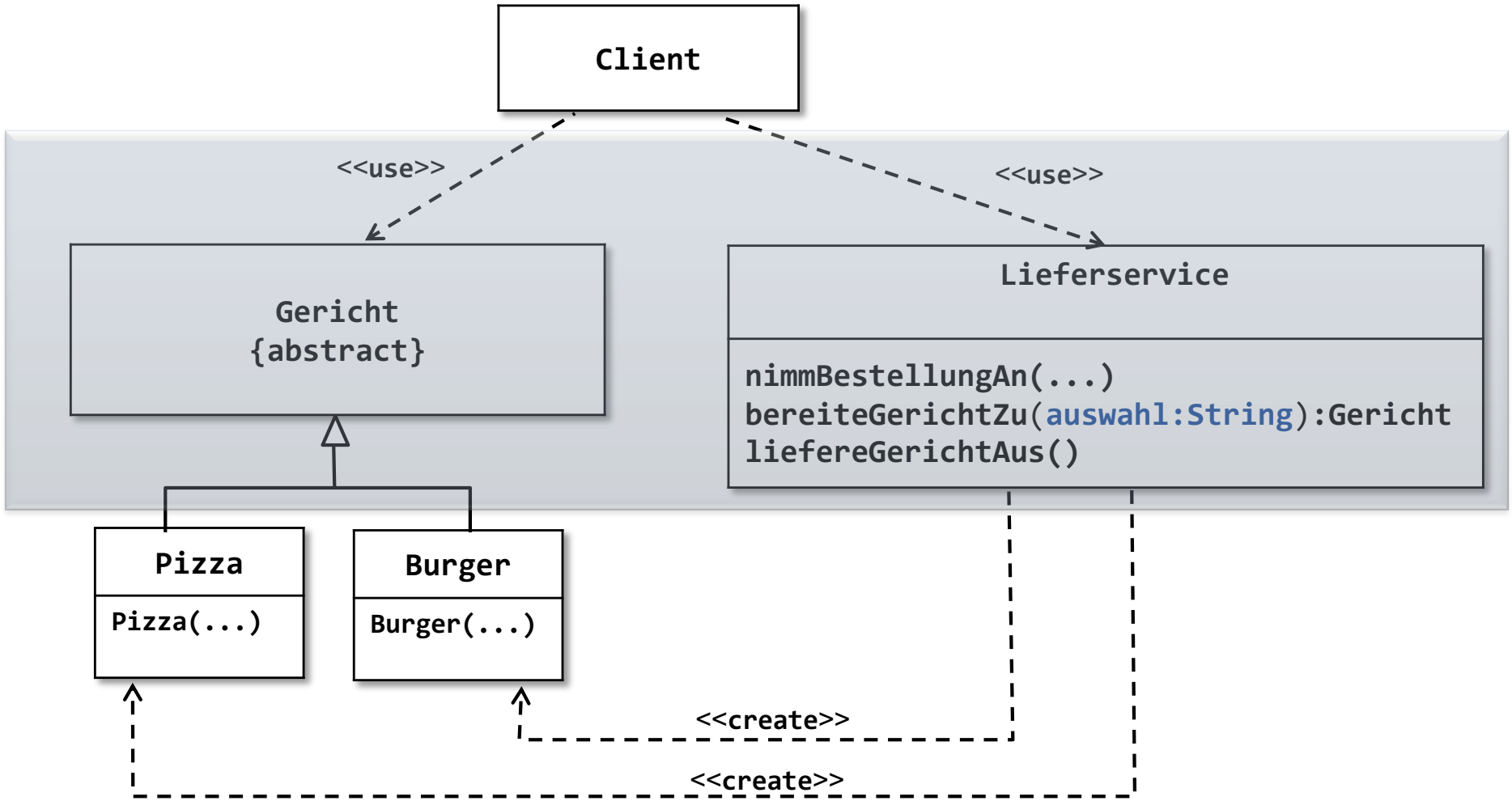


Wie sieht das Ganze bei verschiedenen Gerichten aus?

Fabrikmethode (klassenbasiertes Erzeugungsmuster)

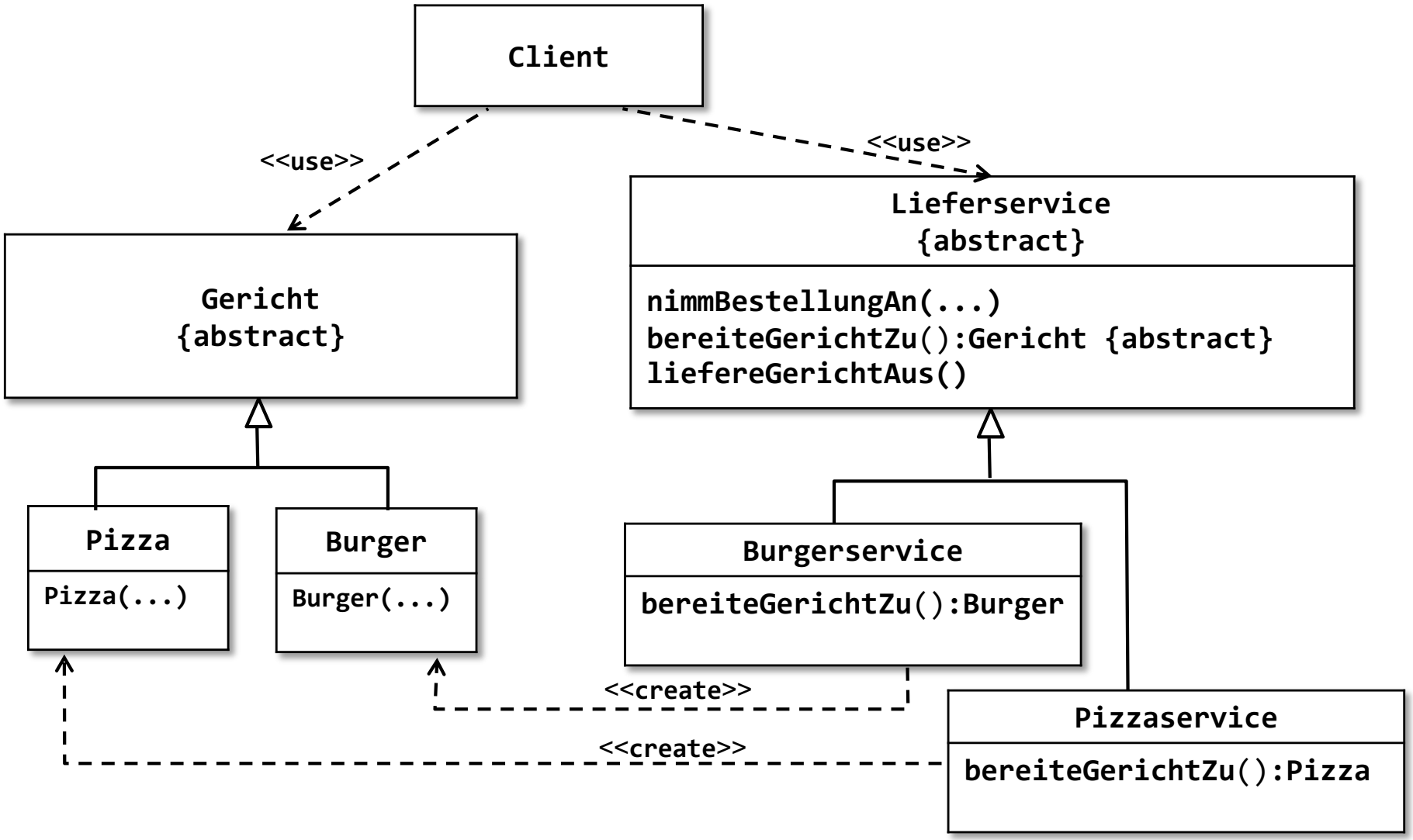


Fabrikmethode (klassenbasiertes Erzeugungsmuster)



Als Framework geeignet? ☐ ja ☐ nein ☐ vielleicht

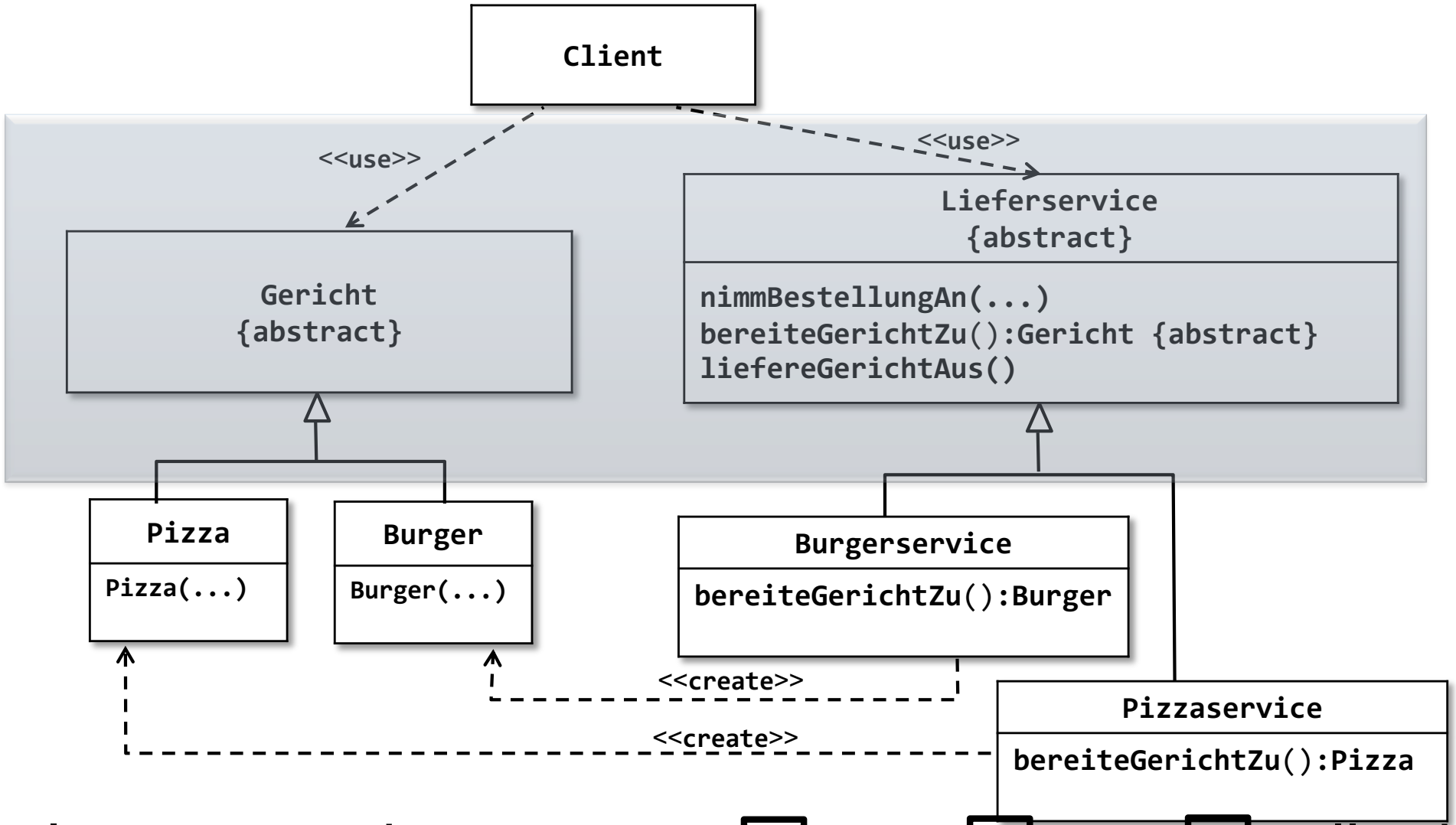
Fabrikmethode (klassenbasiertes Erzeugungsmuster)



Fabrikmethode (klassenbasiertes Erzeugungsmuster)

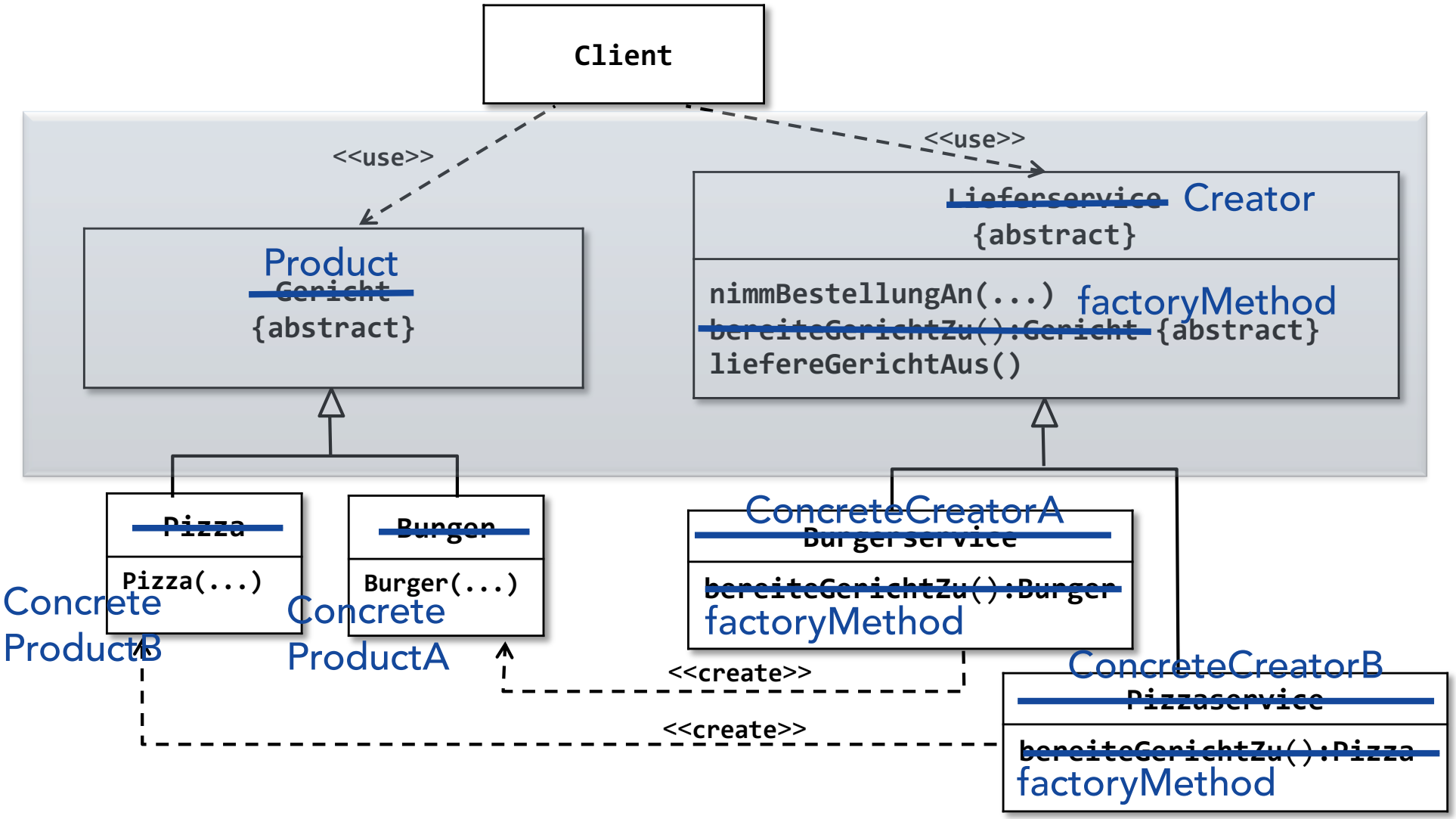
```
public class Client {

  static Lieferservice l;

  public static void main(String [] argv) {

    if (argv[0] == "pizza") {
      l = new Pizzaservice();
    }
    if (argv[0] == "burger")  {
      l = new Burgerservice();
    }

    Gericht g = l.bereiteGerichtZu();
  }
}
```

Fabrikmethode (klassenbasiertes Erzeugungsmuster)



Als Framework geeignet? ☐ ja ☐ nein ☐ vielleicht

Fabrikmethode (klassenbasiertes Erzeugungsmuster)



Fabrikmethode (klassenbasiertes Erzeugungsmuster)

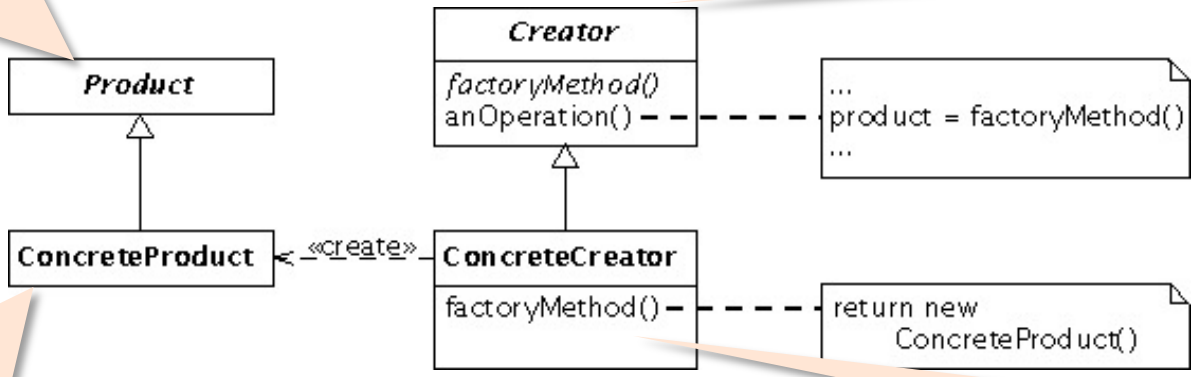
Anwendbarkeit

- Verwendung des Musters, wenn
  - eine Klasse die von ihr zu erzeugenden Objekte nicht im voraus kennen kann
  - die Unterklassen festlegen sollen, welche Objekte sie erzeugen

Definiert die Schnittstelle der Objekte, die von der Fabrikmethode erzeugt werden.

Deklariert die abstrakte Fabrikmethode.

Struktur



Implementiert die Schnittstelle des Products.

Überschreibt die Fabrikmethode, so dass sie ein Objekt von ConcreteProduct zurückgibt.

Fabrikmethode (klassenbasiertes Erzeugungsmuster)

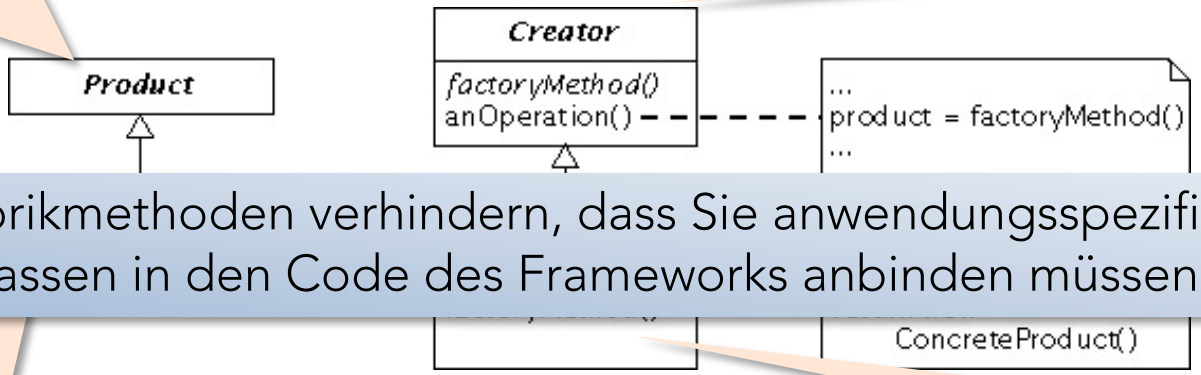
Anwendbarkeit

- Verwendung des Musters, wenn
  - eine Klasse die von ihr zu erzeugenden Objekte nicht im voraus kennen kann
  - die Unterklassen festlegen sollen, welche Objekte sie erzeugen

Definiert die Schnittstelle der Objekte, die von der Fabrikmethode erzeugt werden.

Deklariert die abstrakte Fabrikmethode.

Struktur



⇒ Fabrikmethoden verhindern, dass Sie anwendungsspezifische Klassen in den Code des Frameworks anbinden müssen!

Implementiert die Schnittstelle des Products.

Überschreibt die Fabrikmethode, so dass sie ein Objekt von ConcreteProduct zurückgibt.

Struktur

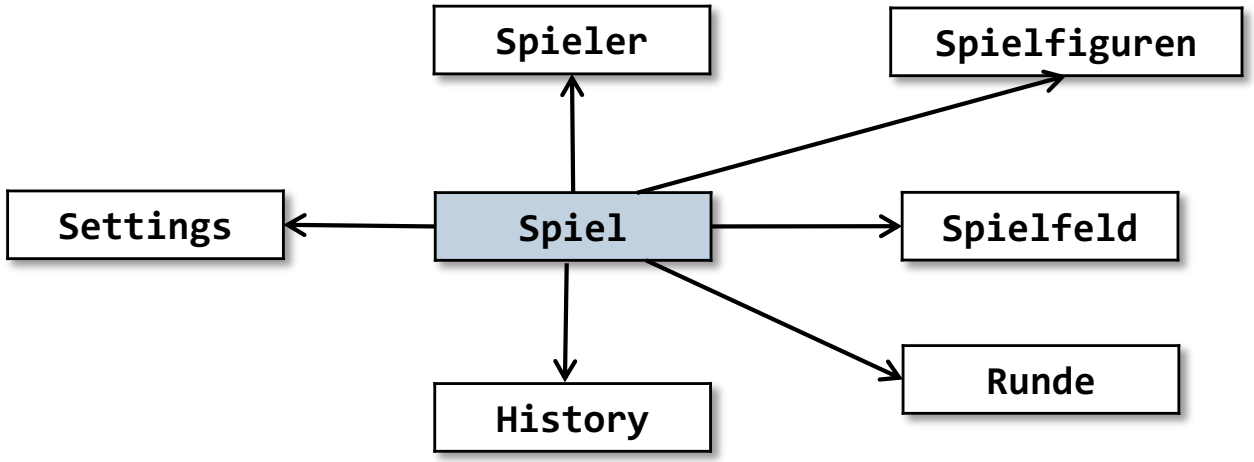
7. Entwurfsmuster

7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2. Beispiele für Muster

Singleton (objektbasiertes Erzeugungsmuster)

Problemstellung zentrales Objekt,
das eigentlich überall zugreifbar sein muss.



Singleton (objektbasiertes Erzeugungsmuster)

Anwendbarkeit

- Verwenden Sie dieses Muster, wenn
  - es genau ein Objekt eine Klasse geben soll und ein einfacher Zugriff darauf bestehen soll
  - das einzige Exemplar durch Spezialisierung mittels Unterklassen erweitert wird und Klienten das erweiterte Exemplar verwenden können, ohne ihren Code zu ändern
  - => einfache Implementierung

Schalter um Verfeinerungsoption abzuschalten

```
public final class Singleton {
   public static final Singleton uniqueInstance = new Singleton();
   private Singleton(){   } //Konstruktor
}
```

Singleton (objektbasiertes Erzeugungsmuster)

Anwendbarkeit

- Verwenden Sie dieses Muster, wenn
  - es genau ein Objekt eine Klasse geben und ein einfacher Zugriff darauf bestehen soll
  - das einzige Exemplar durch Spezialisierung mittels Unterklassen erweitert wird und Klienten das erweiterte Exemplar verwenden können, ohne ihren Code zu ändern
  - => kein öffentlicher Zugriff, sondern über Methode + Erzeugung erst bei Zugriff

Schalter um Verfeinerungsoption abzuschalten

```
public final class Singleton {
   private static final Singleton uniqueInstance = null;
   private Singleton(){  } //Konstruktor

   public static Singleton getInstance(){
      if (uniqueInstance == null){
         uniqueInstance = new Singleton();
      }
      return uniqueInstance;
   }
}
```

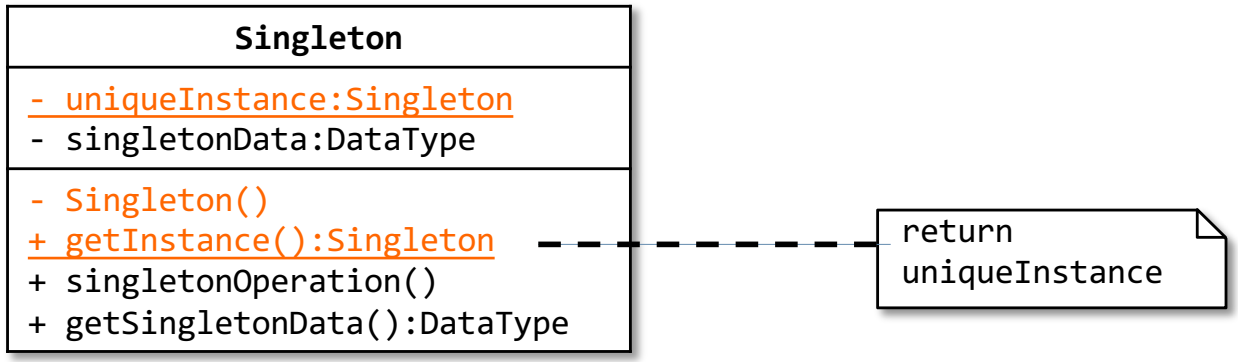
Singleton (objektbasiertes Erzeugungsmuster)

Anwendbarkeit

- Verwenden Sie dieses Muster, wenn
  - es genau ein Objekt für eine Klasse geben und ein einfacher Zugriff darauf bestehen soll
  - das einzige Exemplar durch Spezialisierung mittels Unterklassen erweitert wird und Klienten das erweiterte Exemplar verwenden können, ohne ihren Code zu ändern

  => zusätzlich noch Daten im Objekt

Struktur

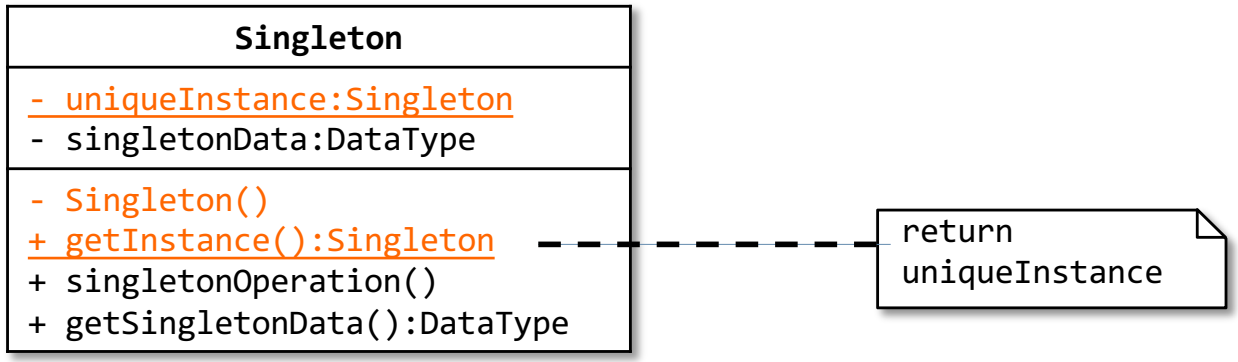


Singleton (objektbasiertes Erzeugungsmuster)

Anwendbarkeit

- Verwenden Sie dieses Muster, wenn
  -
  - ⇒ Verbesserung gegenüber globalen Variablen
  - ⇒ Singleton-Klasse kann durch Unterklassen spezialisiert werden

Struktur



Struktur

7. Entwurfsmuster

7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2. Beispiele für Muster

Kompositum-Muster (objektbasiertes Strukturmuster)

Matruschkas

Stapelkisten

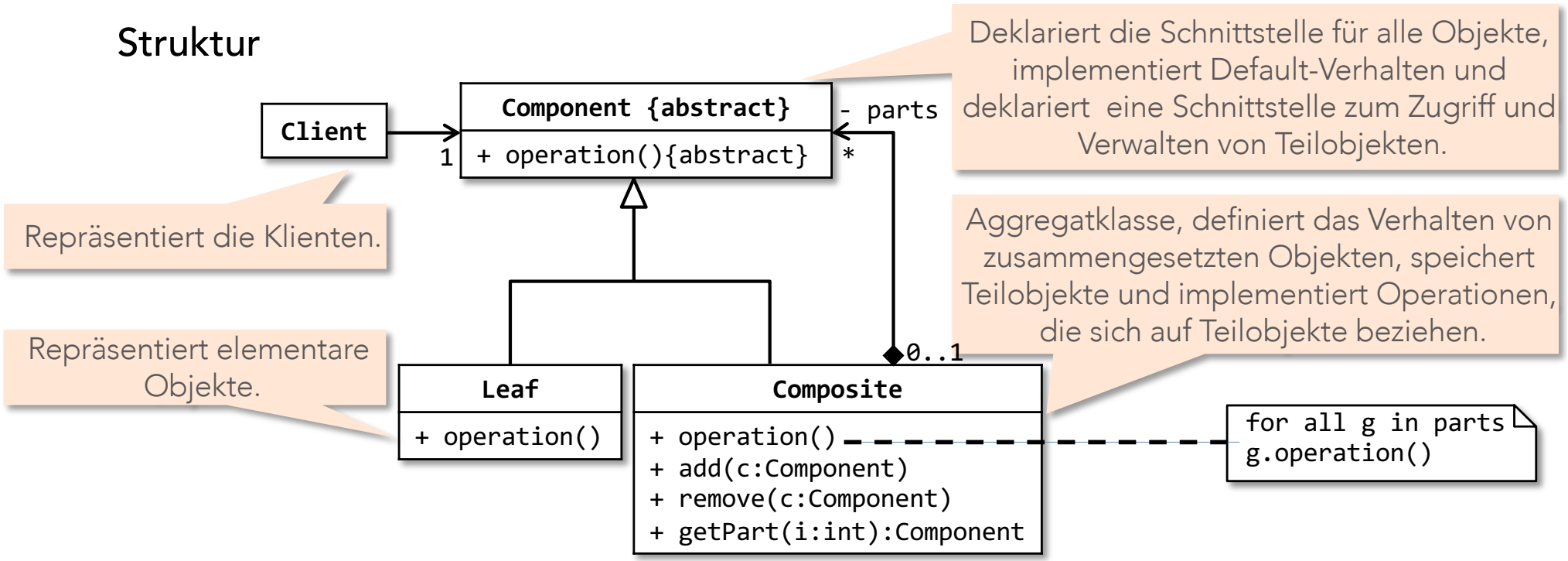
Frischhalteboxen

Kompositum-Muster (objektbasiertes Strukturmuster)

Anwendbarkeit

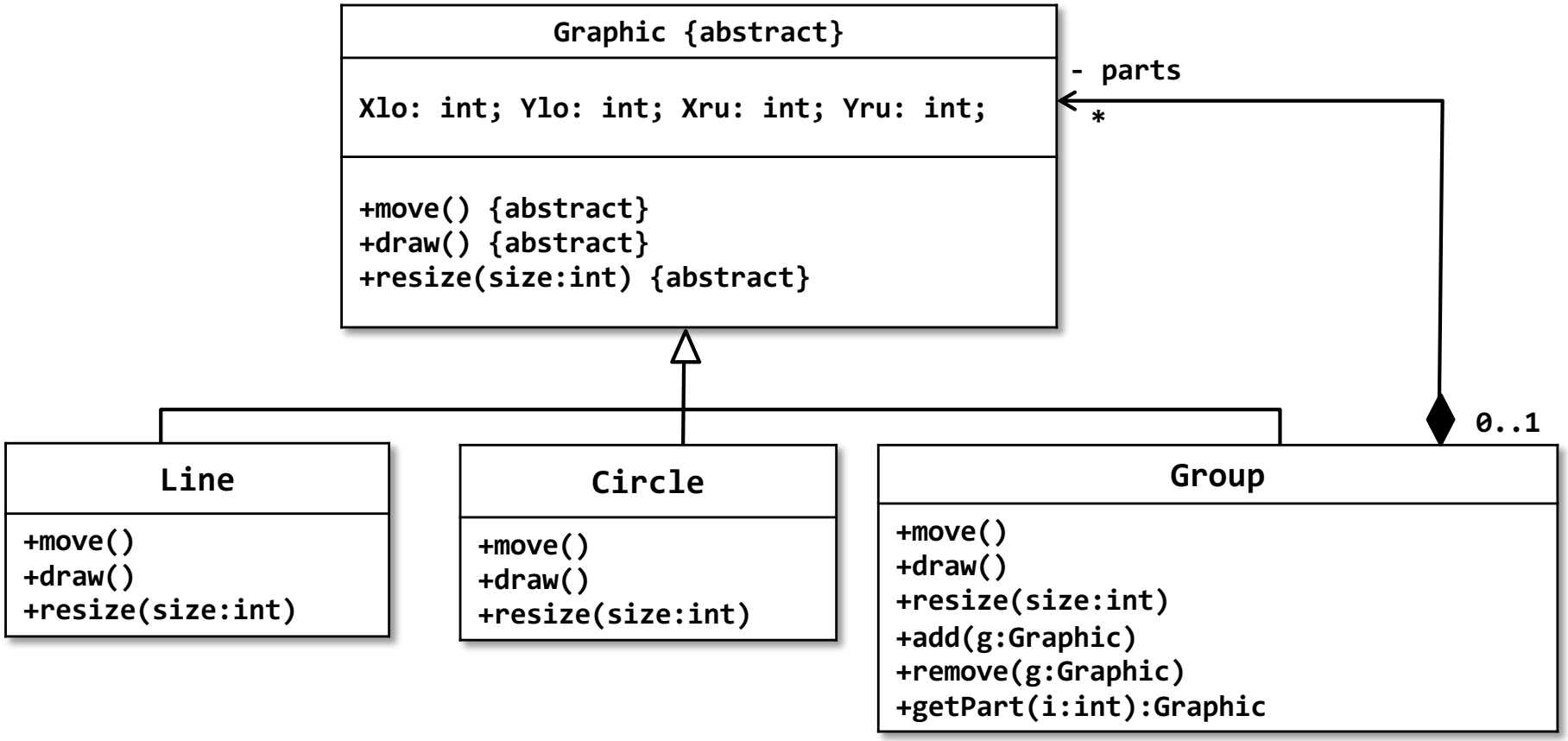
- Verwenden Sie dieses Muster, wenn
  - Sie whole-part-Hierarchien von Objekten darstellen wollen
  - die Klienten keinen Unterschied zwischen elementaren und zusammengesetzten Objekten wahrnehmen und alle Objekte gleich behandeln sollen

Struktur



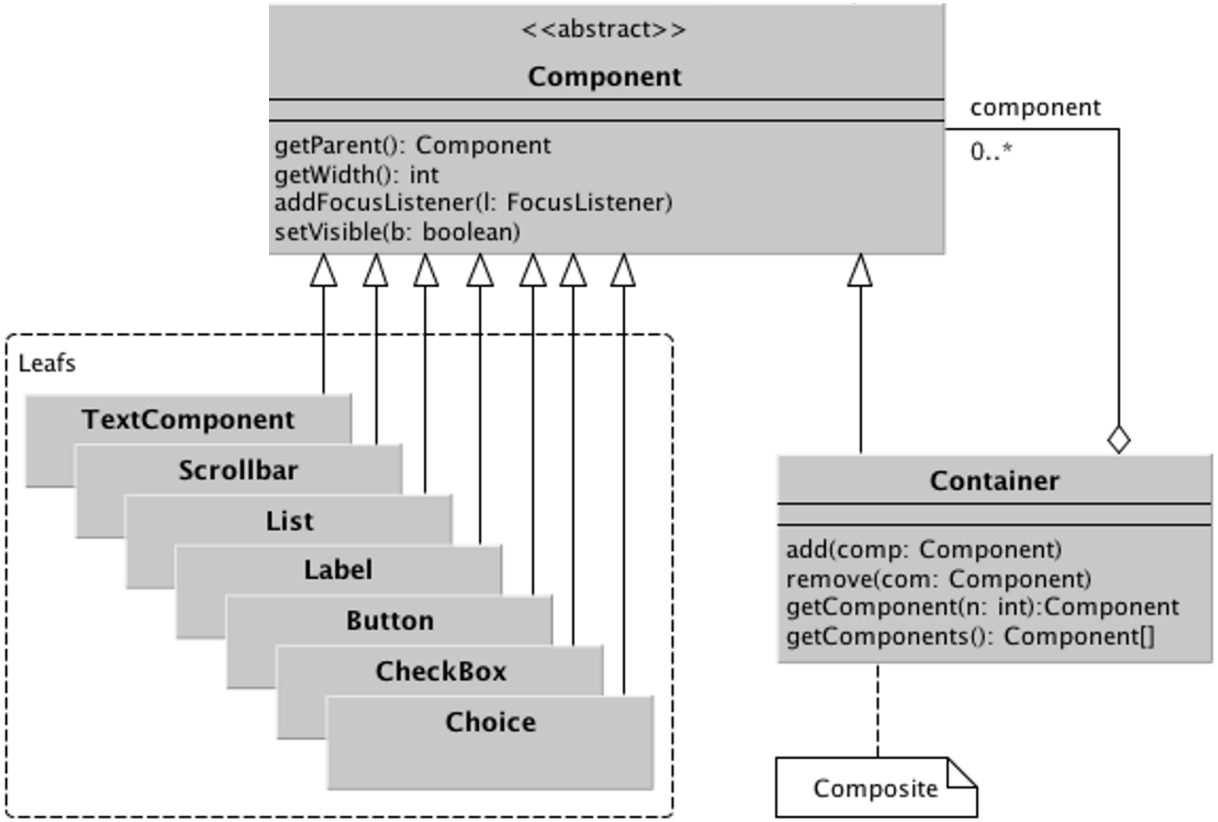
Kompositum-Muster (objektbasiertes Strukturmuster)

Beispiel 1: Gruppierung von Grafikelementen



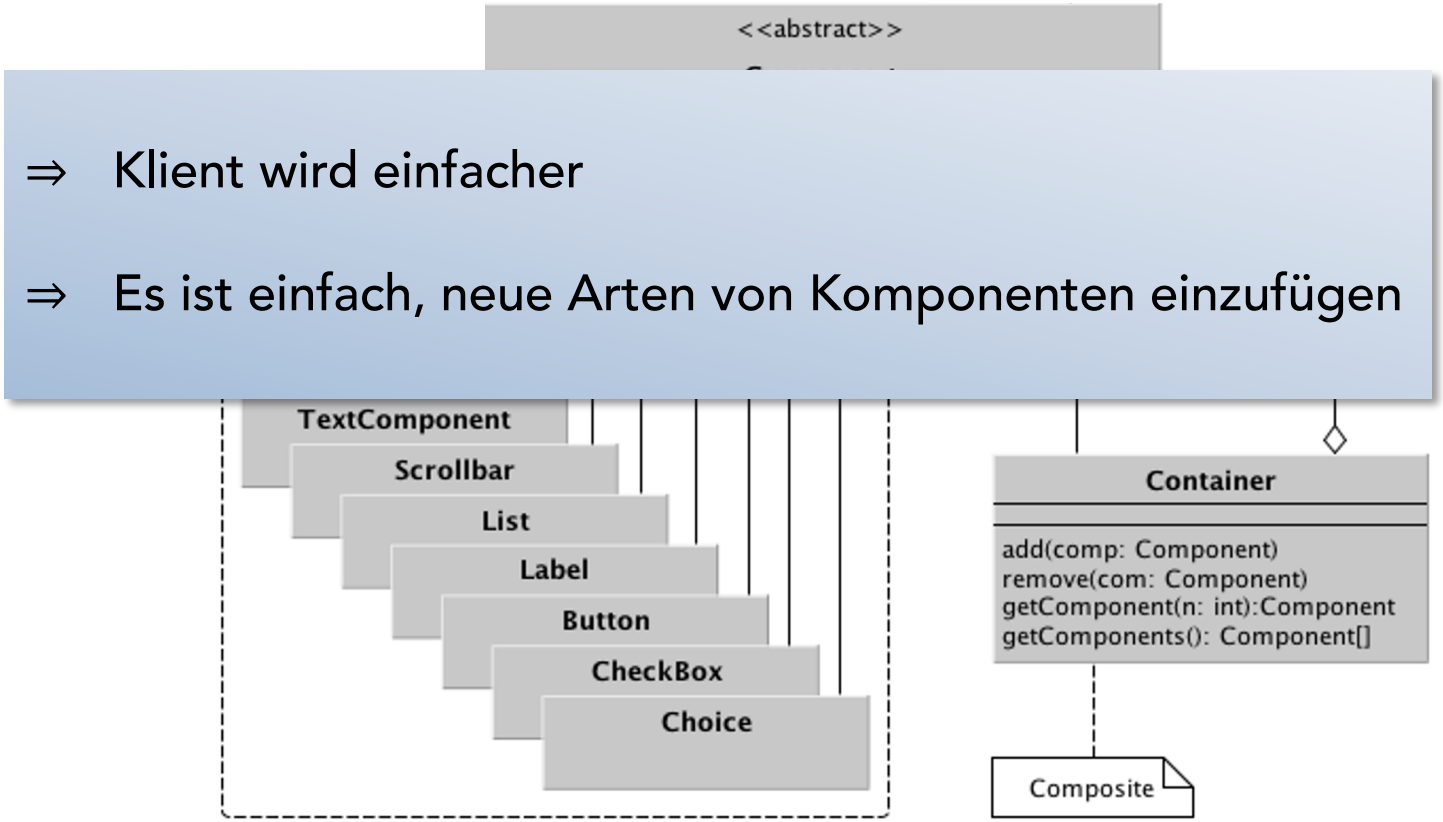
Kompositum-Muster (objektbasiertes Strukturmuster)

Beispiel 2: GUI-Komponenten



Kompositum-Muster (objektbasiertes Strukturmuster)

Beispiel 2: GUI-Komponenten



⇒ Klient wird einfacher

⇒ Es ist einfach, neue Arten von Komponenten einzufügen

Struktur

7. Entwurfsmuster

7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2. Beispiele für Muster

- 7.2.1. Fabrikmethode-Muster (klassenbasiertes Erzeugungsmuster)
- 7.2.2. Singleton-Muster (objektbasiertes Erzeugungsmuster)
- 7.2.3. Kompositum-Muster (objektbasiertes Strukturmuster)
- 7.2.4. Proxy-Muster (objektbasiertes Strukturmuster)
- 7.2.5. Fassaden-Muster (objektbasiertes Strukturmuster)
- 7.2.6. Beobachter-Muster (objektbasiertes Verhaltensmuster)
- 7.2.7. Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)
- 7.2.8. Zustands-Muster (objektbasiertes Verhaltensmuster)
- 7.2.9. ….

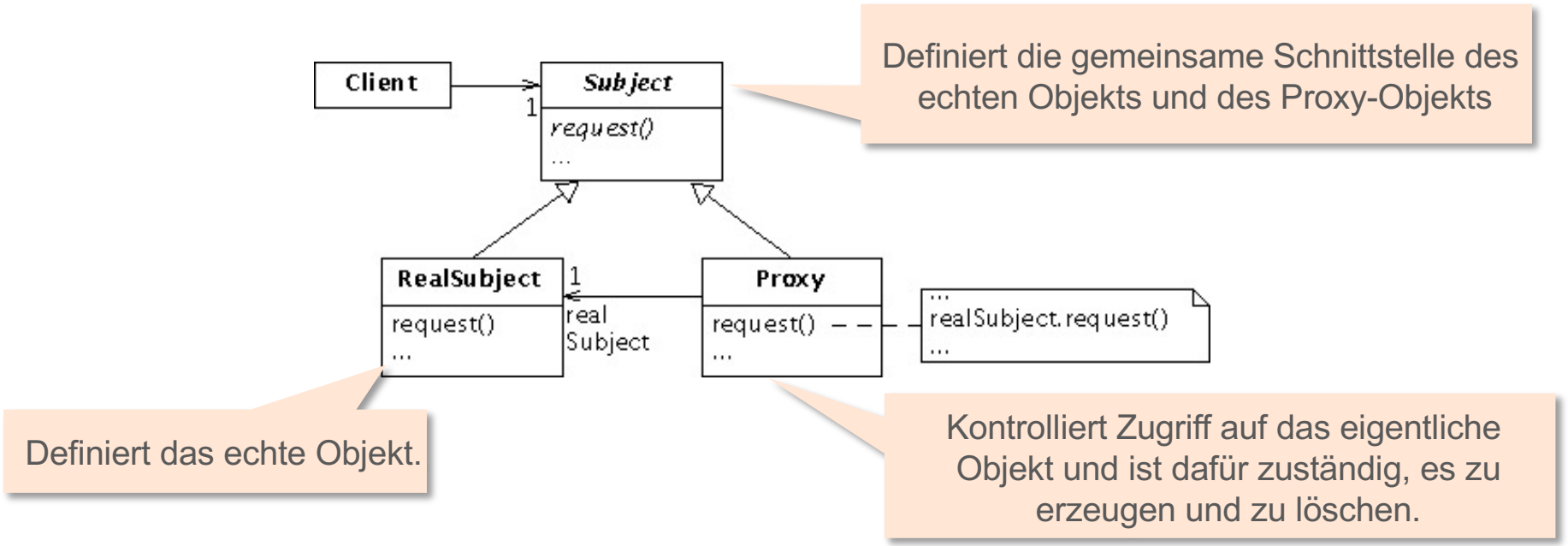
Proxy-Muster (objektbasiertes Strukturmuster)

Anwendungsbeispiele

- **Remote-Proxy** als lokaler Vertreter für ein Objekt auf einem anderen Computer
- **Virtuelles Proxy** erzeugt »teure« Objekte auf Verlangen → z.B. **Cache**
- **Schutz-Proxy** kontrolliert Zugriff auf das Original
- **Smart Reference** als Ersatz für einen einfachen Zeiger, der zusätzlich folgende Funktionen anbietet:
  - Zählen der Referenzen auf das eigentliche Objekt
  - Automatische Freigabe, wenn es keine Referenzen mehr besitzt
  - Laden eines persistenten Objekts, wenn es erstmalig referenziert wird
  - Testen eines Objekts auf locking, bevor darauf zugegriffen wird

Proxy-Muster (objektbasiertes Strukturmuster)

Struktur

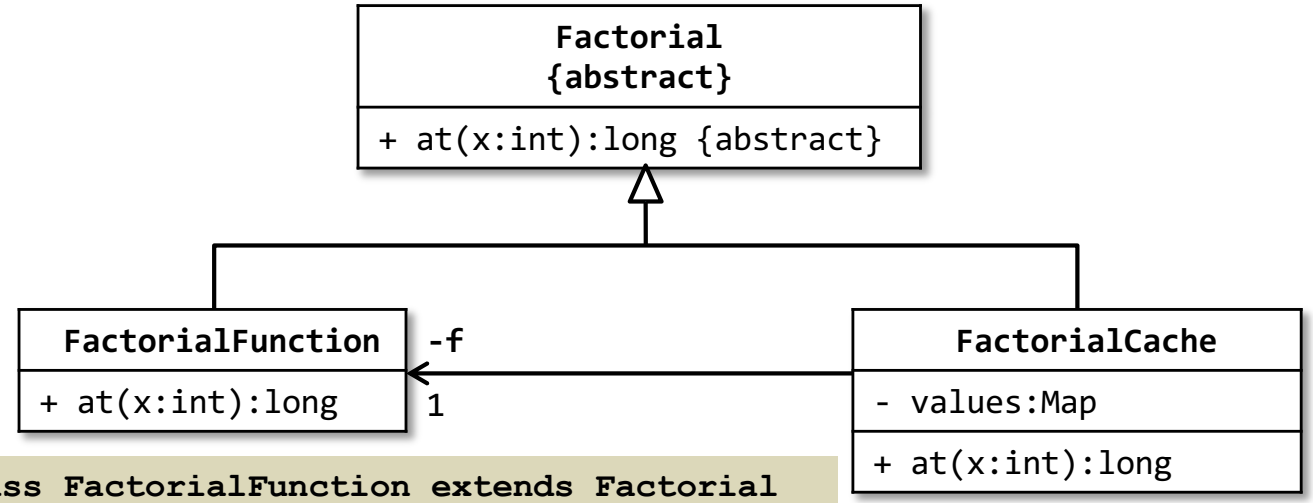


Proxy-Muster (objektbasiertes Strukturmuster)

Beispiel: Cache für Funktionswerte (Fakultät)

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{falls } n > 0 \end{cases}$$

$0! = 1$
$1! = 1$
$2! = 1 \cdot 2 = 2$
$3! = 1 \cdot 2 \cdot 3 = 6$
$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$
$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$

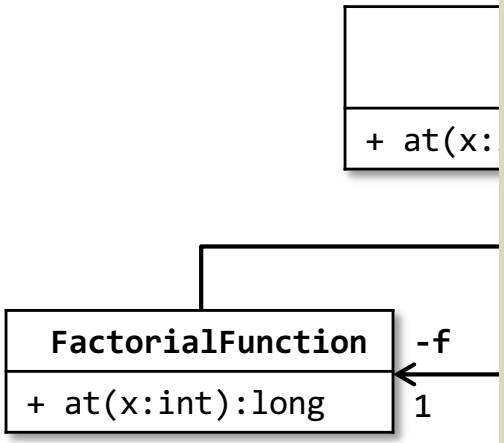



```
public class FactorialFunction extends Factorial
{
    @Override
    public long at(int x) {
        long res = 1;
        while(x > 0) {
            res *= x;
            x--;
        }
        return res;
    }
}
```

Proxy-Muster (objektbasiertes Strukturmuster)

Beispiel: Cache für Funktionswerte (Fakultät)

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ \end{cases}$$



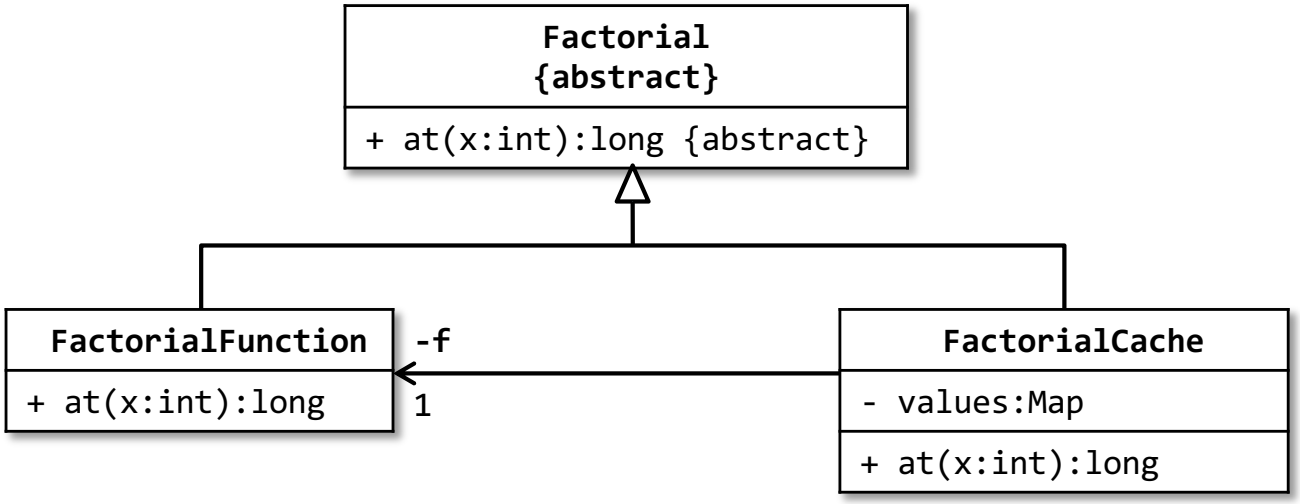

```
import java.util.HashMap;
import java.util.Map;

public class FactorialCache extends Factorial {
    private FactorialFunction f;
    private Map<Integer, Long> values;

    public FactorialCache() {
        f = new FactorialFunction();
        values = new HashMap<Integer, Long>();
    }

    @Override
    public long at(int x) {
        long y;
        if(values.containsKey(x)) {
            y = values.get(x);  // use cached result
        } else {
            y = f.at(x);        // compute result
            values.put(x, y);   // insert result into cache
        }
        return y;
    }
}
```

Proxy-Muster (objektbasiertes Strukturmuster)

Beispiel: Cache für Funktionswerte (Fakultät)

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{falls } n > 0 \end{cases}$$

$0! = 1$
$1! = 1$
$2! = 1 \cdot 2 = 2$
$3! = 1 \cdot 2 \cdot 3 = 6$
$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$
$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$



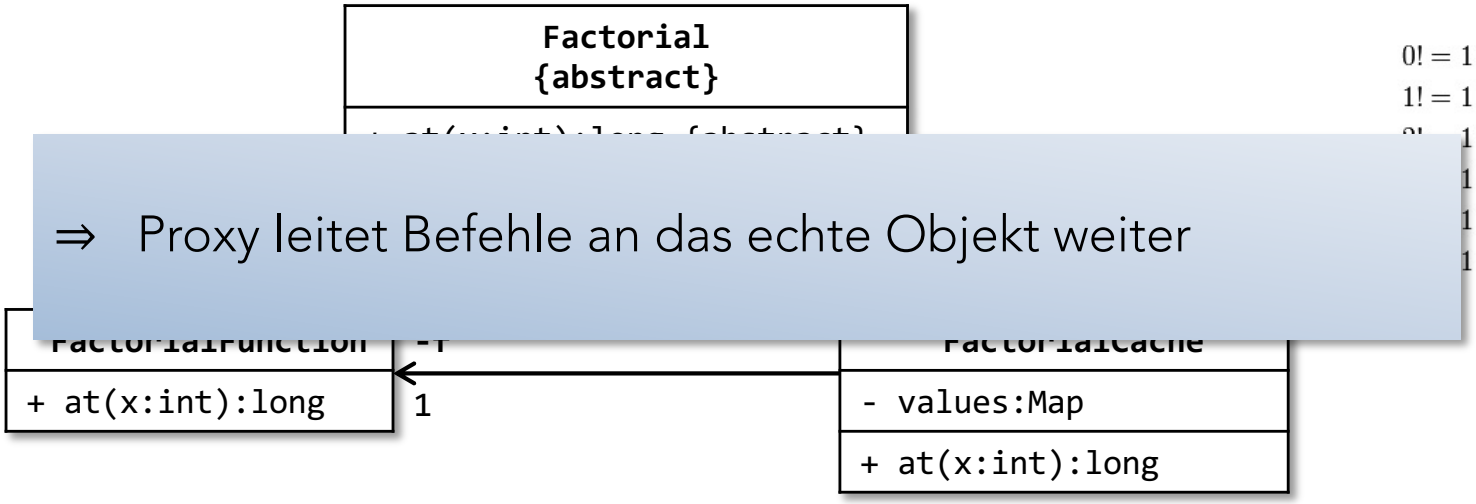

```
public class FactorialTest {
    public static void main(String[] args) {
        // cached version of factorial
        Factorial factorial = new FactorialCache();
        factorial.at(10);
        factorial.at(10);
        factorial.at(20);
    }
}
```

Proxy-Muster (objektbasiertes Strukturmuster)

Beispiel: Cache für Funktionswerte (Fakultät)

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{falls } n > 0 \end{cases}$$

$0! = 1$
$1! = 1$
$1 \cdot 2 = 2$
$1 \cdot 2 \cdot 3 = 6$
$1 \cdot 2 \cdot 3 \cdot 4 = 24$
$1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$



⇒ Proxy leitet Befehle an das echte Objekt weiter

```
public class FactorialTest {
    public static void main(String[] args) {
        // cached version of factorial
        Factorial factorial = new FactorialCache();
        factorial.at(10);
        factorial.at(10);
        factorial.at(20);
    }
}
```

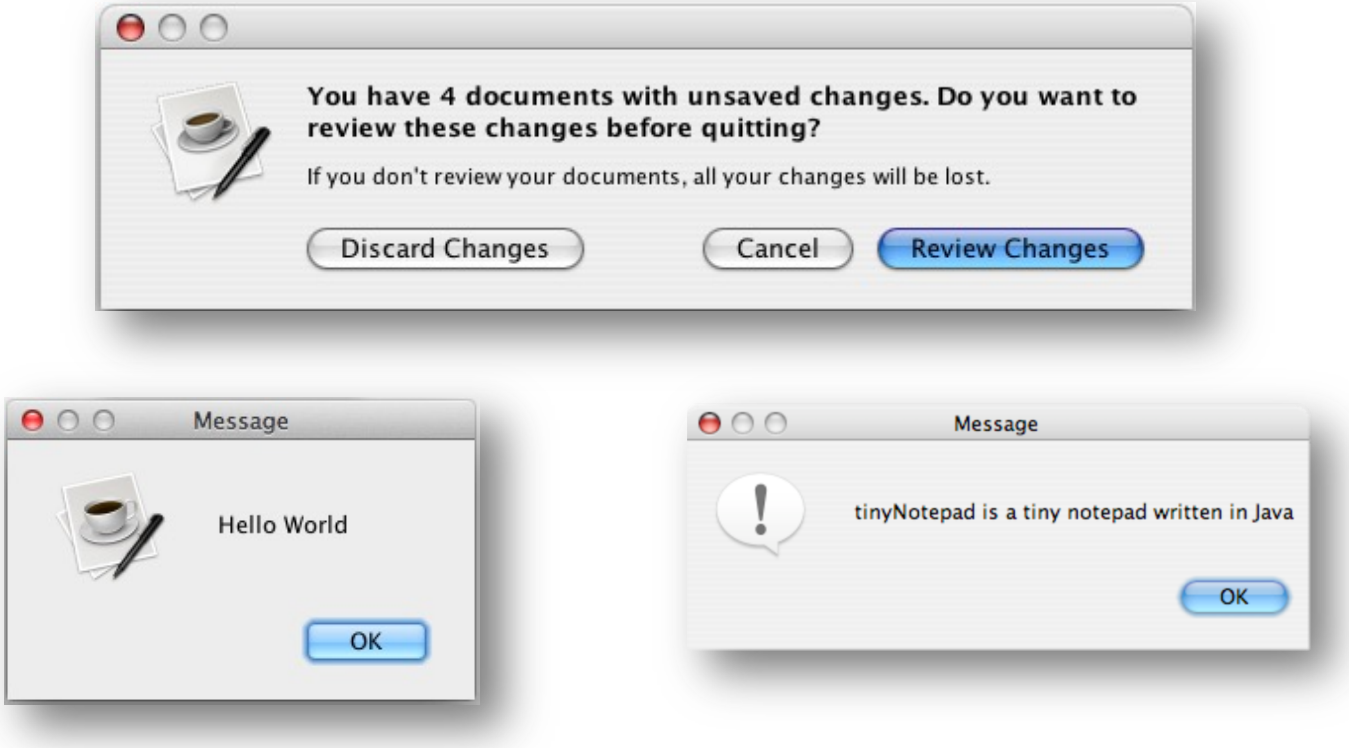
Struktur

7. Entwurfsmuster

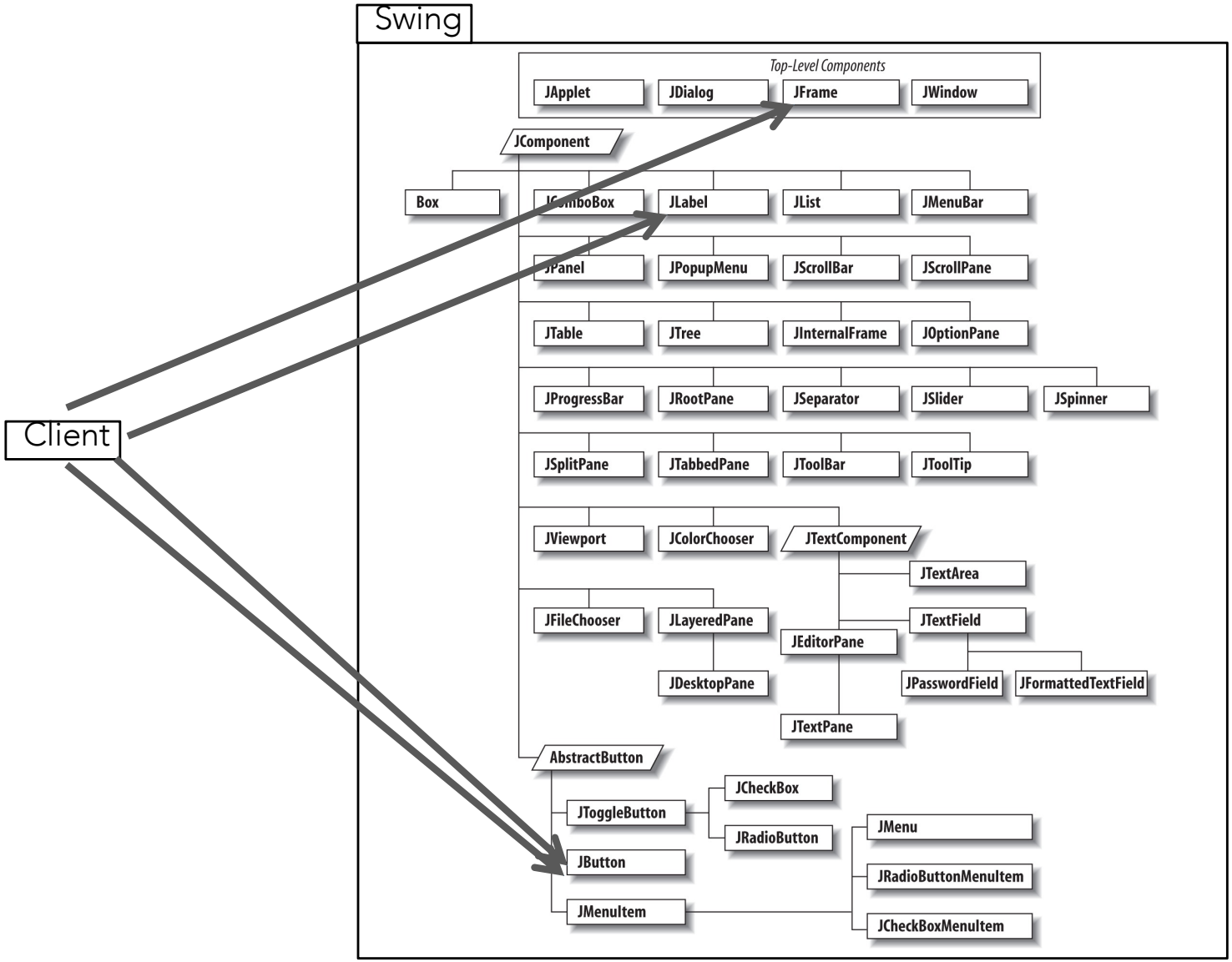
7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2. Beispiele für Muster

Fassaden-Muster (objektbasiertes Strukturmuster)



Fassaden-Muster (objektbasiertes Strukturmuster)



Fassaden-Muster (objektbasiertes Strukturmuster)

Beispiel: Messages in graphischen Nutzungsoberflächen

Probleme

- Jeder Nutzer geht anders vor
- Sind die Abhängigkeiten nicht bekannt => Fehler
- Treten Änderungen an der Umsetzung des Packages auf, müssen die Nutzer sich erst einarbeiten
- Jedes Message-Fenster sieht ggf. anders aus

Probleme (allgemein)

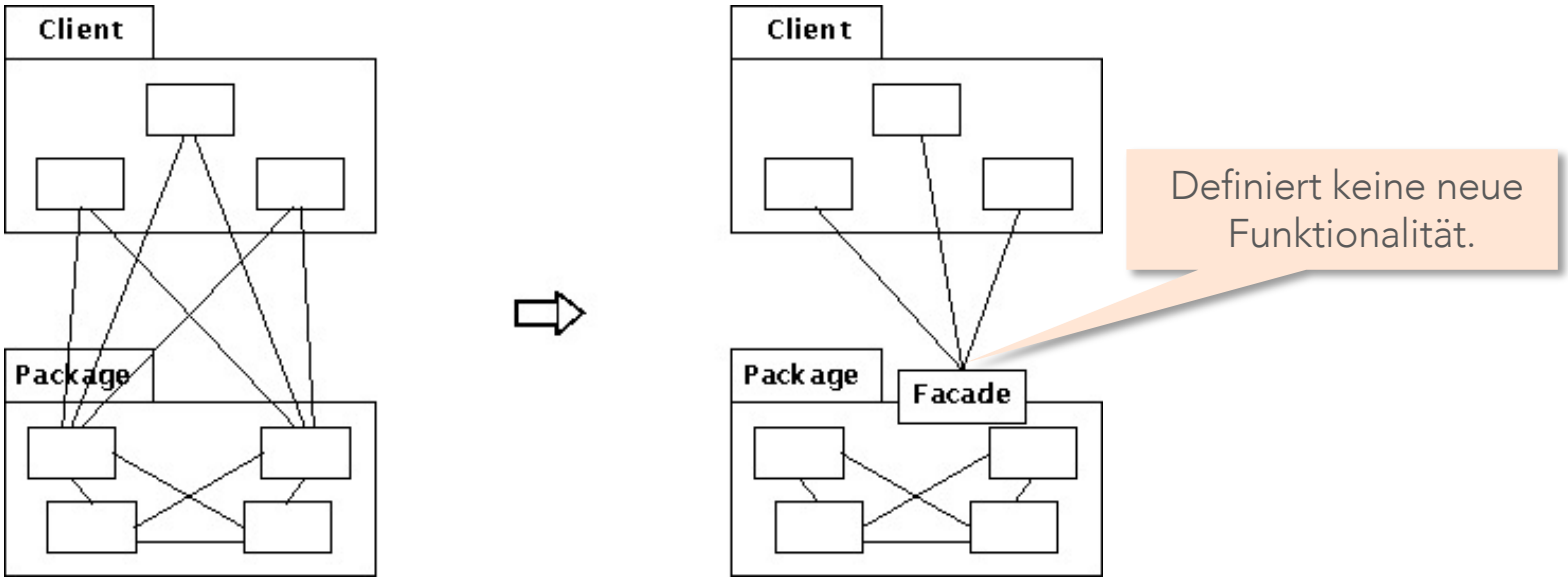
- **Systemwissen notwendig.** Jeder Client muss nicht nur jede benötigte Klasse des Systems kennen, sondern auch ihr Zusammenspiel und ihre Funktionsweise, um sie nutzen zu können.
- **Abhängigkeiten** und geringe **Änderungsstabilität**. Da jeder Client viele verschiedene Klassen kennen muss, steigen seine Abhängigkeiten. Er ist hart an das System gekoppelt. Änderungen am System führen zwangläufig dazu, dass der Clientcode bricht oder nicht mehr korrekt funktioniert - und das gilt für ***jeden*** Client, der das System nutzt. Die Folge ist hoher **Wartungsaufwand.**
- **Coderedundanz** und **Gefahr von Inkonsistenz**. Alle Clients, die eine Message ausgeben wollen, müssen immer den gleichen Code schreiben.

Fassaden-Muster (objektbasiertes Strukturmuster)

Anwendbarkeit

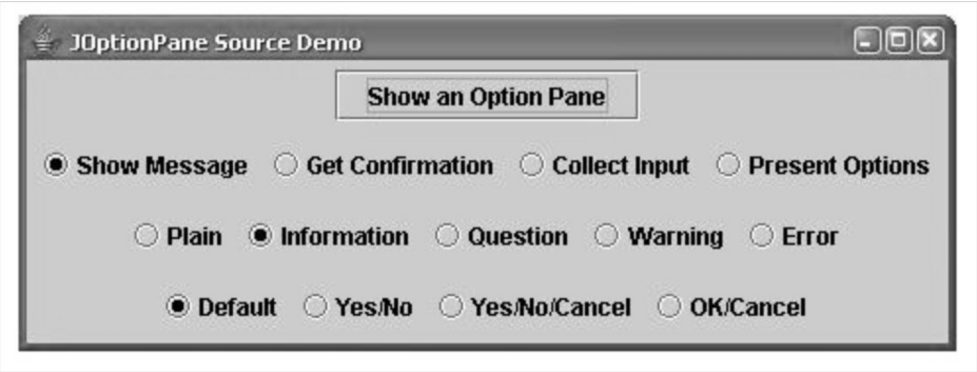
- Verwendung des Muster, wenn
  - einfache Schnittstellen zu einem komplexen Paket angeboten werden sollen
  - es zahlreiche Abhängigkeiten zwischen Klienten und einem Paket gibt
  - Pakete in Schichten organisiert werden sollen

Struktur



Fassaden-Muster (objektbasiertes Strukturmuster)

Beispiel

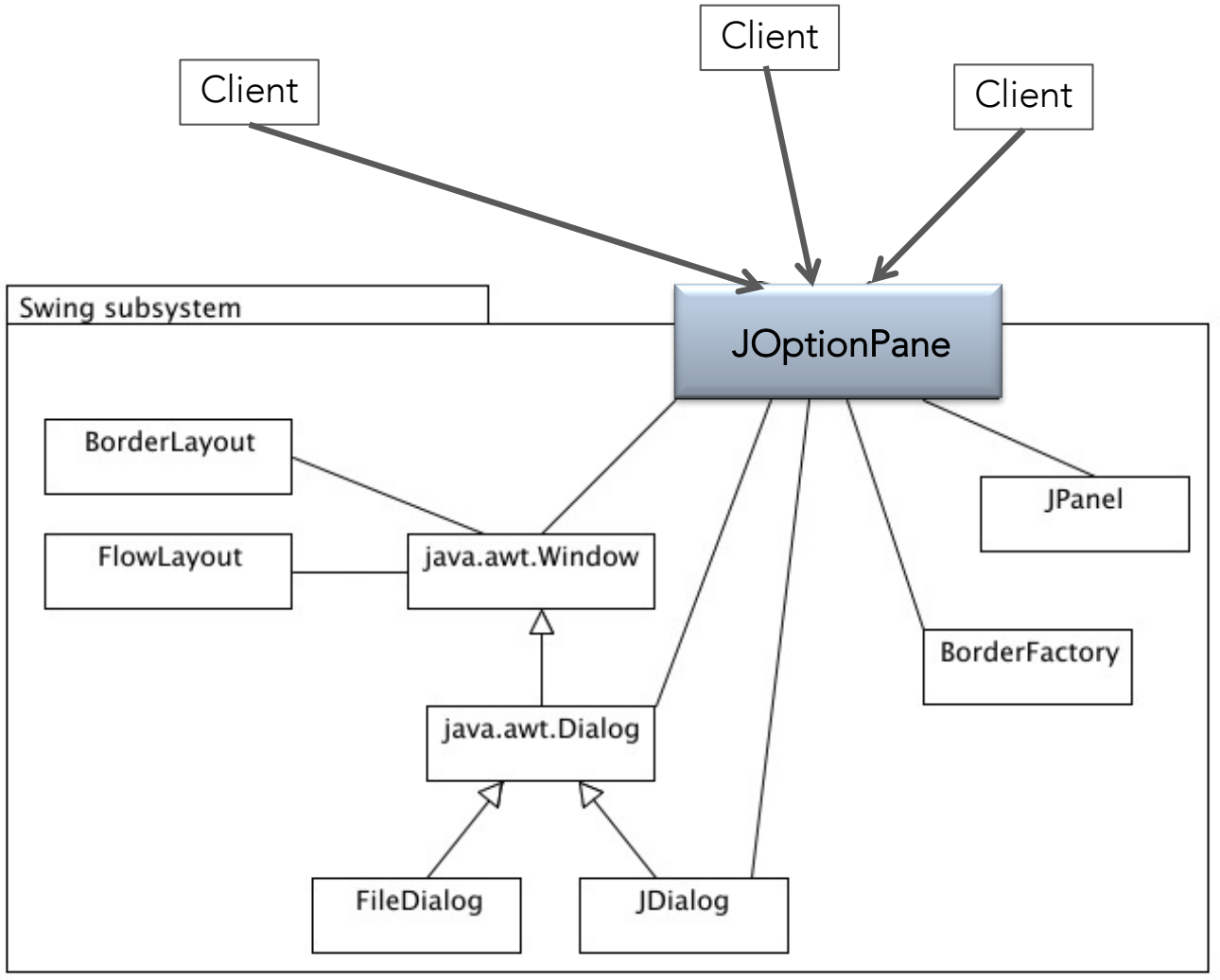


Icon	Code	IDE Value
No icon	JOptionPane.PLAIN_MESSAGE	-1
	JOptionPane.ERROR_MESSAGE	0
	JOptionPane.INFORMATION_MESSAGE	1
	JOptionPane.WARNING_MESSAGE	2
	JOptionPane.QUESTION_MESSAGE	3

Method Name
showConfirmDialog
showInputDialog
showMessageDialog
showOptionDialog

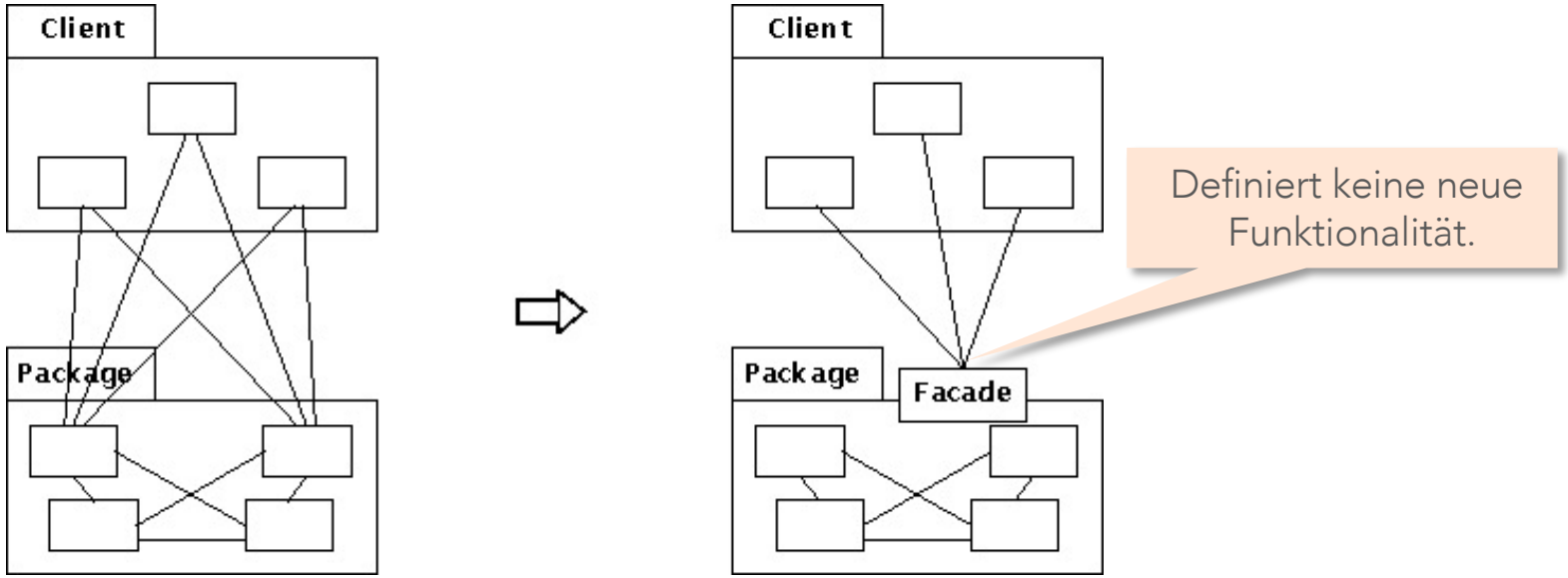
Fassaden-Muster (objektbasiertes Strukturmuster)

Beispiel



Fassaden-Muster (objektbasiertes Strukturmuster)

Struktur



Fassaden-Muster (objektbasiertes Strukturmuster)

Struktur

⇒ Vereinfachung der Benutzung des Systems durch Reduzierung der Klassen, die den Klienten bekannt sein müssen

⇒ Lose Kopplung erleichtert Austausch von Paketen und deren unabhängige Implementierung

⇒ Klienten können die Fassade umgehen und direkt auf die Klassen des Pakets zugreifen

Definiert keine neue Funktionalität.

Struktur

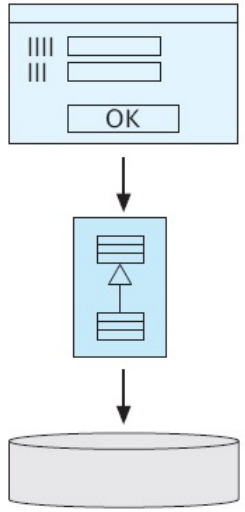
7. Entwurfsmuster

7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

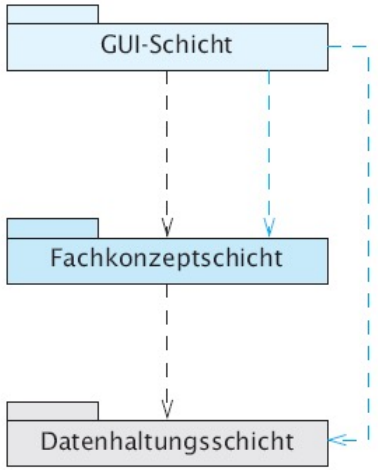
7.2. Beispiele für Muster

Beobachter-Muster (objektbasiertes Verhaltensmuster)

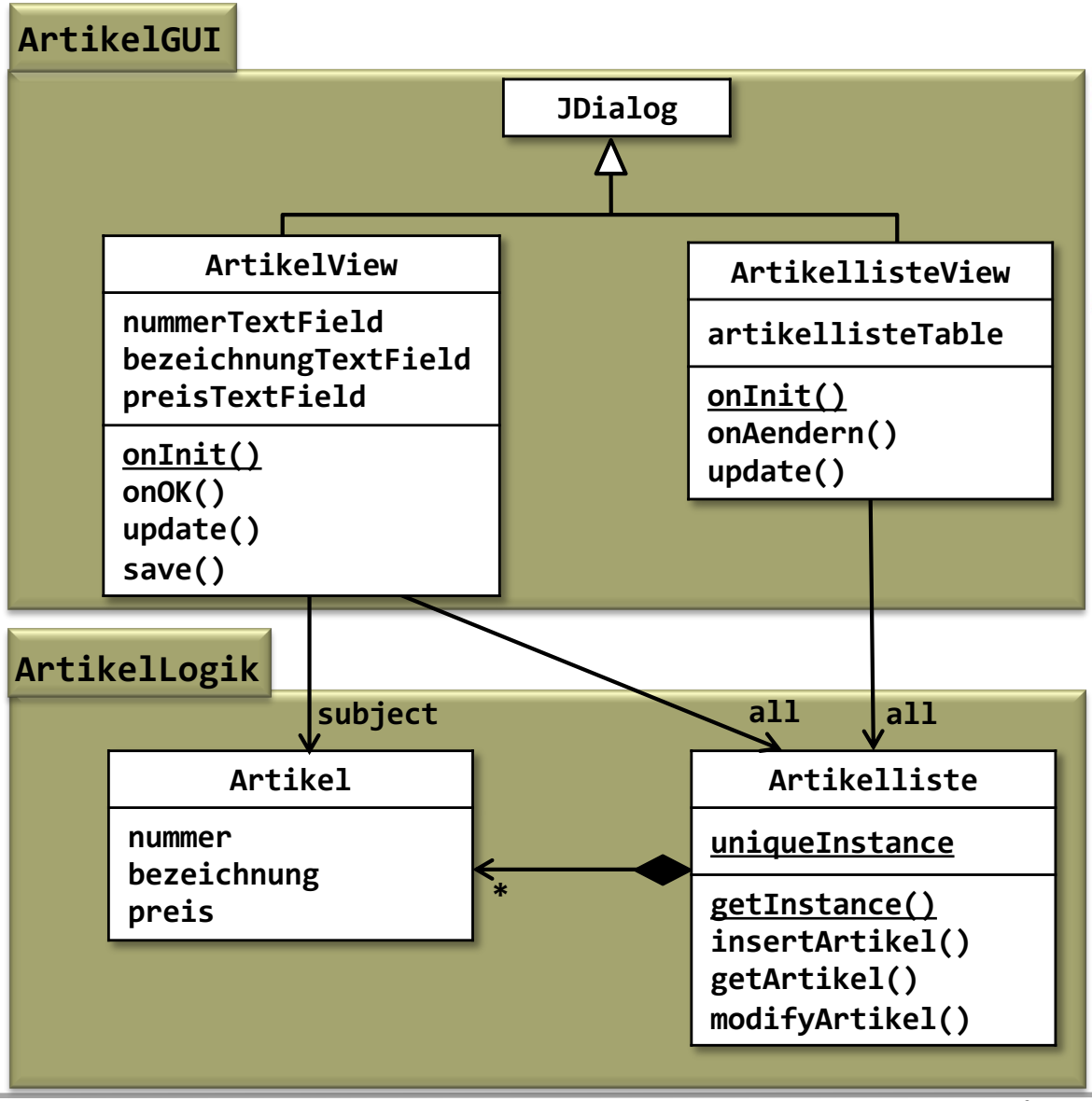
- Strenge Drei-Schichten-Architektur
  - GUI-Schicht kann nur auf Fachkonzeptschicht zugreifen
  - Fachkonzeptschicht kann nur auf Datenhaltungsschicht zugreifen
  - Vorteil:
    GUI-Schicht unabhängig von gewählter Speicherung der Daten

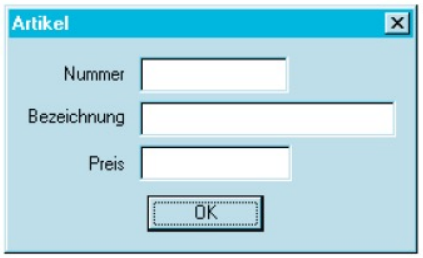


- Flexible Drei-Schichten-Architektur
  - GUI-Schicht kann auf Fachkonzept-schicht und Datenhaltungsschicht zugreifen
  - Vorteile: größere Flexibilität, bessere Performance
  - Nachteile: geringere Wartbarkeit, Änderbarkeit und Portabilität



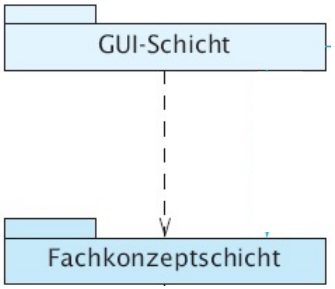
Beobachter-Muster (objektbasiertes Verhaltensmuster)



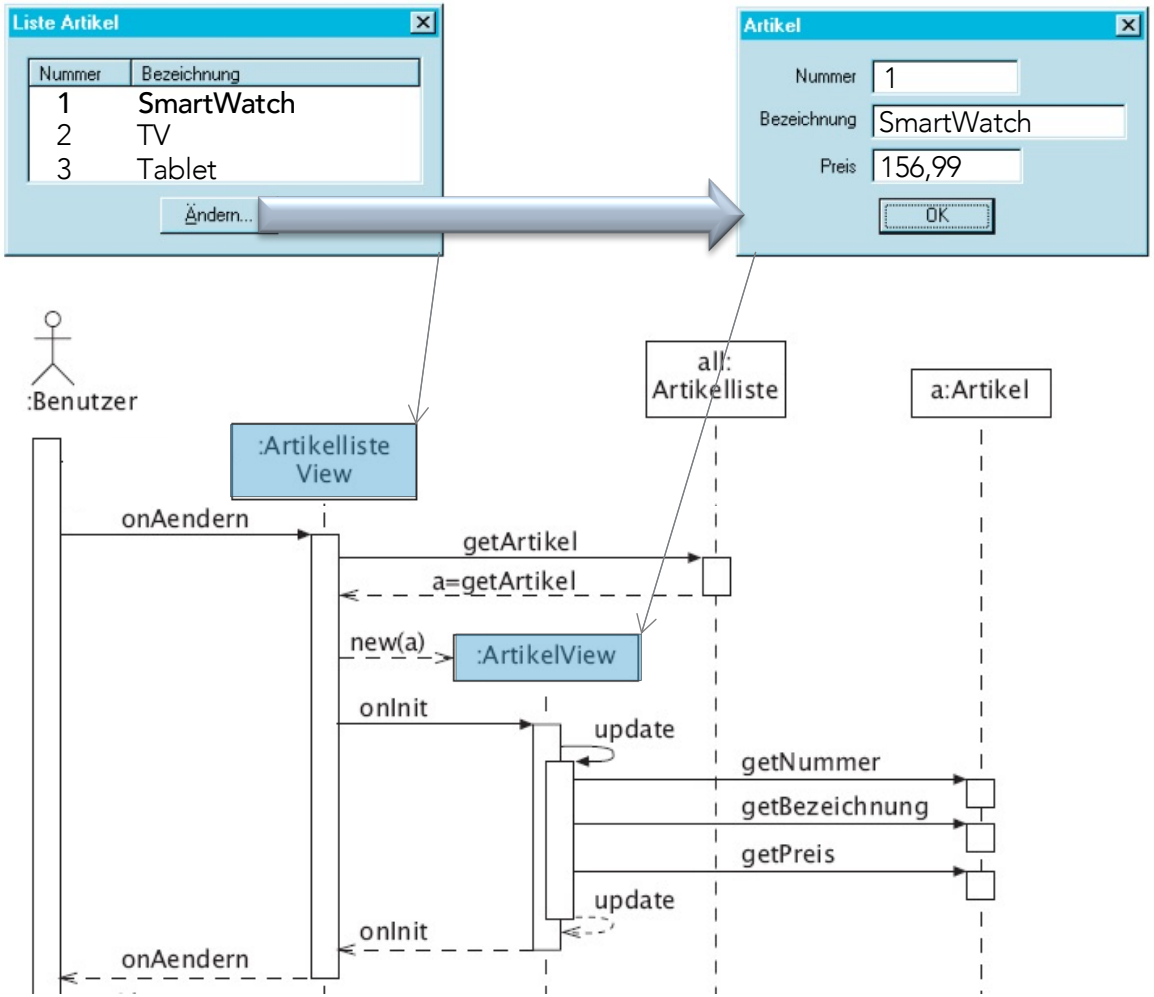


ArtikelView

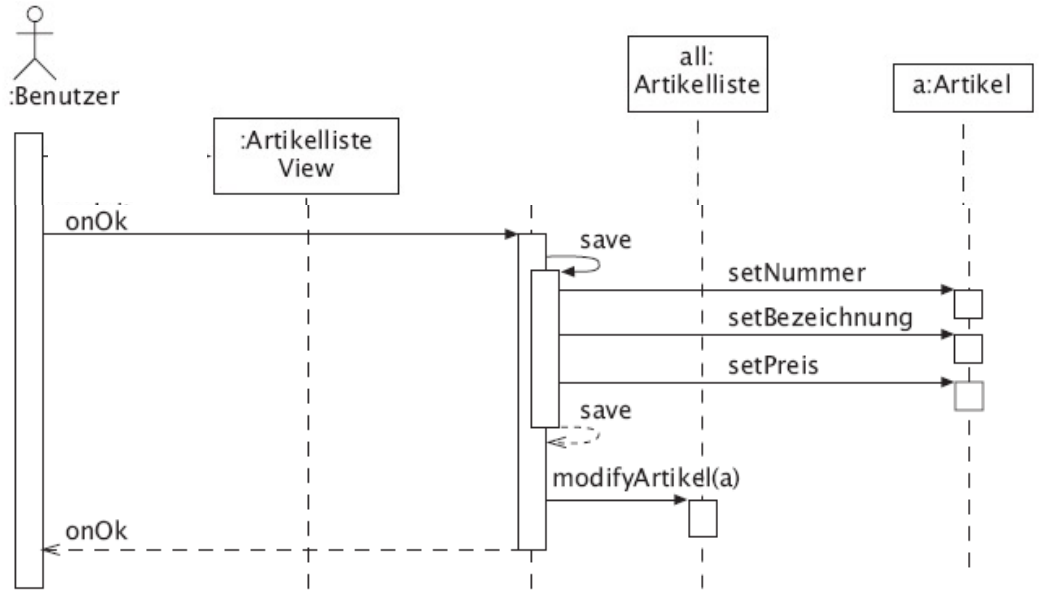
Artikelliste View



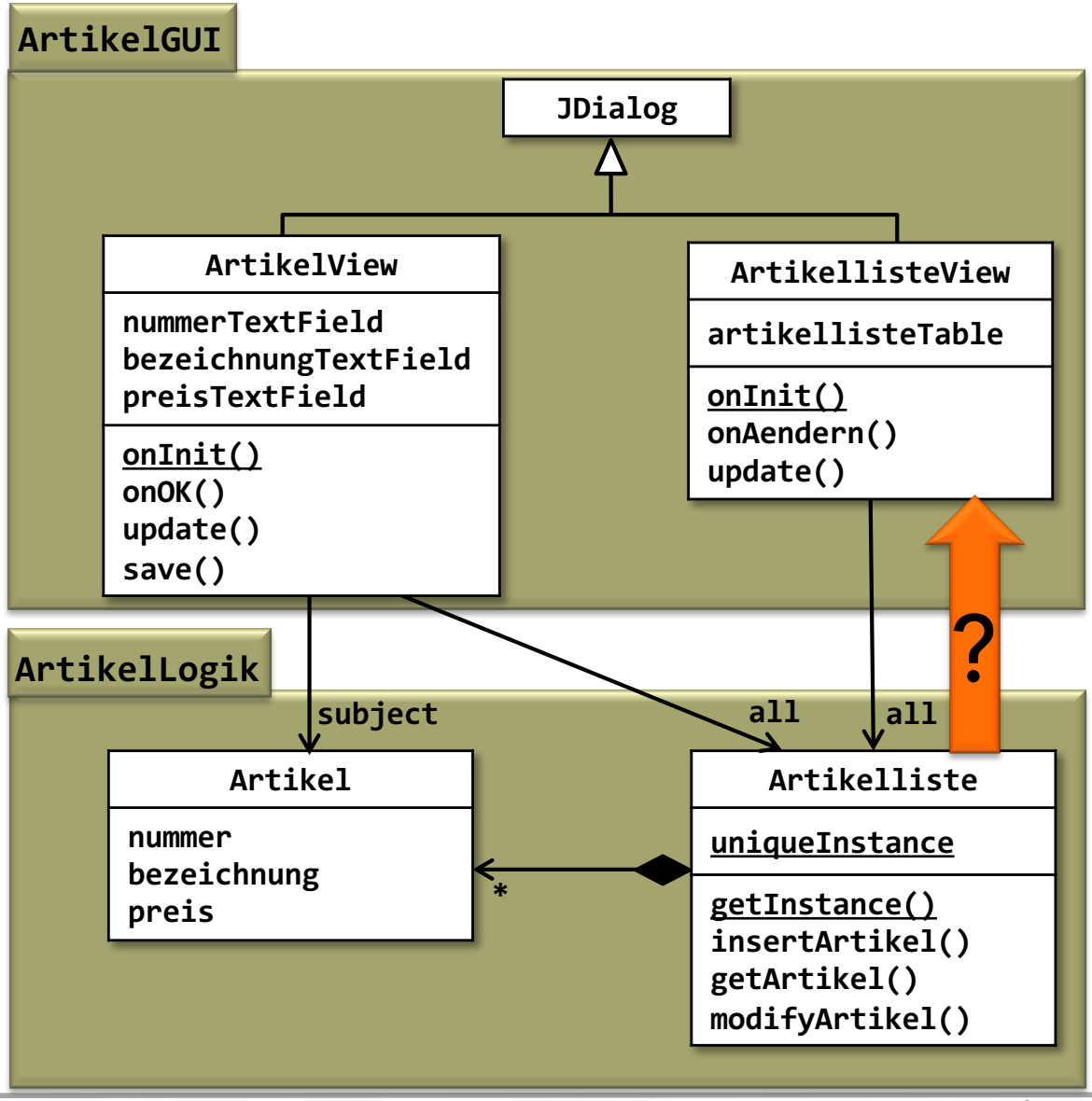
Beobachter-Muster (objektbasiertes Verhaltensmuster)

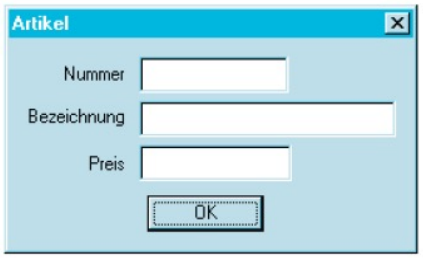


Beobachter-Muster (objektbasiertes Verhaltensmuster)



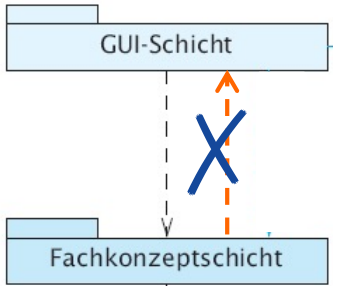
Beobachter-Muster (objektbasiertes Verhaltensmuster)



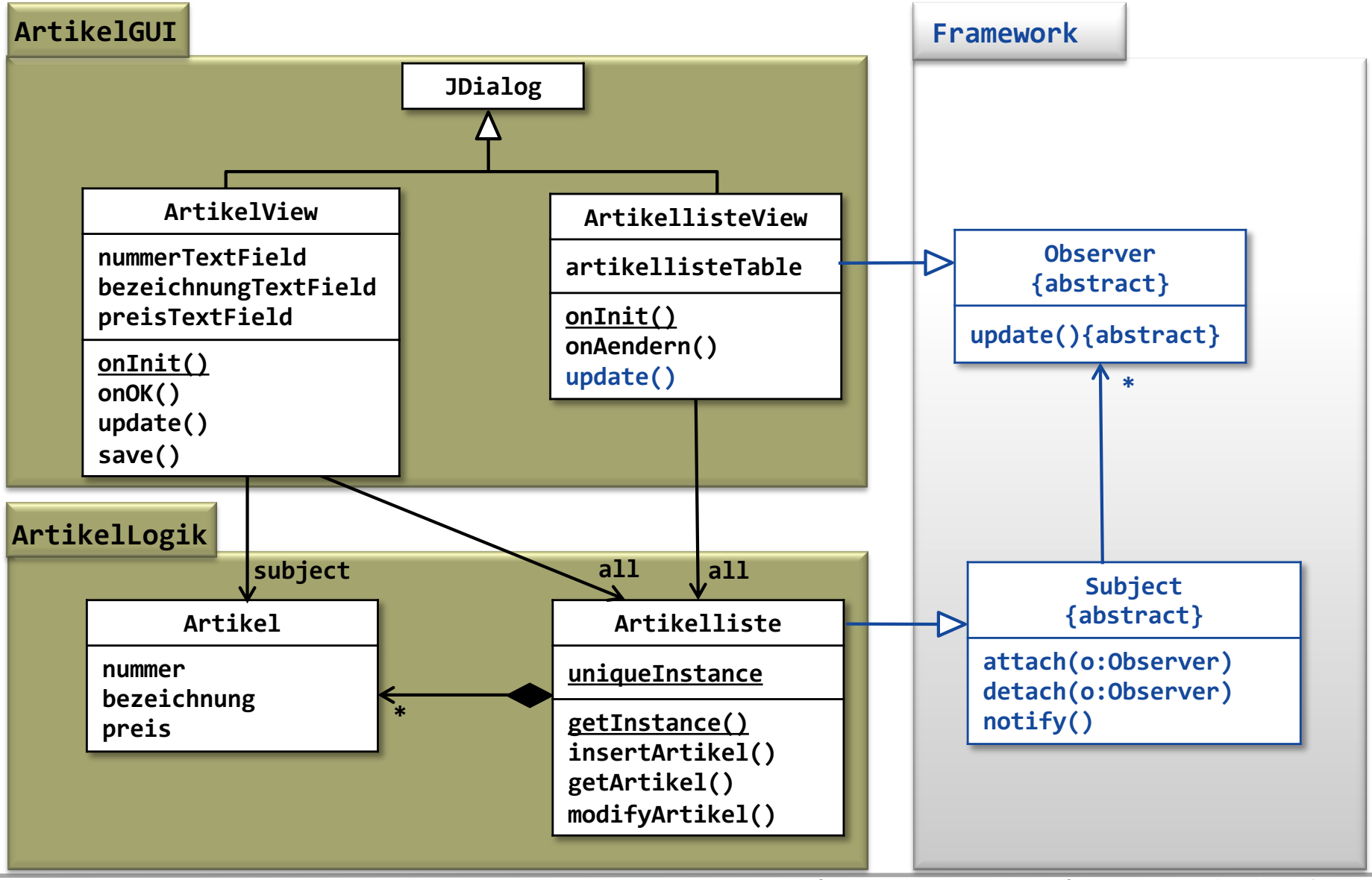


ArtikelView

Artikelliste View



Beobachter-Muster (objektbasiertes Verhaltensmuster)

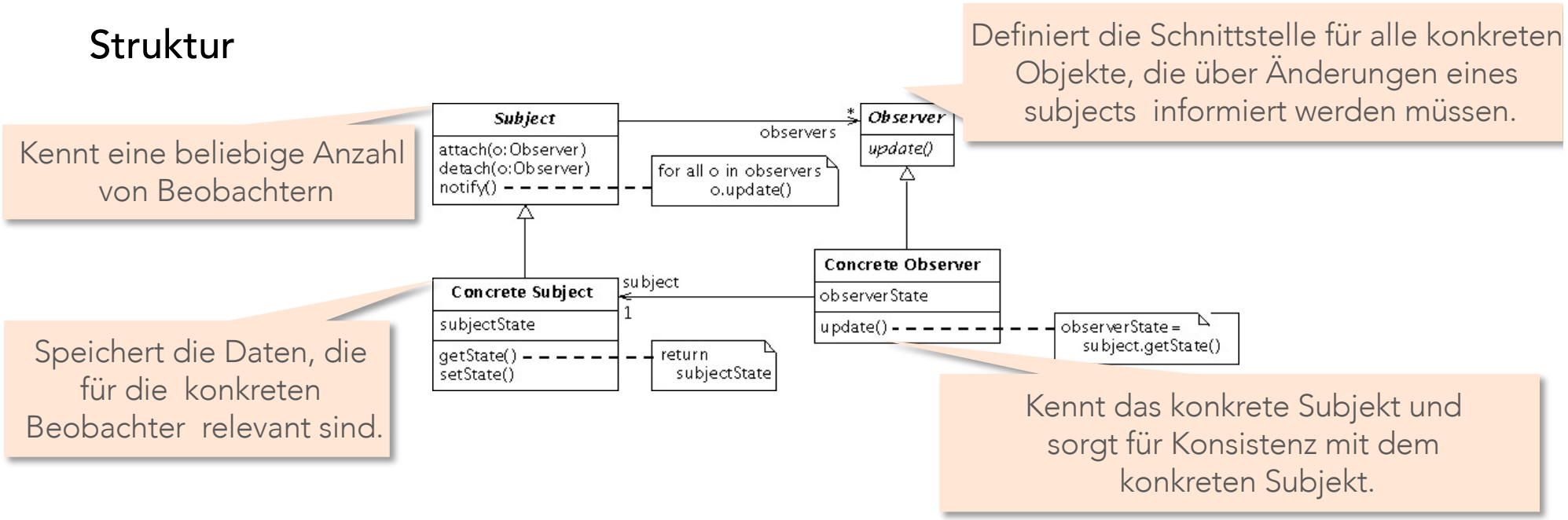


Beobachter-Muster (objektbasiertes Verhaltensmuster)

Anwendbarkeit

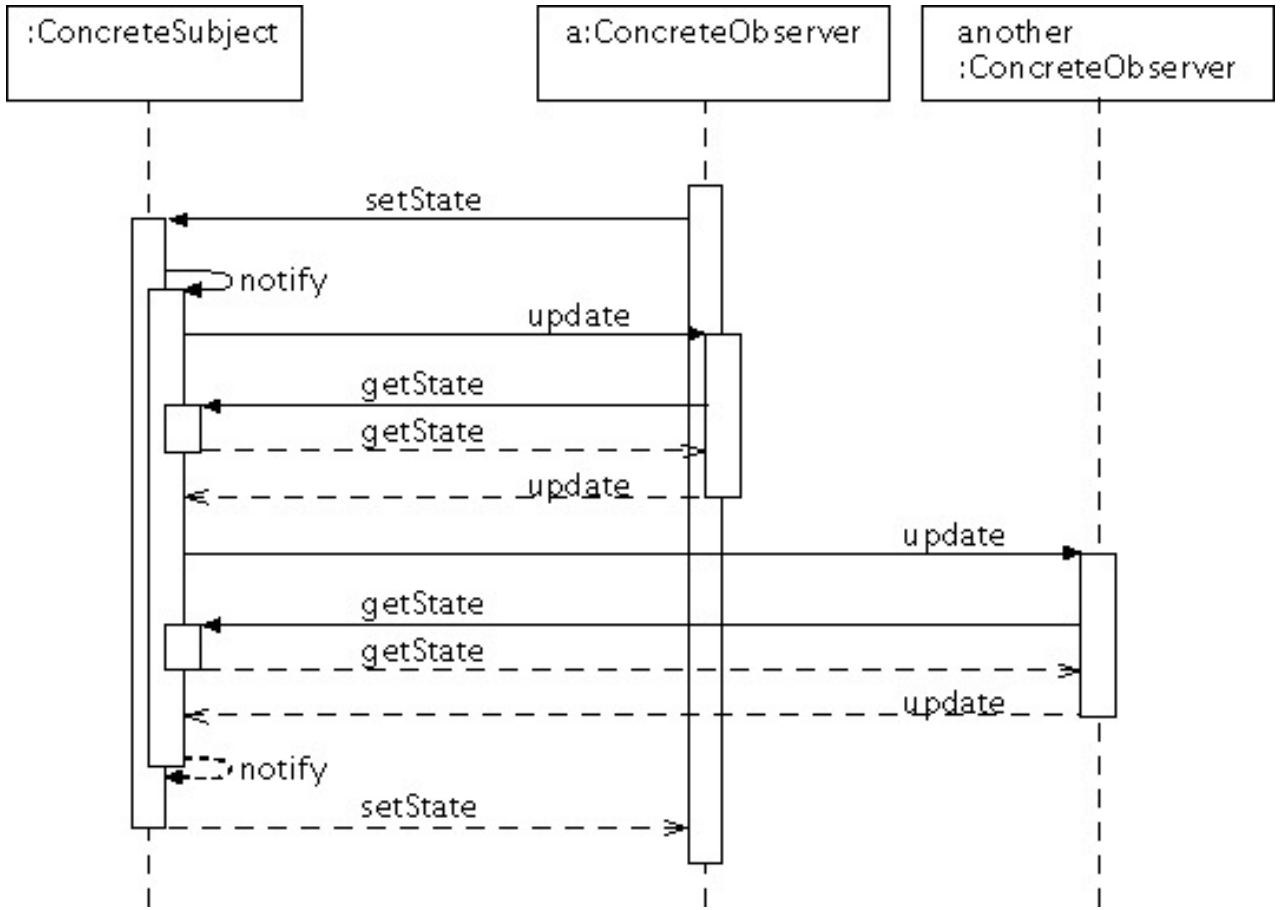
- Verwendung des Muster, wenn
  - eine Abstraktion besitzt zwei Aspekte, die wechselseitig voneinander abhängen
  - die Änderung eines Objekts die Änderung anderer Objekte unbekannter Anzahl impliziert
  - ein Objekt andere Objekte benachrichtigen soll und diese Objekte sind nur lose gekoppelt

Struktur

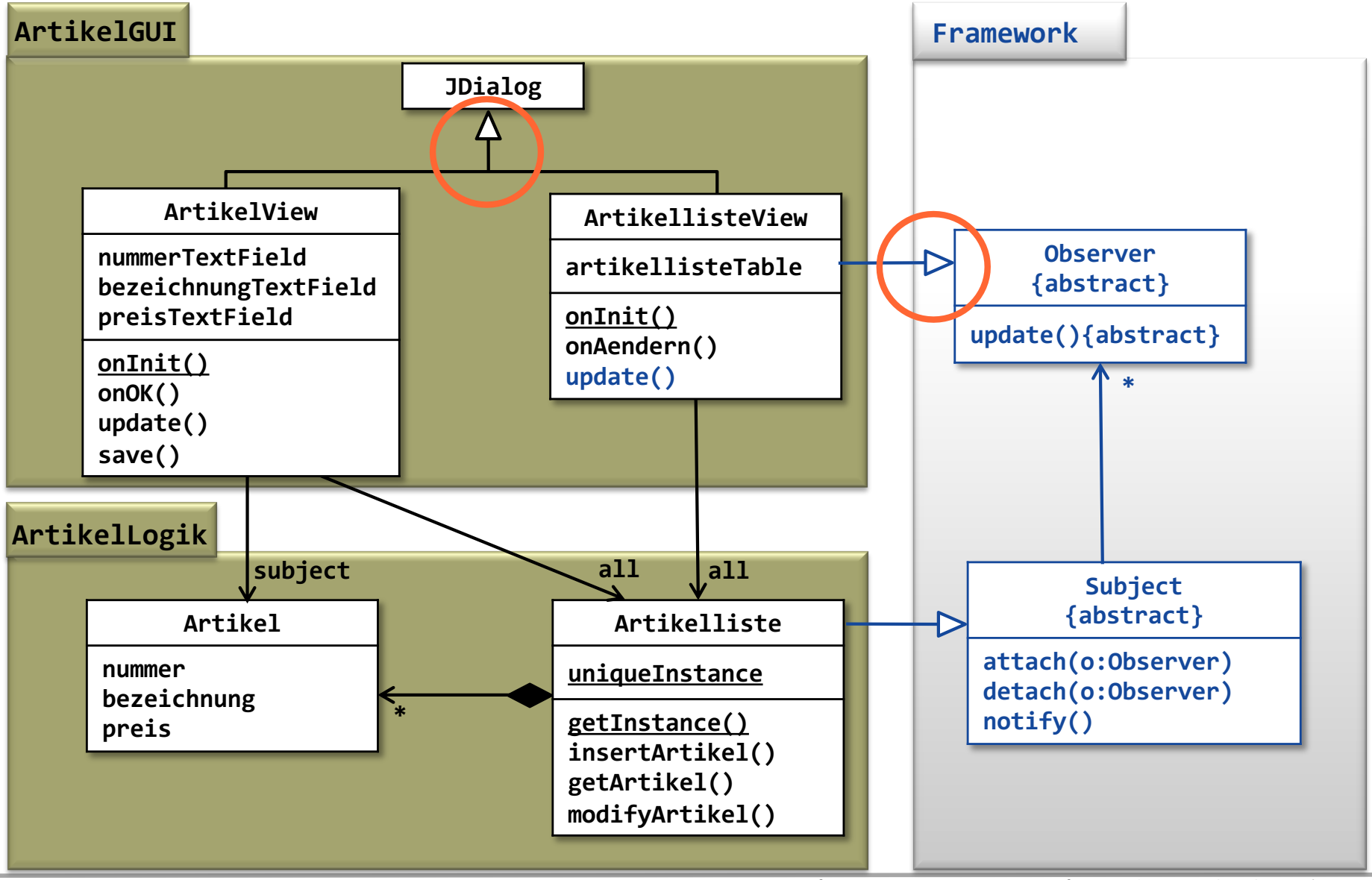


Beobachter-Muster (objektbasiertes Verhaltensmuster)

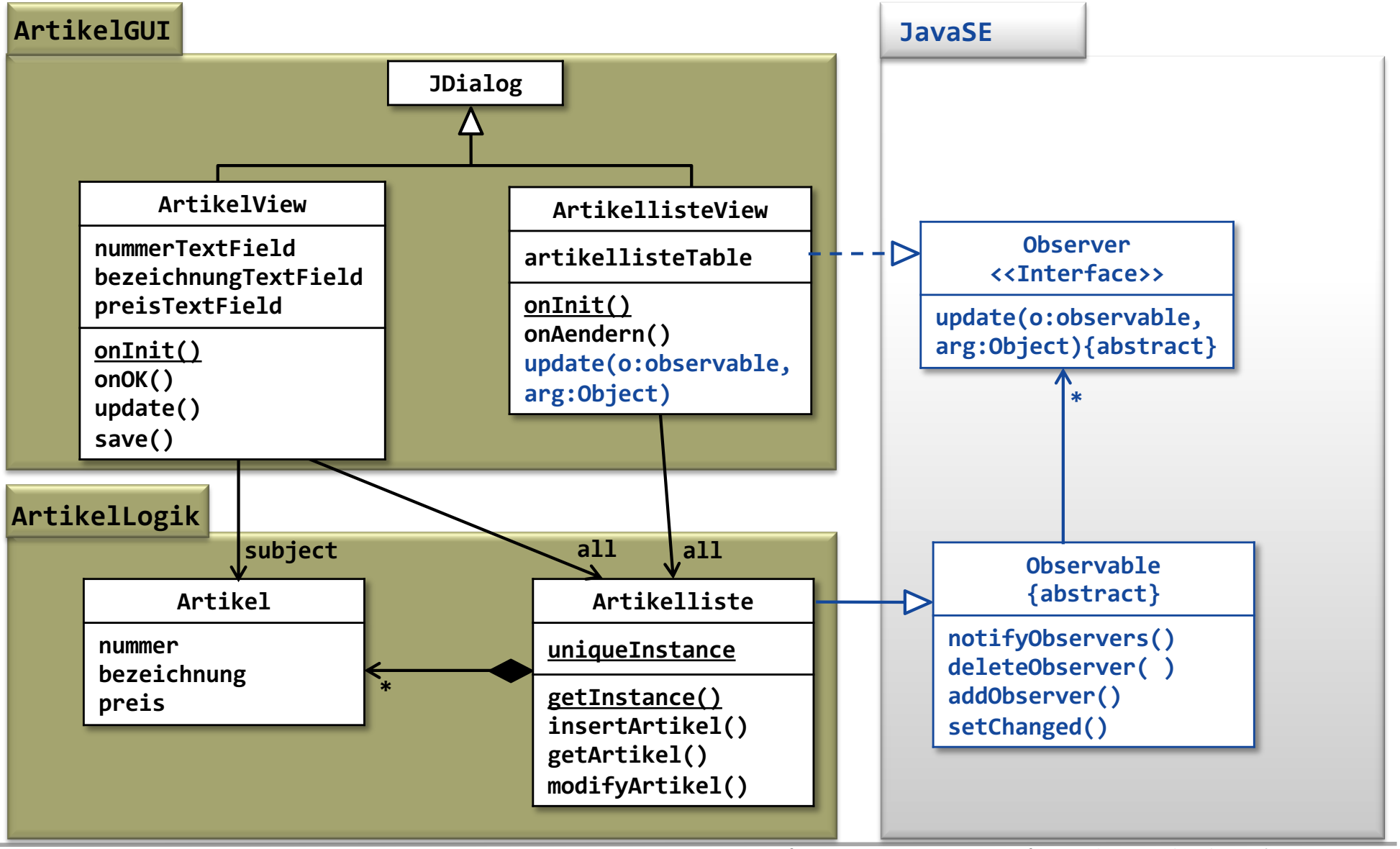
Dynamische Sicht



Beobachter-Muster (objektbasiertes Verhaltensmuster)



Beobachter-Muster (objektbasiertes Verhaltensmuster)



Beobachter-Muster (objektbasiertes Verhaltensmuster)

Anwendbarkeit

- Verwendung des Muster, wenn
  - eine Abstraktion besitzt zwei Aspekte, die voneinander abhängen
  - eine Änderung an einem Objekt erfordert, dass andere Objekte impliziert
  - ein Objekt andere Objekte benachrichtigen muss, ohne Annahmen über entkoppelt

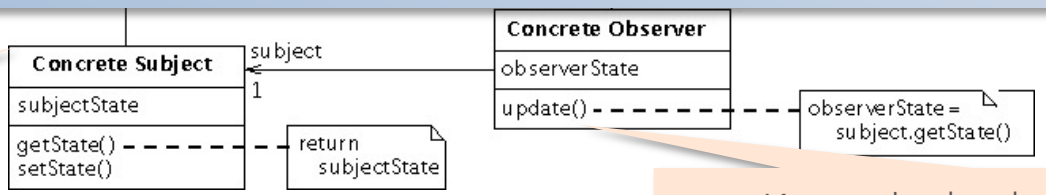
⇒ Subjekte und Beobachter können unabhängig voneinander modifiziert und einzeln wiederverwendet werden

⇒ Neue Beobachter können ohne Änderung des Subjekts hinzugefügt werden

Strukt

für alle konkreten derungen eines erden müssen.

Kennt eine b von Be

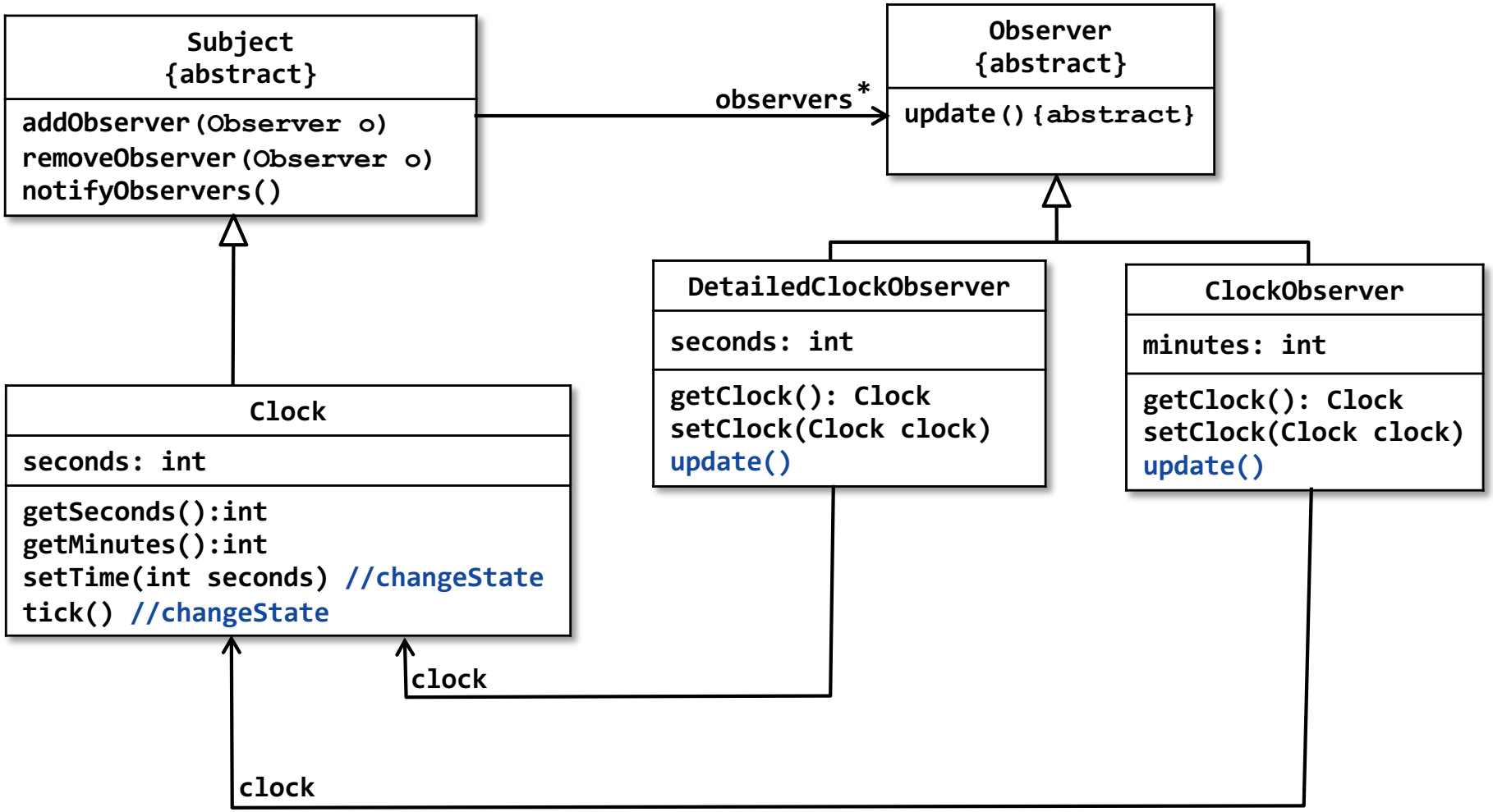


Speichert die Daten, die für die konkreten Beobachter relevant sind.

Kennt das konkrete Subjekt und sorgt für Konsistenz mit dem konkreten Subjekt.

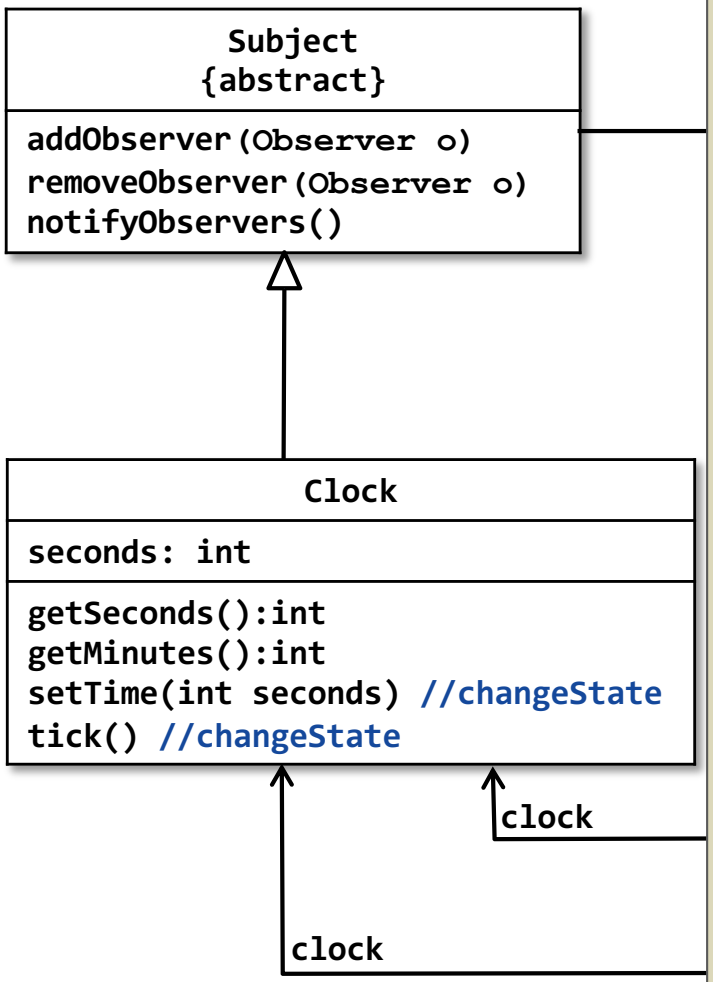
Beobachter-Muster (objektbasiertes Verhaltensmuster)

Uhren-Beispiel



Beobachter-Muster (objektbasiertes Verhaltensmuster)

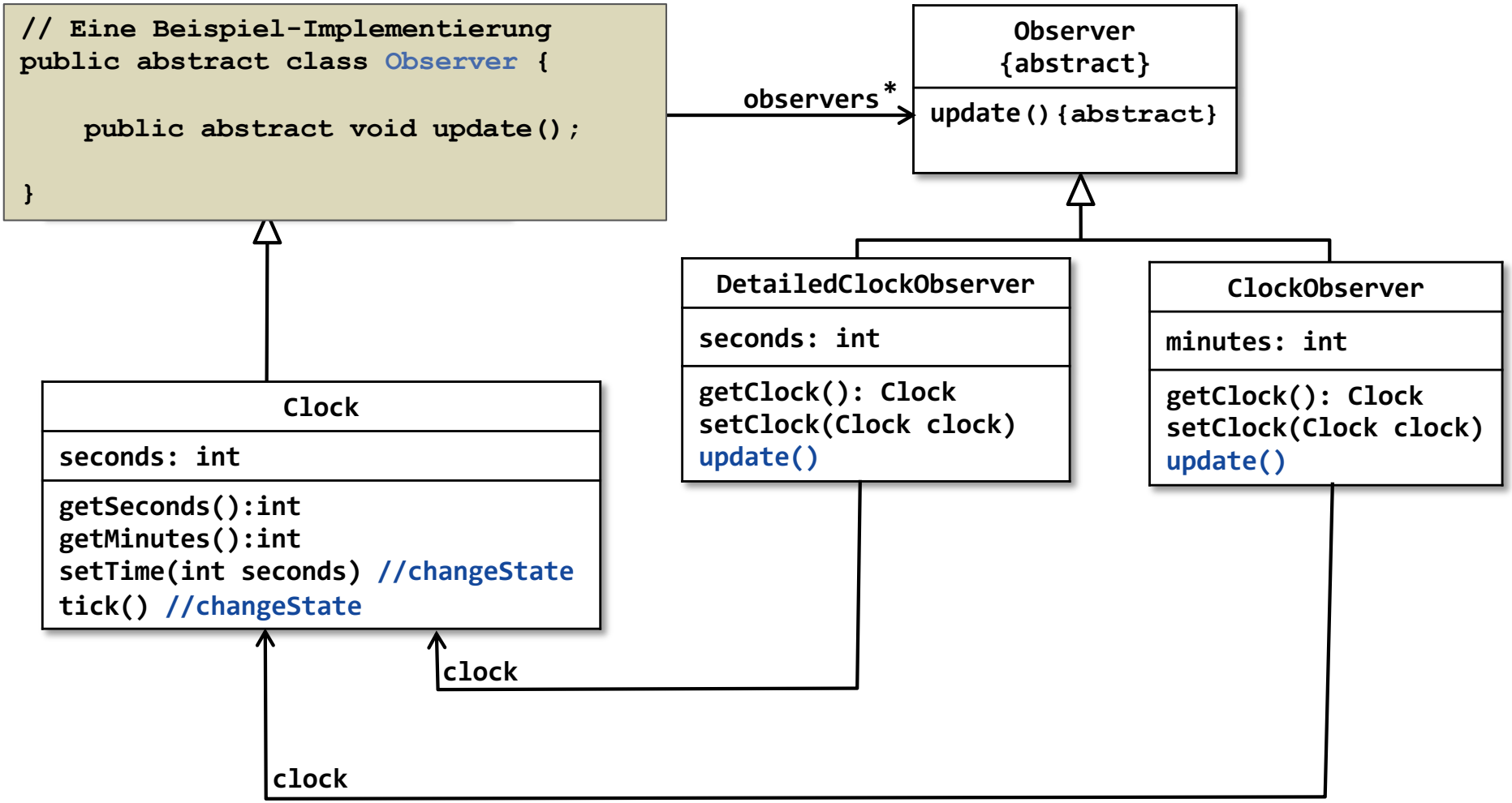
Uhren-Beispiel




```
// Eine Beispiel-Implementierung
import java.util.ArrayList;
import java.util.List;

public abstract class Subject {
     private List<Observer> observers;

     protected Subject() {
          observers = new ArrayList<Observer>();
     }

     public synchronized void addObserver(Observer observer) {
          if(observer != null && !observers.contains(observer))
          {
               observers.add(observer);
          }
     }

     public synchronized void removeObserver(Observer observer)
     {
          if(observer != null && observers.contains(observer))
          {
               observers.remove(observer);
          }
     }

     public synchronized void notifyObservers() {
          for(Observer observer : observers) {
               observer.update();
          }
     }
}
```

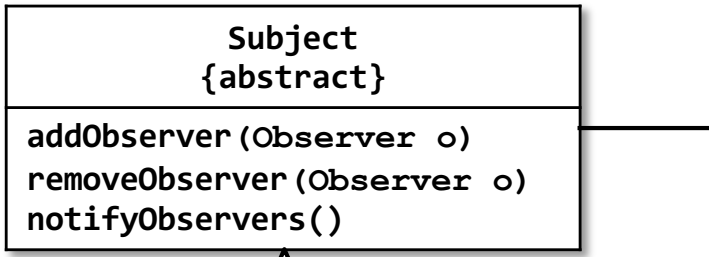
Beobachter-Muster (objektbasiertes Verhaltensmuster)

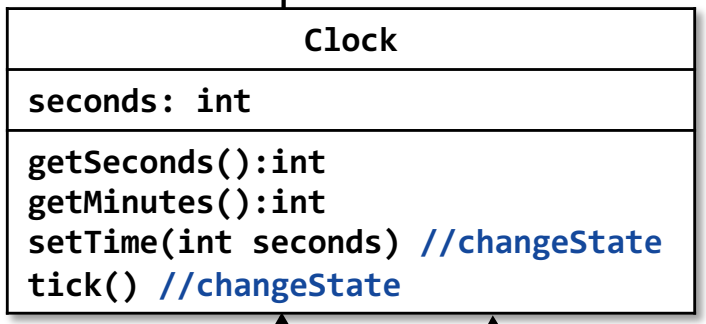
Uhren-Beispiel



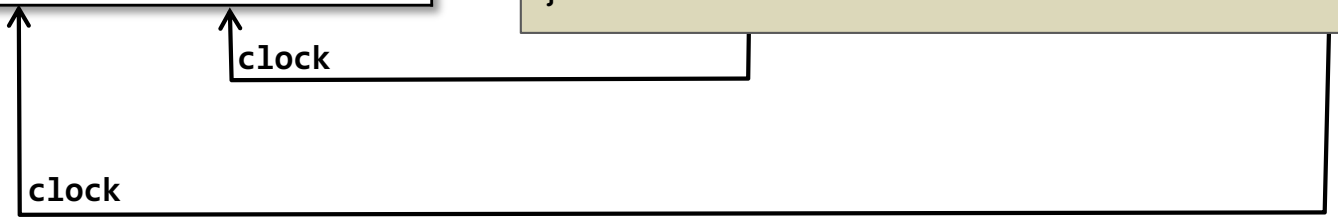
Beobachter-Muster (objektbasiertes Verhaltensmuster)

Uhren-Beispiel



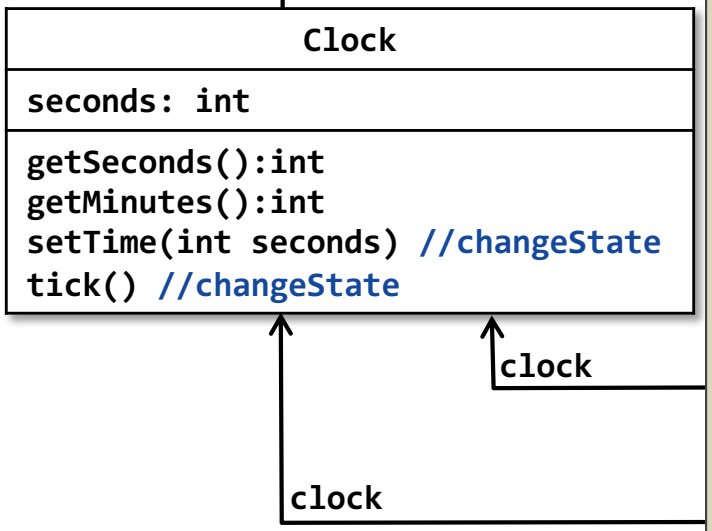



```
public class Clock extends Subject {
    private int seconds;    // state

    public int getSecounds() {
        return seconds;
    }

    public int getMinutes() {
        return seconds / 60;
    }

    public void setTime(int seconds) {
        this.seconds = seconds;
        notifyObservers();
    }

    public void tick() {
        seconds++;
        notifyObservers();
    }
}
```



ensmuster)

```
public class DetailedClockObserver extends Observer {
     private int seconds = -1;    // observer state
     private Clock clock;

     public Clock getClock() { ... }
     public void setClock(Clock clock) { ... }

     @Override
     public void update() {
          if(clock != null && seconds != clock
               seconds = clock.getSeconds();
               System.out.println("time in sec
          }
     }
}
```

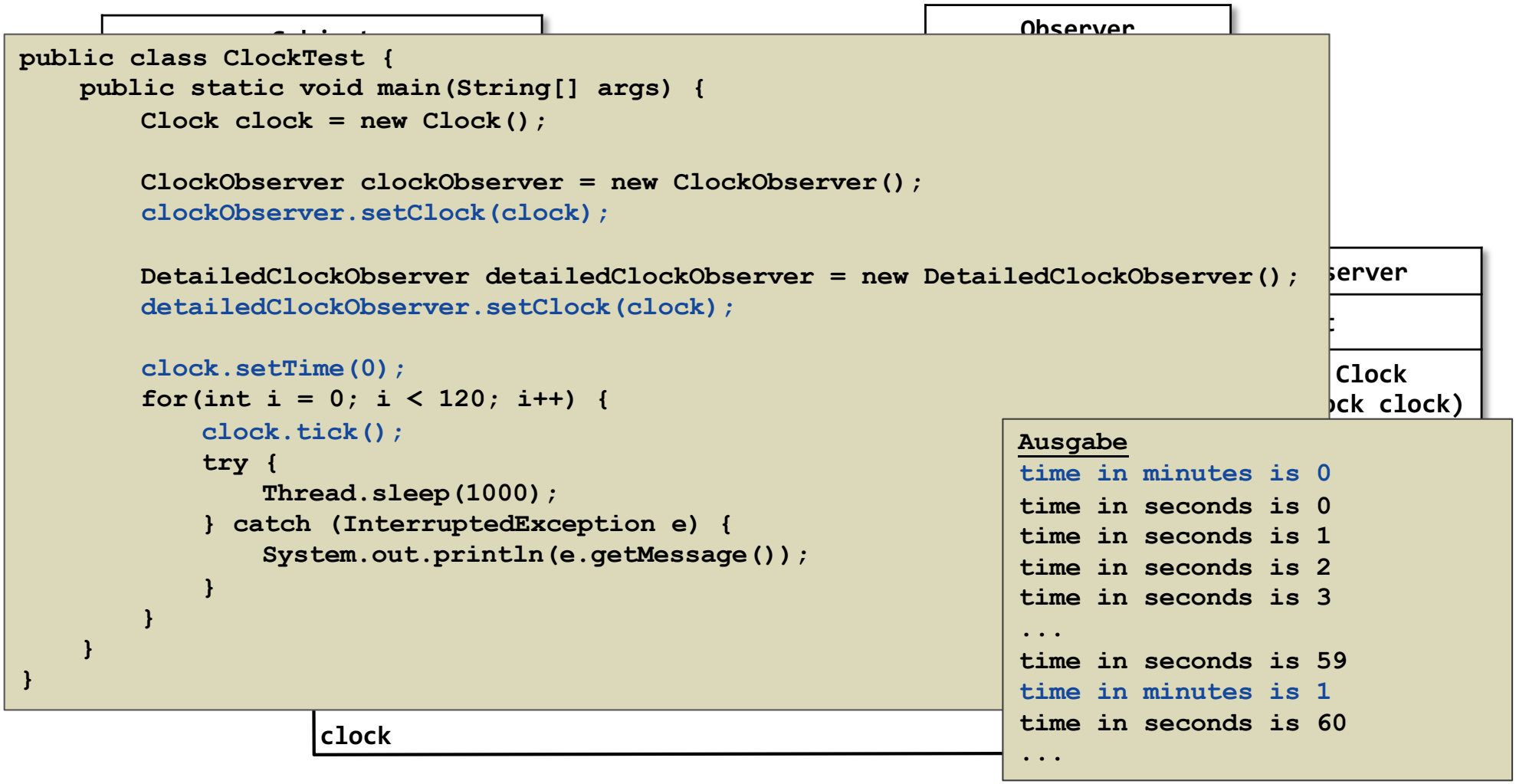


Observer
{abstract}

```
public class ClockObserver extends Observer {
     private int minutes = -1;    // observer state
     private Clock clock;

     public Clock getClock() {
          return clock;
     }

     public void setClock(Clock clock) {
          if(this.clock != clock) {
               if(this.clock != null) {
                    this.clock.removeObserver(this);
               }
               this.clock = clock;
               if(clock != null) {
                    this.clock.addObserver(this);
               }
          }
     }

     @Override
     public void update() {
          if(clock != null && minutes != clock.getMinutes()) {
               minutes = clock.getMinutes();
               System.out.println("time in minutes is " + minutes);
          }
     }
}
```

Beobachter-Muster (objektbasiertes Verhaltensmuster)

Uhren-Beispiel



Struktur

7. Entwurfsmuster

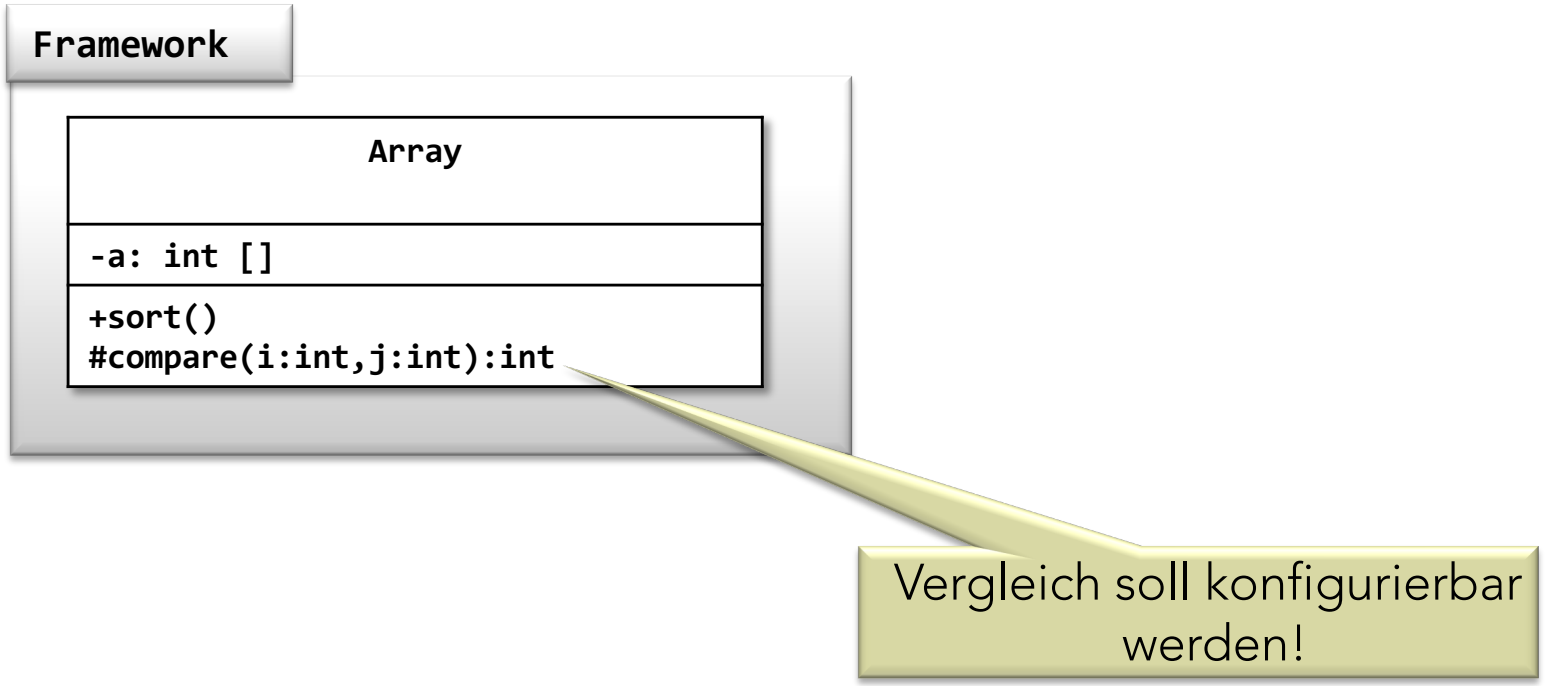
7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2. Beispiele für Muster

Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)

Problemstellung

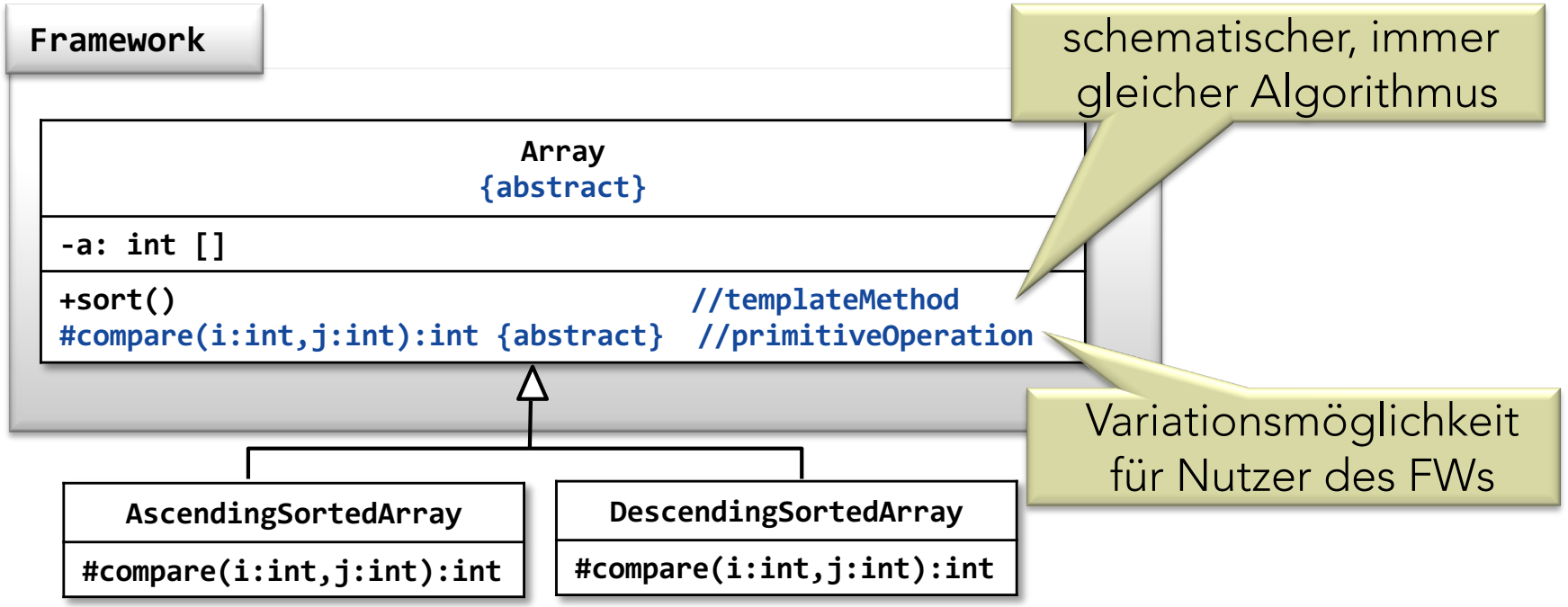
- Array anbieten mit konfigurierbarem Vergleichskriterium



Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)

Problemstellung

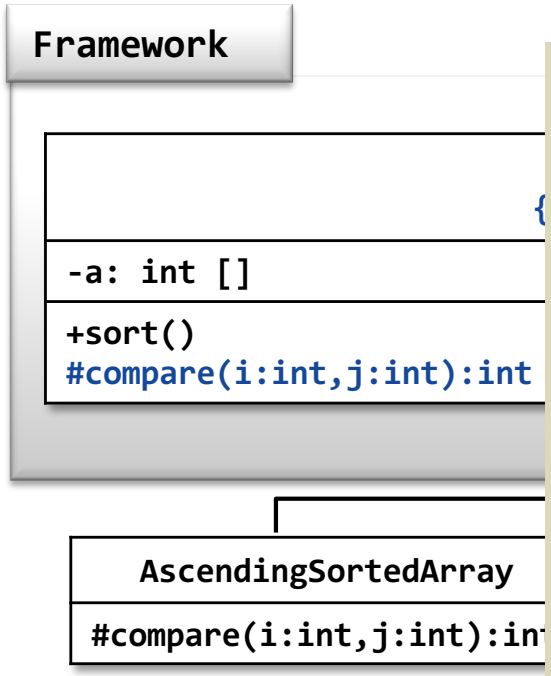
- Array anbieten mit konfigurierbarem Vergleichskriterium



Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)

Problemstellung

- Array anbieten mit konfigurierbarem Vergleichskriterium



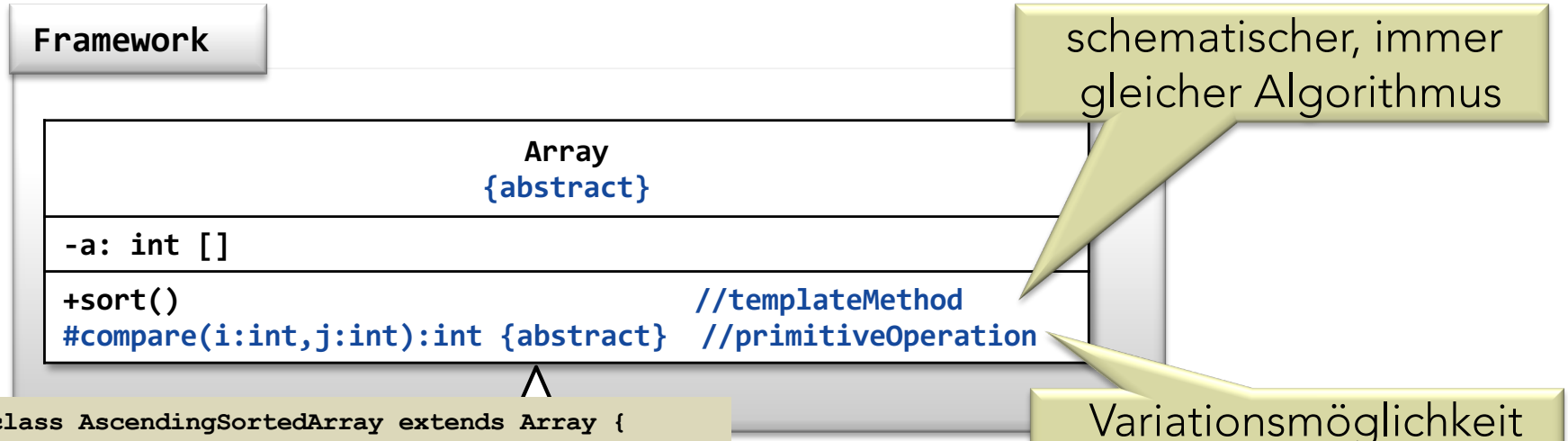
schematischer immer

```
public abstract class Array {
     private int[] a;
     public Array(int[] array) {
          this.a = array;
     }

     public void sort() { // simple bubble sort
          for(int i = a.length-1; i >= 0; --i) {
               for(int j = 0; j < i; ++j) {
                    if((compare(a[j], a[j+1]) > 0) {
                         int temp = a[j];
                         a[j]     = a[j+1];
                         a[j+1]   = temp;
                    }
               }
          }
     }

     protected abstract int compare(int i, int j);

     public int[] getArray() { return a; }
     public void setArray(int[] a) { this.a = a; }
}
```

Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)

Problemstellung

- Array anbieten mit konfigurierbarem Vergleichskriterium




```
public class AscendingSortedArray extends Array {
     public AscendingSortedArray(int[] array) {
          super(array);
     }

     @Override
     protected int compare(int i, int j) {
          if(i == j) {
               return 0;
          } else if(i > j) {
               return +1;
          } else {
               return -1;
          }
     }
}
```

Für kleiner werdende Werte Vorzeichen umdrehen!

Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)

```
public class ArrayTest {
    public static void main(String[] args) {
        int[] array = {2, 4, 3, 1};
        System.out.println("Array");
        System.out.println("-----");
        for(int value : array) {
            System.out.println(value);
        }


        Array ascending = new AscendingSortedArray(array);
        ascending.sort();

        System.out.println("\nAscending Sorted Array");
        System.out.println("----------------------");
        for(int value : ascending.getArray()) {
            System.out.println(value);
        }


        Array descending = new DescendingSortedArray(array);
        descending.sort();

        System.out.println("\nDescending Sorted Array");
        System.out.println("-----------------------");
        for(int value : descending.getArray()) {
            System.out.println(value);
        }
    }
}
```

rium

Ausgabe

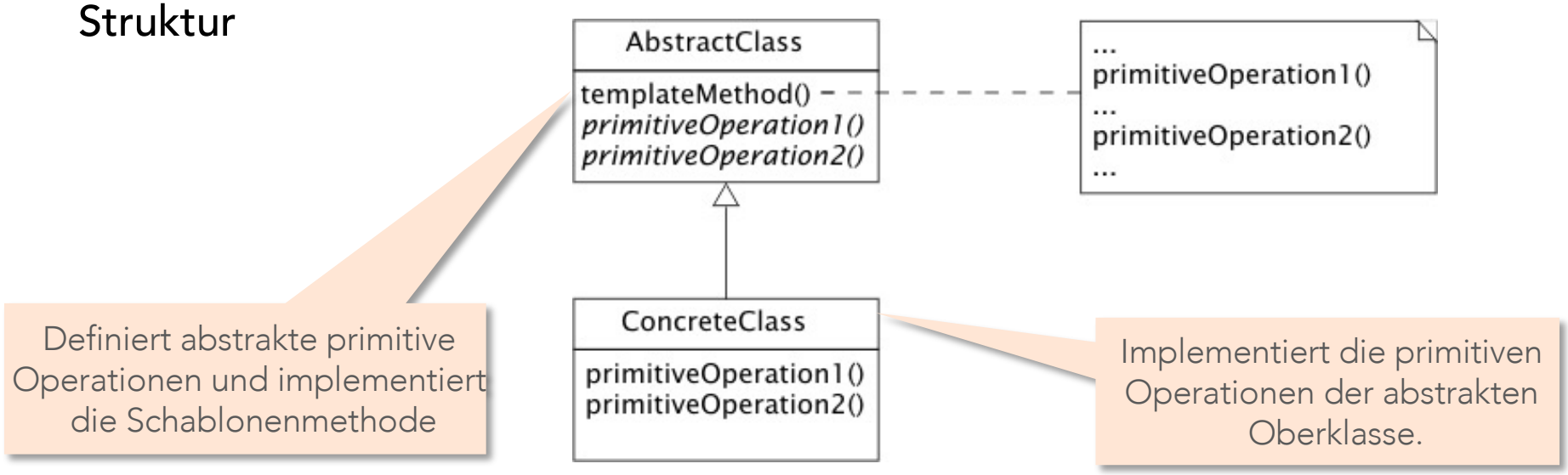
```
Array
-----
2
4
3
1

Ascending Sorted Array
----------------------
1
2
3
4

Descending Sorted Array
-----------------------
4
3
2
1
```

Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)

Anwendbarkeit

- Verwendung des Muster,
  - um die invarianten Teile eines Algorithmus genau einmal festzulegen; konkrete Ausführung der variierenden Teile wird den Unterklassen überlassen
  - wenn gemeinsames Verhalten von Unterklassen in einer Oberklasse realisiert werden soll; Vermeidung der Duplikation von Code

Struktur



Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)

Anwendbarkeit

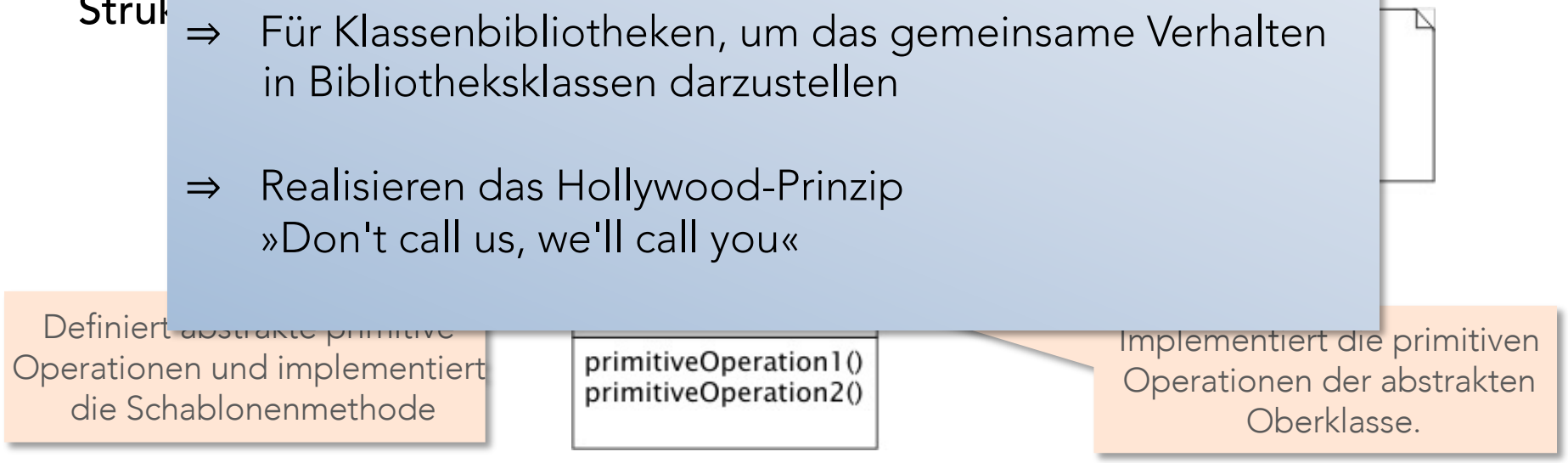
- Verwendung des Muster,
  - um die invarianten Teile eines Algorithmus genau einmal festzulegen; konkrete Ausführung der variierenden Teile wird den Unterklassen überlassen
  - den soll;

⇒ Grundlegende Technik zur Wiederverwendung von Code

⇒ Für Klassenbibliotheken, um das gemeinsame Verhalten in Bibliotheksklassen darzustellen

⇒ Realisieren das Hollywood-Prinzip
»Don't call us, we'll call you«

Struk



Struktur

7. Entwurfsmuster

7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

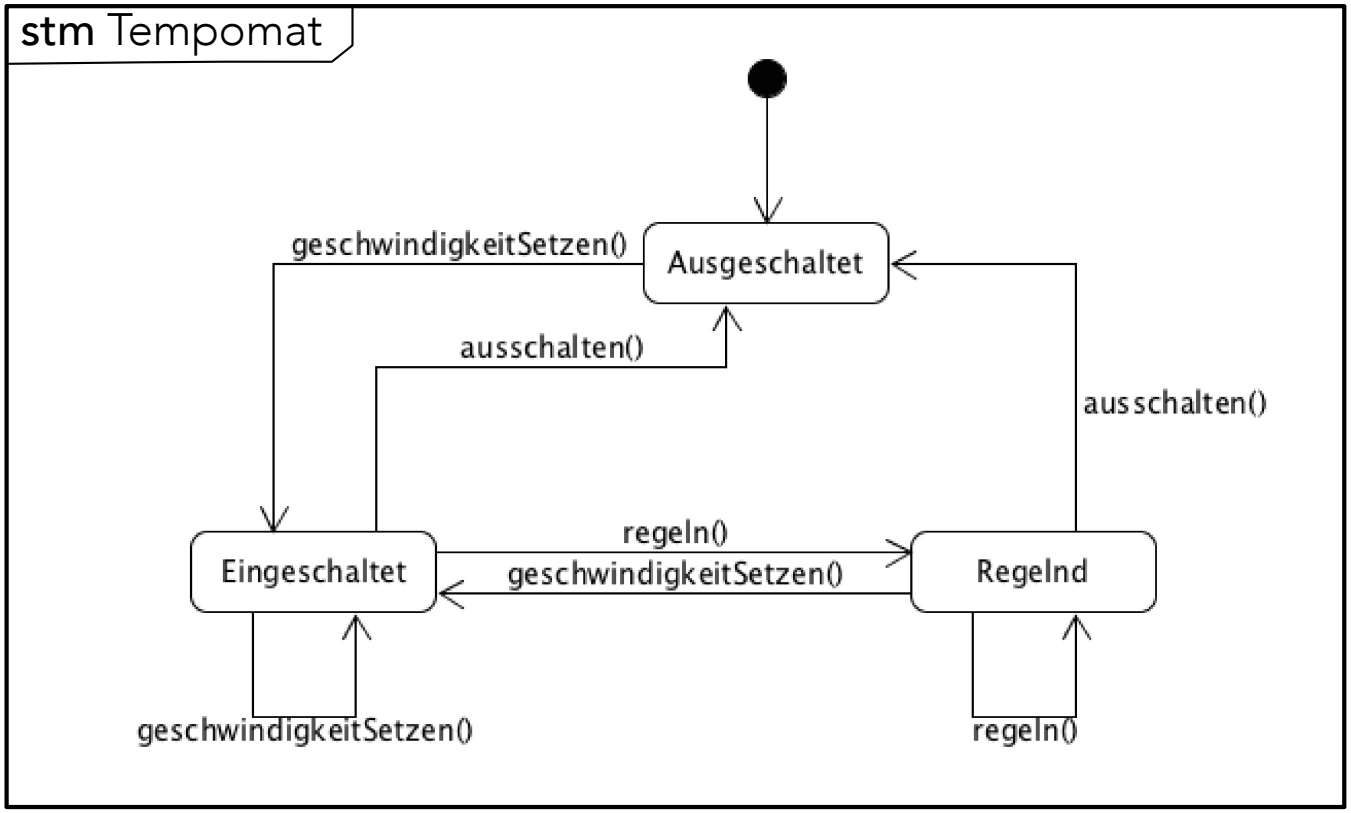
7.2. Beispiele für Muster

- 7.2.1. Fabrikmethode-Muster (klassenbasiertes Erzeugungsmuster)
- 7.2.2. Singleton-Muster (objektbasiertes Erzeugungsmuster)
- 7.2.3. Kompositum-Muster (objektbasiertes Strukturmuster)
- 7.2.4. Proxy-Muster (objektbasiertes Strukturmuster)
- 7.2.5. Fassaden-Muster (objektbasiertes Strukturmuster)
- 7.2.6. Beobachter-Muster (objektbasiertes Verhaltensmuster)
- 7.2.7. Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)
- 7.2.8. Zustands-Muster (objektbasiertes Verhaltensmuster)
- 7.2.9. ....

Zustand-Muster (objektbasiertes Verhaltensmuster)

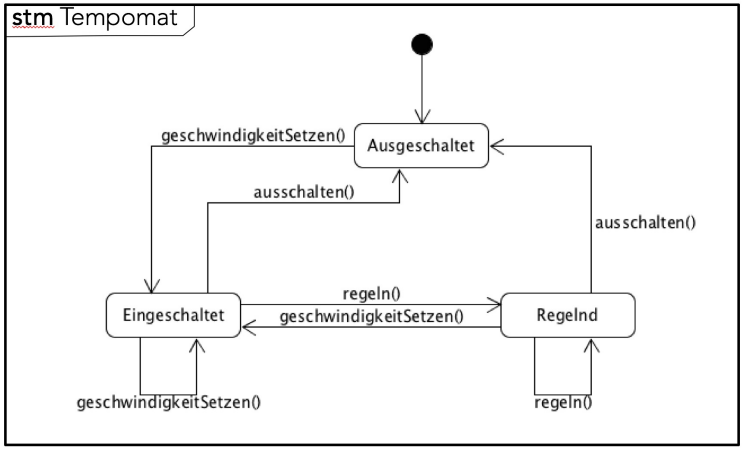
Problemstellung

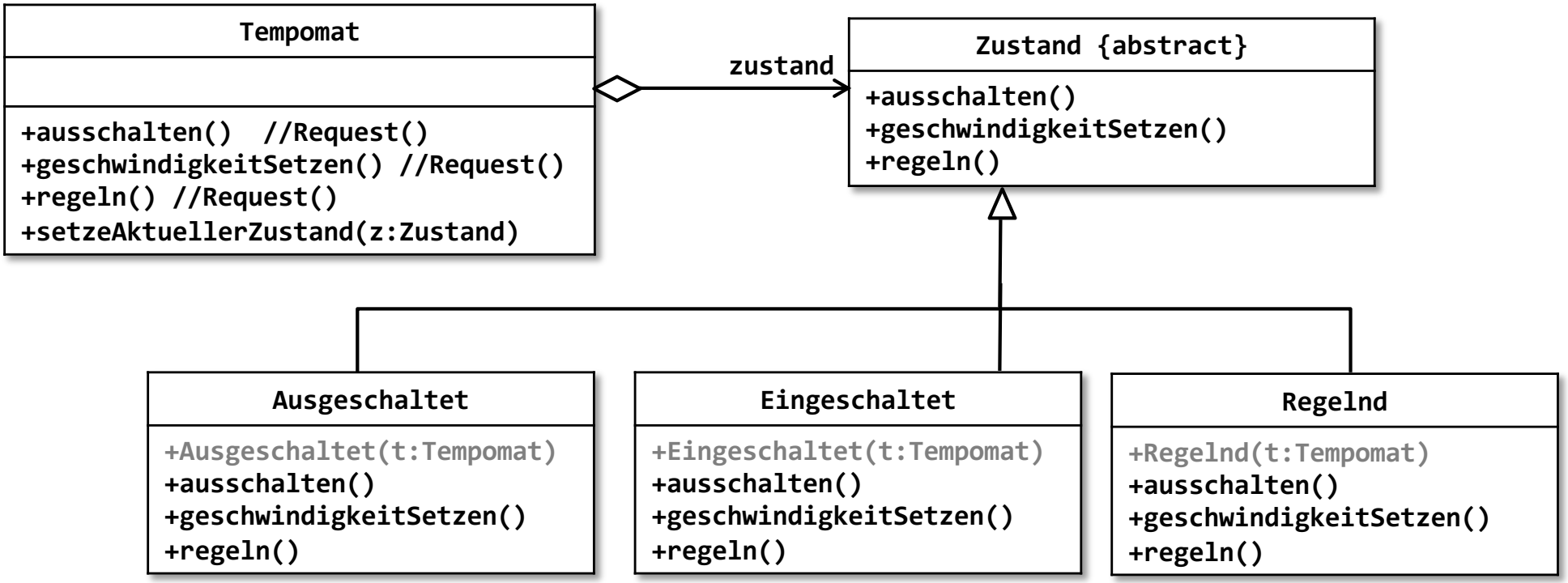
- Verhaltenszustandsautomat ist programmatisch umzusetzen



Zustand-Muster (objektbasiertes Verhaltensmuster)

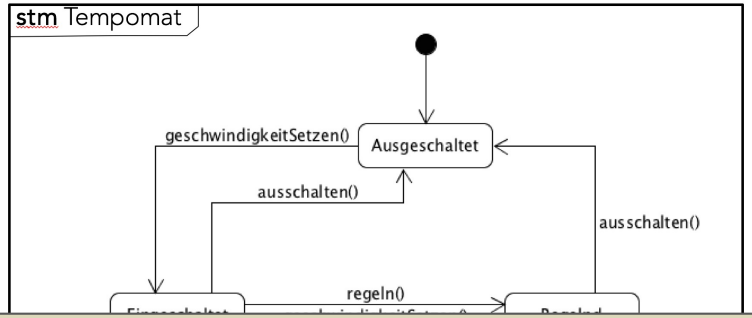
Lösungsansatz

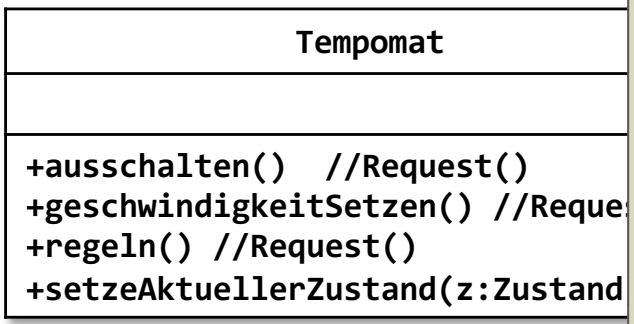


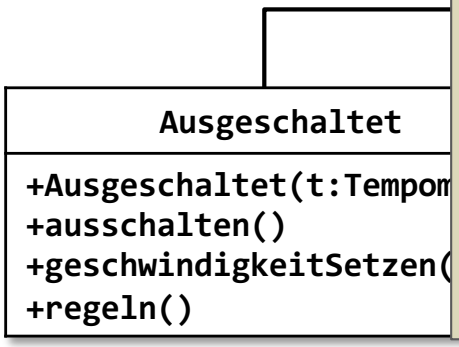


Zustand-Muster (objektbasiertes Verhaltensmuster)

Lösungsansatz



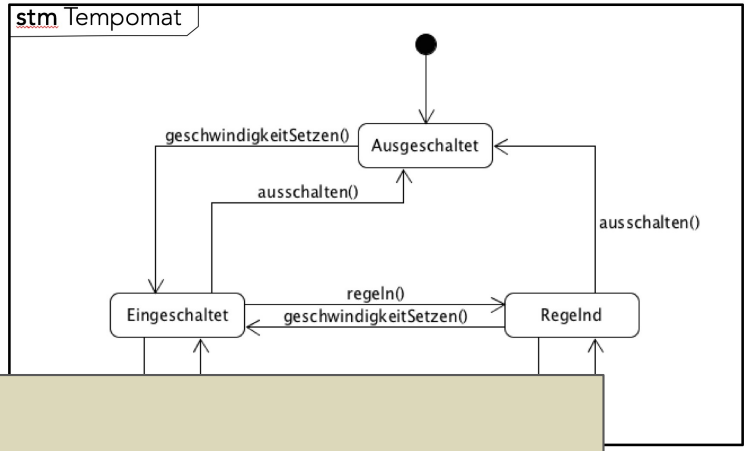





```
public class Tempomat {
     private Zustand aktuellerZustand;

     public Tempomat() {
          setAktuellerZustand(new Ausgeschaltet(this));
     }

     public Zustand getAktuellerZustand() {
          return aktuellerZustand;
     }

     public void setAktuellerZustand(Zustand aktuellerZustand) {
          this.aktuellerZustand = aktuellerZustand;
          System.out.println("Setze Zustand auf: "+ aktuellerZustand);
     }

     public void geschwindigkeitSetzen() {
          aktuellerZustand.geschwindigkeitSetzen();
     }

     public void regeln() {
          aktuellerZustand.regeln();
     }

     public void ausschalten() {
          aktuellerZustand.ausschalten();
     }
}
```

```
public abstract class Zustand {
     protected Tempomat tempomat;
     public Zustand(Tempomat tempomat) {
          this.tempomat = tempomat;
     }
     public abstract void geschwindigkeitSetzen();
     public abstract void regeln();
     public abstract void ausschalten();
}
```

haltensmuster)

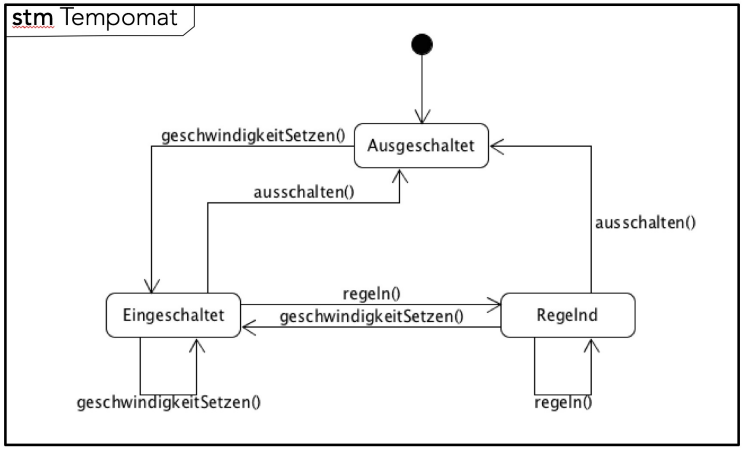



```
public class Ausgeschaltet extends Zustand {

    public Ausgeschaltet(Tempomat tempomat) {
        super(tempomat);
    }

    @Override
    public String toString() {
        return "Ausgeschaltet";
    }

    @Override
    public void geschwindigkeitSetzen() {
        System.out.println("Geschwindigkeit wird gesetzt");
        tempomat.setAktuellerZustand(new Eingeschaltet(tempomat));
    }

    @Override
    public void regeln() {
        System.out.println("Es kann nicht geregelt werden, wenn keine Geschwindigkeit gesetzt wurde");
    }

    @Override
    public void ausschalten() {
        System.out.println("Tempomat bleibt weiterhin ausgeschaltet");
    }
}
```

n()

Zustand-Muster (objektbasiertes Verhaltensmuster)

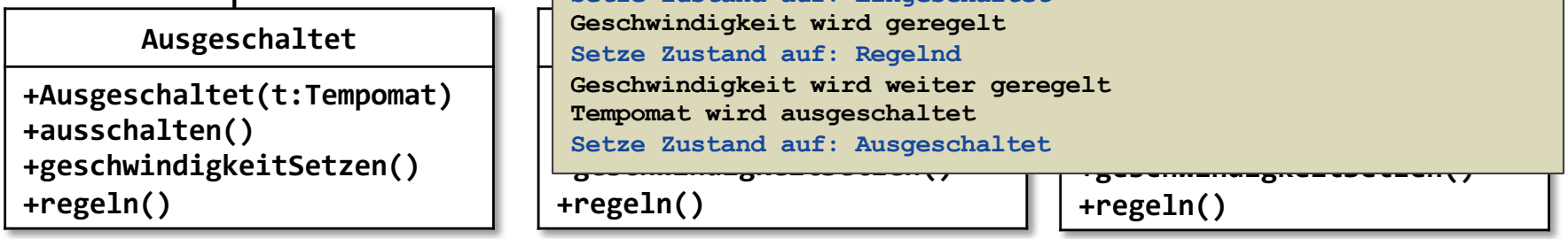
Lösungsansatz




```
public class TempomatTest {

     public static void main(String[] args) {
         Tempomat tempomat = new Tempomat();

         tempomat.geschwindigkeitSetzen();
         tempomat.regeln();
         tempomat.geschwindigkeitSetzen();
         tempomat.ausschalten();
         tempomat.geschwindigkeitSetzen();
         tempomat.regeln();
         tempomat.regeln();
         tempomat.ausschalten();
     }
}
```

Ausgabe

```
Setze Zustand auf: Ausgeschaltet
Setze Zustand auf: Eingeschaltet
Geschwindigkeit wird neu gesetzt
Geschwindigkeit wird geregelt
Setze Zustand auf: RegelInd
Geschwindigkeit wird gesetzt
Setze Zustand auf: Eingeschaltet
Tempomat wird ausgeschaltet
Setze Zustand auf: Ausgeschaltet
Geschwindigkeit wird gesetzt
Setze Zustand auf: Eingeschaltet
Geschwindigkeit wird geregelt
Setze Zustand auf: RegelInd
Geschwindigkeit wird weiter geregelt
Tempomat wird ausgeschaltet
Setze Zustand auf: Ausgeschaltet
```

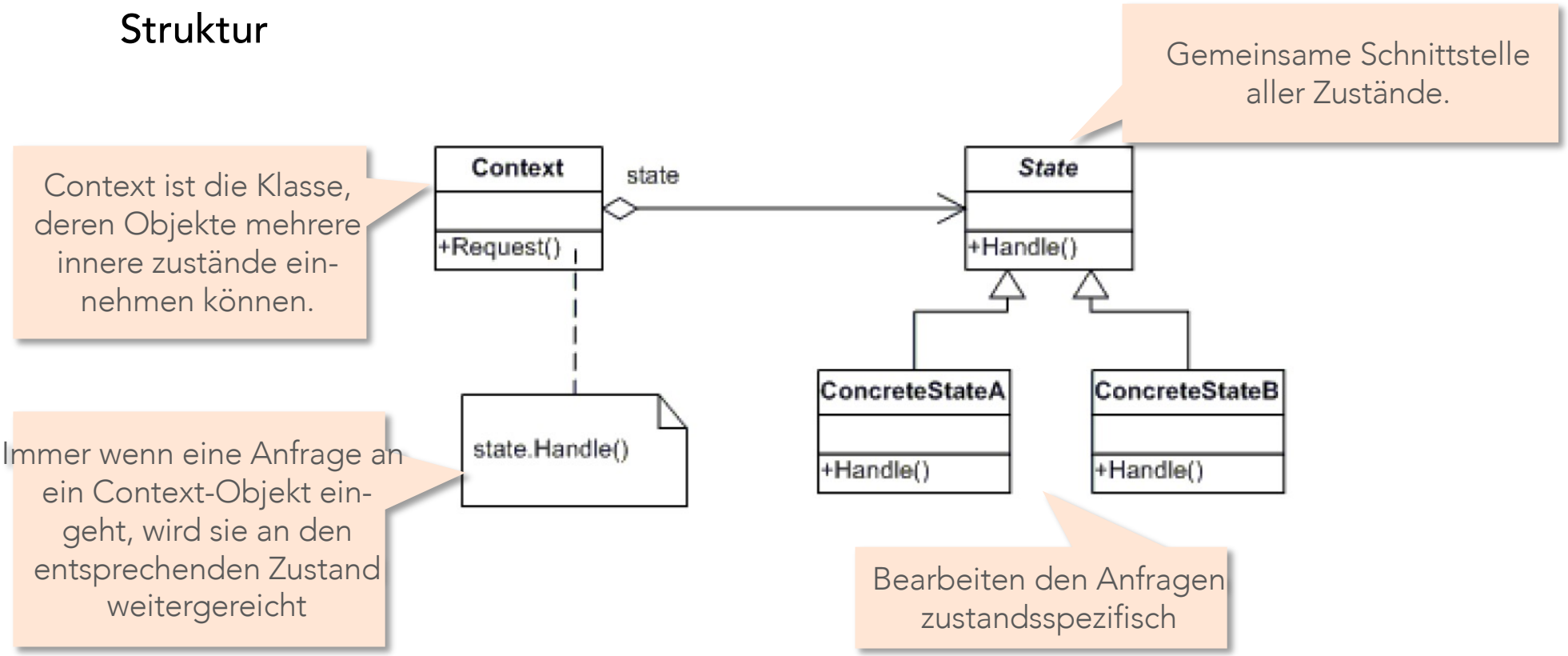


Zustand-Muster (objektbasiertes Verhaltensmuster)

Anwendbarkeit

- Verwendung des Muster,
  - um es einem Objekt zu ermöglichen, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert.
  - es sieht so aus, als ob das Objekt seine Klasse gewechselt hat.

Struktur



Zustand-Muster (objektbasiertes Verhaltensmuster)

Anwendbarkeit

- Verwendung des Muster,
  - um es einem Objekt zu ermöglichen, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert.
  -

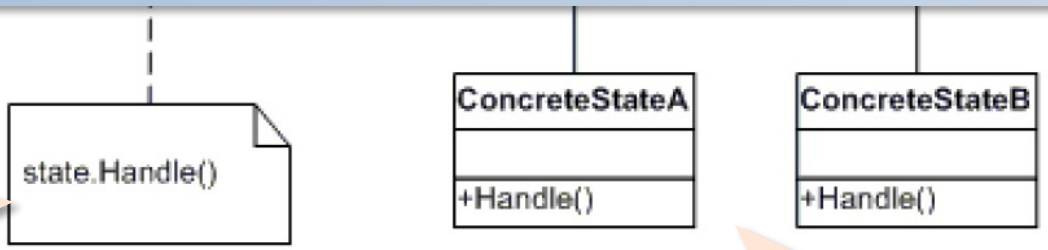
Struk

⇒ Grundlegende Technik zur Realisierung verschiedener zustandsbasierter Verhaltensweisen.

⇒ Kapselung jedes Zustands in einer eigenen Klasse ermöglicht gute Änderbar- und Wartbarkeit

Schnittstelle
stände.

Context ist
deren Objek
innere zust
nehmen



Immer wenn eine Anfrage an ein Context-Objekt ein-geht, wird sie an den entsprechenden Zustand weitergereicht

Bearbeiten den Anfragen zustandsspezifisch

Struktur

7. Entwurfsmuster

7.2. Entwurfsmuster, Frameworks, Klassenbibliotheken

7.2. Beispiele für Muster

- 7.2.1. Fabrikmethode-Muster (klassenbasiertes Erzeugungsmuster)
- 7.2.2. Singleton-Muster (objektbasiertes Erzeugungsmuster)
- 7.2.3. Kompositum-Muster (objektbasiertes Strukturmuster)
- 7.2.4. Proxy-Muster (objektbasiertes Strukturmuster)
- 7.2.5. Fassaden-Muster (objektbasiertes Strukturmuster)
- 7.2.6. Beobachter-Muster (objektbasiertes Verhaltensmuster)
- 7.2.7. Schablonenmethode-Muster (objektbasiertes Verhaltensmuster)
- 7.2.8. Zustands-Muster (objektbasiertes Verhaltensmuster)
- 7.2.9. ….

Entwurfsmuster

Die klassischen Entwurfsmuster

Erzeugende Muster	Strukturelle Muster	Verhaltensmuster
Singleton (Einzelstück)	**Facade(Fassade)**	Mediator (Vermittler)
Prototype (Prototyp)	Decorator (Dekorierer)	Iterator
Factory Method (Fabrikmethode)	Bridge (Brücke)	Interpreter
Builder (Erbauer)	**Composite (Kompositum)**	Command (Kommando)
Abstract Factory (Abstrakte Fabrik)	Adapter	Chain of Responsibility (Zustandigkeitskette)
	Flyweight (Fliegengewicht)	Memento
	Proxy (Stellvertreter)	**Observer (Beobachter)**
		State (Zustand)
		Strategy (Strategie)
		Template Method (Schablonenmethode)
		Visitor (Besucher)

Fragen