



# Design Patterns

<https://github.com/Michaeli71/DesignPatterns>

**Michael Inden**

**Freiberuflicher Consultant, Buchautor und Trainer**

---

# Speaker Intro



- **Michael Inden, Jahrgang 1971**
- **Diplom-Informatiker, C.v.O. Uni Oldenburg**
- **~8 ¼ Jahre SSE bei Heidelberger Druckmaschinen AG in Kiel**
- **~6 ¾ Jahre TPL, SA bei IVU Traffic Technologies AG in Aachen**
- **~4 ¼ Jahre LSA, Trainer bei Zühlke Engineering AG in Zürich**
- **~3 Jahre TL / CTO bei Direct Mail Informatics / ASMIQ in Zürich**
- **Derzeit Freiberuflicher Consultant, Autor und Trainer**
- **Regelmässiger Speaker / Trainer auf Konferenzen wie (W-)JAX, JAX London, ch.open, Oracle Code One**

E-Mail: [michael.inden@hotmail.ch](mailto:michael.inden@hotmail.ch)

Blog: <https://jaxenter.de/author/minden>



# Agenda

---



- **Part 1: Einführung**
- **Part 2: Erzeugungsmuster**
  - Erzeugungsmethode
  - Fabrikmethode (Factory Method)
  - Erbauer (Builder)
  - Singleton (Singleton)
  - Prototyp (Prototype)
  - *Abstrakte Fabrik (Abstract Factory)\**

\* Von den Teilnehmern als Mini-Workshops ausgearbeitet

---

# Agenda

---



- **Part 3: Strukturmuster**

- Adapter
- Dekorierer (Decorator)
- Fassade (Facade)
- Kompositum (Composite)
- Proxy
- **Dynamic Proxy in Java als Zusatzwunsch**
- *Fliegengewicht (Flyweight)*
- *Brücke (Bridge)*

# Agenda

---



- Part 4: Verhaltensmuster
  - Iterator
  - Schablonenmethode (Template Method)
  - Strategie (Strategy)
  - Null-Objekt (Null Object)
  - Befehl (Command)
  - Beobachter (Observer)
  - Model View Controller (MVC)
  - Vermittler (Mediator)
  - Zustand (State)
  - Zuständigkeitskette (*Chain of Responsibility*)
  - Besucher (Visitor)
  - Interpreter
  - Memento
- Blackboard

# Agenda

---



- **Part 5: Entwurfsmuster im Kontext von Java 8**

- Builder
- Factory
- Template Method
- Strategy
- Execute Around



# Part 1: Einführung

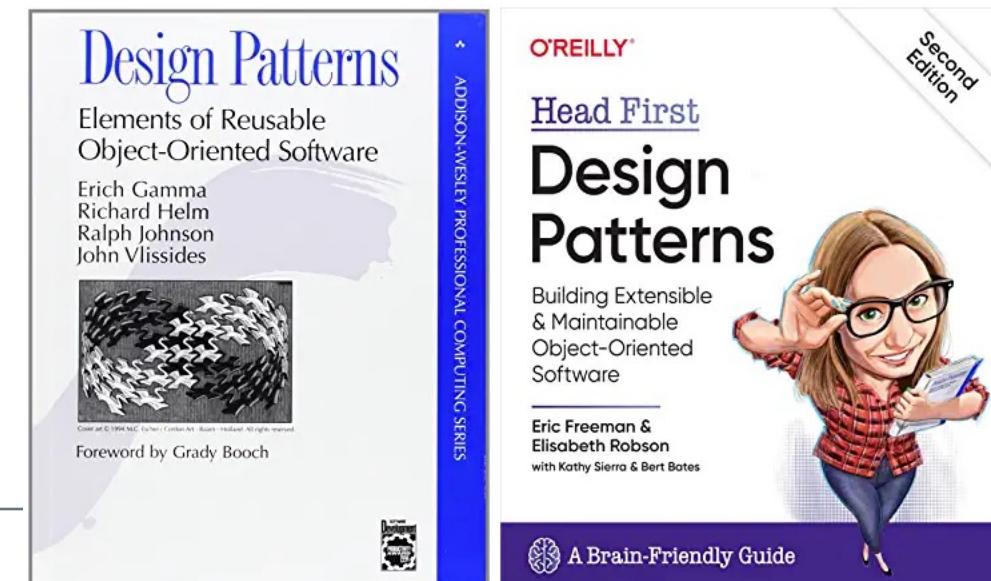


# Agenda

---



- Vielfach **erprobte** und **regelmäßig eingesetzte** allgemeingültige **Verfahren** zur Lösung eines Entwurfsproblems werden **Entwurfsmuster** oder auch Designpattern genannt.
- Jedes Entwurfsmuster hilft, ein **spezifisches Problem** auf eine **dokumentierte Art zu lösen**.
- Populär durch das Buch »**Design Patterns – Elements of Reusable Object Oriented Software**« von der sogenannten **Gang of Four (GoF)**: Gamma, Helm, Johnson und Vlissides
- Mitte der 1990er-Jahre löste das Buch einen Run aus
- Viele weitere Bücher sind zu diesem Thema erschienen



# Aufgliederung

---



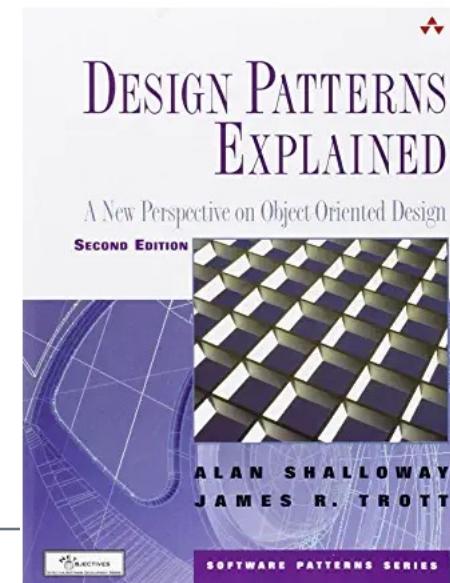
- Auch hier erfolgt die von der GoF vorgenommene Gliederung in die drei Kategorien
  - 1. **Erzeugungsmuster** -- Der Einsatz von Erzeugungsmustern kann den **Konstruktionsprozess** von Objekten vereinfachen kann. Strukturmuster
  - 2. **Strukturmuster** -- **Funktionalitäten** flexibel erweitern und anpassen
  - 3. **Verhaltensmuster** -- Durch den Einsatz von Verhaltensmustern strukturiert und vereinfacht man komplexe **Abläufe** oder **Interaktionen** zwischen Objekten.
- Naheliegende **Fehlverwendungen** werden als **Anti-Pattern** beschrieben

# Eigenschaften

---



- Jedes **Entwurfsmuster** trägt einen **eindeutigen Namen** und beschreibt eine **Lösungsidee** für ein Entwurfsproblem.
- Verwendung von Entwurfsmustern kann zu **qualitativ hochwertiger Software** beitragen
- **Kommunikation** von Designentscheidungen unter Entwicklern **erleichtert => Entwurfssprache**
- Diskussionen auf der **verständlichen Ebene von Konzepten** möglich
- Infos zur gemeinsamen Designsprache finden sich im empfehlenswerten Buch »Design Patterns Explained« [66] von Alan Shalloway und James R. Trott



## Bedenkenswertes

---

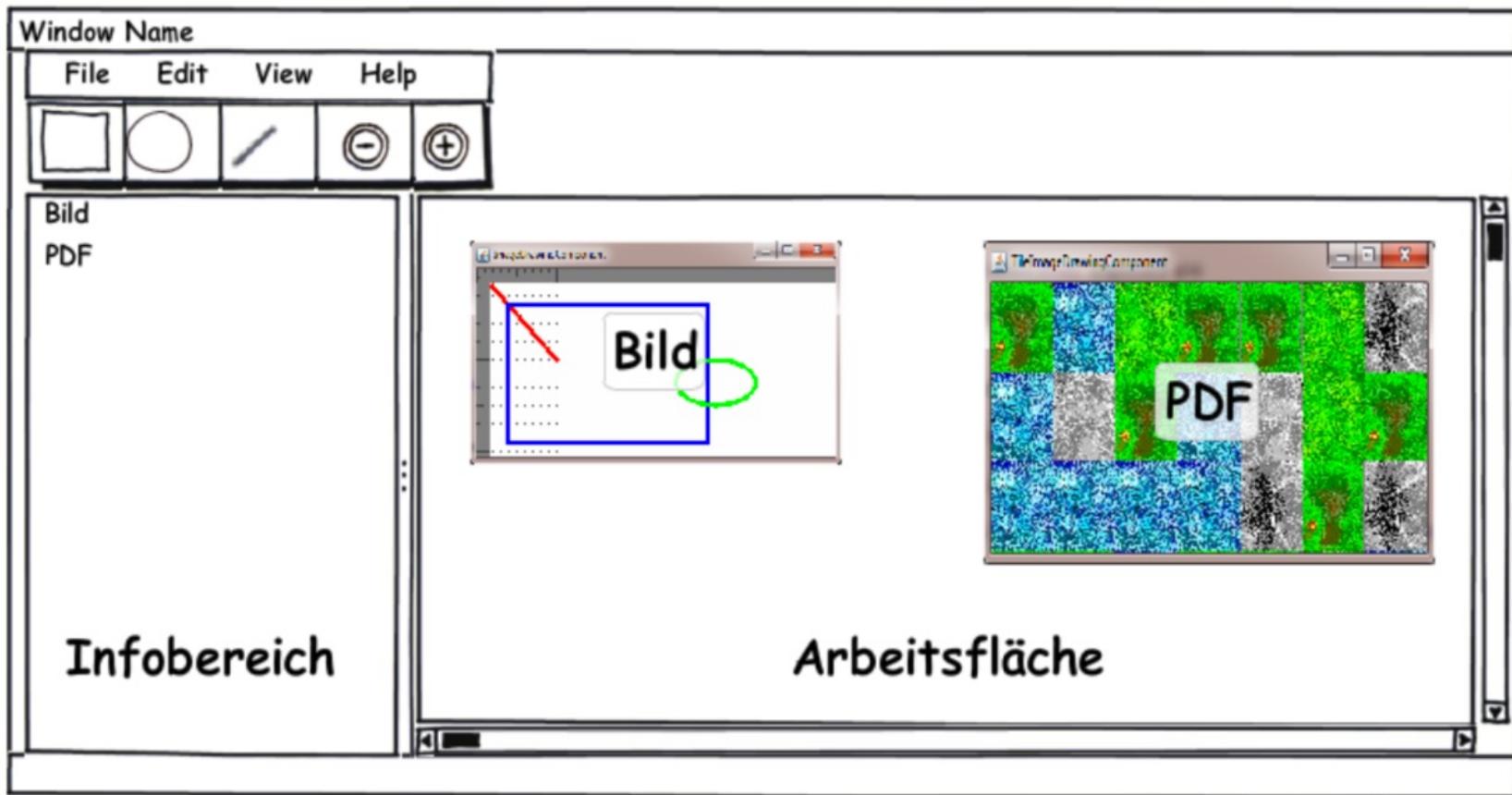


- *Entwurfsmuster* helfen, ein **Problem** auf eine **dokumentierte** Art und Weise **zu lösen**
  - Durch Verwendung von Entwurfsmustern kommt es aber **nicht automatisch zu einem guten Design und sinnvollen Lösungen**
  - **Allein durch Einsatz von Entwurfsmustern betreibt man nicht automatisch erfolgreiche Softwareentwicklung! => Pattern-itis / Pattern-o-holics**
  - Zwar erreicht man **oft größere Flexibilität**, allerdings entsteht **mehr Sourcecode** und weitere **Komplexität**.
  - **Der Einsatz von Entwurfsmustern sollte daher immer auf die jeweilige Situation abgestimmt werden.**
-

# Beispielapplikation



Einige Entwurfsmuster werden im Kontext eines **grafischen Editors** vorgestellt, etwa sollen **Zeichenfunktionen** für **Figuren**, wie Linien, Rechtecke und Kreise, existieren. Zudem ist die Anzeige von **Bildern** gefordert. Die **Bedienung** soll über **Menüs**, **Kontextmenüs** und **Toolbars** erfolgen. Als **Layouthilfe** sind ein **Lineal** und ein **Raster** gewünscht.



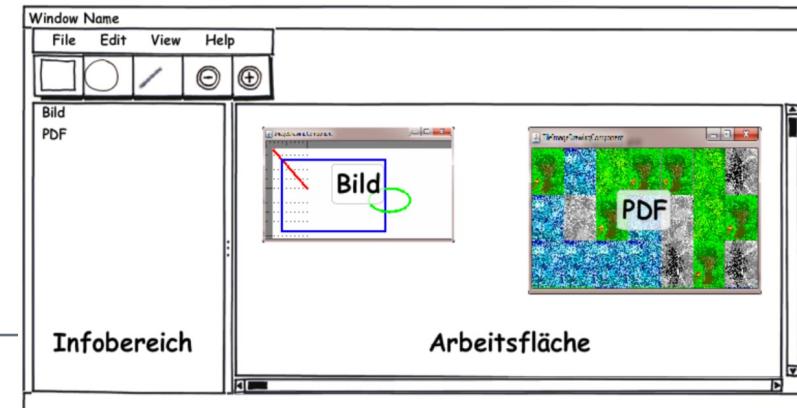
# Entwurfsideen



Die zu zeichnenden **Elemente**, also die grafischen Figuren, wollen wir einheitlich behandeln. Daher definieren wir ein **gemeinsames Interface** sowie eine **abstrakte Basisklasse**

**Wir werden zur Konstruktion des grafischen Editors folgende Muster nutzen:**

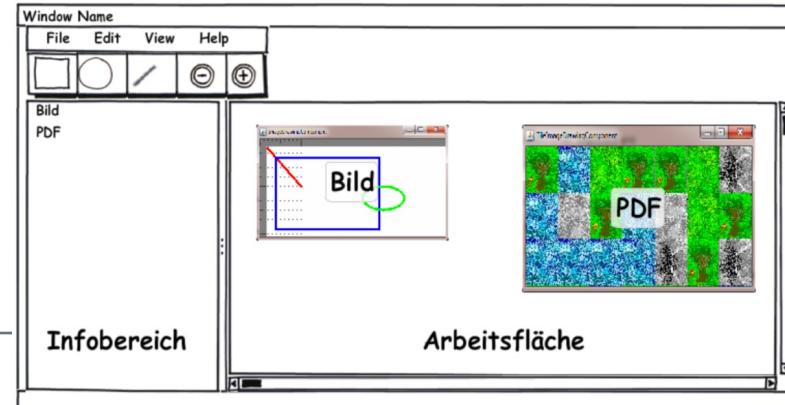
- **BEFEHL (COMMAND)** – In Menüs und Toolbars sind vielfach Aktionen mit gleicher Funktionalität definiert. Diese Aktionen, wie etwa Laden oder Speichern, können über einen Menüeintrag, Shortcut, Toolbar-Button usw. ausgelöst werden. Um Sourcecode-Duplikation zu vermeiden, werden die **Aktionen** in Form von **speziellen Befehlsobjekten** implementiert. Eine **Undo-Funktionalität** wird durch Einsatz dieses Musters **nachträglich leichter realisierbar**.
- **BEOBACHTER (OBSERVER)** – **Änderungen** im Modell werden an **verschiedene Views propagiert** und führen zu Modifikationen in der Übersichtsdarstellung und auf der Zeichenfläche.



# Struktur der Musterbeschreibungen



- **ERZEUGUNGSMETHODE (CREATE METHOD)** und **FABRIKMETHODE (FACTORY METHOD)** – Beide helfen beim *Erzeugen von Objekten*.
- **ITERATOR** – Dieses Muster (in Java eher ein **Idiom**) verwenden wir häufig zum *Durchlaufen* eines Datenmodells, etwa beim Zeichnen verschiedener Figuren.
- **PROTOTYP (PROTOTYPE)** – Dieses Muster erleichtert *Copy-Paste-Operationen*
- **SCHABLONENMETHODE (TEMPLATE METHOD)** – Das Zeichnen erfolgt nach einem genau *definierten Algorithmus*: zunächst Lineale sowie Gitterpunkte und anschließend Figuren
- **MODEL-VIEW-CONTROLLER (MVC)** – Views (Arbeitsfläche und Übersichtsdarstellung) mit *verschiedenen Ansichten gleicher Modelldaten*





Die nachfolgende Vorstellung der jeweiligen Muster besitzt nahezu immer folgende Struktur:

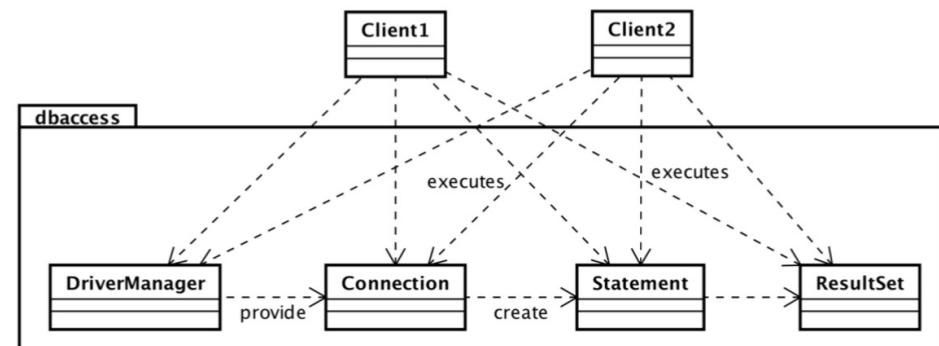
1. Motivation und Kurzbeschreibung
2. Darstellung als UML-Klassendiagramm
3. Anwendungsbeispiel
4. Bewertung

## Beschreibung und Motivation

Das Entwurfsmuster FASSADE kann dabei helfen, die Komplexität eines Subsystems zu verbergen und den Zugriff darauf für externe Klassen zu vereinfachen bzw. zu steuern. Eine Fassadenklasse definiert dazu ein »High-Level«-Interface, um komplizierte, sehr feingranulare Interaktionen und Beziehungen zwischen Package-internen und -externen Klassen zu vermeiden. Dazu delegiert ein Fassadenobjekt die Aufrufe durch Klienten entsprechend der Zuständigkeit an spezielle Klassen des Subsystems. Es herrscht eine gerichtete Abhängigkeit: Die Fassadenklasse kennt die Klassen des Subsystems, aber nicht umgekehrt, d. h., keine Klasse des Subsystems kennt die Fassadenklasse.

## Struktur

Betrachten wir zunächst ein System mit zwei Klienten Client1 und Client2, die Datenbankzugriffe durchführen wollen. In Abbildung 18-9 erkennt man viele Abhängigkeiten von den eingesetzten Klassen. Dies liegt daran, dass die Logik und Steuerung jeweils in den Klienten erfolgt.





## Was ist die UML?

---

- **UML** = Unified Modelling Language
- grafische Notationsform für die objekt-orientierte Analyse und Design (OOA/OOD)
- einheitliche Sprache bei der Architektur und der Modellierung von Software
- Begründer waren die „drei Amigos“ Grady Booch, James Rumbaugh und Ivar Jacobson

<b>Objekt-Guru</b>	<b>Modellierungstechnik</b>
Grady Booch	OOAD - Object Oriented Analysis and Design
James Rumbaugh	OMT - Object Modeling Technique
Ivar Jacobson	OOSE - Object-Oriented Software Engineering



## Ziele der UML

---

- keine Festlegung auf spezielle Programmiersprache
- universelle **Beschreibungssprache** für objektorientierte Softwaresysteme
- standardisierte Sprache mit deren Hilfe Entwickler ihre **Modelle beschreiben** und mit anderen Entwicklern **austauschen** können
- Alle Phasen der Softwareentwicklung abgedeckt: Anforderungsanalyse bis Timing-Verhalten
- Beschreibung **statischen** und **dynamischen** Verhaltens



Strukturdiagramme

Verhaltensdiagramme



## Das kongeniale Duo: UML und Entwurfsmuster

- **Früher:**

Wollte man einem anderen Software-Entwickler eine Realisierung beschreiben,  
Dazu waren **viele Worte und Zeichnungen nötig + unterschiedliche Notationen**  
**und kaum gemeinsame Entwurfssprache!**
- **Heute:**
  - UML bietet uns, eine standardisierte Notation
  - Entwurfsmuster helfen uns, Designentscheidungen zu kommunizieren

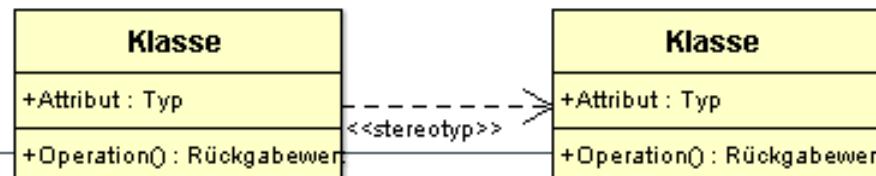


# Klassendiagramm

- **Ziel:** Beschreibt statische Elemente des Systems sowie deren Beziehung.
- **Wichtigste Bestandteile:**
  - **Klasse:** repräsentiert ein Element mit seinen Merkmalen (Attribute und Operationen). Die Art eines Elementes kann durch sogenannte Stereotypen näher spezifiziert werden.
  - **Interface:** repräsentiert eine Menge von (kohärenten) Operationen. Unterschied zur Klasse: das „was“ wird spezifiziert, nicht das „wie“. Entspricht konzeptionell einem „Typ“.



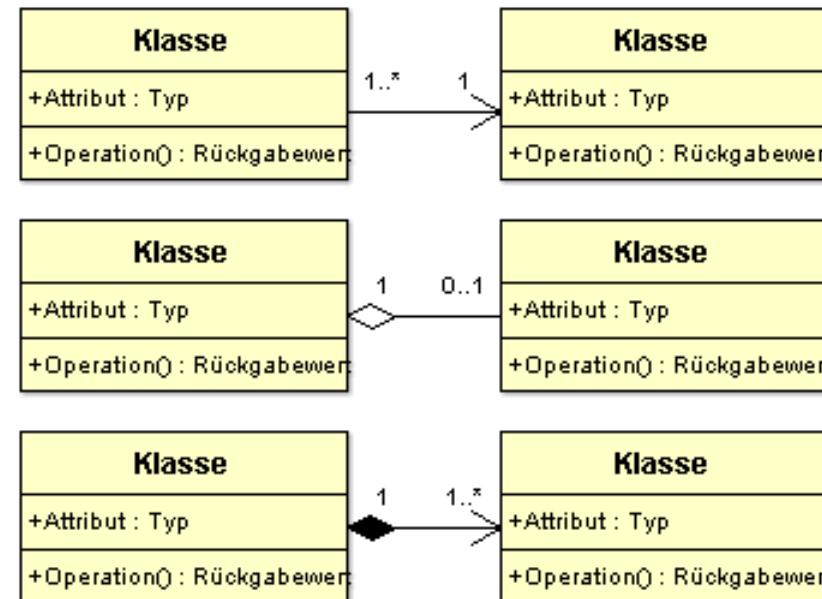
- **Abhängigkeiten:** allgemein, die über Stereotypen konkretisiert werden kann.





## Klassendiagramm

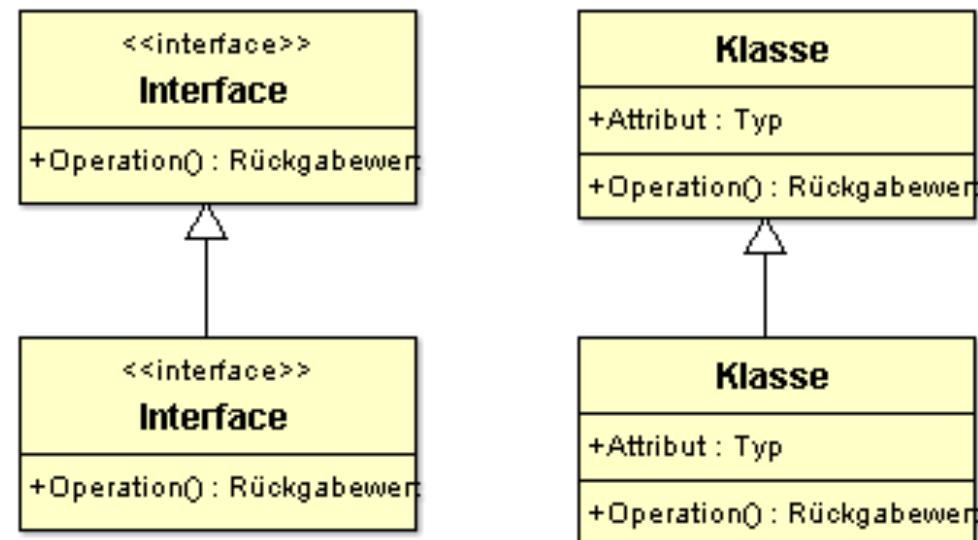
- **Beziehungen zwischen Instanzen von Klassen**
  - **Assoziation:** Allgemeine Beziehung
  - **Aggregation:** Beziehung zwischen einem „Ganzen“ und seinen „Teilen“
  - **Komposition:** Beziehung zwischen einem „Ganzen“ und seinen „Teilen“, wobei die Lebenszeit der Teile an die des Ganzen gebunden ist.





# Klassendiagramm

- **Generalisierung:**
  - Zwischen **Klassen**: Konzeptionell, eine Klasse ist eine spezielle Ausprägung einer anderen Klasse. Aus Nutzungssicht, vererben (Wiederverwenden) von Verhalten.
  - Zwischen **Interfaces**: Ein Interface erweitert die Menge der Operationen eines anderen Interfaces. Konzeptionell, Typing/Sub-Typing.

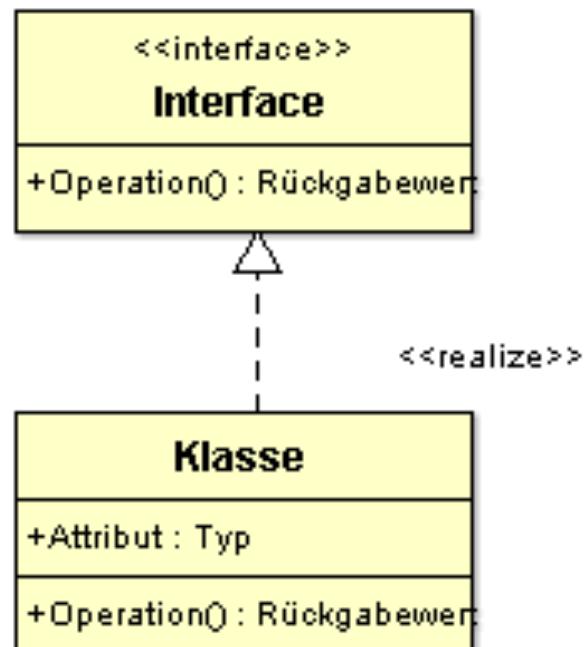




## Klassendiagramm

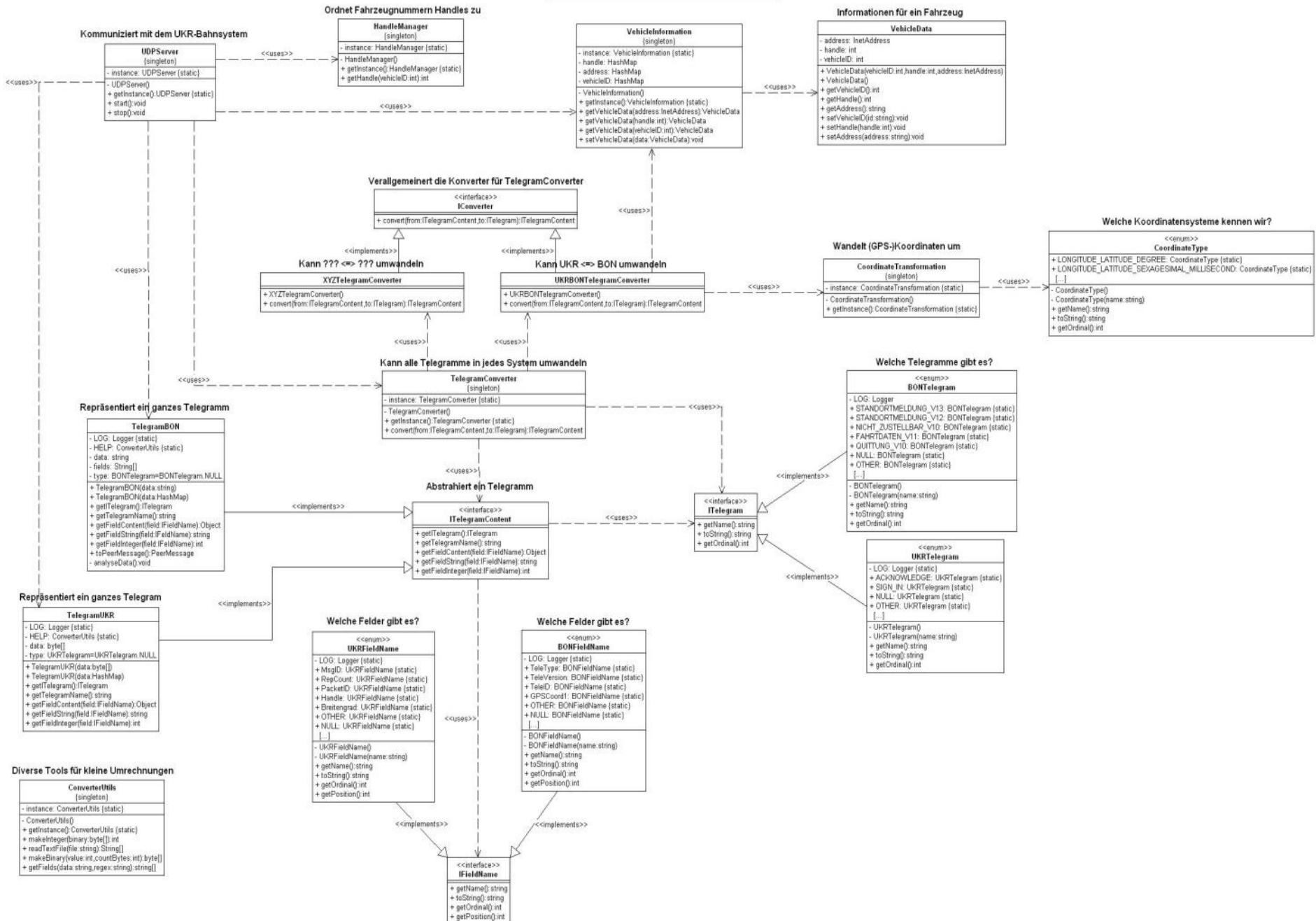
- **Realisierung:**

Beziehung zwischen einem Interface und einer Klasse. Die öffentliche Schnittstelle der Klasse ist konform zu den Operationen des Interfaces. „Vererben“ von Typ-Informationen, Typ-Konformität.





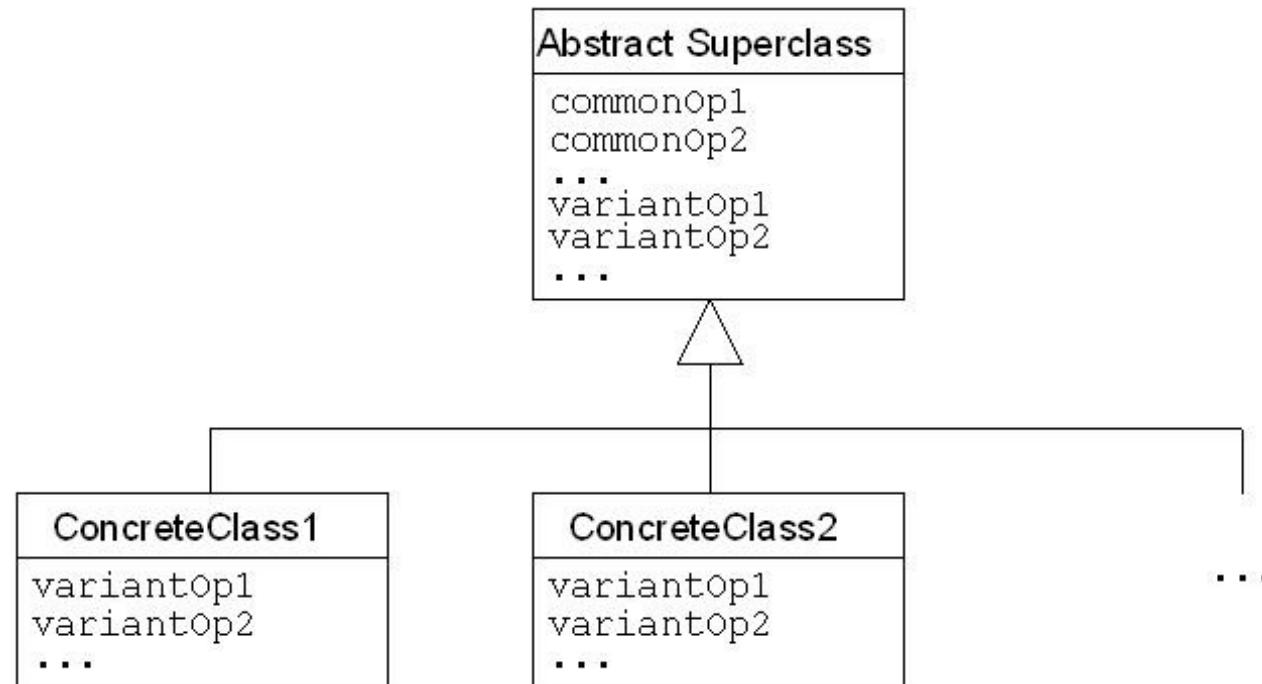
## Entwurf für Converter UKR <=> BON / RBL, Version 6





## Abstrakte Basisklassen

- Fortführung von OO-Design-Gedanken der **Sammlung von Funktionalität** in Oberklassen
- Generalisierung erzeugt evtl. **Basisklasse**, die **selbst nicht mehr sinnvoll instanziert** werden kann, da ihr **Modellierungsdetails fehlen**



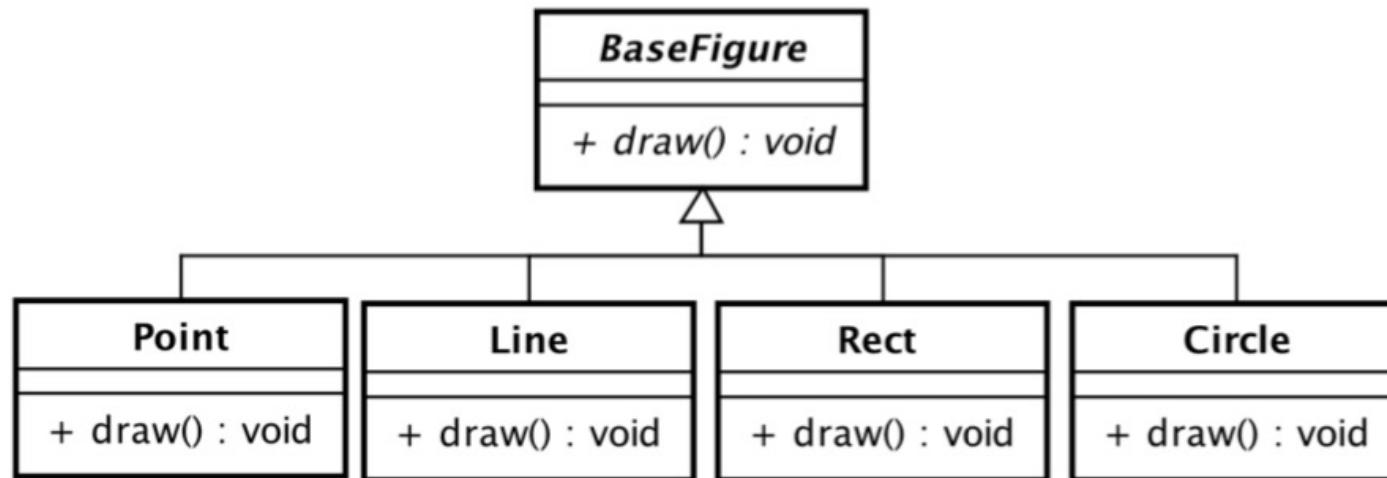
# Abstrakte Basisklassen – Beispiel Grafische Figuren



- **Figuren = Klassen mit Methoden, um sich zu zeichnen**



- **Figuren sollen einheitlich behandelt werden: Abstrakte Basisklasse**





## Vor- und Nachteile

[+] Konsistente Implementierung der Funktionalitäten, Subklassen bringen nur an speziellen, gewünschten Stellen ihre Realisierungen ein.

[+] Vermeidet Code-Duplikation und Wartungsaufwand

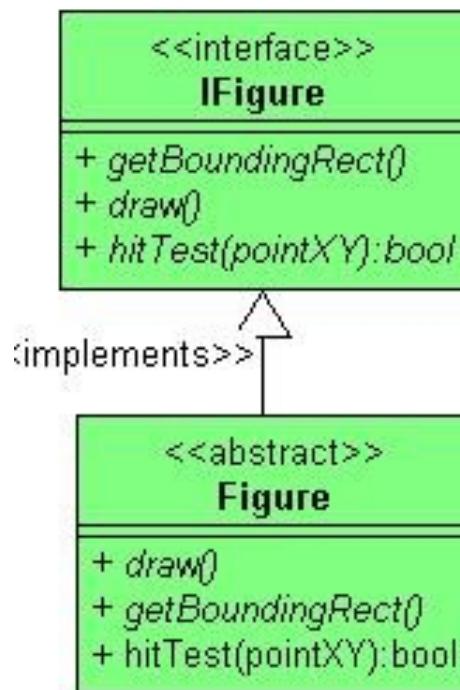
[+] Erzeugen neuer Subklassen ist einfacher

[-] Vielfach ist Ableitung nicht der richtige Designweg



# Interfaces

- Interfaces = Beschreibung, welche Methoden Klassen anbieten
- „Vertrag“ zwischen Aufrufer und Aufrufendem
- Entkopplung von Realisierung und Spezifikation





# Interfaces

---

## Vor- und Nachteile

**[+] Wiederverwendbarkeit einer Klasse ist höher, wenn andere Klassen sich nicht auf spezielle Implementierungen oder Implementierungsdetails verlassen**

**[+] konkrete Implementierung kann einfach ausgetauscht werden, ohne andere Objekte zu beeinträchtigen.**

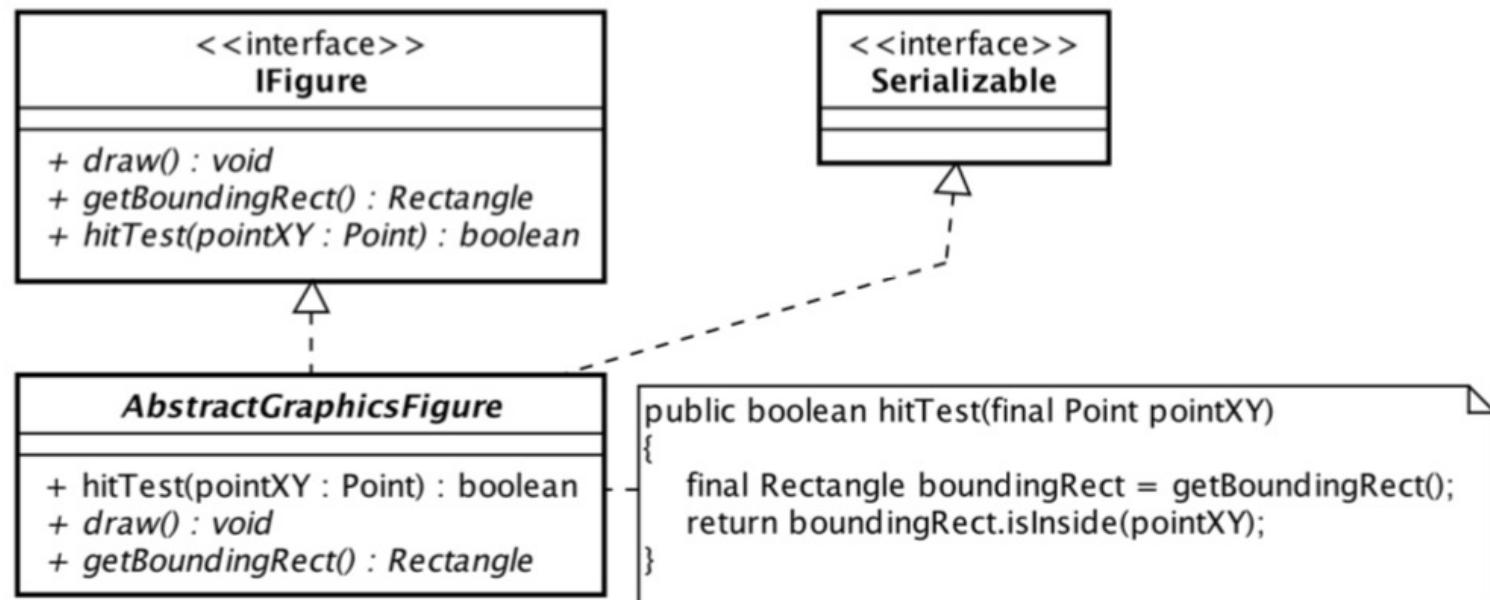
**[+] erleichtert den Überblick über die angebotenen Methoden, da diese in Interfaces zentral definiert werden.**

**[o] kein Zugriff auf den Konstruktor**



## Interface + abstrakte Basisklasse

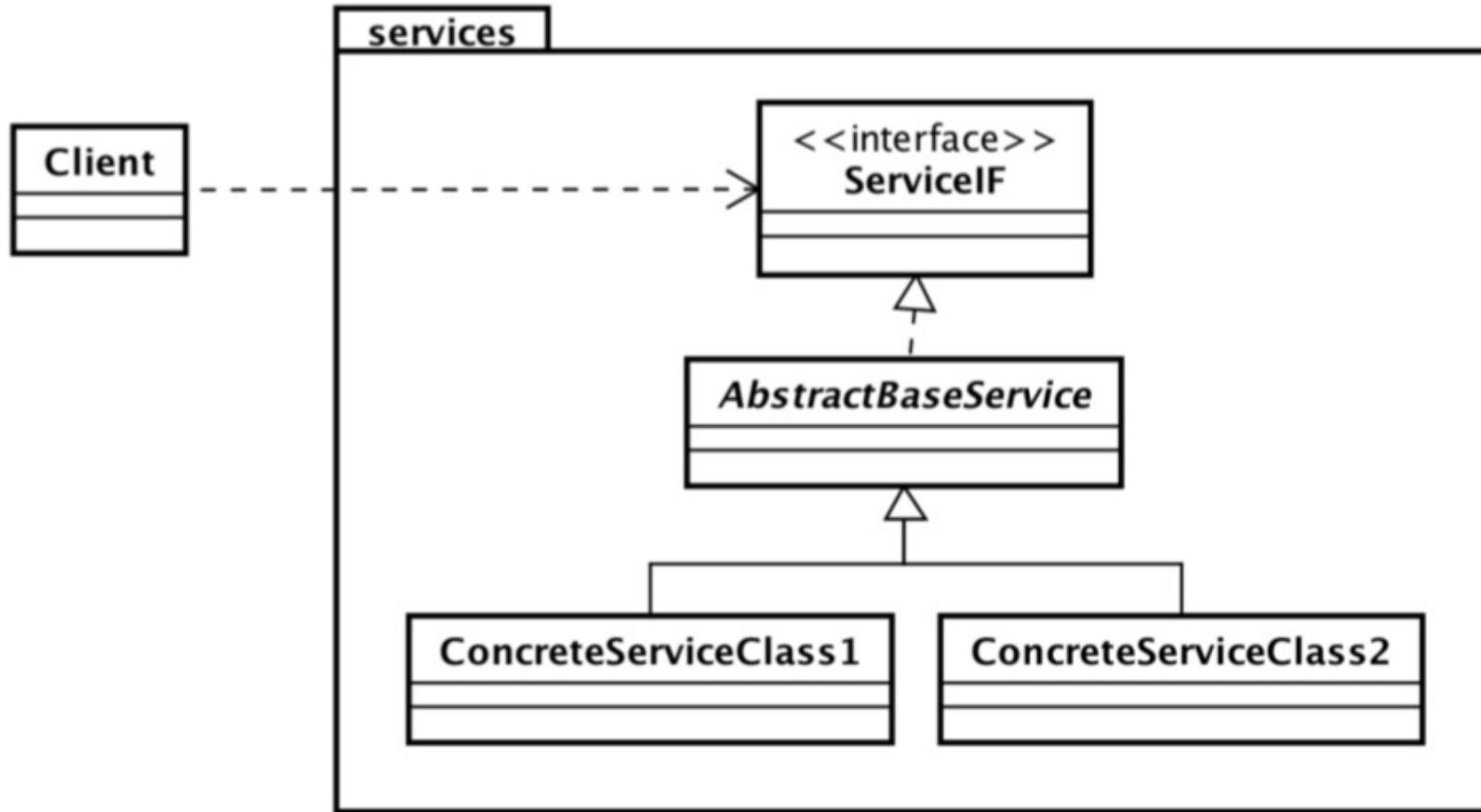
- Techniken Interface und abstrakte Basisklasse gut kombinierbar
- Vereint die Vorteile beider Techniken
  - Trennung von Spezifikation und Realisierung (Interface)
  - Sinnvolle Vorgabe von Basisfunktionalität (abstrakte Basisklasse)
  - Subklassen bringen an speziellen, gewünschten Stellen ihre Realisierungen ein





## Interface + abstrakte Basisklasse

- Techniken Interface und abstrakte Basisklasse gut kombinierbar





## Vor- und Nachteile

- [+] Kapselung der Funktionalität über Interfaces sowie eine sinnvolle Vorgabe von Basisfunktionalität durch eine abstrakte Basisklasse.
- [+] Subklassen müssen nur noch an speziellen, gewünschten Stellen ihre Erweiterungen einbringen.
- [+] konkrete Implementierung kann einfach ausgetauscht werden, ohne andere Objekte zu beeinträchtigen.
- [+] ist insbesondere für Frameworks sinnvoll und empfehlenswert. Ein schönes Beispiel ist das Collections-Framework aus dem JDK, wo dieses Schema sauber umgesetzt wird.
- [o] erfordert nur einen geringen Mehraufwand in der Implementierung.
- [o] Nachvollziehbarkeit kann durch die Indirektionen und Abstraktionen leiden.



## Part 2: Erzeugungsmuster



# Erzeugungsmuster im Überblick

---



**Erzeugungsmuster dienen dazu, einen potenziell aufwendigen Konstruktionsprozess von Objekten zu vereinfachen:**

- Wenn **Methoden** statt **Konstruktoren** zum **Erzeugen von Objekten** verwendet werden, so entspricht dies den Ideen der Muster **ERZEUGUNGSMETHODE** bzw. **FABRIKMETHODE**.
- Auch mithilfe des Musters **ERBAUER** kann man die **Objektkonstruktion** einfacher gestalten.
- Durch ein **SINGLETON** wird sichergestellt, dass **lediglich eine Instanz** einer Klasse erzeugt und ein definierter Zugriffspunkt darauf angeboten wird.
- Mit einem **PROTOTYP** können **Objekte** leicht aus einem definierten Mustersatz **durch Kopie** und Parametrierung **erstellt** werden.



# **Erzeugungsmethode (Creation Method)**

---

# Motivation und Kurzbeschreibung Erzeugungsmethode

---



- Man kann sich die **ERZEUGUNGSMETHODE** etwa wie eine **Bestellung in einem Restaurant** vorstellen. Man sagt, welches Gericht man essen möchte, nennt jedoch nicht jede Zutat. Auch die Art der Zubereitung bleibt verborgen.
- Wenn der **Konstruktionsprozess komplex** oder **fehleranfällig** ist, weil beispielsweise **viele Parameter** übergeben werden müssen, so wird eine Erzeugung durch einen **Konstruktoraufruf** von außen **vermieden**.
- Stattdessen wird die **Klasse selbst für die Objekterzeugung verantwortlich**. Zur Objektkonstruktion wird eine spezielle, statische **Konstruktionsmethode**, die **Erzeugungsmethode**, bereitgestellt.
- Um **Objekterzeugungen** von außen ohne Aufruf der Erzeugungsmethode sicher zu **verhindern**, dürfen lediglich **private Konstruktoren** definiert werden.

## Beispiel



```
DrawingPad(final String title, final boolean runAsApplet,  
           final JApplet applet, final ConsoleParams appParams)
```



```
new DrawingPad("Application", false, null, consoleParams);  
new DrawingPad("Applet", true, applet, null);
```

# Beispiel



- Trick 1: Explaining Variables `RUN_AS_APPLET`, `NO_APP_PARAMS`, `NO_APPLET`, ...
- Trick 2: Zwei Methoden zur Konstruktion

```
private static final boolean RUN_AS_APPLET = true;

public static DrawingPad createAsApplet(final String title,
                                         final JApplet applet)
{
    final ConsoleParams NO_APP_PARAMS = null;
    return new DrawingPad(title, RUN_AS_APPLET, applet, NO_APP_PARAMS);
}

public static DrawingPad createAsApplication(final String title,
                                             final ConsoleParams appParams)
{
    final JApplet NO_APPLET = null;
    return new DrawingPad(title, !RUN_AS_APPLET, NO_APPLET, appParams);
}
```

## Bewertung Erzeugungsmethode



+ **Lesbarkeit** – Die Details des Konstruktionsprozesses werden versteckt => mehr Lesbarkeit

1. Die **Signatur** einer Erzeugungsmethode ist in der Regel **kürzer** als die Parameterliste des Konstruktors, weil lediglich einige der Parameter des Konstruktors entgegengenommen und die restlichen beim Aufruf des Konstruktors mit Defaultwerten belegt werden.
2. Zudem kann man für eine **Erzeugungsmethode** einen **sprechenden Namen** wählen, der die erzeugten Objekte besser charakterisiert als der Konstruktor mit dem Namen der Klasse.

+ **Konstruktionssicherheit** – Man kann garantieren, dass **vollständig** und **korrekt initialisierte Objekte erzeugt** werden. **Parameterprüfung** zu Beginn einer Erzeugungsmethode sichert Instanzerzeugung ab. Es ist **leichter, angemessen auf Fehlersituationen zu reagieren**. Weiterhin kann man eine Art Transaktion bei der Objekterzeugung erreichen: Ein Objekt wird entweder vollständig erzeugt oder es wird im Fehlerfall kein Objekt, sondern der Wert null, ein **NULLOBJEKT** oder **Optional.empty** zurückgeliefert.

o **Mehraufwand** – Es ist minimal mehr Sourcecode erforderlich.



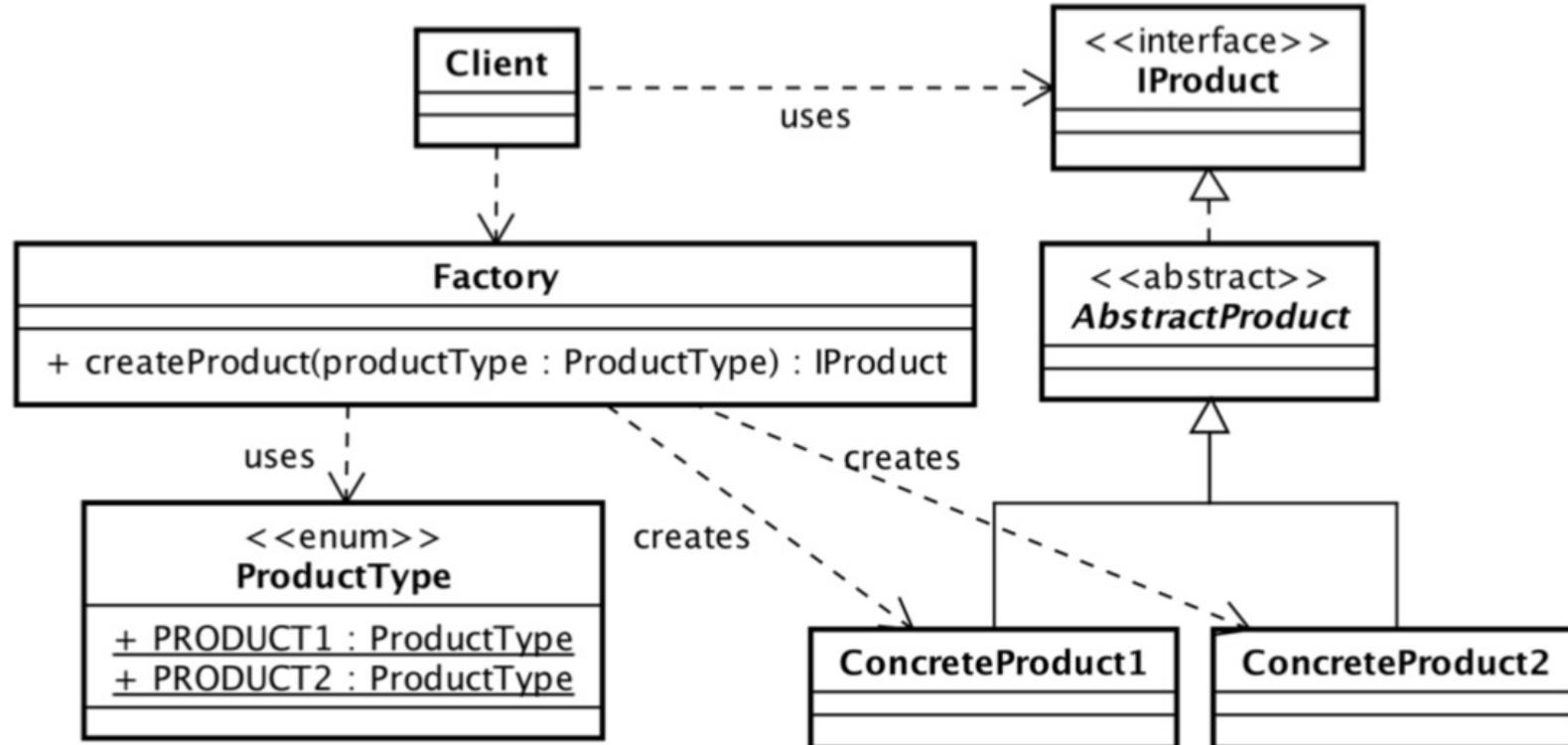
# Fabrikmethode (Factory Method)

## Motivation und Kurzbeschreibung Fabrikmethode

---

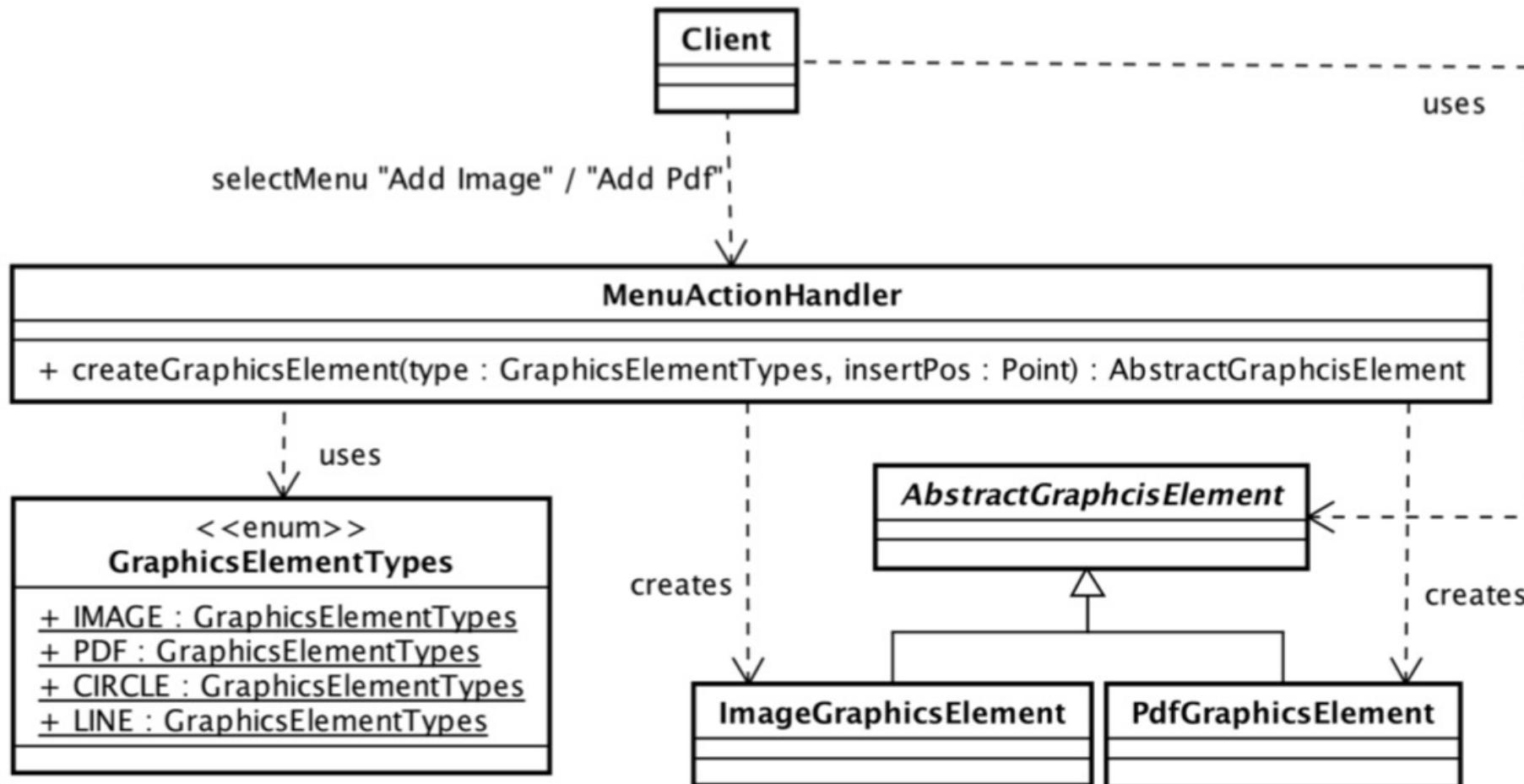


- Die Idee hinter dem Muster **FABRIKMETHODE** ist ähnlich einer **Produktion** in einer realen **Fabrik**, die aufgrund einer **Bestellung** gewünschte **Dinge produzieren** kann.
  - Bei einer solchen **Bestellung** sind **lediglich** die **Artikelnummern** oder **Bezeichnungen** der Teile, nicht aber die konkreten Ausprägungen und Realisierungen bekannt.
  - Es findet demnach **analog** zum Muster **ERZEUGUNGSMETHODE** eine **Kapselung** der **Objekterzeugung** statt: Objekte werden **nicht direkt per Konstruktoraufruf** erzeugt, sondern dieser Vorgang wird an eine spezielle Methode oder ein spezielles Objekt, eine sogenannte Fabrik, delegiert. Die dort definierten Fabrikmethoden erzeugen Objekte verschiedenen Typs.
  - Der Unterschied zum Muster **ERZEUGUNGSMETHODE**, das den Konstruktionsprozess einer speziellen Klasse vereinfacht, liegt darin, dass beim Muster **FABRIKMETHODE** eine andere Methode / Klasse, die Fabrik, die Konstruktion von Objekten eines gewünschten Typs durchführt.
-



- Es wird eine Fabrikkklasse mit einer Methode `createProduct(ProductType)` zum Erzeugen von Objekten definiert. Dort wird anhand eines Parameters entschieden, welcher konkrete Typ erzeugt wird.

# Anwendungsbeispiel



# Anwendungsbeispiel

---



```
public class SimpleProductFactory
{
    public static Product createProduct(String name)
    {
        switch (name)
        {
            case "book":
                return new Book();
            case "pizza":
                return new Pizza();
            case "car":
                return new Car();
            default:
                throw new RuntimeException("No such product " + name);
        }
    }

    static interface Product {}

    static class Book implements Product {}

    static class Pizza implements Product {}

    static class Car implements Product {}
}
```

## Bewertung Fabrikmethode



- + **Lesbarkeit** – Die Details des Konstruktionsprozesses werden versteckt => mehr Lesbarkeit
- + **Abstraktion** – Der konkret erzeugte Objekttyp kann vor dem Aufrufer versteckt werden, indem lediglich eine Referenz auf ein Interface oder eine abstrakte Klasse des erzeugten Objekttyps zurückgegeben wird. Stärkere Kapselung und lose Kopplung.
- + **Kapselung** – Die Kapselung verstärkt sich: Ein Klient nutzt lediglich eine Schnittstelle zur Erzeugung, wodurch die konkrete Realisierung einer Fabrikklasse ausgetauscht werden kann.
- + **Konstruktionssicherheit** – Durch Konsistenzprüfung möglich können vollständig initialisierte Objekte erzeugt / garantiert werden bzw. Fehler entsprechend korrigiert oder behandelt werden.
- o **Mehraufwand** – Es entsteht ein wenig mehr Sourcecode.
- o **Mehr Komplexität** – Geringfügig mehr Komplexität durch die zu realisierenden Klassen



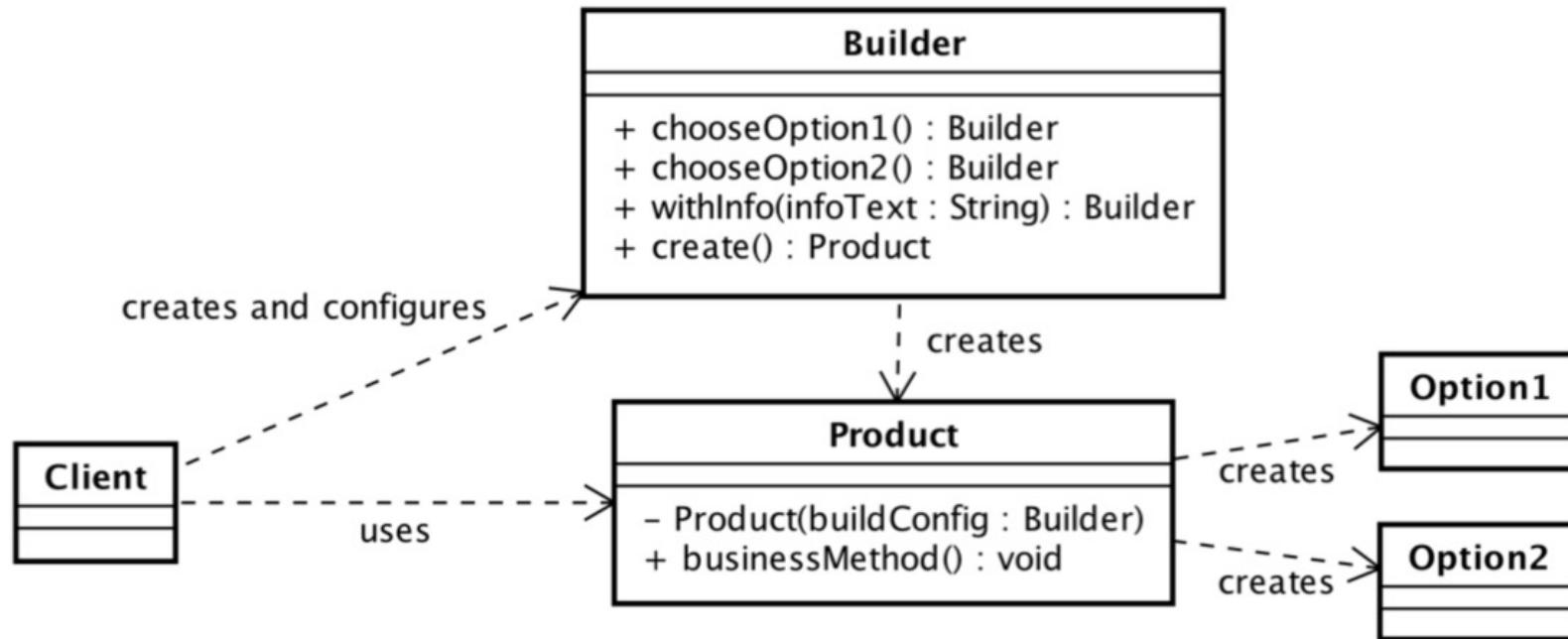
# Erbauer (Builder)

## Motivation und Kurzbeschreibung Erbauer

---



- Das Vorgehen beim ERBAUER-Muster lässt sich mit der **Bestellung** einer **Pizza** oder eines **Autos** vergleichen. Auf einer **Bestellliste** kreuzt man aus vielen verschiedenen **Bestandteilen** und **Extras** genau diejenigen an, die man bekommen möchte. Man gibt also eine **spezielle Bestellung** auf und **konfiguriert** sich damit sein Objekt mit den gewünschten **Eigenschaften**. Erst nach Abschluss der Auswahl wird mit der eigentlichen Herstellung begonnen.
- Das ERBAUER-Muster kann immer dann **sinnvoll** eingesetzt werden, wenn ein zu erzeugendes Objekt **viele optionale** oder **komplex zu konstruierende Bestandteile** besitzt. Auch wenn zur Objektkonstruktion **benötigte Werte iterativ** ermittelt werden, lässt sich eine Folge von **set()-Methoden** zur Objektinitialisierung durch den Einsatz des ERBAUER-Musters **vermeiden**. Dadurch kann man einen **gültigen Objektzustand** leichter **sicherstellen**.
- Im Unterschied zum Muster FABRIKMETHODE, bei dem eine abstrakte Bezeichnung, beispielsweise eine Artikelnummer, das zu erzeugende Produkt beschreibt, liegt der Fokus beim ERBAUER-Musters darauf, die einzelnen Bestandteile eines Objekts auf einfache Weise zu spezifizieren.



- Eine Klasse **Builder** definiert eine **Fabrikmethode** `create()` oder `build()`, die später zur **Konstruktion** des gewünschten **Product**-Objekts aufgerufen werden muss. Um die **Erzeugung** von Objekten mittels **Konstruktoraufruf** durch externe Klassen zu **verhindern**, definiert man den **Konstruktor** der **Product**-Klasse mit der **Sichtbarkeit** `private`.

# Anwendungsbeispiel

---



```
public static final class PizzaBuilder
{
    // Defaultwerte, gelten wenn keine korrespondierende Methode aufgerufen wird
    boolean mitSalami          = false;
    boolean mitExtraSardellen = false;
    String  info               = "";
    Size    size                = Size.MEDIUM;

    public PizzaBuilder mitSalami()
    {
        this.mitSalami = true;
        return this;
    }

    public PizzaBuilder mitExtraSardellen()
    {
        this.mitExtraSardellen = true;
        return this;
    }

    public PizzaBuilder small()
    {
        this.size = Size.SMALL;
        return this;
    }
}
```

...

## Anwendungsbeispiel

---



```
public static void main(final String[] args)
{
    final PizzaBuilder builder = new PizzaBuilder();
    builder.mitExtraSardellen();
    System.out.println("Normale Pizza mit extra Sardellen:\n" + builder.create());

    final PizzaBuilder builder2 = new PizzaBuilder();
    builder2.mitSalami().small().bitteBeachten("Ohne Mais!");
    System.out.println("Kleine Salami-Pizza ohne Mais:\n" + builder2.create());
}
```



# DEMO

**BuilderExample.java**

---

## Bewertung Erbauer



- + **Lesbarkeit und Vereinfachung** – Die **Details des Konstruktionsprozesses** werden **versteckt**, was für **mehr Lesbarkeit** sorgt: Es werden **lediglich gewünschte Eigenschaften spezifiziert**. Die eigentliche Konstruktion wird intern geregelt und bleibt verborgen. **Sprechende Methodennamen**, um die Eigenschaften des zu erzeugenden Objekts besser beschreiben als Parameternamen
- + **Konstruktionssicherheit** – Es lassen sich **vollständig und korrekt initialisierte Objekte** erzeugen: Die **Parametrierung** erfolgt ausschließlich über **Methoden**, die die korrespondierenden Attribute setzen und später als Eingabewerte für die Konstruktion dienen.
- + **Unterstützung optionaler Attribute** – Der **Umgang** mit vielen **optionalen Attributen** wird **erleichtert**: Defaultwerte müssen nicht explizit gesetzt bzw. im Konstruktor übergeben werden.
- **Mehr Komplexität** – Mehr Komplexität und Sourcecode, da gewünschte Werteberechnungen von Attributen über den Aufruf von Methoden spezifiziert werden. Der für Aufrufer verborgene Konstruktionsprozess sorgt für mehr Komplexität innerhalb der Klassen des Musters.

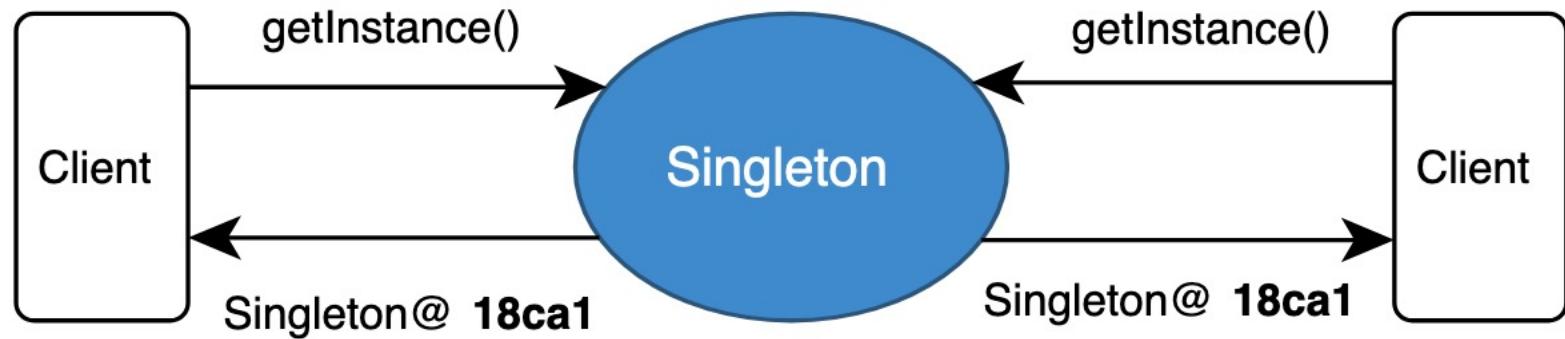


# Singleton

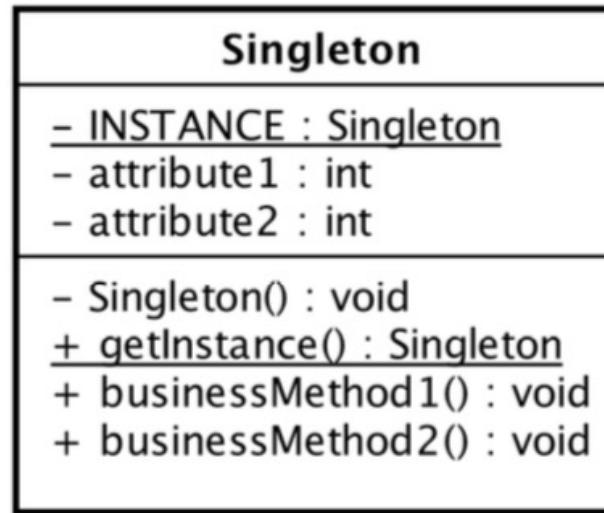
# Motivation und Kurzbeschreibung Singleton



- **Sicherstellen**, dass **höchstens eine Instanz** einer bestimmten **Klasse** erzeugt werden kann
- Außerdem wird dadurch ein **globaler Zugriffspunkt** auf die Instanz bereitgestellt.



- Beispiele für Einsatzgebiete sind jede Art von **zentraler Registrierung**, etwa ein zentraler Plugin-Manager.
- **Fabrikklassen** werden häufig gemäß dem **SINGLETON-Muster** implementiert, da man Klienten nicht zusätzlich die Erzeugung von Fabrikklassen auferlegen möchte.



- Um eine **Objekterzeugung außerhalb** der Klasse zu **verhindern**, verwenden wir einen **privaten Konstruktor**.
- Zur Speicherung der **einzigsten Instanz** wird ein **statisches Attribut INSTANCE** verwendet. Der **Zugriff** auf dieses erfolgt **ausschließlich** über eine statische **Zugriffsmethode**, typischerweise **getInstance()** genannt.

## Bad Singleton



```
public final class BadSingleton
```

```
{
```

```
    private static BadSingleton INSTANCE = null;
```

```
    private int attribute1;
```

```
    private int attribute2;
```

```
// ACHTUNG: SCHLECHT !!!
```

```
    public static BadSingleton getInstance()
```

```
{
```

```
        if (INSTANCE == null)
```

```
        {
```

```
            INSTANCE = new BadSingleton();
```

```
        }
```

```
        return INSTANCE;
```

```
}
```

```
    private BadSingleton()
```

```
{
```

```
}
```

```
    public void businessMethod1() { /* ... */ }
```

```
    public void businessMethod2() { /* ... */ }
```

```
}
```

## Better Singleton

---



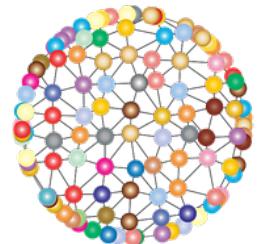
```
public final class Singleton
{
    private static final Singleton INSTANCE = new Singleton();

    private int attribute1;
    private int attribute2;

    public static Singleton getInstance()
    {
        return INSTANCE;
    }

    private Singleton()
    {
    }

    public void businessMethod1() { /* ... */ }
    public void businessMethod2() { /* ... */ }
}
```



**Geht es noch kompakter?**

# Singleton mit Enum!?

---



```
public enum SingletonEnum
{
    INSTANCE;

    private int attribute1;
    private int attribute2;

    public void businessMethod1() { /* ... */ }
    public void businessMethod2() { /* ... */ }
}
```

## Bewertung Singleton



- + **Zentraler Zugriffspunkt** – Es gibt einen zentralen Zugriffspunkt auf benötigte Funktionalität, der eine sichere Initialisierung erlaubt.
- + **Strukturierung** – Die Zugriffe werden **strukturiert** und **Implementierungsdetails** lassen sich **verstecken**.
  - o **Realisierungsprobleme** – Der Einsatz ist komplizierter, als man meint: Intuitive Lösungen unter Verwendung von Lazy Initialization führen zu Problemen in Multithreading-Umgebungen. Schwer verständliche Abhilfen wie Double Checked Locking.
- **Keine Eindeutigkeitsgarantie** – Der Einsatz ist problematisch, wenn man mehrere ClassLoader verwendet. Es gibt dann keine Eindeutigkeitsgarantie mehr und es sind somit mehrere Instanzen eines Singletons möglich – pro ClassLoader eine.



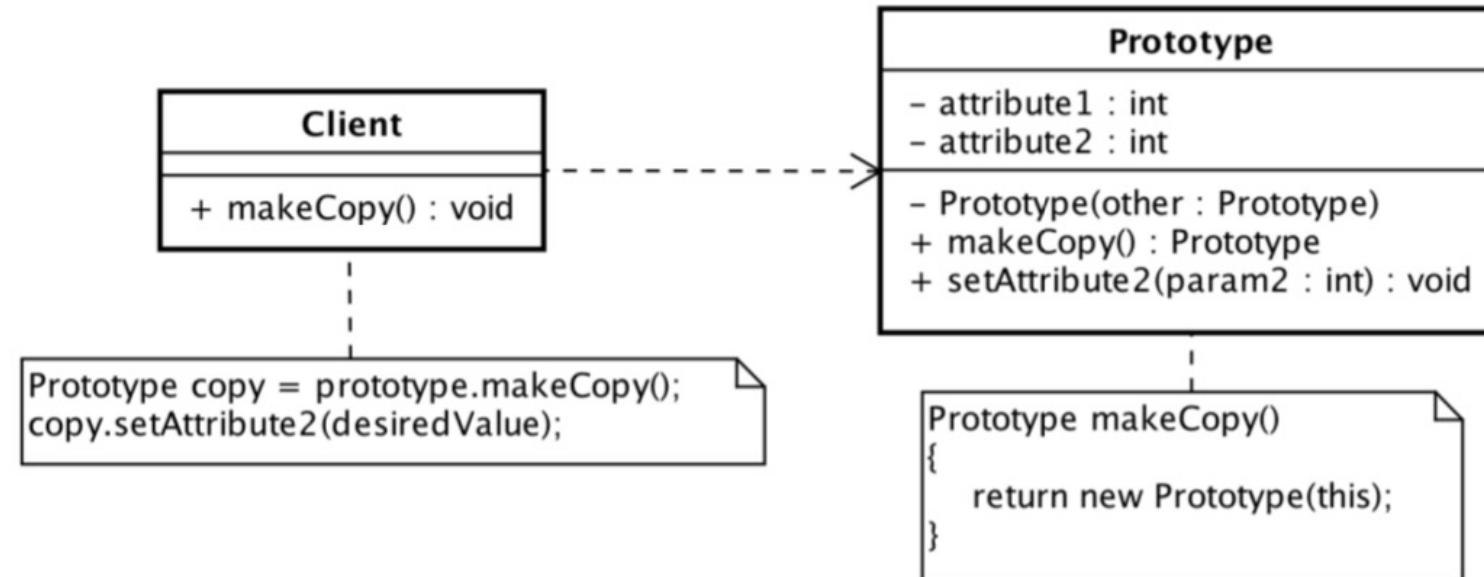
# Prototyp (Prototype)

# Motivation und Kurzbeschreibung Prototyp

---



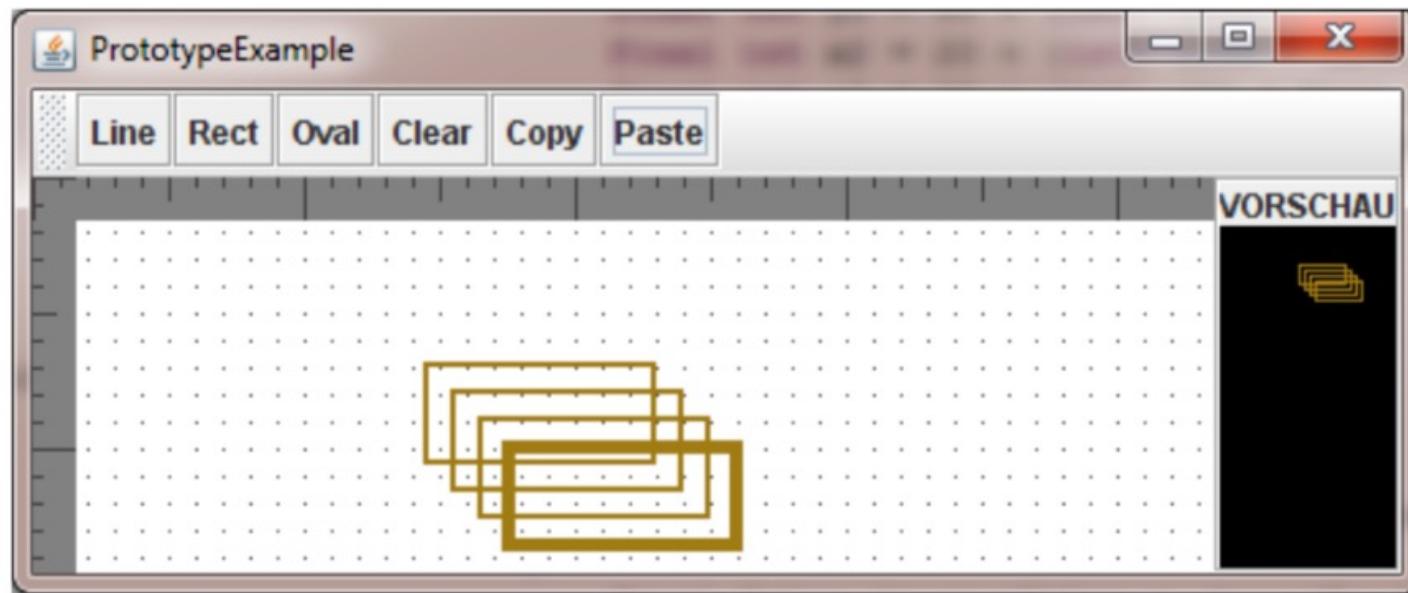
- Das Vorgehen beim PROTOTYP-Muster besteht darin, basierend auf speziellen **Vorlagenobjekten** neue Objekte zu generieren.
- Man nutzt dieses Muster, wenn **Instanzen** einer Klasse einen **großen gemeinsamen Grundstock** an gleichen Werten ihrer Attribute haben und durch **wenige Modifikationen** gebrauchsfertig gemacht werden können.
- Außerdem kann der Einsatz zur **Performance-Steigerung** und zur **ressourcenschonenden Objekterzeugung** verwendet werden, wenn diese per Konstruktor zeitaufwendig ist (z. B. durch Datenbankzugriffe zum Ermitteln der Werte der Attribute) und somit ein **Kopieren** der Ergebnisse und anschließendes **Parametrieren** günstiger ist.
- Dieses Muster besitzt **zwei Varianten**: Bei der **statischen** Variante dienen **eine oder mehrere Kopiervorlagen (Prototypen)** als Basis für neue Objekte. Bei der **dynamischen** Variante kann sogar der **Typ der Kopiervorlage** zur Laufzeit unterschiedlich sein.



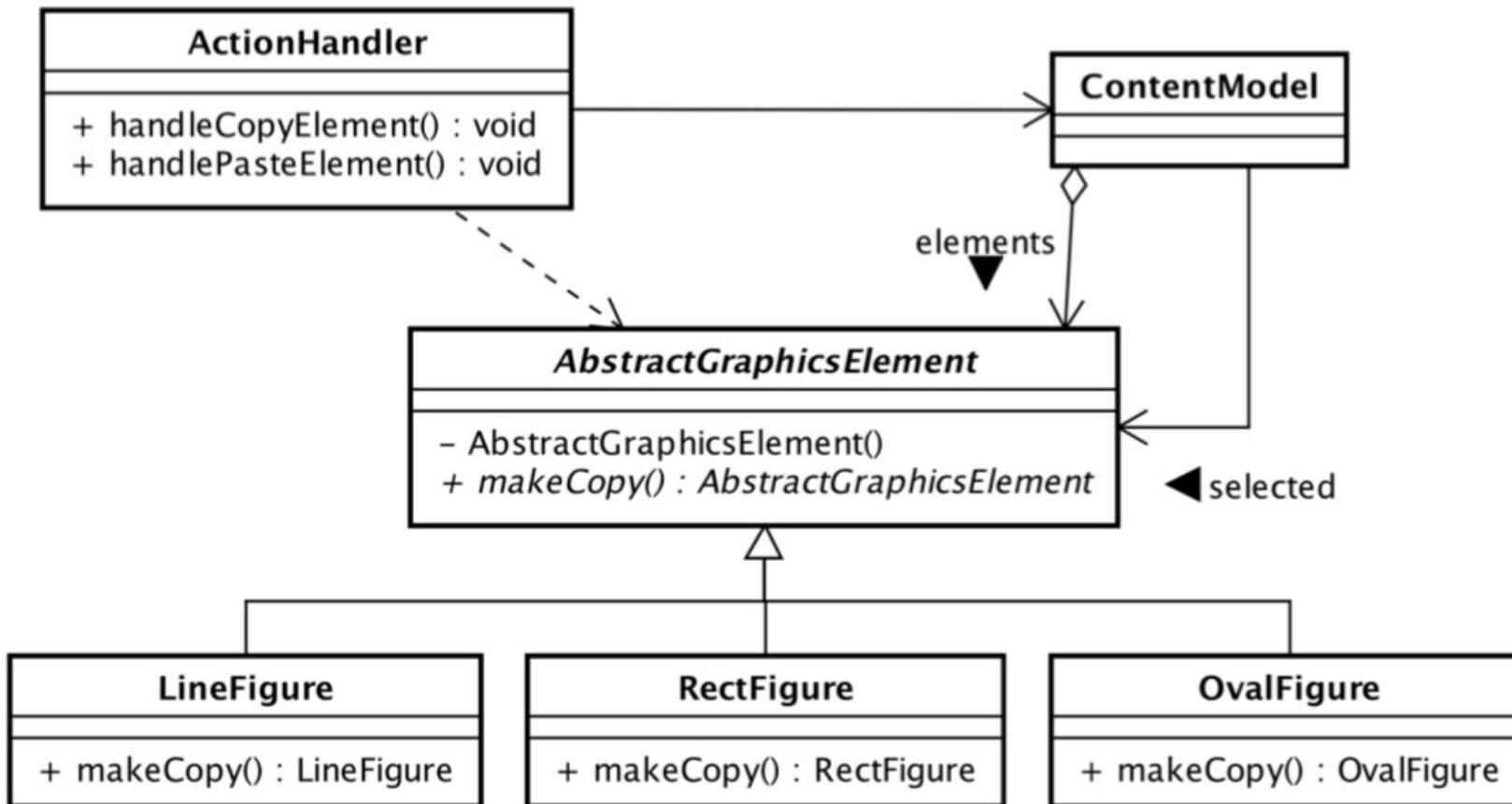
Es existiert ein **Objekt** vom Typ **Prototype**, der eine **makeCopy()**-**Methode** anbietet, um **sich selbst zu kopieren**. Damit kann **jedes Objekt** dieses Typs als **Kopiervorlage** agieren. Hier wird bewusst nicht der Name `clone()` verwendet, um eine klare Abgrenzung zum `Cloneable`-Interface aus dem JDK zu gewährleisten.

# Anwendungsbeispiel

---



# Anwendungsbeispiel



## Anwendungsbeispiel

---



```
public void handleCopyElement()
{
    final AbstractGraphicsElement selected = contentModel.getSelectedElement();

    if (selected != null)
    {
        insertPos = selected.getPosition();
        clipboardElement = selected.makeCopy();
    }
    else
    {
        throw new IllegalStateException("Copy must only be activated" +
                                         "if an element is selected!");
    }
}

@Override
public AbstractGraphicsElement makeCopy()
{
    return new LineFigure(getDrawingColor(), getPosition().x, getPosition().y, x2, y2);
}
```

# Anwendungsbeispiel

---



```
public void handlePasteElement()
{
    if (clipboardElement != null)
    {
        // Kaskadierende Position
        increaseImageInsertPos();

        clipboardElement.setPosition(insertPos.x, insertPos.y);
        contentModel.addElement(clipboardElement);

        // hier nochmal kopieren, damit wir mehrfach pasten k?nnen
        // und neue Elemente ins Modell aufnehmen
        clipboardElement = clipboardElement.makeCopy();
    }
    else
    {
        throw new IllegalStateException("Paste must only be activated" +
            "if an element is selected!");
    }
}
```



# DEMO

**PrototypeExample.java**

---

## Bewertung Prototype



- + **Vereinfachung** – Die **Objektvervielfältigung** und -konstruktion wird **vereinfacht**. Statt neue Objekte mit umfangreicher Parametrierung neu erzeugen zu müssen, wird hier auf eine **Vorlage** zurückgegriffen, die als Basis dient und **lediglich einige Modifikationen** zum Einsatz benötigt.
- + **Flexibilität** – Bei der **dynamischen Variante** ist das **Kopieren** von zur **Kompilierzeit** unbekannten Typen möglich
- + **Nachvollziehbarkeit** – Der Ablauf ist im Gegensatz zum in die JVM integrierten `clone`-Mechanismus besser nachvollziehbar. In diesem Fall wird explizit eine Implementierung der `makeCopy()`-Methode gefordert. `Cloneable`-Interface gibt es nur implizite Annahmen über die Existenz von `clone()`-Methoden: `Cloneable` ist lediglich ein Marker-Interface
  - o **Fallstrick Referenzsemantik** – Bei Containerklassen ist schwierig zu entscheiden, ob durch die Referenzsemantik Probleme entstehen können.



# Vorbereitung Design Pattern Workshop





---

## Exercises Part 2

<https://github.com/Michaeli71/DesignPatterns.git>





# Part 3: Strukturmuster



# Erzeugungsmuster im Überblick

---



Strukturmuster helfen dabei, Funktionalitäten leichter konfigurierbar oder handhabbar zu machen sowie Verhalten flexibel erweitern und anpassen zu können:

- Eine **FASSADE (FAÇADE)** kapselt und versteckt die Komplexität eines Subsystems.
  - Mit einem **ADAPTER** lassen sich inkompatible Softwarestücke verbinden.
  - Ein **DEKORIERER (DECORATOR)** erlaubt es, Klassen neue Funktionalität hinzuzufügen.
  - Mit dem **KOMPOSITUM (COMPOSITE)** lassen sich Einzelemente und Gruppen aus Baumstrukturen einheitlich behandeln.
-

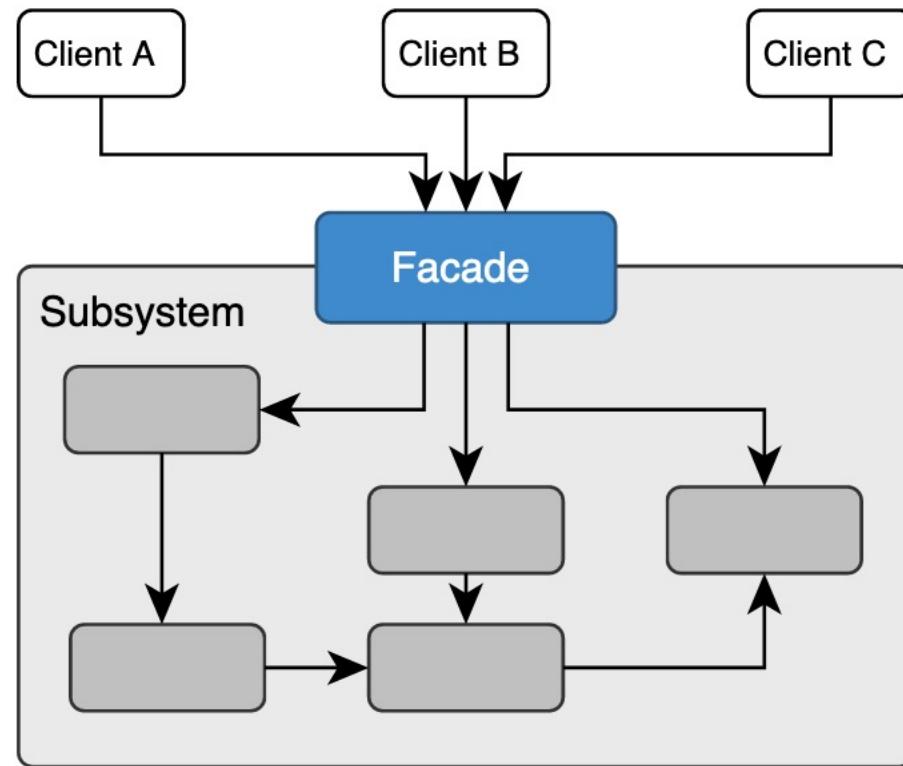


# Fassade (Facade)

# Motivation und Kurzbeschreibung Fassade



- **Komplexität eines Subsystems verbergen**
  - **Zugriff für externe Klassen vereinfachen bzw. steuern.**



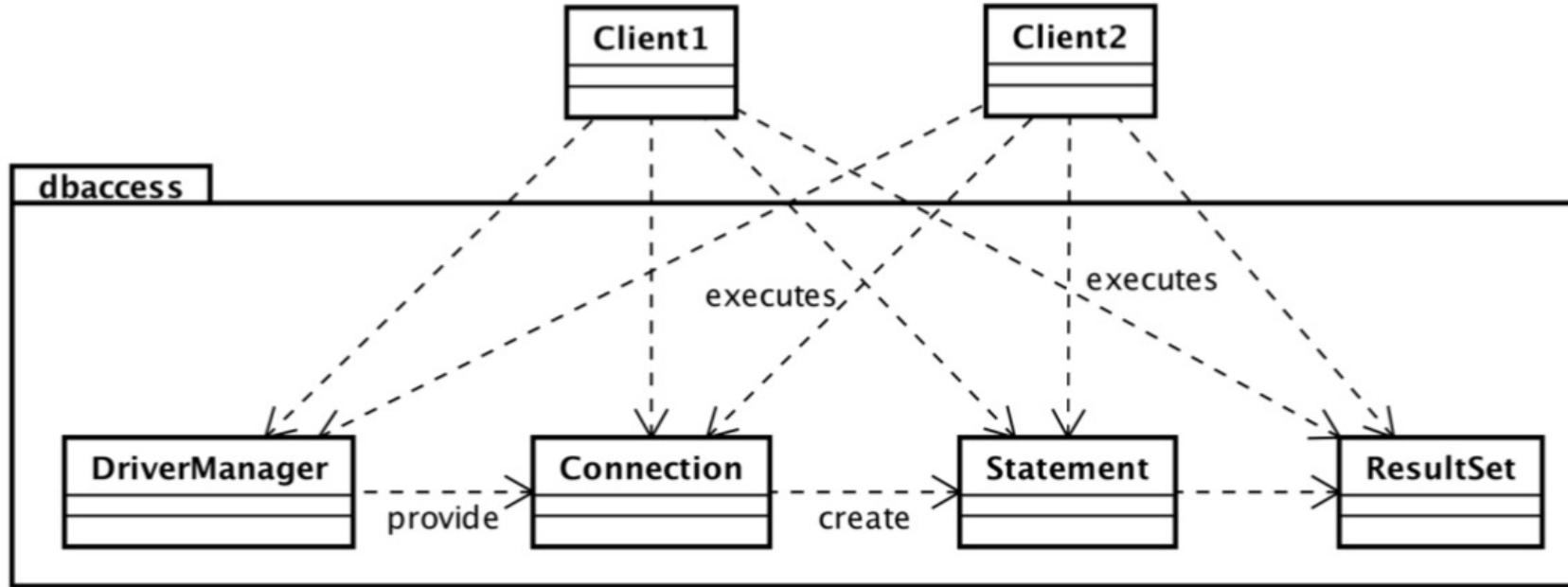
# Motivation und Kurzbeschreibung Fassade

---



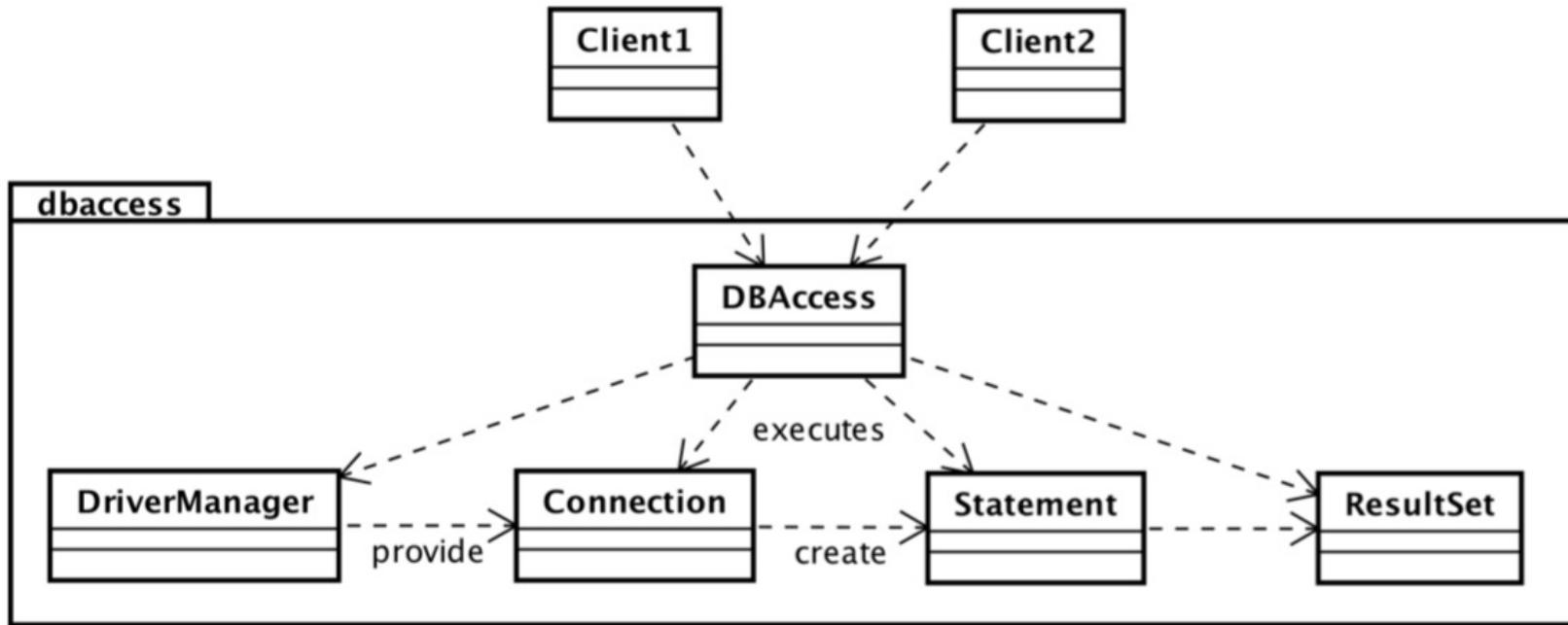
- Eine Fassadenklasse definiert dazu ein »**High-Level**«-Interface, um komplizierte, sehr **feingranulare Interaktionen** und **Beziehungen** zwischen Package-internen und -externen Klassen zu vermeiden.
- **Fassadenobjekt delegiert** die Aufrufe durch Klienten **entsprechend der Zuständigkeit** an spezielle Klassen des **Subsystems**.
- Es herrscht eine **gerichtete Abhängigkeit**: Die Fassadenklasse kennt die Klassen des Subsystems, aber nicht umgekehrt, d. h., keine Klasse des Subsystems kennt die Fassadenklasse.
- Interne Änderungen lassen sich besser kontrollieren und wirken sich weniger stark aus.

# UML -- Ausgangslage



- Betrachten wir ein System mit zwei Klienten `Client1` und `Client2`, die Datenbankzugriffe durchführen wollen. In der Abbildung erkennt man viele Abhängigkeiten von den eingesetzten Klassen. Dies liegt daran, dass die Logik und Steuerung jeweils in den Klienten erfolgt.

# UML -- Lösung

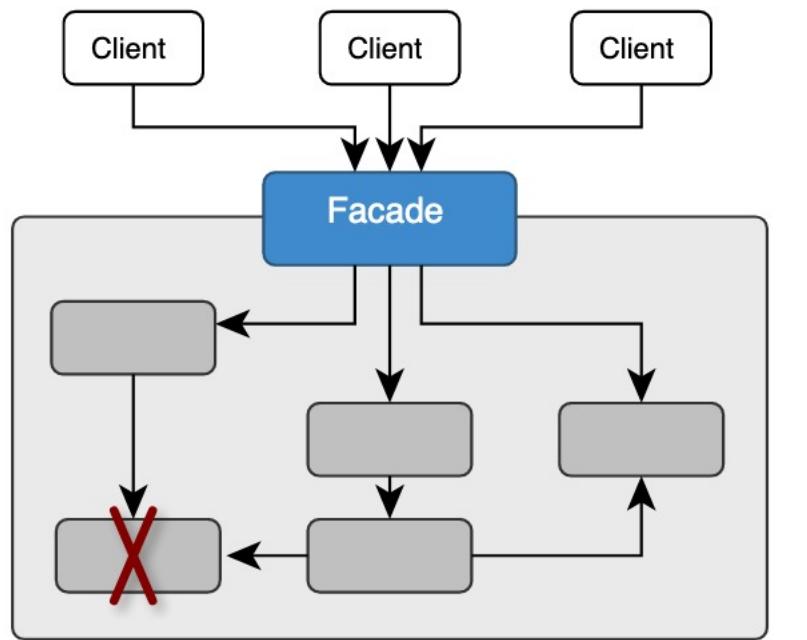


- Führt man eine Fassadenklasse **DBAccess** ein, so lassen sich dadurch die Zugriffe auf die Datenbankzugriffsklassen strukturieren.
- Die Fassadenklasse verbirgt den Verbindungsaufbau und die Komplexität der Datenbankzugriffe vor aufrufenden Klienten.

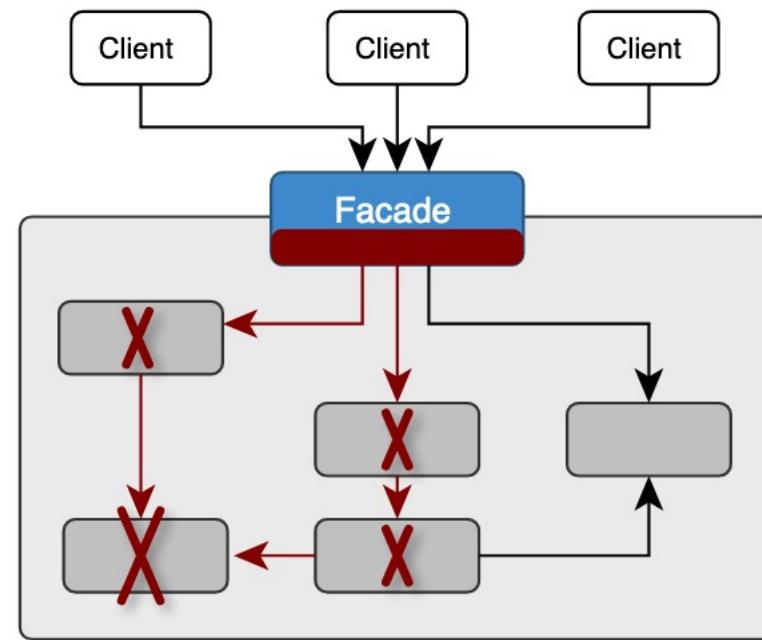
# Änderungsfortplanzung MIT Fassade



- Änderungen im Subsystem pflanzen sich nicht mehr unkontrolliert durch die gesamte Applikation fort, sondern höchstens bis zur Implementierung der Facade.
- Die Schnittstelle der Facade bleibt unverändert.
- Gewonnene Änderungsstabilität und den gesenkten Änderungsaufwand

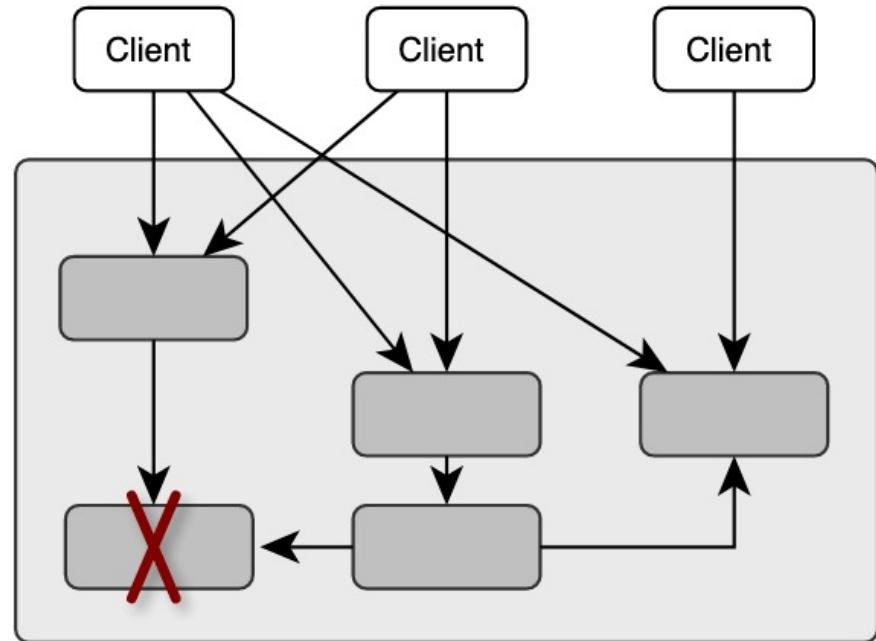


Änderung einer Komponente

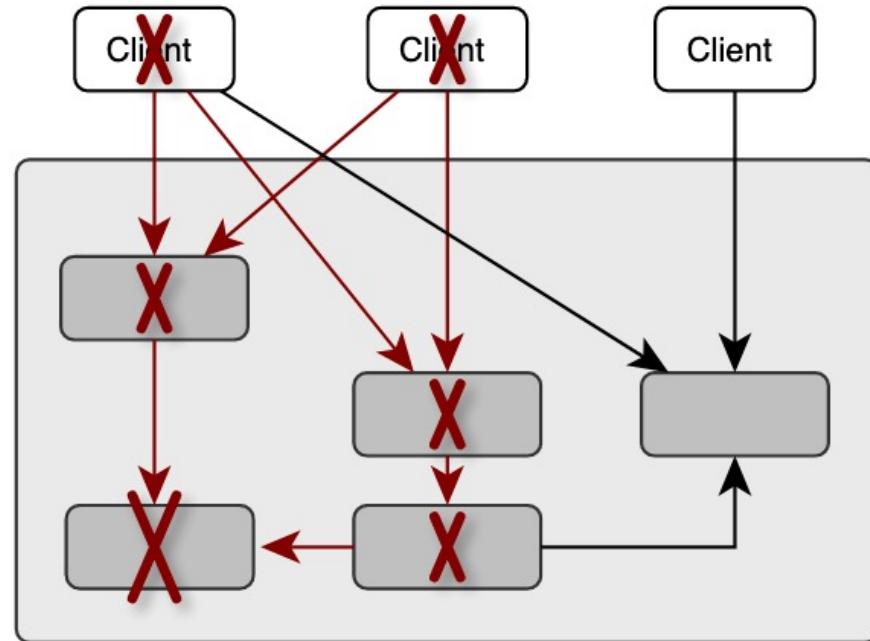


Clients nicht betroffen

# Änderungsfortplanzung OHNE Fassade



Änderung einer Komponente



Clients betroffen!

## Bewertung Fassade



- + **Vereinfachung des APIs** – Der **Zugriff** auf die Funktionalität eines Subsystems fällt **leichter**, weil eine einfachere und **oftmals verständlichere Schnittstelle** zum Zugriff auf dessen Klassen angeboten wird.
- + **Trennung zwischen Klienten und Subsystem** – Die **Zugriffe** werden **strukturiert** und **Implementierungsdetails** lassen sich **verstecken**. Klienten müssen somit **weniger Kenntnis** über die Komponenten des Subsystems haben. Dadurch kommt es zu einer **besseren Trennung und loseren Kopplung** zwischen Klienten und dem Subsystem.
- + **Zentralisierung von Funktionalität und Updates** – Die Steuerung von sogenannten *Cross-Cutting Concerns* erfolgt einheitlich an einer Stelle. Außerdem kann man eine **Fassade** zu einer Art **Transaktionsverwaltung** bei der **Kommunikation** von einem GUI mit einem komplexeren Modell nutzen. **Propagation von Modelländerungen** nicht feingranular, sondern **gebündelt** und gegebenenfalls **gepuffert** durchführen. Dadurch vermeidet man beispielsweise ständiges Neuzeichnen und »Geflacker«. Erst nachdem Änderungen vollständig im Modell verarbeitet wurden, wird das GUI durch die Fassadenklasse über Änderungen informiert und dann aktualisiert.

## Bewertung Fassade

---



- o **Gefahr eines breiten Interface** – Die Kapselung der Funktionalität des Subsystems führt schnell zu einem sehr breiten Interface mit vielen Methoden.
- o **Keine Nutzungsgarantie** – Es ist **nicht gewährleistet**, dass Klienten die **Fassadenklasse** auch **benutzen**. Stattdessen können Klienten **daran vorbei programmieren**, wodurch die zuvor genannten positiven Effekte verloren gehen. Durch »**Schleichwege**« **leidet** die **Wartbarkeit**, da der Sourcecode unübersichtlich und **schlechter nachvollziehbar** wird. Abhilfe: Um derartige »**Fehlverwendungen**« aus anderen Packages zu unterbinden, kann man **Klassen** des Subsystems **Package-private** definieren.
- o **Mehr Indirektionen** – Es kommt zu Weiterleitungen von Methodenaufrufen, *Indirektionen*.

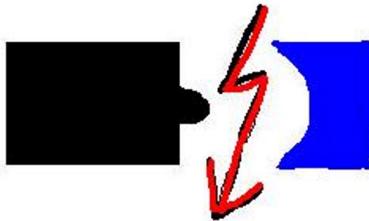


# Adapter

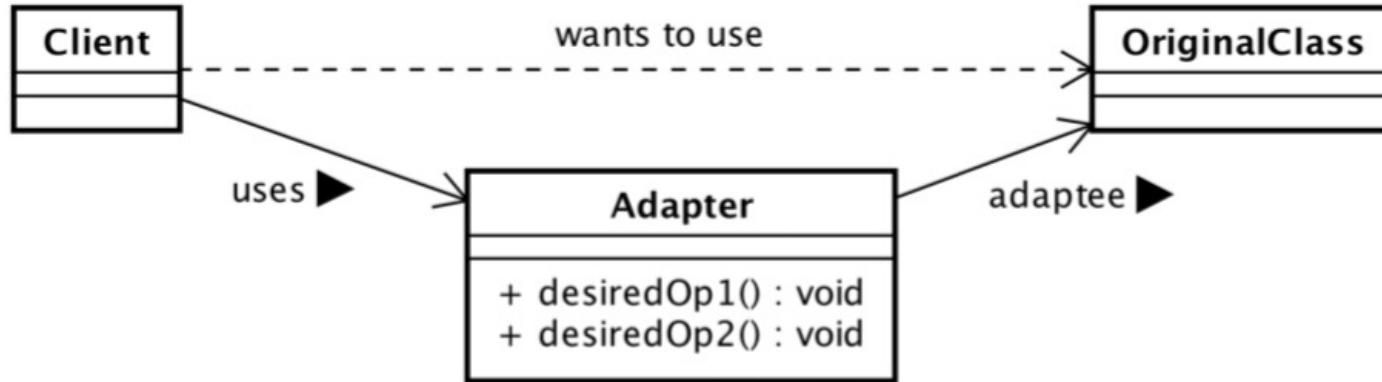
# Motivation und Kurzbeschreibung Adapter



- **ansonsten inkompatible Software kompatibel gemacht werden kann**
- Beispiel von zwei Steckern oder Puzzleteilen motivieren, die nicht ineinander passen.
- Adapterstück, um die nicht passenden Teile zu verbinden.



- Adapter bildet die **Schnittstellen ansonsten inkompatibler Software** aufeinander ab, um diese **miteinander zu verbinden**.
- Der entscheidende Vorteil gegenüber einem direkten Ansprechen einer anderen Klasse ist, dass durch den Einsatz eines Adapters **keine Änderungen** an den **Schnittstellen** der bereits **vorhandenen Implementierungen** nötig sind. Dafür muss allerdings ein neues Softwarestück, der Adapter, entwickelt werden.



- Ein Klient **Client** nutzt ein Objekt vom Typ **Adapter**, um die **Funktionalität** einer vorhandenen Klasse **OriginalClass** mit einer für den Klienten **passenden Schnittstelle verwenden** zu können.

# Anwendungsbeispiel Adapter

---



ListModel	Adapter (Aufruf)	List
getSize()	=>	size()
getElementAt()	=>	get()

# Anwendungsbeispiel Adapter

---



```
private static class ListToAbstractListAdapter extends AbstractListModel<String>
{
    private final List<String> content = new ArrayList<>();

    public ListToAbstractListAdapter(final List<String> content)
    {
        this.content.addAll(content);
    }

    @Override
    public int getSize()
    {
        return content.size();
    }

    @Override
    public String getElementAt(int index)
    {
        return content.get(index);
    }
}
```



# DEMO

**SimpleAdapterExample.java**

---

## Bewertung Adapter



- + **Sicherstellung von Kompatibilität** – Zwei unabhängige Klassen (oder Systeme) werden durch den Adapter kompatibel zueinander gemacht.
- + **Keine Änderungen am bestehenden System** – Die Funktionalität einer anderen Klasse lässt sich nutzen, ohne deren Implementierung ändern zu müssen. Es ist nur die Realisierung des Adapters erforderlich. Dadurch wird das Risiko für Fehler minimiert, da die bisherige (getestete) Funktionalität unverändert bleibt.
  - o **Aufwand durch Delegation** – Es wird Delegation genutzt und es ist manchmal einiges an Codierungsaufwand notwendig.

# Übung Adapter



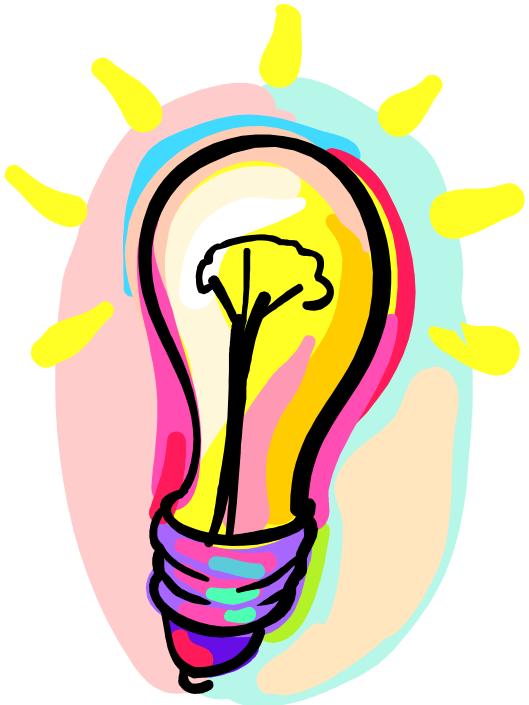
Nutze das Adapter-Pattern, um eine Möglichkeit zu bieten, Arrays mit einem Iterator zu durchlaufen.

```
public class ArrayIteratorAdapter implements Iterator
{
    int currentPosition;
    final Object[] adaptee;

    public ArrayIteratorAdapter(final Object[] adaptee)
    {
        this.adaptee = adaptee;
        this.currentPosition = 0;
    }

    // TODO
    public boolean hasNext()
    public Object next()
    public void remove()
}
```

# Adapter – Beispiellösung



```
public class ArrayIteratorAdapter implements Iterator
{
    int currentPosition;
    final Object[] adaptee;

    public ArrayIteratorAdapter(final Object[] adaptee)
    {
        this.adaptee = adaptee;
        this.currentPosition = 0;
    }

    public boolean hasNext()
    {
        return currentPosition < adaptee.length;
    }

    public Object next()
    {
        final Object next = adaptee[currentPosition];
        currentPosition++;
        return next;
    }

    // Da wir nur einen Wrapper darstellen, dürfen und
    // wollen wir nicht verändern!!!
    public void remove()
    {
        throw new UnsupportedOperationException("Adapter does not " +
            "implement remove!");
    }
}
```

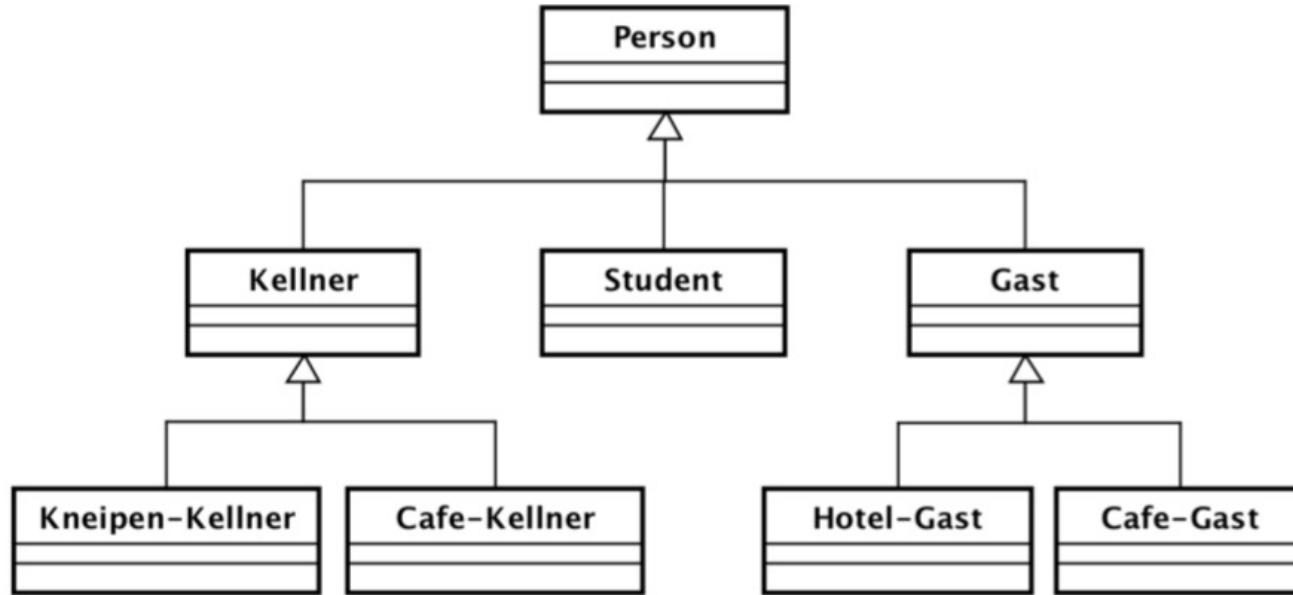


# Dekorierer (Decorator)

# Probleme mit Vererbung



- Selbst beim Einhalten der „is-a“-Eigenschaft kann es zu Problemen kommen:

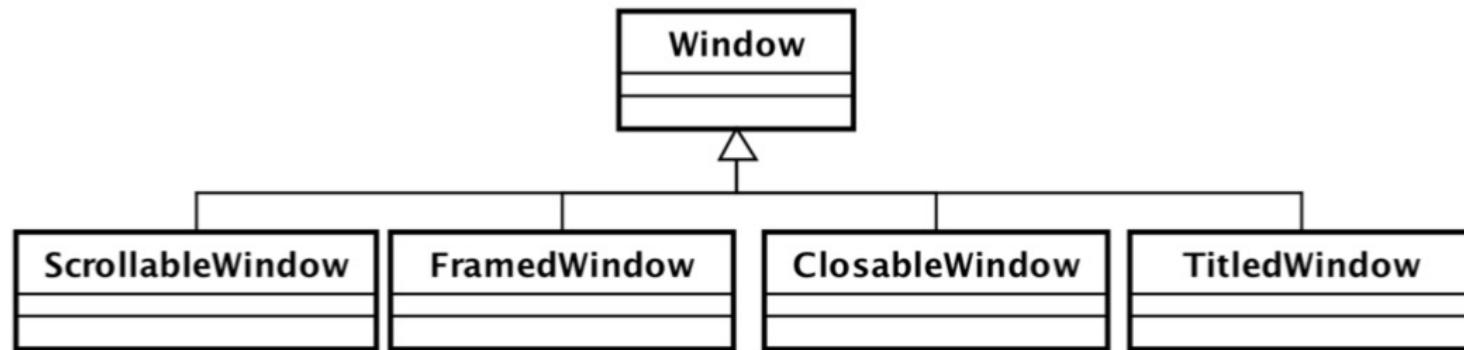


- Sind alles Spezialisierungen
- ABER: Zielen mehr auf eine Rolle/Aufgabe, die zeitweilig ausgeübt wird
  - „is-a-role-played-by“ => Delegation
  - „can-act-like“ => Interface



## Probleme mit Vererbung

- Eigenschaften per Vererbung hinzufügen:

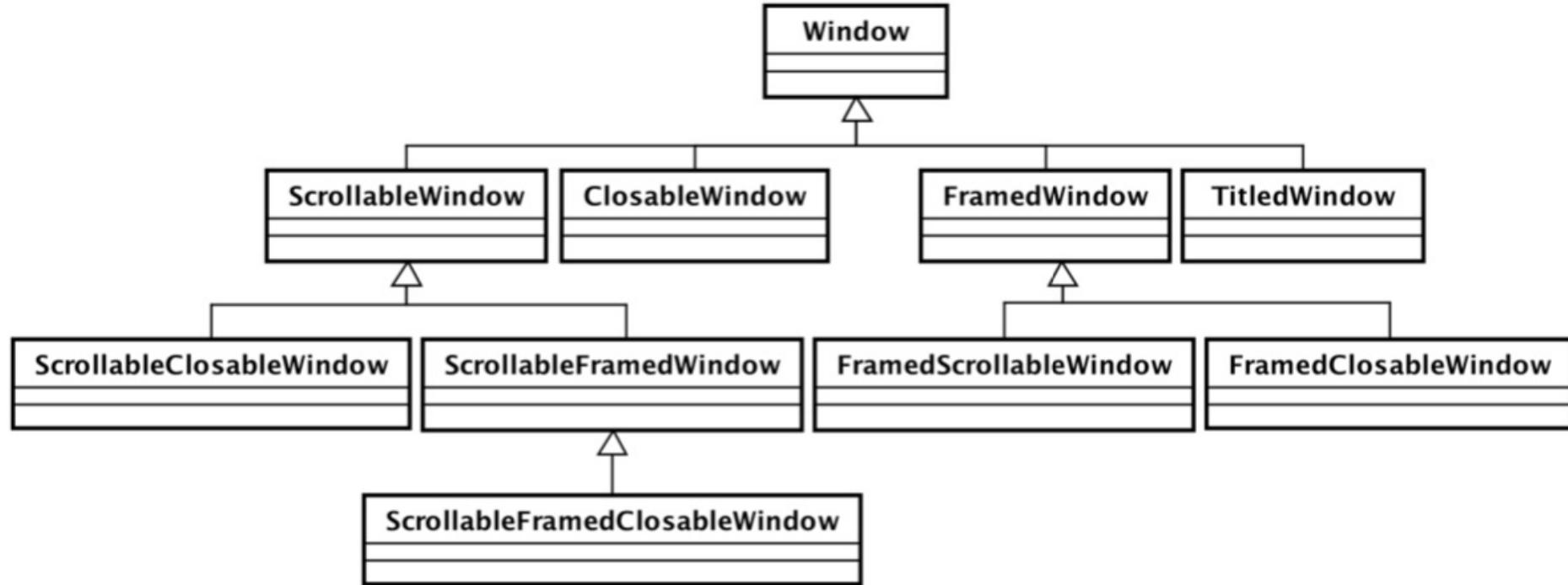


- Sind alles Spezialisierungen
- ABER: Beschreiben mehr eine (boolesche) Eigenschaft
- Problem: Man möchte die Eigenschaften kombinieren  
=> weitere Ableitungen notwendig  
=> Kombinatorische Explosion



# Probleme mit Vererbung

- Kombinatorische Explosion:



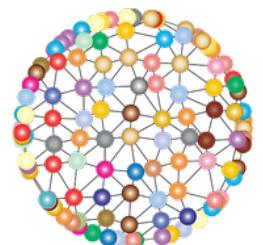
- Gibt es Unterschiede bei verschiedenen Ableitungsreihenfolgen?
- Für orthogonale (voneinander unabhängige) Eigenschaften: => Decorator-Pattern: Füge Funktionalität dynamisch ohne Vererbung hinzu

## Motivation und Kurzbeschreibung Dekorierer

---



- **zusätzliches Verhalten transparent zur Verfügung stellen**
  - **Keine Modifikation der ursprünglichen Klasse**
  - **Dies ist dann praktisch, wenn entweder eine zu erweiternde Klasse nicht als Sourcecode vorliegt oder dieser nicht verändert werden darf.**
  - **Ein erster Gedanke ist häufig, eine Subklasse zu bilden und dort die gewünschten Erweiterungen vorzunehmen. Vererbung hat aber so ihre Tücken**
  - **Dekorierer arbeitet ohne Vererbung**
  - **Kann neue Funktionalität zur Laufzeit, also dynamisch, bereitstellen**
-

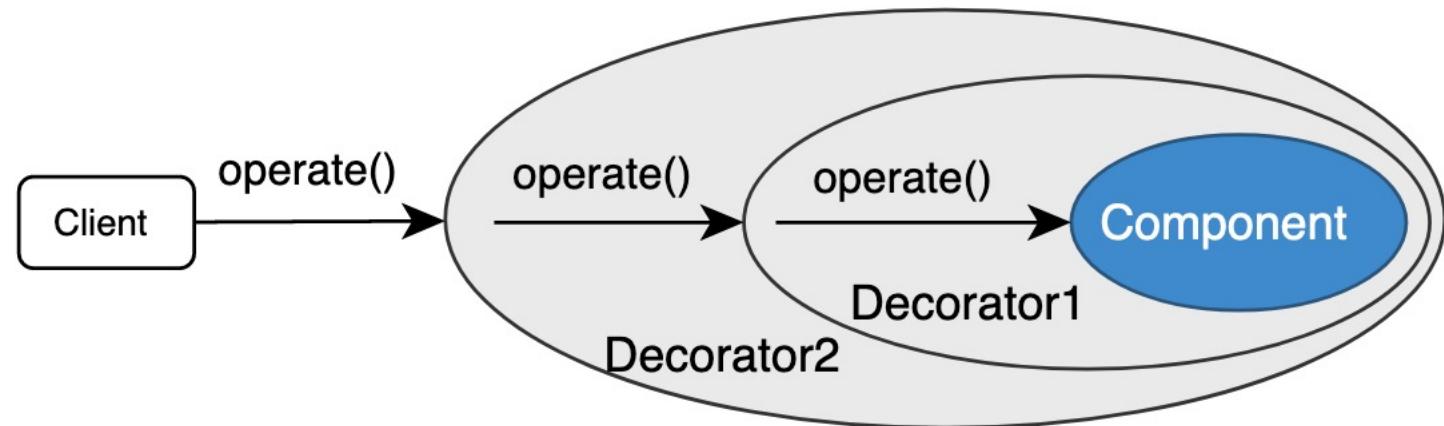


**Unglaublich!**  
**Wie soll das den gehen?**

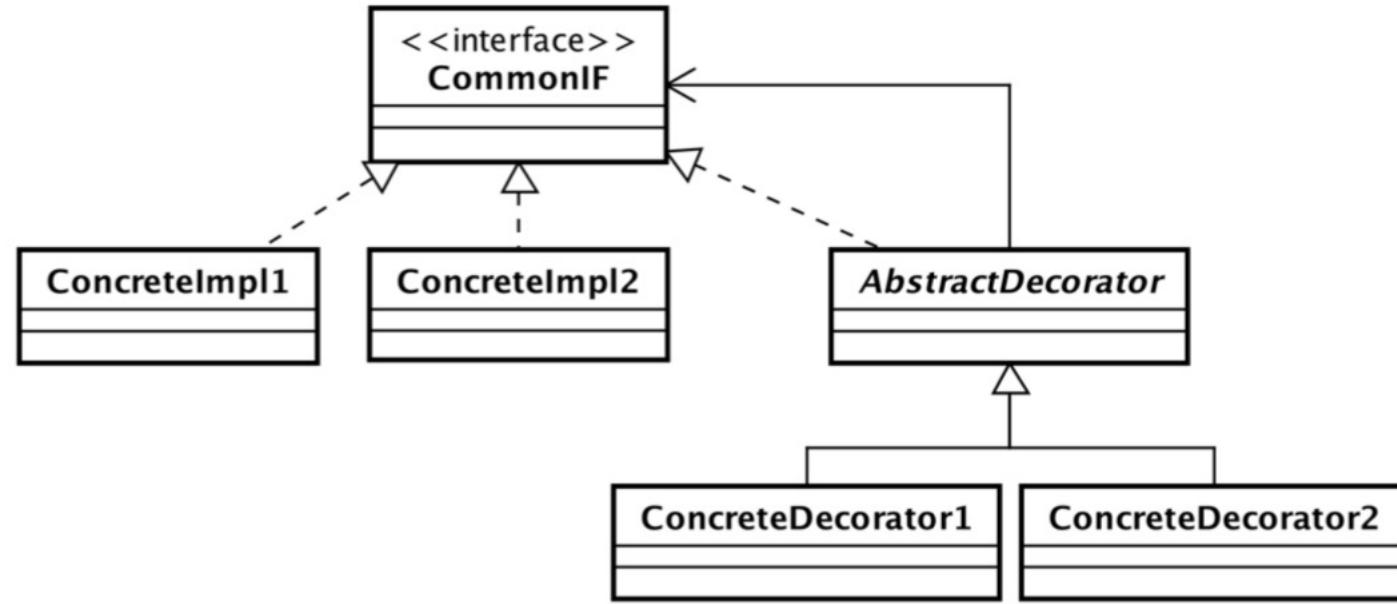


## Dekorierer

- Über das Erfüllen eines Interfaces, Ummanteln mit Funktionalität
- Erweiterungen in der Funktionalität werden wie ein Mantel / eine Schicht um die vorhandene Funktionalität gelegt. Daher manchmal auch Wrapper genannt.

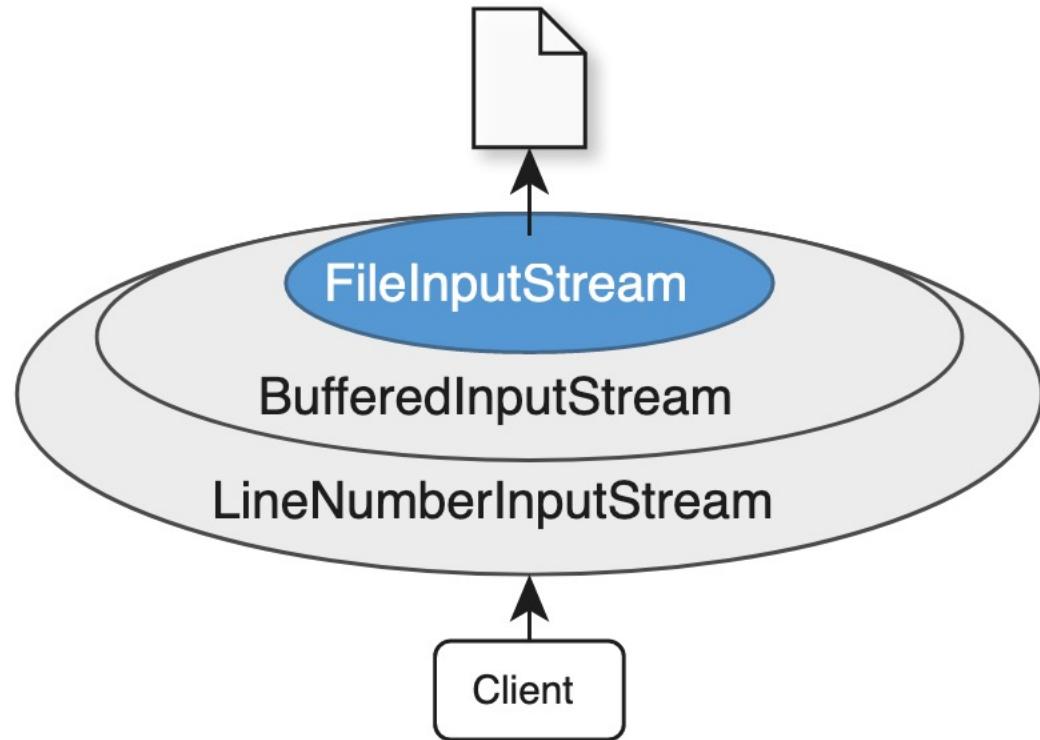


- Jede Dekoriererkasse realisiert nur einen Teil der Gesamtfunktionalität.
- Aber: Keine Kontrolle, welche Funktionalität hinzugefügt wird
- Kombination mit z. B. Factory Method



- **Grundlage ist ein gemeinsamer Basistyp (CommonIF)**
- Bei Aufruf von Methoden eines Dekoriererobjekts werden **korrespondierende Methoden** des referenzierten, zu dekorierenden Objekts aufgerufen und somit der Auftrag delegiert
- Problemlos können auch mehrere **Dekoriererobjekte hintereinander geschaltet** werden, d. h., es findet dann eine **mehrfache Ummantelung** statt.

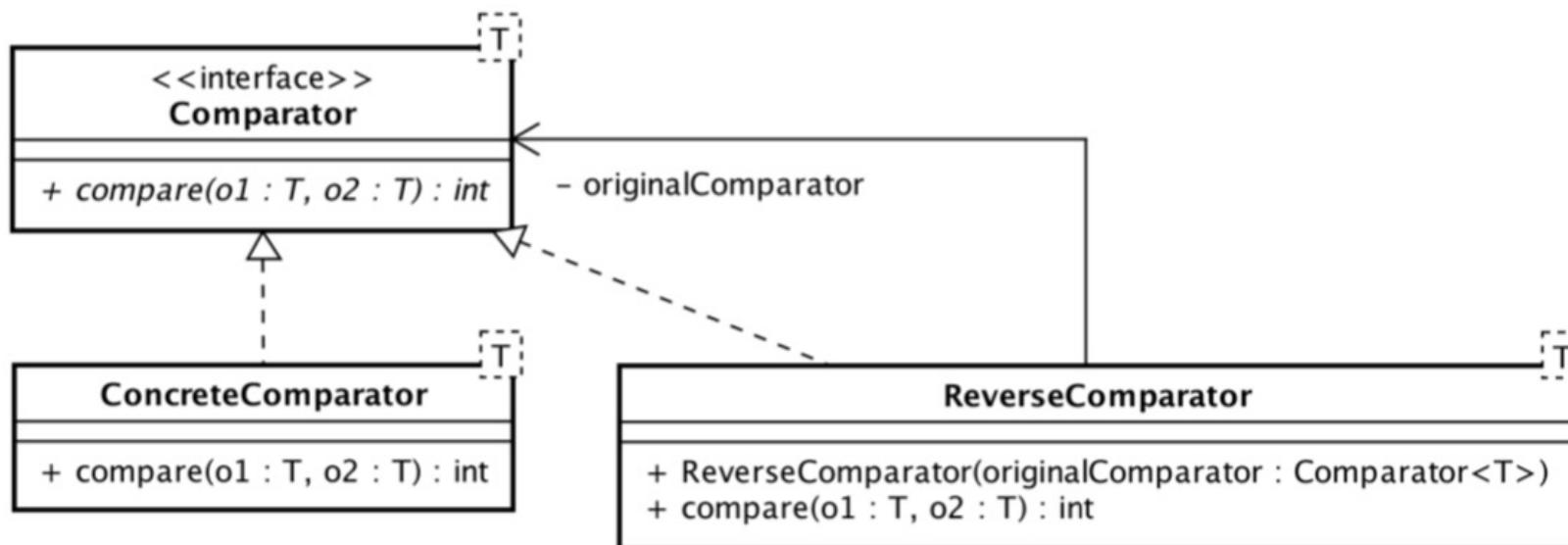
# Anwendungsbeispiel I



## Anwendungsbeispiel II



- Umdrehen der Sortierreihenfolge



## Anwendungsbeispiel II

---



```
public final class ReverseComparator<T> implements Comparator<T>
{
    private final Comparator<T> originalComparator;

    public ReverseComparator(final Comparator<T> originalComparator)
    {
        this.originalComparator = Objects.requireNonNull(originalComparator,
                "originalComparator must not be null!");
    }

    @Override
    public int compare(final T o1, final T o2)
    {
        return originalComparator.compare(o2, o1);
    }
}
```

## Bewertung Dekorierer



- + **Transparente Ergänzung zusätzlicher Funktionalität** – Funktionalität lässt transparent hinzufügen. Dies ist sogar zur Laufzeit möglich, wodurch eine Realisierung gemäß diesem Muster einer puren statischen Vererbung überlegen ist.
- + **Hintereinanderschaltung** – Mehrere unterschiedliche Dekoriererklassen lassen sich hintereinander schalten, um komplexere Funktionalitäten zu realisieren.
- + **Flexibilität** – Die zu dekorierende Klasse ist nicht festgelegt, da lediglich gegen eine gemeinsame Schnittstelle programmiert wird. Dekoriererklassen können für verschiedene zu dekorierende Klassen genutzt / wiederverwendet werden. Statische Vererbung erlaubt das nicht.
- + **Vereinfachung von Vererbungshierarchien** – Komplexe und unübersichtliche Vererbungshierarchien lassen sich durch Einsatz dieses Musters vermeiden.
- o **Gemeinsamer Basistyp benötigt** – Es ist ein gemeinsamer Basistyp erforderlich, der die öffentliche Schnittstelle für Dekoriererklassen und für zu dekorierende Objekte definiert.

## Bewertung Dekorierer



- o **Fehlende Kontrolle** – Eine Kontrolle, wer welche Funktionalität wie und wann hinzufügt und ob dies sinnvoll ist, bleibt der Disziplin des Entwicklers überlassen. Eine mehrfache Hintereinanderschaltung von Objekten derselben Dekoriererkasse, beispielsweise mehrmals Instanzen von `BufferedInputStream` oder `ReverseComparator<T>`, ist somit möglich, aber meistens nicht sinnvoll.
- **Zugriff auf Spezialisierungen schwieriger möglich** – Die Funktionalität wird durch Dekoriererobjekte transparent hinzufügt, wodurch ein Aufrufer nicht direkt darauf zugreifen kann, da dieser nur eine Referenz auf die allgemeine Dekoriererkasse hält. Als Lösung kann man eine Referenz auf eine konkrete Dekoriererkasse speichern, um deren Zusatzfunktionalität explizit nutzen zu können. Beispiel `BufferedInputStream`
- **Implementierungsaufwand** – Enthält das zu ummantelnde Interface relativ viele Methoden, so müssen all diese implementiert werden.

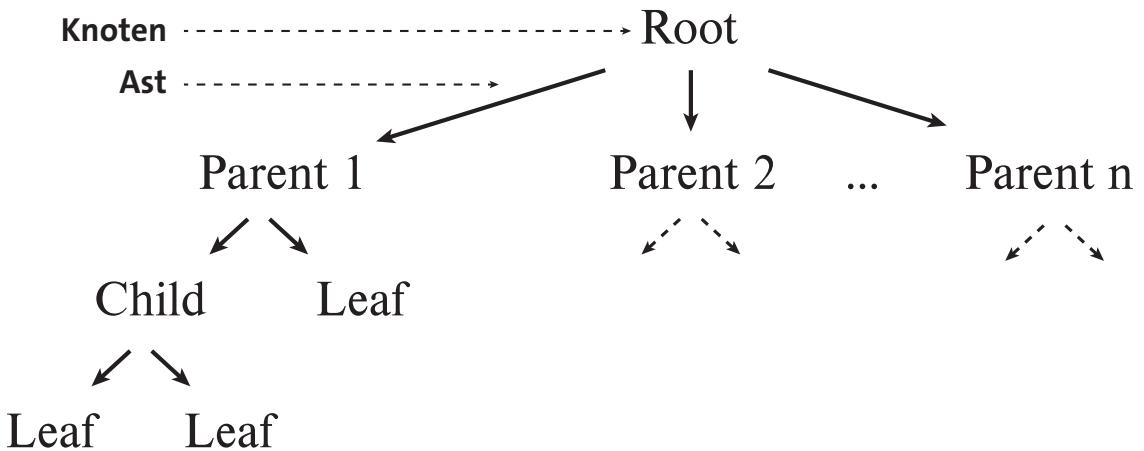


# Kompositum (Composite)

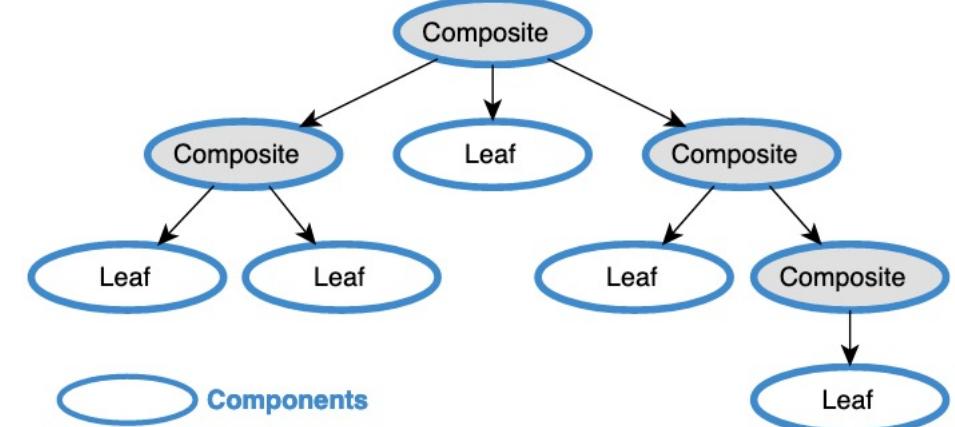
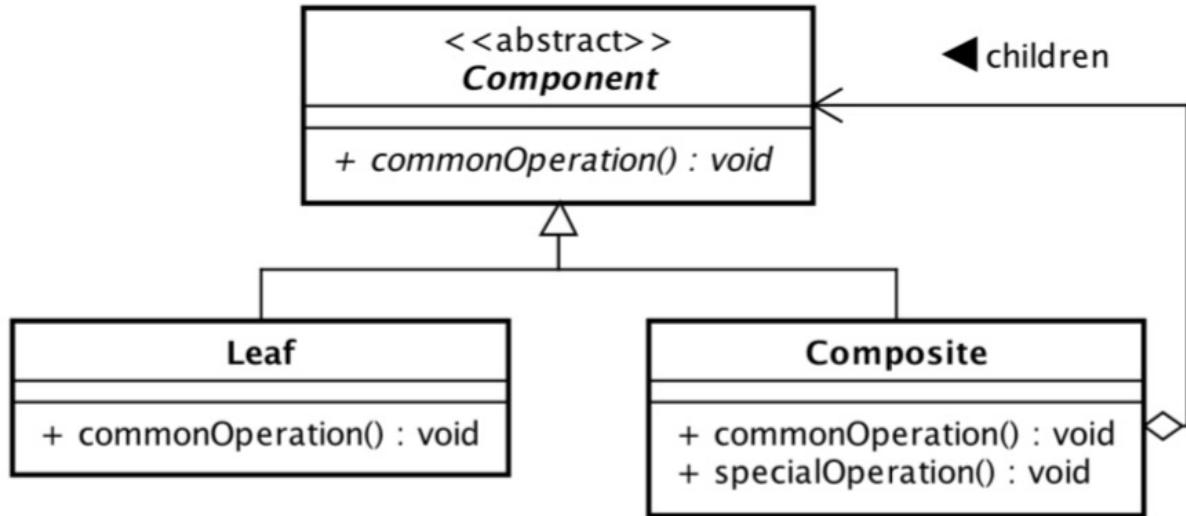
# Motivation und Kurzbeschreibung



- Das KOMPOSITUM-Muster ermöglicht es, in hierarchischen Datenstrukturen sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln.



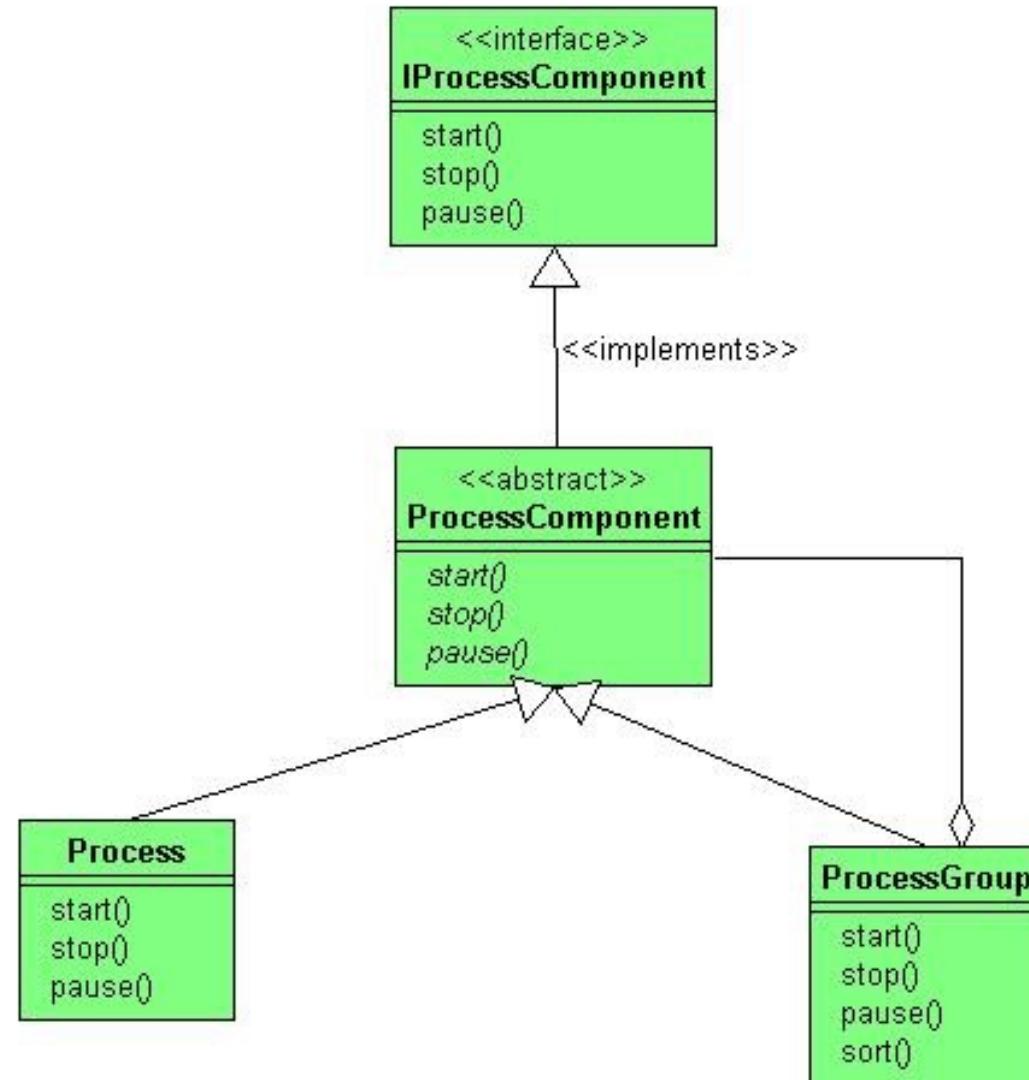
- Damit kann man in den meisten Fällen den Unterschied zwischen Einzelobjekten und Kompositionen außer Acht lassen.
- Ähnliche Ideen wie Iterator: Möglichkeit verschiedene Datenstrukturen auf gleiche Art zu durchlaufen; hier Behandlung der Elemente



- Um sowohl **Einzelobjekte** als auch **Kompositionen einheitlich behandeln** zu können, definiert eine **abstrakte Basisklasse** (alternativ ein Interface) **Component** die gemeinsame Schnittstelle. Diese wird sowohl von **Einzelobjekten** mit dem Typ **Leaf** als auch vom **Kompositum** mit dem Typ **Composite** implementiert.
- Das Kompositum wird in der Regel noch weitere Methoden besitzen, um beispielsweise die hierarchische Struktur aufzubauen.



- Kompositum in Kombination mit der Technik Interface + Abstrakte Basisklasse





**Diskussion:**  
**Wie bauen wir den Baum**  
**auf und wo sind diese**  
**Methoden beheimatet?**

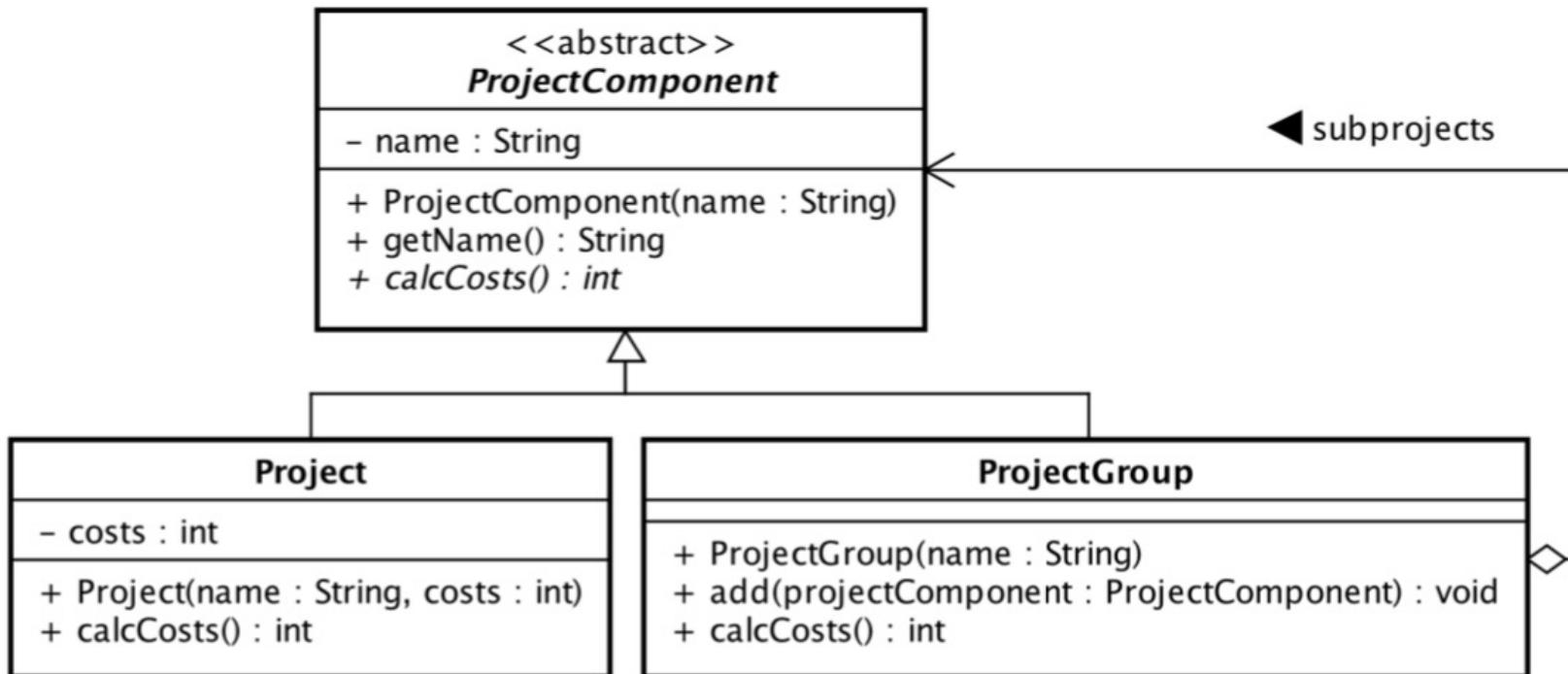
# DISKUSSION

---



- Bei der GoF sind die **speziellen Operationen**, unter anderem diejenigen, die zum Aufbau der Hierarchie notwendig sind, **Bestandteil der gemeinsamen Schnittstelle**.
- Durch die Definition innerhalb der Basisklasse müssen diese Methoden in der Klasse der **Einzelobjekte** implementiert werden. Dies kann entweder in Form eines leeren Methodenrumpfs geschehen oder aber, indem dort eine UnsupportedOperationException ausgelöst wird. Beide Varianten können problematisch sein. **Meiner Ansicht nach sollten daher Spezialoperationen nicht Bestandteil der gemeinsamen Schnittstelle sein.**
- In **grafischen Oberflächen** wird für einige Aktionen die **Unterscheidung** zwischen Einzelobjekt und Kompositum **benötigt**, etwa um **Aktionen** auf Gruppen **zu aktivieren** oder Aktionen für Einzelemente **zu deaktivieren**. Dafür bietet es sich an, die gemeinsame Schnittstelle um eine **Methode isComposite()** zu erweitern. Die Implementierung für Einzelobjekte gibt immer den Wert false zurück, die für Komposita den Wert true. Nur im letzteren Fall kann ein expliziter Cast auf die Kompositum-Klasse erfolgen, um spezifische Operationen auszuführen.

# Anwendungsbeispiel



## Anwendungsbeispiel

---



```
@Override  
public int calcCosts()  
{  
    int costs = 0;  
    for (final ProjectComponent current : subprojects)  
    {  
        costs += current.calcCosts();  
    }  
  
    return costs;  
}
```



# DEMO

`CompositeExample.java`

---

## Bewertung Kompositum



- + **Vereinfachung** – Der Zugriff auf die Funktionalität einer **komplexeren Datenstruktur** wird **vereinfacht**, weil ein Klient nicht wissen muss, welche Art von Element er anspricht.
- + **Strukturierung** – Zugriffe werden **strukturiert** und **Implementierungsdetails** lassen sich **verstecken**.
  - o **Erschwerter Zugriff auf Spezialisierungen** – Durch die Gleichbehandlung über eine gemeinsame Schnittstelle wird ein Zugriff auf Erweiterungen und Eigenschaften von Spezialisierungen von Blättern oder Kompositum verhindert. Beide können weitere Operationen anbieten, die sich jedoch nur mit Aufwand ansprechen lassen.



---

# Design Pattern Workshop

- Abstract Factory
- Bridge
- Flyweight





---

## Exercises Part 3

<https://github.com/Michaeli71/DesignPatterns.git>





# Part 4: Verhaltensmuster



# Verhaltensmuster im Überblick

---



## Verhaltensmuster strukturieren oder vereinfachen komplexe Abläufe:

- Ein **ITERATOR** bietet eine **einheitliche Möglichkeit, verschiedene Datenstrukturen zu durchlaufen** und dabei die **Details zu verstecken**.
- Ein **NULLOBJEKT** repräsentiert einen **null-Wert als Objekt** und hilft, **Zustandsabfragen zu vermeiden**.
- Eine **SCHABLONENMETHODE** erlaubt es, gewisse **Teilschritte eines Algorithmus vorzugeben** und an anderen Stellen **Variationspunkte bereitzustellen**.
- Die **STRATEGIE** hilft dabei, **verschiedene Ausprägungen von Algorithmen** zu definieren.
- ...



## Verhaltensmuster strukturieren oder vereinfachen komplexe Abläufe:

- ...
- Das **BEFEHL**-Muster fasst **Methodenaufrufe** zu **Befehlsaktionen** zusammen.
- Ein **PROXY** agiert als **Stellvertreter** für ein anderes **Objekt** und kontrolliert den **Zugriff** darauf.
- Ein **EOBACHTER** ermöglicht es, **Zustandsänderungen** anderer Objekte zu **observieren** und darauf zu **reagieren**.
- Die **MODEL-VIEW-CONTROLLER**-Architektur (MVC) dient zur **Trennung von Zuständigkeiten** und zur **Darstellung** verschiedener **Sichten** auf gleiche Daten.
- Ein **VERMITTLER** steuert und koordiniert die Kommunikation zwischen Objekten.



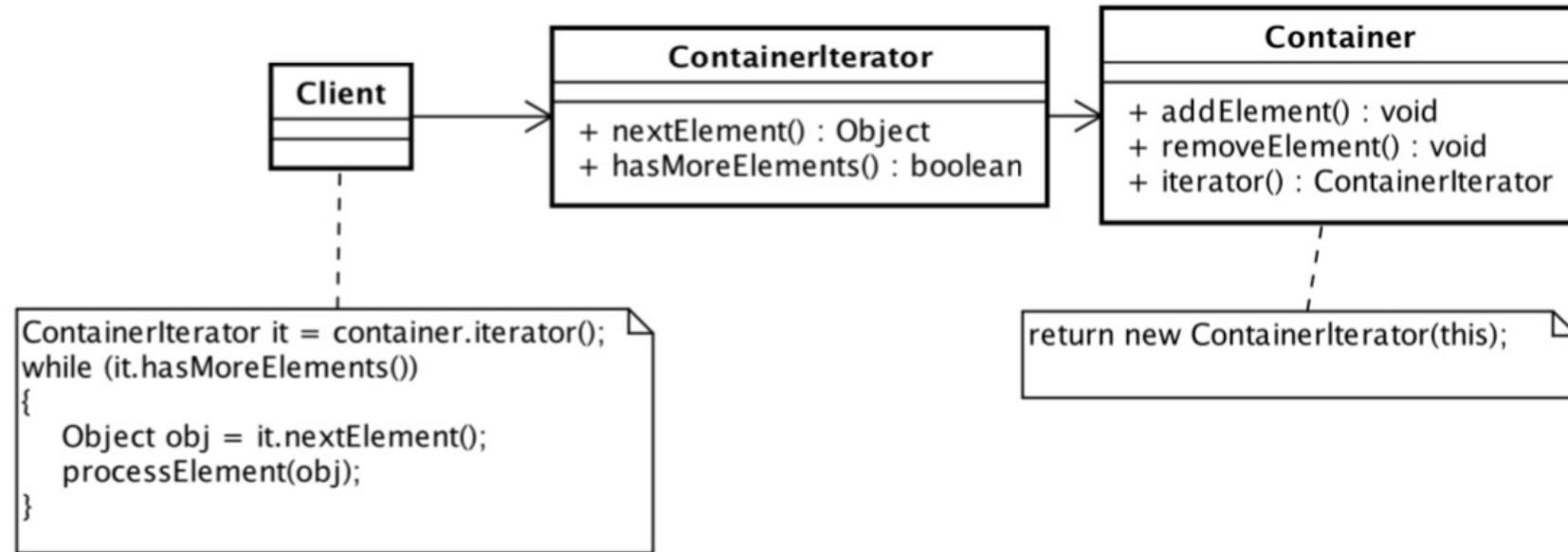
# Iterator

# Motivation und Kurzbeschreibung Iterator

---



- **Möglichkeit verschiedene Datenstrukturen auf gleiche Art zu durchlaufen**
- **Abstraktion von der zu traversierenden Datenstruktur:**
  - Keine Kenntnis des internen Aufbaus nötig
  - Bäume lassen sich durchlaufen wie Listen oder Sets
- **Traversierer wird Iterator genannt**
- **Mindestfunktionalität:**
  - nächste Element abfragen
  - Test auf weitere Elemente



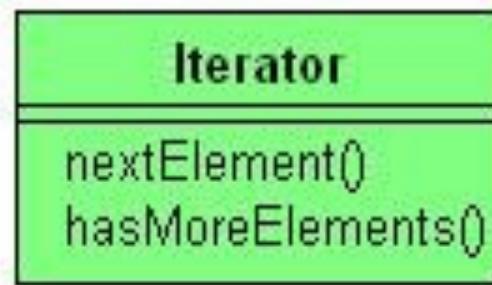
- **ContainerIterator trennt Container-Objekt vom Traversierungsmechanismus:**
  - Datenstruktur verwaltet nur die Daten,
  - jedoch nicht die Art, wie sie durchlaufen wird.
- Die Iteratorklasse bietet die Möglichkeit, zu testen, ob ein **nächstes Element existiert**, und erlaubt es zudem, ein solches zu **ermitteln**. Die dazu notwendigen Methoden werden oftmals `hasNext()` und `next()` bzw. `hasMoreElements()` und `nextElement()` oder ähnlich genannt.

## Probleme «Iterator»

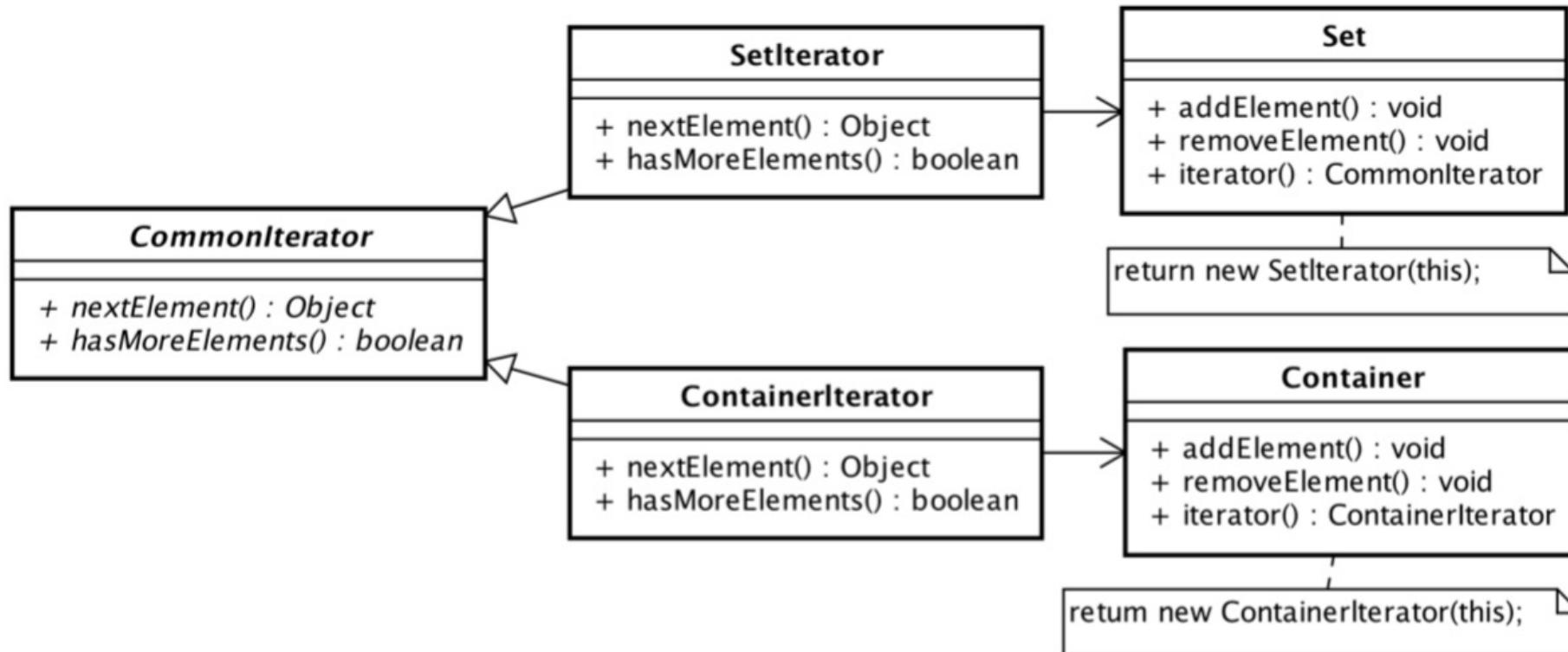
---



- **Werden mehrere eigene Datenstrukturen erzeugt, so wäre es unpraktisch, dafür jeweils spezielle Iteratoren als Anwender kennen zu müssen. Erschwert u.a. Änderungen der Aggregationsklasse, da der Klientcode auf den speziellen Iteratortyp verweist.**
- **Wir nutzen die beim OO-Design besprochene Technik des gemeinsamen Interfaces und definieren einen polymorphen Iterator mit Interface Iterator / CommonIterator als Basis. Dies definiert eine Schnittstelle mit Methoden zur Traversierung.**



# UML: Polymorpher Iterator



# Iterator im JDK

---



- **Massiver Einsatz im Collections-Framework**
- **Alle dort definierten Collection lassen sich per Iterator traversieren.**
- **Performance und Nebenbemerkungen**
  - Minimalen Einfluss auf die Performance
  - Lesbarkeit und Abstraktion viel besser als bei einer for-Schleife
  - Java 5 bietet for-each-Schleife um die Lesbarkeit zu erhöhen:

```
for (TimerTask t : collection)  
    t.cancel();
```

## Bewertung Iterator



- + **Abstraktion und Kapselung** – Die zu durchlaufende Datenstruktur wird gekapselt und bei Bedarf kann diese im Nachhinein bei der Implementierung verändert werden, ohne dass dies Änderungen aufseiten von Klienten verursacht.
- + **Vereinfachung** – Die Realisierung eigener Iteratorklassen für komplexere Datenstrukturen kann dabei helfen, eine ansonsten schwierige Implementierung des Iterationsprozesses zu kapseln, aus der Programmlogik herauszulösen und diese zu vereinfachen.
- + **Bessere Lesbarkeit** – Die Lesbarkeit ist deutlich besser als bei einer for-Schleife mit indiziertem Zugriff. Es wird das Durchlaufen auf einer konzeptionellen Ebene statt auf einer syntaktischen Ebene abgebildet.
- o **Existenz einer alternativen Schreibweise in der Sprache** – Seit Java 5 existiert mit der for-each-Schleife eine Möglichkeit, die Lesbarkeit von Iterationen zu verbessern. Es lassen sich beliebige Datenstrukturen durchlaufen, die das Interface `Iterable<T>` implementieren. Dies wird implizit mit einem `Iterator<E>` implementiert – versteckt aber dieses Detail.



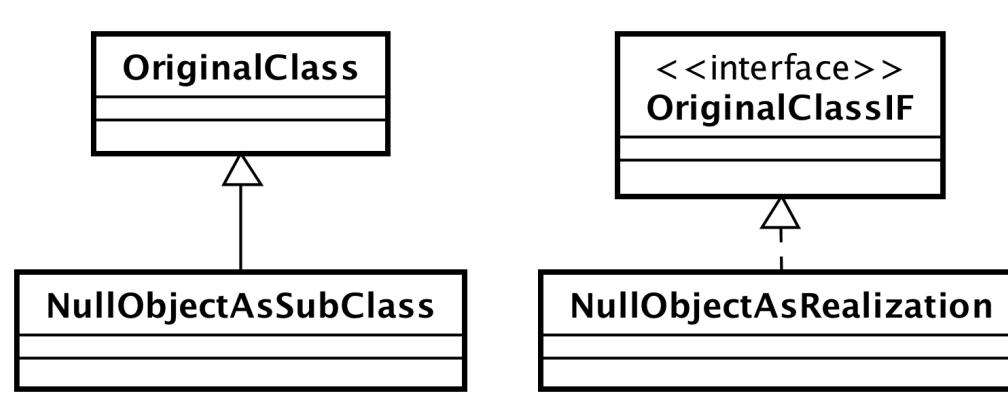
# Null-Objekt (Null Object)

# Motivation und Kurzbeschreibung Null-Objekt

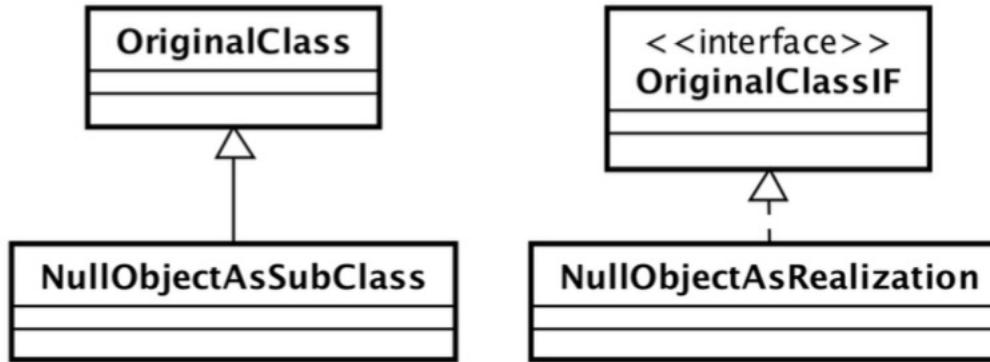
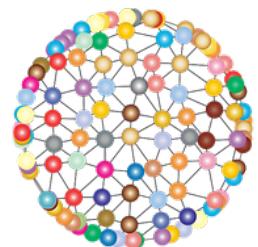
---



- ein **Platzhalter** für null-Referenzen zu verwenden
- Modelliert ein **nicht vorhandenes Objekt**.
- ein **spezielles Objekt**, das **kein eigenes Verhalten im fachlichen Sinn definiert**.
- Es kann **überall dort eingesetzt** werden, wo ansonsten **null** als semantische Aussage für **»kein Objekt«** dient.
- Damit **vereinfacht** sich die **Behandlung** für **Klienten**: Sie können **auf null-Prüfungen** und weitere **Spezialbehandlungen** im Applikationscode **verzichten**.
- Beispiele für das Null-Object-Pattern findet man im Collections-Framework mit den Klassen **EMPTY\_LIST, EMPTY\_SET, ..**

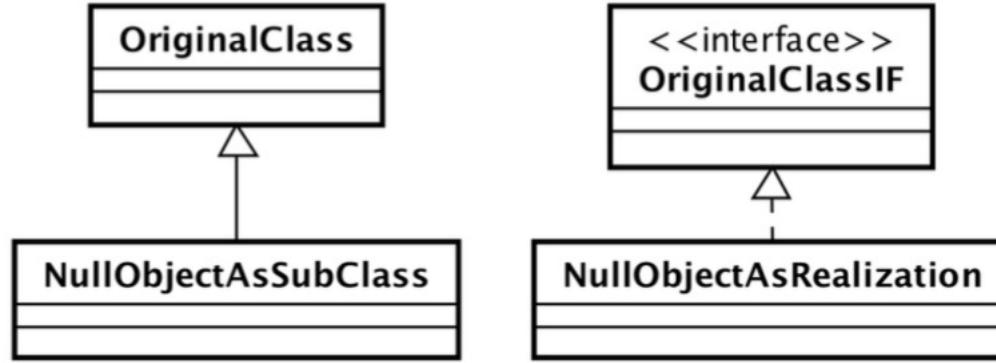


- Um als »funktionsloser« Ersatz dienen zu können, muss ein Null-Objekt die **Schnittstelle der zu vertretenden Klasse** erfüllen.
- Die **Implementierung** erfolgt entweder durch **Vererbung** oder durch **Realisierung eines Interface**.
- Alle Methoden „leer“ implementiert.



**Diskussion:**  
**Was sind die Unterschiede?**  
**Was ist zu bevorzugen?**

# DISKUSSION



- Nutzt man **Vererbung**, so ist die Null-Objekt-Klasse `NullObjectAsSubClass` eine **Spezialisierung** der Klasse `OriginalClass`, soll aber **keine Anwendungsfunktionalität** zur Verfügung stellen. Demnach muss hier **künstlich Funktionalität** der Basisklasse wieder **entfernt** werden, indem Methoden überschrieben werden und eine leere Implementierung besitzen. **Da die Null-Objekt-Klasse jedoch standardmäßig die ganze in der Basisklasse implementierte Funktionalität erbt, sind (versehentliche) Aufrufe an die Basisklasse möglich.**
- Implementiert die Klasse `OriginalClass` ein Interface, so sollte dieses bevorzugt zur Realisierung des Null-Objekts genutzt werden. Es ist sinnvoll, ein solches Interface direkt zu implementieren, d.h. ohne von der Basisklasse abzuleiten. **Dadurch sind Implementierungsdetails der Basisklasse nicht mehr im Zugriff.**

# Null-Eigenschaft ausdrücken

---



- **void-Methoden werden leer implementiert.**
- **Methoden mit Rückgabewert erfordern etwas mehr Aufwand:**
  - boolean-Methoden geben häufig `false` zurück
  - Für `int` bieten sich `0` oder `-1` als Rückgabewerte an gleiches gilt für weitere Zahlentypen (`float`, `double`, usw.)
  - Objekt-Typen kann entweder rekursiv Null-Object-Pattern oder `null`
  - Für Collections verwenden wir die Konstanten aus dem Collections- Framework: `EMPTY_LIST`, `EMPTY_SET` usw.
  - Arrays sollten ein leeres Array zurückgegeben

# Anwendungsbeispiel

---



Nutze das Null-Object-Pattern, um einen „Leer“-Iterator zu schreiben.

```
public final class NullIterator<E> implements Iterator<E>
{
    public boolean hasNext()
    {
        // TODO
    }

    public E next()
    {
        // TODO
    }

    public void remove()
    {
        // TODO
    }
}
```

---

# Anwendungsbeispiel

---



```
public final class NullIterator<E> implements Iterator<E>
{
    public boolean hasNext()
    {
        return false;
    }

    public E next()
    {
        throw new NoSuchElementException("NullIterator provides no elements!");
    }

    public void remove()
    {
        throw new UnsupportedOperationException("NullIterator does not " +
            "implement remove()!");
    }
}
```

## Bewertung Null-Objekt



- + **Bessere Lesbarkeit und keine Spezialbehandlungen** – Der Anwendungscode kann auf Spezialbehandlungen verzichten und ist dadurch klarer lesbar.
- + **Sourcecode oftmals kürzer, übersichtlicher und verständlicher**
- o **Konzeptionelle Probleme** – Mitunter **schwierig**, eine **sinnvolle Leerimplementierung** zu finden. Manchmal gibt es keine Repräsentation von null-Werten. Beispiel NullIterator und dessen next()-Methode, weswegen auf das **Auslösen einer Exception als Hilfsmittel** zurückgegriffen wird. Auch eine sinnvolle Implementierung der Methode remove() lässt sich schwerlich finden – es gibt ja keine Daten, die man löschen könnte.
- o **Fehlerverschleierung möglich** – Manchmal ist es erforderlich, »**es gibt ein Objekt**« und »**es gibt kein Objekt**« deutlich voneinander unterscheiden zu können. Dann ist dieses Musters eher ungeeignet und kann Fehler eher verschleiern.
- o **Alternative mit Java 8** – Java 8 bietet die Klasse `Optional<T>` für die Modellierung optionaler Werte.



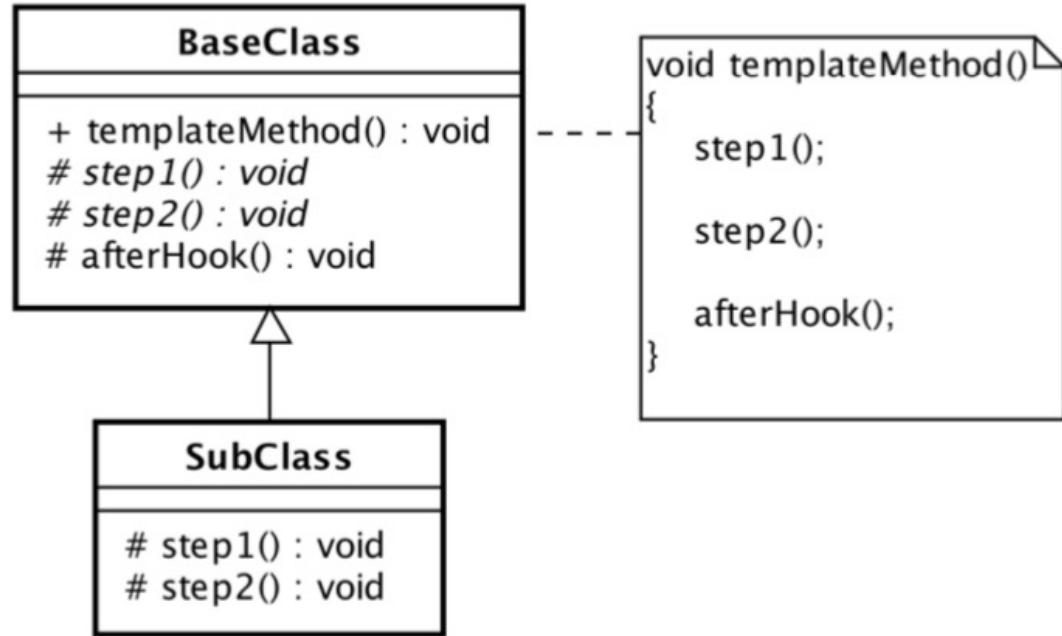
# Schablonenmethode (Template Method)

# Motivation und Kurzbeschreibung Null-Objekt

---



- Idee ist es, einen **Algorithmus** in verschiedene **Schritte** aufzuteilen
- **Grundzüge** des Algorithmus in einer Basisklasse definieren und es **Subklassen** erlauben, einige der **Berechnungsschritte** zu **implementieren**.
- Die Abfolge der Schritte wird in einer **speziellen Methode** der **Basisklasse** realisiert, die **Schablonenmethode** genannt wird. Sie ist in der Regel `final` definiert, um die grundsätzliche Abfolge zu schützen.
- Einige **Schritte** sind in **Basisklasse** noch undefiniert und können dann von **Subklassen** **ausformuliert** werden.
- Das beschriebene Vorgehen stellt sicher, dass die Struktur des Algorithmus unverändert bleibt, Subklassen jedoch Möglichkeiten der Einflussnahme gegeben wird.



```

public void paint(Graphics g)
{
    super.paint(g);

    final Graphics2D g2d = (Graphics2D) g;

    drawSheetAndBackground(g2d);

    if (showRuler)
        drawRuler(g2d);

    if (showGrid)
        drawGrid(g2d);

    drawContent(g2d);
}

public abstract drawContent(Graphics2D g2d);
  
```

- Der grundsätzliche Ablauf und **Algorithmus** ist in der Basisklasse **BaseClass** in der **Methode templateMethod()** **realisiert** und **umfasst** die **Teilschritte** 1 und 2 und einen abschließenden **Hook**. Diese **Funktionalität** wird durch die **korrespondierenden Methoden** **step1()**, **step2()** sowie **afterHook()** **realisiert**. Die Methode **templateMethod()** ist **final** und die Methoden **step1()** und **step2()** sind **abstrakt**. Die Methode **afterHook()** ist in der Basisklasse leer **implementiert**, kann aber in Subklassen **redefiniert** werden.

# Anwendungsbeispiel



```
public abstract class BaseDrawingComponent extends JComponent
{
    public final void paint(final Graphics g)
    {
        super.paint(g);

        final Graphics2D g2d = (Graphics2D) g;

        drawBackground(g2d);
        drawContent(g2d);
        postDraw(g2d);           // hook
    }

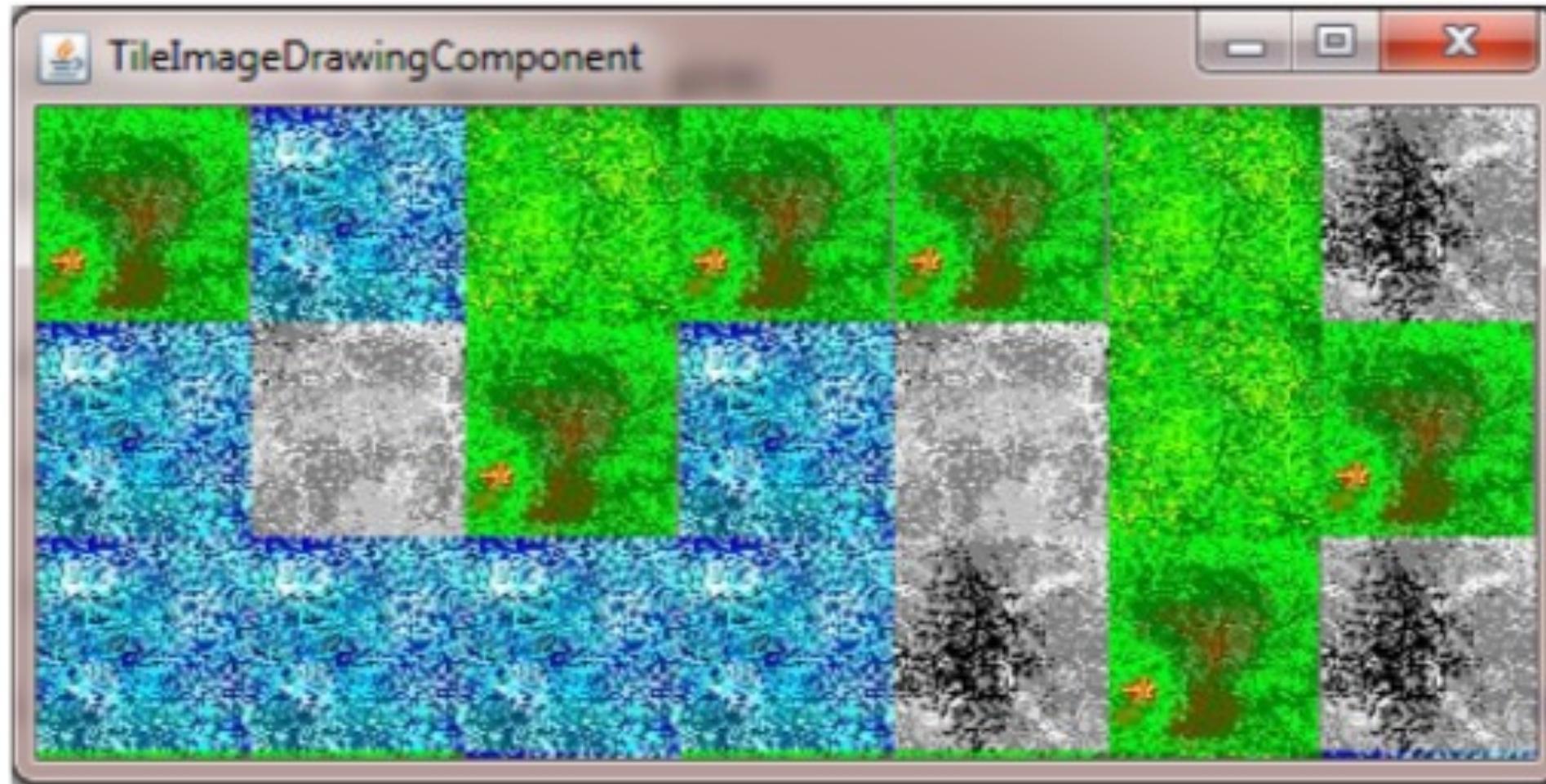
    private void drawBackground(final Graphics2D g2d)
    {
        final Dimension componentSize = getSize();
        g2d.setColor(Color.GRAY);
        g2d.fillRect(0, 0, componentSize.width, componentSize.height);
    }

    protected abstract void drawContent(final Graphics2D g2d);

    protected void postDraw(final Graphics2D g2d) { }
}
```

## Anwendungsbeispiel

---



## Anwendungsbeispiel

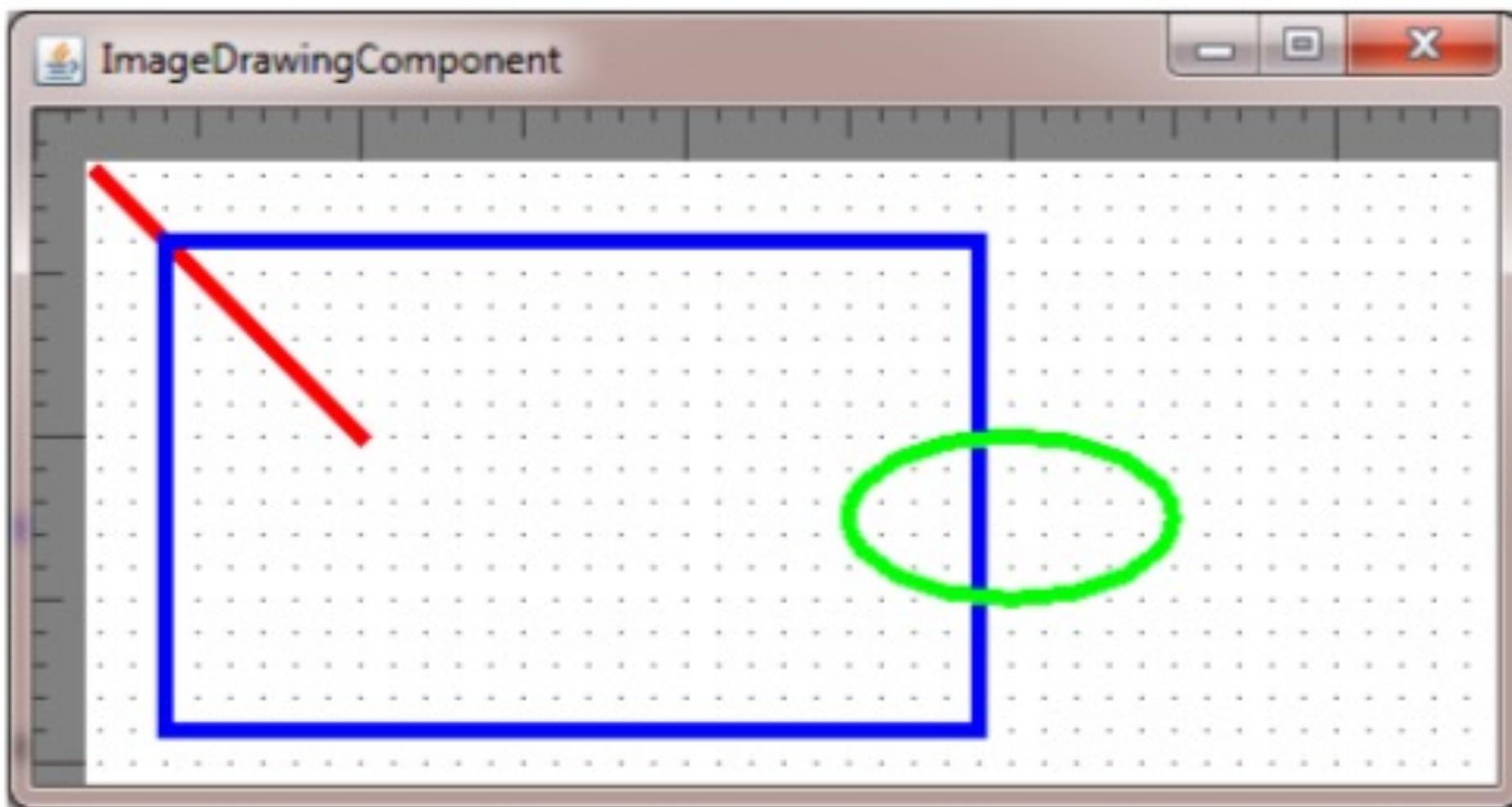
---



```
@Override  
public void drawContent(final Graphics2D g2d)  
{  
    for (int x = 0; x < getSize().width; x += TILES_WIDTH)  
    {  
        for (int y = 0; y < getSize().height; y += TILES_HEIGHT)  
        {  
            final int tileIndex = (int) (Math.random() * tileImages.length);  
            g2d.drawImage(tileImages[tileIndex], x, y, null);  
        }  
    }  
}
```

## Anwendungsbeispiel

---



# Anwendungsbeispiel

---



```
@Override  
public void drawContent(final Graphics2D g2d)  
{  
    drawSheet(g2d);  
    drawGrid(g2d);  
    drawFigures(g2d);  
    drawRuler(g2d);  
}  
  
private void drawSheet(final Graphics2D g2d)  
{  
    g2d.setColor(Color.WHITE);  
    g2d.fillRect(0, 0, getSize().width, getSize().height);  
}  
  
private void drawGrid(final Graphics2D g2d)  
{  
    g2d.setColor(Color.DARK_GRAY);  
  
    for (int x = 0; x < getSize().width; x += GRID_SIZE_X)  
    {  
        for (int y = 0; y < getSize().height; y += GRID_SIZE_Y)  
        {  
            g2d.drawLine(x, y, x, y);  
        }  
    }  
}
```

---



---

# DEMO

`TileImageDrawingComponent.java`  
`ImageDrawingComponent.java`

## Bewertung Schablonenmethode



- + **Definition von Erweiterungsstellen** – Es wird ein Algorithmus vorgegeben. Subklassen können an speziellen Stellen eigene Funktionalität einbringen.
- + **Stabilität** – Standardablauf bleibt erhalten, weniger Fehlermöglichkeiten durch versehentliche Abänderung eines Ablaufs.
- o **Weniger Flexibilität** – Man ist darin eingeschränkt, den Algorithmus zu modifizieren. In diesem Fall ist diese Eigenschaft explizit gewünscht. Soll der Algorithmus jedoch variiert werden können, so ist das Muster SCHABLONENMETHODE nicht die geeignete Wahl. Alternativ kann man mit dem im Folgenden vorgestellten STRATEGIE-Muster zwar den Algorithmus variieren, dafür aber keine Teilschritte vorgeben.



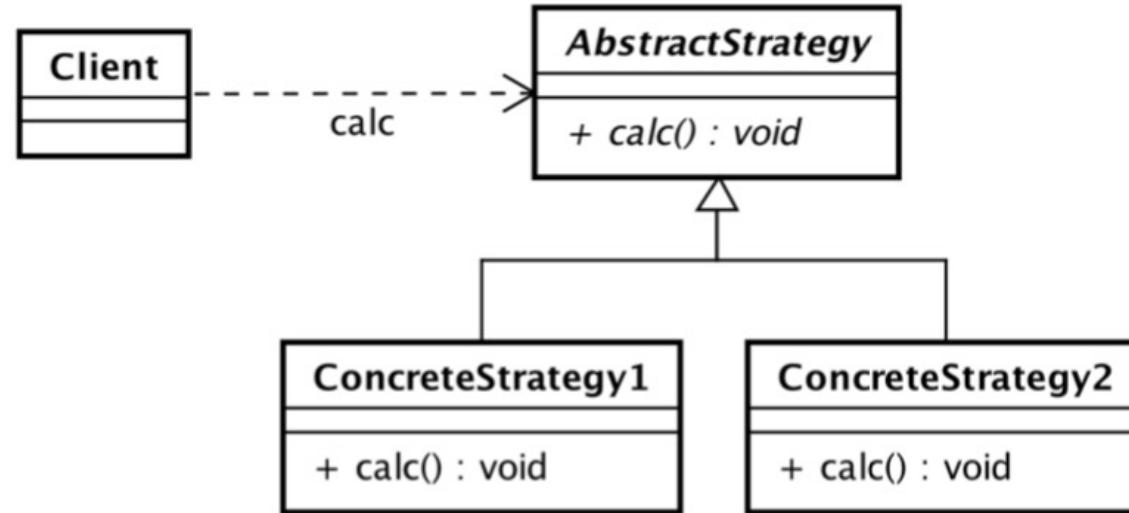
# Strategie (Strategy)

## Motivation und Kurzbeschreibung Strategie

---



- das Verhalten eines Algorithmus an ausgesuchten Stellen anzupassen.
- Im **Unterschied** zum Muster SCHABLONENMETHODE werden die variablen Bestandteile eines Algorithmus durch **eigene Klassen statt durch überschriebene Methoden** realisiert.

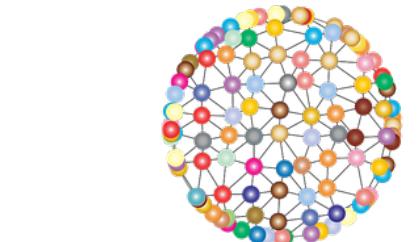


- Beim STRATEGIE-Muster werden daher die variablen Teile eines Algorithmus jeweils in eigenen Klassen mit gemeinsamem Basistyp gekapselt und sind dadurch austauschbar. Das konkrete Verhalten kann bei Bedarf sogar erst zur Laufzeit durch Wahl einer beliebigen vorhandenen Realisierung festgelegt werden.

# Wahl der unterschiedlichen Funktionalitäten bzw. Strategien mit if-Anweisungen

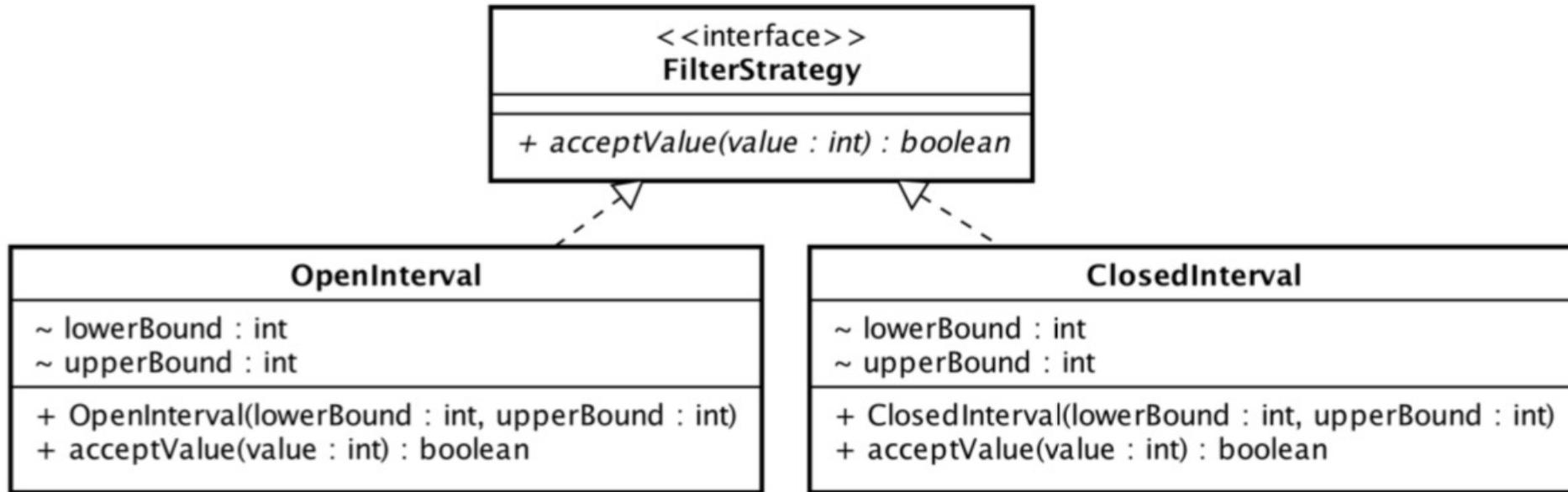


```
public static List<Integer> filterAll(final List<Integer> inputs, final FilterType filterStrategy,  
                                    final int lowerBound, final int upperBound)  
{  
    final List<Integer> resultSet = new LinkedList<>();  
  
    if (filterStrategy == FilterType.CLOSED_INTERVAL)  
    {  
        for (final Integer value : inputs)  
        {  
            if (value >= lowerBound && value <= upperBound)  
                resultSet.add(value);  
        }  
    }  
    if (filterStrategy == FilterType.OPEN_INTERVAL)  
    {  
        for (final Integer value : inputs)  
        {  
            if (value > lowerBound && value < upperBound)  
                resultSet.add(value);  
        }  
    }  
  
    return resultSet;  
}
```



**Was ist daran  
problematisch?  
Gegen welches OO Principle  
wird verstößen?**

# Anwendungsbeispiel



- Die Basis bildet ein Interface **FilterStrategy** mit der dort definierten Methode `acceptValue(int)`. Zur Intervallprüfung werden zwei konkrete Strategieklassen **OpenInterval** und **ClosedInterval** definiert

# Anwendungsbeispiel



```
/** Mathematisch: [lower,upper] --> lower <= value <= upper */
public static class ClosedInterval implements FilterStrategy
{
    private final int lowerBound;
    private final int upperBound;

    public ClosedInterval(final int lowerBound, final int upperBound)
    {
        if (upperBound < lowerBound)
            throw new IllegalArgumentException("lowerBound must be <= upperBound");

        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
    }

    @Override
    public boolean acceptValue(final int value)
    {
        return lowerBound <= value && value <= upperBound;
    }

    @Override
    public String toString()
    {
        return "ClosedInterval [" + lowerBound + ", " + upperBound + "]";
    }
}
```

# Anwendungsbeispiel

---



**Variable Bestandteile durch FilterStrategy realisiert.**

```
public static List<Integer> filterAll(final List<Integer> inputs,  
                                    final FilterStrategy filterStrategy)  
{  
    final List<Integer> result = new ArrayList<>();  
    for (final Integer value : inputs)  
    {  
        if (filterStrategy.acceptValue(value))  
            result.add(value);  
    }  
  
    return result;  
}
```



---

# DEMO

**StrategyFilterExample.java**

---

## Bewertung Strategie



- + **Erhöhte Flexibilität** – Die Auswahl aus **verschiedenen Implementierungen** kann zur Laufzeit erfolgen. Dadurch erhöht sich die Flexibilität und die Wiederverwendbarkeit.
- + **Bessere Erweiterbarkeit** – Die Alternativen besitzen **keine Abhängigkeiten** untereinander, sodass sich verschiedene Anforderungen (Bereichsgrenzen, Primzahlen usw.) unabhängig realisieren lassen. Zudem kann man **zusätzliche Anforderungen**, wie für die inverse Abbildung gezeigt, **leicht und gezielt integrieren**.
- o **Kopplung an konkrete Strategien** – Klienten müssen die konkreten **Strategieklassen** kennen und **instanziieren**, um die gewünschte Funktionalität auszuführen. Eine Kombination mit einer **FABRIKMETHODE** bietet sich zur Kapselung und loseren Kopplung an.
- **Erhöhter Umfang** – Die Anzahl der Klassen erhöht sich.



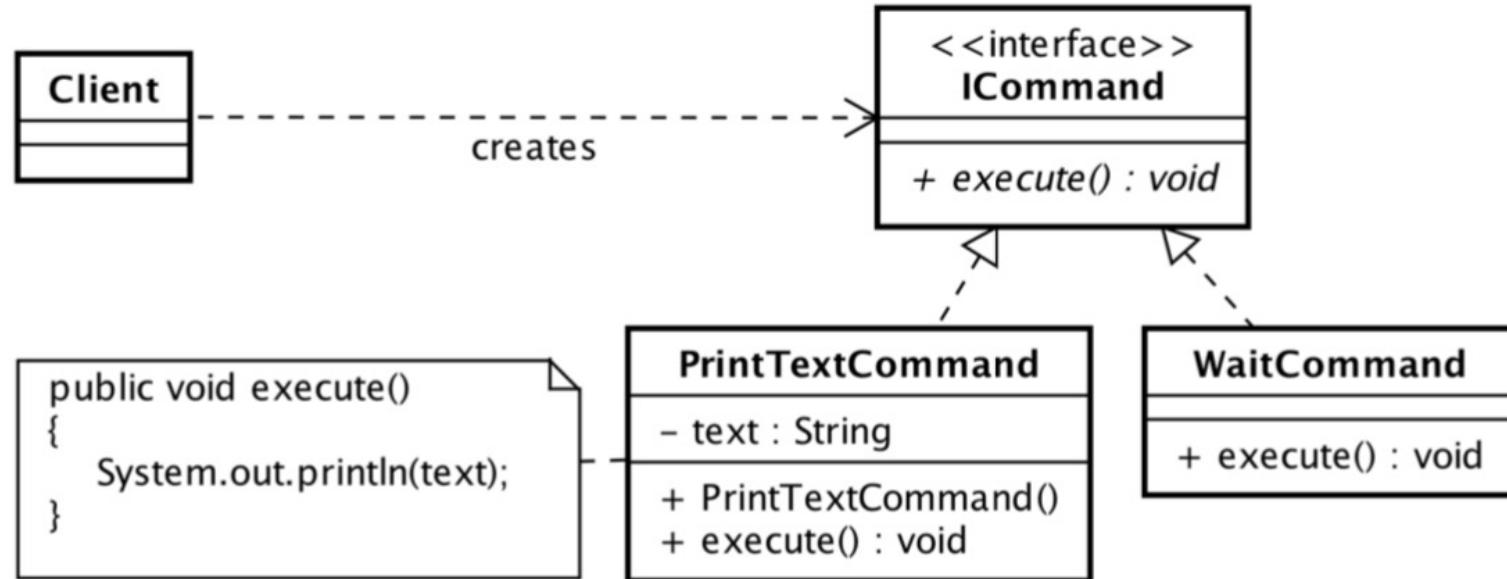
# Befehl (Command)

# Motivation und Kurzbeschreibung Befehl

---



- vereinfacht die Ausführung komplexerer Aufgaben.
- Beispiels aus der Realität, einer **Bestellung** in einem **Restaurant**, verdeutlichen. Ein Gast gibt eine Bestellung auf, etwa: »**Ein Steak medium mit Pommes und ein Bier**«. Der **Kellner notiert** diese **Bestellung** und **übergibt** sie der **Küche** und der **Theke** als **Arbeitsanweisungen** zur Ausführung. Die ausführenden **Einheiten** können anhand der Bestellung verschiedene konkrete **kleinere** Handlungen durchführen (Braten, Frittieren, Bier zapfen usw.), die **für** den **Gast** aber **uninteressant** sind (etwa die **Temperatureinstellung** am **Herd**).
- Auf die Software übertragen erkennen wir folgende Akteure bei diesem Muster: Ein Klient (Gast) gibt eine Bestellung (einen oder mehrere (abstrakte) Befehle) auf, die von einer Komponente (Kellner) registriert und von anderen Komponenten (Küche und Theke) ausgeführt wird. Ein klassisches Beispiel sind **Aktionen** eines **Editors**, die durch Menüs, Kontextmenüs oder Toolbars ausgelöst werden und beispielsweise zum Speichern eines Dokuments oder zum Erzeugen einer grafischen Figur führen.



- Einheitliche Schnittstelle **ICommand** zur Ausführung
- Aktionen werden dann innerhalb konkreter Klassen, etwa **PrintTextCommand**, realisiert
- Entkopplung: Client benötigt kein Wissen zum Kommando

## Simples Beispiel und Zwischenfazit

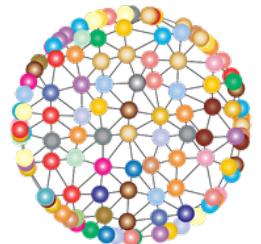
---



```
public static void main(final String[] args)
{
    // Erzeugen der Command-Objekte
    final ICommand command1 = new PrintTextCommand("Dies ist ein Text");
    final ICommand command2 = new WaitCommand();
    final ICommand command3 = new PrintTextCommand("Der Test ist beendet!");

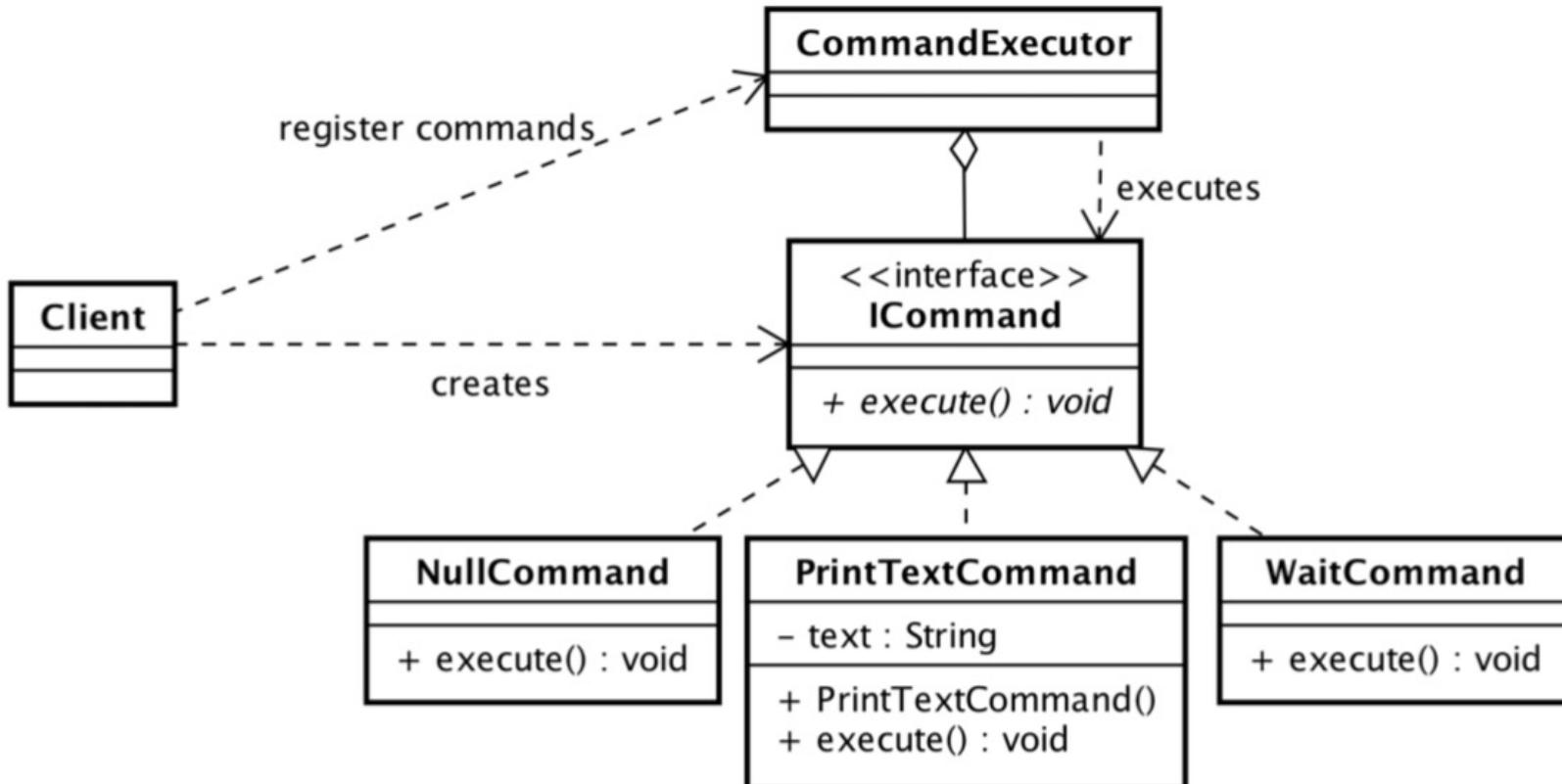
    // Ausführen der Command-Objekte
    command1.execute();
    command2.execute();
    command3.execute();
}
```

- **Einheitliche Schnittstelle zur Ausführung, Aktionen innerhalb konkreter Klassen gekapselt**
  - **Vorteil noch nicht klar ersichtlich, da dies auch durch Methoden möglich wäre**
  - **Zudem: Client führt direkt aus**
-



# Wie ist es praxisnäher?

# UML mit Ausführungskomponente CommandExecutor



- Klient «spricht» mit CommandExecutor zur Registrierung
- CommandExecutor eigenverantwortlich zur Ausführung

# Anwendungsbeispiel



```
public static void main(final String[] args) throws InterruptedException
{
    // Die Abarbeitung der Kommandos erfolgt nebenläufig
    // zu den folgenden Aktionen
    final CommandExecutor executor = new CommandExecutor();
    final Thread executorThread = new Thread(executor);
    executorThread.start();

    // Client erzeugt Kommandos
    final ICommand command1 = new PrintTextCommand("Dies ist ein Text");
    final ICommand command2 = new WaitCommand();
    final ICommand command3 = new PrintTextCommand("Der Test ist beendet!");

    // Client übergibt Kommandos an Executor
    executor.registerCommand(command1);
    executor.registerCommand(command2);
    executor.registerCommand(command3);

    // wait and quit
    Thread.sleep(7_000);
    executorThread.interrupt();
}
```



---

# DEMO

`SimpleCommandExample.java`  
`CommandExecutor.java`



**Was passiert, wenn die  
Befehle auch weitere  
Informationen zur  
Ausführung benötigen?**

## Ausführungskontext

---



- Bislang sind die vorgestellten `execute()`-Methoden funktional sehr einfach. **In der Praxis benötigen Befehle meistens einen Kontext oder Eingabeparameter zu ihrer Ausführung bzw. Daten, um auf diesen zu operieren.** Selbst für das einfache `PrintTextCommand`.
- **Sind zur Ausführung benötigte Informationen bereits zum Erzeugungszeitpunkt der Befehlsobjekte bekannt, sollten diese als Parameter an deren Konstruktor übergeben werden.** Auf diese Weise kann man verschiedene **Rahmenbedingungen und Eigenschaften festlegen**. Bezogen auf das Beispiel einer Bestellung im Restaurant sind dies beispielsweise die Namen des Gerichts und des Getränks.
- **Allerdings sind mitunter einige Informationen, etwa die konkrete Umgebung der Ausführung oder die ausführende Einheit, eventuell noch unbekannt.** Dies sind Detailinformationen, die zur tatsächlichen Abarbeitung eines Befehls, also von dessen `execute()`-Methode, benötigt werden. **Bezogen auf das Restaurantbeispiel kennt der Gast wesentliche Details der Zubereitung nicht:** Welches Stück Fleisch gebraten wird und welcher Koch die Zubereitung übernimmt, wird erst in der Küche entschieden.

## Ausführungskontext

---



Aus diesen beiden Randbedingungen folgt, dass es in der Regel zwei Arten von Parametrierungen eines Befehls gibt:

- **Spezifikationsdaten** – Einige Parameter werden genutzt, um eine konkrete Spezifikation des Befehls und damit eine Präzisierung des Auftrags vorzunehmen: Es wird das »**Was**« beschrieben, etwa: »Steak medium mit Pommes«. **Diese Daten werden von Klienten festgelegt**. Diese sollten von Klienten möglichst zum Konstruktionszeitpunkt eines Befehlsobjekts übergeben und in dessen Attributen gespeichert werden.
- **Ausführungsdaten** – Um den gewünschten Auftrag auszuführen und aus der zuvor angegebenen Spezifikation eine konkrete Abarbeitung des Befehls zu ermöglichen, werden in der Regel **weitere Daten und Akteure** benötigt, die an den Befehl übergeben werden müssen. Es wird das »**Wie**« und »**Wer**« beschrieben, etwa: »Welcher Koch die Arbeit übernimmt, welches Steak zubereitet wird, welcher Grill verwendet wird.« **Auf diese Daten hat ein Klient in der Regel keinen Einfluss**.

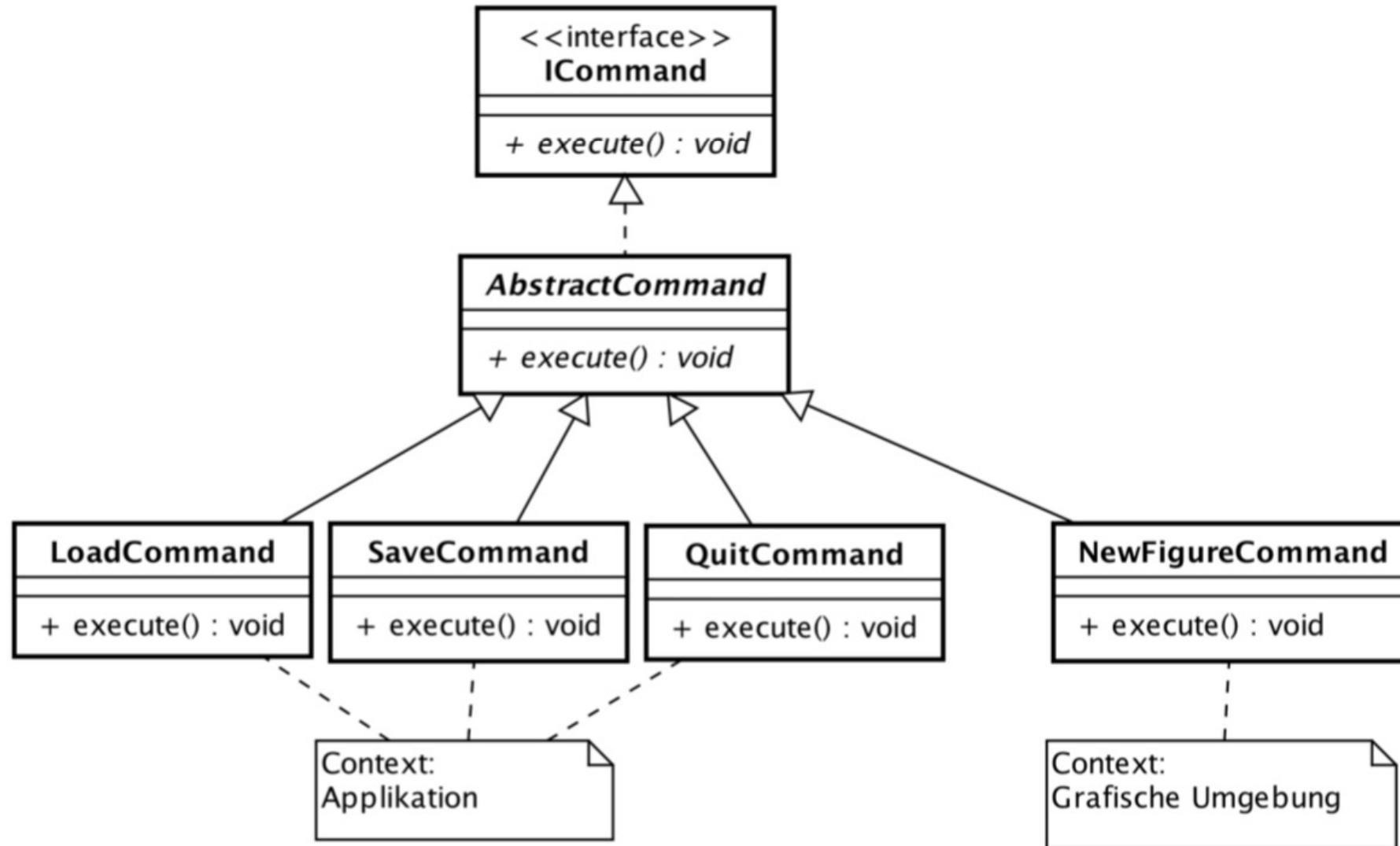
# Ausführungskontext

---



- Normalerweise benötigt ein Befehl zur Ausführung seiner `execute()`-Methode Zugriff auf diverse andere Objekte und Informationen. Zur Vereinfachung ist es sinnvoll, die **erforderlichen Daten** als Laufzeitkontext in Form einer **Value-Object-Klasse** `ExecutionContext` zu bündeln.
- Daten sind wie beschrieben **Spezifikationsdaten oder Ausführungsdaten**
- **Einfacher und klarer ist es deshalb meistens, diese Aufgabe der Informationsbereitstellung durch ein Objekt vom Typ `CommandExecutor` erledigen zu lassen.**
- Der Methode `execute()` werden die Spezifikationsdaten bzw. Kontextdaten als Klasse `ExecutionContext` übergeben. Diese stellt dann alle zur Ausführung benötigten Informationen zur Verfügung. Die Konstruktion von Befehlsobjekten kann nun unabhängig von der Bereitstellung der Ausführungsdaten erfolgen.

# Agenda



## Bewertung Befehl



- + **Lose Kopplung** – Durch die **Kapselung** von **Aktionen** in Form von **Objekten** können diese von **beliebigen Programmteilen erzeugt** und von **anderen Programmteilen ausgeführt** werden. **Funktionalität** lässt sich auf **verschiedene Wege bereitstellen**, etwa als Reaktion auf Menüs, Buttons oder andere Eingaben, **ohne dass es zu einer Kopplung** an konkrete Elemente der Benutzerschnittstelle kommt: **Aufrufer** und **ausführende Einheit** sind **nicht miteinander verbunden**, wie dies **bei einem Methodenaufruf** der Fall wäre.
- + **Bessere Abstraktion** – In einem Befehl werden verschiedene auszuführende Methodenaufrufe an einer Stelle gebündelt. Dies ist **ähnlich** zu der **Extraktion** einer **Methode**, geht allerdings einen Schritt weiter, da ein **neues, in anderen Kontexten wiederverwendbares Befehlsobjekt** entsteht.
- + **Wiederverwendbarkeit** – Ein **Befehl** stellt eine **Verhaltensbeschreibung**, eine Art **semantische Klammer** über verschiedene Methoden, dar und ist vielseitig einsetzbar. Ein **Befehl** kann von **unterschiedlichen Auftraggebern** (Menü, Kontextmenü, Toolbar usw.) an **verschiedene ausführende Einheiten weitergereicht** und dort in diversen Kontexten (Einzelselection, Mehrfachselection usw.) **ausgeführt** werden.

## Bewertung Befehl

---



- + **Fernsteuerung** – Eine Speicherung und spätere Ausführung von Befehlen ist möglich: **Befehlsobjekte** können wie alle anderen Objekte auch verarbeitet oder in **Methodenaufrufen verwendet** werden. Im Speziellen lassen sich diese **serialisieren** und zu späteren Zeitpunkten **erneut ausführen**. Daher kann dieses Muster gut zur zeitgesteuerten Fernsteuerung eingesetzt werden. Tatsächlich sind auch andere Programme – die sogar in eigenständigen JVMs laufen können – in der Lage, die Befehle zu verarbeiten.
- + **Undo-/Redo-Fähigkeit** – Eine Undo-/Redo-Fähigkeit ist leichter mit dem Muster BEFEHL zu realisieren als ohne. Zu definierten Zeitpunkten können Zwischenstände des zugrunde liegenden Modells gesichert werden, die bei Bedarf wiederhergestellt werden können.
- **Erhöhter Umfang** – Die Anzahl der Klassen erhöht sich.



---

# Design Pattern Workshop

- Visitor
- Memento
- State





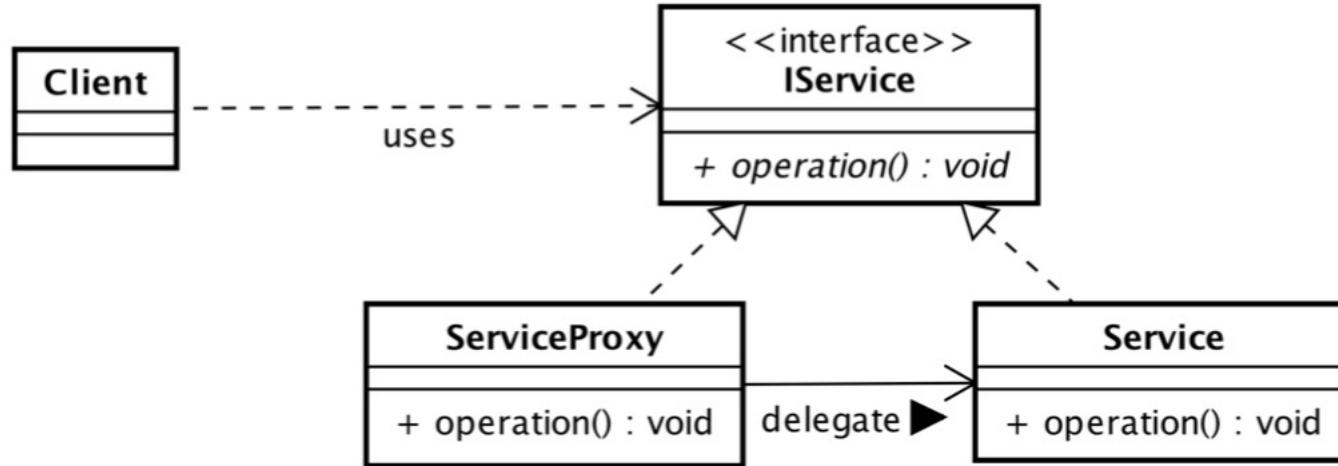
# Proxy

# Motivation und Kurzbeschreibung Proxy

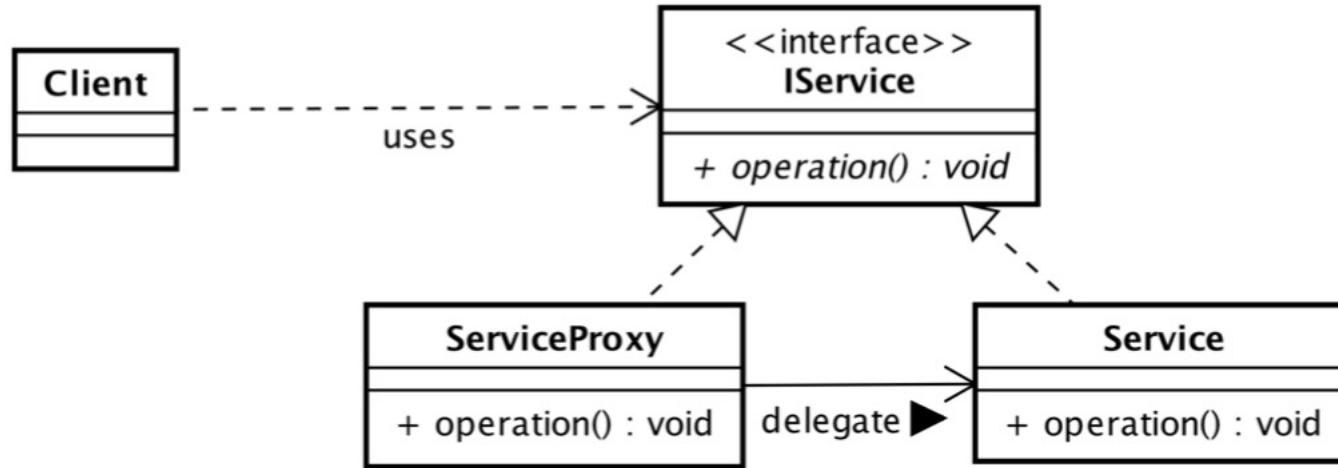
---



- **Verlagern der Kontrolle** über ein **Objekt** auf ein **Stellvertreterobjekt**, den sogenannten **Proxy**.
- Nutzer wenden sich immer an den **Proxy** und nicht direkt an das vertretene Objekt.
- Ein **Klient** kennt das eigentliche **Objekt** nicht.
- Dadurch kann der **Proxy** sowohl die **Erzeugung** als auch den **Zugriff** auf das eigentliche **Objekt** kontrollieren:
  - **Zugangskontrolle** realisieren oder
  - **Remote Calls** verstecken
  - **Vereinfachung** von **Lazy Initialization**



- Die **Basis** dieses Musters ist eine **gemeinsame Schnittstelle** **IService**. Diese stellt sicher, dass sowohl der **Proxy** in Form der Klasse **ServiceProxy** als auch das **eigentliche Objekt** vom Typ **Service** **nach außen gleich behandelt** werden können. Zur Durchführung der eigentlichen Aufgaben **verwaltet** der **Proxy** eine **Referenz** auf das **Objekt**.



- Der Proxy ermöglicht es, **zusätzlich zum Verhalten** der Methoden eines Objekts **verschiedene weitere Aktionen auszuführen**. Ein Proxy kann die **Aktionen** sowohl **vor** als auch **nach** der **Delegation des Methodenaufrufs** ausführen. Dadurch ist es möglich, **Funktionalität zu ergänzen** und **sogar zu entfernen** (indem ein Methodenaufruf nicht weitergeleitet wird).

# Anwendungsbeispiel



```
public class Service implements IService
{
    @Override
    public void doSomething()
    {
        System.out.println("doSomething");
    }

    @Override
    public String calculateSomething(final int value)
    {
        System.out.println("calculateSomething");
        try
        {
            TimeUnit.SECONDS.sleep(3);
        }
        catch (final InterruptedException e)
        {
            // can't happen here, no other thread to interrupt us
        }
        return "" + value;
    }
}
```

```
public interface IService
{
    void doSomething();
    String calculateSomething(int value);
}
```

## Varianten Proxy

---



- **Access Control Proxy** – Kann den **Zugriff** auf gewisse **Daten** steuern. Beispielsweise lässt sich **vor jeder Methodendelegation eine Rechteprüfung** ausführen, die bei Bedarf die Eingabe eines Passworts verlangt.
- **Decorating/Interceptor Proxy** – Fügt weitere **Funktionalität transparent** gemäß dem DEKORIERER-Muster **hinz**u. Häufig spricht man auch von INTERCEPTOR.
- **Lazy Init Proxy oder Virtual Proxy** – **Vermittelt** den **Eindruck**, ein **Objekt** stehe schon zur **Verfügung**, bevor es **tatsächlich erzeugt** wurde. **Objektbestandteile** werden erst in dem Moment **konstruiert**, in dem diese auch tatsächlich benutzt werden. Ein **virtueller Proxy** fungiert als **Platzhalter**, etwa beim Zugriff auf Objekte, die aufwendig zu konstruieren oder zu laden sind, **beispielsweise** werden für **Bilder** zunächst **Platzhalter** dargestellt. Das kann dies zu signifikanten **Performance-Verbesserungen** führen.
- **Remote Proxy** – Agiert als **Stellvertreter** für Objekte, die über ein **Netzwerk** (remote) angesprochen werden. Für Nutzer bleibt diese Tatsache (weitgehend) **transparent**. Ein Remote Proxy kann den **Eindruck erwecken**, ein **entferntes Objekt** wäre ein **lokales**.

# Anwendungsbeispiel: Decorating Proxy



```
public class ServicePerformanceProxy implements IService
{
    private final IService service;                                ↓

    ServicePerformanceProxy(final IService service)
    {
        this.service = service;
    }

    @Override
    public String calculateSomething(int value)
    {
        final long startTime = System.nanoTime();                  ↓
        final String result = service.calculateSomething(value);

        printExecTime("calculateSomething", System.nanoTime() - startTime);

        return result;
    }

    ...
}
```

# Anwendungsbeispiel: Decorating Proxy



```
public class ServicePerformanceProxy implements IService
{
    ...
    @Override
    public void doSomething()
    {
        final long startTime = System.nanoTime();
        service.doSomething();
        printExecTime("doSomething", System.nanoTime() - startTime);
    }

    private void printExecTime(final String methodName, final long duration)
    {
        System.out.println("Method call of '" + methodName + "' took: " +
                           TimeUnit.NANOSECONDS.toMillis(duration) + " ms");
    }
}
```

## Anwendungsbeispiel: Decorating Proxy



```
public class StaticProxyExample
{
    public static void main(String[] args)
    {
        final IService service = createService();
        service.calculateSomething(42);
        service.doSomething();
    }

    private static IService createService()
    {
        final IService service = new Service();
        return new ServicePerformanceProxy(service);
    }
}
```



```
calculateSomething
Method call of 'calculateSomething' took:
3000 ms
doSomething
Method call of 'doSomething' took: 0 ms
```



# DEMO

**StaticProxyExample.java**

---

## Anwendungsbeispiel II: Access Control Proxy



```
public class RestrictedAccessMap<K, V> implements Map<K, V>
{
    private final Map<K, V> underlyingMap = new HashMap<>();

    public void ensureAccessGranted() throws InvalidAccessRightsException {
        if (!LoggedInUserService.INSTANCE.getLoggedInUser().equals("ADMIN"))
            throw new InvalidAccessRightsException("Invalid User");
    }

    @Override
    public V put(final K key, final V value) {
        ensureAccessGranted();
        return underlyingMap.put(key, value);
    }

    @Override
    public int size() {
        ensureAccessGranted();
        return underlyingMap.size();
    }

    @Override
    public boolean isEmpty() {
        ensureAccessGranted();
        return false;
    }
}
```

Wie kann man die Prüfung dynamisch realisieren  
und nicht immer in jeder Methode selbst aufrufen?  
=> Übung Dynamic Proxy

## Bewertung Proxy



+ **Steuerung von Funktionalität** – Ähnlich zum DEKORIERER-Muster **lässt** sich die eigentliche **Anwendungsfunktionalität** um weitere Funktionalität **ergänzen**. Bei diesem Muster steht jedoch der **steuernde Charakter im Vordergrund**.

o **Aufwand durch Delegation** – Besitzt ein **Originalobjekt viele Methoden**, so ist die **Realisierung** des Proxy-Objekts durch die vielen notwendigen Delegationen **aufwendig**. Dafür können **Dynamic Proxies** hilfreich sein, deren Basis ins JDK integriert ist. Dadurch **lässt** sich der **durch Delegation verursachte Aufwand** mitunter **deutlich reduzieren**.



# Dynamic Proxy

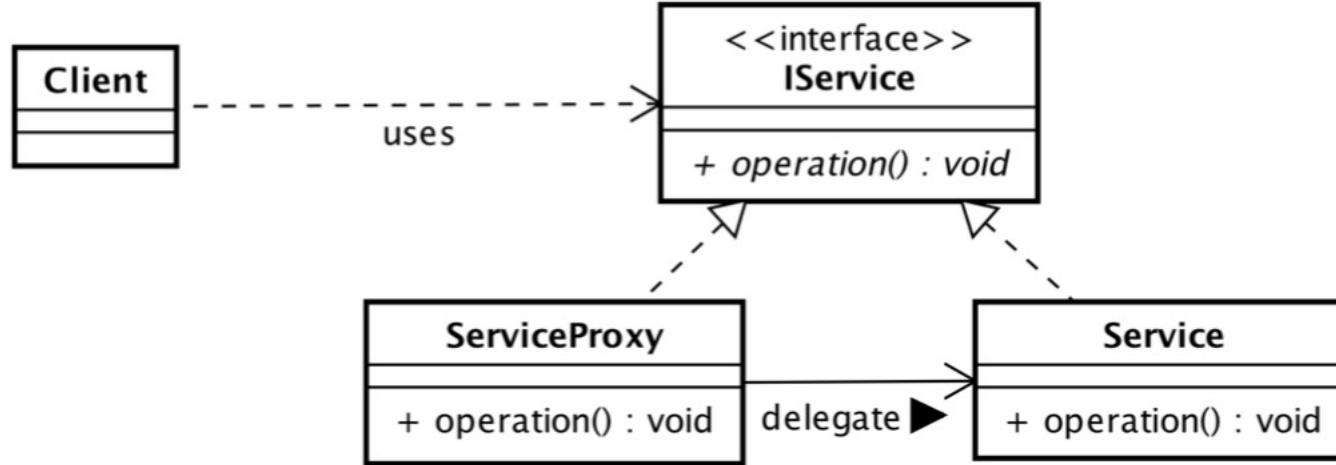
# Motivation und Kurzbeschreibung Dynamic Proxy

---



- dynamische Proxys als spezielle Form von Proxys und damit auch ein Stellvertreter für ein anderes Objekt
- Recp: Nutzer wenden sich immer an den Proxy und nicht direkt an das vertretene Objekt.
- Recp: Proxy kann kontrollieren bzw. steuern, wann, wie und von wem auf das vertretene Objekt zugegriffen wird.
- Problem: Der Proxy muss all diejenigen öffentlichen Methoden implementieren, die das ursprüngliche Objekt in seiner Schnittstelle definiert. Das kann ziemlich aufwendig werden.

# UML des statischen Proxy + Motivation Dynamic Proxy



Wenn man einen Proxy selbst implementiert, sind schnell relativ viele Methoden zu schreiben. Man spricht dann auch von einem **statischen Proxy**, da dieser bereits **während der Kompilierung** vorliegt. Manchmal ist es aber wünschenswert, einen **Proxy** zu einem bestimmten Interface **erst zur Laufzeit zu erstellen**. Einen solchen nennt man dann **dynamischen Proxy**.

# Dynamic Proxy im JDK

---



- Das JDK bietet im Package `java.lang.reflect` die **Klasse Proxy** zur **Konstruktion dynamischer Proxys**
- Das **Interface InvocationHandler**, um **dynamischen Proxy-Funktionalität zu implementieren**
- Zentrale Methode  
`Object invoke(Object proxy, Method method, Object[] args) throws Throwable;`
- **Dynamischer Proxy** wird erst zur **Laufzeit** (daher dynamisch) per Aufruf von `Proxy.newProxyInstance()` **erzeugt**
- Basiert auf der **Angabe zu erfüllender Interfaces und eines InvocationHandler**

# Anwendungsbeispiel



```
public class PerformanceMeasureInvocationHandler implements InvocationHandler
{
    private final IService service;

    public PerformanceMeasureInvocationHandler(final IService service)
    {
        this.service = service;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
    {
        final long startTime = System.nanoTime();

        Object result = null;
        try
        {
            // Achtung hier nicht versehentlich proxy übergeben
            result = method.invoke(service, args);
        }
        catch (InvocationTargetException ex)
        {
            throw ex.getTargetException();
        }
        printExecTime("calculateSomething", System.nanoTime() - startTime);
        return result;
    }
}
```

# Anwendungsbeispiel



```
public class DynamicProxyExample
{
    public static void main(final String[] args)
    {
        final IService service = createService();
        service.calculateSomething(42);
    }

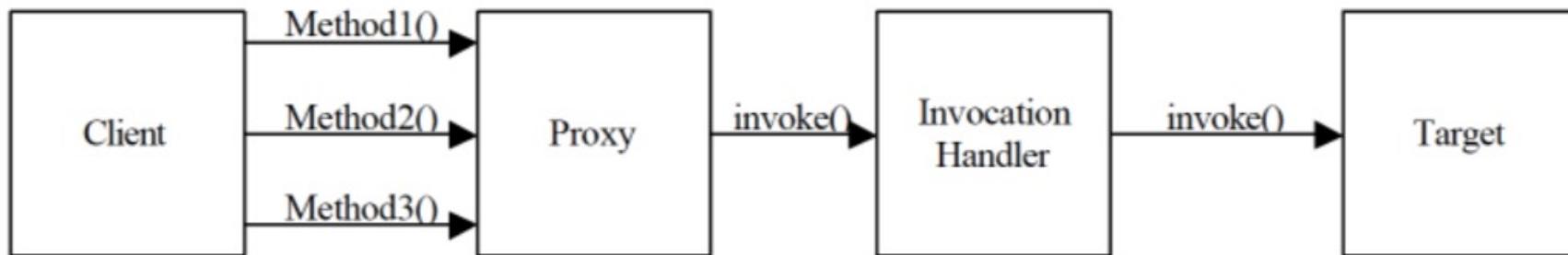
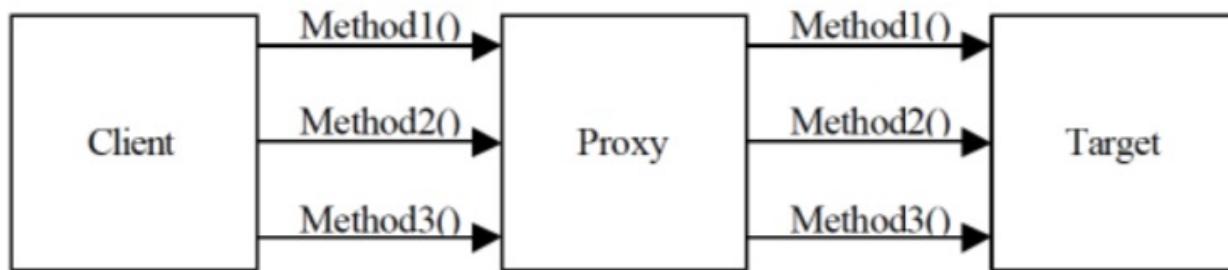
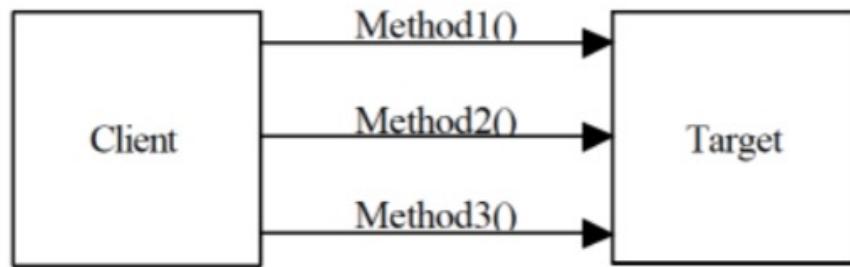
    private static IService createService()
    {
        final IService service = new Service();

        final InvocationHandler handler = new PerformanceMeasureInvocationHandler(service);

        final Class<?>[] proxyInterfaces = { IService.class };
        return (IService) Proxy.newProxyInstance(Service.class.getClassLoader(),
                                                proxyInterfaces, handler);
    }
}
```

```
private static IService createService()
{
    final IService original = new Service();
    return new ServicePerformanceProxy(original);
}
```

# Aufrufe Direkt + Proxy + Dynamic Proxy



## Anwendungsbeispiel II



```
public class LoggingInvocationHandler implements InvocationHandler
{
    private Object target;

    public LoggingInvocationHandler(final Object target)
    {
        this.target = target;
    }

    @Override
    public Object invoke(final Object proxy, final Method mmethod,
                        final Object[] args) throws Throwable
    {
        System.out.println("Invoking " + mmethod.getName() + " " +
                           Arrays.toString(args));
        return mmethod.invoke(target, args);
    }
}
```

## Anwendungsbeispiel II: Dynamic Proxy in Kombination



## Fazit zu dynamischen Proxys



- **bestehende Klassen lassen sich mit dynamischen Proxys ohne viel Mühe um Funktionalität erweitern**
- **Vor allem Querschnittsfunktionalitäten sind ideale Kandidaten, die mit dynamischen Proxys realisiert werden können, etwa**
  - Performance-Messung,
  - Logging usw.
- **Zudem lassen sich dynamische Proxys einfach kombinieren.**
- **Überlegen Sie einmal, wie viel Sourcecode Sie schon für die hier gezeigten beiden Proxys hätten in Ihren Applikationsklassen schreiben müssen – besser noch, man kann die Funktionalität nahezu überall wiederverwenden.**



---

## Exercises Part 4-I

<https://github.com/Michaeli71/DesignPatterns.git>



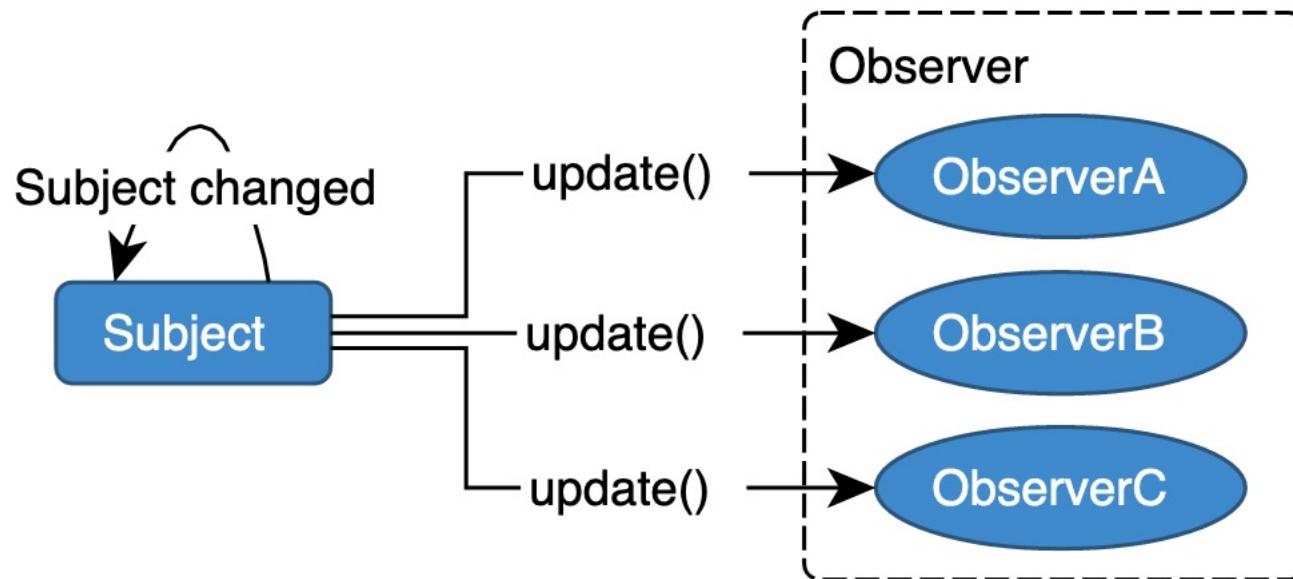


# Beobachter (Observer)

# Beobachter (Observer)



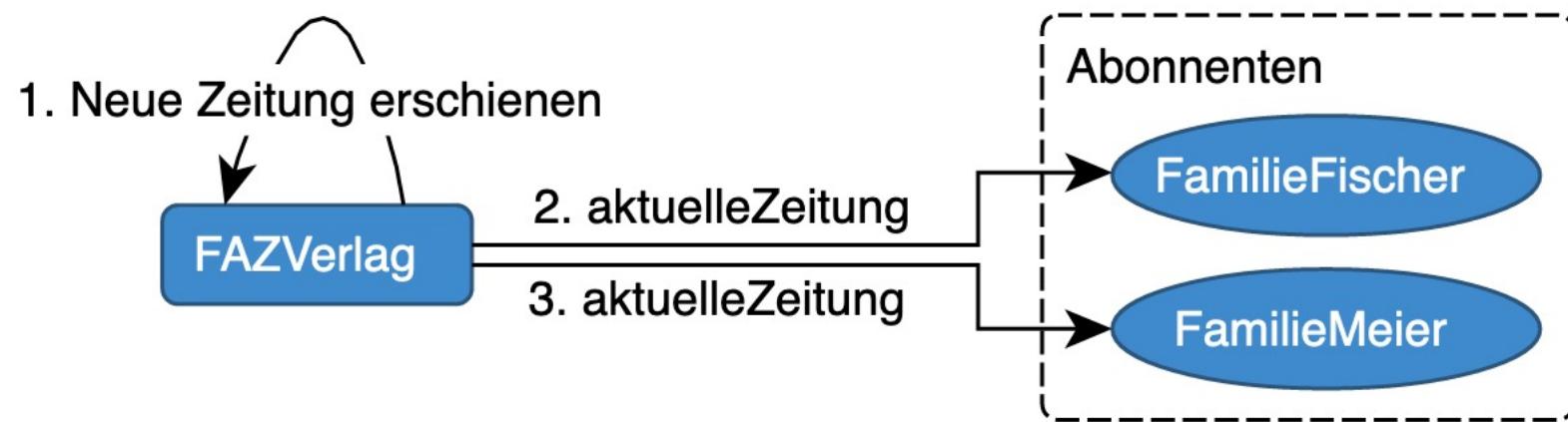
- Zustandsänderungen eines **Objekts (Subjekt)** durch andere interessierte Objekte (**Beobachter**) **beobachten**
- ohne dass sich ein **Objekt** und seine **Beobachter direkt kennen**
- **Beobachter melden** sich dazu als **Interessenten** bei einem Objekt an und »**abonnieren**« damit **Informationen** über Veränderungen, die ihnen das Objekt in der Folge mitteilt.
- Sind Interessenten nicht mehr an Änderungen interessiert, so können sie sich abmelden.



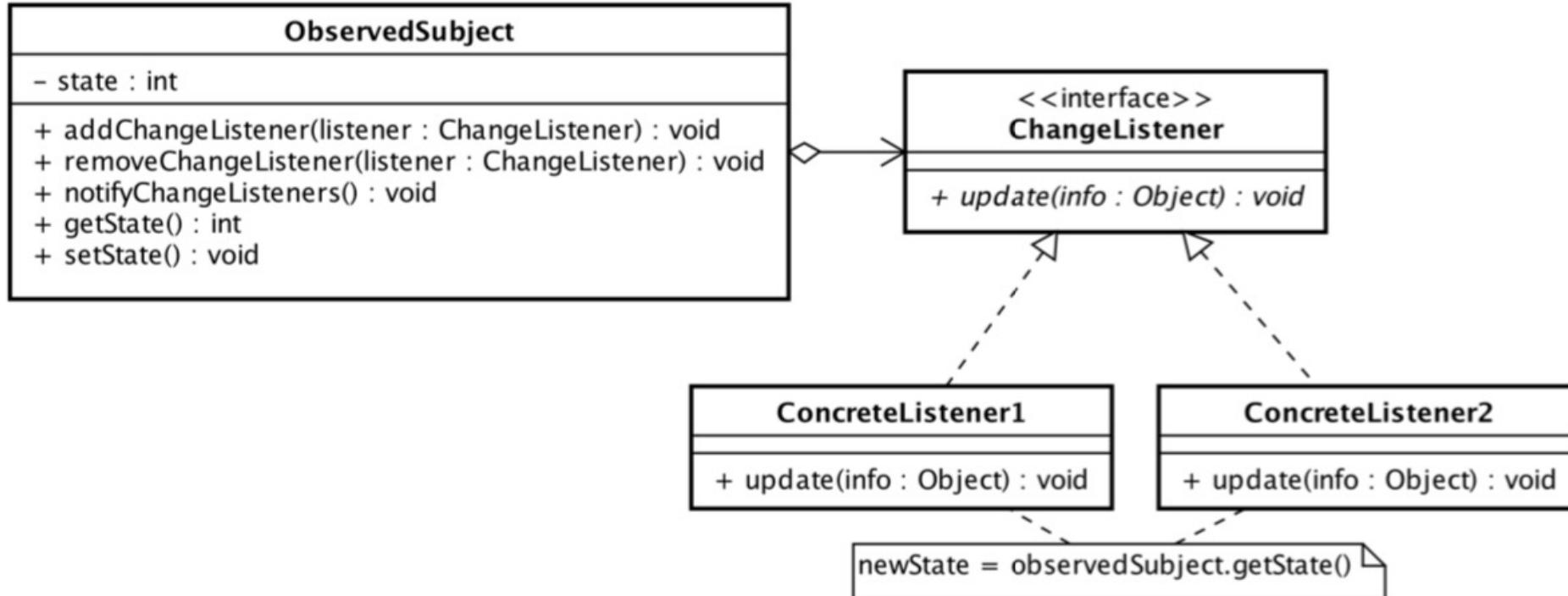
# Beobachter (Observer)



- zugrunde liegende Idee mit dem Abonnement einer Zeitschrift vergleichen. Hat man sich als Abonnent registriert, so erhält man regelmäßig die neuesten Ausgaben, ohne ständig am Kiosk nachschauen zu müssen.

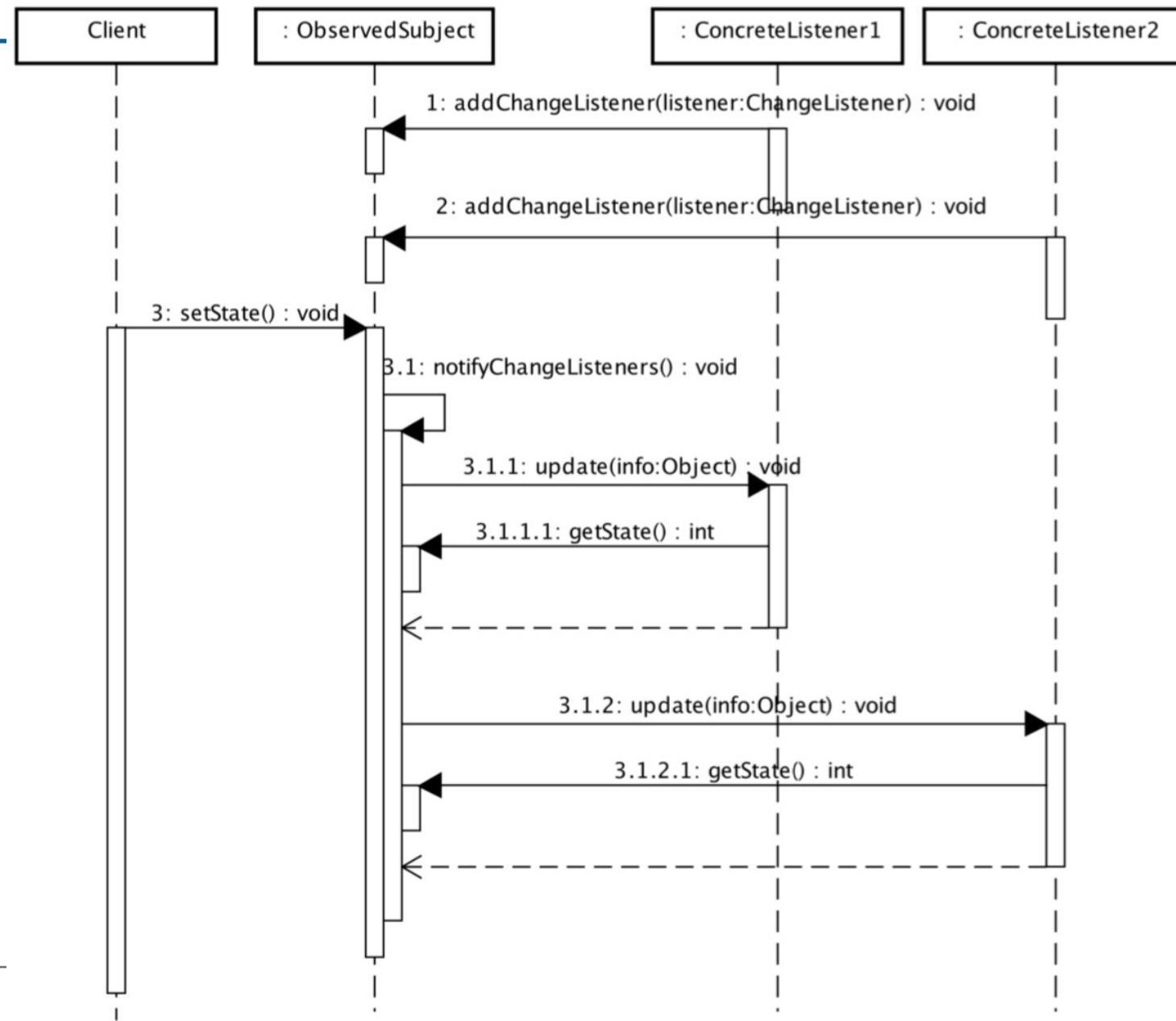


- Ereignisbehandlungen in grafischen Oberflächen basieren in der Regel auf diesem Prinzip – häufig durch Einsatz einer Model-View-Controller-Architektur
- Dieses Muster ist auch unter dem Namen **Publisher/Subscriber** oder **Listener** bekannt.



- Sobald sich der **relevante Zustand** des ObservedSubjects **verändert**, löst dieses über die Methode `notifyChangeListeners()` eine **Benachrichtigungsmeldung** an jeden angemeldeten **Beobachter** vom Typ `ChangeListener` aus.

# Beobachter (Observer)



# Varianten der Information über Zustandsänderungen

---



Um Beobachter über Zustandsänderungen zu informieren, wird eine `update()`-Methode aufgerufen. Dabei gibt es zwei mögliche Varianten zur Übermittlung benötigter Zustandsdaten:

- **Pull** – die `update()`-Methode dient **lediglich als Hinweis**, dass eine **Änderung stattgefunden hat**. Jeder einzelne **Beobachter holt** sich **zur Reaktion** auf die Änderung **benötigte Zustandsdaten** beim beobachteten Subjekt **selbst ab**.
- **Push** – Bei der Push-Variante werden der `update()`-Methode **alle benötigten (relevanten) Zustandsinformationen** als Parameter **mitgegeben**. Dadurch werden **keine Rückrufaktionen** durch die Beobachter **erforderlich**.

# Verzögerung durch Bearbeitung der Listener

---



Bei der **Benachrichtigung** der Beobachter besteht das Problem, dass es sich beim Aufruf der Methode `update()` um **synchrone Aufrufe** an die Beobachter handelt. Das bedeutet auch, dass eine **weitere Verarbeitung** durch Aufruf der `notifyChangeListeners()`-Methode **so lange blockiert** wird, bis alle Beobachter informiert wurden. Dies kann unter Umständen **einige Zeit** dauern, wenn einer oder mehrere Beobachter lang andauernde `update()`-Methoden realisieren.

Dafür gibt es zwei Möglichkeiten zur Lösung:

1. Man kann die gesamte Benachrichtigung asynchron in einem eigenen Thread ablaufen lassen oder
2. jeden einzelnen `update()`-Aufruf asynchron ausführen.

Beides führt aber schnell zu weiteren Synchronisationsproblemen, weil es zu konkurrierenden Lesezugriffen kommt, die dann eventuell auf bereits von anderen Threads aktualisierte Daten zugreifen.

# Benachrichtigungsmechanismus bei Multithreading



Im Fall einer Zustandsänderung sollen alle angemeldeten Beobachter informiert werden. Dies klingt viel einfacher, als es tatsächlich ist.

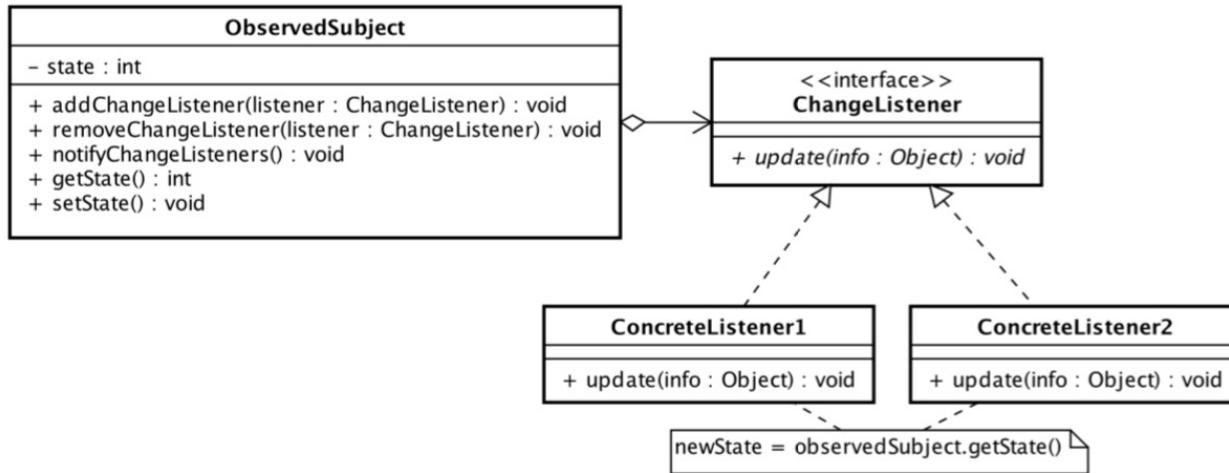
Betrachten wir dazu folgendes Szenario: Zu beliebigen Zeitpunkten können sich Beobachter nebenläufig an- oder abmelden, theoretisch auch während einer laufenden Benachrichtigung. **Wie sorgt man für eine konsistente Beobachterliste bei Multithreading-Zugriffen? Um Probleme zu verhindern, muss man alle Zugriffe auf die Beobachterliste synchronisieren. Das bedeutet allerdings auch, dass Beobachter sich während eines Benachrichtigungsvorgangs weder an- noch abmelden können**, wenn die Realisierung synchronisiert wie in der folgenden Methode erfolgt:

```
private synchronized void notifyChangeListeners()
{
    final Iterator<ChangeListener> it = listeners.iterator();
    while (it.hasNext())
    {
        final ChangeListener listener = it.next();

        listener.update(this);
    }
}
```

Einsatz von `CopyOnWriteArrayList<ChangeListener>` zur Speicherung der Beobachter

# Mythos lose Kopplung



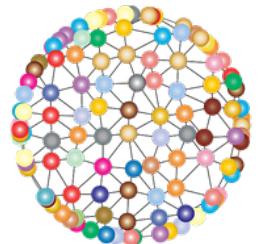
**Bei Realisierungen analog zum gezeigten Beispiel wird oftmals keine lose Kopplung erreicht. Schauen wir auf die Gründe.**

1. Es gibt eine Abhängigkeit durch die Speicherung von Referenzen der einzelnen `ChangeListener` im beobachteten Subjekt. Da dies jedoch in der Regel in Form eines Interface, also gekapselt, erfolgt, ist das meistens nicht störend.
2. Es kommt zu einer Kopplung von Beobachtern in Richtung Subjekt. Diese ist in der Regel unangenehmer, da Beobachter häufig eine Referenz auf den konkreten Typ des Subjekts nutzen, um nach Änderungsmitteilungen auf den Zustand des Subjekts zuzugreifen. **Es kommt deshalb zu einer starken, unidirektionalen Kopplung zwischen Beobachter und Subjekt.** Ein Beobachter kann dadurch sämtliche Methoden der Schnittstelle seines Subjekts aufrufen.



---

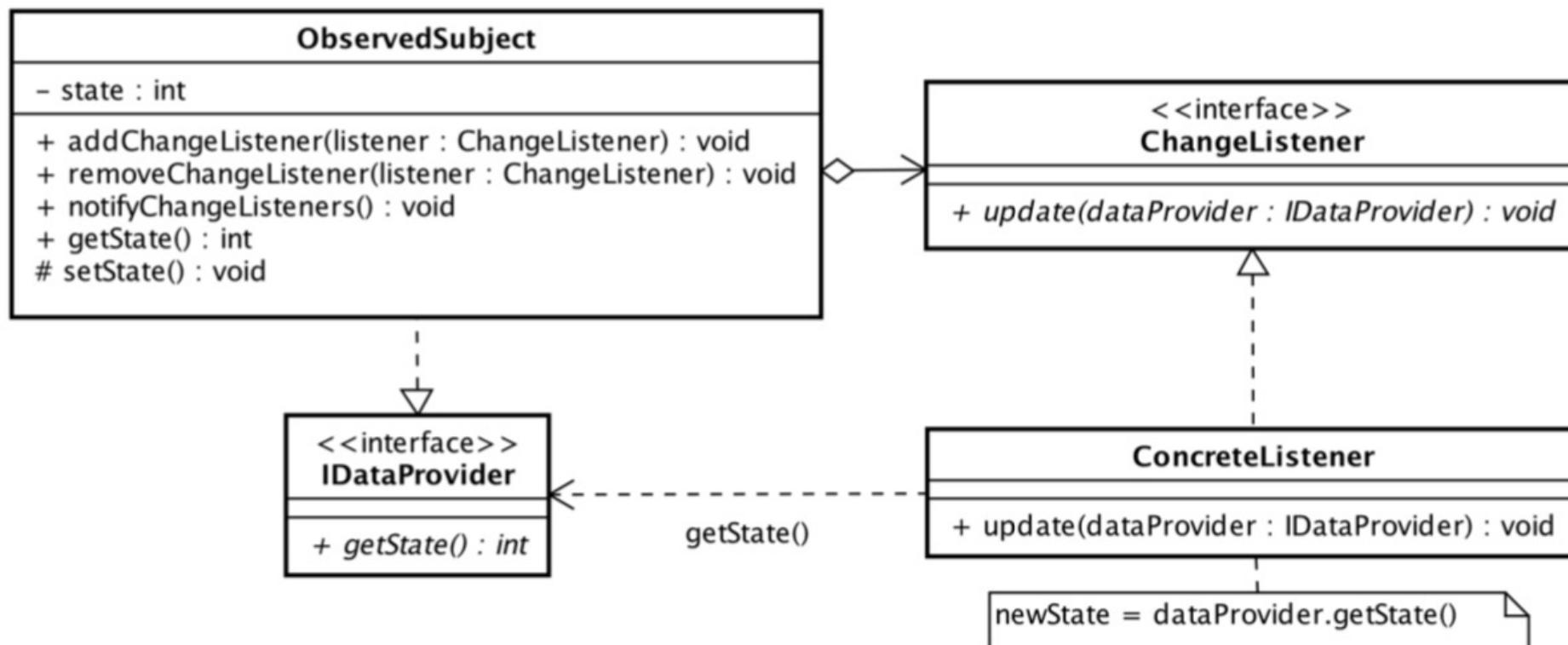
Wie macht man es  
besser?



# Lose Kopplung durch Interface



- Eine Verbesserung erreicht man durch Einsatz eines Interface **IDataProvider**, das vom Subjekt zu erfüllen ist und lediglich Zugriff auf die für Beobachter relevanten Daten erlaubt: Nutzen die Beobachter nur noch Referenzen auf dieses Interface, so löst man die Kopplung. Nachfolgende Abbildung eine mögliche Umsetzung.

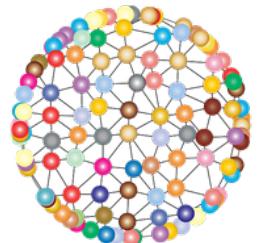


## Varianten: Beobachtung mehrerer Subjekte

---



- Es ist möglich, dass sich ein Beobachter bei mehreren Subjekten als Interessent registriert.
- Zur Unterscheidung der Quelle der Änderungen ist es daher wichtig, dass eine Referenz auf den Urheber der Änderung in der update()-Methode mitgesendet wird.
- Arbeitet man jedoch nur mit einer simplen update()-Methode ohne einen solchen Quellenparameter, so weiß der Beobachter nicht, bei welchem Subjekt er den aktuellen Zustand nachfragen soll.



# Warum ist Java-Built-In-Observer ungünstig?

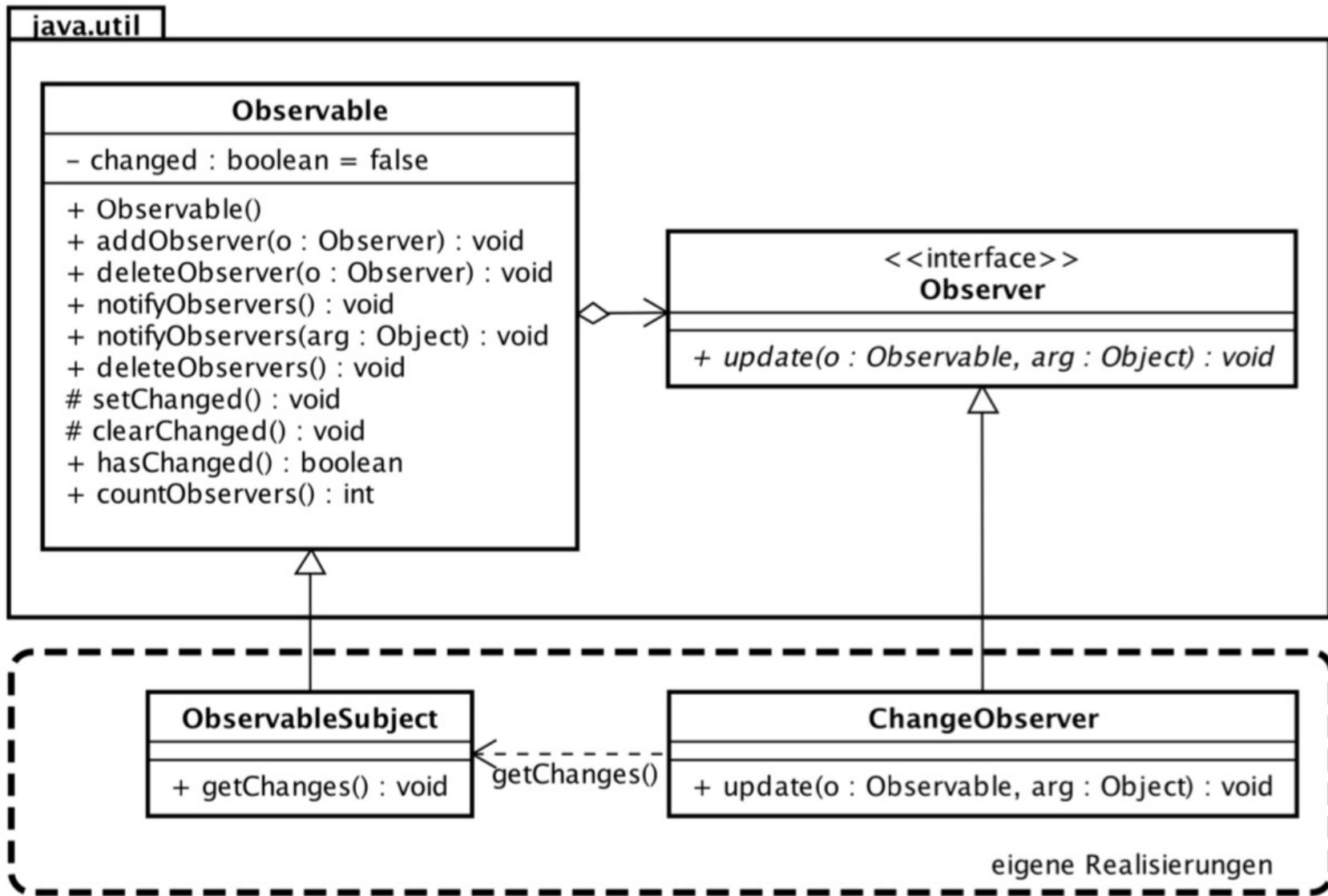
# Warum ist Java-Built-In-Observer ungünstig?

---



- Die Entwickler von Sun haben in das JDK mehrere **BEOBACHTER**-Muster integriert.
- Die Realisierungen durch die Klassen **ActionListener** und **EventListener** sind gelungen.
- Leider kann man das nicht für die Realisierung in Form der Klasse **Observable** und des Interface **Observer** sagen.
- Diese Realisierung verstößt gegen verschiedene OO-Gedanken:
  - Beobachtbar zu sein ist eine Rolle und keine Erweiterung einer Basisklasse.
  - Merkwürdigerweise erfordert die JDK-Realisierung aber hier eine technisch bedingte Ableitung von der Basisklasse **Observable** (Implementierungsvererbung).
  - *Interessanterweise widerspricht zudem die gewählte Namensgebung allen Konventionen im JDK:* Die Endung **-able** wird ansonsten immer für Namen von Interfaces verwendet, hier jedoch zur Benennung einer Klasse. Diese etwas krude Klassenhierarchie ist in folgender Abbildung dargestellt.

# Warum ist Java-Built-In-Observer ungünstig?



# Warum ist Java-Built-In-Observer ungünstig?

---



- **Erschwerend kommt hinzu, dass durch den Zwang, von einer konkreten Klasse abzuleiten, eine Nutzung dieser vorgefertigten Realisierung in einer eigenen Klassenhierarchie schwierig wird:** Besitzt eine eigene Klasse bereits eine Basisklasse, so kann sie nicht mit dem Java-Built-In-Observer genutzt werden.
  - **Weiterhin ist die Methode update() zu unspezifisch.** Erstens besitzt ihr Name wenig Aussagekraft. Zweitens ist **unklar**, welche **Teile** des geänderten **Zustands** als **Parameter** verwendet werden sollten. Für eine solche generische Lösung ist damit der **Datentyp** der **mitzuteilenden Daten unbekannt**. Daher muss in diesem Fall ein Parameter vom Typ Object übergeben werden. Um sinnvoll mit den Daten arbeiten zu können, muss jeder Beobachter daraus die benötigten Informationen zurückgewinnen. Dies erfordert eine **explizite Typprüfung** per instanceof, um den ansonsten nicht typsicheren Cast auf den erwarteten Typ der Zustandsinformation abzusichern.
  - **Aufgrund der geschilderten Nachteile sind die Klasse Observable und das Interface Observer seit JDK 9 als deprecated markiert.**
-

# Verbesserungen

---



- **Statt einer allgemeinen update()-Methode ist es oft sinnvoller, mehrere Methoden anzubieten**, mit denen jeweils **spezifische**, mögliche **Zustandsänderungen** propagiert werden
- Neben **sprechenderen Methodennamen** lassen sich **Änderungen wesentlich besser nachvollziehen** als bei einer update()-Methode.
- Beispielhaft verdeutlicht dies das folgende Interface IModelListener:

```
public interface IModelListener
{
    public void imageElementsChanged(final List<AbstractGraphicsElement> images);
    public void pdfElementsChanged(final List<AbstractGraphicsElement> pdfs);

    public void nameChanged(final String newName);
}
```

## Verbesserungen

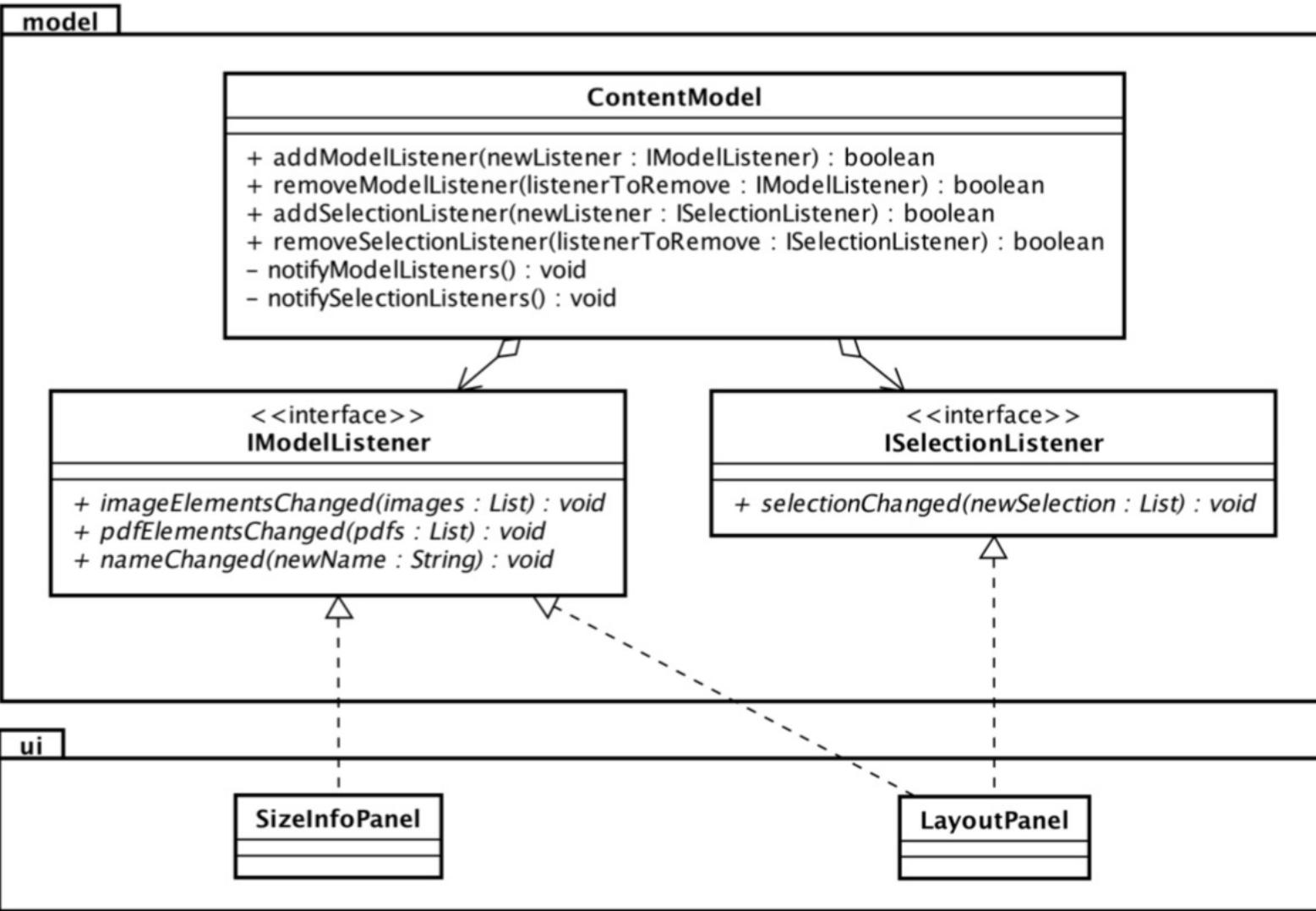
---



- Als Erweiterung kann man **statt eines allgemeinen Beobachters mehrere spezialisierte Beobachter** für ausgewählte **Teilaspekte des Objektzustands** definieren.
- Eine Änderung der Selektion lässt sich etwa durch einen speziellen **ISelectionListener** und dessen **selectionChanged()**-Methode behandeln:

```
public interface ISelectionListener
{
    void selectionChanged(final List newSelection);
}
```

# Verbesserungen



Für die beiden dargestellten Beobachter werden aufgrund der Push-Variante keine Referenzen und Zugriffe auf das Datenmodell ContentModel benötigt. Durch die Trennung der Interfaces für verschiedene Beobachter ist es sogar möglich, spezifische Informationen zu kommunizieren.

## Bewertung Beobachter (Observer)



- + **Lose Kopplung** – Im Idealfall kennen sich Subjekt und registrierte Beobachter nicht direkt, wenn eine Kapselung über Interfaces erfolgt. Durch die Trennung von Subjekt und Beobachtern sind Erweiterungen in beiden Klassen problemlos möglich, solange die Schnittstelle unverändert bleibt.
- + **Flexibilität** – Die Anzahl der Interessenten kann zur Laufzeit verändert werden. Hinzukommende oder später entfernte Beobachter beeinflussen andere Beobachter nicht.
- o **Fehlende Reaktion** – Falls einen Beobachter gewisse Meldungen nicht interessieren, kann er diese gegebenenfalls ignorieren: Es bleibt dem Beobachter überlassen, ob und wie er mit Nachrichten umgeht. Im Extremfall kann ein Beobachter die Änderungsmeldungen einfach nicht beachten.
- o **Probleme der Statusänderung** – Das Subjekt weiß nicht, welche Aktionen die Beobachter bei einer Änderungsmeldung ausführen. Komplexe Aktionen können die Abarbeitung der Benachrichtigung verzögern oder sogar blockieren.

## Bewertung Beobachter (Observer)

---



- **Komplexität** – Es stellt eine große Herausforderung dar, den Benachrichtigungsvorgang konsistent und Thread-sicher durchzuführen. In der Regel muss man mit gewissen Kompromissen leben, etwa dem, dass Beobachter, die sich während einer Benachrichtigung neu anmelden, erst bei zukünftigen Änderungen informiert werden.
- **Nachvollziehbarkeit** – Beim Einsatz dieses Musters werden oft ziemlich viele, mitunter auch (zu) feingranulare Benachrichtigungen erzeugt, was die Übersicht erschwert. Die Ursachen solch unerwünschter Updates sind dadurch schwierig zu finden.
- **Gefahr von Zyklen** – Werden Beobachter wieder von anderen Klassen beobachtet, so kommt man schnell in ein Benachrichtigungschaos oder endet sogar in einem Zyklus.

### Tipp: Einfluss von Business-Methoden auf das BEOBACHTER-Muster

Das Einführen von Business-Methoden hilft, ein mögliches Benachrichtigungschaos beim Einsatz des BEOBACHTER-Musters zu vermeiden. Wenn lediglich Business-Methoden Zustandsänderungen propagieren, sorgt dies für mehr Klarheit und Transparenz.

# Anti-Pattern Beobachter (Observer)



## Achtung: BEOBACHTER als Anti-Pattern

Das BEOBACHTER-Muster kann schnell zum Anti-Pattern werden, wenn Beobachter ihrerseits selbst ein beobachtetes Subjekt sind. Sind Abhängigkeitsbeziehungen nicht klar definiert, kommt es zu schwer nachvollziehbaren Updates. Eine scheinbar harmlose Zustandsänderung auf einem Subjekt kann eine Kaskade von Updates an Beobachter und deren abhängiger Beobachter zur Folge haben. In einem solchen System können kleinste Änderungen dazu führen, dass das System nicht mehr konvergiert. Nur minimale Variationen der Anfangsparameter führen möglicherweise zu einem komplett anderen Systemverhalten.

Es kann im Extremfall schnell eine Lawine von nicht mehr beherrschbaren Ereignissen ausgelöst werden, die im schlimmsten Fall zu Endlosschleifen führen, zumindest jedoch eine häufige Ursache für eine schlechte Performance sind.



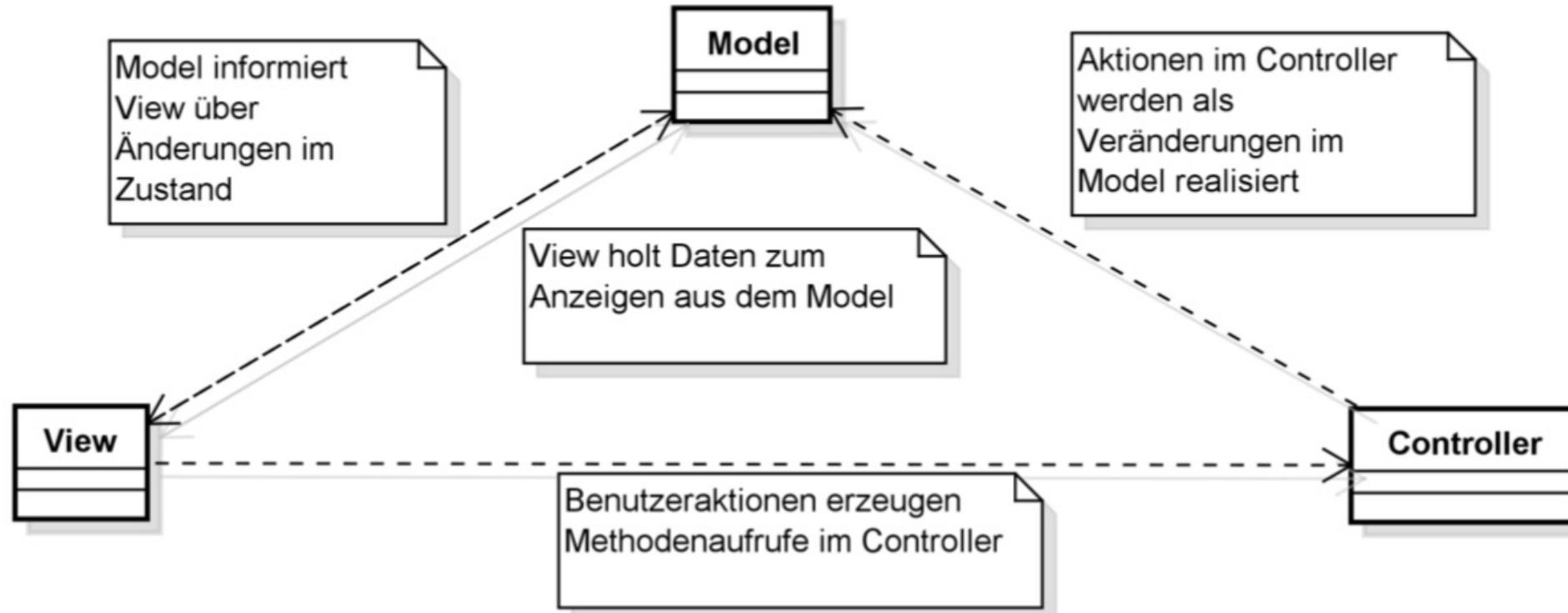
# MVC

# Motivation und Kurzbeschreibung

---



- Die sogenannte Model-View-Controller-Architektur teilt ein zu modellierendes System in die drei Bestandteile **Daten (Model)**, **Darstellung (View)** und **Business- bzw. Kontrolllogik (Controller)**.
- **Jeder Teil wird möglichst unabhängig** von den anderen realisiert und oftmals über Schnittstellen gekapselt.
- Die Idee, Daten und ihre Repräsentation zu trennen, ist insofern sinnvoll, als dass sich **mehrere Darstellungsformen** anbieten lassen.
- Weiterhin erreicht man durch diese Trennung eine klarere Struktur. Die **Trennung** bedingt allerdings, dass **Änderungen** im Modell an die Views **kommuniziert** werden müssen. Dafür kann das **BEOBACHTER-Muster** eingesetzt werden.



## Bewertung MVC



- + **Lose Kopplung** – Kommunizieren die einzelnen **Komponenten** ausschließlich über **Interfaces**, so sind alle Komponenten **nur lose miteinander** verbunden.
- + **Trennung von Zuständigkeiten**– Jede Komponente repräsentiert einen speziellen Aspekt des Gesamtsystems. Es findet eine Trennung von Zuständigkeiten statt.
- + **Flexibilität** – **Mehrere Darstellungen** oder **Ansichten** (z. B. **Balkendiagramm** und **Tortendiagramm**) auf dieselben **Daten** lassen sich **leicht realisieren**. Die einzelnen **Komponenten** können als eine **Klasse** oder als **mehrere Klassen** oder sogar in Form eines **eigenständigen Programms** realisiert sein.
- + **Konsistente Darstellung** – Durch die **zentrale Datenhaltung** in einem Modell und die Änderungsbenachrichtigungen mithilfe des **BEOBACHTER**-Musters können alle Darstellungen (**Views**) **konsistente Daten** anzeigen (sofern die Views korrekt auf Änderungsmeldungen reagieren).

## Bewertung MVC

---



- o **Overengineering** – In einer Auslegung der Objektorientierung sollte ein Objekt alle seine Belange regeln. Nicht immer ist eine Aufteilung wie bei der Model-View- Controller-Architektur sinnvoll und notwendig. **Wie bei allen Techniken sollte man auch hier darauf achten, dass man nicht mit Kanonen auf Spatzen schießt.**
- **Erhöhte Komplexität** – Die Anzahl der benötigten Klassen führt zu etwas höherer Komplexität.
- **Gefahr vieler Änderungsmeldungen** – Werden viele feingranulare Änderungen im Modell vorgenommen und sofort an die Ansichten kommuniziert, so kommt es zu diversen Aktualisierungen der Ansichten. Das kann unerwünscht sein und sich negativ auf die Performance auswirken.
- **Gefahr enger Kopplung** – Werden die Komponenten nicht durch Schnittstellen voneinander abgekoppelt, so führt man ungewollt direkte Abhängigkeiten ein und die Wiederverwendbarkeit der Einzelkomponenten sinkt.

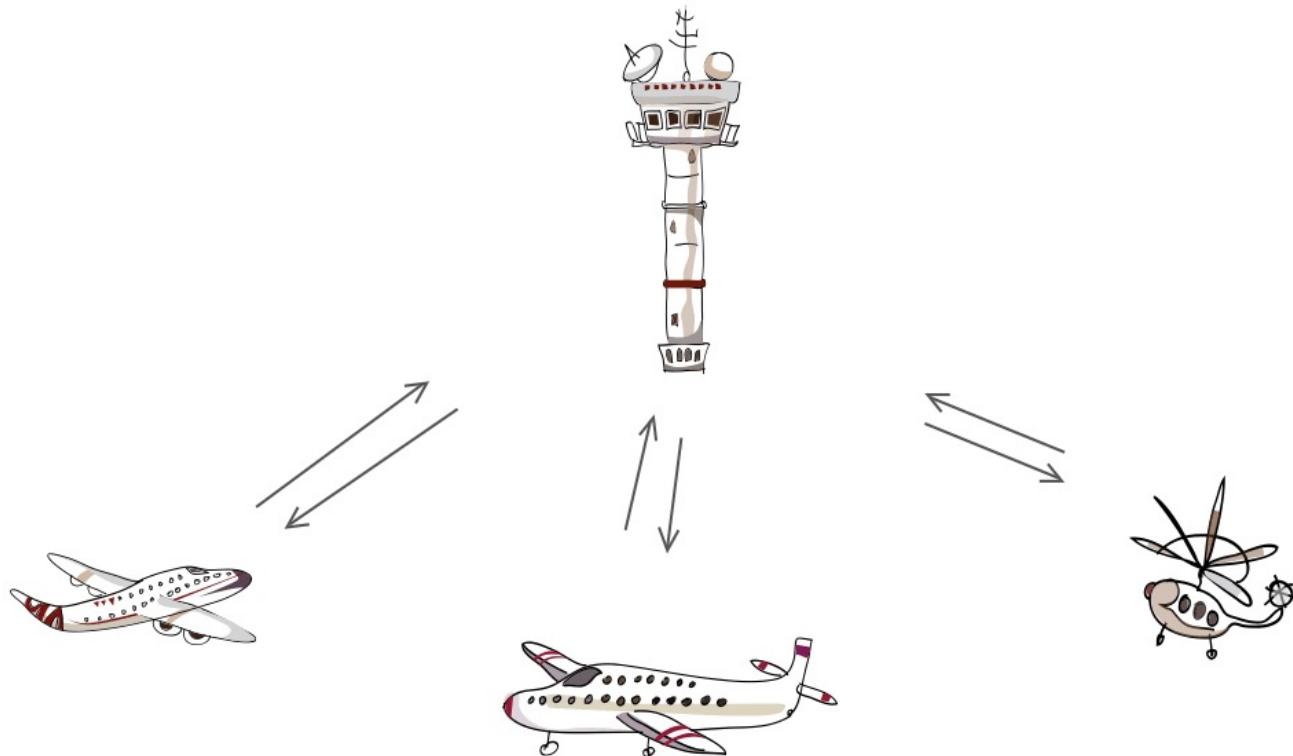


# Vermittler (Mediator)

# Motivation und Kurzbeschreibung Vermittler



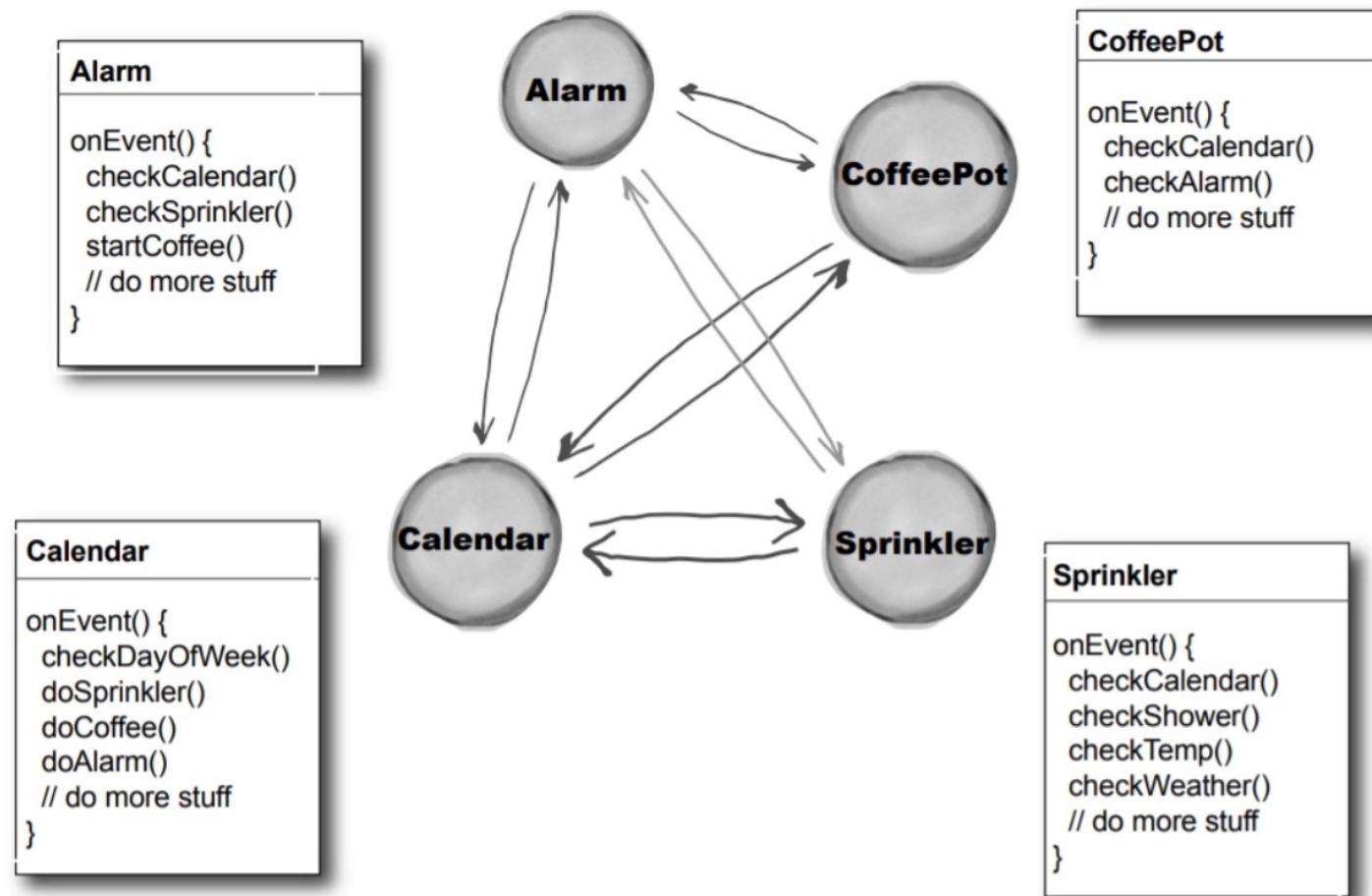
- Der **Vermittler** (Mediator) steuert die **Kommunikation** zwischen verschiedenen Objekten.
- Der **Vermittler** vermeidet Kommunikationschaos und Spaghetti-Code.



# Beispiel Vermittler



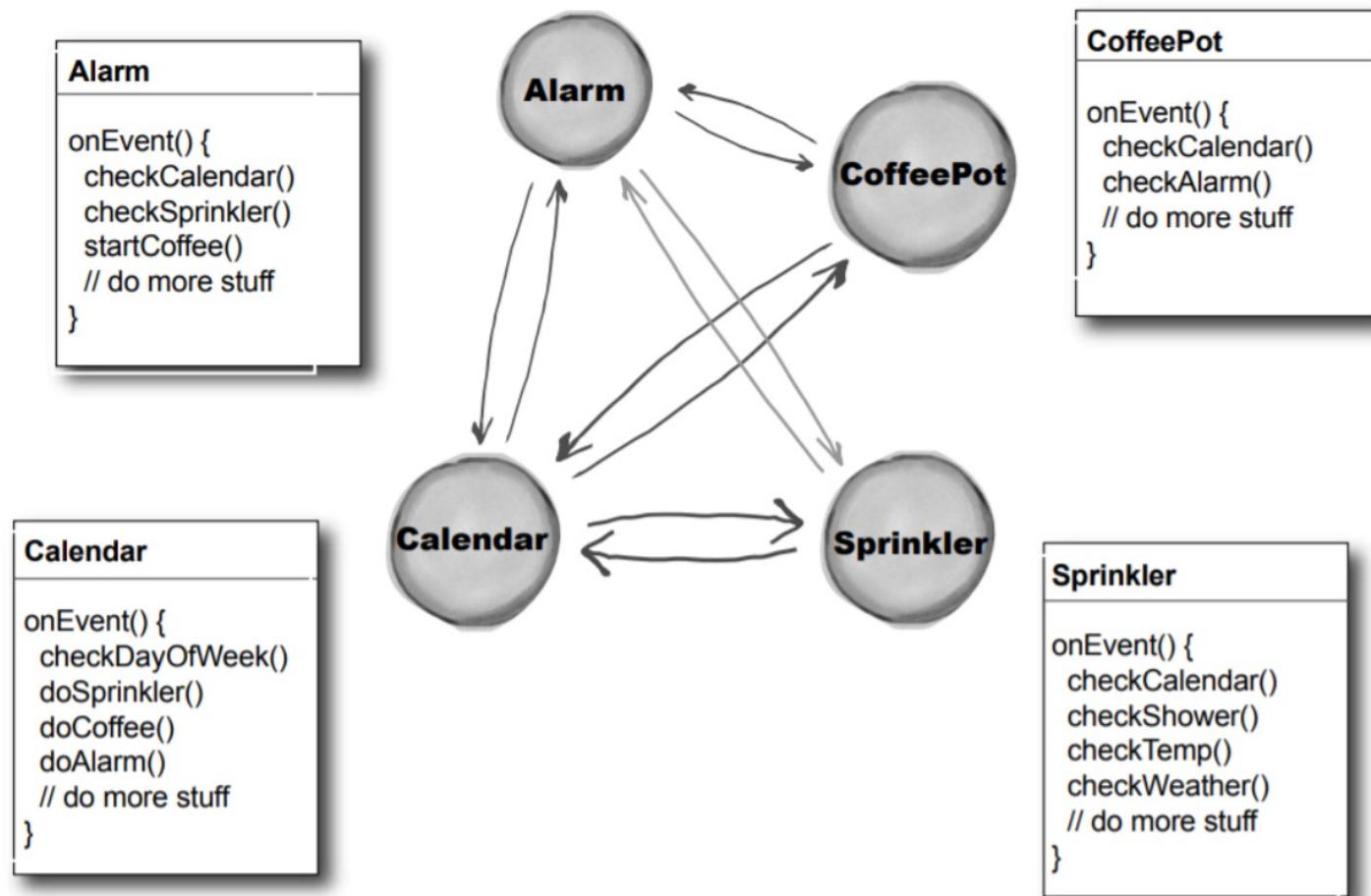
- In einem Smart Home gibt es intelligente Geräte, von denen jedes seine spezifischen Aufgaben erfüllen kann. Wenn die Zeit gekommen ist, informiert der Alarm die Kaffee-maschine, um mit dem Brühen des Kaffees zu beginnen. Es gibt komplexere Bedingungen, z. B. an Wochenenden weicht die Vorgehensweise ab.



# Beispiel Vermittler



- Problem: In diesem Szenario weiß jedes einzelne Objekt genau über die anderen Objekte Bescheid, um mit ihnen zusammenarbeiten zu können. Sie kennen mehr als nötig, oder in der objektorientierten Ausdrucksweise, sie sind eng gekoppelt.





**Müssen die  
Kommunikationspartner  
wirklich voneinander  
wissen?**

## Beispiel

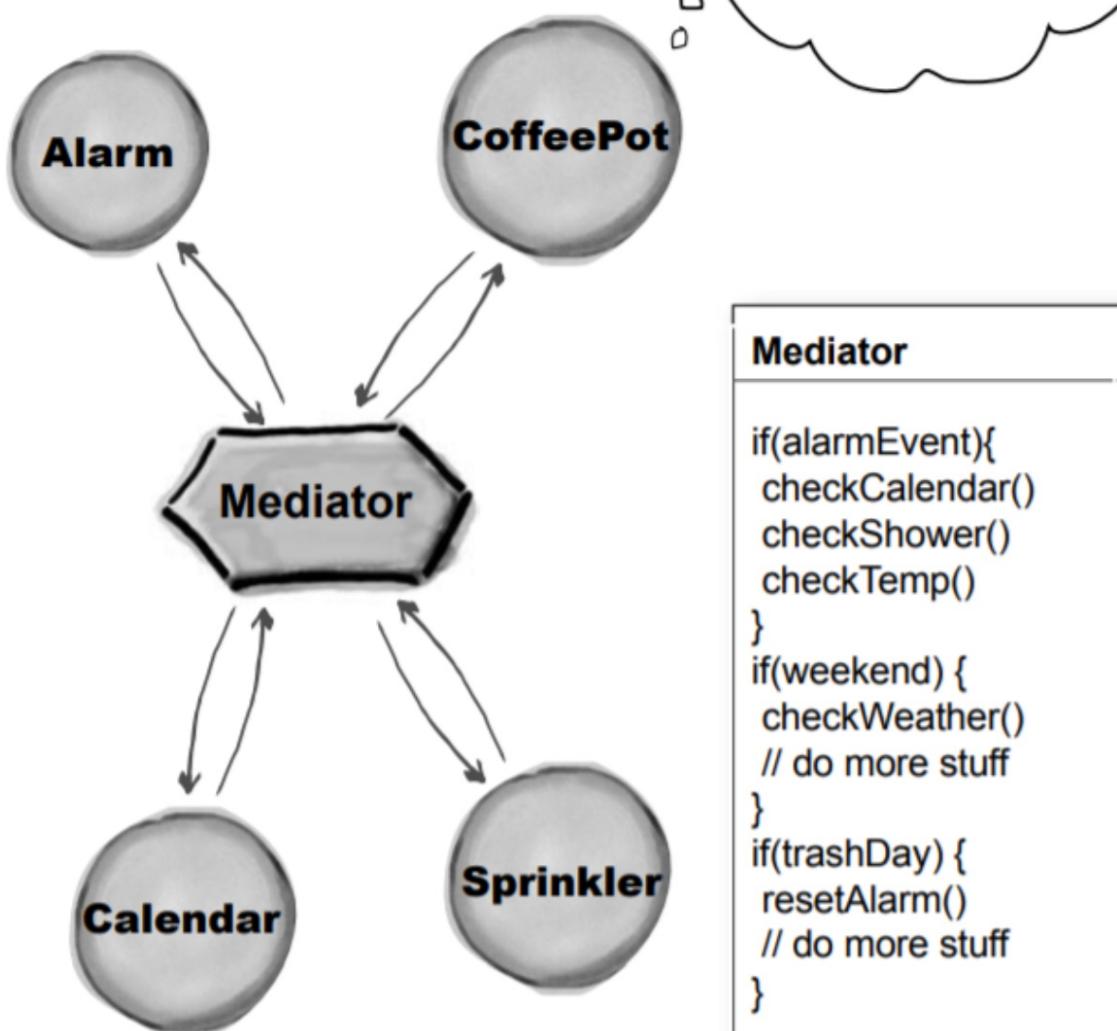
---



- **Vermittler verhindert, dass Objekte explizit aufeinander verweisen,**
- **Vermittler regelt Interaktionen zwischen Objekten**
  - Wenn ein Objekt etwas von einem anderen Objekt benötigt, informiert es einfach den Vermittler und dieser erledigt den Auftrag auf seine eigene Weise. Das aufrufende Objekt kümmert sich nicht darum, wie die Anfrage behandelt wird.
  - Wenn sich der Zustand eines der Mitglieder geändert hat, kann es den Vermittler informieren, um Aktionen auszulösen. Wiederum kümmert sich das aufrufende Objekt nicht darum, ob das weitere Aktionen auslöst, sondern dies ist Aufgabe vom Vermittler.
  - Wenn der Vermittler eines der Objekte auffordert, eine bestimmte Funktion auszuführen (z.B. Kaffee kochen), wird das Objekt dies tun, aber es kümmert sich nicht darum, wer darum gebeten hat und warum.

## Beispiel

---



### Mediator

```
if(alarmEvent){  
    checkCalendar()  
    checkShower()  
    checkTemp()  
}  
if(weekend) {  
    checkWeather()  
    // do more stuff  
}  
if(trashDay) {  
    resetAlarm()  
    // do more stuff  
}
```



## Bewertung Mediator



- + **Vereinfachung** – Die **Kommunikation** ist deutlich einfacher **nachvollziehbar**, da diese nicht mehr im Spinnennetz abläuft, sondern durch den Vermittler geregelt.
- + **Bessere Verständlichkeit** – Die **Kommunikation** ist viel besser **verständlich**, da diese nicht mehr im Spinnennetz abläuft, sondern durch den Vermittler geregelt.
- o **Komplexität & zusätzliche Komponente** – Der **Vermittler** ist eine zusätzliche **Komponente** und die Kommunikation **erfordert etwas Extraaufwand und Sourcecode**.

# Mediator im Kontext

---



- Bei den Mustern **Chain of Responsibility** (Zuständigkeitskette), **Command** (Befehl), **Mediator** (Vermittler) und **Observer** (Beobachter) geht es darum, **Sender und Empfänger zu entkoppeln**
    - **Chain of Responsibility** leitet eine **Anforderung** entlang einer **Kette** von potenziellen **Empfängern** weiter.
    - **Command** spezifiziert eine **Sender-Empfänger-Verbindung**.
    - Beim **Mediator** regelt die **Kommunikation** zwischen **Sender** und **Empfänger(n)**.
    - **Observer** definiert eine **Schnittstelle**, die es erlaubt, mehrere **Änderungsempfänger** zu registrieren und zu informieren (und zur Laufzeit sogar zu ändern).
  - **Mediator** und **Observer** sind konkurrierend.
    - **Observer** kommuniziert **Änderungen** an einem **Subjekt** an **Interessenten**
    - **Mediator** die **Kommunikation** zwischen anderen **Objekten** steuert und regelt (die **Details versteckt**).
-



---

# Design Pattern Workshop

- Chain of Responsibility
- Interpreter
- Blackboard





---

# DEMO

**SimpleInterpreterExample.java**  
**InterpreterExample.java**



---

## Exercises Part 4-II

<https://github.com/Michaeli71/DesignPatterns.git>





# PART 5: Design Patterns mit Java 8



# Builder

# Builder herkömmlich als DTO mit Explaining Methods



```
public static final class PizzaBuilder
{
    // Defaultwerte, gelten wenn keine korrespondierende Methode aufgerufen wird
    boolean mitSalami          = false;
    boolean mitExtraSardellen = false;
    String info                = "";
    Size size                  = Size.MEDIUM;

    public PizzaBuilder mitSalami()
    {
        this.mitSalami = true;
        return this;
    }

    public PizzaBuilder small()
    {
        this.size = Size.SMALL;
        return this;
    }

    public Pizza create()
    {
        return new Pizza(this);
    }
}
```

...

# Builder im Einsatz



- **Einsatz**

```
public static void main(final String[] args)
{
    final PizzaBuilder builder = new PizzaBuilder();
    builder.mitExtraSardellen();
    System.out.println("Normale Pizza mit extra Sardellen:\n" + builder.create());

    final PizzaBuilder builder2 = new PizzaBuilder();
    builder2.mitSalami().small().bitteBeachten("Ohne Mais!");
    System.out.println("Kleine Salami-Pizza ohne Mais:\n" + builder2.create());
}
```

- **Sehr gute Lesbarkeit und Verständlichkeit**
- **Kompakte Schreibweise möglich**
- **Refactoring-sicher**
- **Optionale Attribute können sinnvoll vorbelegt und verarbeitet werden**

# Builder mit Consumer statt spezifischer Methoden



```
public class PizzaBuilder
{
    enum Size {
        SMALL, MEDIUM, LARGE
    }

    public boolean mitSalami          = false;
    public boolean mitExtraSardellen = false;
    public String info                = "";
    public Size size                 = Size.MEDIUM;

    public PizzaBuilder with(final Consumer<PizzaBuilder> builderFunction)
    {
        builderFunction.accept(this);
        return this;
    }

    public Pizza build()
    {
        return new Pizza(this);
    }
}
```

# Builder mit Consumer statt spezifischer Methoden



- Einsatz mit Lambdas

```
public static void main(String[] args)
{
    Pizza pizza = new PizzaBuilder().
        with($ -> $.info = "Kein Mais").
        with($ -> $.size = Size.MEDIUM).
        with($ -> $.mitSalami = true).build();

    System.out.println(pizza);
}
```

- Für einfache Builder okay
- Lesbarkeit schlechter, Spezialsyntax bei Lambdas
- Problematisch: direkter Zugriff auf Attribute, aber nicht Refactoring-sicher



# Template Method

## Template Method herkömmlich

---



```
public final void templateMethod()
{
    first();
    step1();
    step2();
    step3();
    hook();
    last();
}
```

```
abstract protected void first();
```

```
abstract protected void last();
```

```
void step1()
{
    System.out.println("step1");
}
```

---

```
// ...
```

# Template Method herkömmlich mit Ableitung und Überschreiben



```
public static void main(String[] args)
{
    TemplateMethodExample tme = new TemplateMethodExample()
    {
        @Override
        protected void first()
        {
            System.out.println("FIRST");
        }

        @Override
        protected void last()
        {
            System.out.println("LAST");
        }
    };
    tme.templateMethod();
}
```

FIRST  
step1  
step2  
step3  
LAST

# Template Method mit Lambdas als Parameter

---



```
public final void templateMethod(Runnable first, Runnable last)
{
    first.run();
    step1();
    step2();
    step3();
    hook();
    last.run();
}

void step1()
{
    System.out.println("step1");
}

// ...
```

# Template Method neu mit Lambda als Parameter



```
public static void main(String[] args)
{
    TemplateMethodExample tme = new TemplateMethodExample();

    tme.templateMethod(() -> System.out.println("FIRST"),
                      () -> System.out.println("LAST"));
}
```

FIRST  
step1  
step2  
step3  
LAST



# Factory (Method)

## Factory Method herkömmlich

---



```
// Problems: not easily extensible
public static WebDriver getDriver(DriverType type)
{
    WebDriver driver;

    switch (type)
    {
        case CHROME:
            System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
            driver= new ChromeDriver();
            break;
        case FIREFOX:
            System.setProperty("webdriver.gecko.driver",
                               "/Users/username/Downloads/geckodriver");
            driver = new FirefoxDriver();
            break;
        default:
            throw new IllegalArgumentException("Unsupported driver type");
    }
    return driver;
}
```

# Factory Method mit Supplier

---



```
public enum DriverType
{
    CHROME, FIREFOX, SAFARI, IE;
}

// Factory Method, Variante mit Optional später
public static final WebDriver getDriver(DriverType type)
{
    if (DriverType.CHROME == type)
        return chromeDriverSupplier.get();
    if (DriverType.FIREFOX == type)
        return firefoxDriverSupplier.get();

    throw new IllegalArgumentException("Unsupported driver type");
}
```

## Factory Method mit Supplier

---



```
Supplier<WebDriver> chromeDriverSupplier = () ->
{
    System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");

    return new ChromeDriver();
};

Supplier<WebDriver> firefoxDriverSupplier = () ->
{
    System.setProperty("webdriver.gecko.driver",
                       "/Users/username/Downloads/geckodriver");

    return new FirefoxDriver();
};
```

## Factory mit Supplier (Kür)



```
public static class DriverFactory
{
    private static final Map<DriverType, Supplier<WebDriver>> driverMap =
        new HashMap<>();

    static
    {
        driverMap.put(DriverType.CHROME, chromeDriverSupplier);
        driverMap.put(DriverType.FIREFOX, firefoxDriverSupplier);
    }

    public static final Optional<WebDriver> getDriver(DriverType type)
    {
        return Optional.ofNullable(driverMap.get(type).get());
    }

    void registerDriver(DriverType type, Supplier<WebDriver>> driver)
    {
        driverMap.put(...);
    }
}
```



---

# DEMO

**DriverManagerFactory.java**  
**FactoryMethod\_Modernized.java**



# Strategy

# «Strategy Pattern» mit Predicate und Streams



```
public static void main(String[] args)
{
    List<String> names1 = List.of("Tim", "Tom", "Peter", "Mike", "Michael");
    Stream<String> names2 = Stream.of("Tim", "Tom", "Peter", "Mike", "Michael");
    Stream<String> names3 = Stream.of("Tim", "Tom", "Peter", "Mike", "Michael");

    // Strategy by Definition Predicates
    Predicate<String> startsWithT = str -> str.startsWith("T");
    Predicate<String> moreThan4Chars = str -> str.length() > 4;
    Predicate<String> moreThan6Chars = longerThan(6);

    names1.stream().filter(startsWithT).forEach(System.out::println);
    names2.filter(moreThan4Chars).forEach(System.out::println);
    names3.filter(moreThan6Chars).forEach(System.out::println);
}

// Trick dynamische Parametrierung
static Predicate<String> longerThan(int lowerBound)
{
    return str -> str.length() > lowerBound;
}
```

Tim  
Tom  
Peter  
Michael  
Michael



# Execute Around Pattern

## Typische Ressource, die Freigabe erfordert

---



```
class Resource
{
    public Resource()
    {
        System.out.println("created");
    }

    public void op1() throws IOException
    {
        System.out.println("op1");
    }

    public void op2()
    {
        System.out.println("op2");
    }

    public void close()
    {
        System.out.println("close");
    }
}
```

# Resource

---



- Einsatz oftmals fehlerhaft, entweder ohne `close()` oder Problem im Exception-Fall

```
public static void main(String[] args)
{
    Resource rs = new Resource();
    try
    {
        rs.op1();
        rs.op2();
        // rs.close();
    }
    catch (IOException ioe)
    {
        handleIOException(ioe);
    }
    finally
    {
        // rs.close();
    }
}
```

---

# Resource

---



- Schön wäre es, einen einfachen Aufruf zu machen, der an alles «denkt»

```
Resource rs2 = new Resource();
withClose(rs2, res -> {
    try
    {
        res.op1();
        res.op2();
    }
    catch (IOException ioe)
    {
        handleIOException(ioe);
    }
});
```

# Java Locks

---



- Wie ist es nochmal richtig?

```
final Lock lock = ...  
  
lock.lock();  
try  
{  
    // access the resource  
    // protected by this lock  
}  
finally  
{  
    lock.unlock();  
}
```

```
final Lock lock = ...  
  
try  
{  
    lock.lock();  
    // access the resource  
    // protected by this lock  
}  
finally  
{  
    lock.unlock();  
}
```

- Was ist der Unterschied???
-

# Java Locks



- Wie ist es nochmal richtig?

```
final Lock lock = ...  
  
lock.lock();  
try  
{  
    // access the resource  
    // protected by this lock  
}  
finally  
{  
    lock.unlock();  
}
```

~~```
final Lock lock = ...  
  
try  
{  
    lock.lock();  
  
    // access the resource  
    // protected by this lock  
}  
finally  
{  
    lock.unlock();  
}
```~~

- Im linken Fall wird der Lock nur dann freigegeben, wenn wir ihn erhalten haben, dann aber egal, ob es im Block zu Problemen kam oder nicht
- Im rechten Fall wird der Lock auf jeden Fall freigegeben, auch wenn man ihn nicht erhalten hat!



# Chain of Responsibility



---

# DEMO

[\*\*OldStyleChainExample.java\*\*](#)  
[\*\*NewStyleChainExample.java\*\*](#)

---



---

## Exercises Part 5

<https://github.com/Michaeli71/DesignPatterns.git>





# Questions?



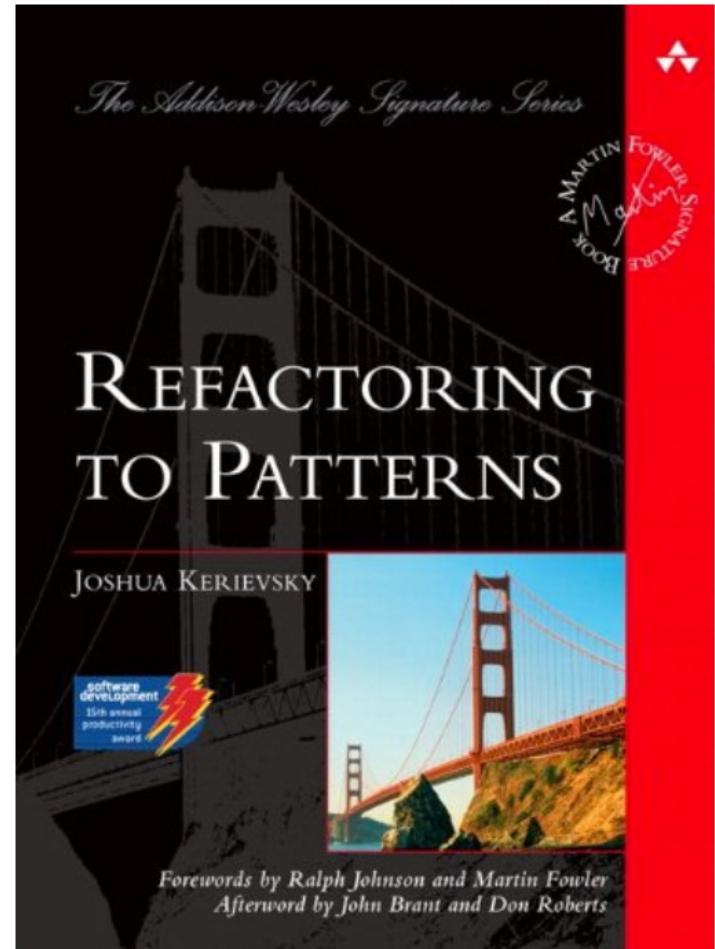
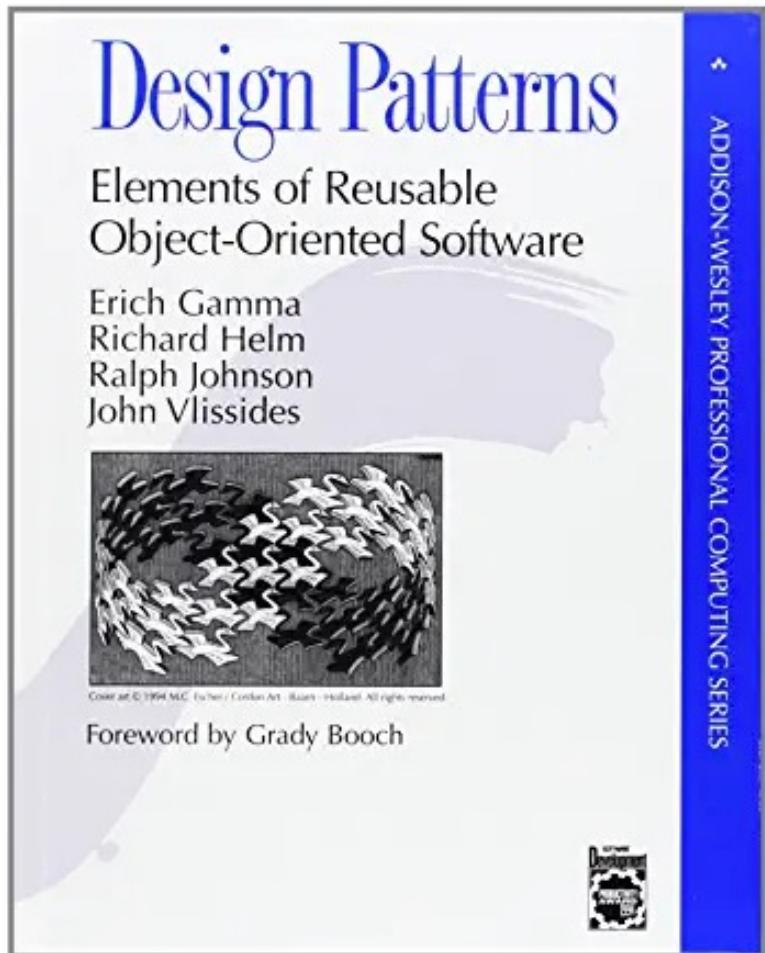
Michael Inden  
**Der Java-Profi:  
Persistenzlösungen  
und REST-Services**

Datenaustauschformate,  
Datenbankentwicklung  
und verteilte Anwendungen

dpunkt.verlag



# Empfehlenswerte Bücher



## Weitere Infos / Quellen

---



- <https://www.philippauer.de/study/se/design-pattern.php>
  - <https://refactoring.guru/design-patterns/java>
  - <https://java-design-patterns.com/patterns/>
-



# Thank You