Entwurf

Prof. Sven Apel

Universität des Saarlandes

Basierend auf Material von Andreas Zeller und Norbert Siegmund

Teil IV

Entwurfsmuster

Entwurfsmuster

In diesem Kapitel werden wir uns mit typischen Einsätzen von objektorientiertem Entwurf beschäftigen – den sogenannten Entwurfsmustern (*Design Patterns*).

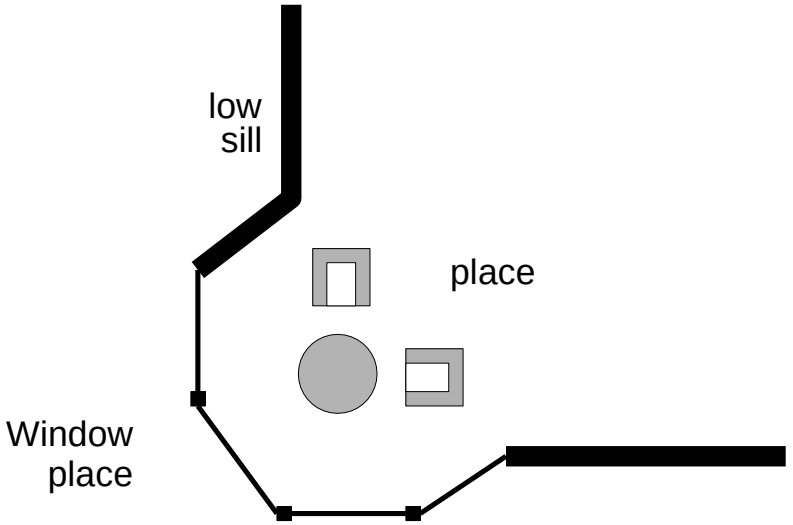
Der Begriff *Entwurfsmuster* wurde durch den Architekten Christopher Alexander geprägt:

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

Ein Muster ist eine *Schablone*, die in vielen verschiedenen Situationen eingesetzt werden kann.

Muster in der Architektur: Window Place

“Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them. In every room where you spend any length of time during the day, make at least one window into a ‘window place’.”



A Pattern Language

Alexander · Ishikawa · Silverstein · Jacobson · Fiksdahl-King · Angel

**

Oxford

A Pattern Language

Towns · Buildings · Construction

Christopher Alexander

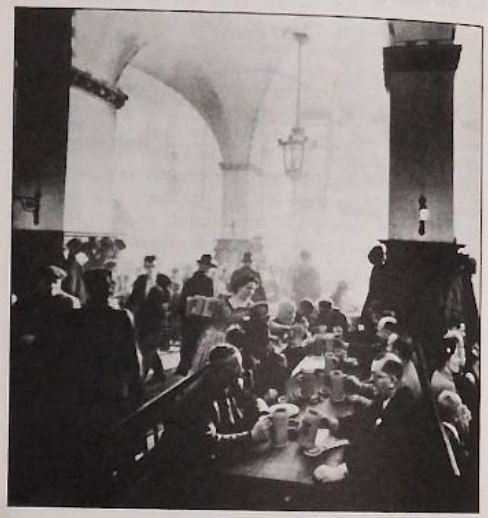
Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel

90 BEER HALL



. . . in an occasional neighborhood, which functions as the focus of a group of neighborhoods, or in a boundary between neighborhoods—NEIGHBORHOOD BOUNDARY (15)—or on the promenade which forms the focus of a large community—PROMENADE (31), NIGHT LIFE (33)—there is a special need for something larger and more raucous than a street cafe.

✣ ✣ ✣

Where can people sing, and drink, and shout and drink, and let go of their sorrows?

A public drinking house, where strangers and friends are drinking companions, is a natural part of any large community. But all too often, bars degenerate and become nothing more than anchors for the lonely. Robert Sommer has described this in "Design for Drinking," Chapter 8 of his book *Personal Space*, Englewood Cliffs, N.J.: Prentice-Hall, 1969.

. . . it is not difficult in any American city to find examples of the bar where meaningful contact is at a minimum. V. S. Pritchett describes the lonely men in New York City sitting speechlessly on a row of barstools, with their arms triangled on the bar before a bottle of beer, their drinking money before them. If anyone speaks to his neighbor under these circumstances, he is likely to receive a suspicious stare for his efforts. The barman is interested in the patrons as customers—he is there to sell, they are there to buy. . . .

Another visiting Englishman makes the same point when he describes the American bar as a "hoked up saloon; the atmosphere is as chilly as the beer . . . when I asked a stranger to have a drink, he looked at me as if I were mad. In England if a guy's a stranger, . . . each guy buys the other a drink. You enjoy each other's company, and everyone is happy. . . ." (Tony Kirby, "Who's Crazy?" *The Village Voice*, January 26, 1967, p. 39.)

Let us consider drinking more in the style of these English pubs. Drink helps people to relax and become open with one another, to sing and dance. But it only brings out these qualities when the setting is right. We think that there are two critical qualities for the setting:

1. The place holds a crowd that is continuously mixing be-

Warum Entwurfsmuster?

- Gemeinsame Sprache für Entwickler
  - Verbessert *Kommunikation*
  - Beugt Missverständnissen vor
- Lernen aus *Erfahrung*
  - Ein guter Entwickler zu werden ist schwer
    - Gute Entwürfe kennen / verstehen ist der erste Schritt
  - *Erprobte Lösungen* für *wiederkehrende Probleme*
    - Durch Wiederverwendung wird man produktiver
    - Eigene Software wird selbst flexibler und wiederverwendbarer

Objektorientierte Entwurfsmuster

Beschreibungen von *kommunizierenden Objekten* und *Klassen*, die angepasst wurden, um ein allgemeines *Entwurfsproblem* in einem bestimmten *Kontext* zu lösen.

Objektorientierte Entwurfsmuster

Entwurfsmuster setzen Entwurfsprinzipien um

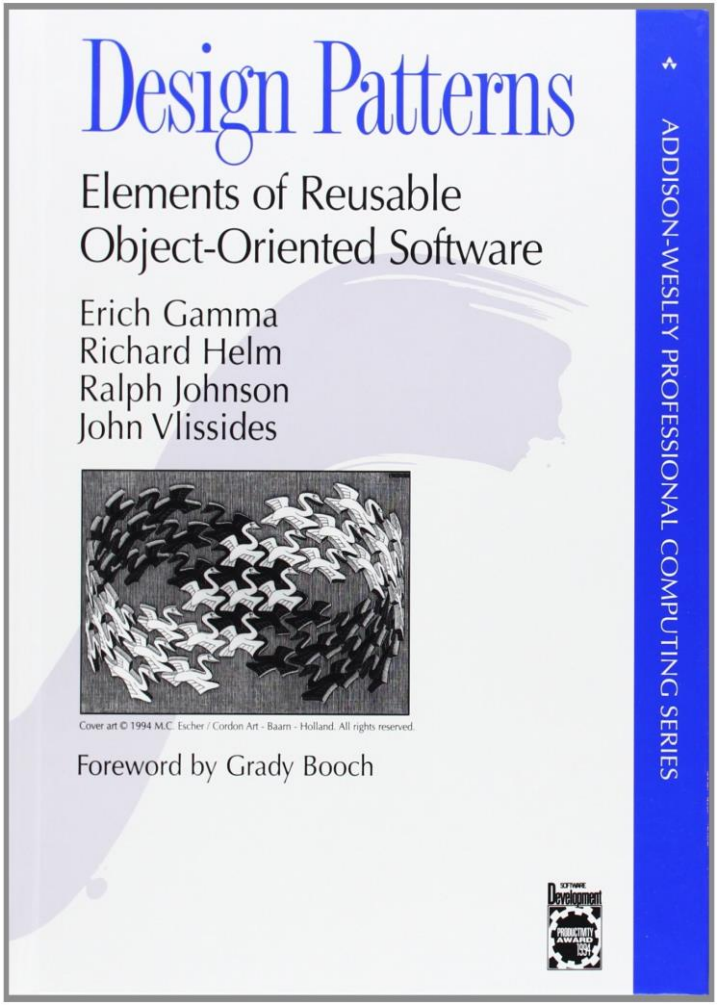
- Details verstecken
- Änderungen lokal halten
- Abstraktionen ordnen
- Schnittstellen für Erweiterung und Wiederverwendung
- …

Muster werden entdeckt und nicht erfunden

- „Best Practice“ von erfahrenen Entwicklern

Gang of Four (GoF) Design Patterns

"A design pattern names, abstracts, and identifies key aspects of a common design structure that makes it useful for creating a reusable object-oriented design."



Klassifikation

Zweck

Erzeugungsmuster

Helfen bei der Objekterstellung

Strukturmuster

Helfen bei der Komposition von Klassen und Objekten

Verhaltensmuster

Helfen bei der Interaktion von Klassen und Objekten (Verhalten kapseln)

Klassifikation

Anwendungsbereich

Klassenmuster

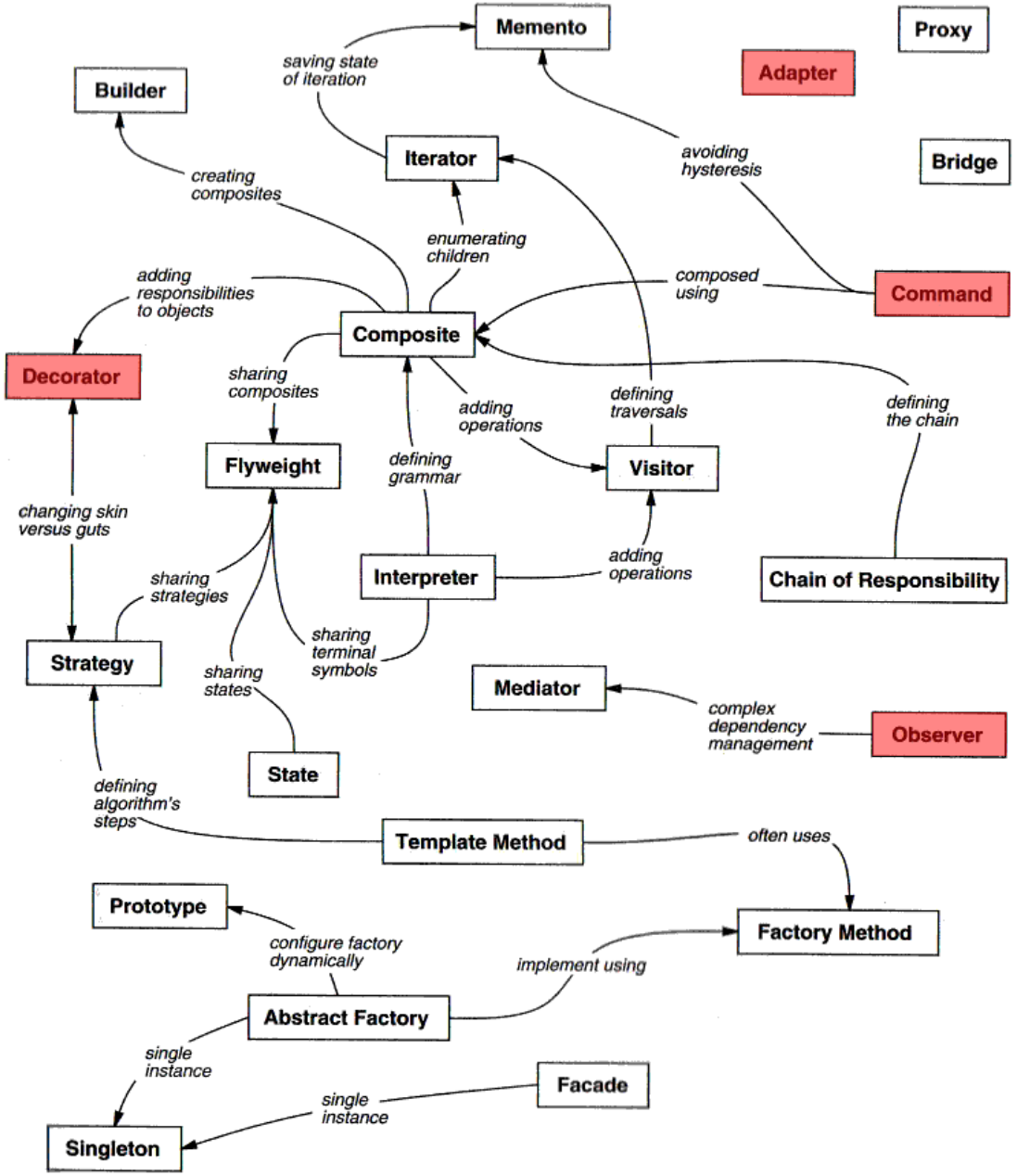
Fokussieren auf die Beziehung zwischen Klassen und ihren Subklassen (Wiederverwendung mittels Vererbung)

Objektmuster

Fokussieren auf die Beziehung zwischen Objekten (Wiederverwendung mittels Komposition)

Beschreibung eines Musters

Beschreibung	Erläuterung
Name und Klassifikation	Präziser Name des Muster und Einordnung
Zweck	Was das Muster bewirkt
Auch bekannt als	Alternativer Name
Motivation	Szenarien, in denen das Muster sinnvoll ist
Anwendbarkeit	Situationen, wann das Muster angewendet werden kann
Struktur	Grafische Repräsentation
Teilnehmer	Involvierte Klassen und Objekte
Kollaborationen	Wie arbeiten die Teilnehmer zusammen
Konsequenzen	Vor- und Nachteile des Musters
Implementierung	Hinweise und Techniken zur Implementierung
Beispiel-Code	Beispielhafte Code-Fragmente
Bekannte Verwendungen	Beispiele in realen Systemen
Verwandte Muster	Auflistung und Beschreibung der Verwandten Muster



Quelle: Design Patterns. Elements of Reusable Object-Oriented Software.

Adapter (Strukturmuster)

Beschreibung	Inhalt
Name und Klassifikation	Adapter – Strukturmuster
Zweck	Konvertiert das Interface einer existierenden Klasse, so dass es zu dem Interface eines Klienten (Client) passt. Ermöglicht, dass Klassen miteinander interagieren können, was sonst nicht möglich wäre aufgrund der Unterschiede im Interface.
Auch bekannt als	Wrapper Pattern oder Wrapper
Motivation	Eine existierende Klasse bietet eine benötigte Funktionalität an, aber implementiert ein Interface, was nicht den Erwartungen eines Clients entspricht.

Adapter (Strukturmuster)

Beschreibung	Inhalt
Anwendbarkeit	Verwende eine ansonsten nicht wiederverwendbare (durch Interface-Inkompatibilität) Klasse wieder: Adaptiere das Interface durch das Ändern der Methodensignaturen. Existierende Klasse bietet nicht die benötigte Funktionalität an: Implementiere die benötigte Funktion in Adapterklasse durch neue Methoden, die zum Interface passen
Struktur	Siehe nächste Folien
Teilnehmer	Siehe nächste Folien
Kollaborationen	Klienten rufen Methoden des Adapters auf, die die Anfragen an die adaptierte Klasse (Dienst) weiterleiten.

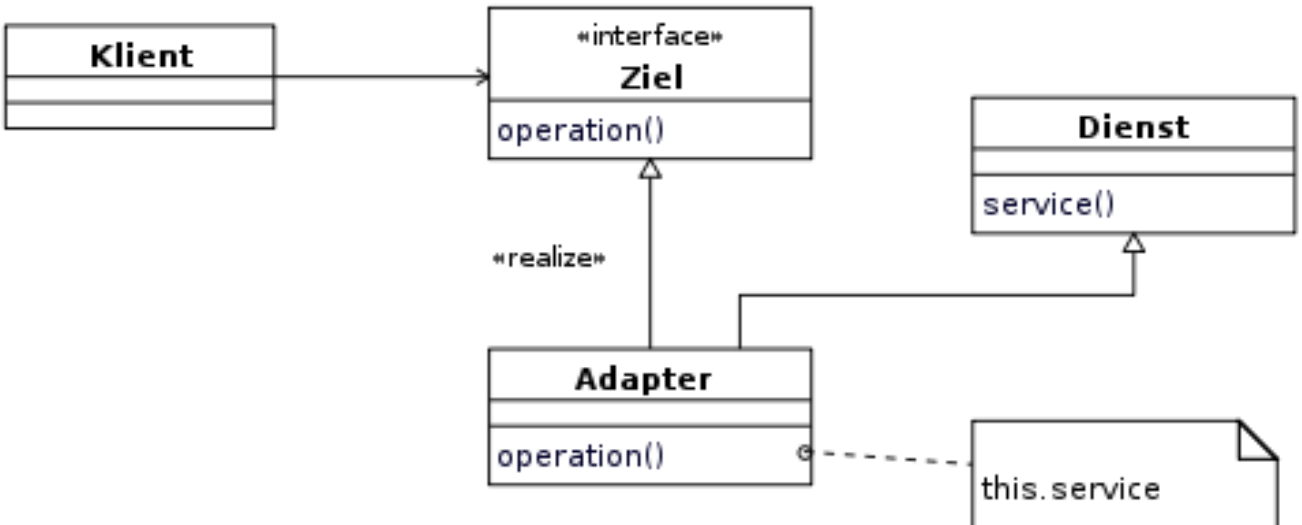
Adapter (Strukturmuster)

Struktur Variante 1:

Adapter nutzt *Mehrfachvererbung*, um ein Interface an ein anderes anzupassen

Adapter erbt von Ziel und Dienst (zu adaptierende Klasse)

Ziel muss Interface-Definition sein bei Sprachen mit Einfachvererbung (wie Java)



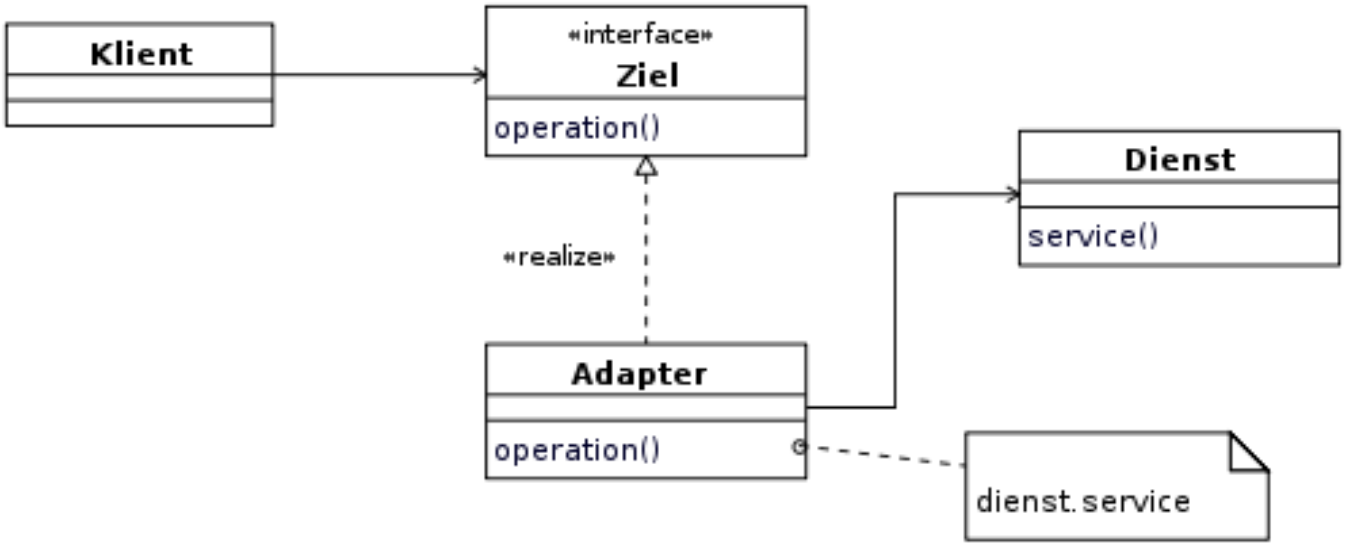
Adapter (Strukturmuster)

Struktur Variante 2:

Adapter nutzt *Delegation*, um Aufrufe weiterzuleiten

Adapter hält eine Referenz auf Dienst (zu adaptierende Klasse)

Methodenaufrufe werden vom Adapter zum Ziel weitergeleitet



Adapter (Strukturmuster)

Beschreibung	Inhalt
Teilnehmer	Ziel → Definiert das domänenspezifische Interface, welches der Client nutzt. Client → Interagiert mit Objekten, die das Zielinterface implementieren. Dienst (zu adaptierende Klasse) → Repräsentiert existierendes Interface, welches nicht kompatibel zum Ziel ist. Adapter → Adaptiert das Interface vom Dienst damit es kompatibel zum Ziel ist.

Adapter (Strukturmuster)

Beschreibung	Erläuterung
Konsequenzen	Klasse Adapter – Überschreibung der Methoden der Superklasse (Dienst) ist möglich Objekt Adapter – Dienst muss vererbbar sein, um Methoden überschreiben zu können Rate der Anpassbarkeit hängt vom Unterschied der Interfaces zwischen Ziel und Dienst ab
Bekannte Verwendungen	GUI Frameworks verwenden existierende Klassenhierarchien, müssen aber adaptiert werden
Verwandte Pattern	Decorator → Reichert Objekt um Funktionalität an ohne das Interface zu ändern Bridge → separiert Interface und Implementierung, so dass unterschiedliche Implementierungen leicht austauschbar sind

Adapter: Beispiel

Ziel

```
interface Stack<T> {
  public void push (T t);
  public T pop ();
  public T top ();
}
```

Dienst (adaptierte Klasse)

```
class DList<T> {
  public void insert (DNode pos, T t);
  public void remove (DNode pos, T t);
  public void insertHead (T t);
  public void insertTail (T t);
  public T removeHead ();
  public T removeTail ();
  public T getHead ();
  public T getTail ();
}
```

Adapter

```
class DListImpStack<T> extends DList<T> implements Stack<T> {
  public void push (T t) {
    insertTail (t);
  }
  public T pop () {
    return removeTail ();
  }
  public T top () {
    return getTail ();
  }
}
```

Aufgabe

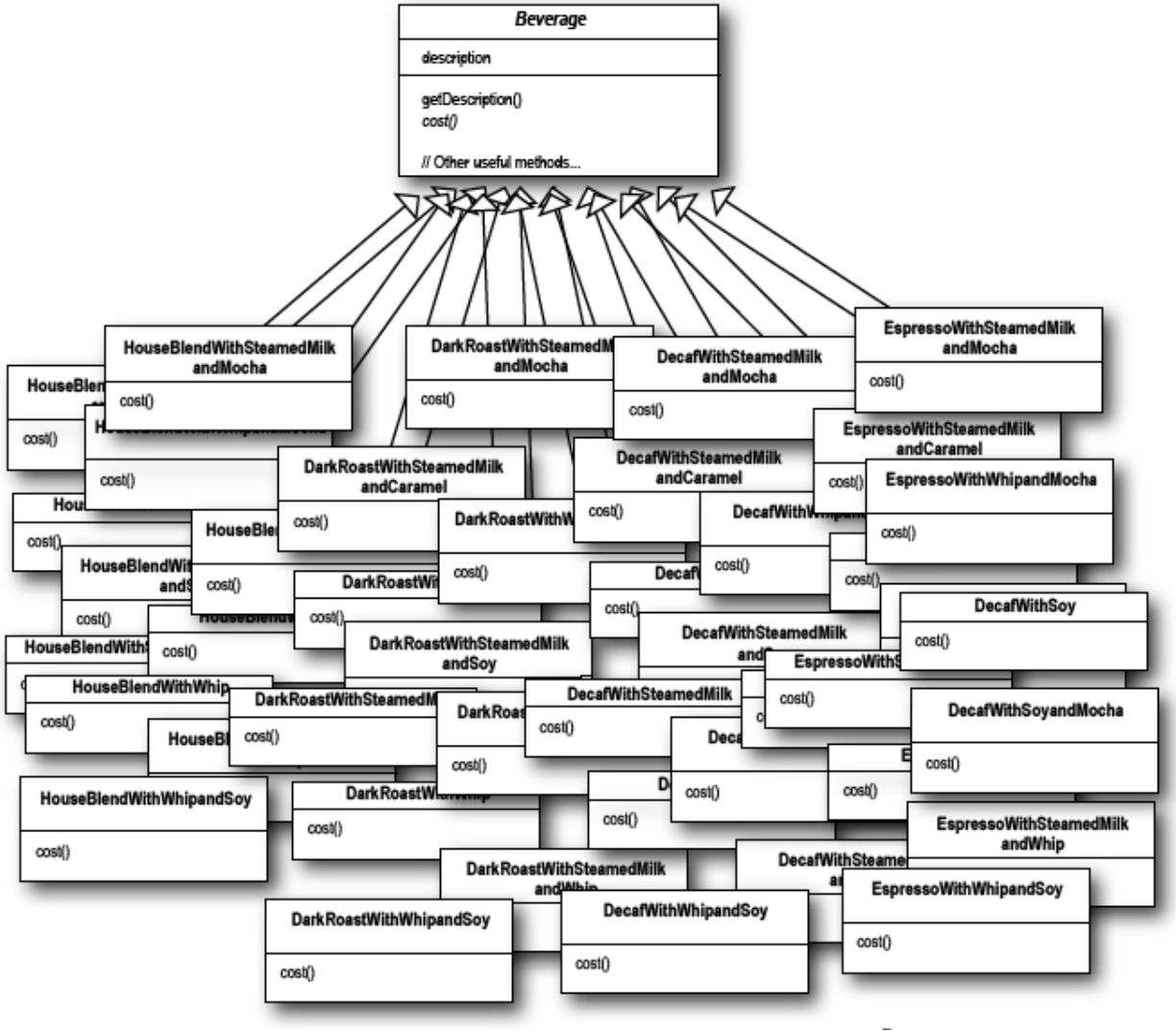
Modellieren Sie folgenden Sachverhalt als UML-Klassendiagramm:

- Für eine Steuersoftware eines Getränkeautomaten sollen verschiedene Getränke modelliert werden
- Getränke bestehen aus einer oder mehreren Zutaten und können frei kombiniert werden
  - DarkRoast, HouseBlend, Decaf, …
  - Soy, WholeMilk, LightMilk, …
  - …
- Abhängig von den Zutaten hat jedes Getränk einen anderen Preis

Lösung 1: Vererbung

Probleme:

- Explosion der Klassenhierarchie
- `DecafWithSoy` versus `SoyWithDecaf`?
- Zur Laufzeit nicht änderbar



Decorator (Strukturmuster)

Beschreibung	Inhalt
Name und Klassifikation	Decorator – Strukturmuster
Zweck	Fügt dynamisch Funktionalität zu bereits bestehenden Klassen hinzu.
Motivation	Wir benötigen flexible Implementierungen einer Klasse, die je nach Kontext unterschiedlich ausfallen.
Anwendbarkeit	Funktionale Erweiterungen sind optional. Anwendbar, wenn Erweiterungen mittels Vererbung unpraktisch ist.
Konsequenzen	Flexibler als statische Vererbung. Problem der Objektschizophrenie (ein Objekt ist zusammengesetzt aus mehreren Objekten). Viele kleine Objekte.

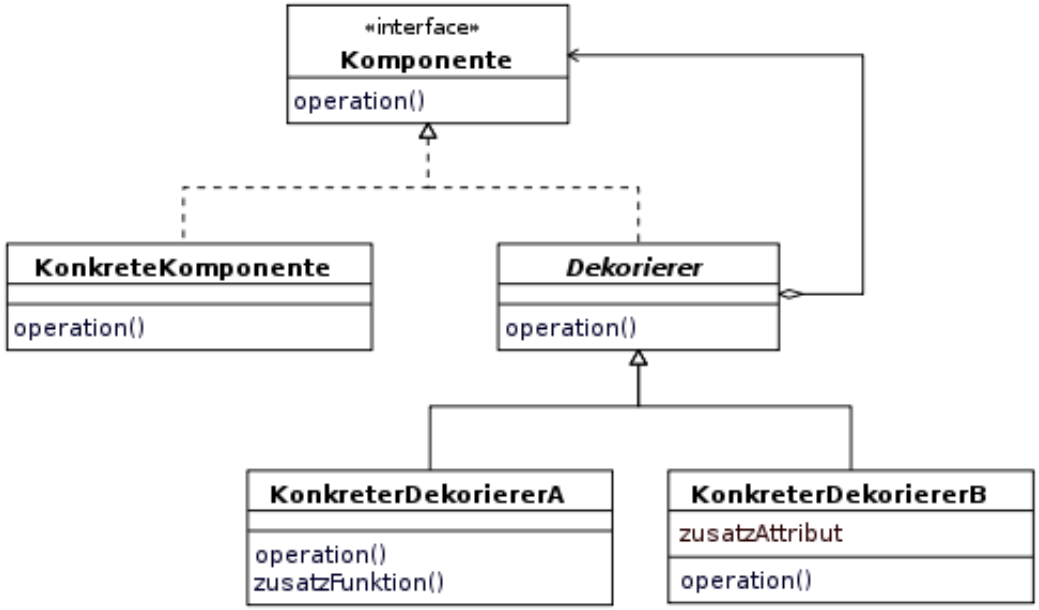
Decorator (Strukturmuster)

Struktur:

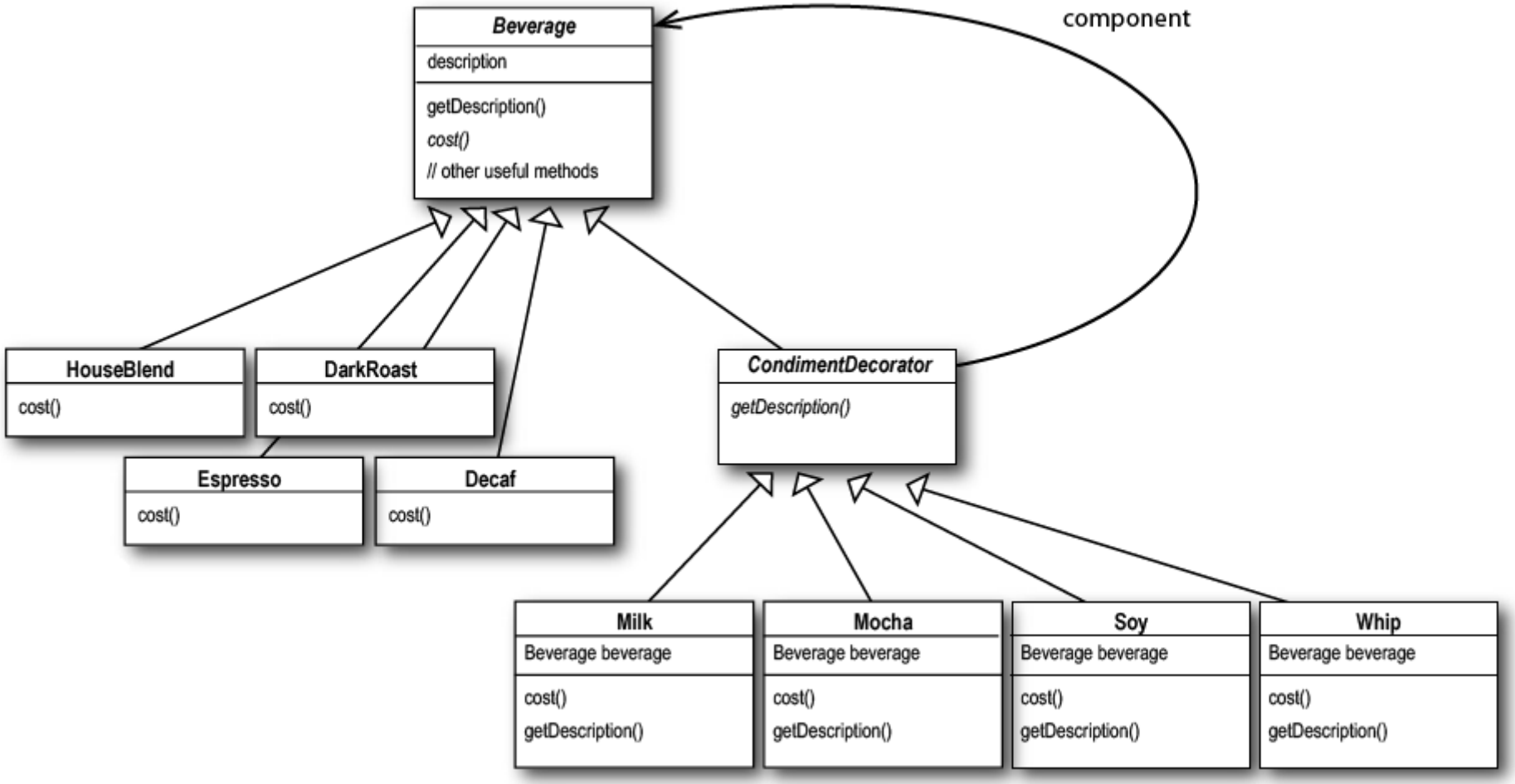
Instanz eines Dekorierers wird vor die zu dekorierende Klasse geschaltet → Funktionalität des Dekorierers wird zuerst ausgeführt

Dekorierer hat gleiche Schnittstelle wie zu dekorierende Klasse

Aufrufe werden weitergeleitet oder komplett selbst verarbeitet



Lösung 2: Decorator

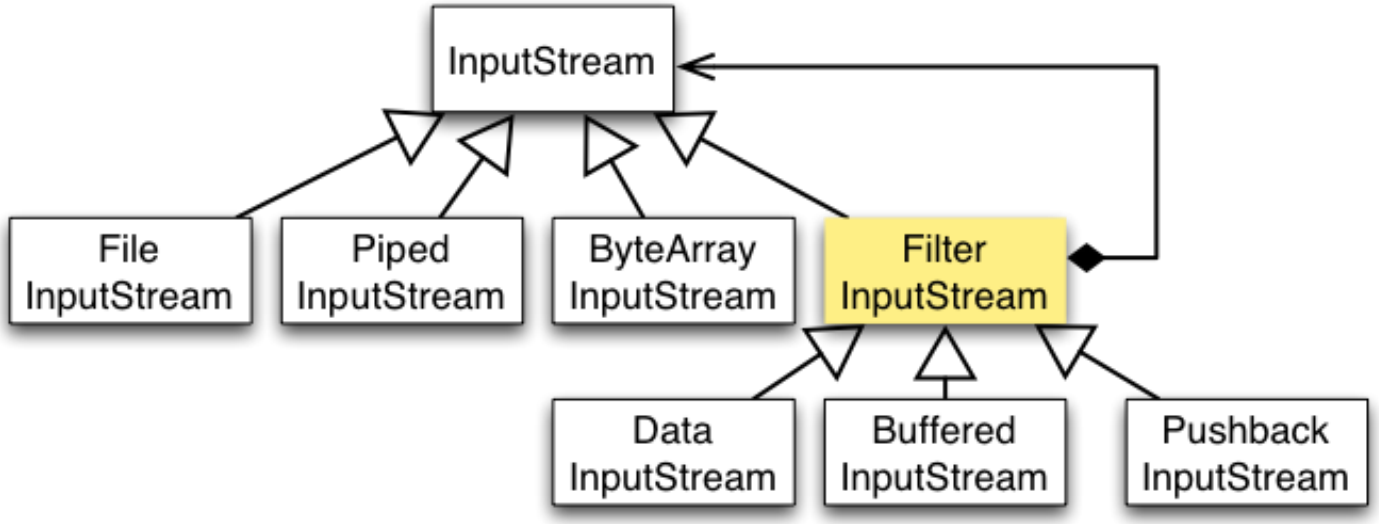


Decorator in Java

`java.io` bietet versch. Funktionen zur Ein- und Ausgabe:

Programme operieren auf Stream-Objekten ...

Unabhängig von der Datenquelle/-ziel und der Art der Daten



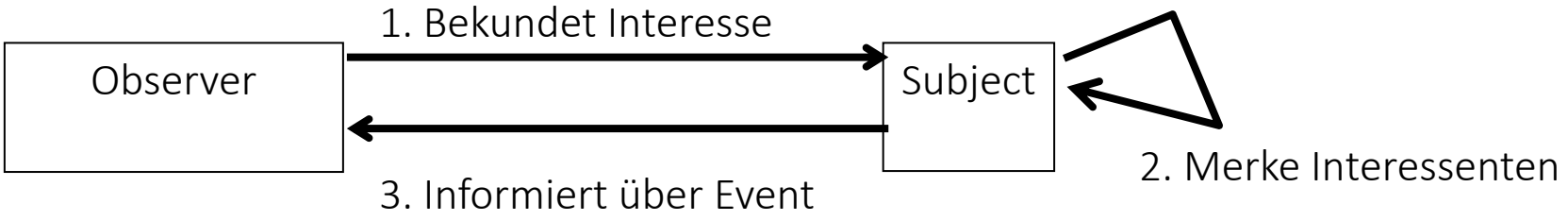

```
FileInputStream fis = new FileInputStream(“my.txt");
BufferedInputStream bis = new BufferedInputStream(fis);
GzipInputStream gis = new GzipInputStream(new ObjectInputStream(bis));
```

Observer (Verhaltensmuster)

Beschreibung	Inhalt
Name und Klassifikation	Observer – Verhaltensmuster
Zweck	Objekt verwaltet Liste von abhängigen Objekten und teilt diesen Änderungen mit
Auch bekannt als	Publish-Subscribe
Motivation	Implementiert verteilte Ereignisbehandlung (bei einem Ereignis müssen viele Objekte informiert werden). Schlüsselfunktion beim Model-View-Controler (MVC) Architekturmuster
Anwendbarkeit	Wenn eine Änderung an einem Objekt die Änderung an anderen Objekten erfordert und man weiß nicht, wie viele abhängige Objekte es gibt. Wenn ein Objekt andere Objekte benachrichtigen will, ohne dass es die Anderen kennt.

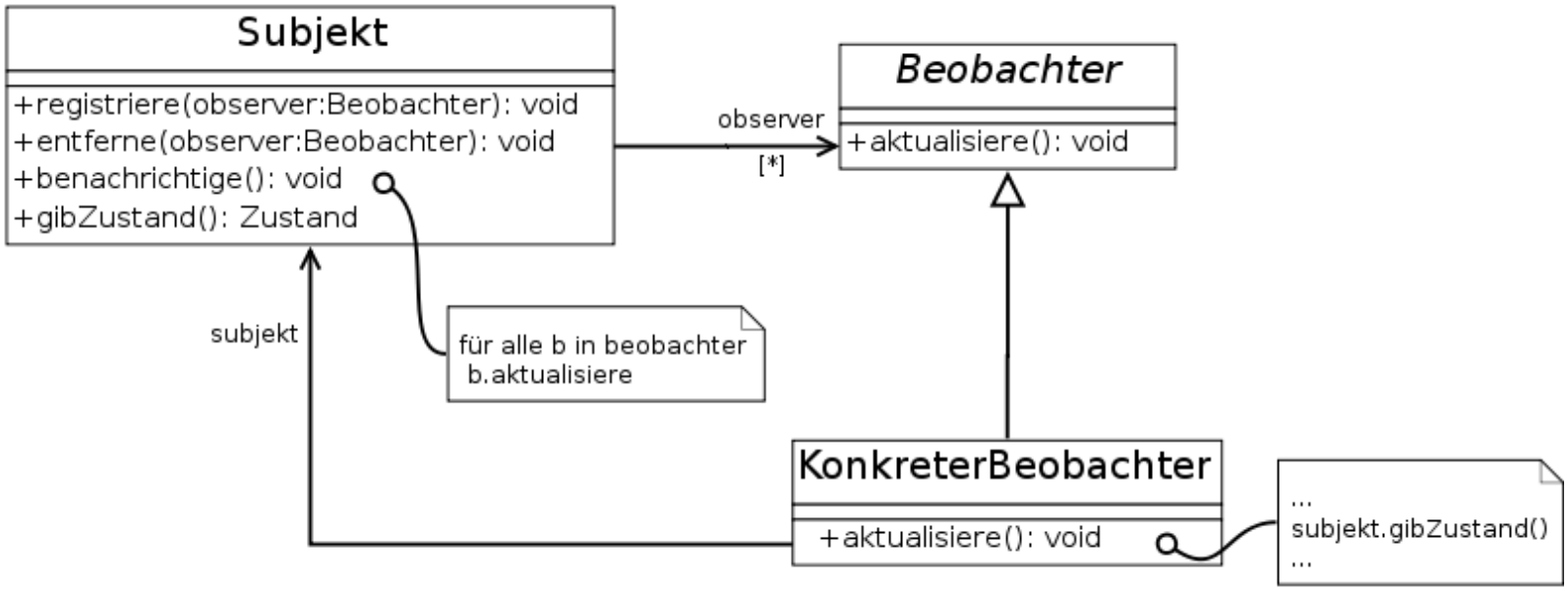
Observer (Verhaltensmuster)

Beschreibung	Inhalt
Konsequenzen	Schwache Kopplung von Objekten verbessert Wiederverwendung. Unterstützt „Broadcast“ Kommunikation (eine Nachricht an alle Teilnehmer verschicken). Mitteilungen können zu weiteren Mitteilungen führen und sich somit aufschaukeln.



Observer (Verhaltensmuster)

Struktur



Aufgabe

Wie sieht die Realisierung des Observer-Musters aus für eine Client-Server Kommunikation?

- Mehrere unterschiedliche Clients verbinden sich mit Server und wollen über Änderungen informiert werden

Observer (Verhaltensmuster)

```
interface ISubject {
       public void registerObserver(Observer observer);
       public void removeObserver(Observer observer);
       public void notifyObservers();
}
```

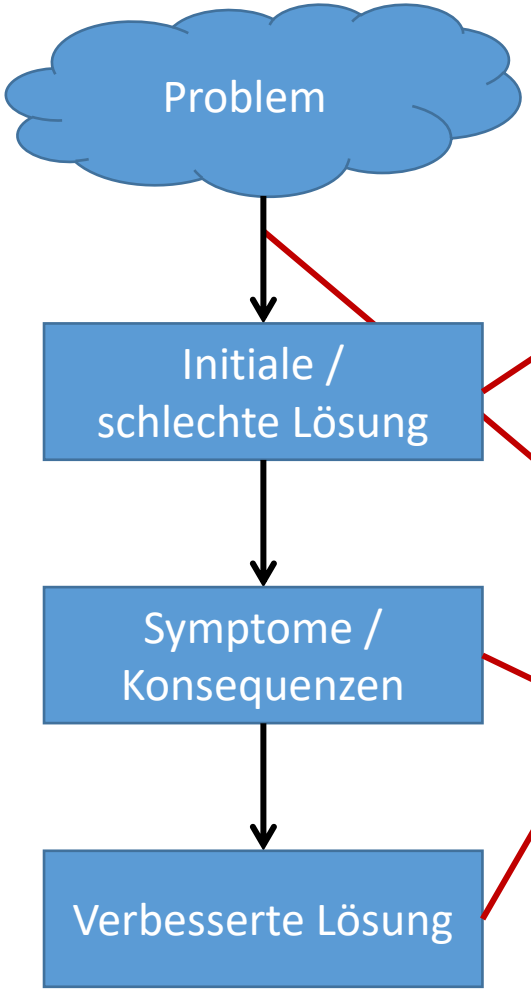
```
interface IObserver {
       public void update(String message);
}
```

```
class Server implements Subject {
    private ArrayList<Observer> observers
      = new ArrayList<Observer>();
    String message;
    public void postMessage(String message) {
        this.message = message;
        notifyObservers();
    }
    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyObservers() {
        for (Observer ob : observers) {
            ob.update(this.message);
        }
    }
}
```

```
class WebClient implements Observer {
    @Override
    public void update(String message) {
        postOnWebPage(message);
    }
    …
}
class Messenger implements Observer {
    @Override
    public void update(String msg) {
        System.out.println("Receiving message: " + msg);
    }
}
class Main {
    public static void main(String [] args){
        Server s = new Server();
        WebClient wc = new WebClient();
        s.registerObserver(wc);
        Messenger m = new Messenger();
        s.registerObserver(m);
        s.postMessage("Hello World!");
    }
}
```

Antimuster (Anti-Patterns)

Antimuster



Antimuster bestehen aus:

- Nicht-optimalen Lösung
- Optimale / verbesserte Lösung

Beschreibung *wie* und *warum* es zur nicht-optimalen Lösung kam

- Ursächliche Wirkketten
- Nach außen sichtbare Symptome

Beispiele für Symptome

“*Wozu ist diese Klasse eigentlich da?*”

Entwurfsdokumente und Code sind bestenfalls entfernte Verwandte

Fehlerrate steigt mit jeder neuen Version an

„*Wenn die Liste mehr als 100 Einträge hat, sinkt die Performance in den Keller.*“

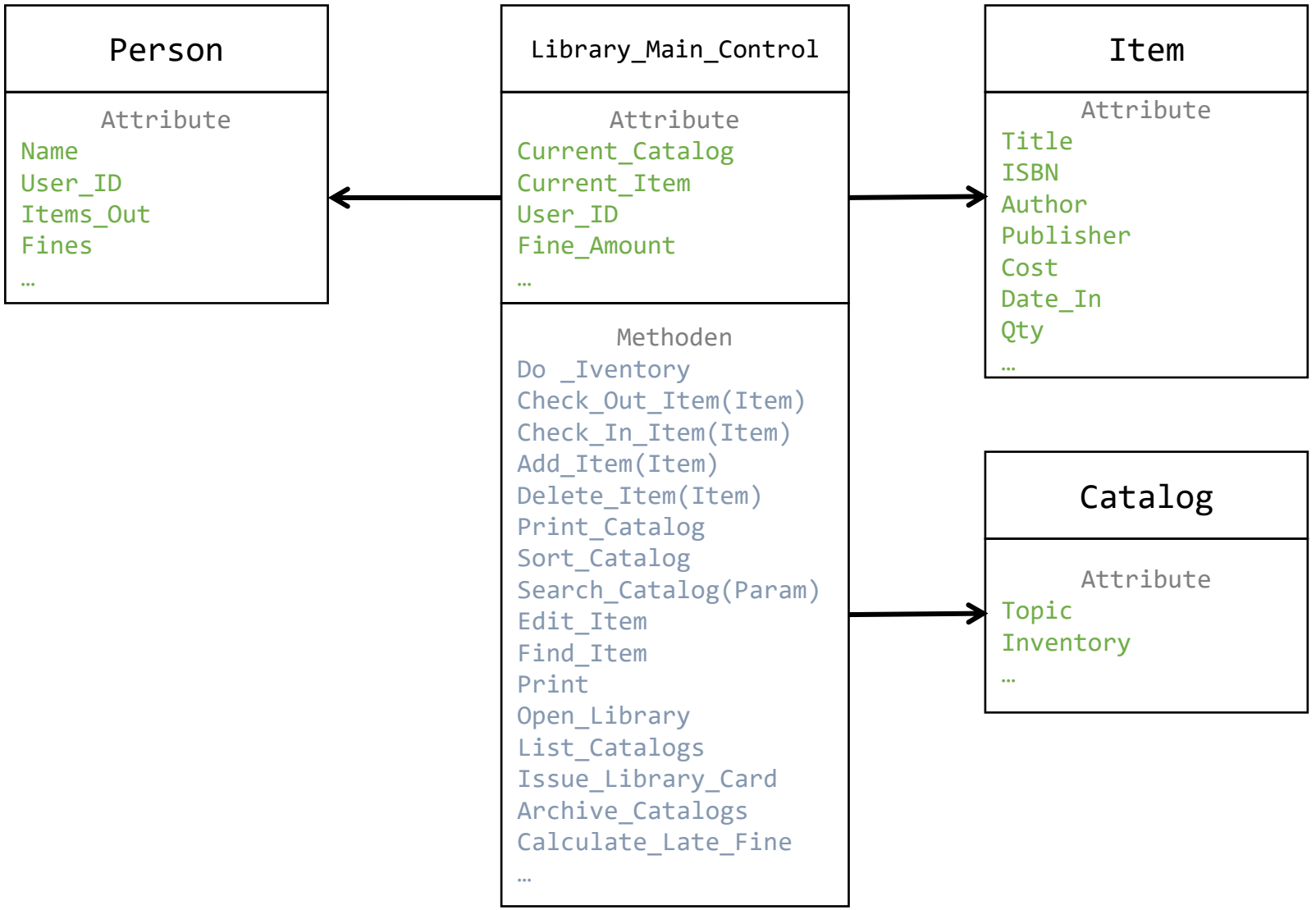
The Blob

„Diese Klasse ist das Herzstück unserer Architektur.“

The Blob

Beschreibung	Inhalt
Name und Klassifikation	The Blob – Antimuster
Grund	Eile, Faulheit
Auch bekannt als	Winnebago and The God Class
Symptome	Klasse mit sehr vielen Methoden. Methoden und Klassen mit sehr unterschiedlichen Funktionen. Verbindung mit sehr vielen anderen Klassen, die jeweils wenige Methoden haben. Klasse zu komplex für Testen und Wiederverwendung.
Lösung	Refaktorisierung der Klasse anhand Verantwortlichkeiten. Ähnliche Attribute/Methoden identifizieren und kapseln. Methoden ggfs. verlagern in bereits existierende Klassen.
Konsequenzen	Performanceeinbußen. Schlechte Wartbarkeit. Kaum Wiederverwendbarkeit der Funktionen.

The Blob

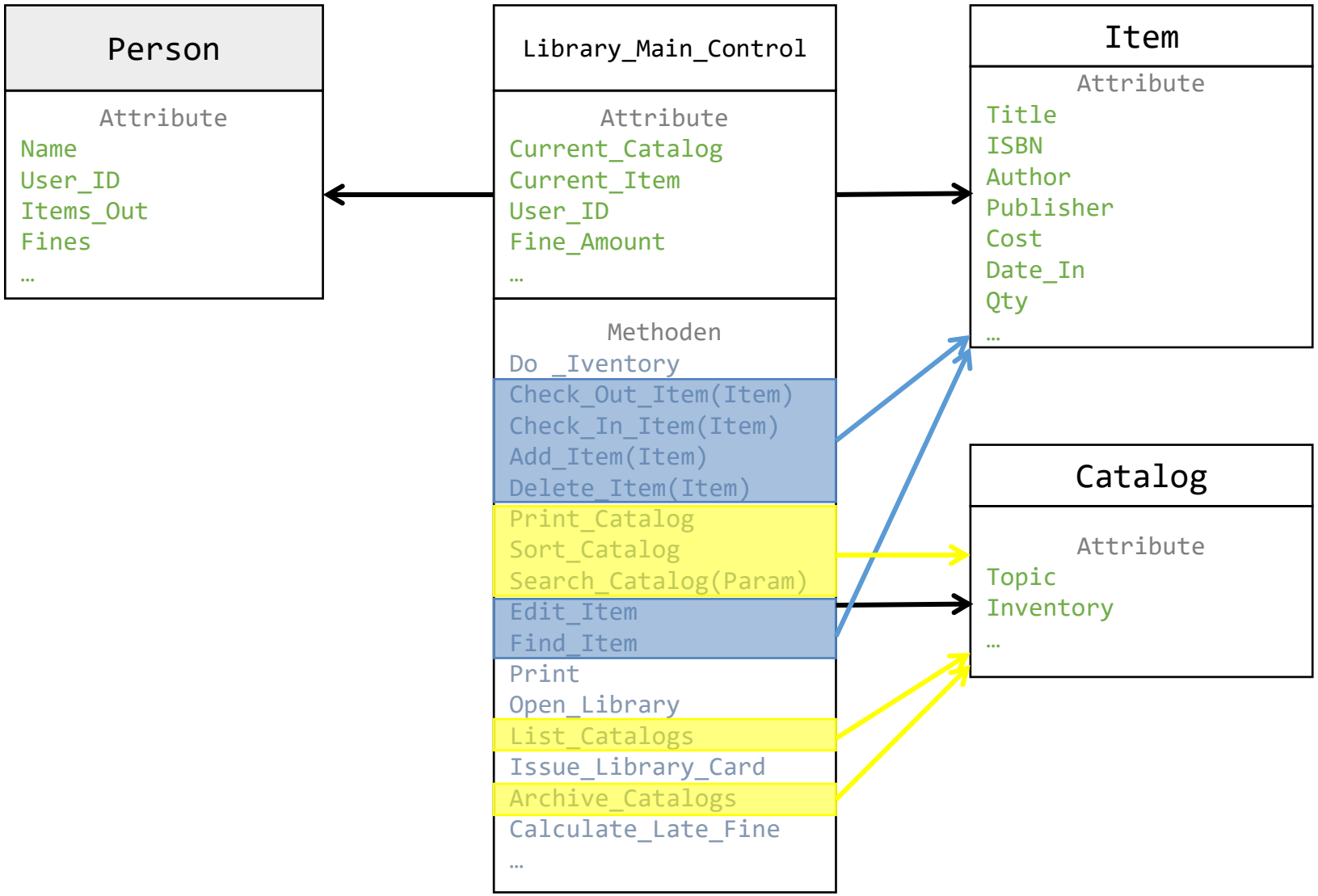

Person
Attribute
Name
User_ID
Items_Out
Fines
…
Library_Main_Control
Attribute
Current_Catalog
Current_Item
User_ID
Fine_Amount
…
Methoden
Do _Iventory
Check_Out_Item(Item)
Check_In_Item(Item)
Add_Item(Item)
Delete_Item(Item)
Print_Catalog
Sort_Catalog
Search_Catalog(Param)
Edit_Item
Find_Item
Print
Open_Library
List_Catalogs
Issue_Library_Card
Archive_Catalogs
Calculate_Late_Fine
…
Item
Attribute
Title
ISBN
Author
Publisher
Cost
Date_In
Qty
…
Catalog
Attribute
Topic
Inventory
…

Aufgabe

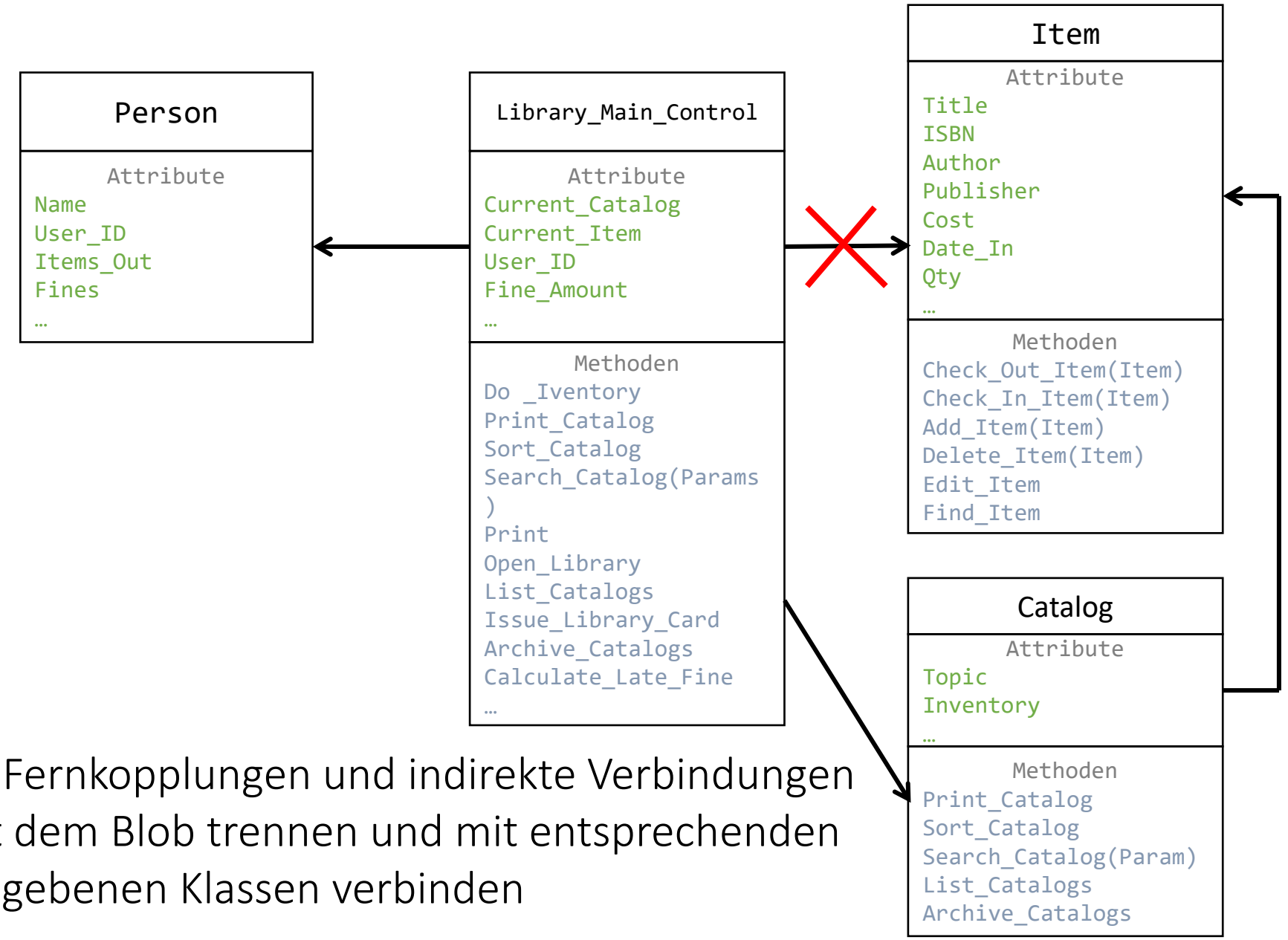
(1) Finde zusammenhängende Attribute und Methoden im Blob und gruppiere diese

(2) Lagere die Gruppen in passende umgebene Klassen aus

The Blob


Person
Attribute
Name
User_ID
Items_Out
Fines
…
Library_Main_Control
Attribute
Current_Catalog
Current_Item
User_ID
Fine_Amount
…
Methoden
Do _Iventory
Check_Out_Item(Item)
Check_In_Item(Item)
Add_Item(Item)
Delete_Item(Item)
Print_Catalog
Sort_Catalog
Search_Catalog(Param)
Edit_Item
Find_Item
Print
Open_Library
List_Catalogs
Issue_Library_Card
Archive_Catalogs
Calculate_Late_Fine
…
Item
Attribute
Title
ISBN
Author
Publisher
Cost
Date_In
Qty
…
Catalog
Attribute
Topic
Inventory
…

The Blob



(3) Fernkopplungen und indirekte Verbindungen mit dem Blob trennen und mit entsprechenden umgebenen Klassen verbinden

Weitere Antimuster

The Golden Hammer

Ein bekanntes Verfahren („Golden Hammer“) wird auf alle möglichen Probleme angewandt: Mit einem Hammer sieht jedes Problem wie ein Nagel aus.

Lösung: Ausbildung verbessern.

Weitere Antimuster

Copy-and-Paste

Code wird an zahlreichen Stellen wiederverwendet, indem er kopiert und verändert wird. Dies sorgt für ein Wartungsproblem.

Lösung: Black-Box-Wiederverwendung, Ausfaktorisieren von Gemeinsamkeiten.

Weitere Antimuster

Lava Flow (schnell wechselnder Entwurf)

Boat Anchor (Komponente ohne erkennbaren Nutzen)

Dead End (eingekaufte Komponente, die nicht mehr unterstützt wird),

…