Softwaretechnik

Design Patterns (Entwurfsmuster)

SOFTWARE SYSTEME

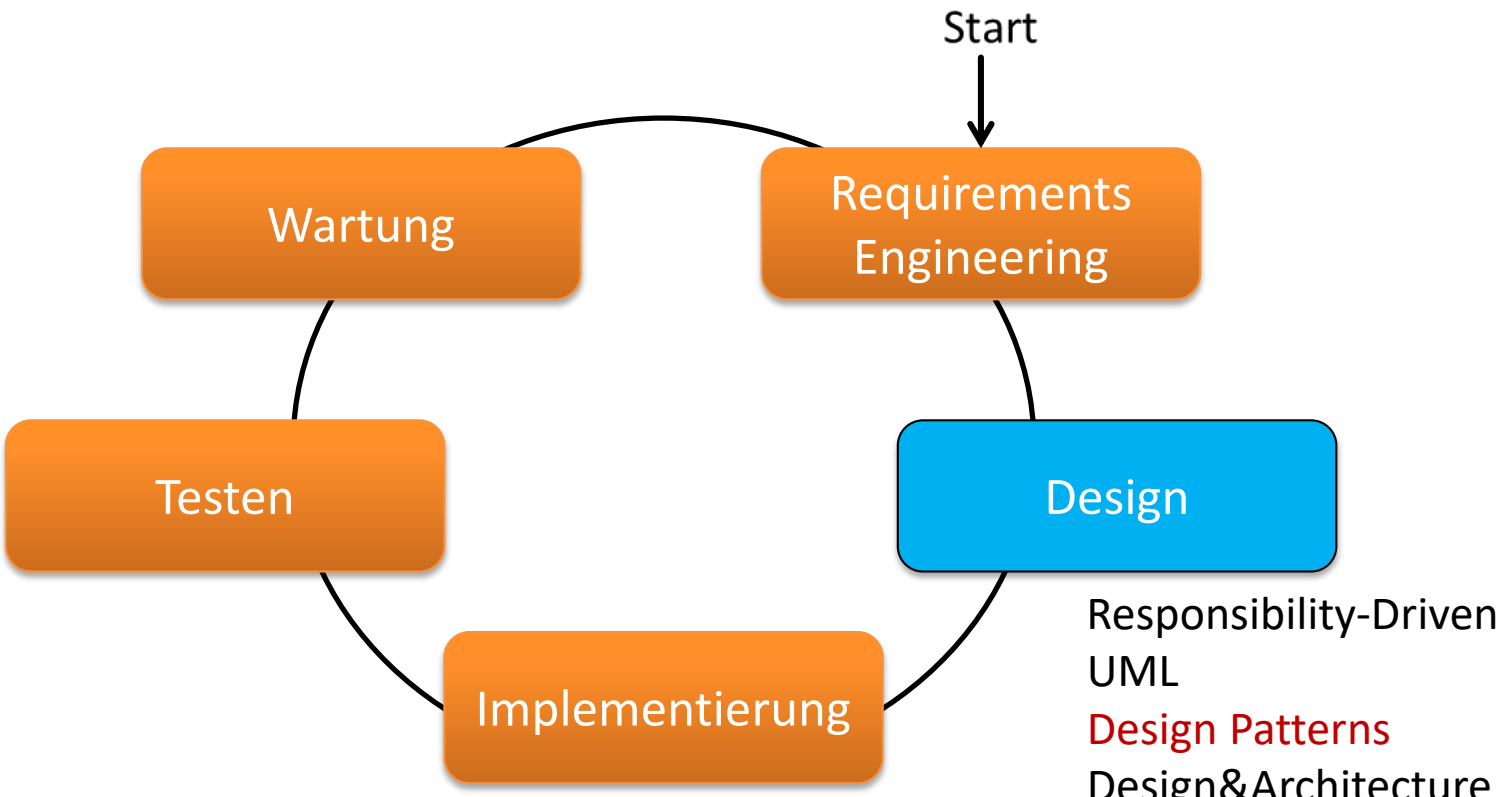
UNIVERSITÄT LEIPZIG

Prof. Dr.-Ing. Norbert Siegmund
Software Systems

"Designing object-oriented software is hard and designing reusable object-oriented software is even harder."

—Erich Gamma

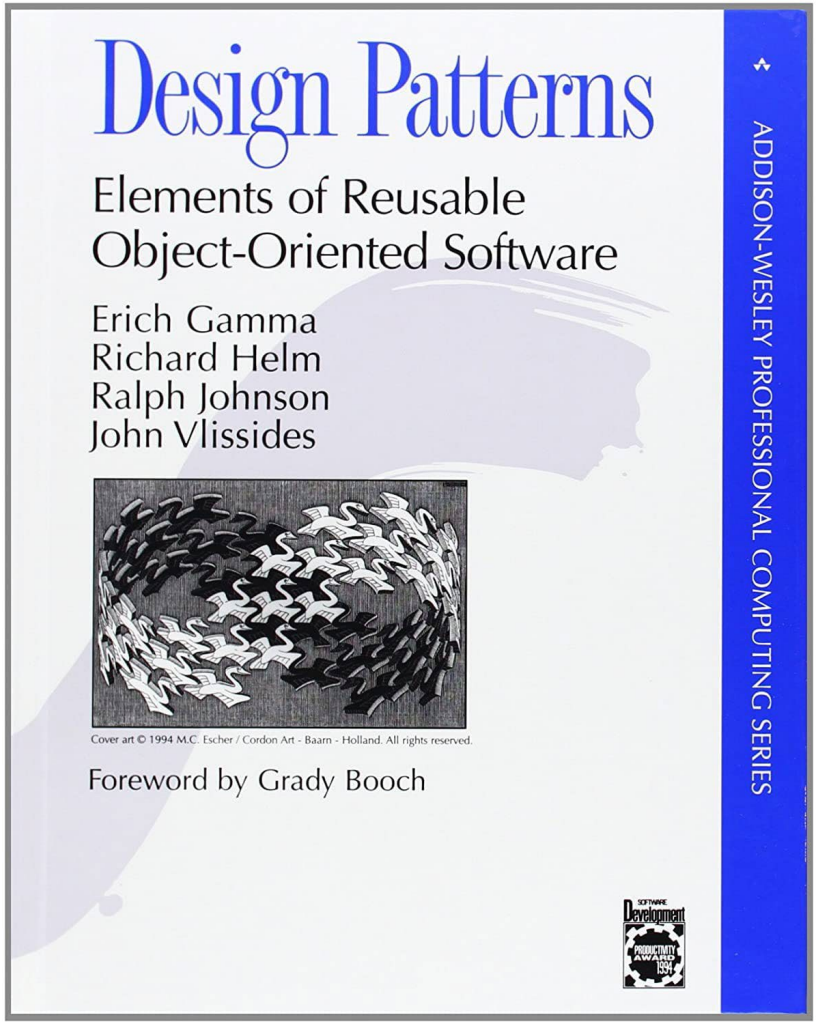
Einordnung



Responsibility-Driven
UML
Design Patterns
Design&Architecture

Inhalt

- Design Patterns
  - Warum notwendig?
  - Klassifizierung von Patterns
  - Beschreibung von Patterns
  - Kennenlernen von Patterns
    - Adapter
    - Iterator
    - Decorator
    - Observer
    - Visitor
    - Auch wichtig (Singleton, Factory Method, Strategy, State, Lazy loading, active record, DAO, Composite, Mediator, Facade)
- Bonus: Anti-Patterns

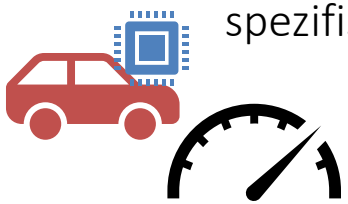


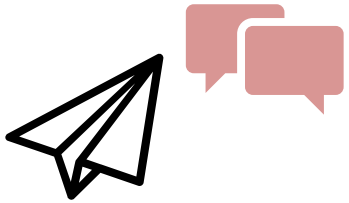
Lernziele

- In der Lage sein:
  - Die Nützlichkeit von Design Patterns zu erklären und dabei deren Unzulänglichkeiten zu kennen
  - Klar und präzise Beispiele für Situationen in der Softwareentwicklung zu benennen, bei denen Design Patterns hilfreich sind
  - Wichtig Patterns benennen und implementieren können
  - Situationen kennen, um Design Patterns anwenden zu können
- Wichtiges „Handwerkszeug“ von Software-Entwicklern kennen
- In Zukunft bei der Implementierung Wiederverwendung im Hinterkopf haben (loose coupling, Interfaces, Vererbung, etc.)

Warum Design Pattern?

Entwurf einer Autoverkaufssoftware, die Grundinformationen (wie z.B. Preis und Geschwindigkeit) zu Autos liefert. Unsere Software basiert auf EU-Daten (Preis in Euro, Abmaße in metrischen System). Wir wollen nun in den US-Markt expandieren, doch deren Daten sind inkompatibel mit unseren. Wie entwerfen wir das System, so dass wir trotzdem die EU-spezifischen Daten für die USA verwenden können?



Wir implementieren ein Benachrichtigungssystem, welches von den Nutzern dynamisch anpassbar sein soll, wie z.B. einfache TCP/IP Kommunikation, senden per Email, senden auf Mastodon, senden auf Slack, etc. Später sollen noch Features für Verschlüsselung, Emojis, Farben, Befehle, etc. dazu kommen können. Wie entwerfen wir eine solche dynamische Erweiterbarkeit?




Bei der Erstellung eines Labyrinths für ein Spiel können wir unterschiedliche Settings haben (z.B. Untergrund, Natur, geheimnisvoll, herausfordernd, etc.). Ein Labyrinth besteht dabei aus Räumen, die entsprechend des Settings instanziiert werden müssten. Wie entwerfen wir die Erstellung, so dass der Code nicht für jedes Setting neu geschrieben werden muss?

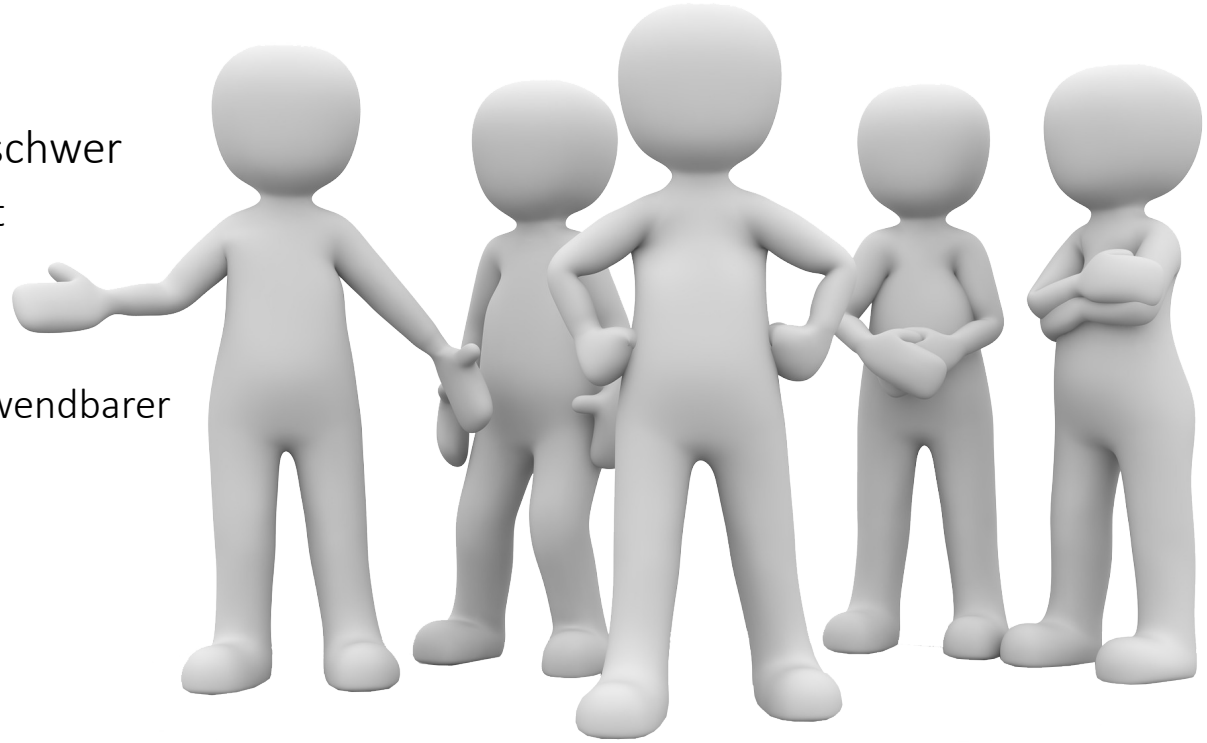
Was sind Patterns (Muster)?

- “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”
- Muster für ein wiederkehrendes Problem im Design (Entwurf) eines Softwaresystems.

- Jedes Entwurfsmuster ist ein Triple, welches für einen bestimmten Kontext ein Problem beschreibt und dessen Lösung.

Warum Patterns benutzen?

- Gemeinsame Sprache für Entwicklerinnen
  - Verbessert Kommunikation
  - Beugt Missverständnisse vor
- Lernen aus Erfahrung
  - Eine gute Designerin / Entwicklerin zu werden, ist schwer
    - Gute Designs kennen / verstehen ist der erste Schritt
  - Erprobte Lösungen für wiederkehrende Probleme
    - Durch Wiederverwendung wird man produktiver
    - Eigene Software wird selbst flexibler und wiederverwendbarer

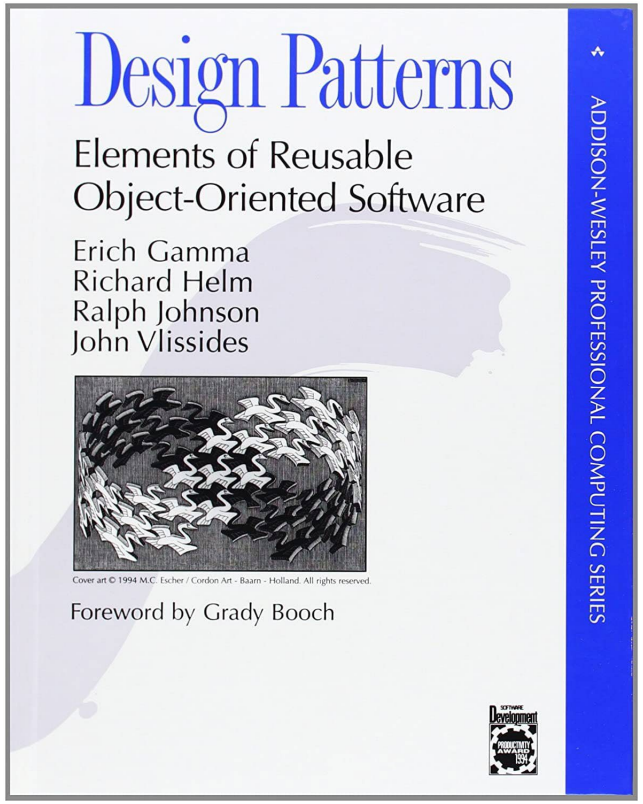


Zusammenhang Patterns und OOP

- Fundamentale OOP-Design-Prinzipien:
  - Patterns folgen Design-Zielen
    - Modularität, Explizite Interfaces, Information Hiding, …
  - Patterns entstehen aus OOP Design-Prinzipien
    - Design nach Schnittstellen
    - Favorisiere Komposition über Vererbung
    - Finde Variabilität und kapsele sie
  - Patterns werden entdeckt und nicht erfunden
    - „Best Practice“ von erfahrenen Entwickelnden

Gang of Four (GoF) Design Patterns

- "The Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, und John Vlissides
  - "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 über Zweck und Anwendungsbereich



Klassifikation I

- Zweck
  - Creational Patterns
    - Helfen bei der Objekterstellung
  - Structural Patterns
    - Helfen bei der Komposition von Klassen und Objekten
  - Behavioral Patterns
    - Helfen bei der Interaktion von Klassen und Objekten (Verhalten kapseln)

Klassifikation II

- Anwendungsbereich
  - Class Patterns
    - Fokussieren auf die Beziehung zwischen Klassen und ihren Subklassen (Wiederverwendung mittels Vererbung)
  - Object Patterns
    - Fokussieren auf die Beziehung zwischen Objekten (Wiederverwendung mittels Komposition)

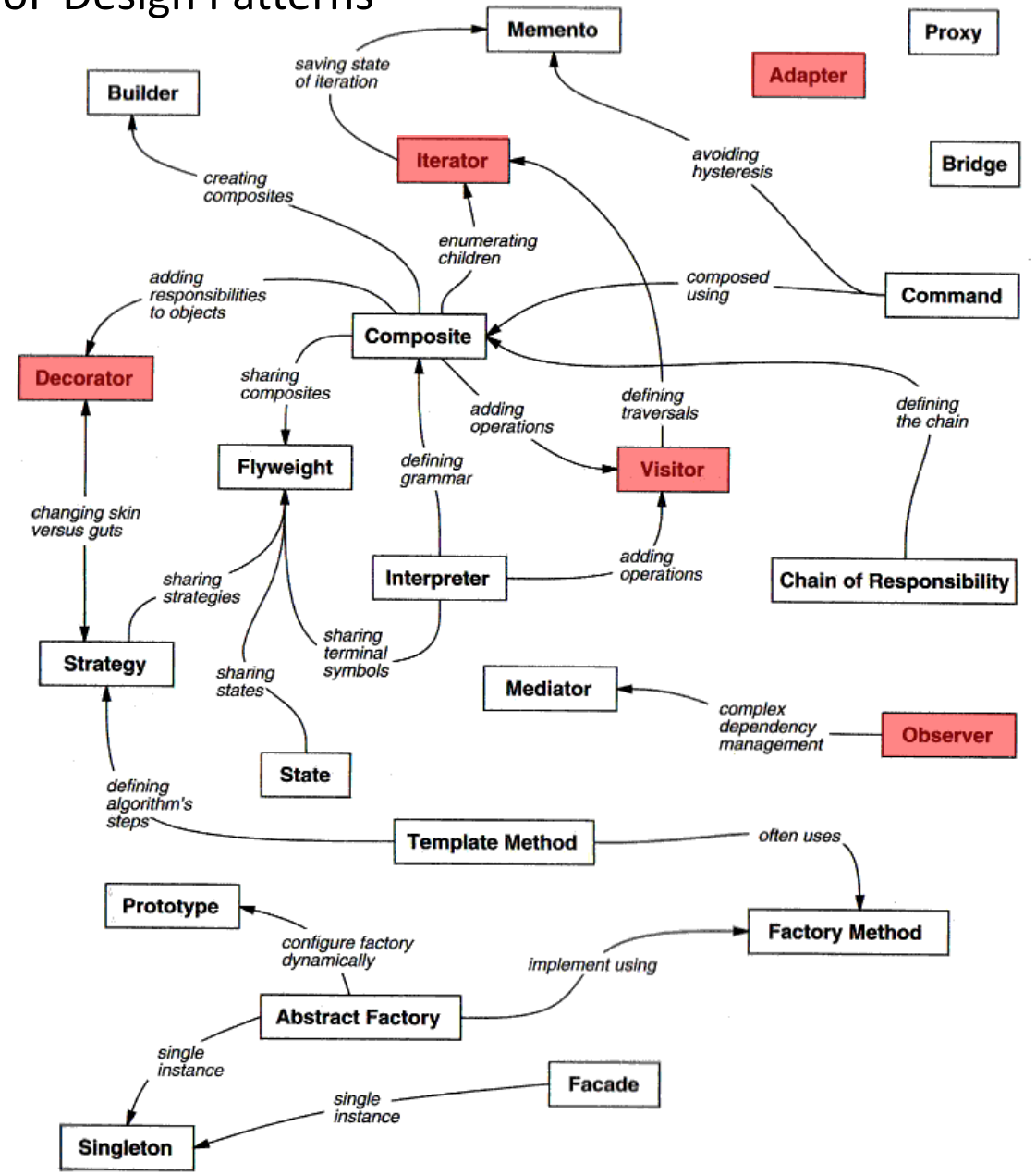
Beschreibung eines Patterns

Beschreibung	Erläuterung
Pattern Name und Klassifikation	Präziser Name des Patterns
Zweck	Was das Pattern bewirkt
Auch bekannt als	Alternativer Name
Motivation	Szenario, wo das Pattern sinnvoll ist
Anwendbarkeit	Situationen, wann das Pattern angewendet werden kann
Struktur	Grafische Repräsentation
Teilnehmer	Involvierten Klassen und Objekte
Kollaborationen	Wie arbeiten die Teilnehmer zusammen

Beschreibung	Erläuterung
Konsequenzen	Vor- und Nachteile des Patterns
Implementierung	Hinweise und Techniken zur Implementierung
Beispiel Code	Code Fragmente
Bekannte Verwendungen	Beispiele in realen Systemen
Verwandte Pattern	Auflistung und Beschreibung der Verwandten Pattern

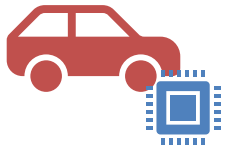
Wichtige Design Patterns…

Beziehungen der GoF Design Patterns



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

Entwurf einer Autoverkaufssoftware, die Grundinformationen (wie z.B. Preis und Geschwindigkeit) zu Autos liefert. Unsere Software basiert auf EU-Daten (Preis in Euro, Abmaße in metrischen System). Wir wollen nun in den US-Markt expandieren, doch deren Daten sind inkompatibel mit unseren. Wie entwerfen wir das System, so dass wir trotzdem die EU-spezifischen Daten für die USA verwenden können?



Adapter – Structural Pattern I

Beschreibung	Inhalt
Pattern Name und Klassifikation	Adapter – Structural Pattern
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 – Structural Pattern II

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.

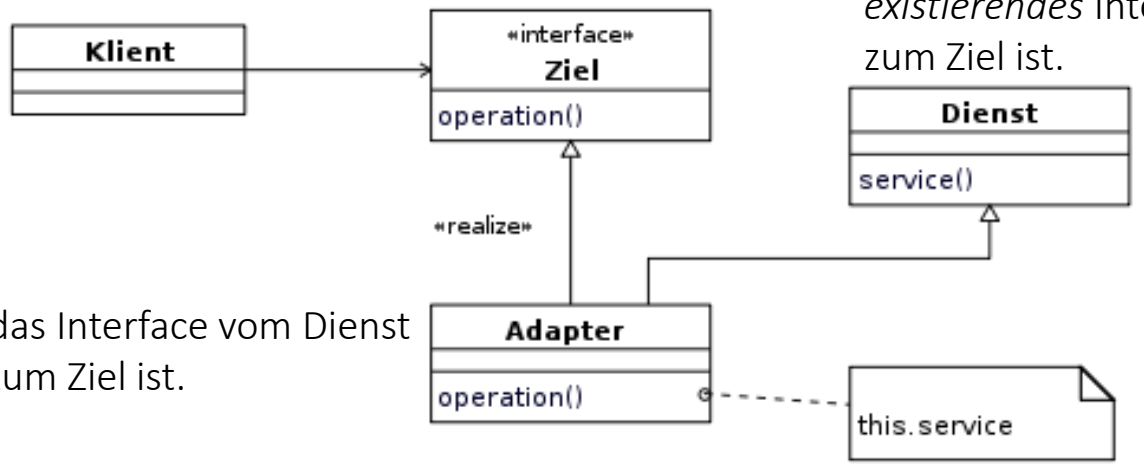
Adapter – Structural Pattern III

- Struktur Variante 1:
  - Adapter nutzt multiple Vererbung um ein Interface auf ein anderes passend zu machen
  - Adapter erbt von Ziel und Dienst (zu adaptierende Klasse)
  - Ziel muss Interface-Definition sein bei Sprachen ohne Mehrfachvererbung (wie Java)

Ziel -> Definiert das domänenspezifische Interface, welches der Klient nutzt

Dienst (zu adaptierende Klasse) -> Repräsentiert *existierendes* Interface, welches nicht kompatibel zum Ziel ist.

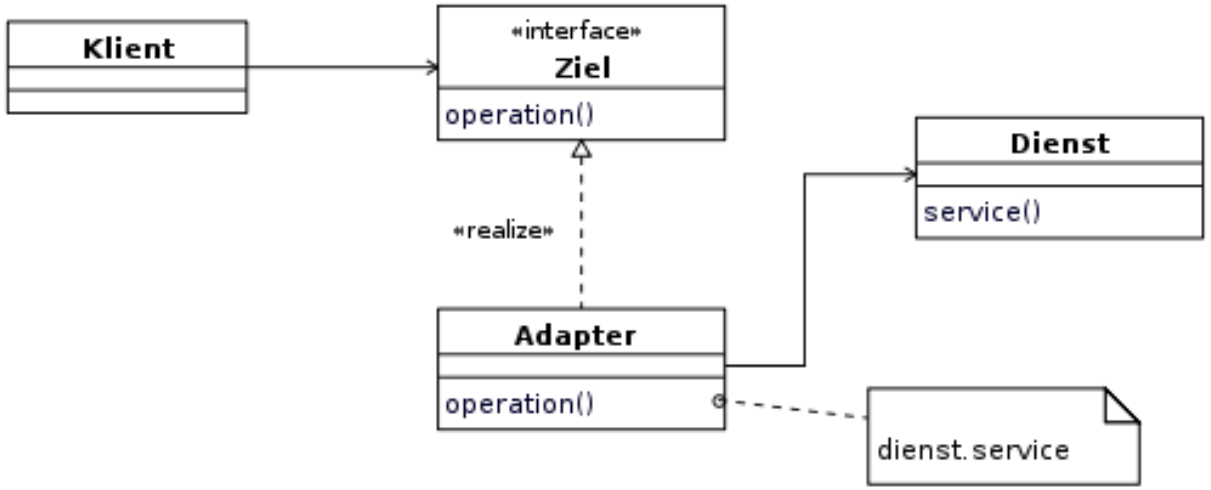
Klient -> Interagiert mit Objekten, die das Zielinterface implementieren.



Adapter -> Adaptiert das Interface vom Dienst damit es kompatibel zum Ziel ist.

Adapter – Structural Pattern IV

- 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 – Structural Pattern VI

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
Implementierung	Siehe nächste Folien
Beispiel Code	Siehe nächste Folien
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

Aufgabe

- Wie sieht der Beispielcode für folgendes Szenario aus:
  - Ziel: Stack
  - Dienst: List
- Implementieren Sie einen Stack mittels des Adapter Patterns bei denen Sie bereits implementierte Funktionen einer generischen Liste wiederverwenden können

Adapter – Structural Pattern VII

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 ();
  }
}
```

Adapter – Structural Pattern VII

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

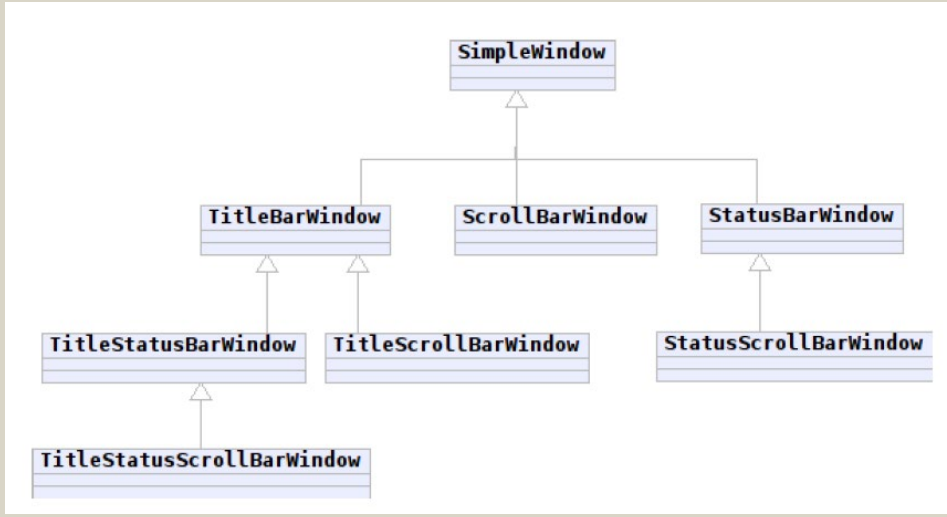
```
class DListToStackAdapter<T> implements Stack<T> {
  private DList<T> m_List;
  public DListToStackAdapter(DList<T> m_List) {
    this.m_List = m_List;
  }
  public void push (T t) {
    m_List.insertTail (t);
  }
  public T pop () {
    return m_List.removeTail ();
  }
  public T top () {
    return m_List.getTail ();
  }
}
```

Aufgabe

- Modellieren Sie folgenden Sachverhalt als UML-Klassendiagramm:
  - Ein Dialogsystem soll die Verwendung folgender Fenstertypen erlauben:
    - einfache Fenster, ohne Zusatzfunktionalität
    - Fenster die eine Titelleiste haben
    - Fenster die eine Statusleiste haben
    - Fenster die horizontal und vertikal “scrollbar” sind
    - alle daraus konstruierbaren “Featurekombinationen”, wie z.B. ein Fenster mit Titelleiste das horizontal und vertikal “scrollbar” ist

Lösung 1: Vererbung

- Probleme:
  - Explosion der Klassenhierarchie
  - TitleStatusScrollBarWindow versus ScrollStatusTitleBarWindow
  - Was passiert, wenn weitere Features (z.B. ColoredTitleBars, 3DScrollBars,…) berücksichtigt werden müssen?
  - Zur Laufzeit nicht änderbar

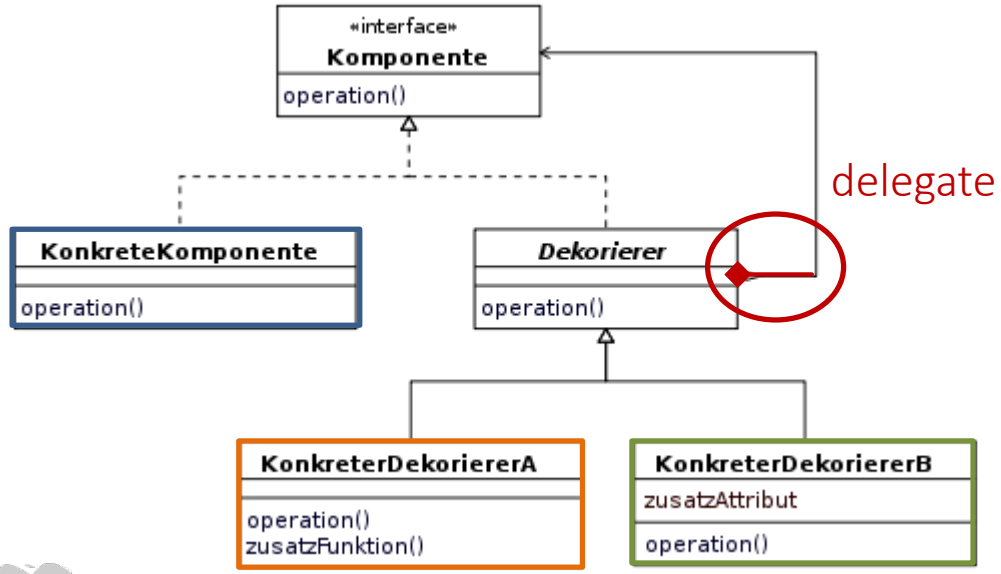


Decorator – Structural Pattern I

Beschreibung	Inhalt
Pattern Name und Klassifikation	Decorator – Structural Pattern
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 – Structural Pattern II

- Struktur:
  - Instanz eines Dekorierers wird vor die zu dekorierende Klasse geschaltet -> Funktionalität des Dokorierers wird zuerst ausgeführt
  - Dekorierer hat gleiche Schnittstelle wie zu dekorierende Klasse
  - Aufrufe werden weitergeleitet oder komplett selbst verarbeitet

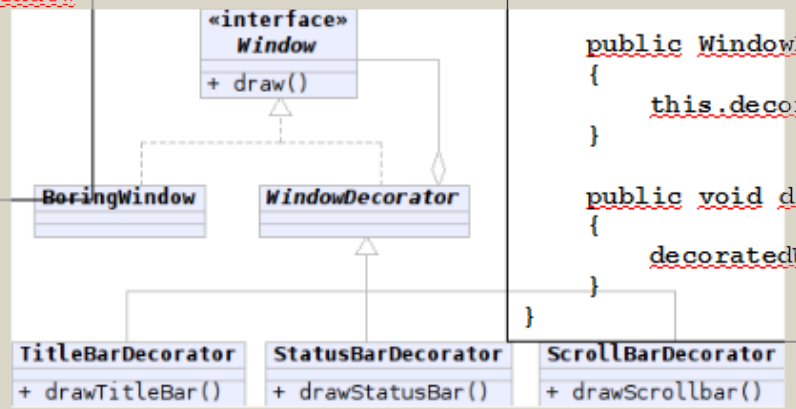


Klient ruft operation() auf und wird durch alle Objekte delegiert

Bessere Lösung: Decorator Pattern

```
interface Window
{
    public void draw();
}
```

```
class BoringWindow implements Window
{
    public void draw()
    {
        // draw window
    }
}
```



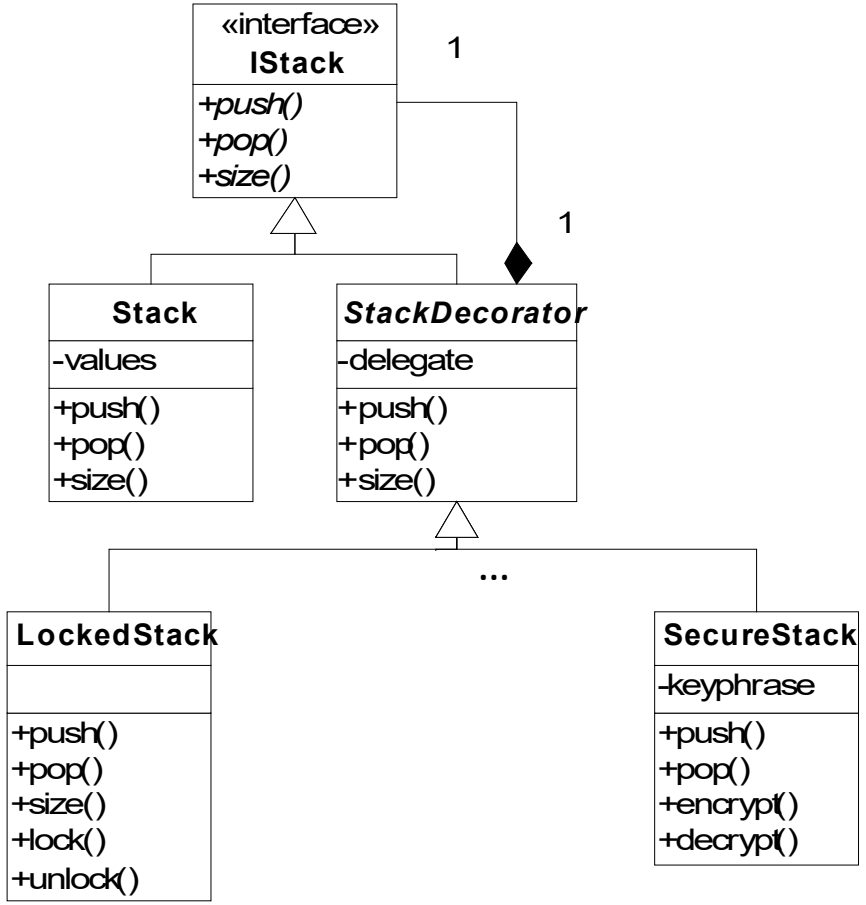

```
abstract class WindowDecorator
    implements Window
{
    // the window being decorated
    protected Window decoratedWindow;

    public WindowDecorator(Window win)
    {
        this.decoratedWindow = win;
    }

    public void draw()
    {
        decoratedWindow.draw();
    }
}
```

```
class ScrollBarDecorator extends WindowDecorator
{
    public ScrollBarDecorator(Window window)
    {
        super(window);
    }

    public void draw() {
        drawScrollBar();
        super.draw();
    }

    private void drawScrollBar()
    {
        // draw the scrollbar
    }
}
```

```
public class DecoratedWindowTest
{
    public static void main(String[] args)
    {
        // create a fully decorated Window
        Window decoratedWindow = new TitleBarDecorator(
            new StatusBarDecorator(
                new ScrollBarDecorator(
                    new BoringWindow())));

        // window with title, status, and scrollbars
        decoratedWindow.draw();
    }
}
```

Decorator – Structural Pattern IV

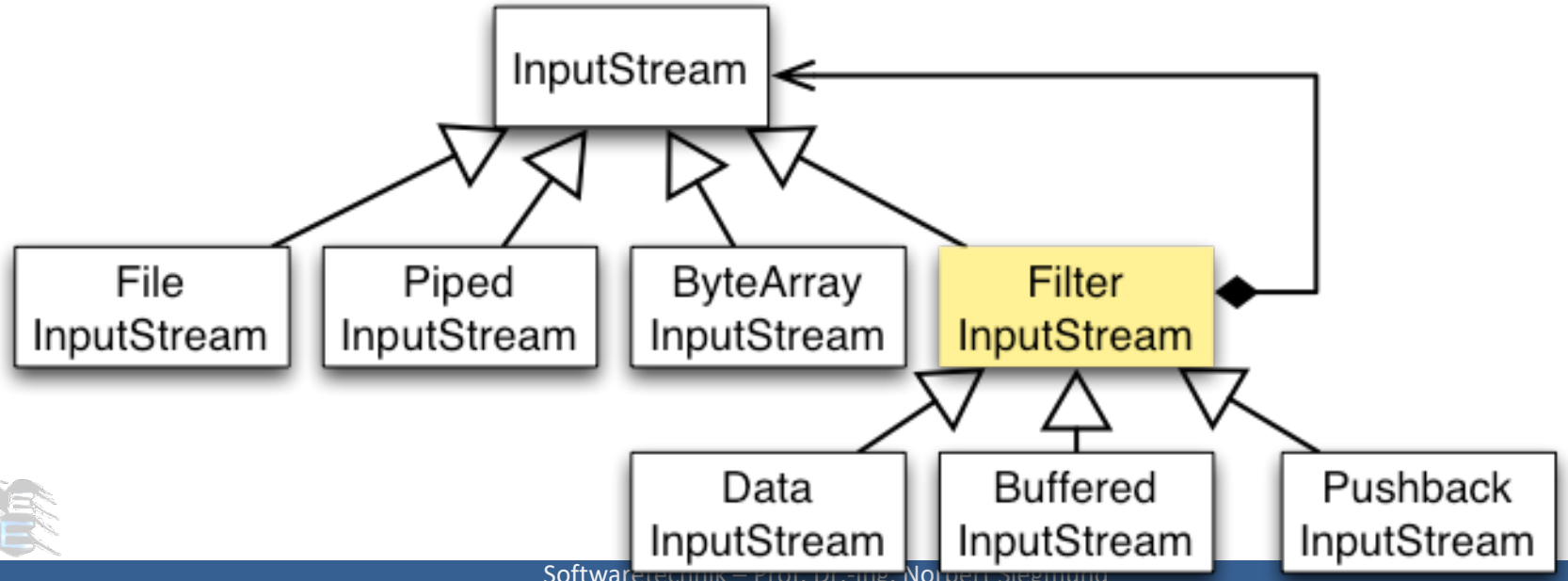
- Struktur des Beispiels



Decorator in Java

- java.io enthält verschiedene 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));
```

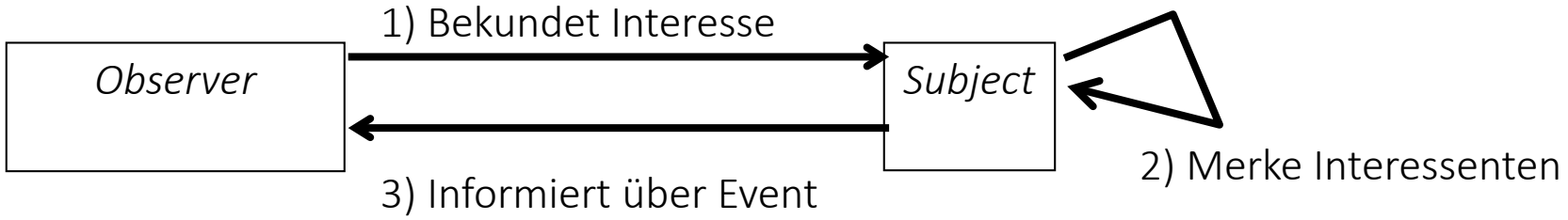


Observer – Behavioral Pattern I

Beschreibung	Inhalt
Pattern Name und Klassifikation	Observer – Behavioral Pattern
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) Architektur-Pattern
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 willl, ohne dass es die Anderen kennt.

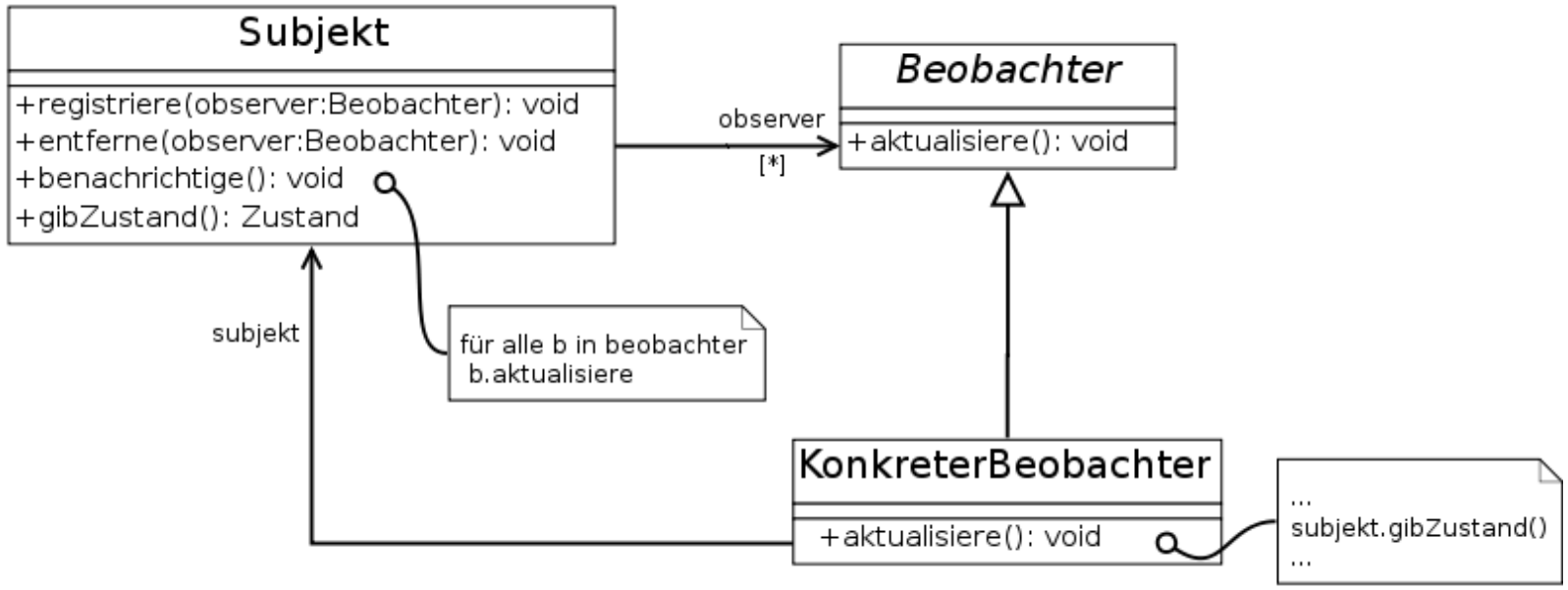
Observer – Behavioral Pattern II

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



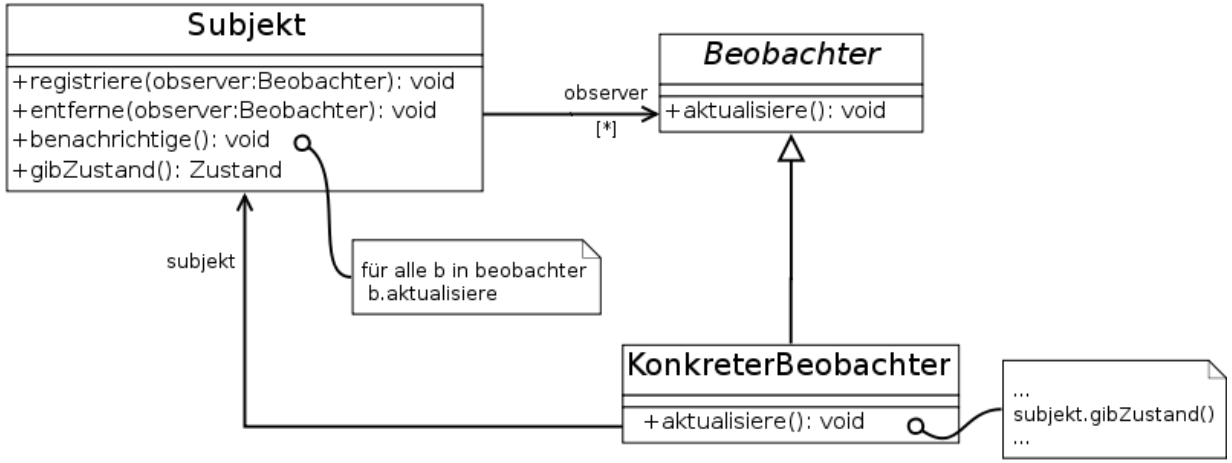
Observer – Behavioral Pattern III

- Struktur



Observer – Behavioral Pattern IV

- Struktur
- Code




```
interface IObserver {
    public void update(String message);
}
interface ISubject {
    public void registerObserver(Observer observer);
    public void removeObserver(Observer observer);
    public void notifyObservers();
}
```

Aufgabe

- Wie sieht die Realisierung des Observer Patterns aus für eine Client-Server Kommunikation?
  - Mehrere unterschiedliche Clients verbinden sich mit Server und wollen über Änderungen informiert werden

Observer – Behavioral Pattern V

```
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 message) {
       System.out.println("Receiving message: " + message);
  }
}
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!");
   }
}
```

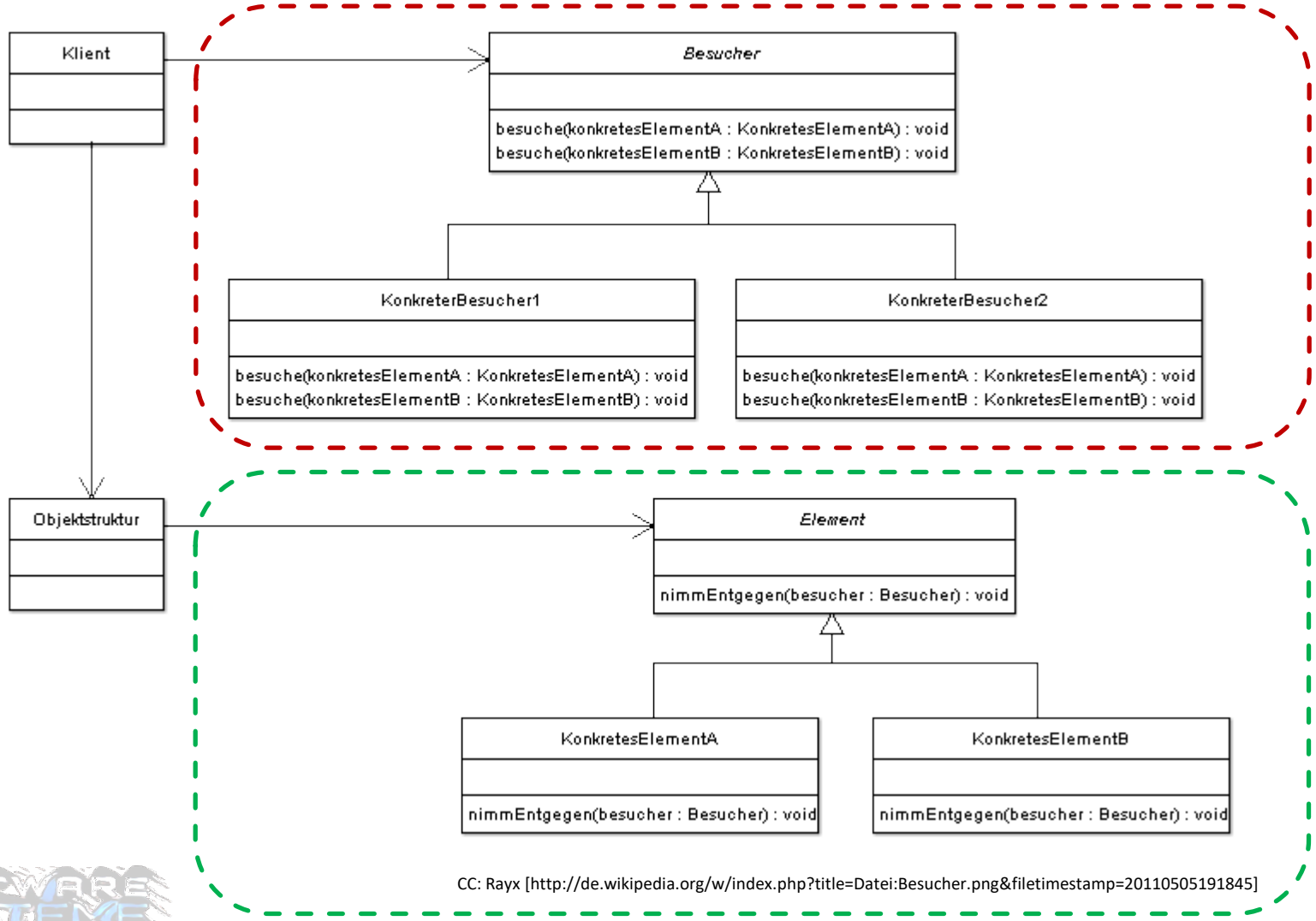
Visitor – Behavioral Pattern

Beschreibung	Inhalt
Pattern Name und Klassifikation	Visitor – Behavioral Pattern
Zweck	Trennung von Algorithmus und Daten auf denen der Algorithmus angewendet wird
Motivation	Durch die Trennung können neue Algorithmen / Funktionen auf existierenden Objekt(-strukturen) angewendet werden, ohne diese Objekte/Strukturen ändern zu müssen.
Anwendbarkeit	Struktur mit vielen Klassen vorhanden. Man möchte Funktionen anwenden, die abhängig von der jeweiligen Klasse sind. Menge der Klassen ist stabil. Man möchte neue Operationen hinzufügen.

Visitor – Behavioral Pattern II

Beschreibung	Inhalt
Konsequenzen	Einfach neue Operationen hinzufügen. Gruppiert verwandte Operationen in einem Visitor. Neue Elemente hinzufügen ist schwierig. Visitor kann Zustand speichern. Elemente müssen ein Interface bereitstellen / impl.

Visitor – Behavioral Pattern III



Einfaches Hinzufügen neuer Algorithmen auf die Datenstruktur

Statische Datenstruktur, die möglichst nicht geändert wird

CC: Rayx [http://de.wikipedia.org/w/index.php?title=Datei:Besucher.png&filetimestamp=20110505191845]

Aufgabe

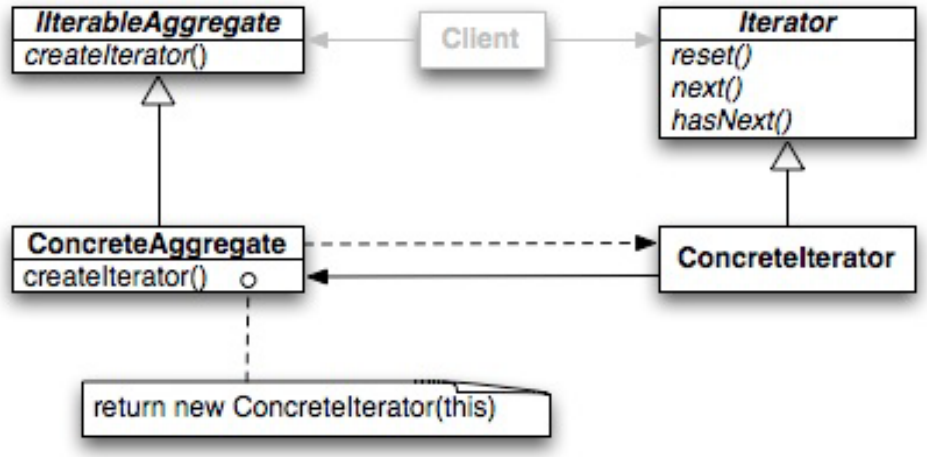
- Warum ist das Hinzufügen neuer Elemente schwierig?

Iterator – Behavioral Pattern I

Beschreibung	Inhalt
Pattern Name	Iterator – Behavioral Pattern
Zweck	Sequentieller Zugriff auf Elemente einer aggregierten Struktur, ohne diese zu kennen.
Auch bekannt als	Cursor
Motivation	Objekte werden häufige in einer Sammlung (z.B. Liste) zusammengefasst. Auf Elemente dieser Sammlung soll möglichst generisch und ohne Kenntnis von Implementierungsdetails zugegriffen werden können.
Anwendbarkeit	Elemente in Sammlung zusammengefasst. Zugriff auf Elemente unabhängig von der Implementierung der Sammlung.
Konsequenzen	Implementierung der Sammlung unsichtbar. Wenn Sammlung sich während Iteration ändert, kann es zu Fehlern kommen.

Iterator – Behavioral Pattern II

- Struktur
  - Aggregate = Sammlung von Elementen
  - IterableAggregate = Interface um Sammlung zu iterieren (spezifiziert nicht, welche Datenstruktur als Sammlung genutzt werden soll)
  - ConcreteAggregate = Implementierung dieses Interfaces
  - createIterator() gibt Objekt vom Typ ConcreteIterator zurück
  - ConcreteIterator implementiert Iterator Interface



Iterator – Behavioral Pattern III

```
public interface IIterableAggregate {
      public IIterator createIterator();
}
public interface IIterator{
      public void remove(); //Löscht zuletzt zurück gegebenes Element aus Sammlung
      public Object next(); //Gibt nächstes Objekt zurück und setzt den Zeiger weiter in der Sammlung
      public boolean hasNext(); //Gibt true zurück, falls es noch weitere Elemente in der Sammlung gibt
}
```

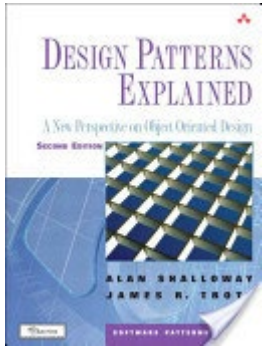
```
public class SimpleList<T> implements IIterableAggregate {
   private ArrayList<T> sammlung = new ArrayList<T>();
  @Override
   public IIterator createIterator() {
       return new SimpleListIterator(sammlung); }
  … // Add, delete, etc.
}
public class SimpleListIterator<T> implements IIterator{
      int index = -1;      ArrayList<T> sammlung; // Im Konstruktor initialisieren
      public Object next() {
                  index++;
                   return sammlung[index]; }
      public boolean hasNext() {
                  return (index < (sammlung.length -1)); }
                              //remove() nicht hier gezeigt, muss aber implementiert werden
}
```

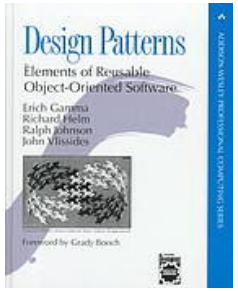
Was Sie mitgenommen haben sollten

- Warum sind Design Patterns wichtig?
- Was sind folgende Pattern und wie realisiere ich sie?
  - Adapter
  - Iterator
  - Decorator
  - Observer
  - Visitor
- Was verbessern Design Patterns und wie sind sie klassifiziert?

Literatur

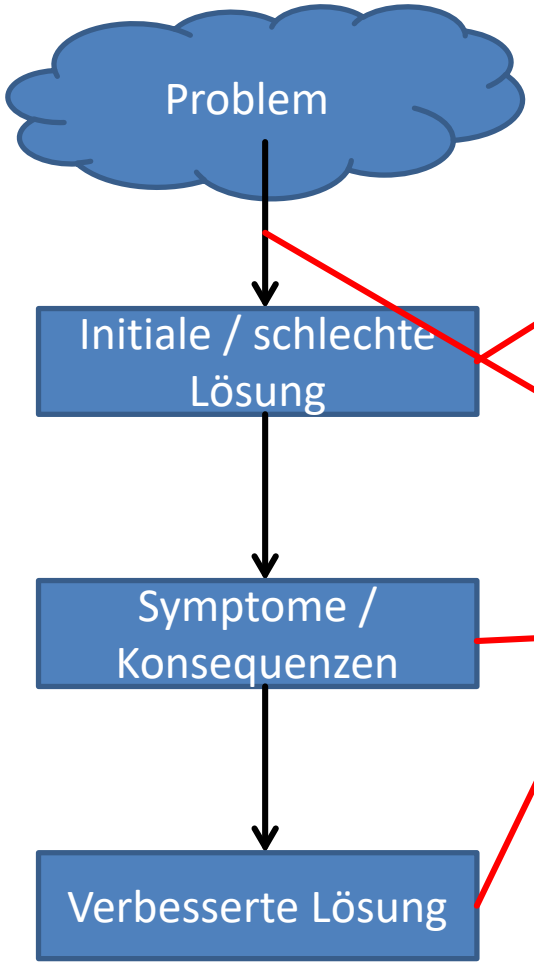
- *Design Patterns. Elements of Reusable Object-Oriented Software,* Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995.
- *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Alan Shalloway, James R. Trott, 2004.
  - *Benutzt Java*
  - *Viele Beispiele -> Lesenswert!*
- *Head First Design Patterns / Entwurfsmuster von Kopf bis Fuß.* Eric Freeman, Elisabeth Robson, O'Reilly, 2021.
- Quelle Diagramme: Wikipedia





Anti-Patterns…

Anti-Patterns



- Anti-Patterns 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?”
- Designdokumente 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.“

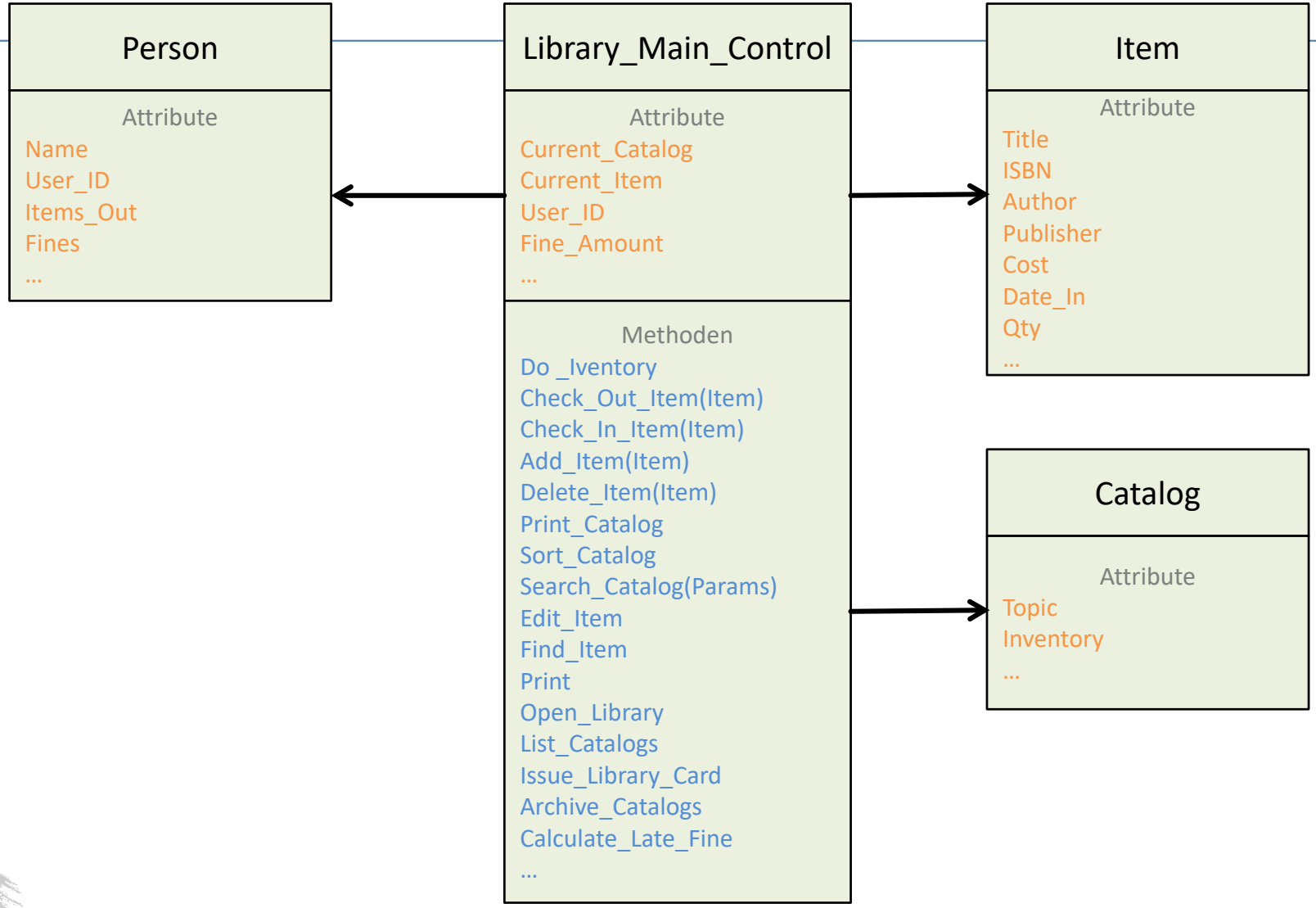
TheBlob

„Diese Klasse ist das Herzstück unserer Architektur.“

The Blob I

Beschreibung	Inhalt
Pattern Name	The Blob – Anti Pattern
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.

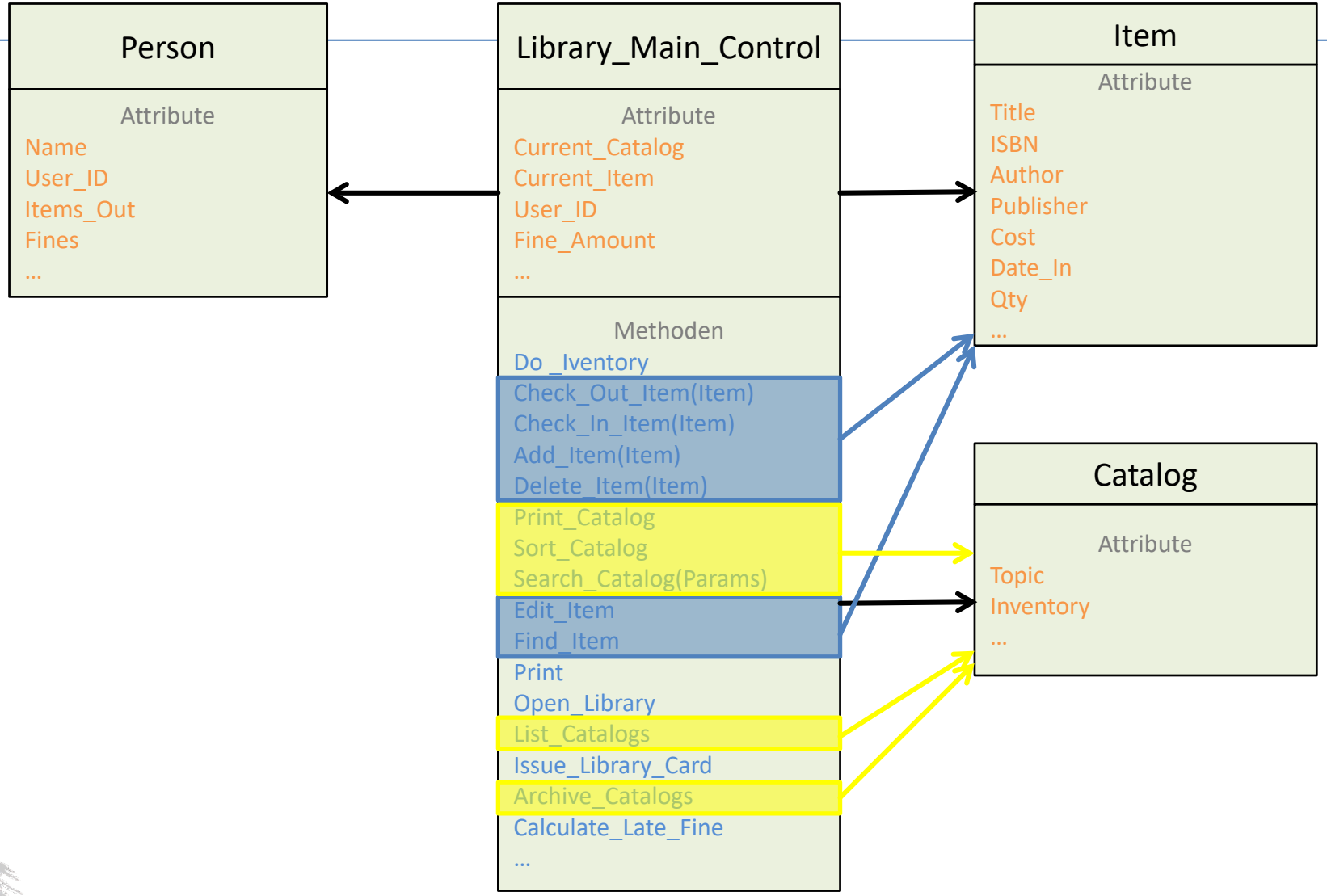
The Blob II


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(Params)
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
…

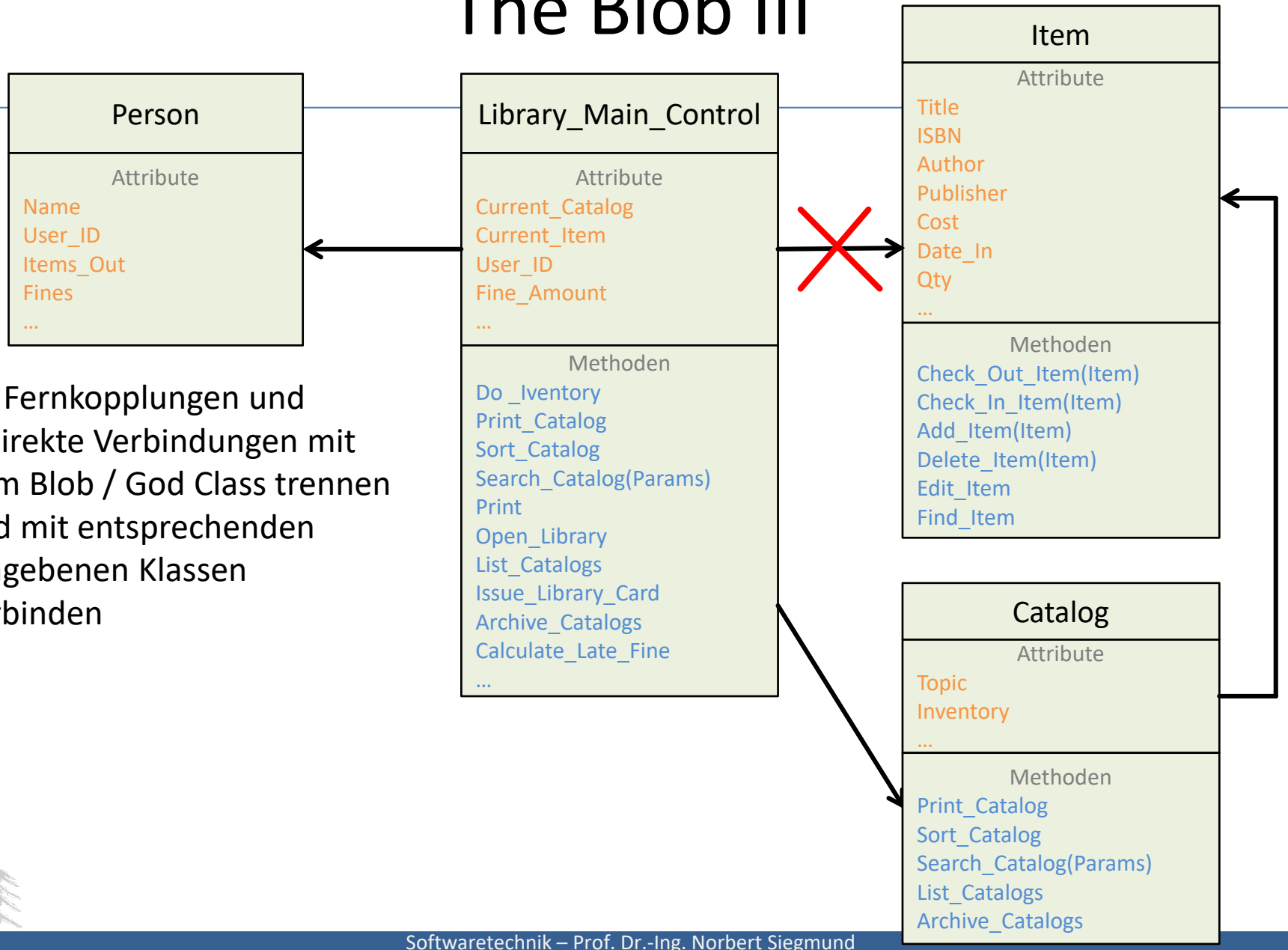
Aufgabe

- (1) Finde zusammenhängende Attribute und Methoden im Blop / God Class und gruppiere diese
- (2) Lagere die Gruppen in passende umgebene Klassen aus

The Blob III


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(Params)
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 III



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

Weitere Anti-Pattern

- Lava Flow (Dead Code)

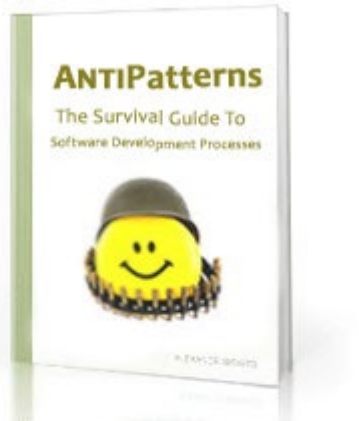
```
// This class was written by someone earlier (Alex?) to manage the indexing
// or something (maybe).  It's probably important.  Don't delete.  I don't
// think it's used anywhere - at least not in the new MacroINdexer module which
// may actually replace whatever this was used for…
class IndexFrame extends Frame
{
   // IndexFrame constructor
   //-------------------------------------------------------------------------
   public IndexFrame(String index_parameter_1)
   {
   // Note: need to add additional stuff here…
   super (str);
   }
   //-------------------------------------------------------------------------
```

- Spaghetti Code

Literatur

AntiPatterns: The Survival Guide

Only available as online PDF:
http://sourcemaking.com/antipatterns-book



Find more in the web

http://www.antipatterns.com/