Übung 06: Entwurfsmuster

Basierend auf dem Material von Oscar Nierstrasz, Sven Apel,
Janet Siegmund, Norbert Siegmund, Timo Kehrer, Ina Schaefer

Organisatorisches

- Programmieraufgabe

3313015 SE/MMSE (WiSe 2024/25) / Abschnitte / Übungsabschnitt 06 / Abgabe Blatt 06

Abgabe Blatt 06

Aufgabe | Einstellungen | Erweiterte Bewertung | Mehr

Hinweise beachten und bereitgestellten Quellcode verwenden!

Fällig: Fr., 20. Dezember 2024, 10:00

Hinweise zu den Materialien und Abgabe dieses Blatts:

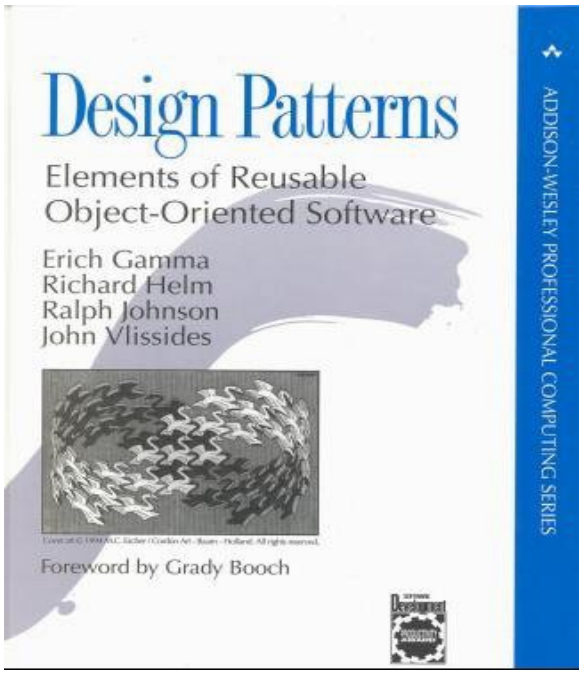
- In Moodle steht ein ZIP-Paket namens Uebung_06_Gruppe_YY.zip zur Verfügung. Dieses Paket beinhaltet einen Ordner bzw. ein Softwareprojekt namens Uebung_06_Gruppe_YY, das in Eclipse oder IntelliJ geöffnet/importiert werden kann. Dieses Projekt umfasst den in den Aufgabenstellungen verwendeten Quellcode.
- Benennen Sie das Projekt und dadurch auch den Ordner entsprechend Ihrer Gruppe um, indem Sie den Platzhalter YY durch Ihre Gruppennummer ersetzen.
- Ergänzen Sie den bereitgestellten Quellcode in den jeweiligen Java-Paketen für jede Aufgabe um den vollständigen Java-Quellcode Ihrer Lösung. Zu jeder Aufgabe muss Ihr Quellcode ausführbar sein. Ergänzen bzw. stellen Sie entsprechend für jede Aufgabe eine Klasse mit einer Main-Methode bereit, die Ihre Lösung beispielhaft ausführt.
- Die Verwendung von Bibliotheken außer des JDKs ist nicht erlaubt.
- Der abgegebene Quellcode darf keine Fehler bei der Kompilierung haben. Andernfalls wird der Programmierteil der entsprechenden Aufgabe mit 0 Punkte bewertet.
- Ihr Code muss mindestens auf gruenau5 kompiliert und ausgeführt werden können.
- Bei Code-Plagiaten wird dieses Blatt für alle beteiligten Gruppen mit 0 Punkten bewertet.
- Erstellen Sie zur Abgabe Ihrer Lösung ein ZIP-Paket namens Uebung_06_Gruppe_YY.zip aus dem Ordner Uebung_06_Gruppe_YY, wobei YY durch Ihre Gruppennummer ersetzt wurde.
- Bitte geben Sie dieses ZIP-Paket über Moodle ab.

Entwurfsmuster (Design Patterns)

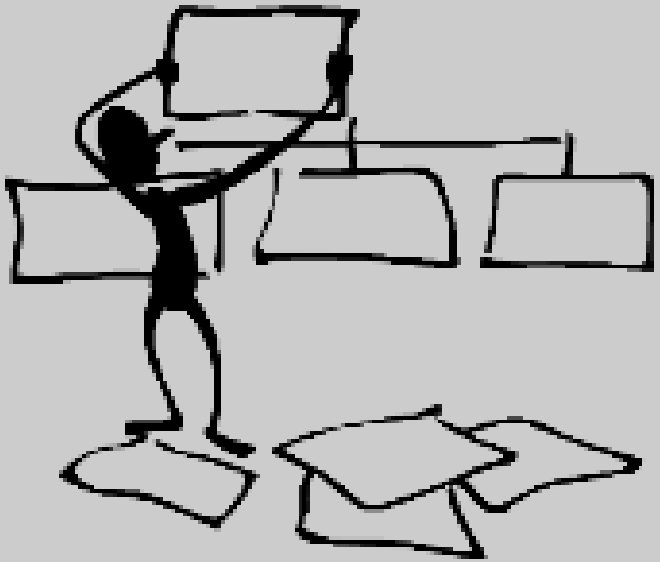
"Designing object-oriented software is hard and designing reusable object-oriented software is even harder."

—Erich Gamma et al.

- Ein **Entwurfsmuster** besteht aus
  - einem Kontext bzw. Situation, in dem ein wiederkehrendes Entwurfsproblem auftritt, und
  - einer generischen und bewährten Lösung, die dieses Problem meistern kann.
- **Objektorientierte Entwurfsmuster**
  - Folgen objektorientierte Entwurfsziele
    - Modularität, explizite Interfaces, Information Hiding, …
  - Verbesserung der Qualität und Wiederverwendung der Software
  - Katalog von 23 Mustern der „Gang of Four“ (GoF)



Anwendung eines Musters

Musteranwendung	
Kein mechanisches "Pattern Matching"!	
Entwurfsmuster sind abstrakte Lösungen für „recurring problems“!!	
Eher **Übertragung der Idee** des Musters	
Grundstruktur des Musters sollte sich wiederfinden lassen, ggf. vorliegenden Entwurf etwas anpassen bzw. anders darstellen	
Auch Verhaltensschema muss im Code ähnlich zur Musteridee auftreten.	

Klassifikation

- 3 Kategorien
  - **Erzeugungsmuster** (Creational Patterns)
    - Helfen bei der Objekterstellung
  - **Strukturmuster** (Structural Patterns)
    - Helfen bei der Komposition von Klassen und Objekten
  - **Verhaltensmuster** (Behavioral Patterns)
    - Helfen bei der Interaktion von Klassen und Objekten und Kapselung von Verhalten
- 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 Patterns

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

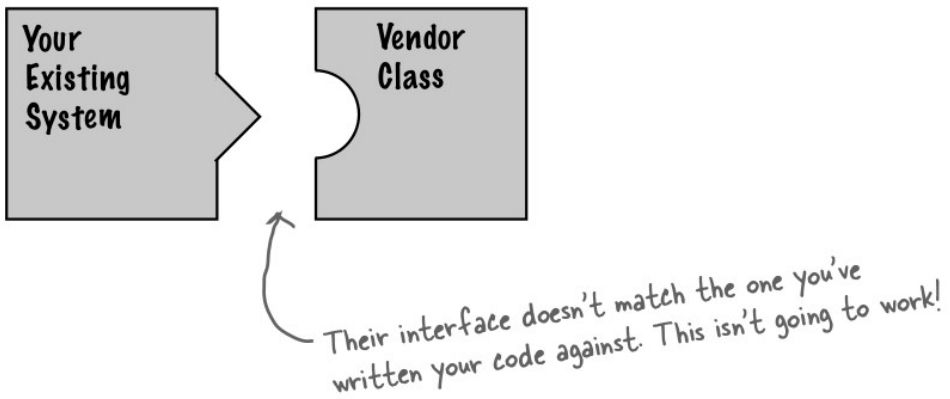
...ausführlicher als in der Vorlesung, jedoch betrachten wir auch nicht alles im Detail.

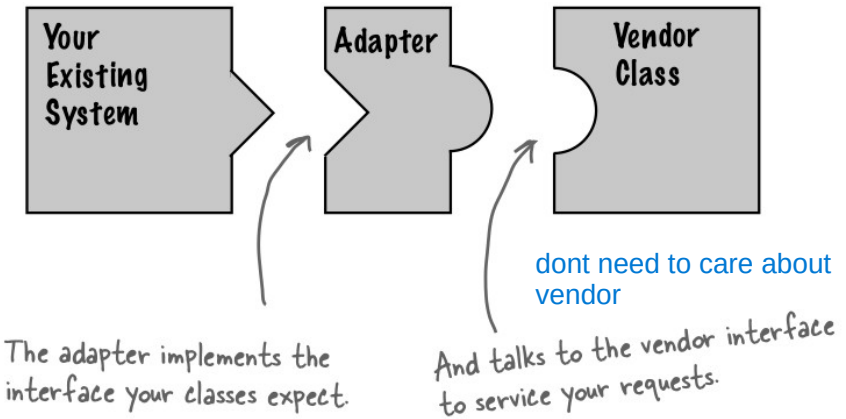
Wichtige Entwurfsmuster

- **Adapter**
- **Decorator**
- **Visitor**
- Observer
- Singleton
- Composite
- Command

Adapter: Übersicht

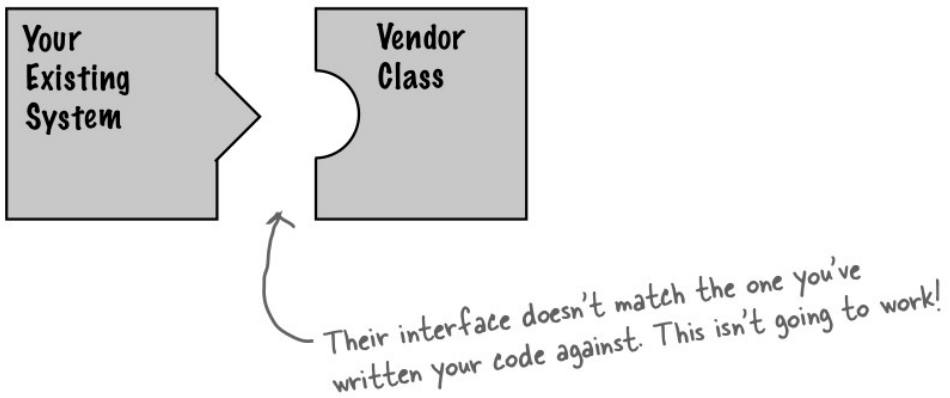
Beschreibung	Inhalt
Pattern-Name + Klassifikation	Adapter – Strukturmuster
Zweck	Konvertiert das Interface einer existierenden Klasse, so dass es zu dem Interface eines Klienten (Client) passt. Ermöglicht, dass Klassen miteinander interagieren können, was sonst nicht möglich wäre aufgrund der Unterschiede 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.

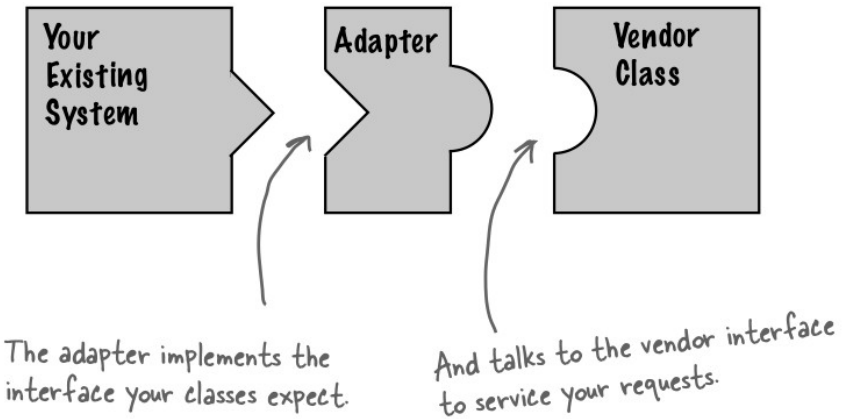




Adapter: Übersicht II

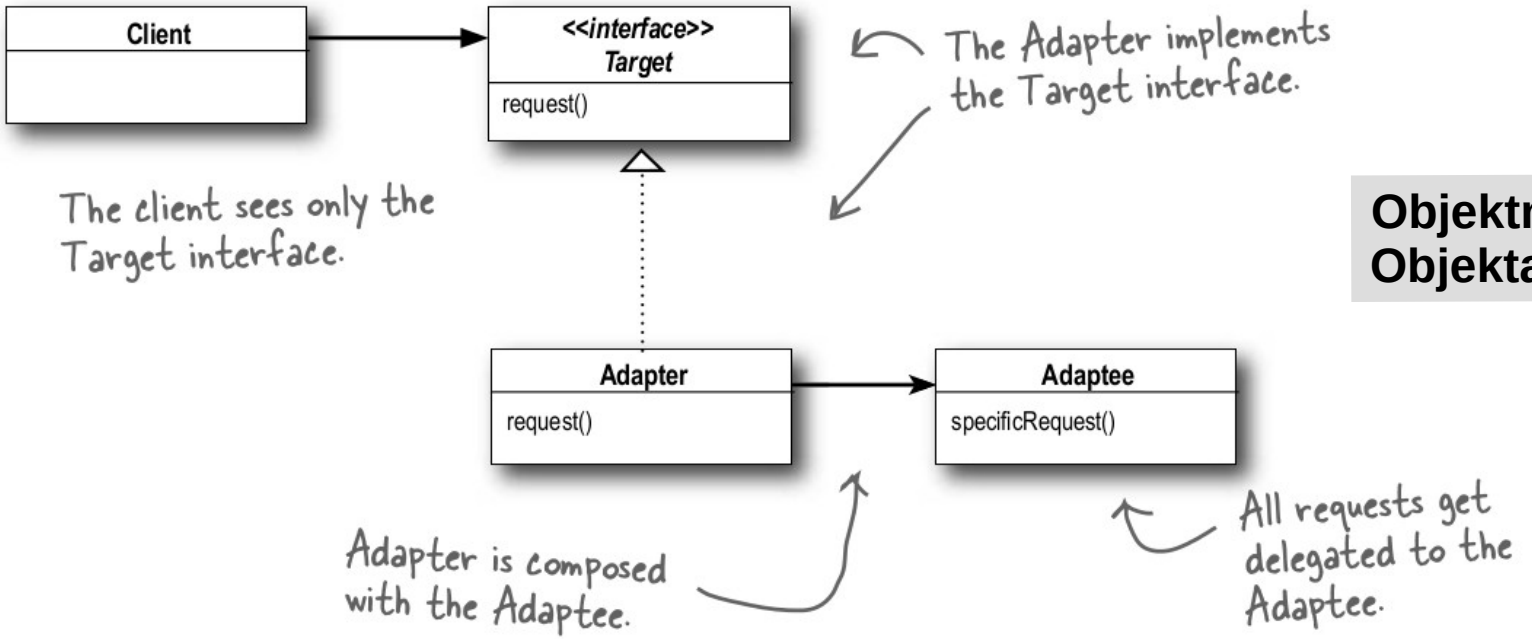
Beschreibung	Inhalt
Anwendbarkeit	- Verwende eine ansonsten nicht wiederverwendbare (durch Interface-Inkompatibilität) Klasse wieder: Adaptiere das Interface durch das Ändern der Methodensignaturen im Adapter. - Existierende Klasse bietet nicht die benötigte Funktionalität an: Implementiere die benötigte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





Adapter: Struktur und Teilnehmer/Rollen

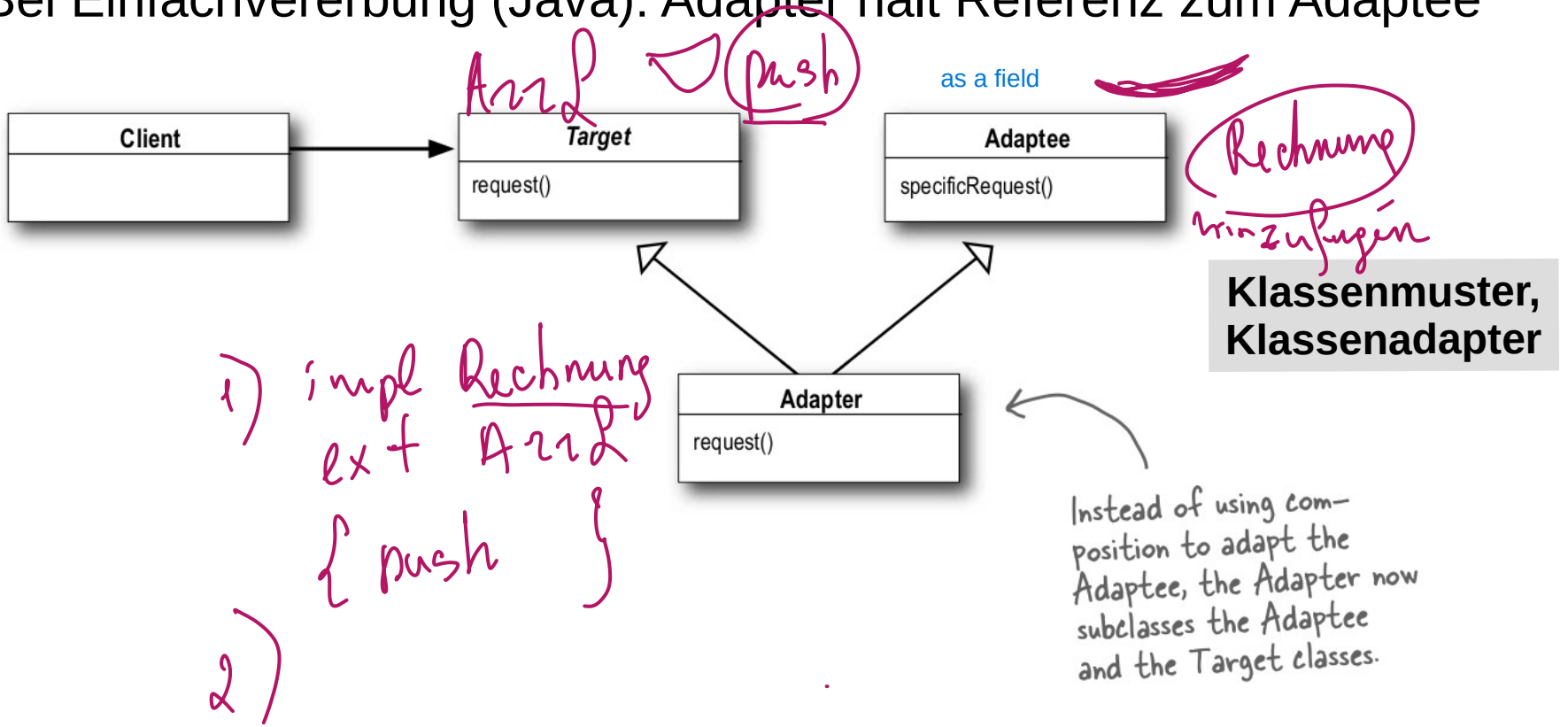
- **Rollen:** Client, Target, Adapter, Adaptee
- Adapter hält eine Referenz auf Adaptee (zu adaptierende Klasse)
- Adapter nutzt Delegation, um Aufrufe an Adaptee weiterzuleiten



Objektmuster, Objektadapter

Adapter: Struktur und Teilnehmer/Rollen II

- **Rollen:** *Client, Target, Adapter, Adaptee*
- Adapter erbt von Target und Adaptee
- Bei Einfachvererbung (Java): Adapter hält Referenz zum Adaptee



Klassenmuster, Klassenadapter

Adapter: Übersicht III

Beschreibung	Inhalt
Teilnehmer	**Target:** Definiert das (domänenspezifische) Interface, das der Client nutzt. **Client:** Interagiert mit Objekten, die das Target-Interface implementieren. **Adaptee (zu adaptierende Klasse):** Repräsentiert existierendes Interface, welches nicht kompatibel zum Target ist. **Adapter:** Adaptiert das Interface vom Adaptee, damit es kompatibel zum Target ist.
Konsequenzen	Bei Verwendung von Vererbung: Klasse Adapter – Überschreibung der Methoden der Superklasse (Adaptee) ist möglich, wenn vom Adaptee geerbt werden kann Anpassbarkeit hängt vom Unterschied der Interfaces zwischen Target und Adaptee ab
Implementierung	Siehe nächste Folie für ein Beispiel
Beispiel Code	Siehe nächste Folie für ein Beispiel
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

```java
public interface Rechnung {

    public void hinzufuegen(Rechnungsposition rp);

    public void loeschen(int pos);

    public void print();
}
```

```java
public class Rechnungsposition {

    private String beschreibung;

    public Rechnungsposition(String beschreibung) {
        super();
        this.beschreibung = beschreibung;
    }

    public String getBeschreibung() {
        return beschreibung;
    }

    public void setBeschreibung(String beschreibung) {
        this.beschreibung = beschreibung;
    }
}
```

- Eine Rechnung umfasst eine Liste von Rechnungspositionen
- Implementierung des Interface `Rechnung`
  - kann eine eigene Liste implementieren (aufwendig)
  - eher Nutzung einer vorhanden Implementierung der Liste `java.util.List<E>`, z.B. `ArrayList`, `LinkedList` etc.

Adapter: Beispiel II

Target

```
public interface Rechnung {

    public void hinzufuegen(Rechnungsposition rp);

    public void loeschen(int pos);

    public void print();
}
```

Adaptee

```
public interface List<E> extends Collection<E> {
    boolean add(E e);
    E remove(int index);
```

Ausschnitt

Client

```
public class Client {

    public static void main(String[] args) {
        Rechnung rechnung = new RechnungImpl();
        rechnung.hinzufuegen(new Rechnungsposition("Pizza"));
        rechnung.hinzufuegen(new Rechnungsposition("Cola"));
        rechnung.print();
    }
}
```

- **Problem:** Client nutzt das Interface `Rechnung`, das sich vom Interface `java.util.List<E>` unterscheidet

Adapter: Beispiel III

Target

```
public interface Rechnung {

    public void hinzufuegen(Rechnungsposition rp);

    public void loeschen(int pos);

    public void print();
}
```

Adaptee

```
public interface List<E> extends Collection<E> {
    boolean add(E e);
    E remove(int index);
```

Ausschnitt

ArrayList implements List<E>

Adapter

```
public class RechnungImpl extends ArrayList<Rechnungsposition> implements Rechnung {

    @Override
    public void hinzufuegen(Rechnungsposition rp) {
        super.add(rp);
    }

    @Override
    public void loeschen(int pos) {
        super.remove(pos);
    }

    @Override
    public void print() {
        for (Iterator<Rechnungsposition> iterator = this.iterator(); iterator.hasNext();) {
            System.out.println(iterator.next().getBeschreibung());
        }
    }

}
```

Adapter: Beispiel IV

Klassenmuster, Klassenadapter

Target

```
public interface Rechnung {

    public void hinzufuegen(Rechnungsposition rp);

    public void loeschen(int pos);

    public void print();
}
```

Adaptee

```
public interface List<E> extends Collection<E> {
    boolean add(E e);
    E remove(int index);
```

Ausschnitt

ArrayList implements List<E>

Adapter

```
public class RechnungImpl extends ArrayList<Rechnungsposition> implements Rechnung {

    @Override
    public void hinzufuegen(Rechnungsposition rp) {
        super.add(rp);
    }

    @Override
    public void loeschen(int pos) {
        super.remove(pos);
    }

    @Override
    public void print() {
        for (Iterator<Rechnungsposition> iterator = this.iterator(); iterator.hasNext();) {
            System.out.println(iterator.next().getBeschreibung());
        }
    }

}
```

Adapter: Beispiel V

Target

```
public interface Rechnung {

    public void hinzufuegen(Rechnungsposition rp);

    public void loeschen(int pos);

    public void print();
}
```

Adaptee

```
public interface List<E> extends Collection<E> {
    boolean add(E e);
    E remove(int index);
```

Ausschnitt

Adapter

```
public class RechnungImpl implements Rechnung {

    private List<Rechnungsposition> adaptee = new ArrayList<Rechnungsposition>();

    @Override
    public void hinzufuegen(Rechnungsposition rp) {
        adaptee.add(rp);
    }

    @Override
    public void loeschen(int pos) {
        adaptee.remove(pos);
    }

    @Override
    public void print() {
        for (Iterator<Rechnungsposition> iterator = adaptee.iterator(); iterator.hasNext();) {
            System.out.println(iterator.next().getBeschreibung());
        }
    }

}
```

Adapter: Beispiel VI

Objektmuster, Objektadapter

Target

```
public interface Rechnung {

    public void hinzufuegen(Rechnungsposition rp);

    public void loeschen(int pos);

    public void print();
}
```

Adaptee

```
public interface List<E> extends Collection<E> {
    boolean add(E e);
    E remove(int index);
```

Ausschnitt

Adapter

```
public class RechnungImpl implements Rechnung {

    private List<Rechnungsposition> adaptee = new ArrayList<Rechnungsposition>();

    @Override
    public void hinzufuegen(Rechnungsposition rp) {
        adaptee.add(rp);
    }

    @Override
    public void loeschen(int pos) {
        adaptee.remove(pos);
    }

    @Override
    public void print() {
        for (Iterator<Rechnungsposition> iterator = adaptee.iterator(); iterator.hasNext();) {
            System.out.println(iterator.next().getBeschreibung());
        }
    }

}
```

Decorator: Übersicht

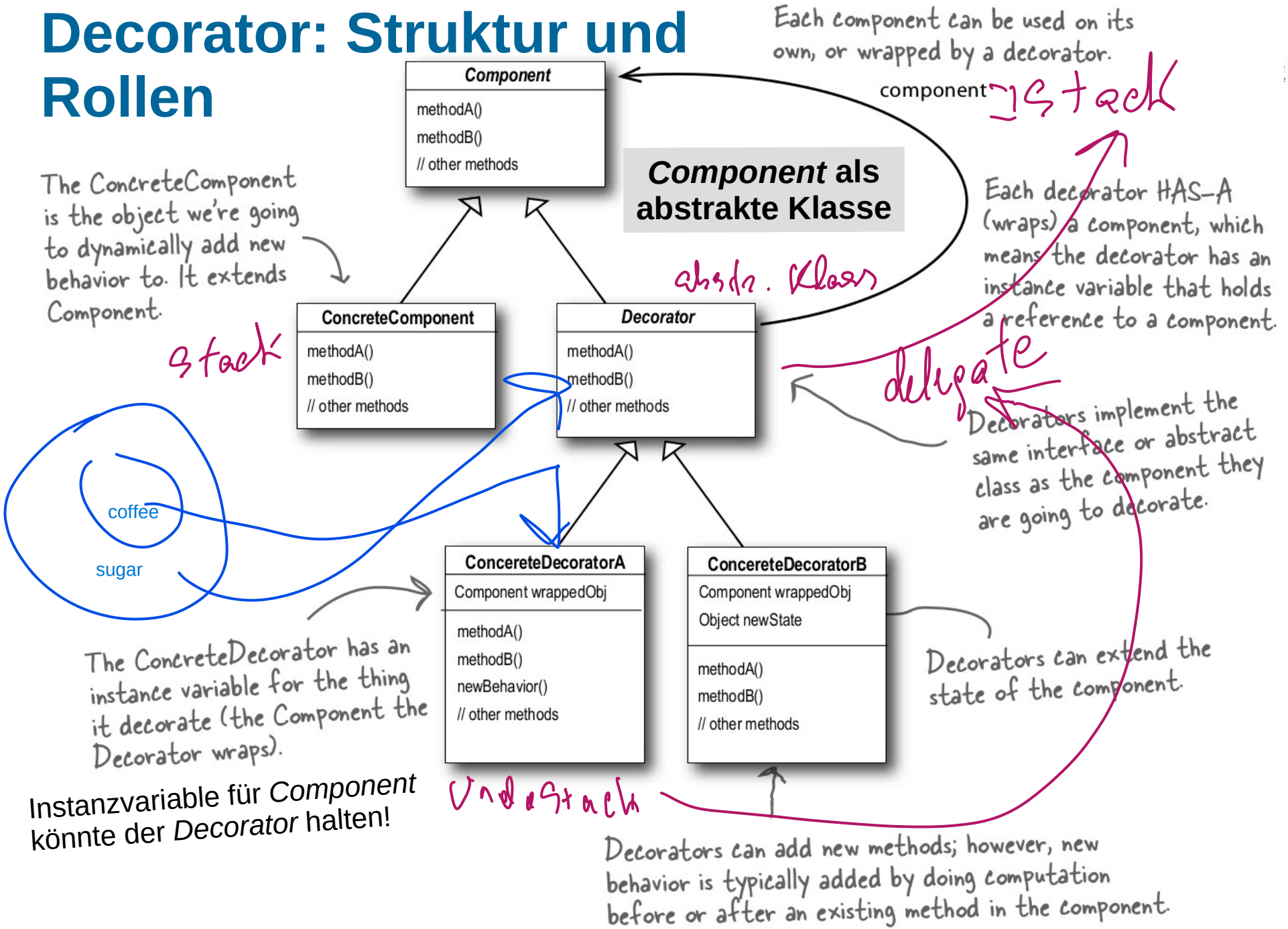
Beschreibung	Inhalt
Pattern-Name + Klassifikation	Decorator – Strukturmuster
Zweck	Fügt flexibel und dynamisch Funktionalität und Eigenschaften zu bereits bestehenden Klassen hinzu.
Motivation	Wir benötigen flexible Implementierungen einer Klasse, die je nach Kontext unterschiedlich ausfallen.
Anwendbarkeit	Erweiterungen sind optional. Anwendbar, wenn Erweiterungen mittels Vererbung unpraktisch ist, da Vererbungsbeziehungen für alle Objekte fest vorgegeben sind (z.B. bei einer Vielzahl von unabhängigen Erweiterungen wäre eine höhere Anzahl an Unterklassen notwendig, um alle möglichen Kombinationen an Erweiterungen abzudecken).
Konsequenzen	Flexibler als (statische) Vererbung. Problem der Objektschizophrenie (ein Objekt ist zusammengesetzt aus mehreren Objekten). Viele kleine Objekte.

cover all possible combinations

Decorator: Übersicht II

- **Zu dekorierende Klasse:** Klasse, die um Funktionalität oder Eigenschaften erweitert werden soll
- **Dekorierer:** Stellt die Erweiterung bereit
- Dekorierer hat gleiche Schnittstelle wie zu dekorierende Klasse
- Instanz eines Dekorierers wird vor die Instanz einer zu dekorierenden Klasse geschaltet
  - Aufrufe werden weitergeleitet oder komplett selbst verarbeitet
  - Funktionalität des Dokorierers kann vor und/oder nach der Weiterleitung ausgeführt werden
- Mehrere Dekorierer für ein Objekt möglich => ein Objekt ist zusammengesetzt aus mehreren Objekten
- **Rollen:**
  - Zu dekorierende Klasse: *Component, ConcreteComponent*
  - Dekorierer: *Decorator, ConcreteDecorator*

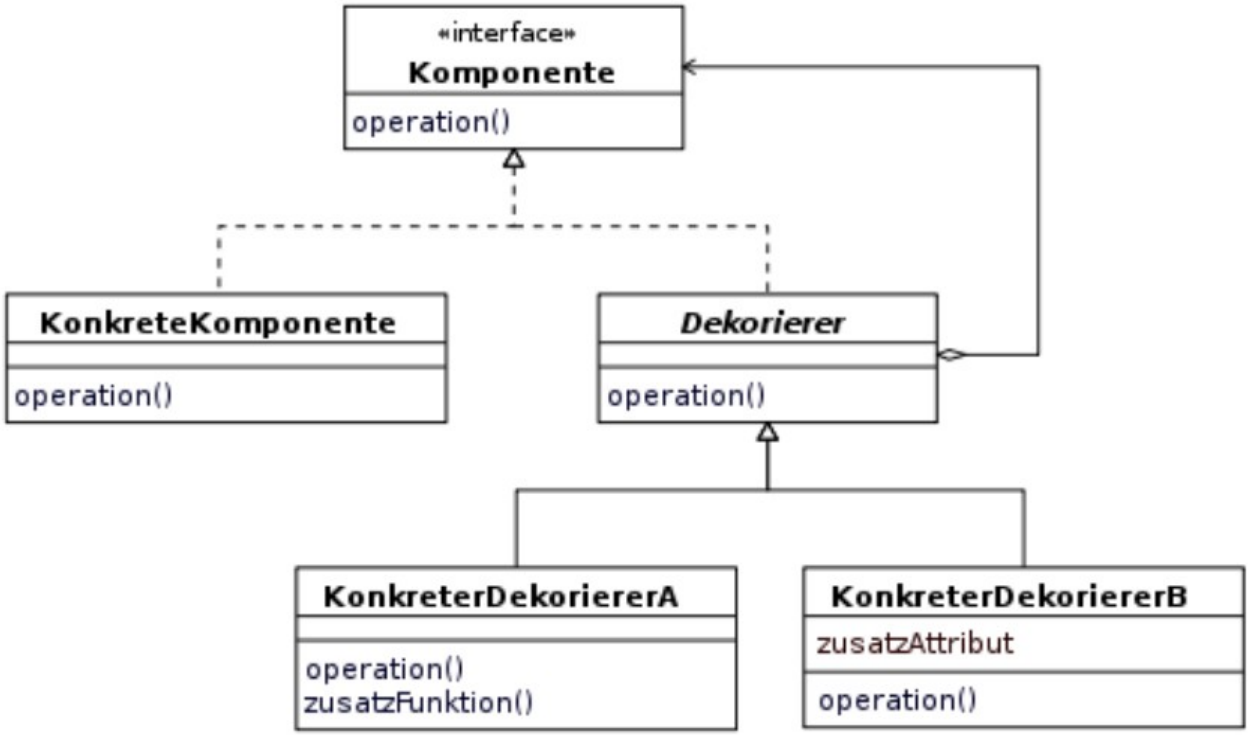
Decorator: Struktur und Rollen



Instanzvariable für *Component* könnte der *Decorator* halten!

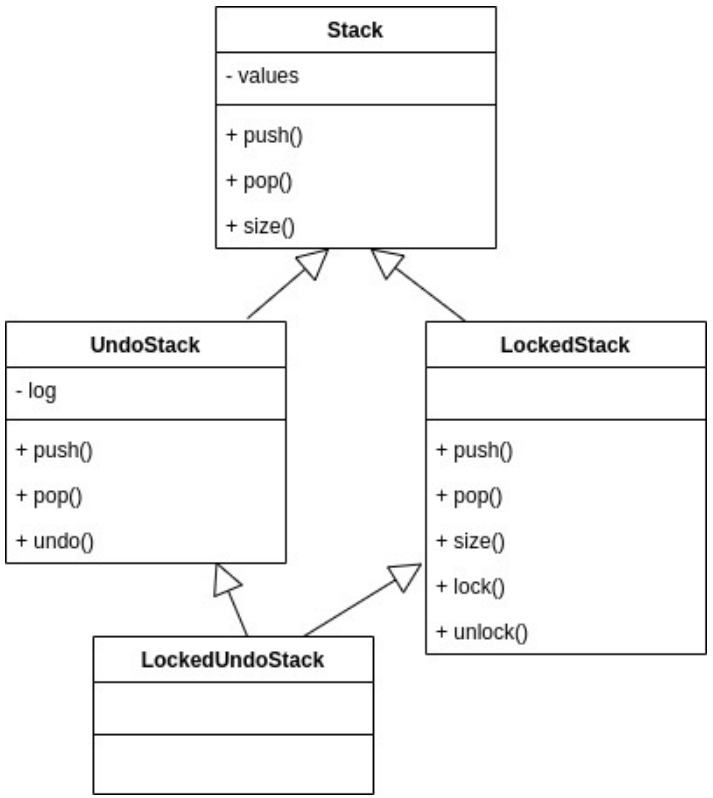
Decorator: Struktur und Rollen II

Component als Interface

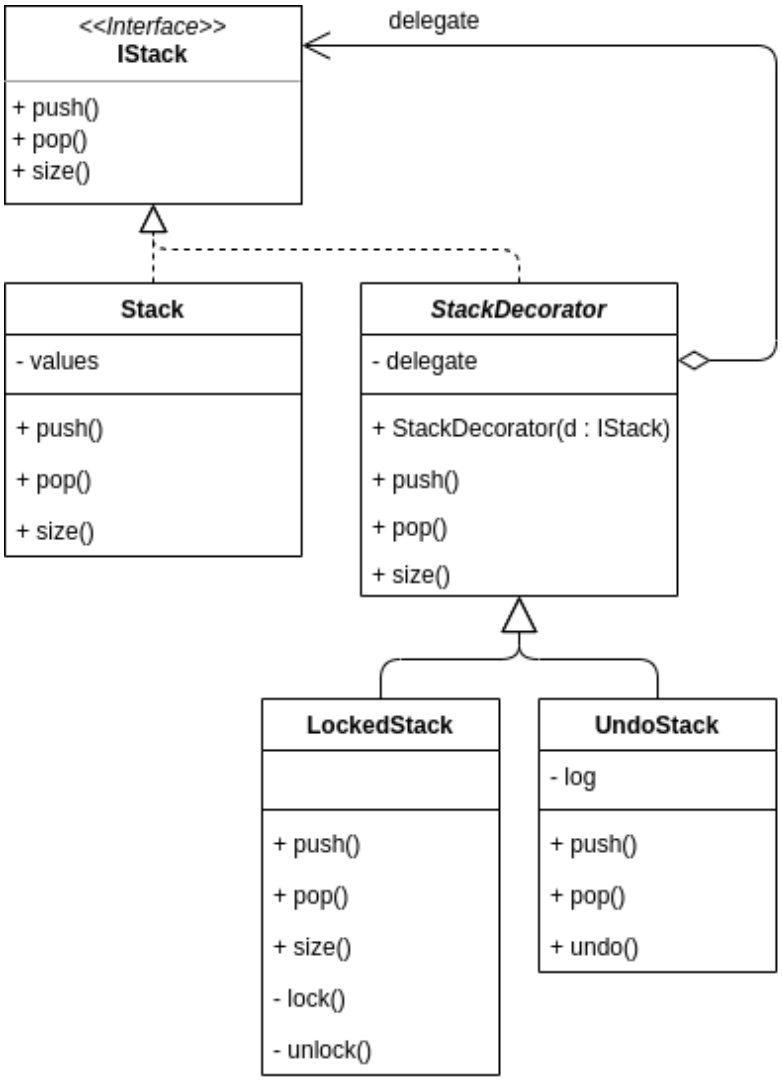


Decorator: Beispiel

- Stack soll Funktionalität für Undo und/oder Locking haben
- **Diamond Problem**
  - `new LockedUndoStack().pop()`
  - Unklarheit, welche `pop()`-Methode `LockedUndoStack` erbt: Von `UndoStack` oder `LockedStack`?



Lösung: Decorator Pattern



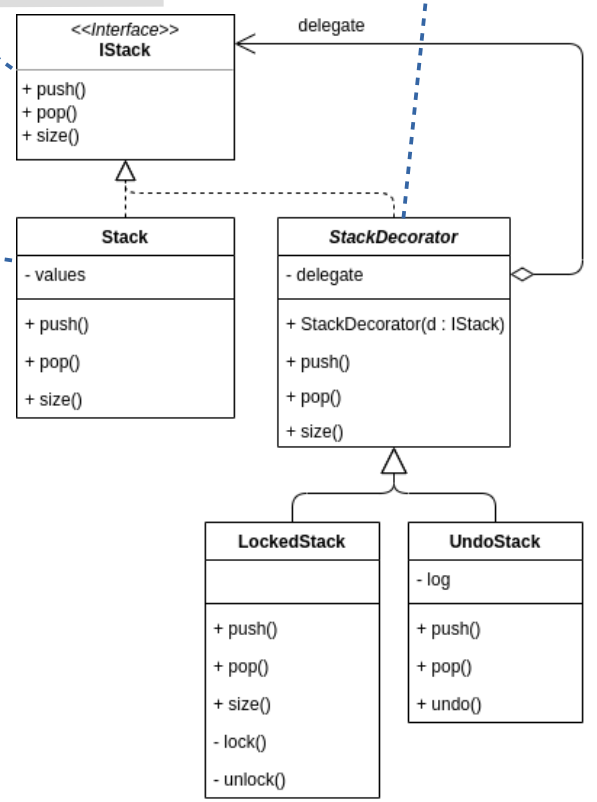
Decorator: Beispiel II

```java
public abstract class StackDecorator implements IStack {

    protected IStack delegate;

    public StackDecorator(IStack delegate) {
        this.delegate = delegate;
    }
}
```

Decorator

```java
public interface IStack {

    public void push(Object o);
}
```

Component




```java
public class Stack implements IStack {

    private List<Object> values = new ArrayList<Object>();

    @Override
    public void push(Object o) {
        System.out.println("push");
        values.add(o);
    }

}
```

Concrete Component

to decrease the tree depth or deleting decorator is easy for later maintenance

Vereinfachter Code: nicht alle Methoden vorhanden

Decorator: Beispiel III

```java
public class UndoStack extends StackDecorator {

    private List<String> log = new ArrayList<>();

    public UndoStack(IStack delegate) {
        super(delegate);
    }

    @Override
    public void push(Object o) {
        remember("push");
        delegate.push(o);
    }

    private void remember(String method) {
        System.out.println("remember " + method);
        log.add(method);
    }
}
```

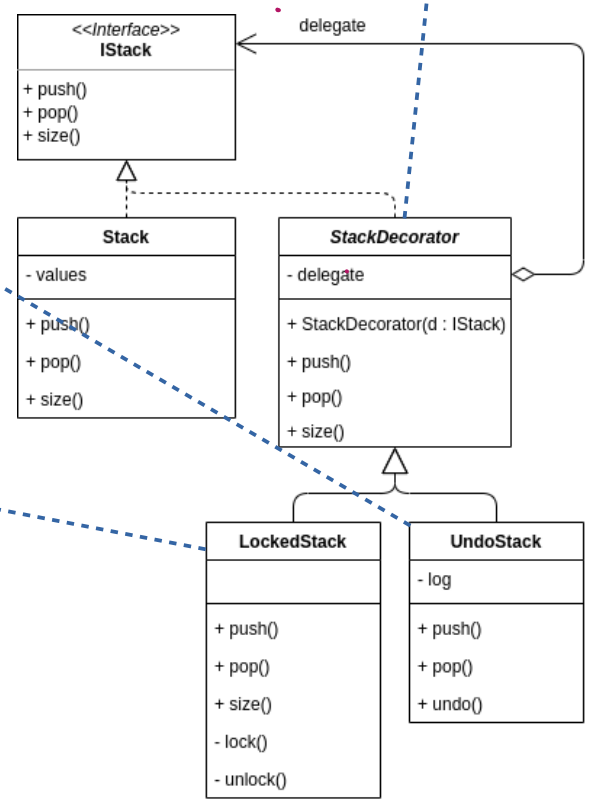
Concrete Decorator

```java
public abstract class StackDecorator implements IStack {

    protected IStack delegate;

    public StackDecorator(IStack delegate) {
        this.delegate = delegate;
    }
}
```

Decorator

```java
public class LockedStack extends StackDecorator {

    public LockedStack(IStack delegate) {
        super(delegate);
    }

    @Override
    public void push(Object o) {
        lock();
        delegate.push(o);
        unlock();
    }

    private void lock() {
        System.out.println("lock");
    }

    private void unlock() {
        System.out.println("unlock");
    }

}
```

Concrete Decorator



Vereinfachter Code: nicht alle Methoden vorhanden

Decorator: Beispiel IV

```java
public class UndoStack extends StackDecorator {

    private List<String> log = new ArrayList<>();

    public UndoStack(IStack delegate) {
        super(delegate);
    }

    @Override
    public void push(Object o) {
        remember("push");
        delegate.push(o);
    }

    private void remember(String method) {
        System.out.println("remember " + method);
        log.add(method);
    }
}
```

Concrete Decorator

```java
public class LockedStack extends StackDecorator {

    public LockedStack(IStack delegate) {
        super(delegate);
    }

    @Override
    public void push(Object o) {
        lock();
        delegate.push(o);
        unlock();
    }

    private void lock() {
        System.out.println("lock");
    }

    private void unlock() {
        System.out.println("unlock");
    }

}
```

Concrete Decorator

```java
public class Stack implements IStack {

    private List<Object> values = new ArrayList<Object>();

    @Override
    public void push(Object o) {
        System.out.println("push");
        values.add(o);
    }

}
```

Concrete Component

```java
public class Client {

    public static void main(String[] args) {
        System.out.println("== Basic Stack");
        IStack basicStack = new Stack();
        basicStack.push(new Object());

        System.out.println("== Locked Stack");
        IStack lockedStack = new LockedStack(new Stack());
        lockedStack.push(new Object());

        System.out.println("== Undo Locked Stack");
        IStack s = new UndoStack(new LockedStack(new Stack()));
        s.push(new Object());

    }
}
```

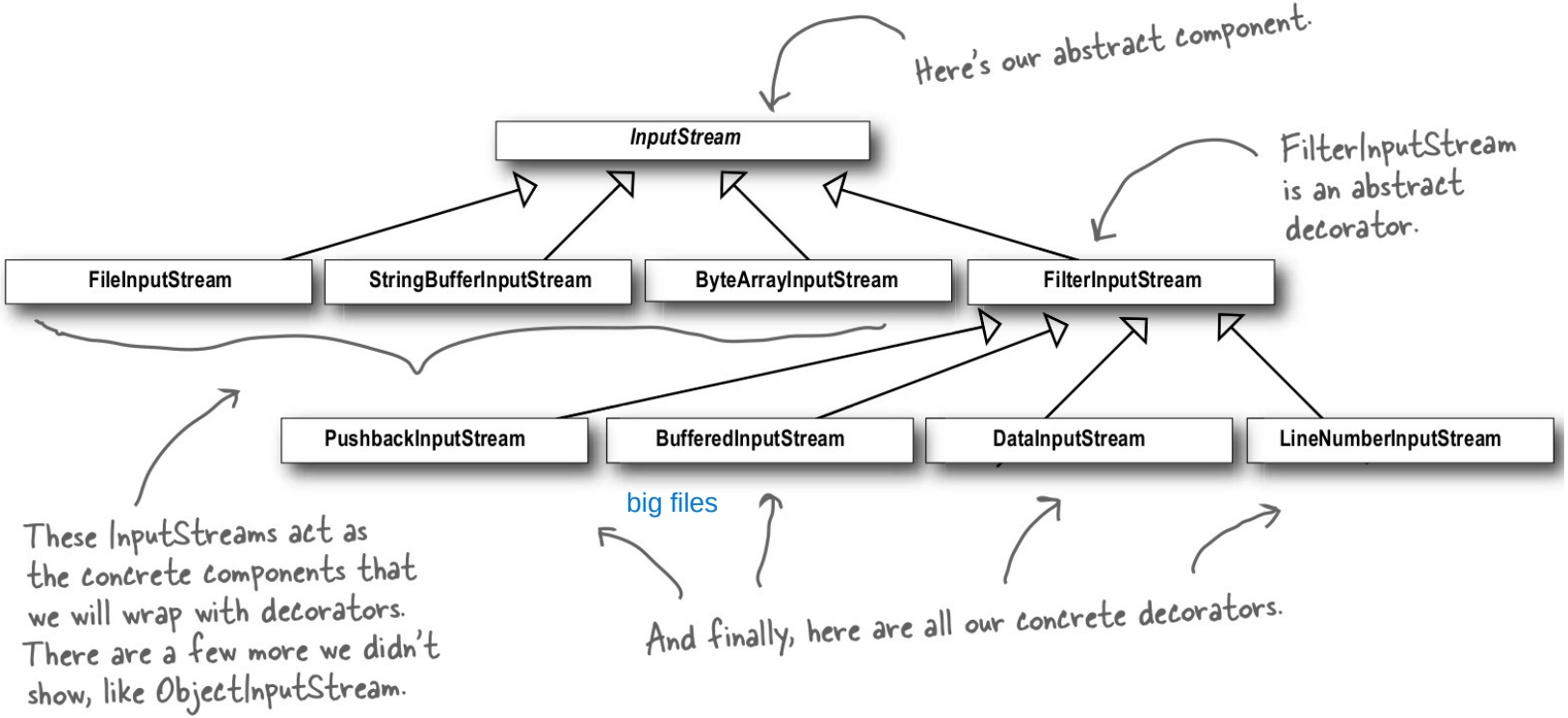
Client

```
== Basic Stack
push
== Locked Stack
lock
push
unlock
== Undo Locked Stack
remember push
lock
push
unlock
```

the order is importnat
if oe waits too long the
operation may not reach

Decorator: Reales Beispiel

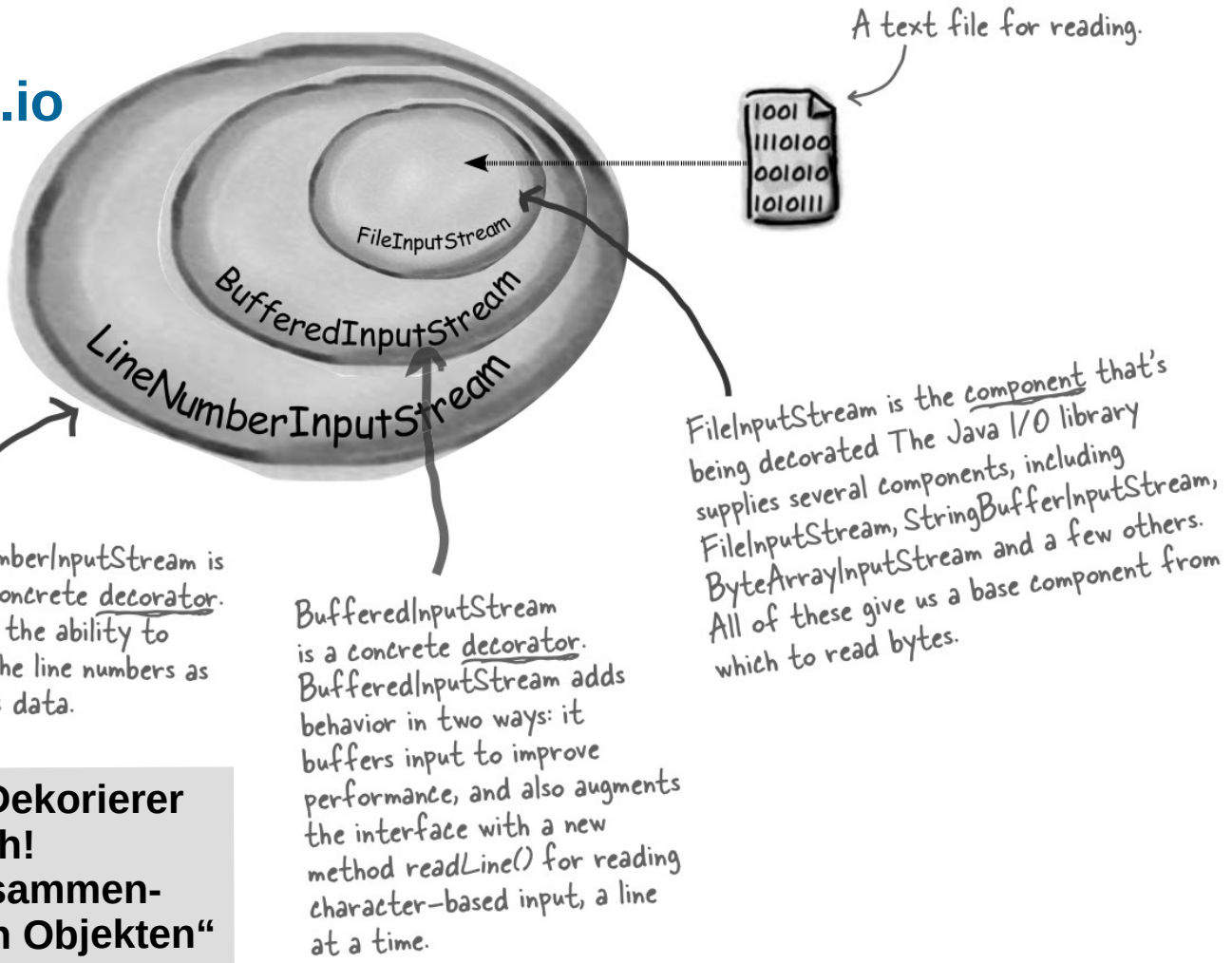
- Beispiel: **java.io**



Allgemein: Es sind mehrere konkrete Komponenten möglich!

Decorator: Reales Beispiel II

- Beispiel: **java.io**



Allgemein: Mehrere Dekorierer für ein Objekt möglich!
=> „ein Objekt ist zusammen-gesetzt aus mehreren Objekten“

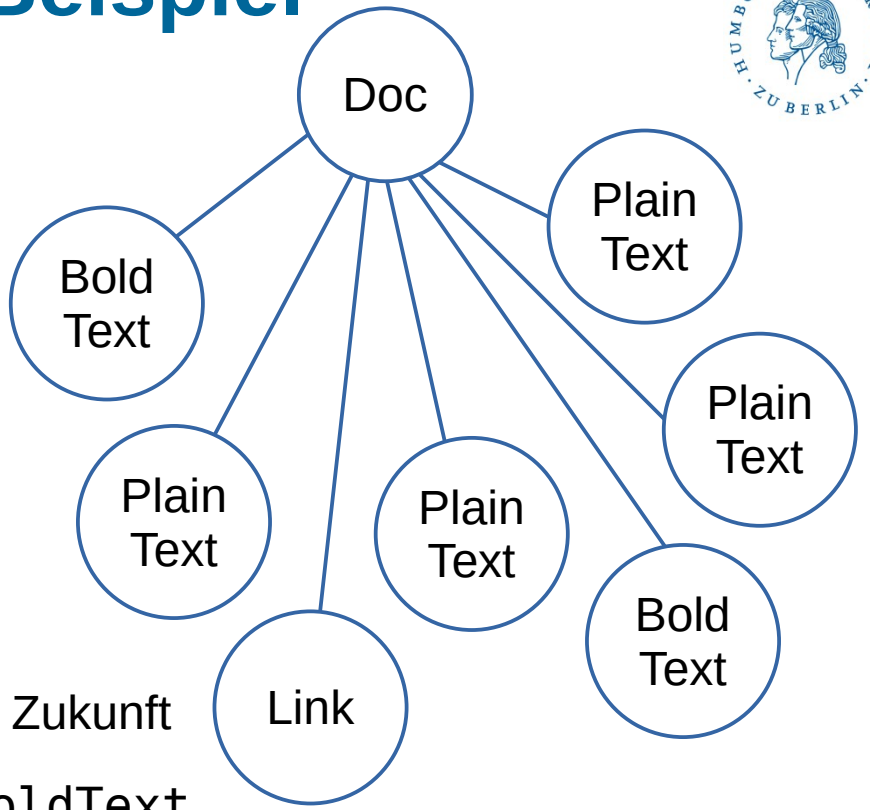
In Java:

```
InputStream is = new FileInputStream(“test.txt”);
InputStream bis = new BufferedInputStream(is);
InputStream lnis = new
LineNumberInputStream(bis);
```

Visitor: Übersicht

Beschreibung	Inhalt
Pattern-Name + Klassifikation	Visitor – Verhaltensmuster
Zweck	Trennung von Algorithmus (Operationen) 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.
Konsequenzen	Einfach neue Operationen hinzufügen. Gruppiert verwandte Operationen in einem Visitor. Neue Elemente hinzufügen ist schwierig (Anpassung aller Visitor). Visitor kann Zustand speichern. Elemente müssen ein Interface bereitstellen / implementieren

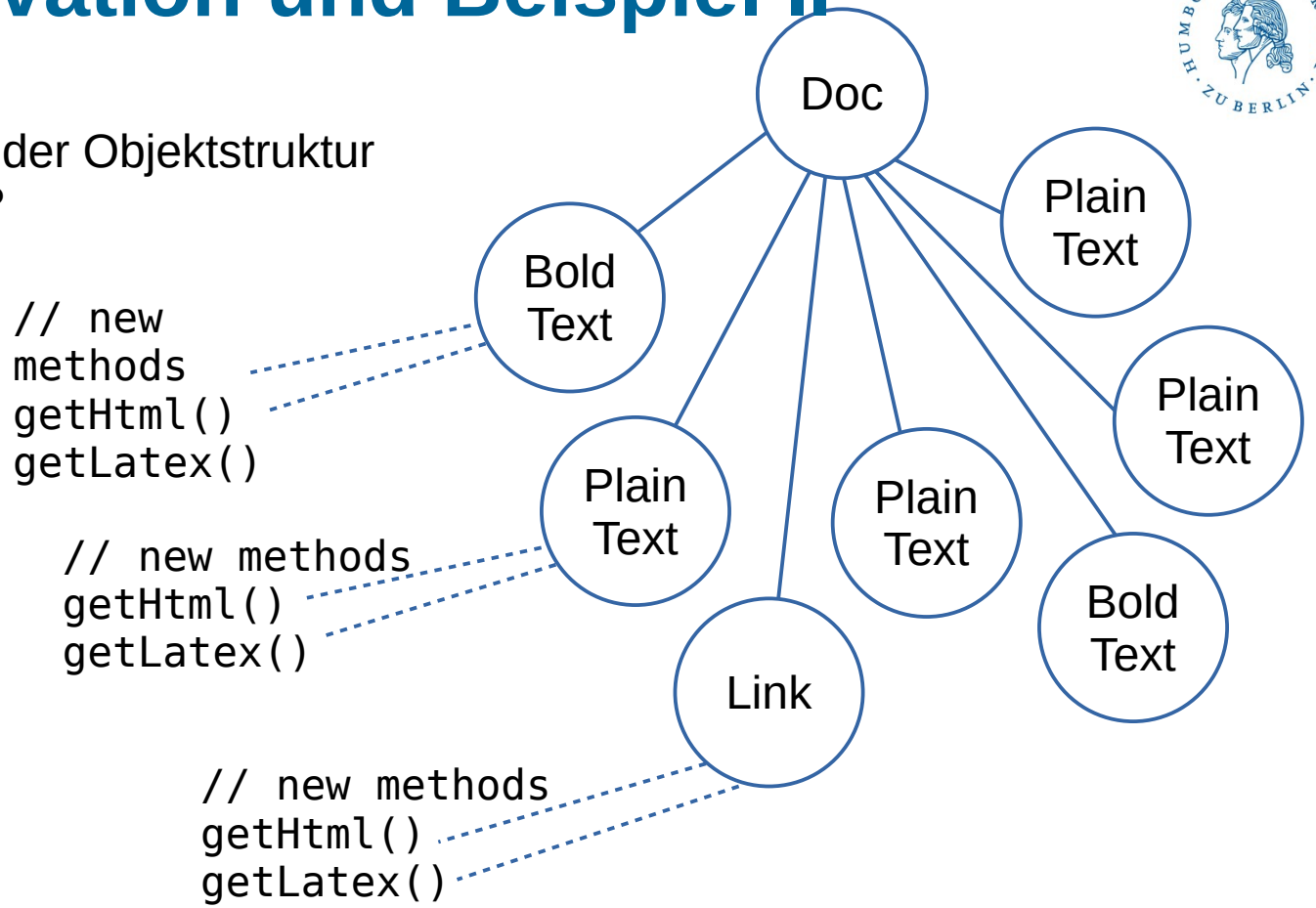
Visitor: Motivation und Beispiel



- Objektstruktur, die ein (vereinfachtes) Dokument beschreibt, das plain text, bold text und links enthält.
- Ein derartiges Dokument soll auf verschiedene Weise verarbeitet werden (Funktionalität)
  - Übersetzung nach HTML
  - Übersetzung nach LaTeX
  - …und potentiell weitere Funktionen in Zukunft
- Jede einzelne Klasse (wie `PlainText`, `BoldText` und `Link`) muss unterschiedlich verarbeitet werden
  - z.B. für HTML: `<b>bold text</b>` und `<a href=”url”>link</a>`
  - z.B. für LaTeX: `\textbf{bold text}` und `\href{url}{link}`

Visitor: Motivation und Beispiel II

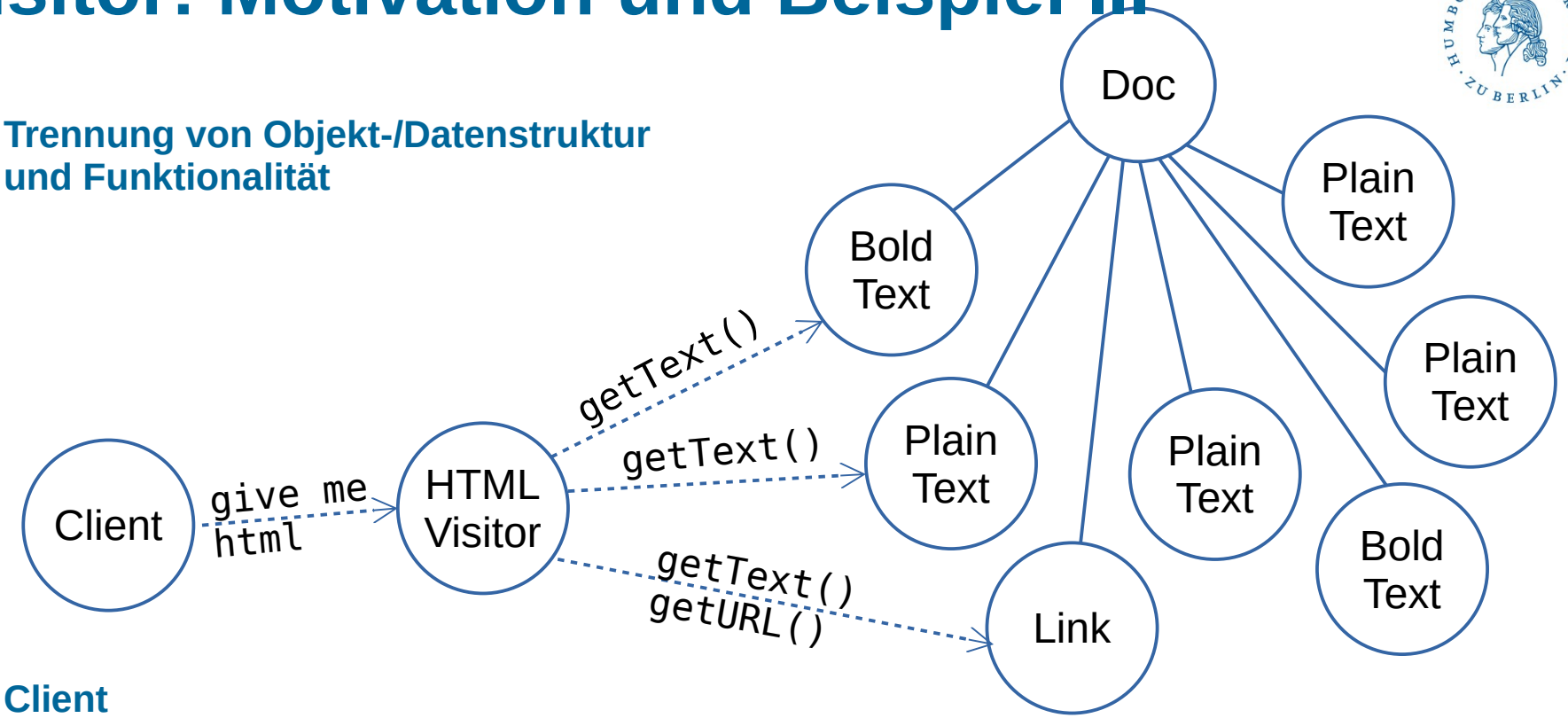
- Implementierung in der Objektstruktur bzw. Datenstruktur?



- Jede (neue) Funktionalität wird in drei Klassen realisiert
  – Funktionalität ist verteilt auf diese Klassen
- Erweiterung der Dokumentenstruktur um ein neues Element (z.B. kursiver Text)
  – Funktionalität wird in vier Klassen realisiert
- **Lösung: Trennung von Objekt-/Datenstruktur und Funktionalität**

Visitor: Motivation und Beispiel III

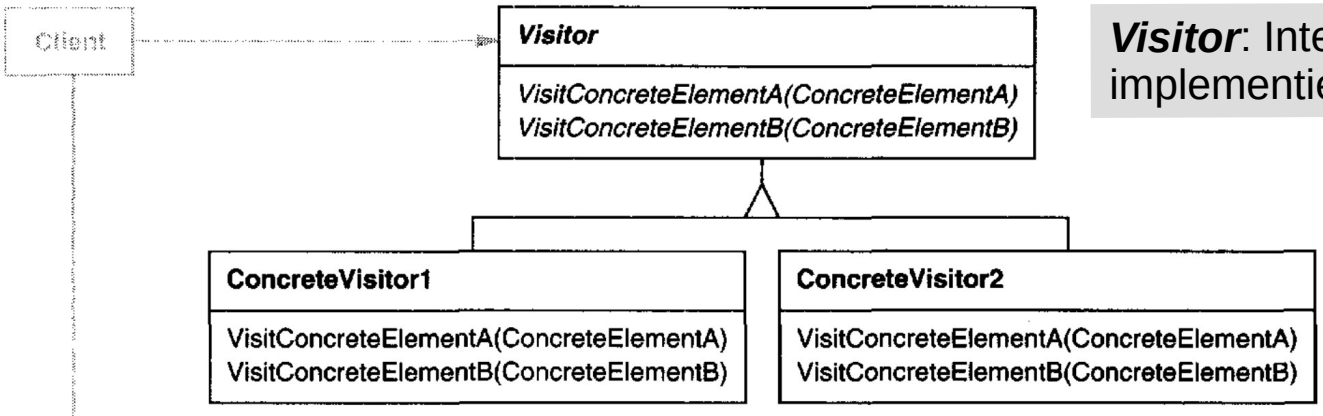
- **Trennung von Objekt-/Datenstruktur und Funktionalität**



- **Client**
  - Nutzt Visitor, um die gewünschte Funktionalität auf der Objektstruktur auszuführen
  - Leitet den Visitor durch die Objektstruktur
- **Visitor**
  - Holt sich Informationen von jedem Objekt der Struktur und führt die gewünschte Funktionalität aus
  - Gruppiert verwandte Operationen (z.B. Konvertierung nach HTML)
- Hinzufügen von neuem Verhalten, ohne die Objektstruktur anpassen zu müssen
  - Neues Verhalten in vorhandenem Visitor
  - Neuer Visitor (z.B. LaTeX-Visitor für Konvertierung nach LaTeX)

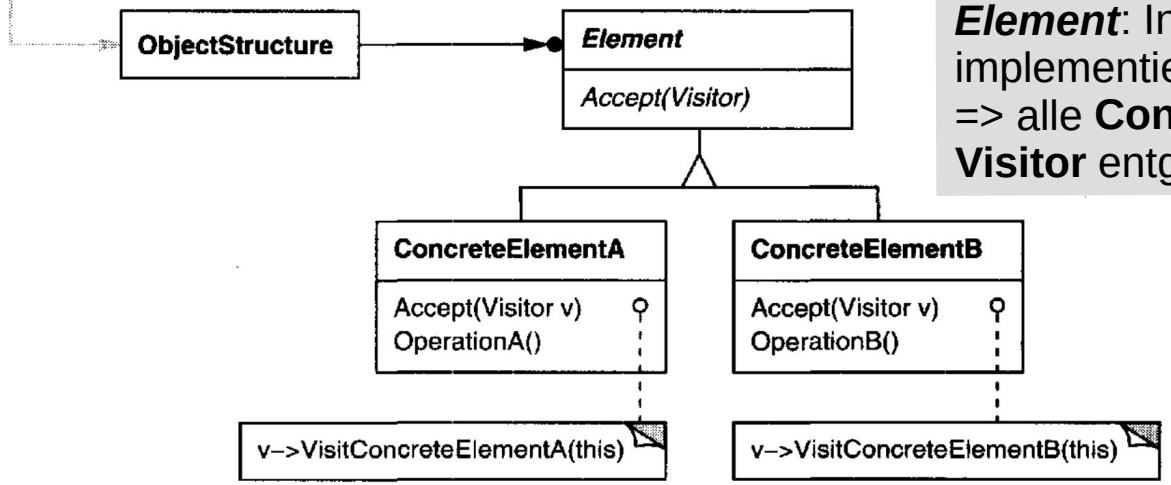
Visitor: Struktur und Rollen

Client nutzt **Visitor**, um gewünschte Funktionalität auf der **Object-Structure**, d.h. auf allen **Elementen** der Struktur auszuführen



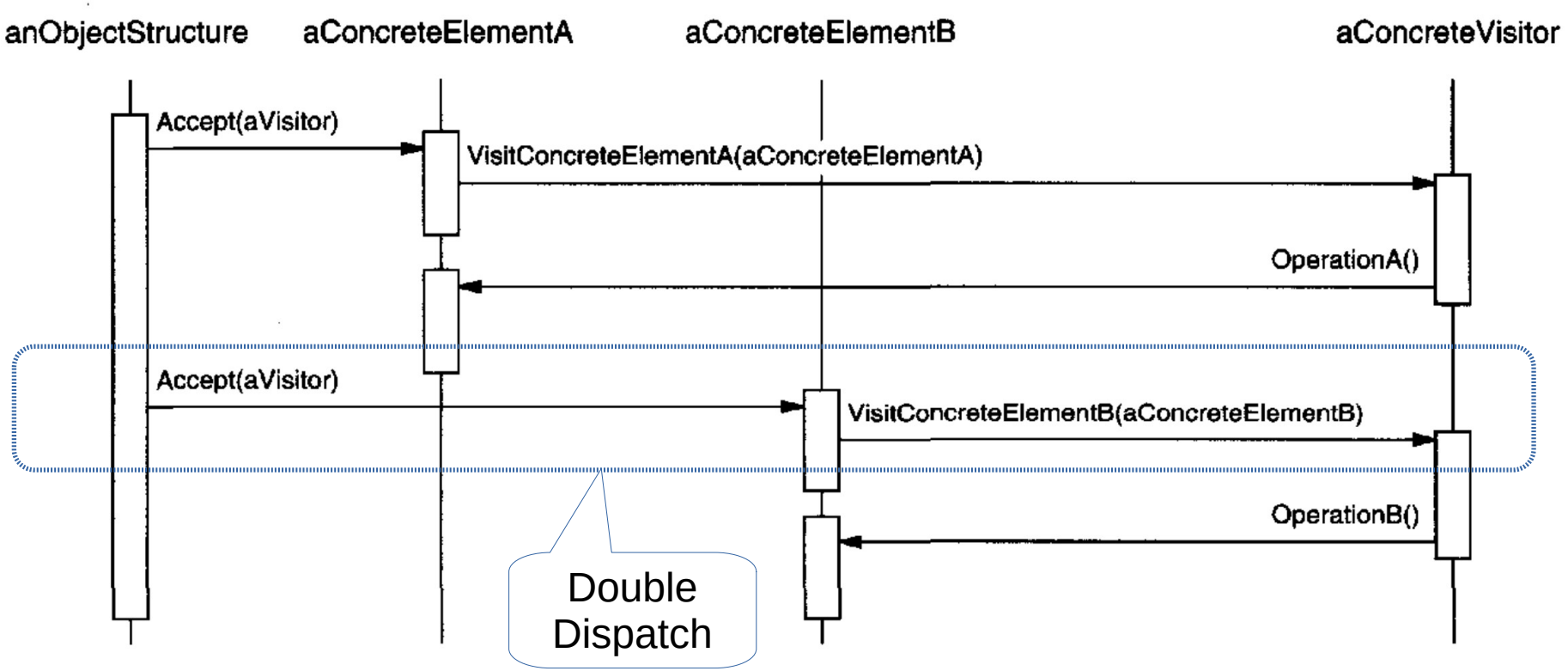
Visitor: Interface oder abstrakte Klasse, implementiert von allen **ConcreteVisitors**

ConcreteVisitor kann Zustand haben und diesen während des Besuchs mehrerer Elemente anreichern



Element: Interface oder abstrakte Klasse, implementiert von allen **ConcreteElements** => alle **ConcreteElements** können einen **Visitor** entgegennehmen

Visitor: Kollaboration



"Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of two receivers. Accept is a double-dispatch operation. Its meaning depends on two types: the Visitor's and the Element's.

Visitor: Beispiel

Objektstruktur

Element

```java
public abstract class Element {

    public abstract String accept(Visitor visitor);

}
```

```java
public class Document extends Element {

    private String title;
    private List<DocumentPart> parts = new ArrayList<>();

    public Document(String title) {
        this.title = title;
    }

    public String getTitle() {
        return this.title;
    }

    public void addPart(DocumentPart part) {
        this.parts.add(part);
    }

    public List<DocumentPart> getParts() {
        return this.parts;
    }

    @Override
    public String accept(Visitor visitor) {
        String result = visitor.visit(this);
        for (DocumentPart part : this.parts) {
            result += part.accept(visitor);
        }
        return result;
    }

}
```

```java
public abstract class DocumentPart extends Element {

    private String text;

    public DocumentPart(String text) {
        this.text = text;
    }

    public String getText() {
        return this.text;
    }

}
```

Concrete Element

Visitor: Beispiel II

Objektstruktur

Alle Elemente der Objektstruktur nehmen einen Visitor entgegen: `accept(Visitor visitor)`

Element

```
public abstract class Element {

    public abstract String accept(Visitor visitor);

}
```

```
public class PlainText extends DocumentPart {

    public PlainText(String text) {
        super(text);
    }

    @Override
    public String accept(Visitor visitor) {
        return visitor.visit(this);
    }

}
```

Concrete Element

```
public abstract class DocumentPart extends Element {

    private String text;

    public DocumentPart(String text) {
        this.text = text;
    }

    publ

    }
```

```
public class Link extends DocumentPart {

    private String url;

    public Link(String text, String url) {
        super(text);
        this.url = url;
    }

    public String getURL() {
        return this.url;
    }

    @Override
    public String accept(Visitor visitor) {
        return visitor.visit(this);
    }

}
```

Concrete Element

```
public class BoldText extends DocumentPart {

    public BoldText(String text) {
        super(text);
    }

    @Override
    public String accept(Visitor visitor) {
        return visitor.visit(this);
    }

}
```

Concrete Element

Visitor: Beispiel III

Visitor

- `visit(…)` kann auch void sein; Ergebnisse im Visitor ansammeln
- Häufig anderes Namensschema:
  `visitDocument(Document document)`
  `visitPlainText(PlainText plainText)`
  `visitBoldText(BoldText boldText)`
  `visitLink(Link link)`

Visitor

```
public interface Visitor {

    public String visit(Document document);

    public String visit(PlainText plainText);

    public String visit(BoldText boldText);

    public String visit(Link link);

}
```

```
public class HTMLVisitor implements Visitor {

    @Override
    public String visit(Document document) {
        return "<h1>" + document.getTitle() + "</h1>\n";
    }

    @Override
    public String visit(PlainText plainText) {
        return plainText.getText() + " ";
    }

    @Override
    public String visit(BoldText boldText) {
        return "<b>" + boldText.getText() + "</b> ";
    }

    @Override
    public String visit(Link link) {
        return "<a href=\"" + link.getURL() +
                "\">" + link.getText() + "</a> ";
    }

}
```

Concrete Visitor

```
public class LatexVisitor implements Visitor {

    @Override
    public String visit(Document document) {
        return "\\title{" + document.getTitle() + "}\n";
    }

    @Override
    public String visit(PlainText plainText) {
        return plainText.getText() + " ";
    }

    @Override
    public String visit(BoldText boldText) {
        return "\\textbf{" + boldText.getText() + "} ";
    }

    @Override
    public String visit(Link link) {
        return "\\href{" + link.getURL() + "}{"
                                + link.getText() + "} ";
    }

}
```

Concrete Visitor

Visitor: Beispiel IV

Client

```
public class Client {

    public static void main(String[] args) {

        Document doc = new Document("Document Title");

        doc.addPart(new PlainText("This is just plain text."));
        doc.addPart(new BoldText("Some bold text."));
        doc.addPart(new Link("Search here!", "http://www.google.com"));
        doc.addPart(new PlainText("This is just more text."));

        System.out.println("==HTML");
        Visitor htmlVisitor = new HTMLVisitor();
        String html = doc.accept(htmlVisitor);
        System.out.println(html);

        System.out.println("\n==LaTeX");
        Visitor latexVisitor = new LatexVisitor();
        String latex = doc.accept(latexVisitor);
        System.out.println(latex);
    }

}
```

```
==HTML
<h1>Document Title</h1>
This is just plain text. <b>Some bold text.</b> <a href="http://www.google.com">Search here!</a> This is just more text.

==LaTeX
\title{Document Title}
This is just plain text. \textbf{Some bold text.} \href{http://www.google.com}{Search here!} This is just more text.
```

Literatur/Referenzen

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. 2004. Head First Design Patterns. O‘Reilly.