Exercise 06:
Design Patterns

Based on the material by Oscar Nierstrasz, Sven Apel, Janet Siegmund, Norbert Siegmund, Timo Kehrer, Ina Schaefer

Organizational

- Programming Task

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

Abgabe Blatt 06

Aufgabe | Einstellungen | Erweiterte Bewertung | Mehr

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.

Observe instructionsand use the provided source code!

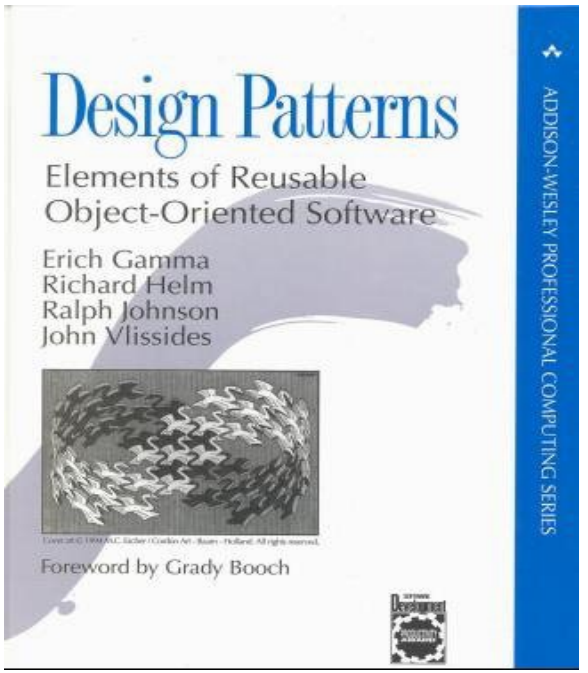
Design Patterns

"Designing object-oriented software is hard and designing reusable object-oriented software is

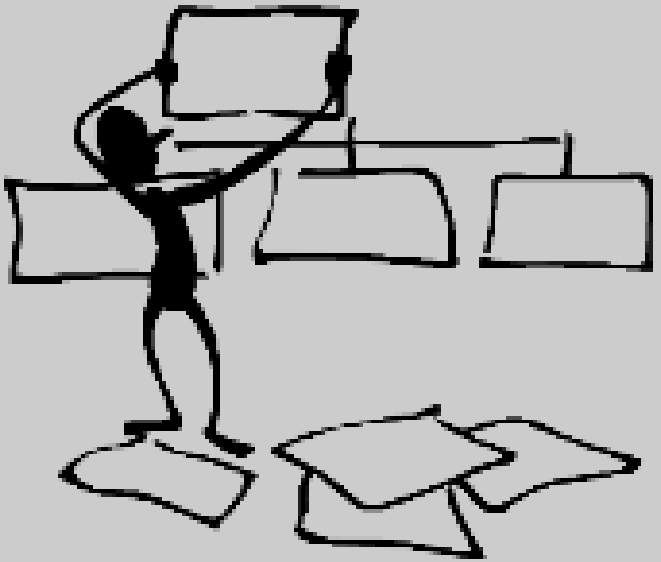
even harder."—Erich Gamma et al.

- A design pattern consists of
  - one context or situation, in which a recurring design problem occurs, and
  - a generic and proven solution that can master the problem.
- **Object -oriented design pattern**
  - Follow object -oriented design goals

    Modularity, explicit interfaces, information hiding, ...
  - Improvement of Quality and Reusability of Software
  - Catalog of 23 Patterns of the "Gang of Four" (GoF)



Application of a Pattern

Template Application	
No mechanical "Pattern Matching"!	
Design patterns are abstract solutions for “recurring problems”!!	
More about the transfer of the idea of the pattern	
Pattern structure should be recognizable, possibly adjust the existing design or represent it differently	
Behavioral schema must also appear in the code similar to the pattern idea	

Classification

- 3 Categories
  - Creation Patterns (Creational Patterns)
    - Help with object creation
  - Structural Patterns (Structural Patterns)
    - Help with the composition of classes and objects
  - Behavioral Patterns
    - Help with the interaction of classes and objects and encapsulation of behavior
- Application Area
  - Class Pattern: Focus on the relationship between classes and their subclasses (reuse through inheritance)
  - Object Pattern: Focus on the relationship between objects (reuse through composition)

Description of a Pattern

Description	Explanation
Pattern-Name + Classification	Exact Name of the Pattern
Purpose	What the Pattern Does
Also Known As	Alternative Name of the Pattern
Motivation	Scenario where the pattern is meaningful
Applicability	Situations when the pattern can be applied
Structure	Graphical Representation (Class Diagram Style)
Participants (Roles)	Involved Classes and Objects as Roles
Collaboration	How do participants work together
Consequences	Pros and cons of the pattern
Implementation	Guidelines and Techniques for Implementation
Example Code	Code Fragments
Known Uses	Examples in Real Systems
Related Patterns	List and Description of Related Patterns

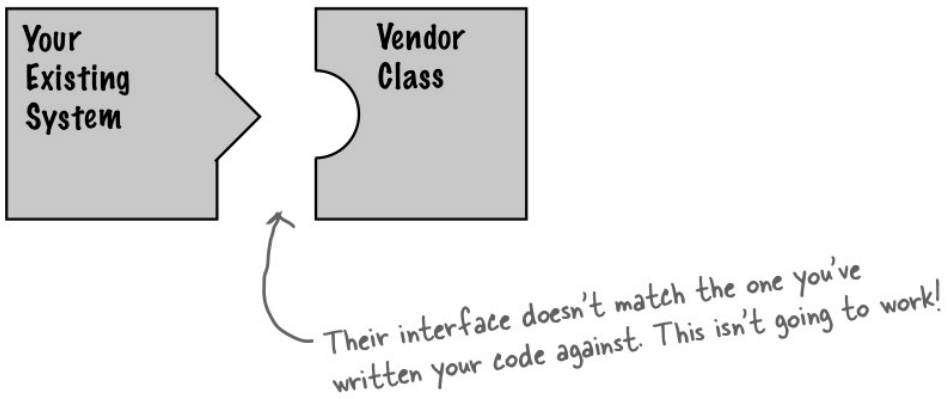
... in more detail than in the lecture, but we do not look at everything in detail.

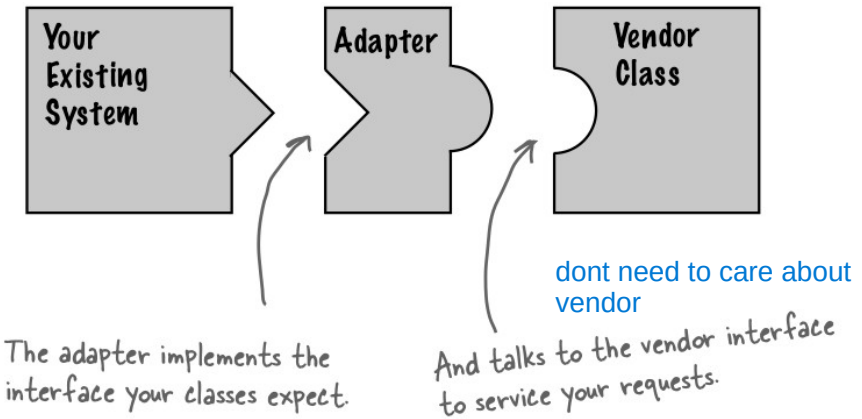
Important design pattern

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

Adapter: Overview

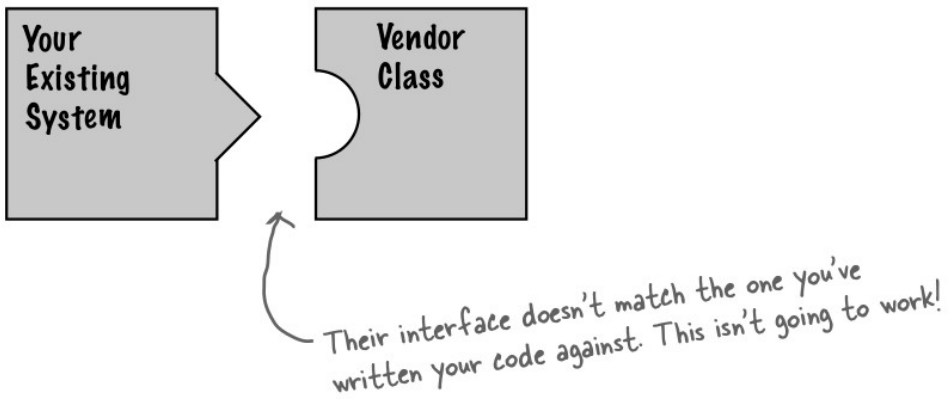
Description	Content
Pattern-Name + Classification	Adapter – Structural Pattern
Purpose	Converts the interface of an existing class to match the interface of a client (Client). Allows classes to interact with each other, which otherwise would not be possible due to differences in the interface.
Also Known As	Wrapper Pattern or Wrapper
Motivation	A class that already exists provides a needed functionality, but implements an interface that does not meet the expectations of a client.

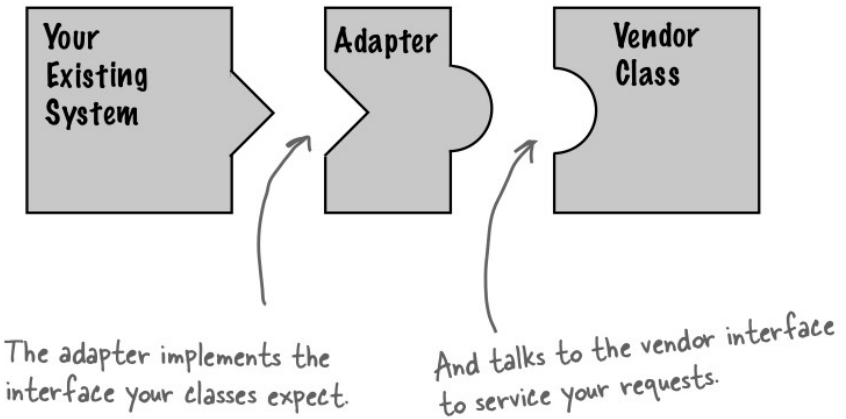




Adapter: Overview II

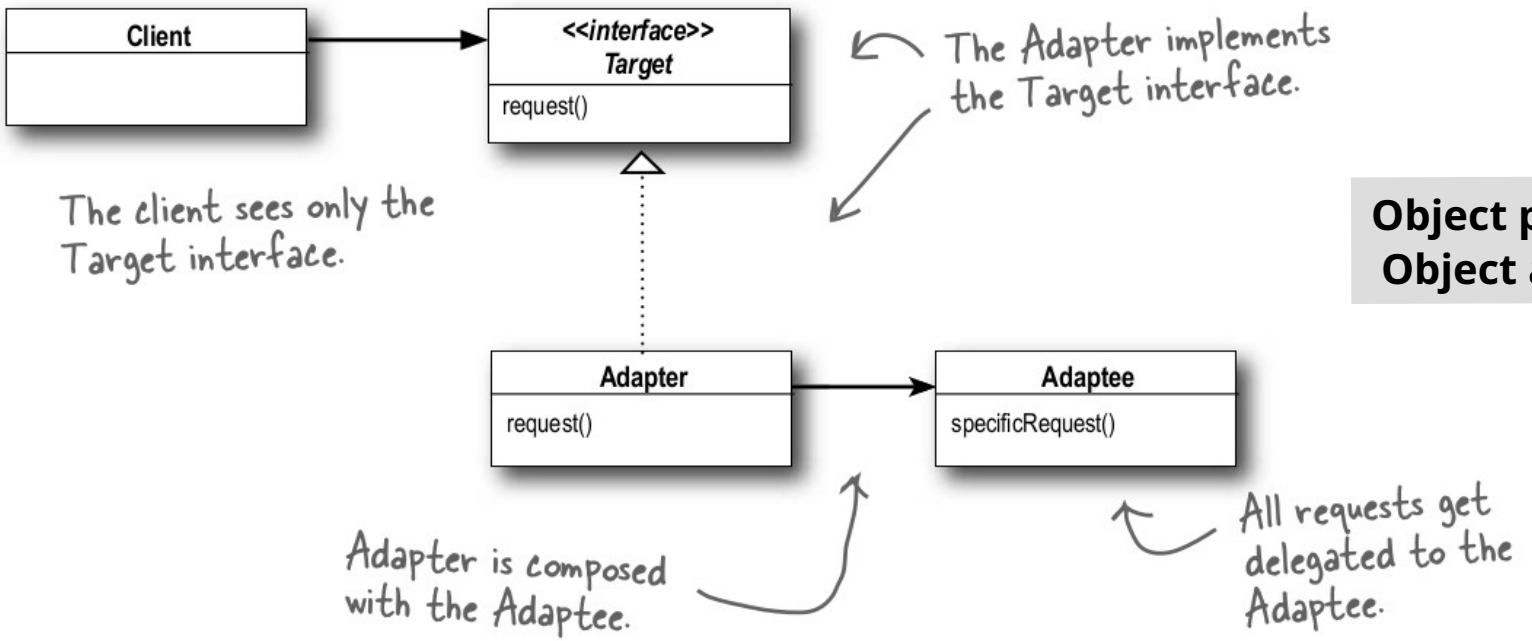
Description	Content
Applicability	- Use a class that is otherwise not reusable (due to interface incompatibility) again: Adapt the interface by changing the method signatures in the adapter. - Existing class does not provide the required functionality: Implement the required function in the adapter class by adding new methods that match the interface
Structure	See next slide
Participants/Roles	See next slide
collaboration	Clients call the adapter methods that forward the inquiries to the adapted class (service).





Adapter: structure and participants/roles

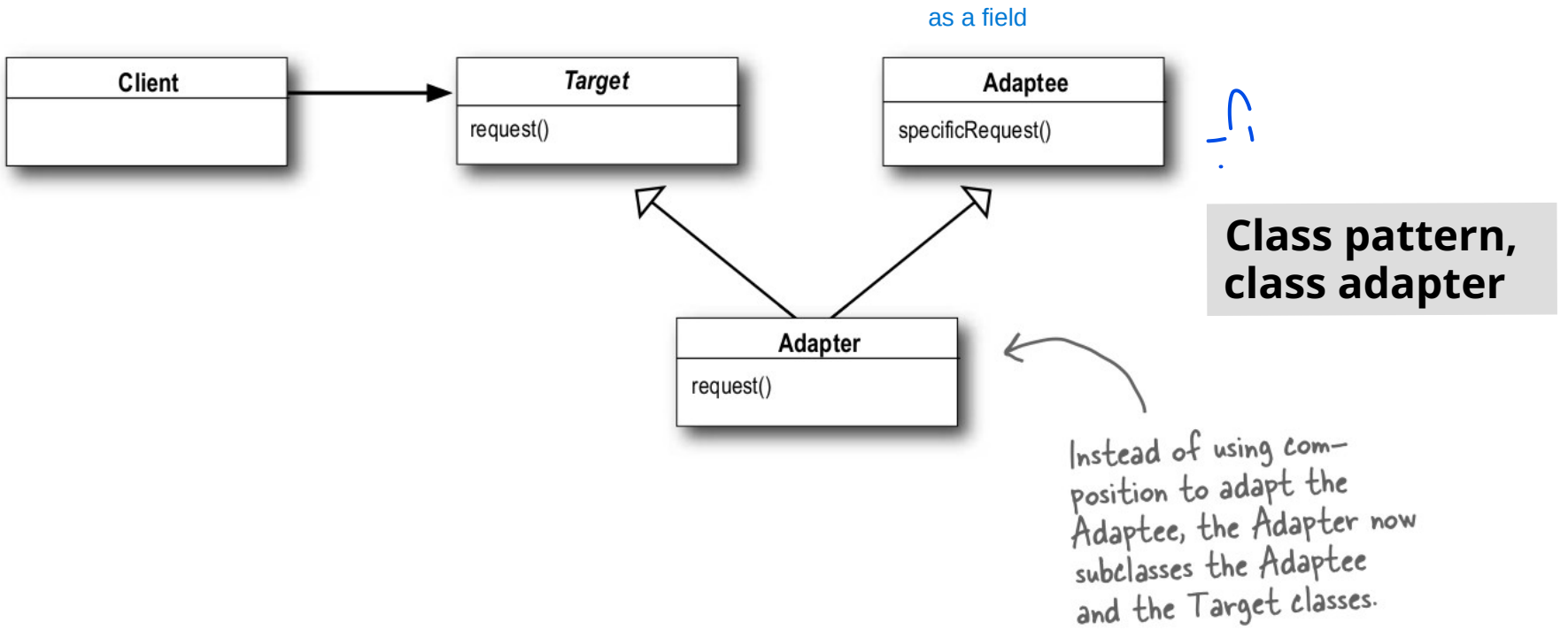
- **Rollen:**Client, Target, Adapter, Adaptee
- Adapter holds a reference to Adaptee (the class to be adapted)
- Adapter uses delegation to forward calls to Adaptee



Object pattern, Object adapter

Adapter: Structure and Participants/Roles II

- *Roles: Client, Target, Adapter, Adaptee*
- Adapter inherits from Target and Adaptee
- In the event of simple inheritance (Java): Adapter gives reference to the adaptea



Adapter: Overview III

Description	Content
Participant	Target: Defines the (domain-specific) interface used by the client. Client: Interacts with objects that implement the Target interface. Adaptee (class to be adapted): Represents an existing interface that is not compatible with the Target. **Adapter:**Adapts the interface from the Adaptee so that it is compatible with the Target.
Consequences	When using inheritance: The class Adapter can override the methods of the superclass (Adaptee) if inheritance is possible from the Adaptee Adaptability depends on the difference between the interfaces between Target and Adaptee
Implementation	See the next slide for an example
Example Code	See the next slide for an example
Known uses	Gui frameworks use existing class hierarchies, but must be adapted
Related patterns	Decorator: Reichert Object for functionality without changing the interface Bridge: Separates interface and implementation, allowing different implementations to be easily interchangeable

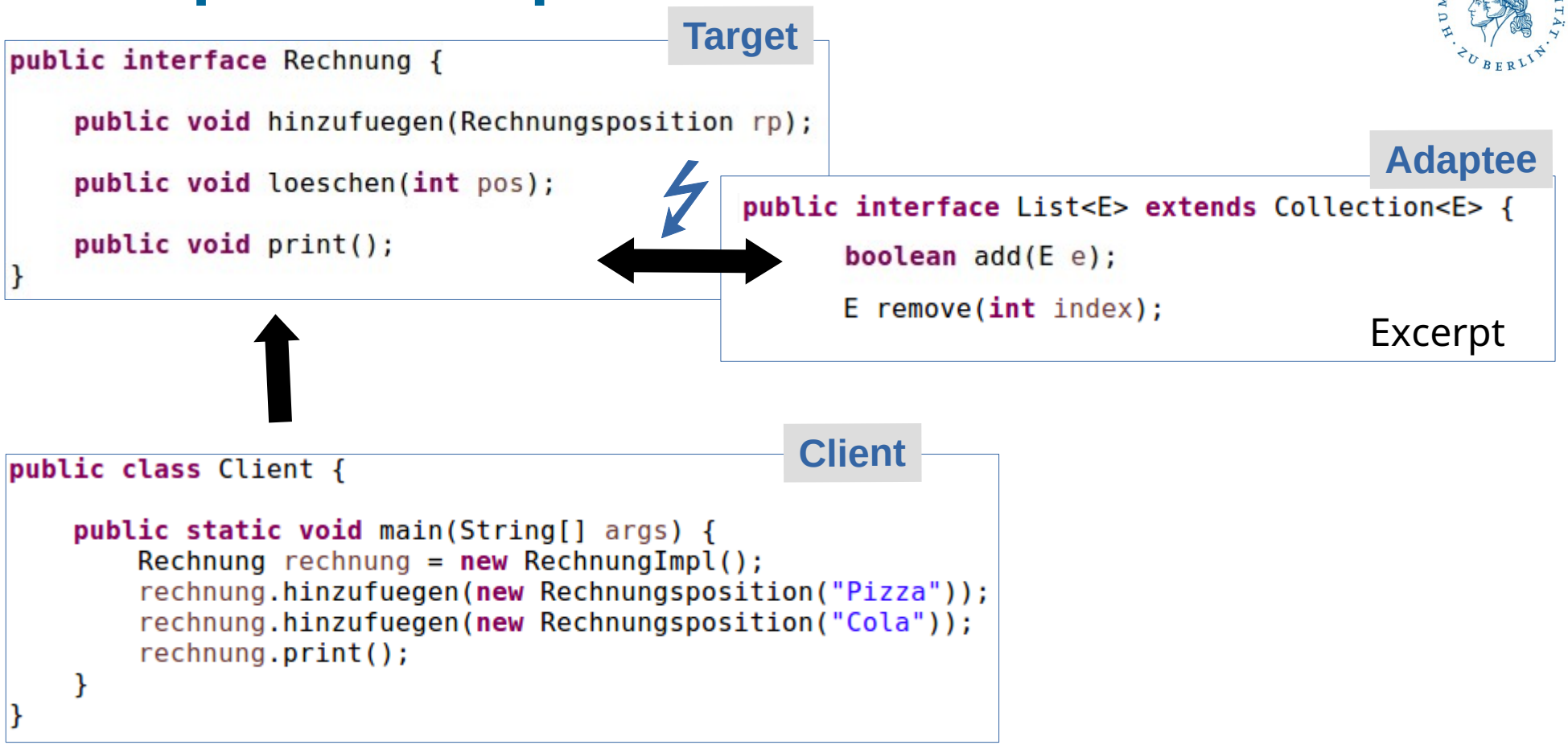
Adapter: Example

```
public interface Rechnung {

    public void hinzufuegen(Rechnungsposition rp);

    public void loeschen(int pos);

    public void print();
}
```

```
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;
    }

}
```

- A billing document includes a list of Billing positions
- Implementation Disinterface invoice
  - can implement your own list (complex)
  - rather use of a existing implementation of the list java.util.List<E>, e.g. ArrayList, LinkedList, etc.

Adapter: Example II



- Problem: Client uses the interface Rechnung, which differs from the interface java.util.List<E>

Adapter: Example 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);
```

Excerpt

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: Example IV

Class pattern, class adapter

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);
```

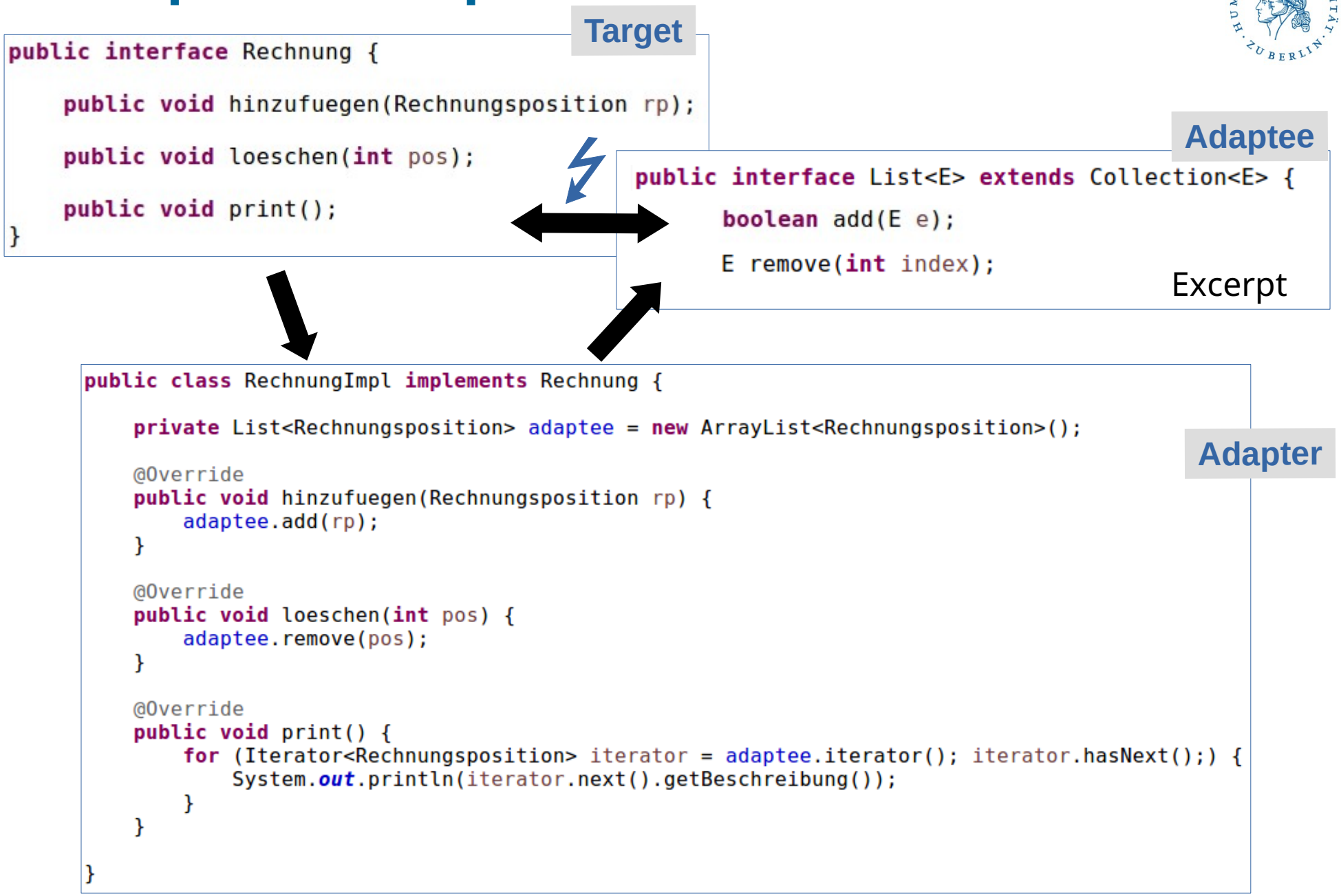
Excerpt

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: Example V



Adapter: Example VI

Object pattern,
Object adapter

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);
```

Excerpt

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());
        }
    }

}
```

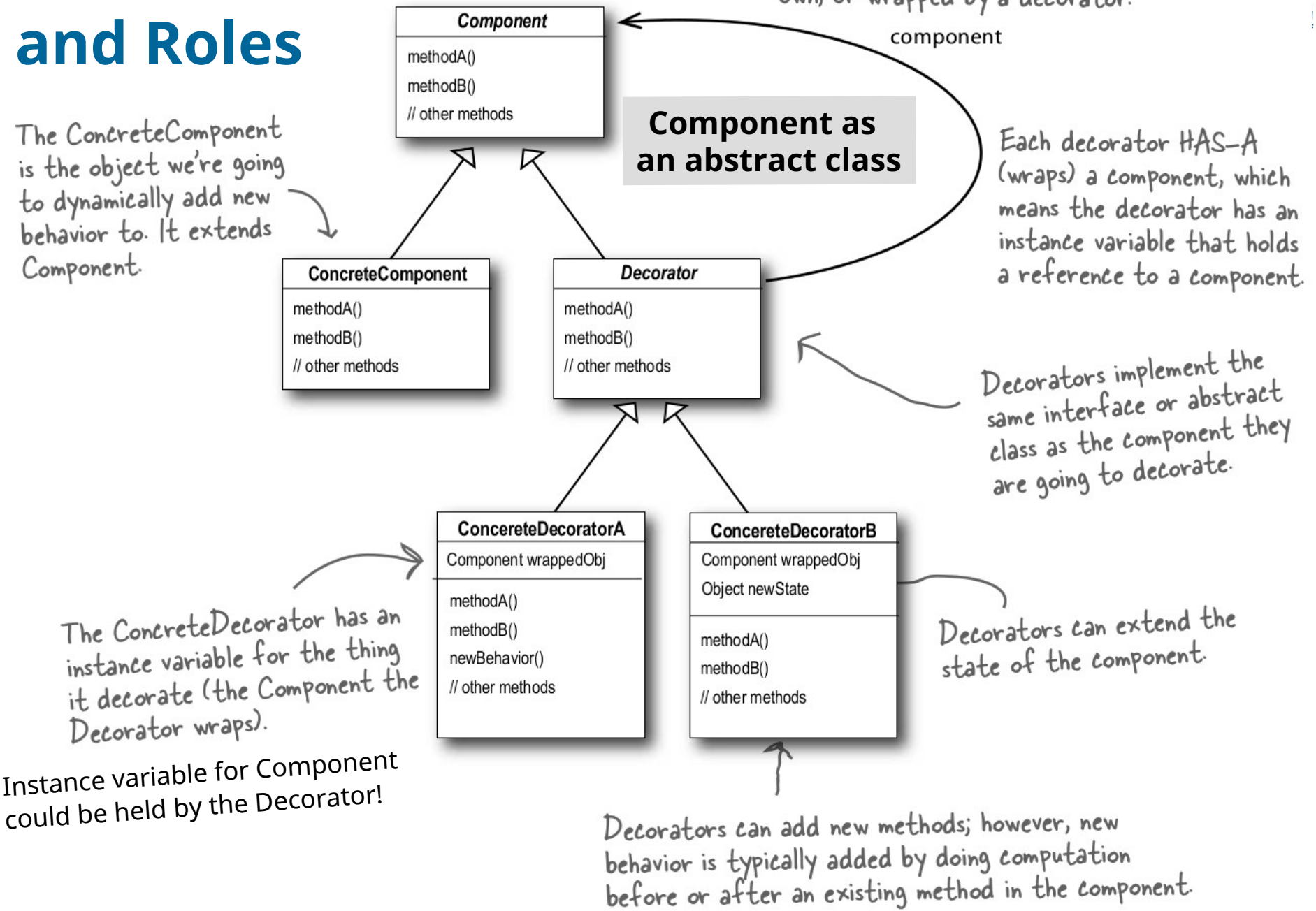
Decorator: Overview

Description	Content
Pattern-Name + Classification	Decorator - Structural Pattern
Purpose	Add functionality and features flexibly and dynamically to existing classes.
Motivation	We require flexible implementations of a class that can vary depending on the context.
Applicability	Extensions are optional. Applicable when extensions through inheritance are impractical, as inheritance relationships are predetermined for all objects (e.g., for a large number of independent extensions, a greater number of subclasses would be necessary to cover all possible combinations of extensions).
Consequences	More flexible than (static) inheritance. Problem of objects schizophrenia (an object is composed of several objects). Many small objects.

Decorator: overview II

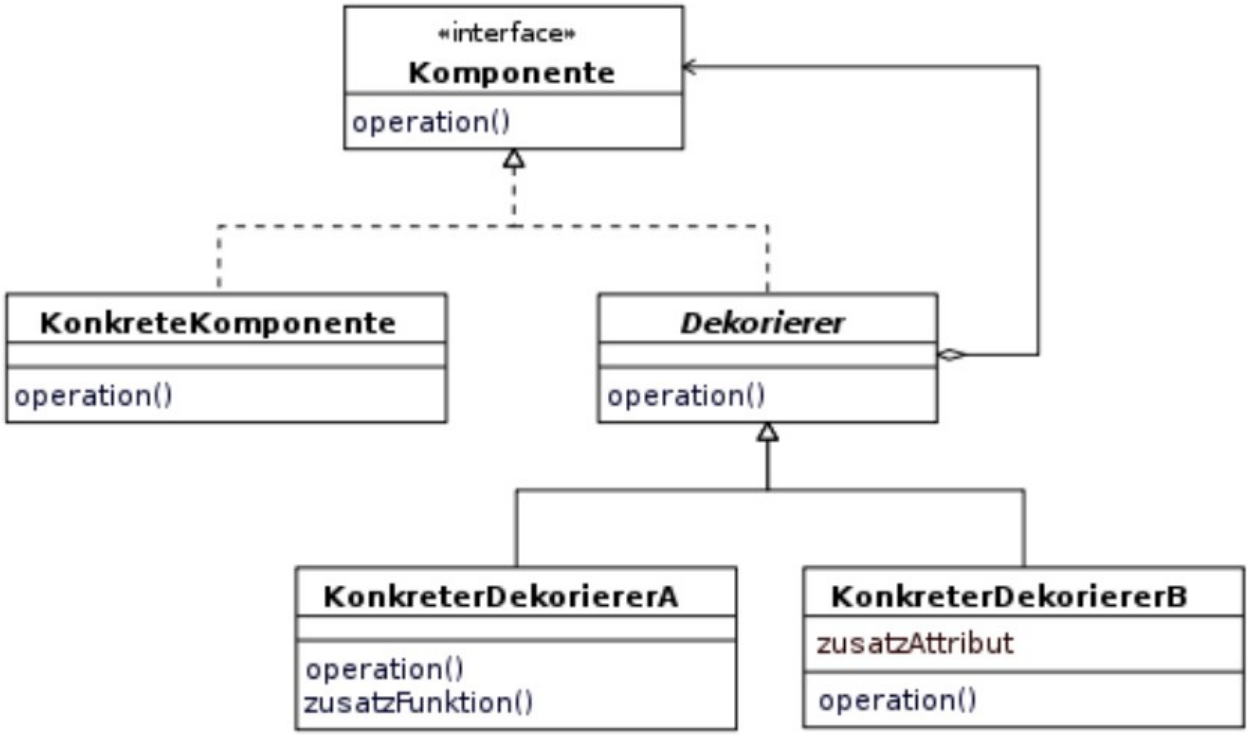
- Class to be decorated: class that is to be expanded to include functionality or properties
- Decorator: provides the expansion
- Decorator has the same interface as class to be decorated
- Decorator instance is placed before the instance of the class to be decorated
  - Calls are redirected or completely processed on their own
  - The functionality of the decorator can be executed before and/or after redirection
- Multiple decorators for an object are possible => an object is composed of multiple objects
- **The role:**
  - *Class to be decorated: Component, ConcreteComponent*
  - Dekorierer: *Decorator, ConcreteDecorator*

Decorator: Structure and Roles



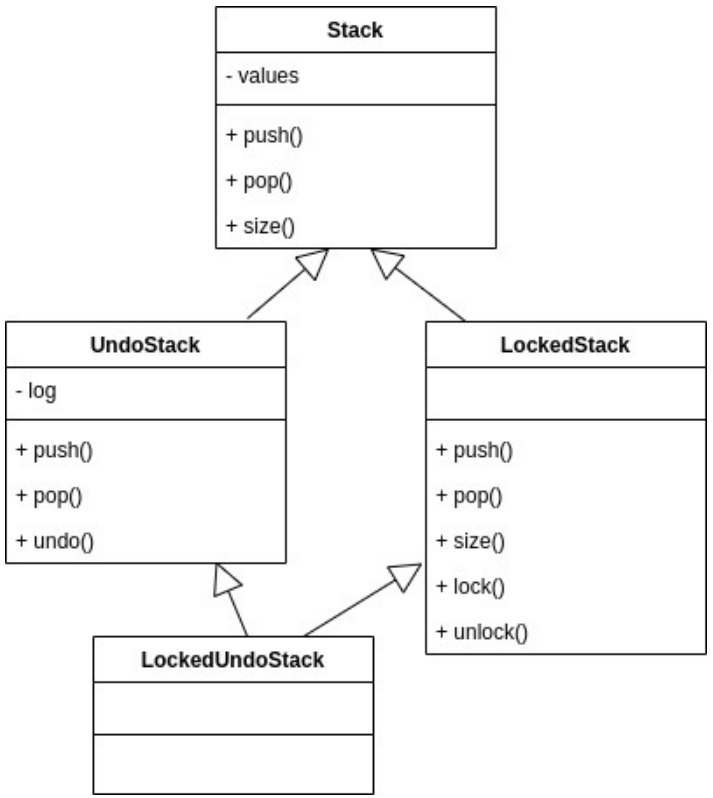
Decorator: Structure and Roles II

Component as Interface

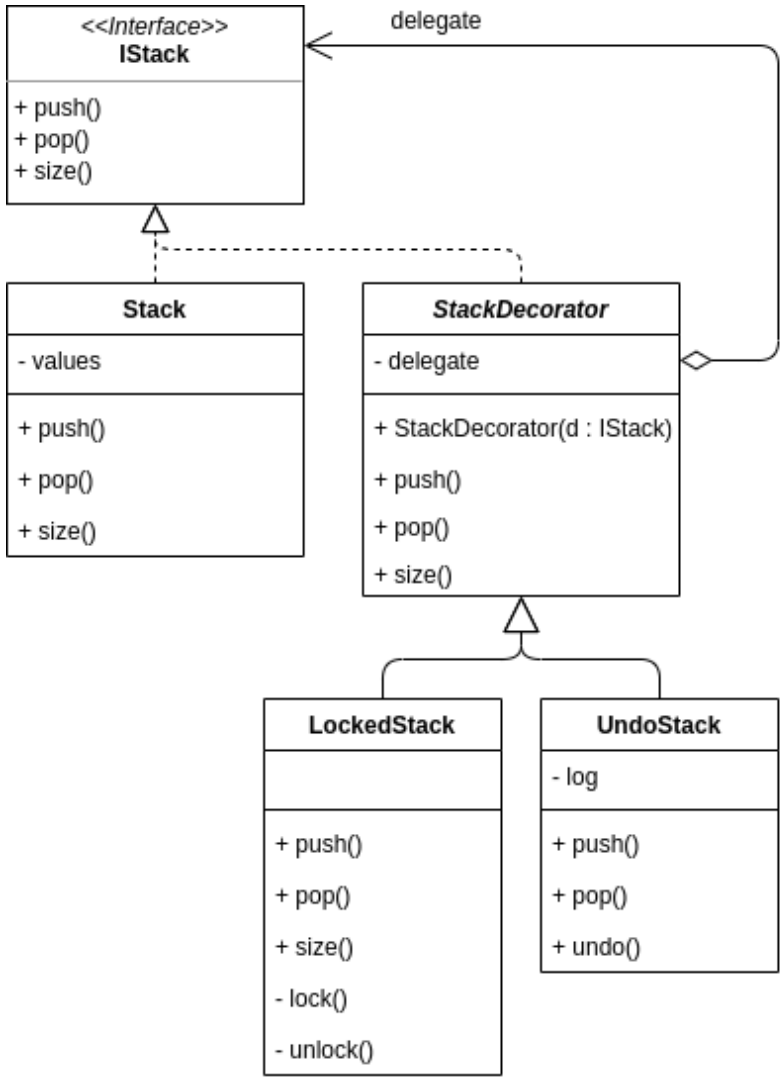


Decorator: Example

- Stack should have functionality for Undo and/or Locking
- **Diamond Problem**
  - `new LockedUndoStack().pop()`
  - Unclear which pop() method LockedUndoStack inherits from: UndoStack or LockedStack?



Solution: Decorator Pattern



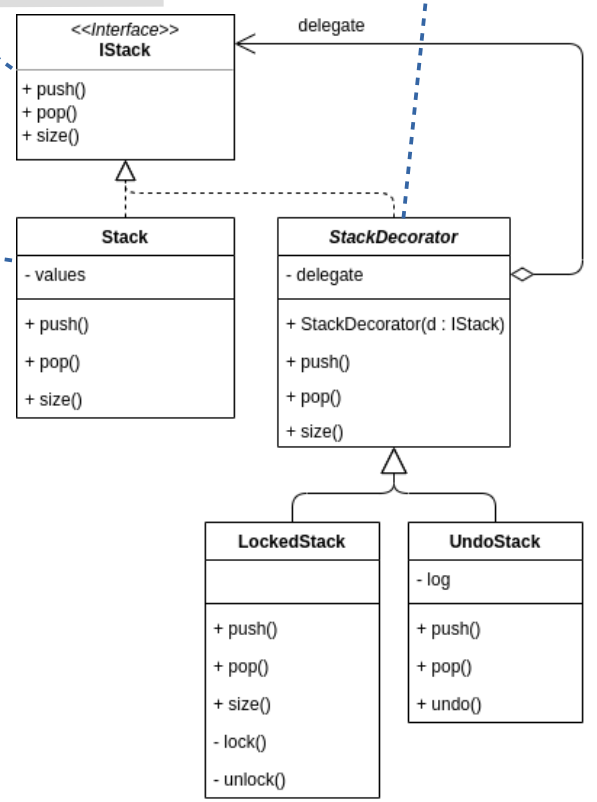
Decorator: Example II

```java
public abstract class StackDecorator implements IStack {

    protected IStack delegate;

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

Decorator

Component

```java
public interface IStack {

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




```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

Simple Code: Not all methods available

Decorator: Example 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);
    }
}
```

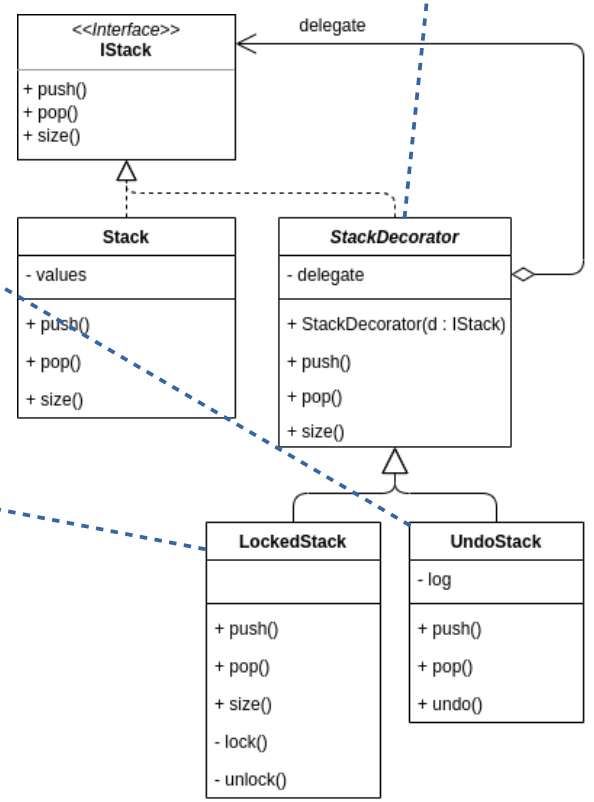
Concrete Decorator

```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



Simple Code: Not all methods available

Decorator: Example 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 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 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

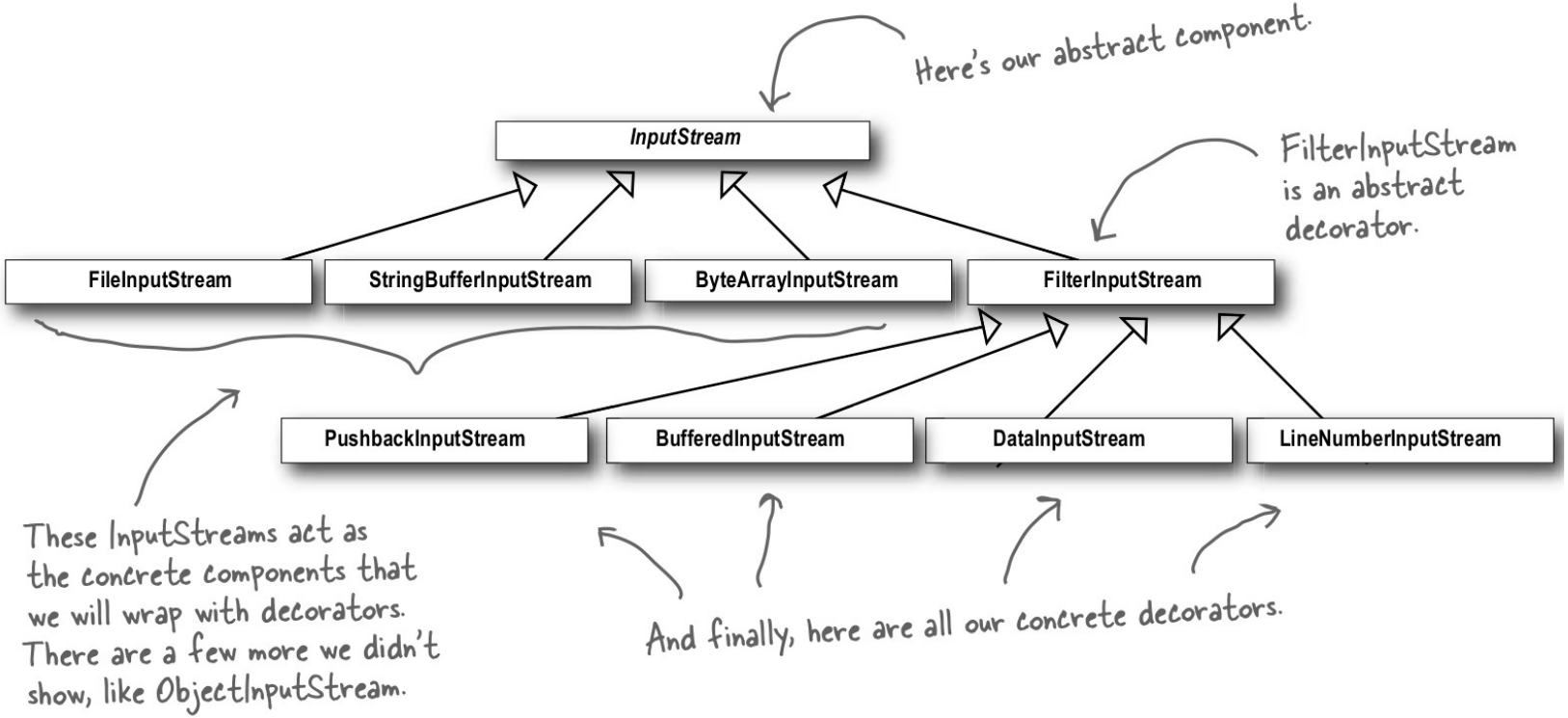
```java
public class Client {

    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
```

Decorator: Real Example

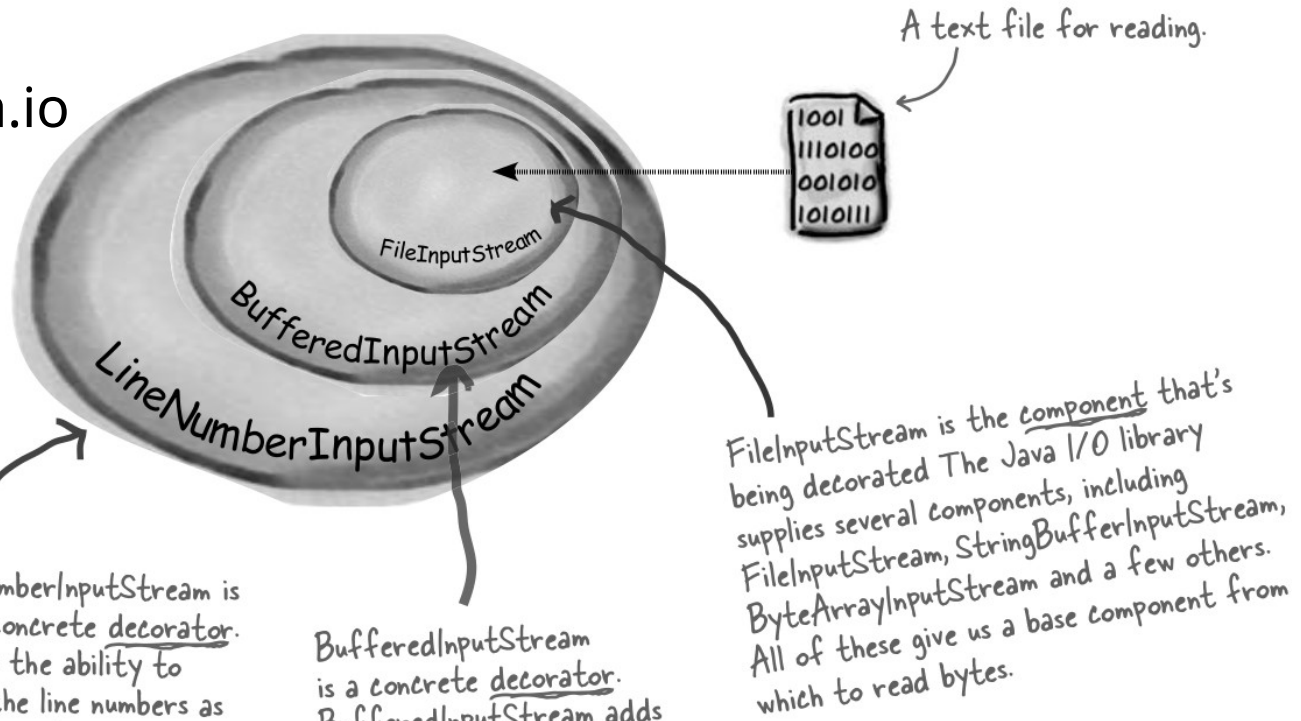
- Example: java.io



General: Several specific components are possible!

Decorator: Real example II

- Example: java.io



General: Multiple decorators possible for an object!
=> "An object is composed of multiple objects"

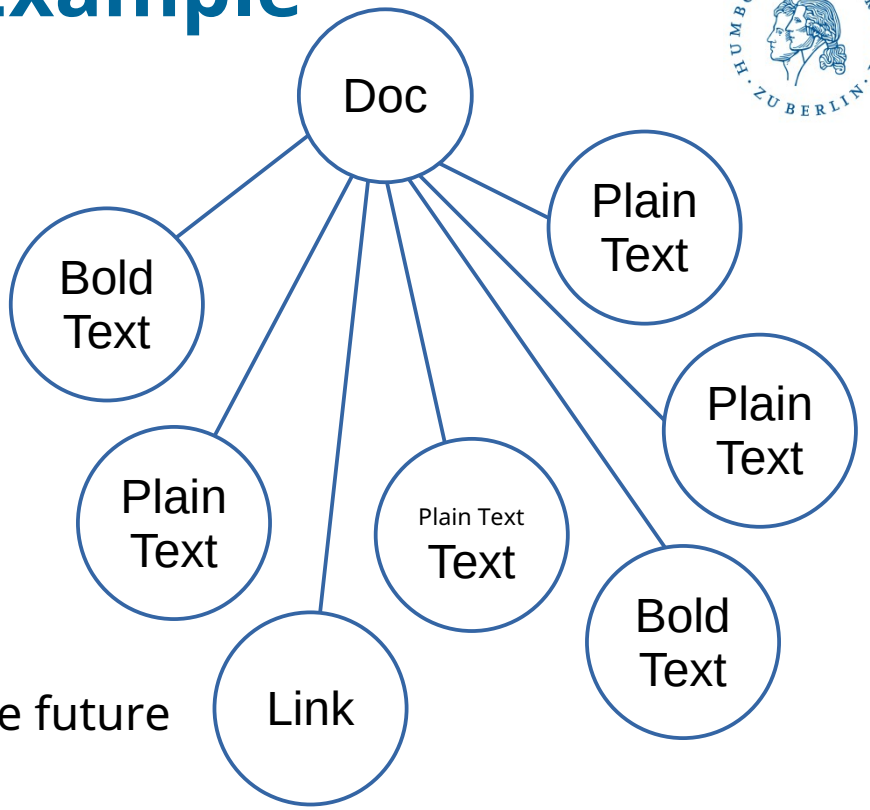
In Java:

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

Visitor: Overview

Description	Content
Pattern-Name + Classification	Visitor – Behavioral Pattern
Purpose	Separation of Algorithm (Operations) and Data on Which the Algorithm is Applied
Motivation	Through separation, new algorithms/functions can be applied to existing object(-structures) without having to change these objects/structures.
Applicability	Structure with many classes present. Functions to be applied depending on the respective class are desired. The amount of classes is stable. You want to add new operations.
Consequences	Simply add new operations. Grouped related operations in a visitor. Add new elements is difficult (adjustment of all Visitors). Visitors can save state. Elements must provide / implement an interface.

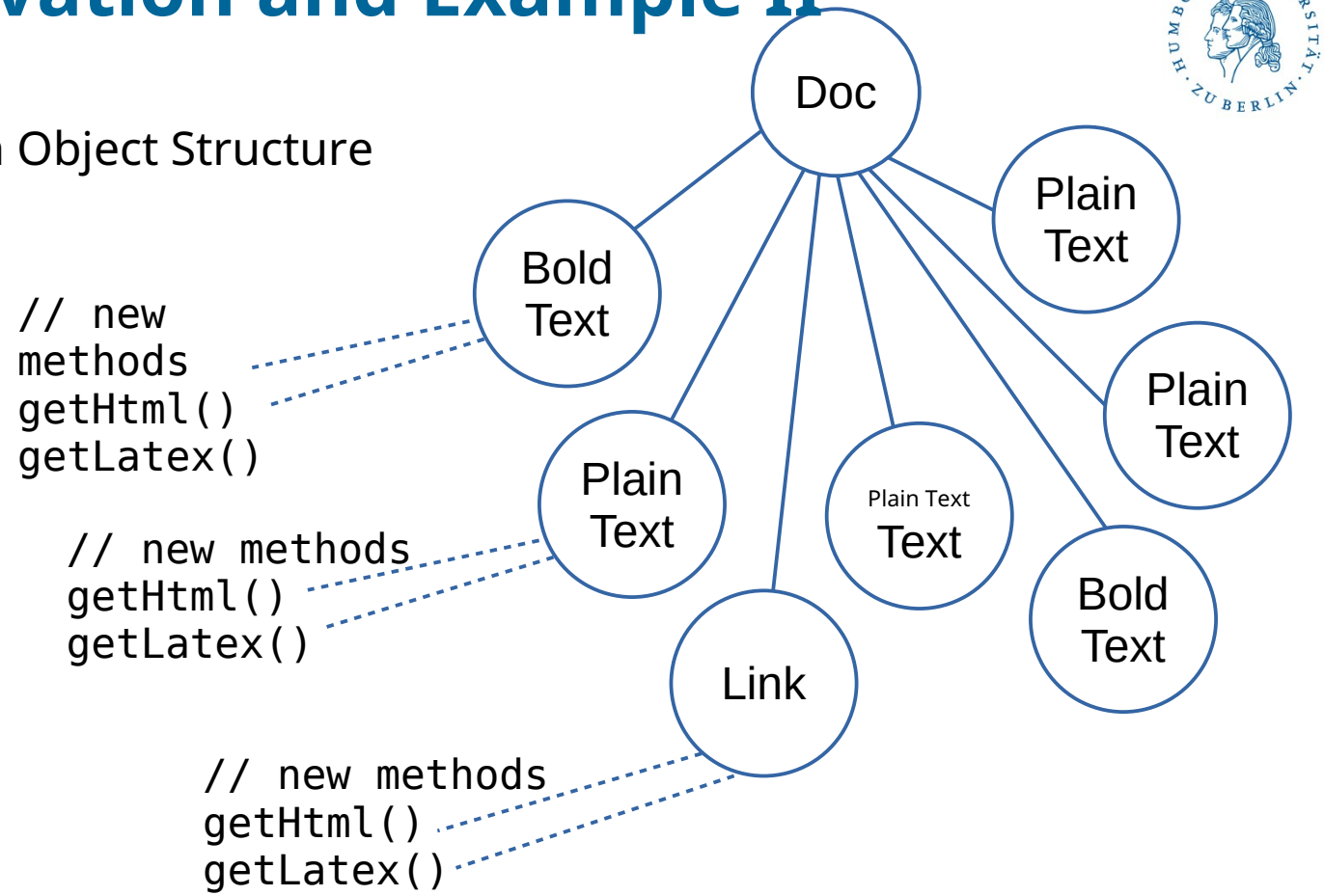
Visitor: Motivation and Example



- Object structure that describes a (simplified) document containing plain text, bold text, and links.
- A document of this kind should be processed in various ways (Functionality)
  - Translation to HTML
  - Translation to LaTeX
  - …and potentially more features in the future
- Each individual class (such as PlainText, BoldText and Link) must be processed differently
  - z.B. for HTML: bold text and link
  - z.B. for LaTeX: \extbf{bold text} and \\href{url}{link}

Visitor: Motivation and Example II

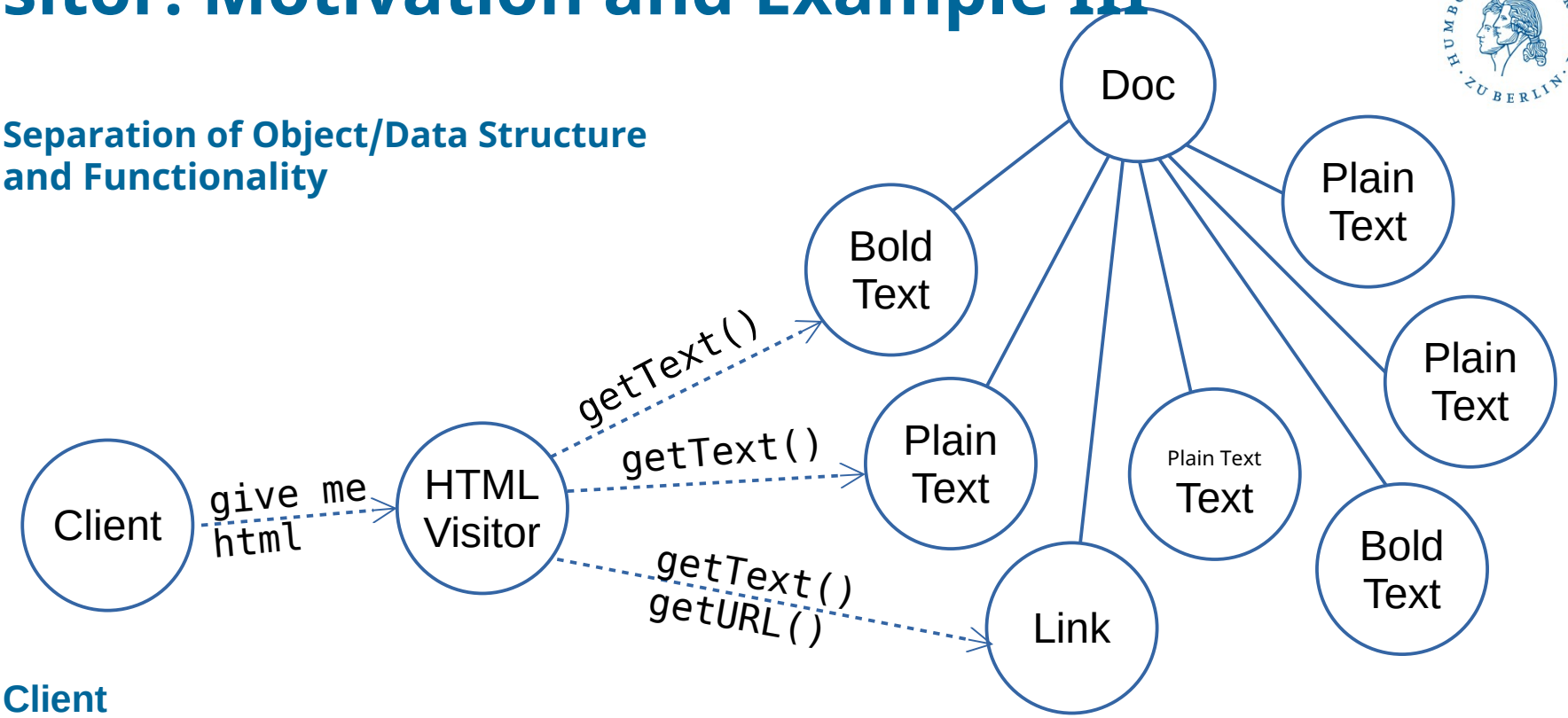
- Implementation in Object Structure or Data Structure?



- Each (new) functionality is implemented in three classes
  – Functionality is distributed across these classes
- Expansion of the document structure with a new element (e.g., italic text)
  – Functionality is implemented in four classes
- **Solution: Separation of object/data structure and functionality**

Visitor: Motivation and Example III

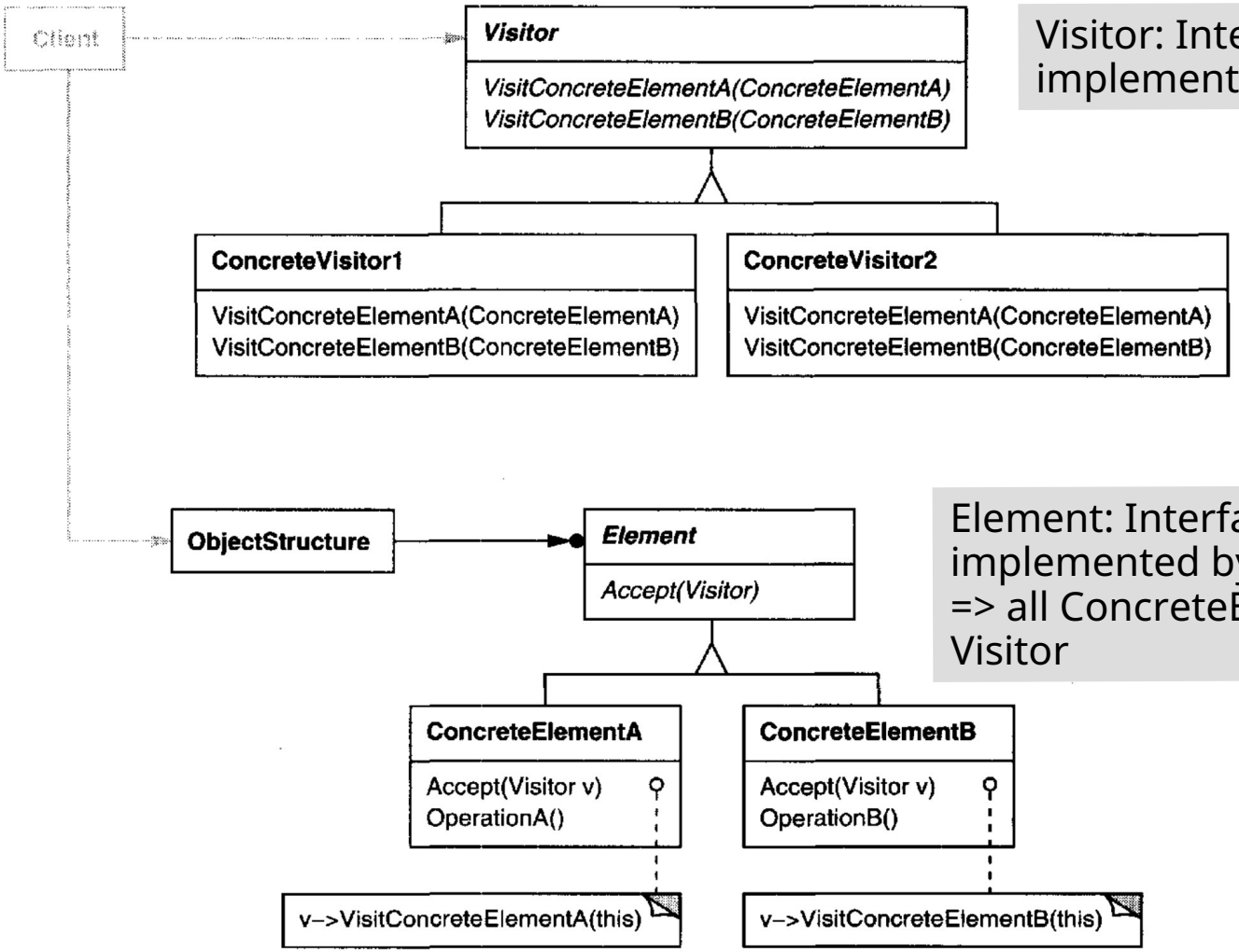
- **Separation of Object/Data Structure and Functionality**



- **Client**
  - Uses Visitor to execute the desired functionality on the object structure
  - Reads the visitor through the object structure
- **Visitor**
  - Get information from every object of the structure and perform the desired functionality
  - Grouped operations (e.g. conversion according to HTML)
- Add new behavior without having to adapt the object structure
  - New Behavior in Existing Visitor
  - New Visitor (e.g., LaTeX Visitor for conversion to LaTeX)

Visitor: Structure and Roles

Client uses Visitor to carry out the desired functionality on the Object Structure, i.e. on all elements of the structure

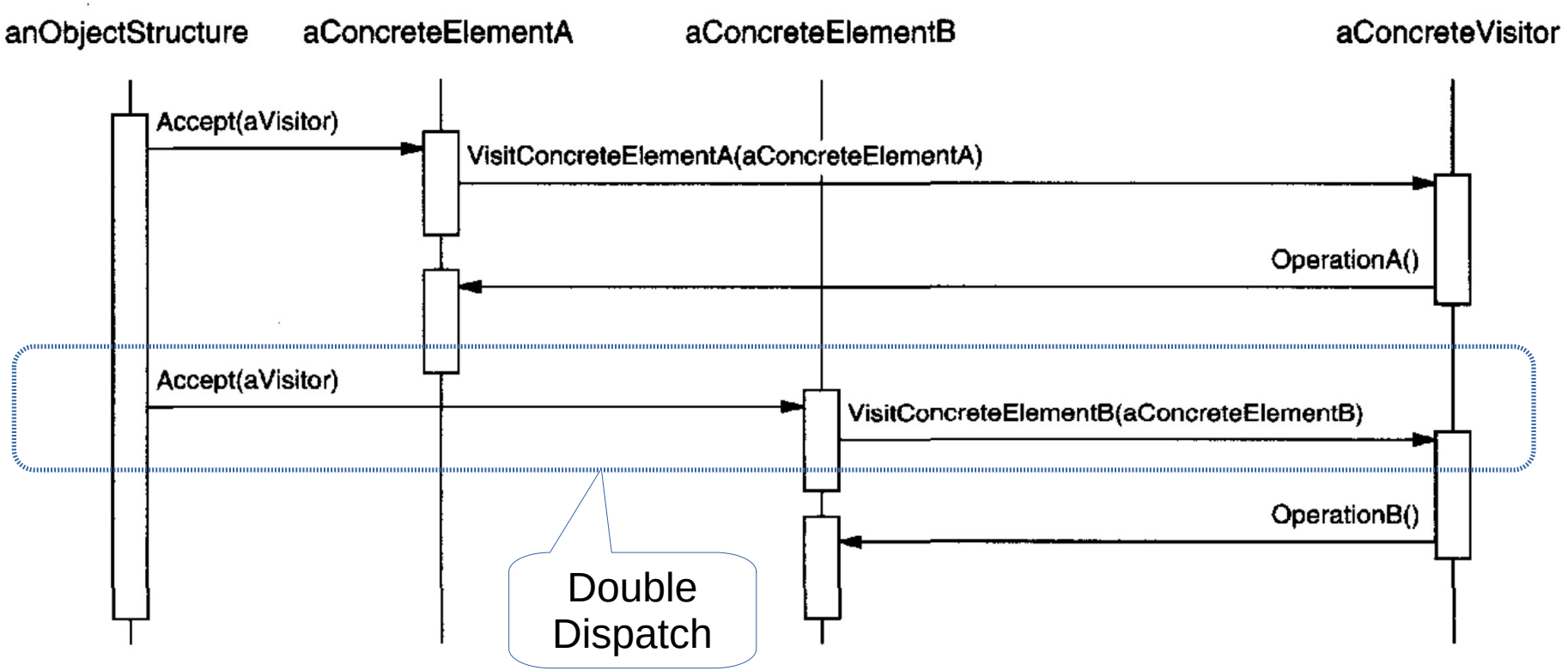


Visitor: Interface or abstract class, implemented by all ConcreteVisitors

Concretevisitor can have condition and enrich it during the visit to several elements

Element: Interface or abstract class, implemented by all ConcreteElements => all ConcreteElements can accept a Visitor

Visitor: collaboration



"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: Example

Object structure

Element

```
public abstract class Element {

    public abstract String accept(Visitor visitor);

}
```

```
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;
    }

}
```

```
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: Example II

Object structure

All elements of the object structure accept a Visitor:
`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: Example III

Visitor

Visit (...) can also be void; Samm
up the results in the visitor
Often different name scheme:

```
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: Example 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
\itle{Document Title}
This is just plain text. \extbf{Some bold text.} \\href{http://www.google.com}{Search here!} This is just more text.
```

References/Literature

- 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.