Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InIT Institut für angewandte
Informationstechnologie

Bachelor of Science (BSc) in Informatik

Modul Software-Entwicklung 1 (SWEN1)

LE 09 – Entwurf mit Design Pattern II
Zusammenfassung

SWEN1/PM3 Team:
R. Ferri (feit), D. Liebhart (lieh), K. Bleisch (bles), G. Wyder (wydg)

Ausgabe: HS24

Um was geht es?

- Weitere GoF Design Patterns, die Lösungen für Probleme anbieten, die oft in der Software-Entwicklung auftauchen.

Lernziele LE 09 – Entwurf mit Design Patterns II

- Sie sind in der Lage:
  - Den Aufbau der folgenden Design Patterns zu erklären und sie anzuwenden:
    - Decorator
    - Observer
    - Strategy
    - Composite
    - State
    - Visitor
    - Facade

Agenda

1. **Repetition Aufbau von Design Patterns**
2. Design Patterns
3. Wrap-up und Ausblick

Repetition Aufbau Design Patterns

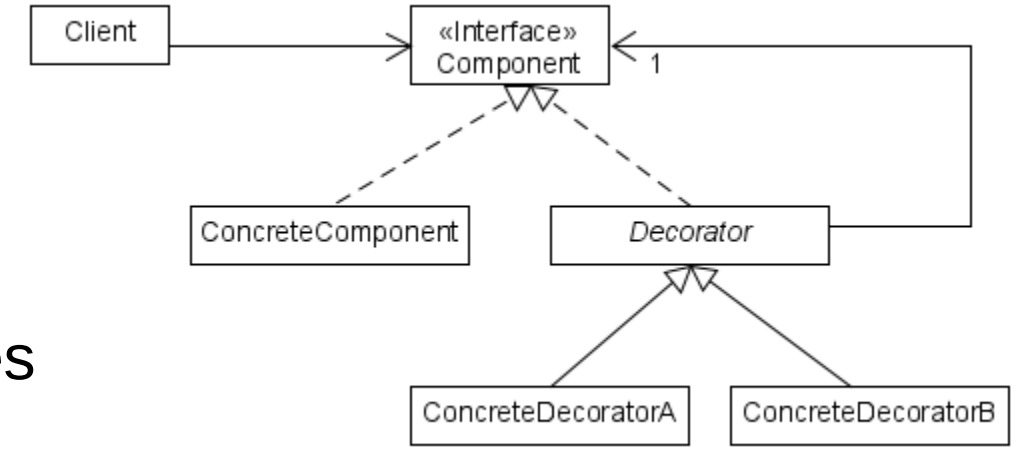
- Beschreibungsschema:
  - Name
  - Beschreibung Problem
  - Beschreibung Lösung
  - Hinweise für Anwendung
  - Beispiele
- GRASP: Design Prinzipien
- GoF: Ausgefeiltere Spezialfälle von GRASP

Agenda

1. Repetition Aufbau von Design Patterns
2. **Design Patterns**
3. Wrap-up und Ausblick

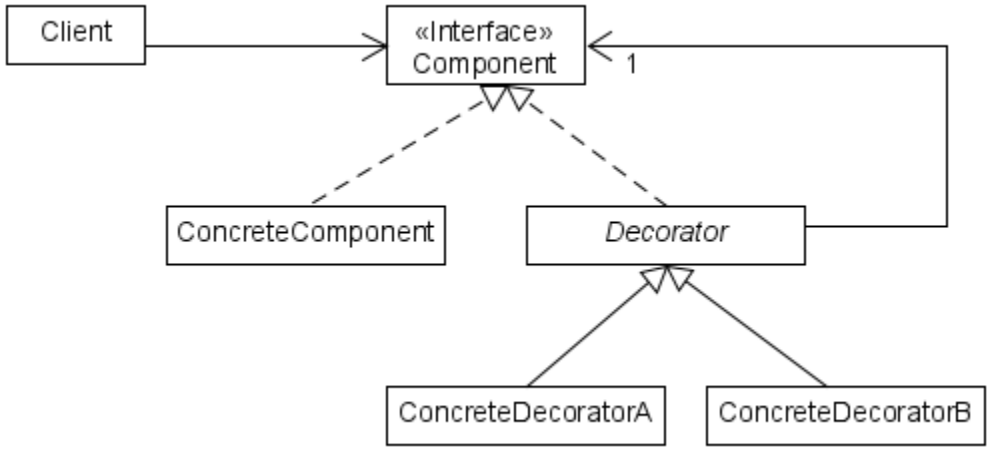
Decorator: Problem und Lösung

- Problem:
  - Ein Objekt (nicht eine ganze Klasse) soll mit zusätzlichen Verantwortlichkeiten versehen werden.
- Lösung
  - Ein Decorator, der dieselbe Schnittstelle hat wie das ursprüngliche Objekt, wird vor dieses geschaltet. Der Decorator kann nun jeden Methodenaufruf entweder selber bearbeiten, ihn an das ursprüngliche Objekt weiterleiten oder eine Mischung aus beidem machen.



Decorator: Hinweise

- Hinweise
  - Strukturell identisch mit dem Proxy Design Pattern, hat aber eine andere Absicht.
  - Eigentlich identisch mit dem «Composite» Design Pattern, wenn die Anzahl Elemente 1 ist, hat aber natürlich auch eine andere Absicht.



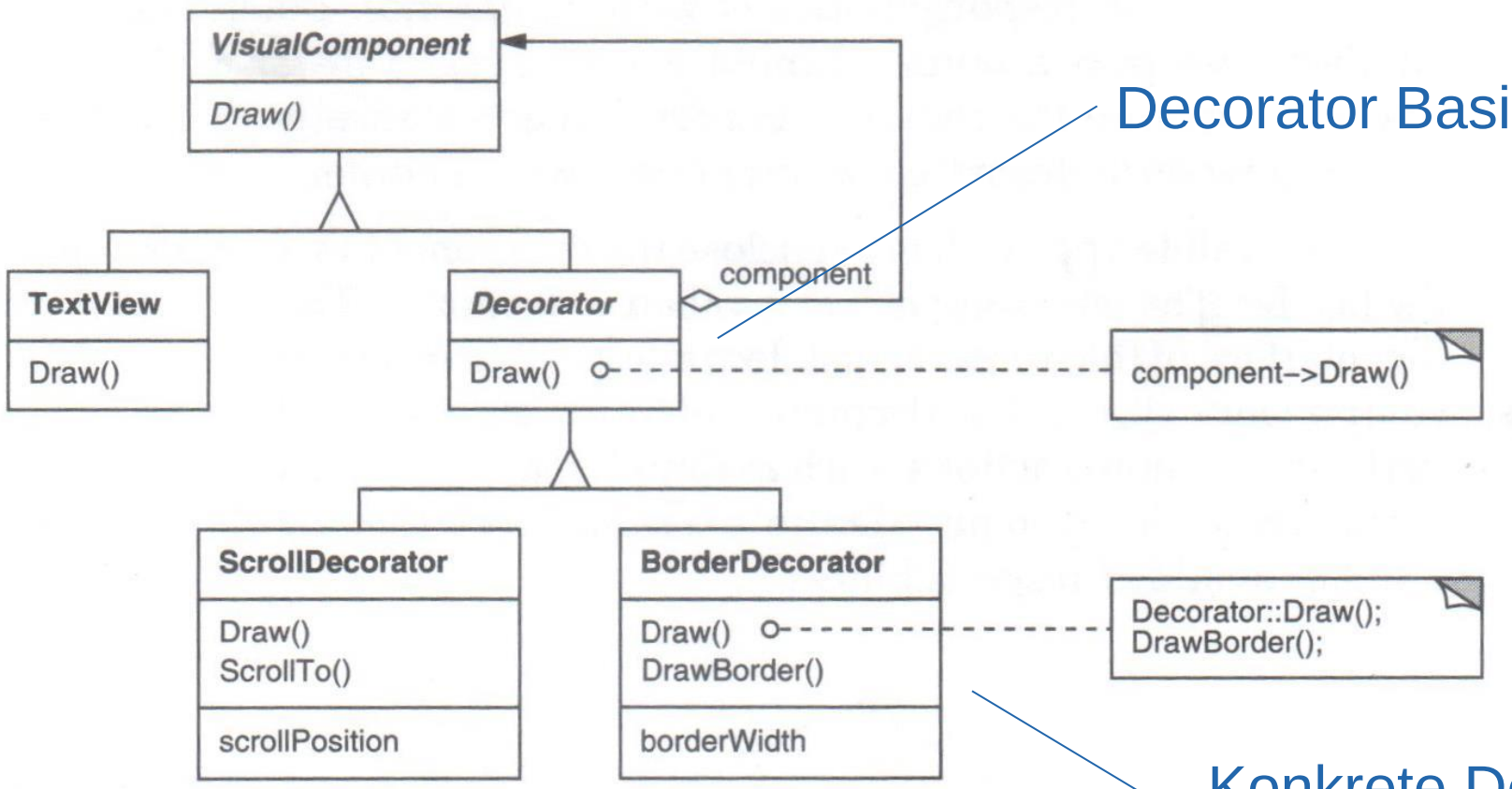
Decorator: Beispiele

- JDK
  - In der Package java.io gibt es die Klasse FilterInputStream, die die Basisklasse für Decorators von InputStream Klassen darstellt. Mit diesen Filter Klassen kann eine ganze Kette von InputStream Klassen flexibel zusammengehängt werden, die dann eine komplexe Gesamtverantwortlichkeit bieten.
  - Analog dazu gibt es FilterOutputStream, FilterReader und FilterWriter Klassen.
- GoF Beispiel (siehe nachfolgende Folie)
  - Hier wird der Begriff «Decorator» wörtlich genommen und ermöglicht, dass eine beliebige Komponente mit einem speziellen Rahmen versehen wird.

Decorator: Beispiele

- Larman, Point Of Sale Terminal (siehe nachfolgende Folie)
  - Auch wenn das Design Pattern explizit nicht erwähnt wird, ist das Beispiel über «Performanz mit lokaler Zwischenspeicherung» genau die Anwendung des Decorator Patterns.
- Forum, inklusive Code: Decorator, der die Überprüfung einer Contribution einleitet.

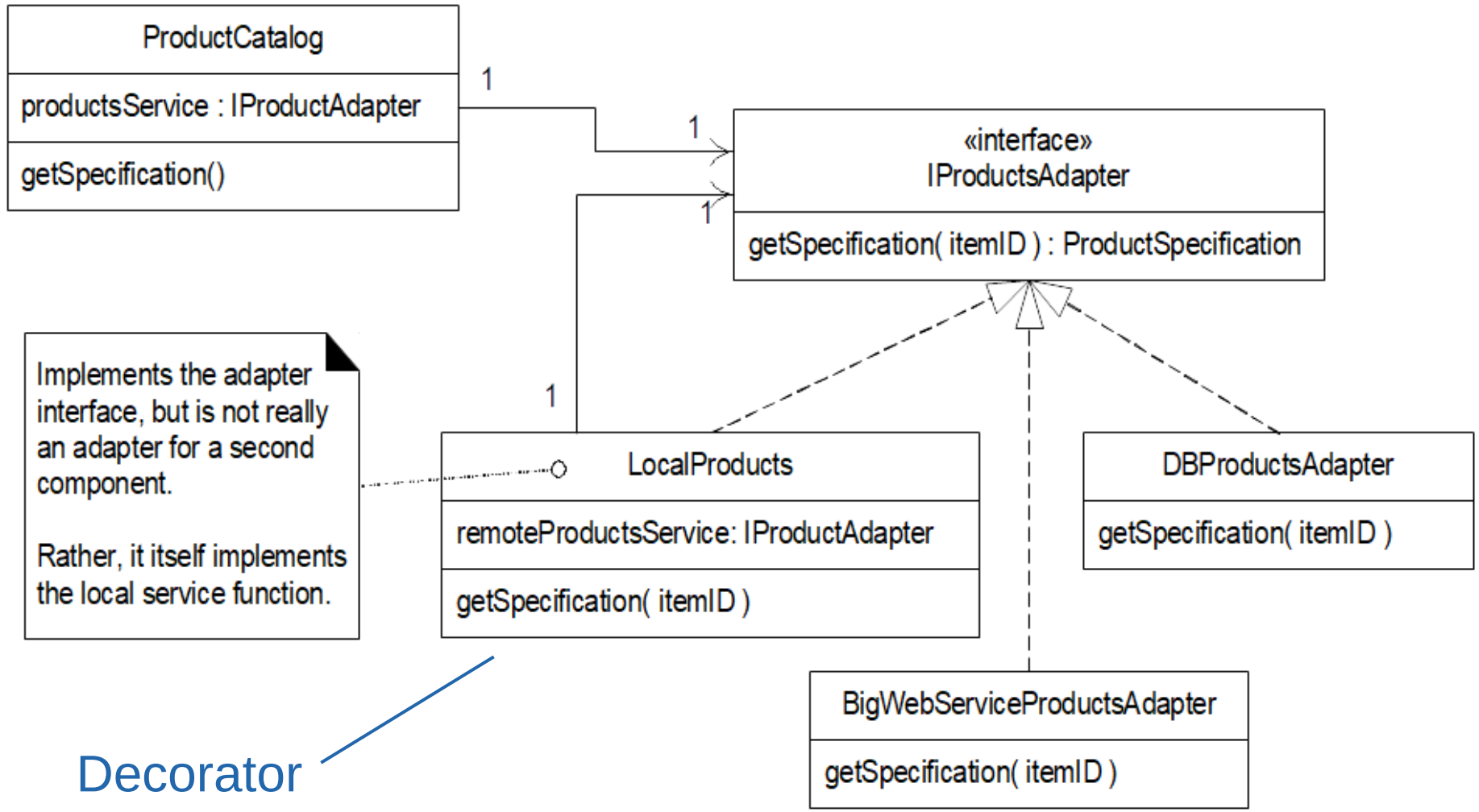
Decorator Beispiel: GoF



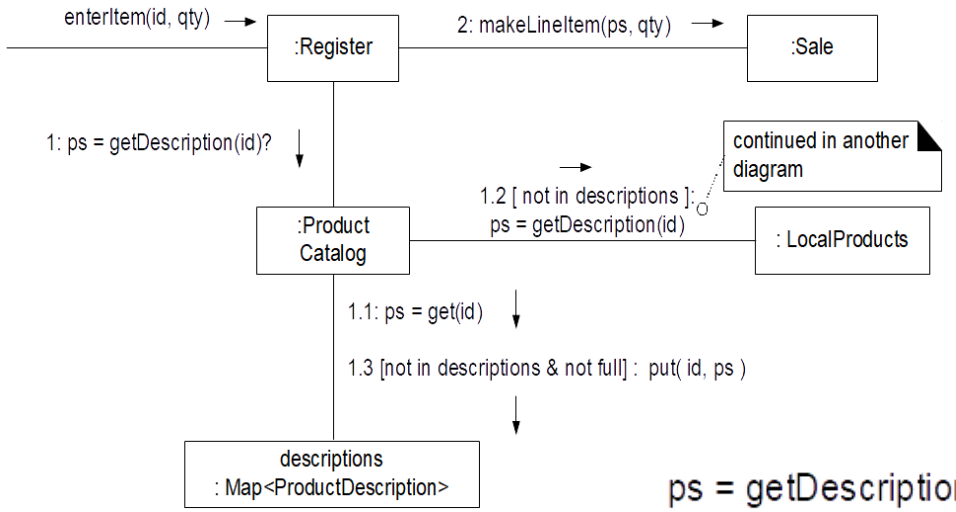
Decorator Basisklasse

Konkrete Decorator Klassen

Decorator Beispiel: Point Of Sale Terminal

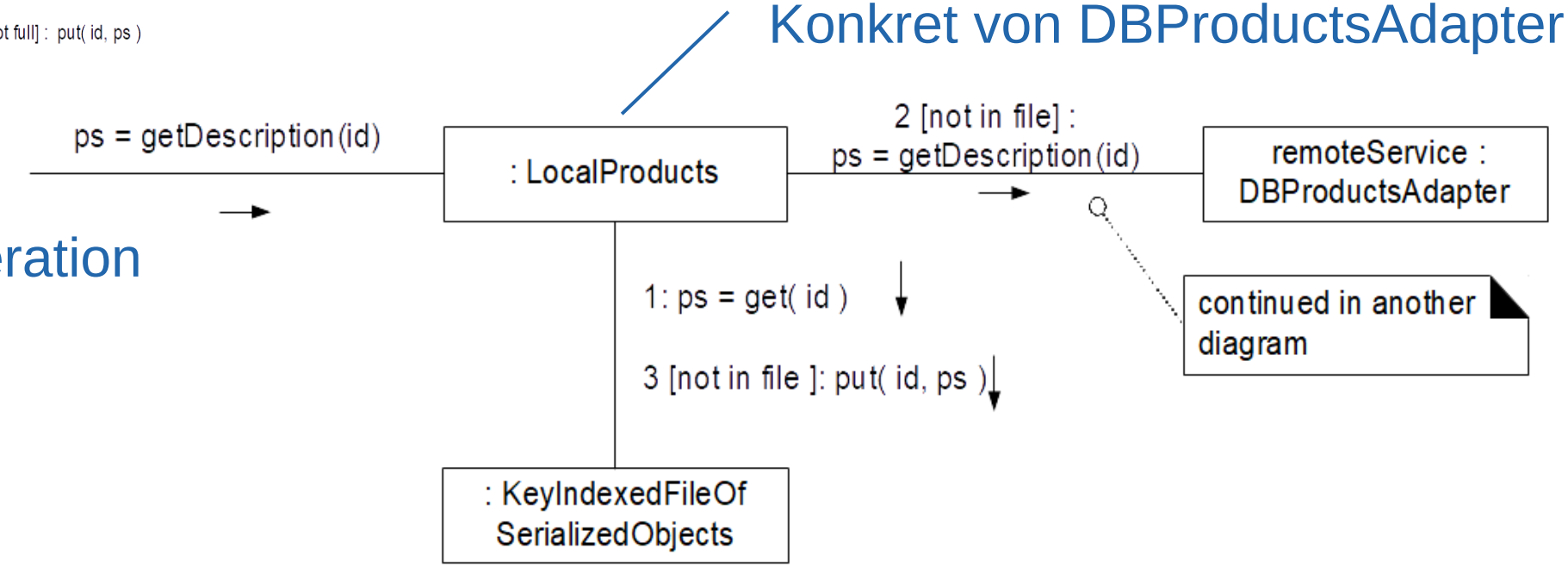


Decorator Beispiel: Point Of Sale Terminal



2. Teil von enterItem(id,qty) mit dem Einsatz des Decorators

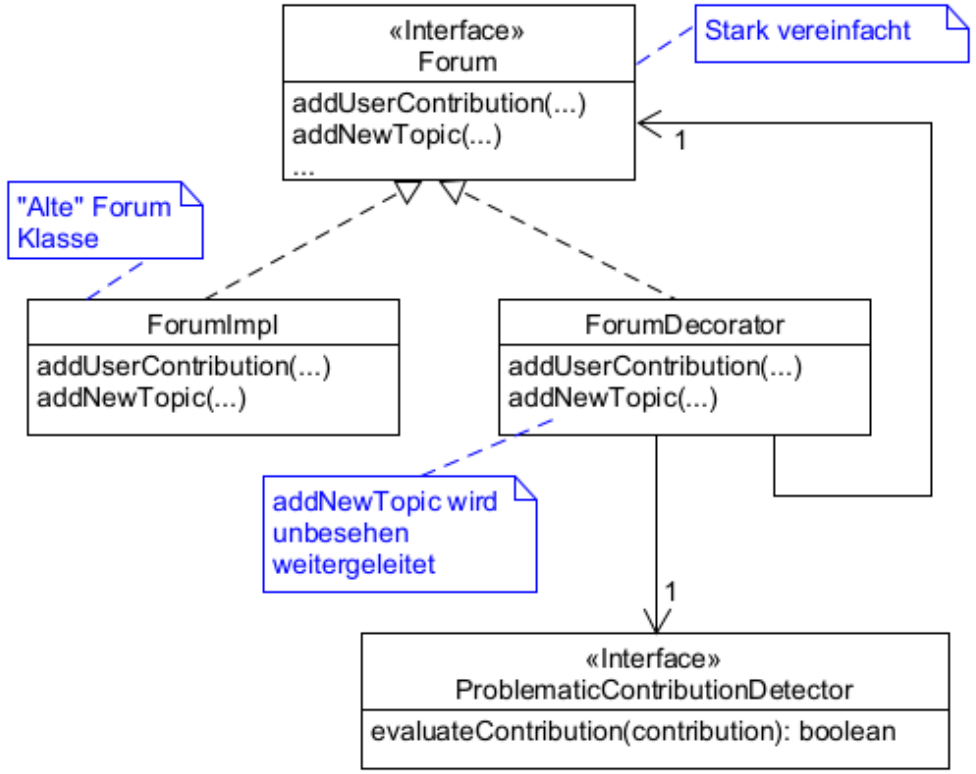
Decorator von IProductsAdapter, Konkret von DBProductsAdapter

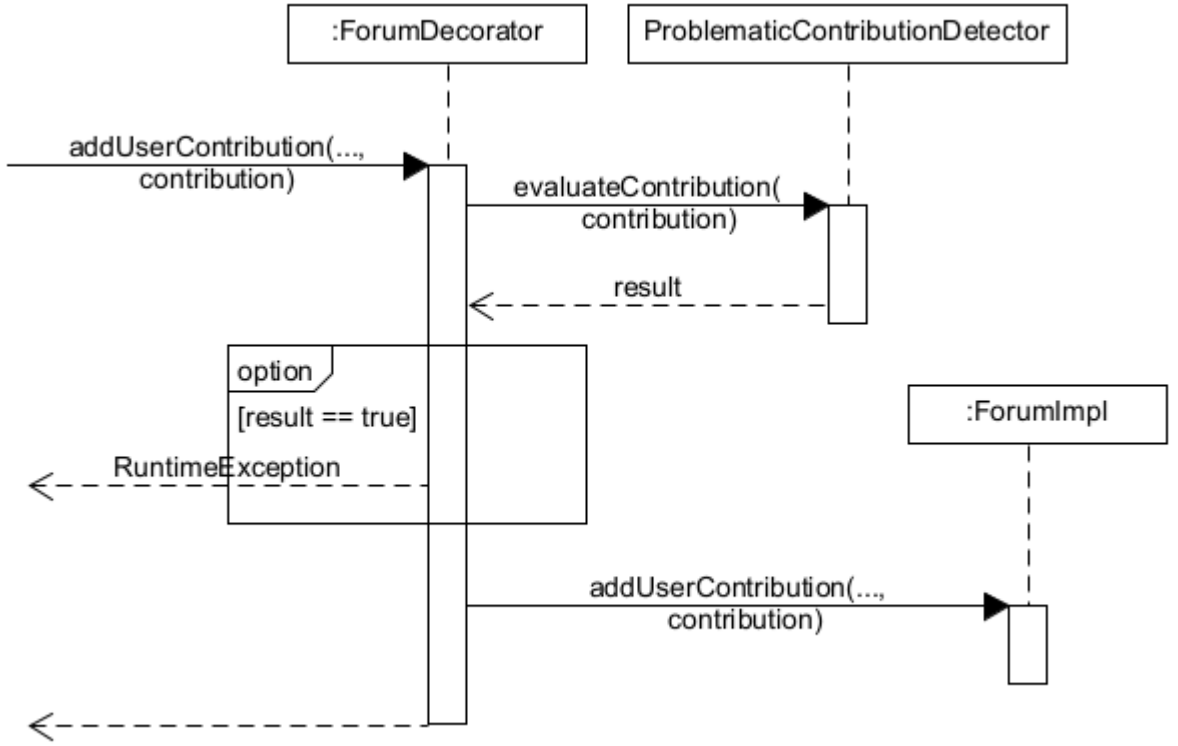


Start der Systemoperation enterItem(id, qty)

Decorator Beispiel: Forum

- Ein Decorator leitet die Überprüfung von Contributions ein. Alle anderen Methoden als addUserContribution werden weitergeleitet





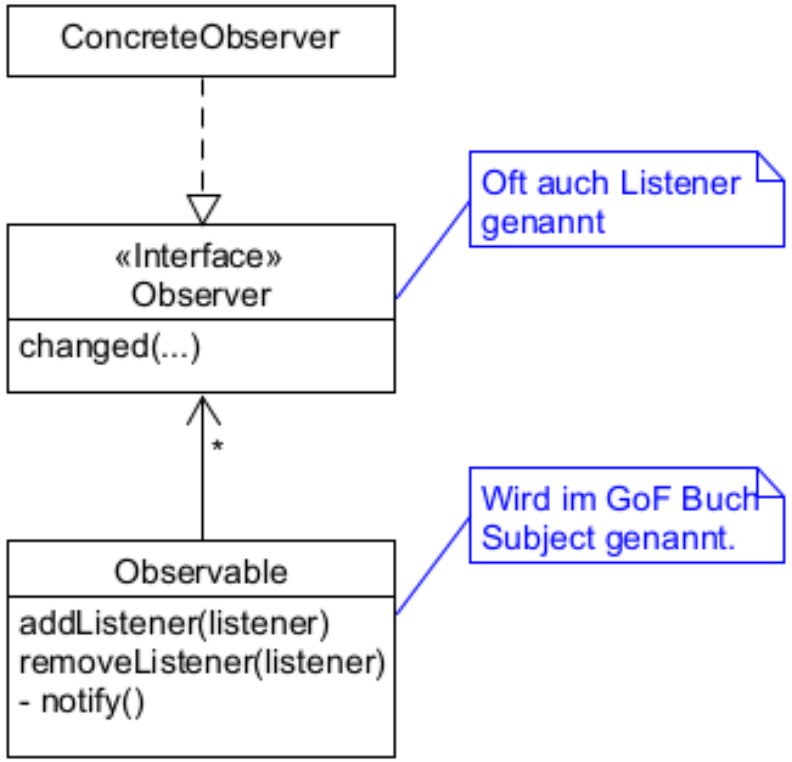
Decorator Beispiel: Forum Code


```
public class ForumDecorator implements Forum {
    private Forum forum;
    private ProblematicContributionDetector problematicContributionDetector;

    @Override
    public void addNewTopic(String sessionId, String topicName, String description) {
        forum.addNewTopic(sessionId, topicName, description);
    }

    @Override
    public boolean addUserContribution(String sessionId, String topicName,
            String discussionName, String contribution) {
        if (problematicContributionDetector.evaluateContribution(contribution)) {
           throw new RuntimeException("Problematic Contribution");
        }
        forum.addUserContribution(sessionId, topicName, discussionName, contribution);
    }
}
```

Decorator Funktionalität

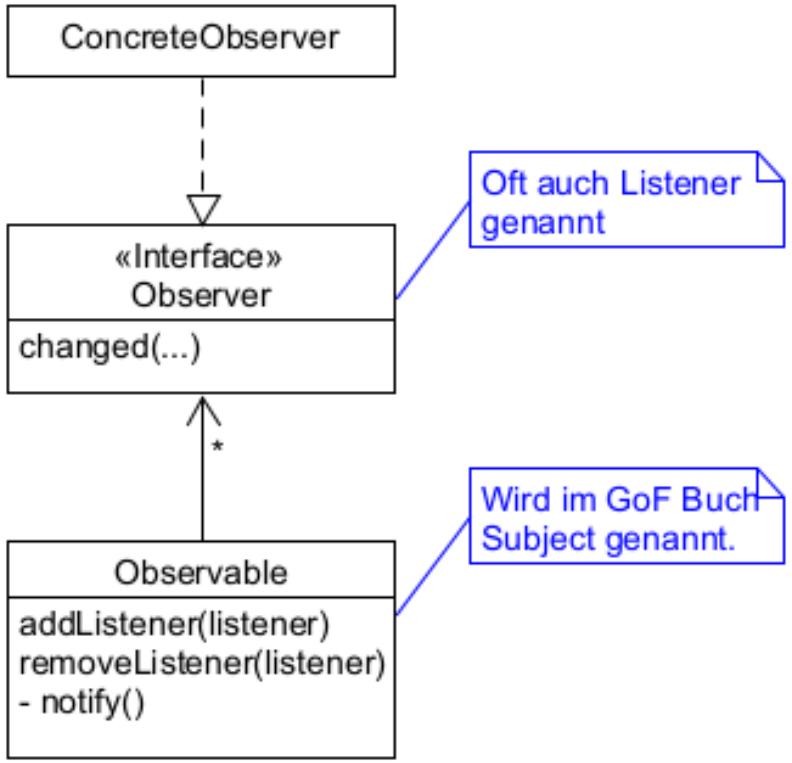
Observer: Problem und Lösung

- Problem:
  - Ein Objekt soll ein anderes Objekt benachrichtigen, ohne dass es den genauen Typ des Empfängers kennt.
- Lösung
  - Ein Interface wird definiert, das nur dazu dient, ein Objekt über eine Änderung zu informieren. Dieses Interface wird vom «Observer» implementiert. Das «Observable» Objekt benachrichtigt alle registrierten «Observer» über eine Änderung.



Observer: Hinweise

- Hinweise
  - Oft wird dieses Pattern auch «Publish-Subscribe» genannt.
  - Observable kennt nur Observer, aber nicht den wahren Typ ConcreteObserver.
  - 2 Phasen : Zuerst die Registrierung der Observer, dann die Benachrichtigungen durch das Observable.



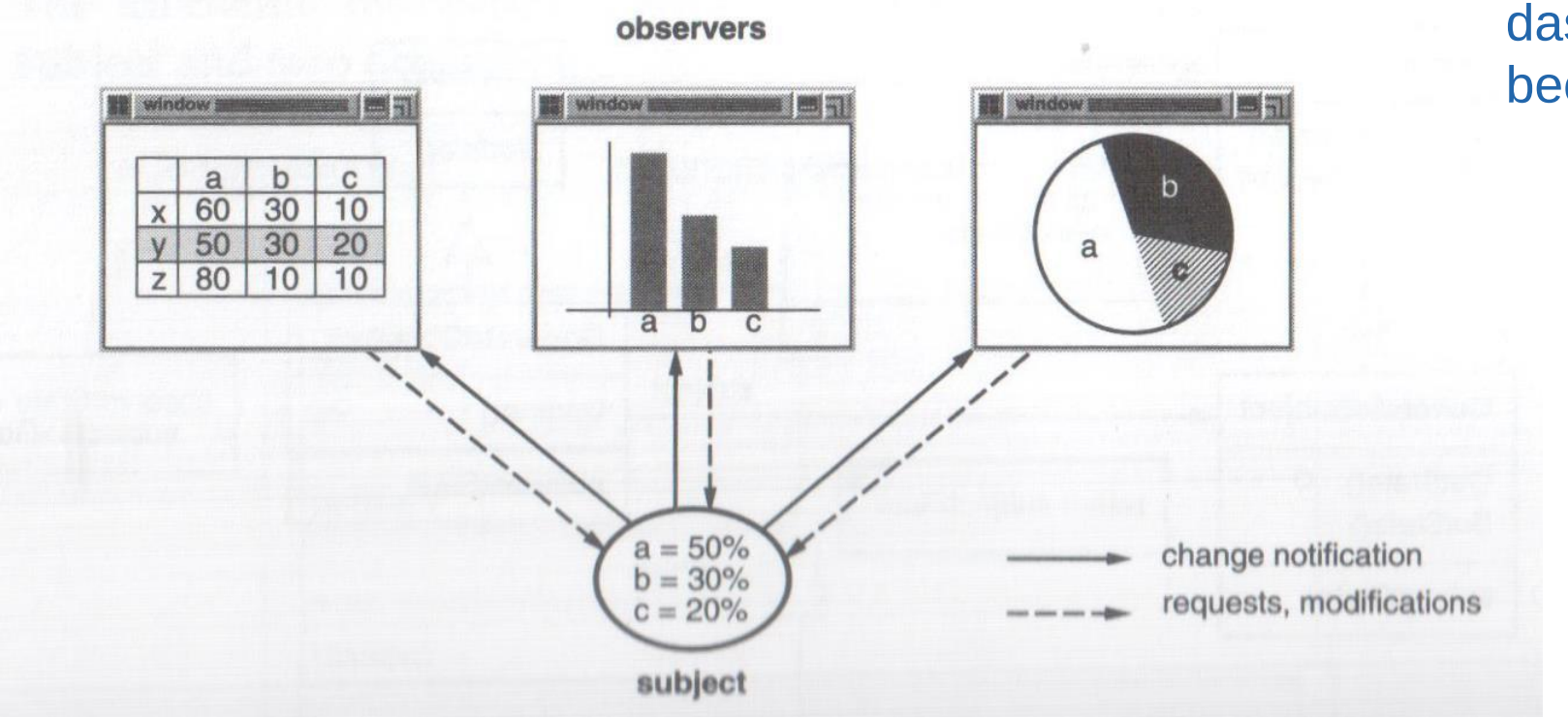
Observer: Beispiele (1/2)

- JDK
  - java.util.Observer und Observable. Gut gemeint, aber schlecht in der Ausführung, daher wird es mittlerweile als veraltet betrachtet. Grund dafür ist das Anti-Pattern «Implementations-Vererbung»
  - AWT, Swing und JavaFX unterstützen Observer für fast alle Properties und für Aktionen, die von den visuellen Komponenten ausgelöst werden können. Diese werden allerdings Listener genannt.
- GoF Beispiel (siehe nachfolgende Folie)
  - Der klassische Fall, dass UI Komponenten Objekte der Domänenlogik beobachten.

Observer: Beispiele (2/2)

- Larman, Point Of Sale Terminal (siehe nachfolgende Folie)
  - Auch hier übernimmt das UI die Rolle des Observers, der die aktuelle Sale-Instanz beobachtet und die Ansicht aktualisiert, sobald der Gesamtpreis sich ändert.
- Erweiterung
  - Mit Hilfe des «Mediator» Pattern kann ein Objekt zwischen Observer und Observable geschaltet werden. Bei diesem Objekt registrieren sich einerseits potentielle Observable, andererseits Observer. Beide benennen dabei eine Eventquelle, die der Observer abonniert und die das Observable mit Events bedient. Im Internet of Things findet dieses Prinzip in Form des Netzwerkprotokolls MQTT – Server/Client grossen Anklang.

Observer Beispiel: GoF

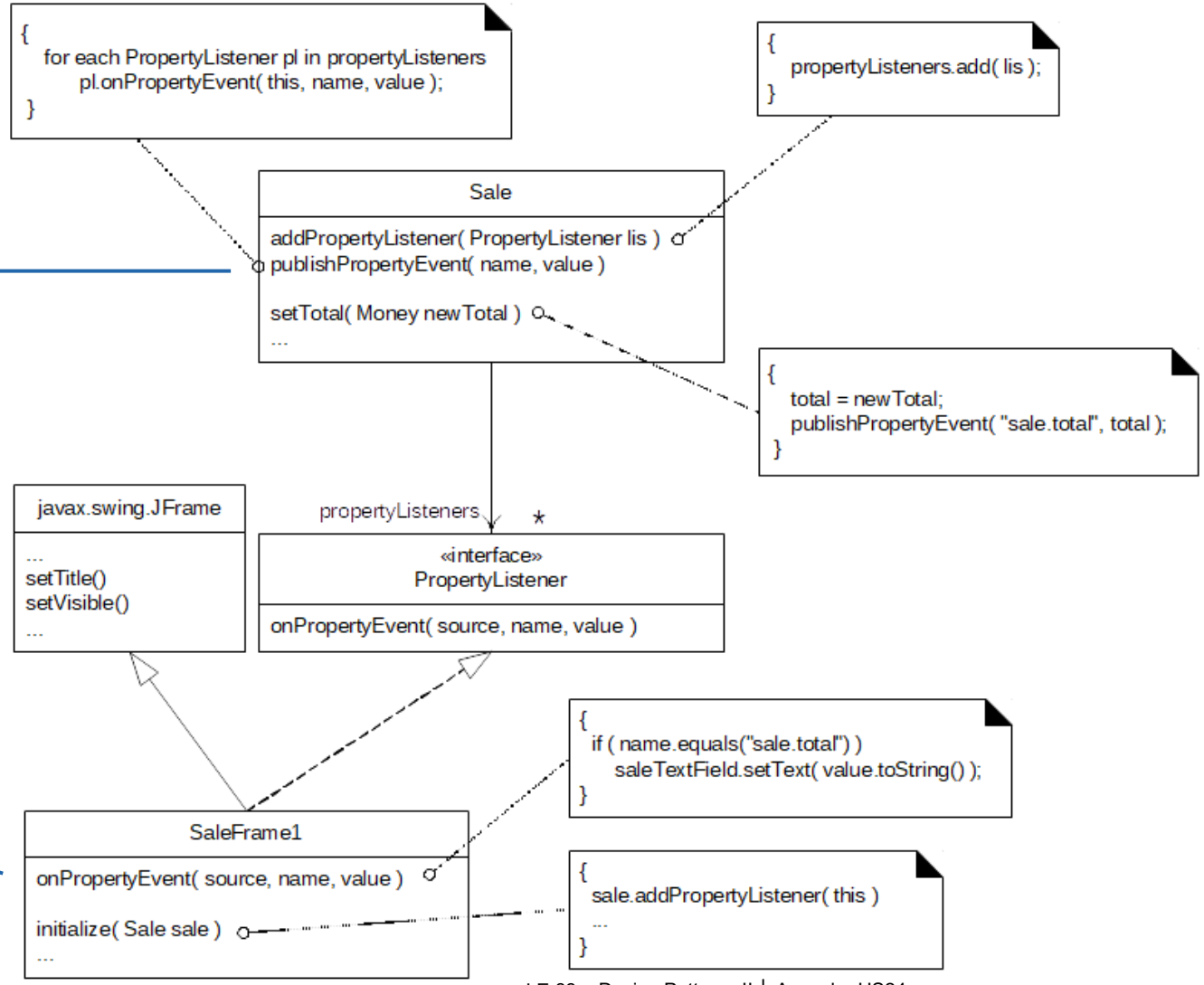


Mehrere Observers, die dasselbe Subject (Observable) beobachten

Observer Beispiel : Point Of Sale Terminal (1)

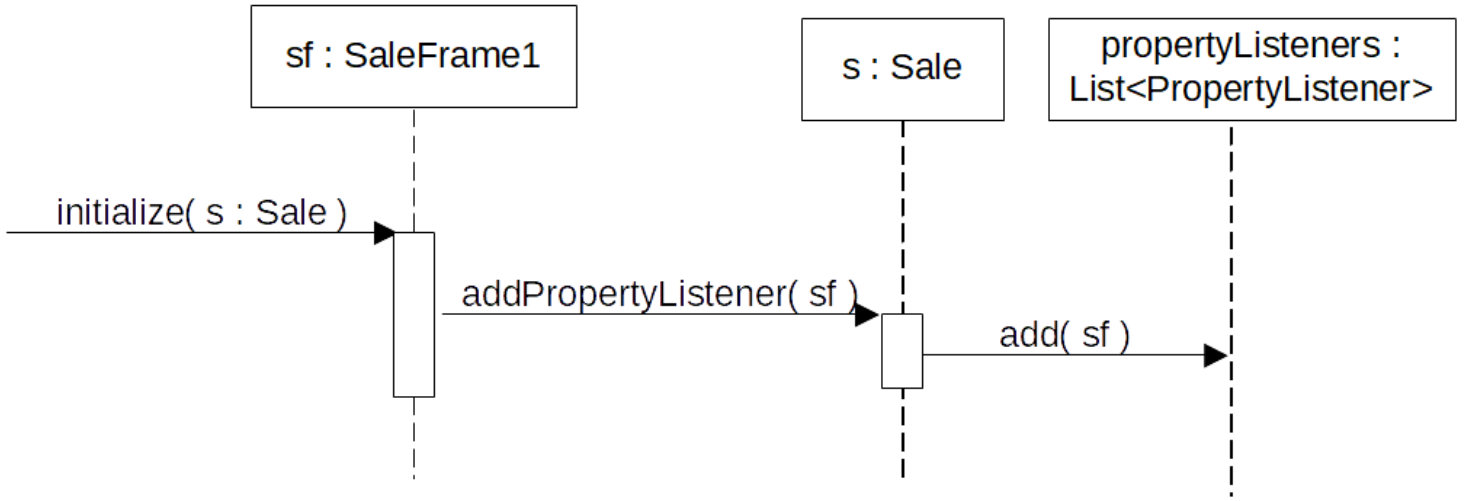
Observable oder Publisher

Observer oder Subscriber oder Listener



Observer Beispiel : Point Of Sale Terminal (1/2)

1. Phase : Registrierung des Observers beim Observable

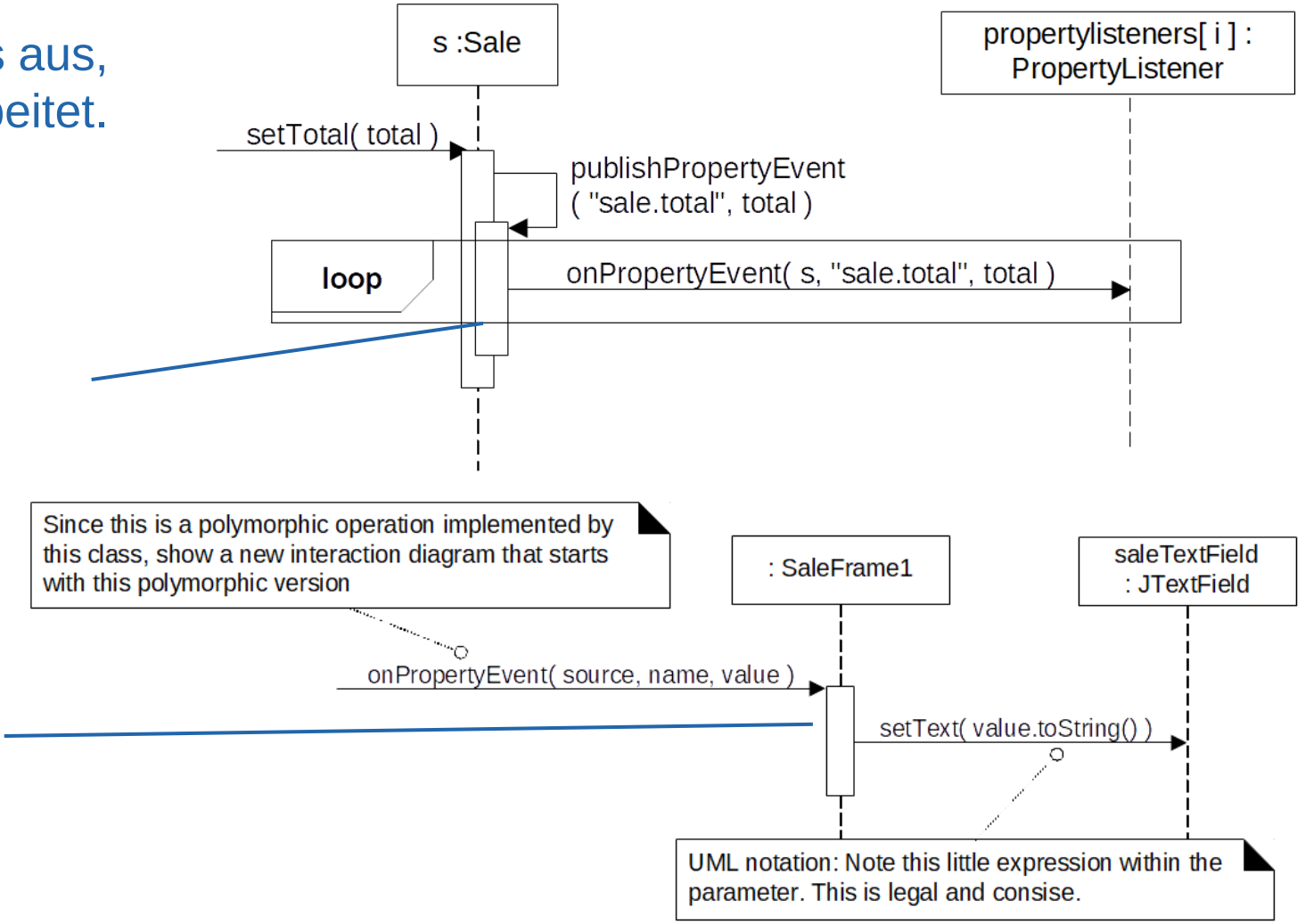


Observer Beispiel: Point Of Sale Terminal (2/2)

2. Phase : Observable löst ein Ereignis aus, das der Observer empfängt und verarbeitet.

Aus der Sicht von Sale werden nur PropertyListener angesprochen

Hinter einem PropertyListener «versteckt» sich aber ein SaleFrame1



Denkpause

Aufgabe 9.1 (5’)

Diskutieren und bearbeiten Sie in Murmelgruppen folgende Fragen:

- Schreiben Sie den Code, der die Observer über die Änderung eines Wertes informiert.
- Was passiert, wenn im Observer während der Verarbeitung des Ereignisses eine Exception geworfen wird?
- Was passiert, wenn sich der Observer während der Behandlung dieser Änderung als Observer austrägt?

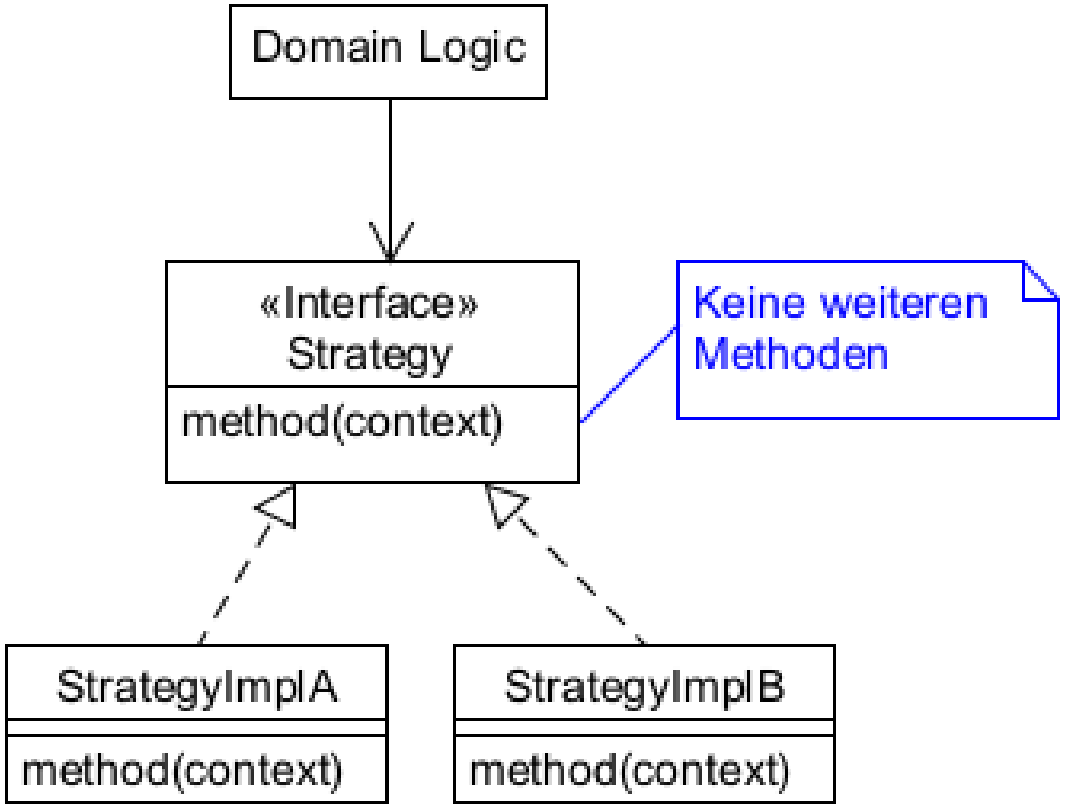
Observer Beispiel: Beispiel Code gemäss Java Standard


```
public class ListenerExample {
    private CopyOnWriteArrayList<ActionListener> listeners = new CopyOnWriteArrayList<>();

    public void addListener(ActionListener listener) { listeners.add(listener); }

    public void removeListener(ActionListener listener) { listeners.remove(listener); }

    public void doSomethingReasonable() {
        //...
        invokeListeners(new ActionEvent(this, 0, "command"));
    }

    protected void invokeListeners(ActionEvent event) {
        listeners.forEach((l)->l.actionPerformed(event));
    }
}
```

Kompakte Darstellung, um Platz zu sparen. Nicht zur Nachahmung empfohlen!

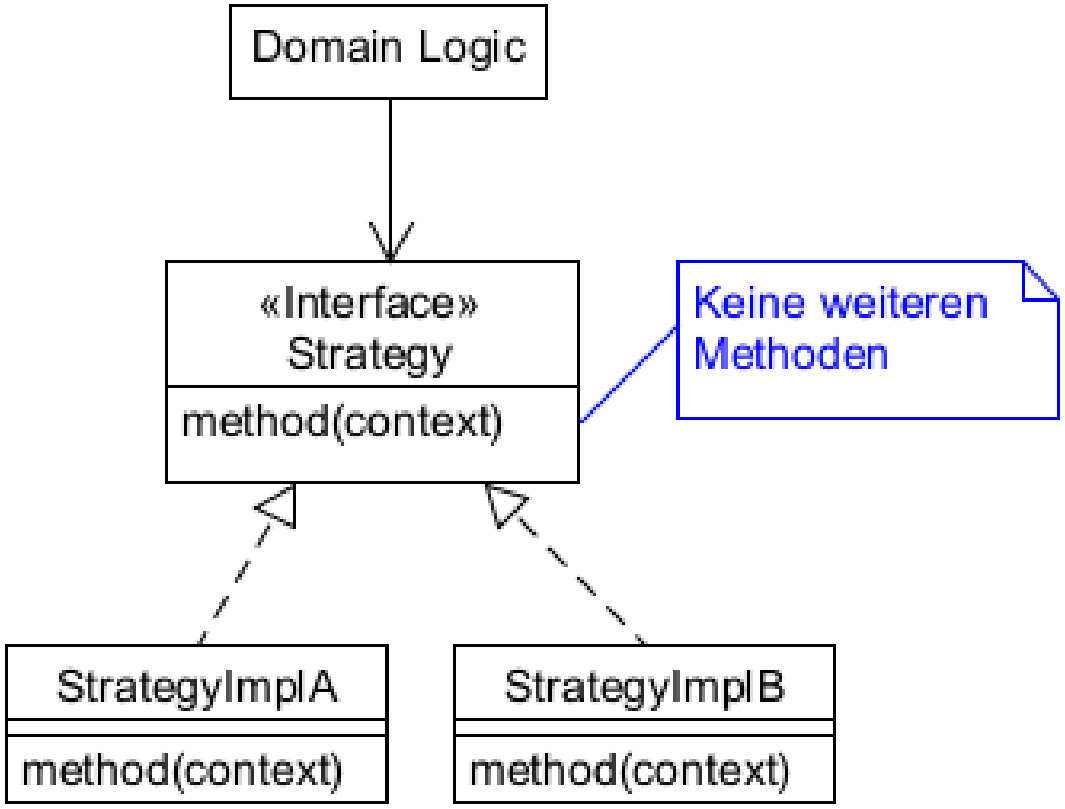
Strategy: Problem und Lösung

- Problem:
  - Ein Algorithmus soll einfach austauschbar sein.
- Lösung
  - Den Algorithmus in eine eigene Klasse verschieben, die nur eine Methode mit diesem Algorithmus hat.
  - Ein Interface für diese Klasse definieren, das von alternativen Algorithmen implementiert werden muss.



Strategy: Hinweise

- Hinweise
  - Als Motivation für die einfache Austauschbarkeit können technische wie auch fachspezifische Gründe vorhanden sein.
  - Das Interface hat nur eine einzige Methode. Als Parameter müssen alle Daten übergeben werden, die der Algorithmus benötigt. Dieser Parameter wird üblicherweise «context» benannt.



Strategy: Beispiele (1/2)

- JDK
  - Die Methode File.listFiles(filter) erwartet einen Parameter vom Typ FileFilter, der eine Strategy darstellt.
  - java.util.Comparator ist ebenfalls eine Strategy.
  - Sogenannte Lambda Ausdrücke, die für Java Streams verwendet werden, sind ebenfalls Strategies.
- GoF Beispiel (siehe nachfolgende Folie)
  - Um Zeilenumbrüche einzufügen, sind verschiedene Algorithmen denkbar.

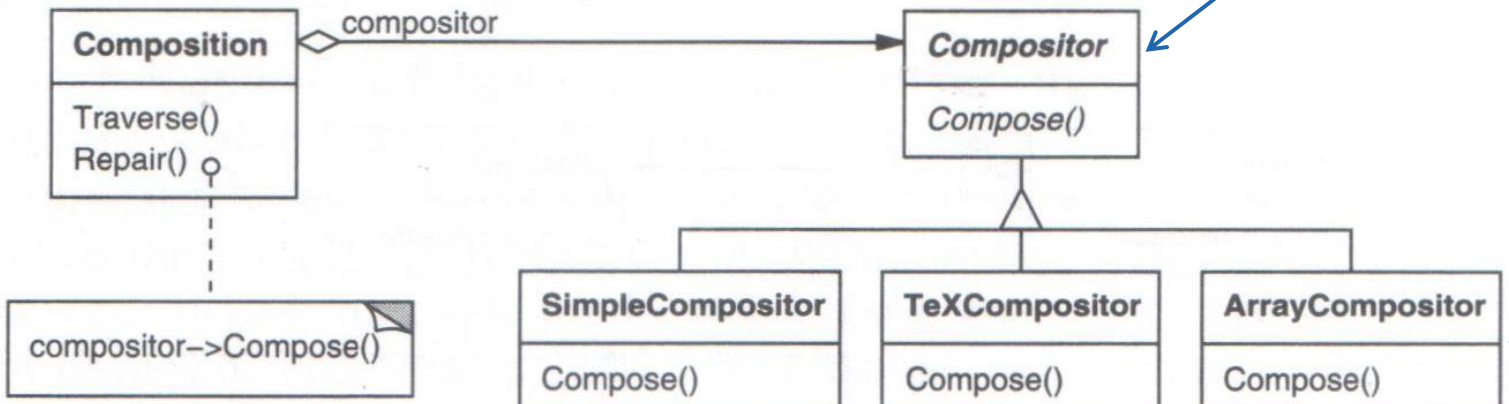
Strategy: Beispiele (2/2)

- Larman, Point Of Sale Terminal (siehe nachfolgende Folie)
  - Die Rabattberechnung wird mit einer Strategy und mit dem Composite Pattern gelöst.
- Code Beispiel: Comparator für Person
- Allgemein
  - Wenn eine Strategy keine Parameter in Instanzvariablen zwischenspeichert (was sie eigentlich nie tun sollte), dann kann sie problemlos als Singleton von mehreren Thread gemeinsam benutzt werden.
  - Wichtig ist, dass alle aktuellen Daten, die eine Strategy benötigt, von aussen über Parameter geliefert werden.

Strategy Beispiel: GoF

Zusammengesetzter Text,
z.B. ein Paragraph

Strategy Interface

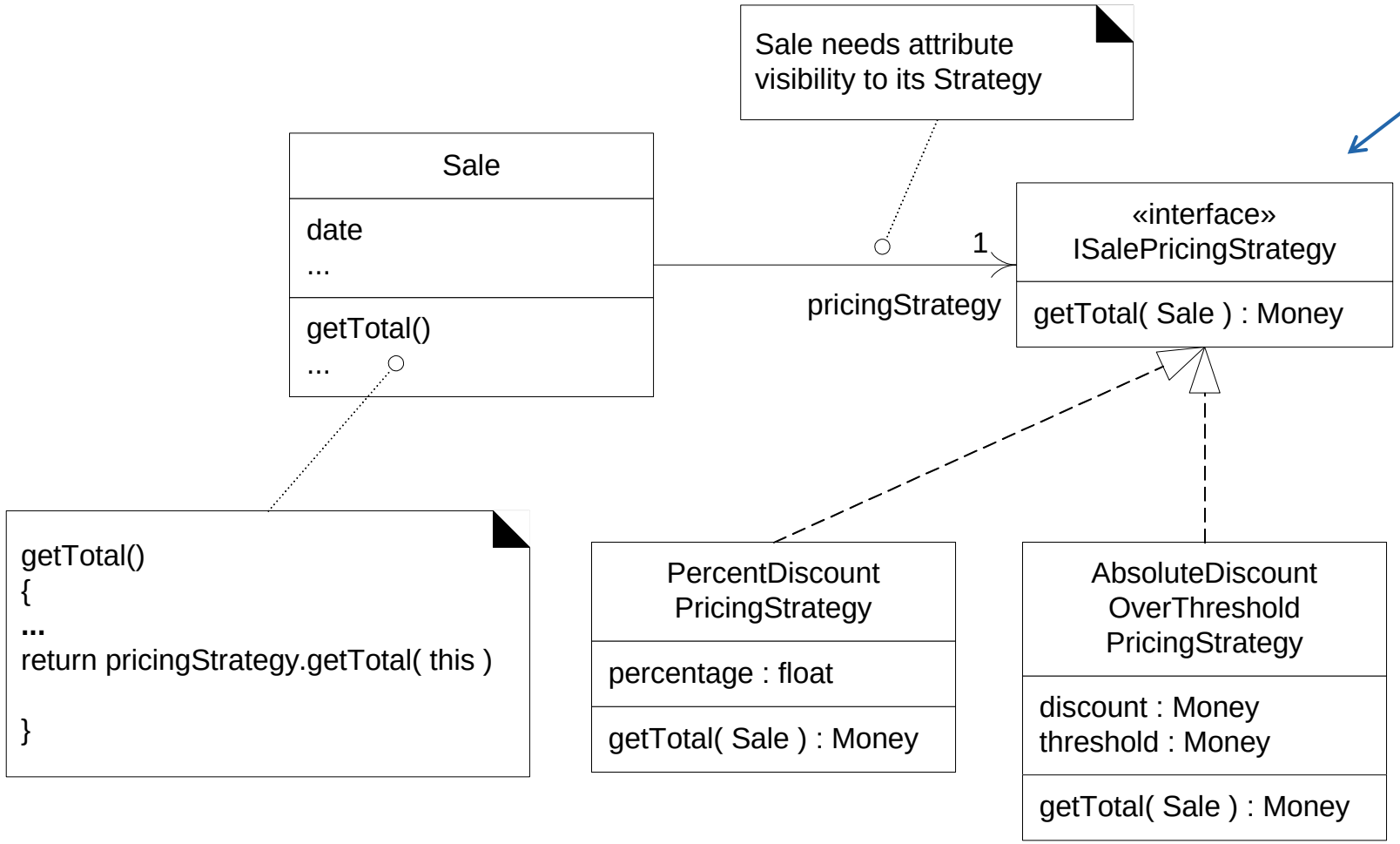


Jede Zeile wird gleich lang

Einfacher Algorithmus, der nur eine Zeile aufs Mal betrachtet

Latex Algorithmus, der eine globale Optimierung vornimmt

Strategy Beispiel POST: DCD

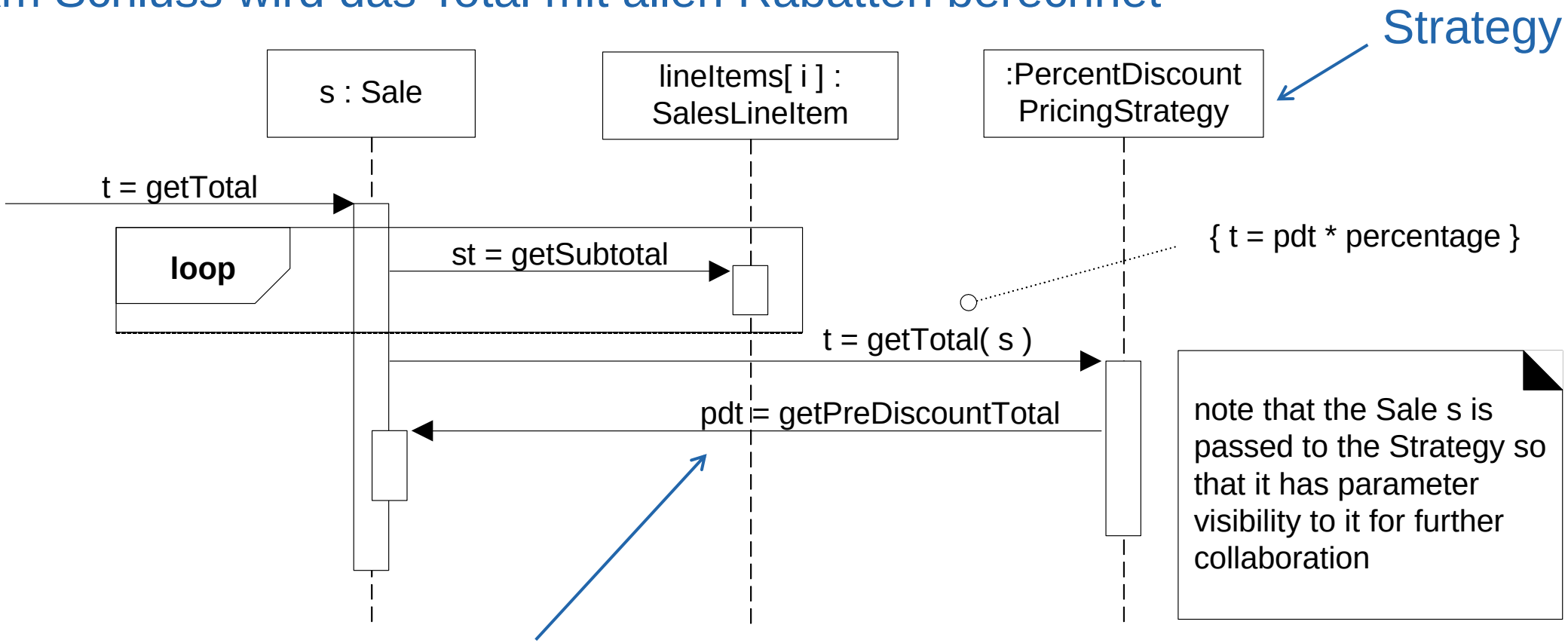


Strategy Interface

Konkrete Implementation, die einen absoluten Rabatt berechnet

Strategy Beispiel POST: Verwendung

Am Schluss wird das Total mit allen Rabatten berechnet



Strategy greift auf Daten von Sale (Context Parameter) zu.

Strategy Beispiel: Comparator<Person> Varianten (1/2)

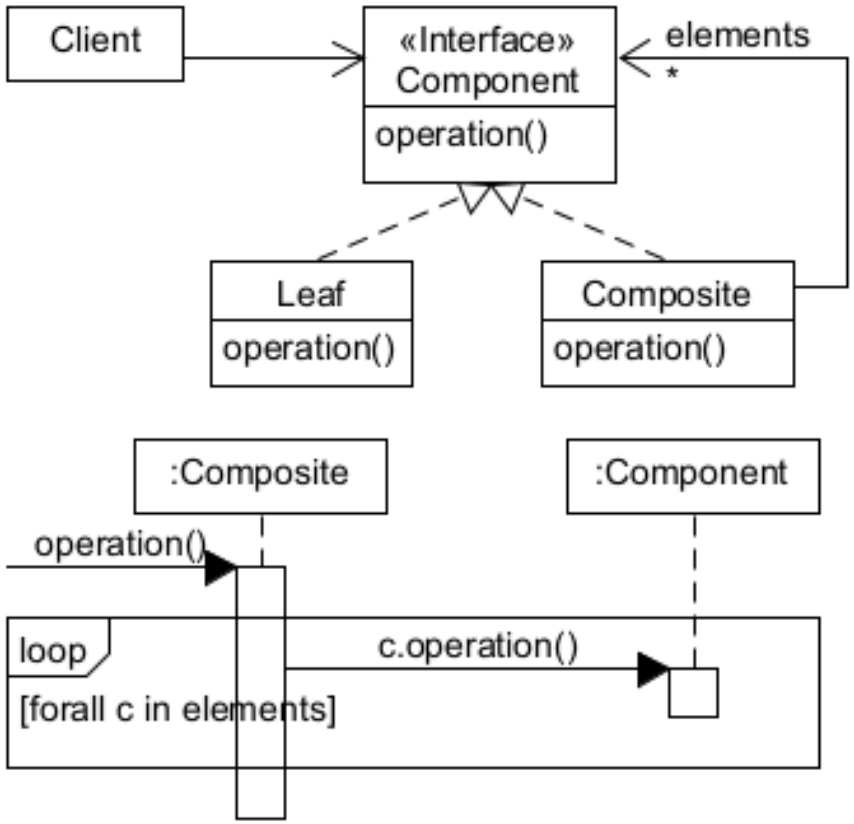

```java
public class Person {
    private String name;
    private String firstName;
    // getter and setter omitted
}
/**
 * Comparator<Person>, der zuerst den Vornamen und erst dann den Nachnamen berücksichtigt
 */
public class ComparatorFirstNameThenName implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        int nameComparison = p1.firstName.compareTo(p2.firstName);
        if (nameComparison != 0) {
            return nameComparison;
        }
        return p1.name.compareTo(p2.name);
    }
}
```

Strategy Beispiel: Comparator<Person> Varianten (2/2)


```java
/**
  * Comparator<Person>, der zuerst den Nachnamen und erst dann den Vornamen berücksichtigt
  */
public class ComparatorNameThenFirstName implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        int nameComparison = p1.name.compareTo(p2.name);
        if (nameComparison != 0) {
            return nameComparison;
        }
        return p1.firstName.compareTo(p2.firstName);
    }
}

public void sortExample(List<Person> personList) {
        Collections.sort(personList, new ComparatorFirstNameThenName());
}
```

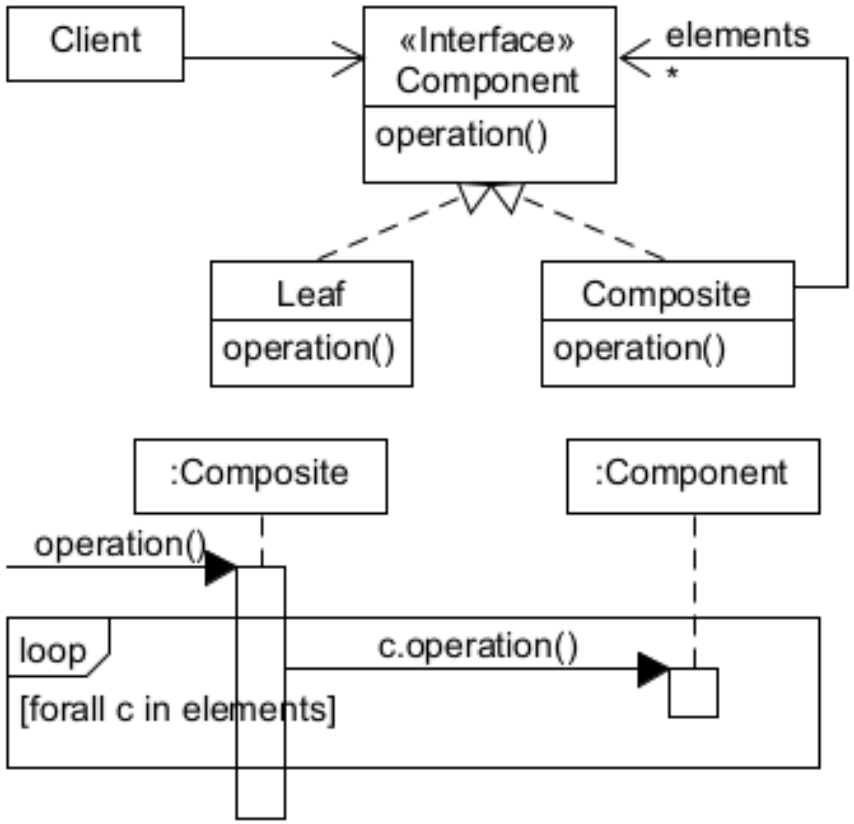
Composite: Problem und Lösung

- Problem:
  - Eine Menge von Objekten haben dasselbe Interface und müssen für viele Verantwortlichkeiten als Gesamtheit betrachtet werden.
- Lösung
  - Sie definieren ein Composite, das ebenfalls dasselbe Interface implementiert und Methoden an die darin enthaltenen Objekte weiterleitet.



Composite: Hinweise

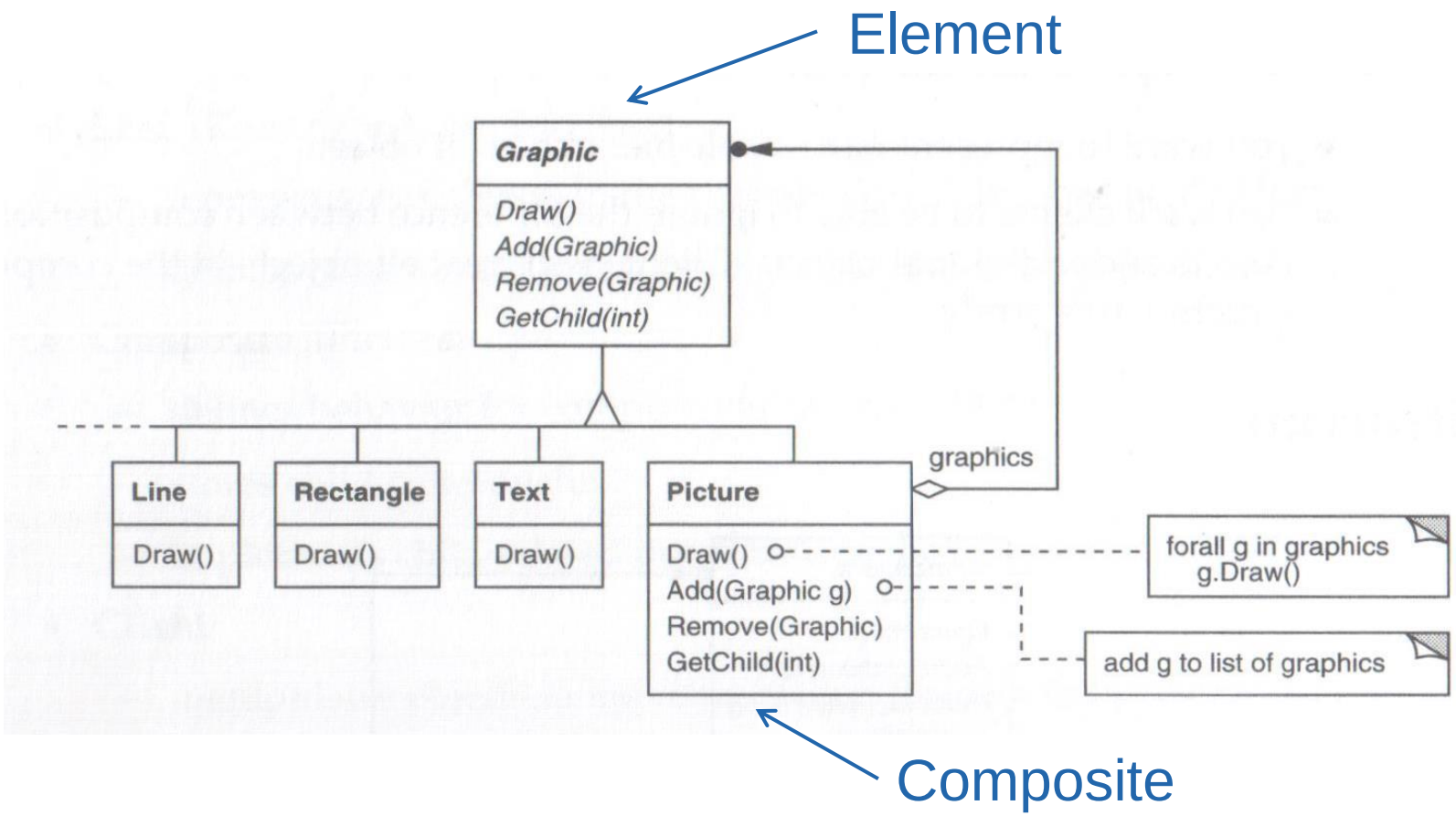
- Hinweise
  - Oft ist die hierarchische Struktur vom Fachgebiet her gegeben.
  - Nicht alle Methoden delegieren einfach auf die enthaltenen Elemente. Vor- und Nachbearbeitung ist üblich, und gewisse Methoden müssen ganz anders implementiert werden.



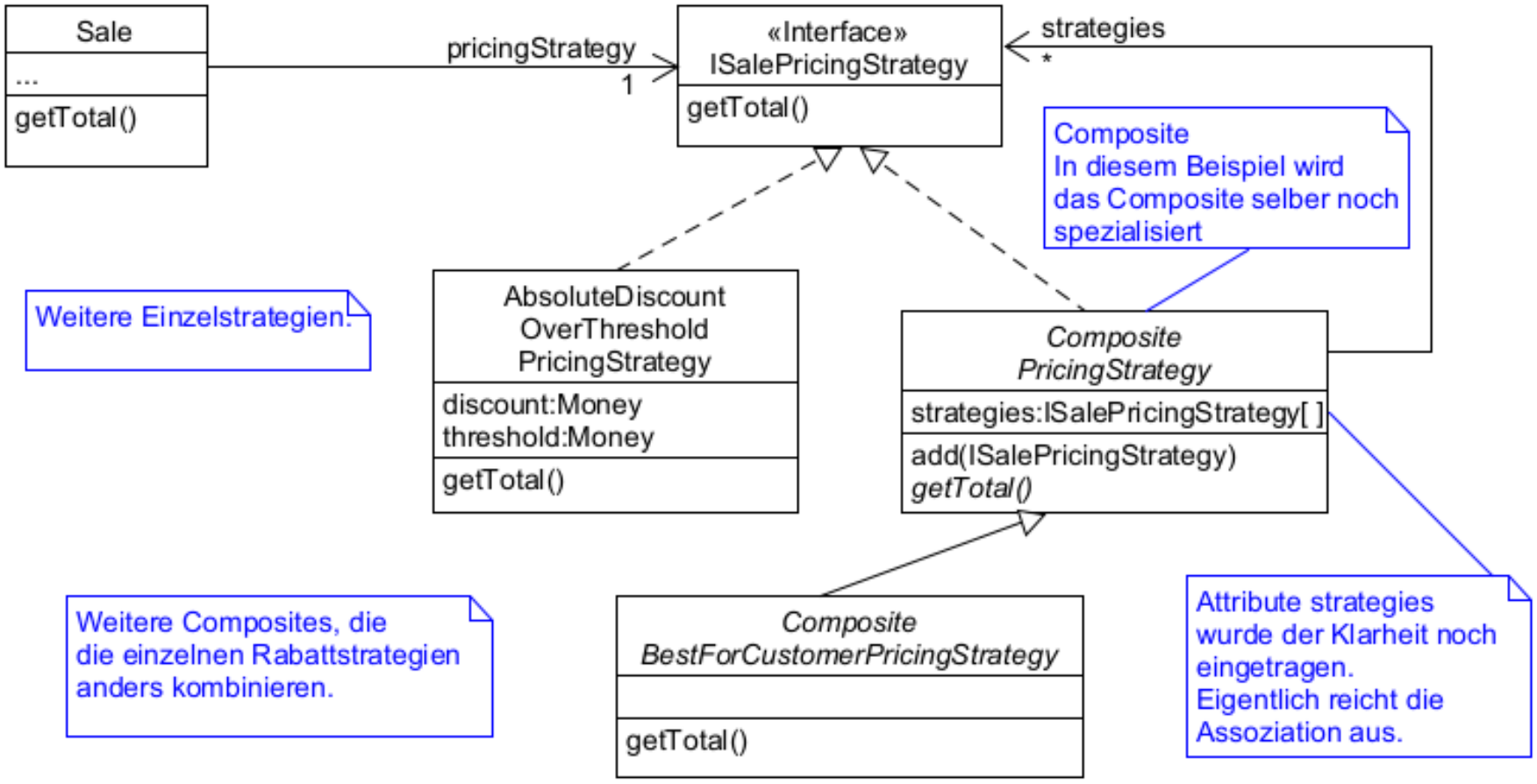
Composite: Beispiele

- JDK
  - Container von visuellen Komponenten in AWT, Swing und JavaFX.
- GoF Beispiel (siehe nachfolgende Folie)
  - Hierarchie von visuellen Komponenten.
- Larman, Point Of Sale Terminal (siehe nachfolgende Folie)
  - Die Rabattberechnung wird mit einer Strategy und mit dem Composite Pattern gelöst.
- Codebeispiel: Eine Gruppe von Lampen wird wie eine einzelne behandelt.

Composite: Beispiel GoF

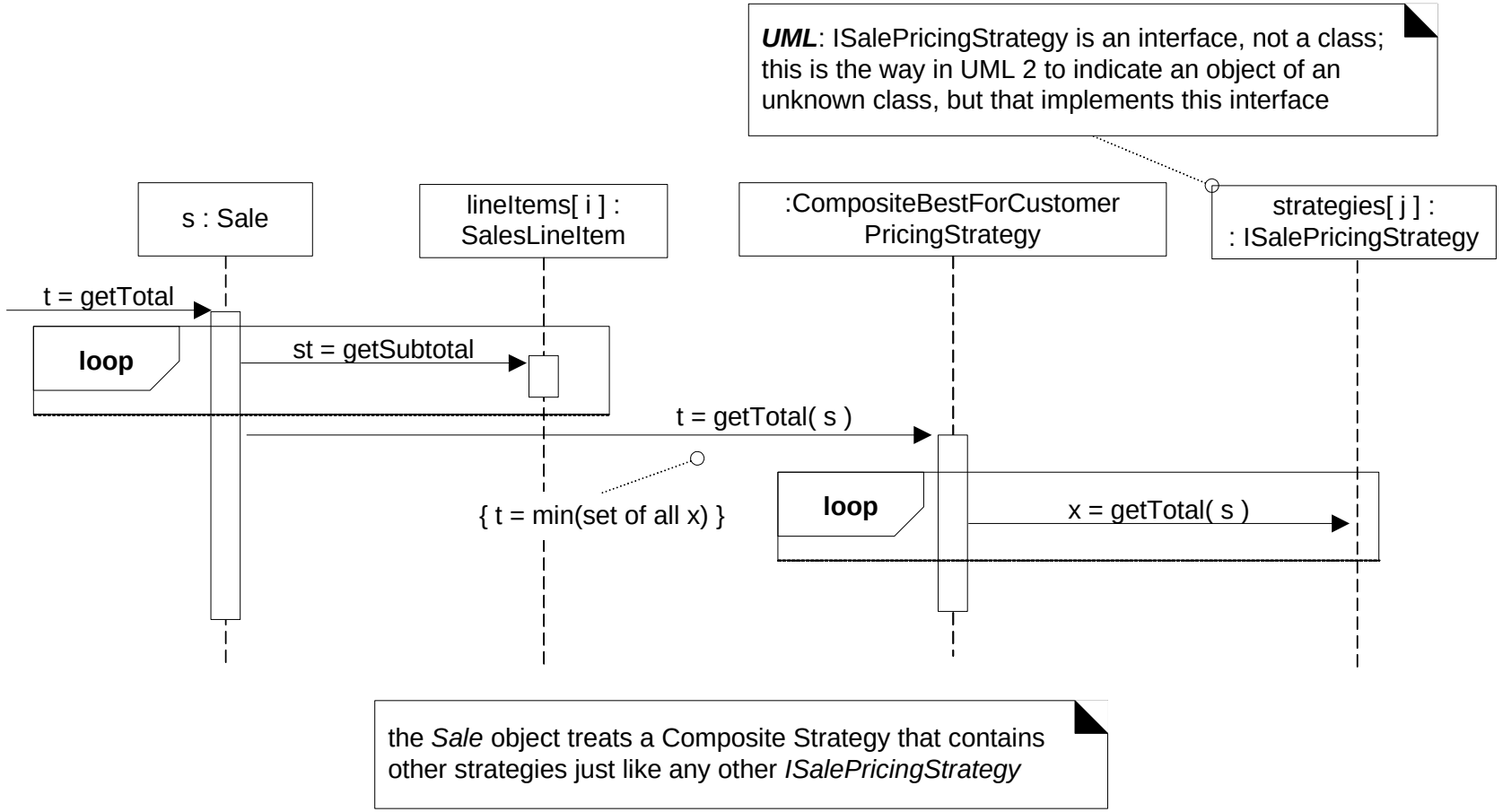


Composite: Beispiel POST DCD



Composite: Beispiel POST, Ablauf

Composite ruft alle enthaltenen Strategien auf und gibt dann den niedrigsten Wert als Resultat zurück.



Denkpause

Aufgabe 9.2 (5’)

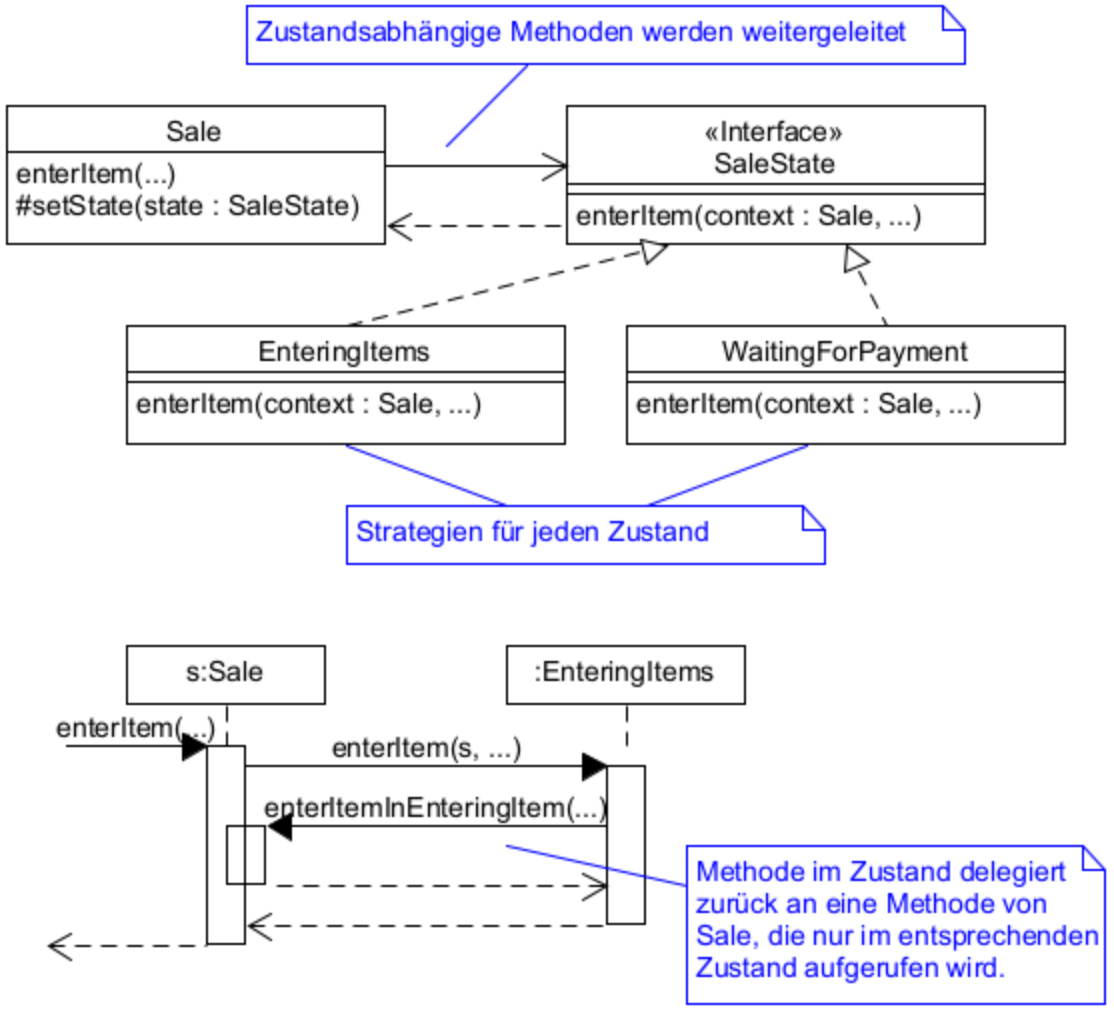
Diskutieren und bearbeiten Sie in Murmelgruppen folgende Fragen:

- Die bisherigen Composite Rabattberechnungs-Strategien wählen im Prinzip einfach eines der Resultate aus, welches eine die darin befindlichen Strategien berechnet haben.
- Sie möchten die Rabattberechnung nun so erweitern, dass Rabatte kumulativ angewendet werden. Was müssen Sie dafür ändern?

```
public interface PricingStrategy {
    BigDecimal getTotal(Sale sale);
}
```

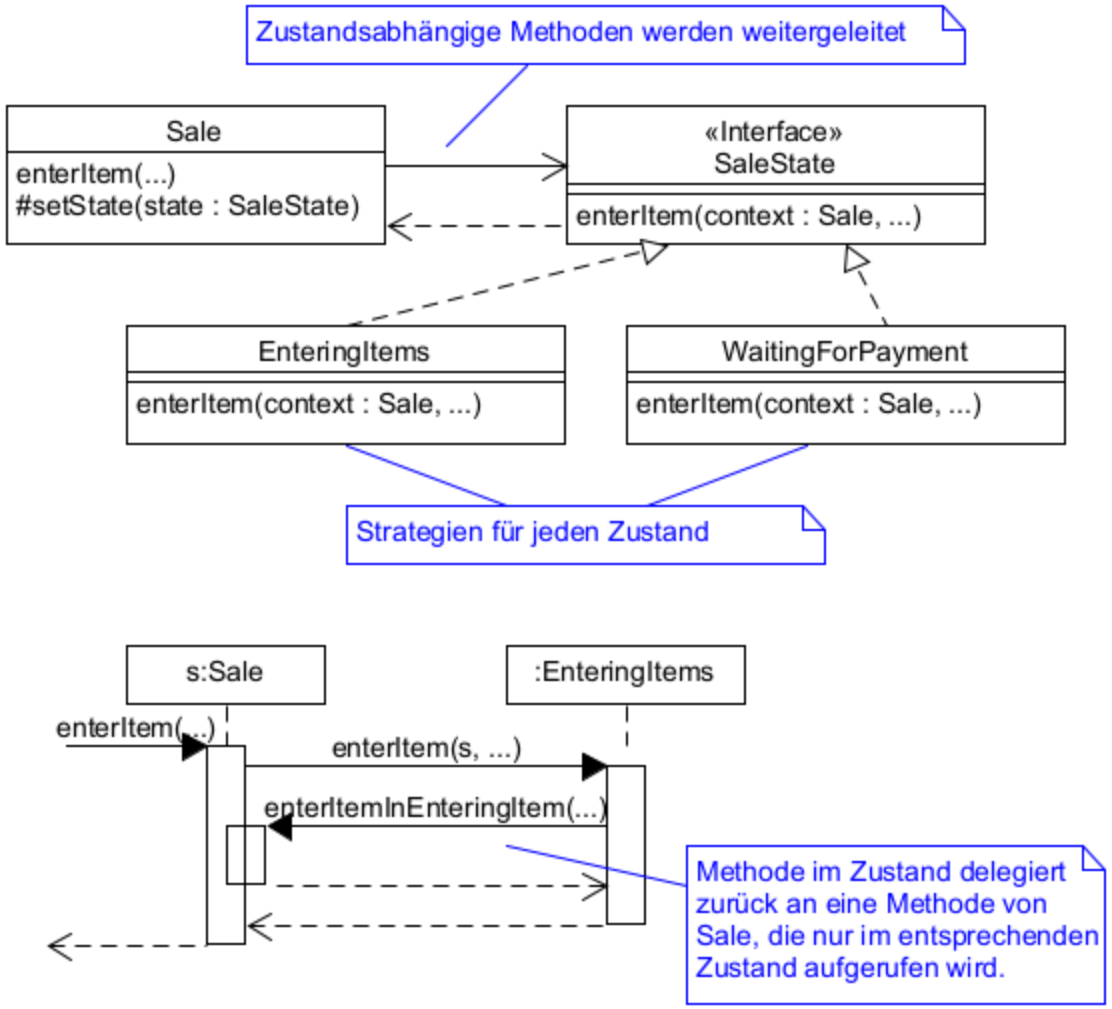
State: Problem und Lösung

- Problem:
  - Das Verhalten eines Objekts ist abhängig von seinem inneren Zustand.
- Lösung
  - Das Objekt hat ein darin enthaltenes Zustandsobjekt.
  - Alle Methoden, deren Verhalten vom Zustand abhängig sind, werden über das Zustandsobjekt geführt.



State: Hinweise

- Hinweise
  - Die Zustands-Klassen implementieren das Zustand-Interface.
  - Die Zustands-Objekte sind nichts anderes als Strategy Objekte und können Singletons sein.
  - Das Zustandsobjekt hat entweder direkt den Code (als innere Klasse) oder delegiert an eine Methode des Objekts weiter.



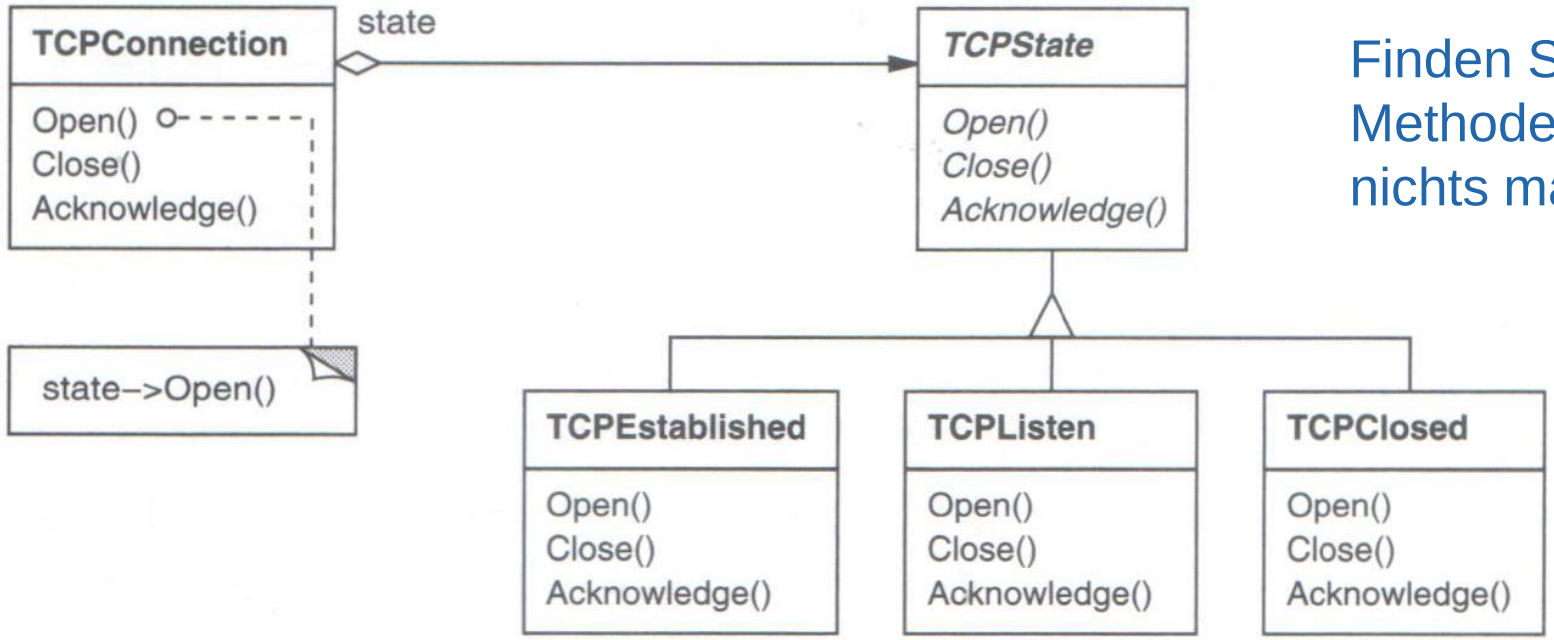
State: Beispiele (1/2)

- JDK
  - Obwohl zustandsabhängiges Verhalten noch relativ häufig vorkommt, wird das State-Pattern in der beschriebenen Form nirgends im JDK angewendet. Viel häufiger wird einfach auf ein boolean Attribut getestet und dann entsprechend gehandelt.
- GoF Beispiel
  - TCPConnection. Je nach Zustand werden Methodenaufrufe anders behandelt oder sind überhaupt erlaubt.

State: Beispiele (2/2)

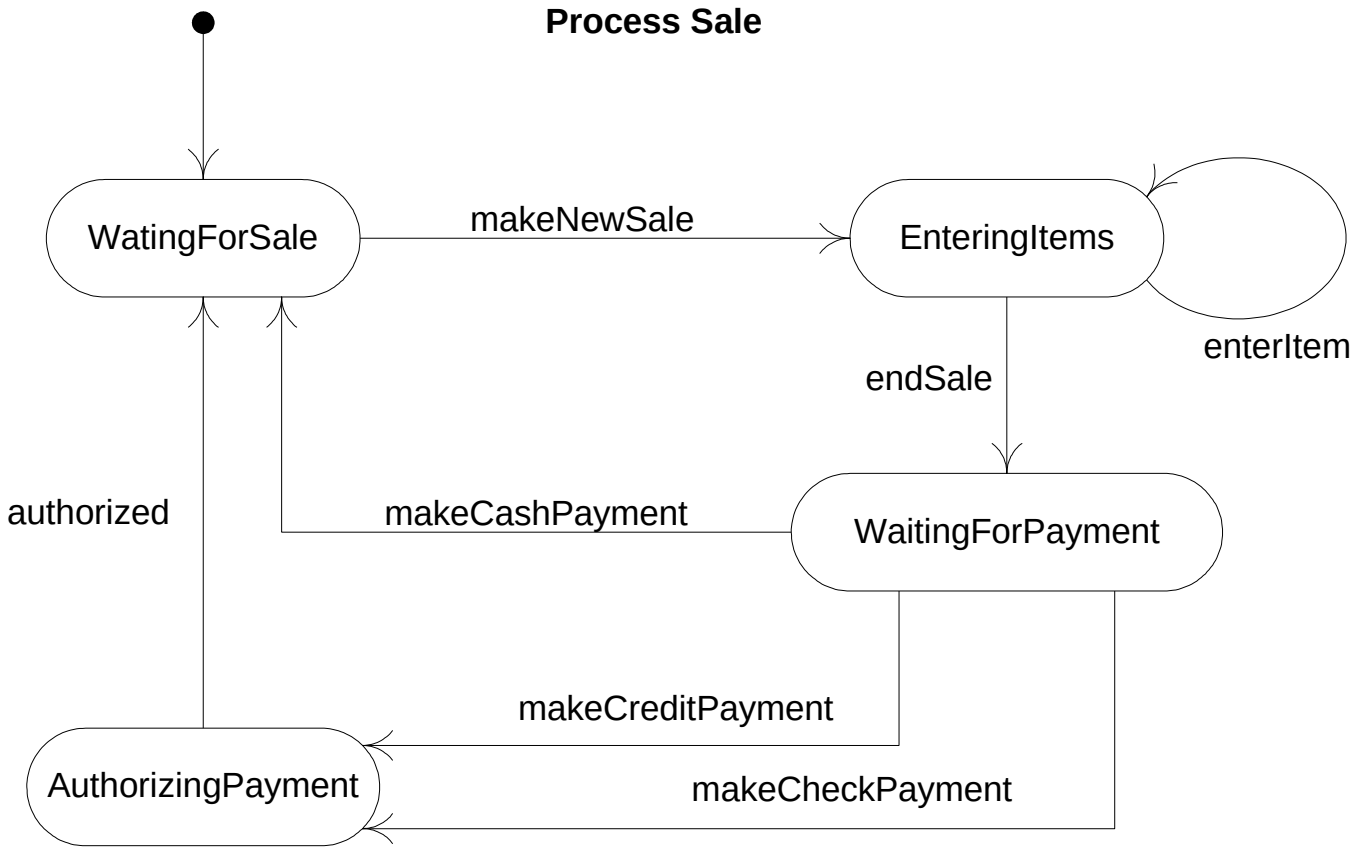
- Point Of Sale Terminal
  - Die Sale Klasse kann auch mit internen Zuständen modelliert werden. Es ist allerdings so, dass im Wesentlichen das Aufrufen von Methoden in gewissen Zuständen unterbunden werden muss. Daher wäre eine Implementation mit dem Abfragen des internen Zustands vermutlich einfacher (analog JDK).
- Allgemein
  - In Geschäftsanwendungen wird das State-Pattern eher selten eingesetzt, dafür sehr häufig in technischen Anwendungen, wie zum Beispiel Protokollhandler oder Maschinensteuerungen.
- Forum (inklusive Code)
  - Verschiedene Stufen der Änderungsmöglichkeiten.

State: Beispiel GoF



Finden Sie heraus, welche Methoden in welchen Zuständen nichts machen?

State: Beispiel POST Larman

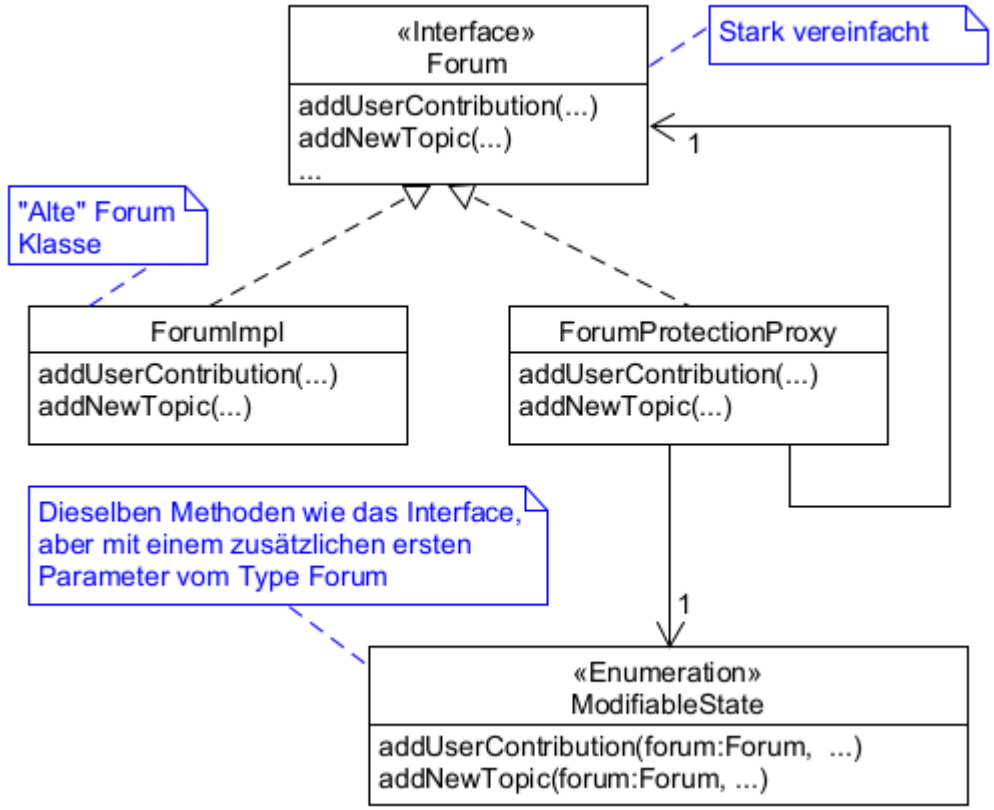


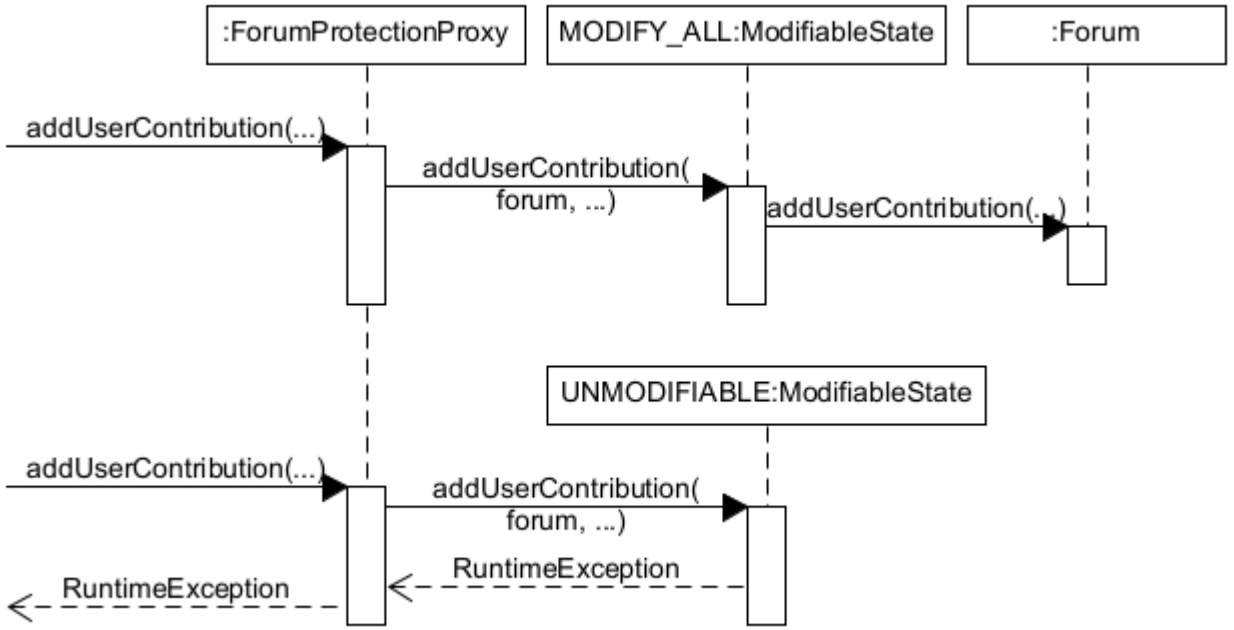
Finden Sie heraus, welche Methoden in welchen Zuständen nichts machen?

State: Beispiel ForumProtectionProxy

- Das Forum soll verschiedene Änderungs-Stufen erlauben:
  - Keine Änderungen sind erlaubt.
  - Es dürfen nur Beiträge hinzugefügt werden.
  - Es sind alle Änderungen erlaubt.
- Ähnlich wie beim Decorator wird hier ein Objekt (Protection Proxy, siehe LE08) vor das Forum geschaltet, das je nach Änderungs-Stufe Systemoperationen weiterleitet oder eine Exception wirft.
- Die Änderungsstufen werden mit einem Zustands-Objekt realisiert, über das die Systemoperationen geleitet werden.

State: Beispiel ForumProtectionProxy Entwurf





State: Beispiel ForumProtectionProxy Code (1)


```
public class ForumProtectionProxy implements Forum {
    private ModifiableState state = ModifiableState.MODIFY_ALL;
    private Forum forum;

    public ForumProtectionProxy(Forum forum) { . . . }

    @Override
    public void addNewTopic(String sessionId, String topicName, String description) {
        state.addNewTopic(forum, sessionId, topicName, description);
    }

    @Override
    public void addUserContribution(String sessionId, String topicName,
            String discussionName, String contribution) {
        state.addUserContribution(forum, sessionId, topicName, discussionName,
                contribution);
    }
}
```

State Pattern.
Behandlung
zustandsabhängig

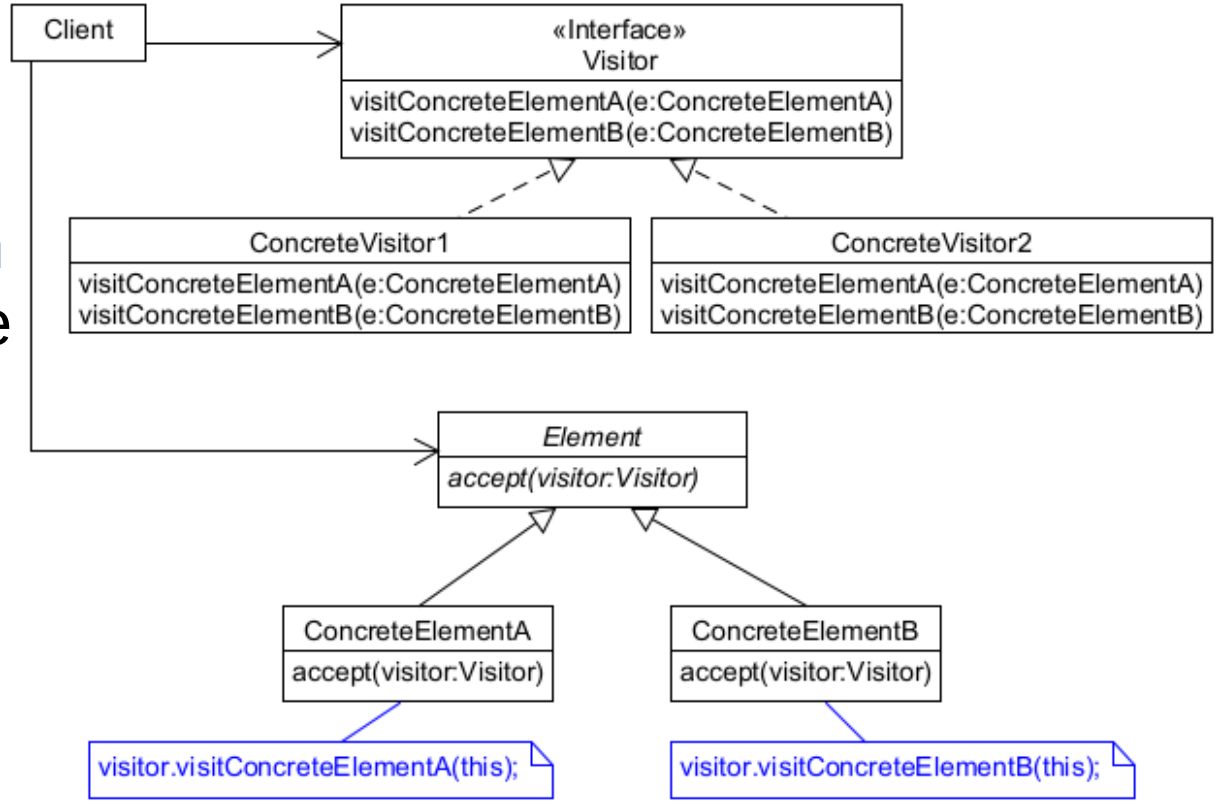
State: Beispiel ForumProtectionProxy Code (2)


```
public enum ModifiableState {
    MODIFY_ALL {
        @Override
        public void addNewTopic(Forum forum, String sessionId, String topicName, String description) {
            forum.addNewTopic(sessionId, topicName, description);
        }
    },
    ADD_CONTRIBUTION {
        @Override
        public void addNewTopic(Forum forum, String sessionId, String topicName, String description) {
            throw new RuntimeException("Add new topic is not allowed");
        }
    },
    UNMODIFIABLE {
        @Override
        public void addNewTopic(Forum forum, String sessionId, String topicName, String description) {
            throw new RuntimeException("Add new topic is not allowed");
        }
    };

    public abstract void addNewTopic(Forum forum, String sessionId, String topicName, String description);

    public abstract void addUserContribution(Forum forum, String sessionId, String topicName, …);
}
```

Methode „addUserContribution“ weggelassen in allen enum Instanzen

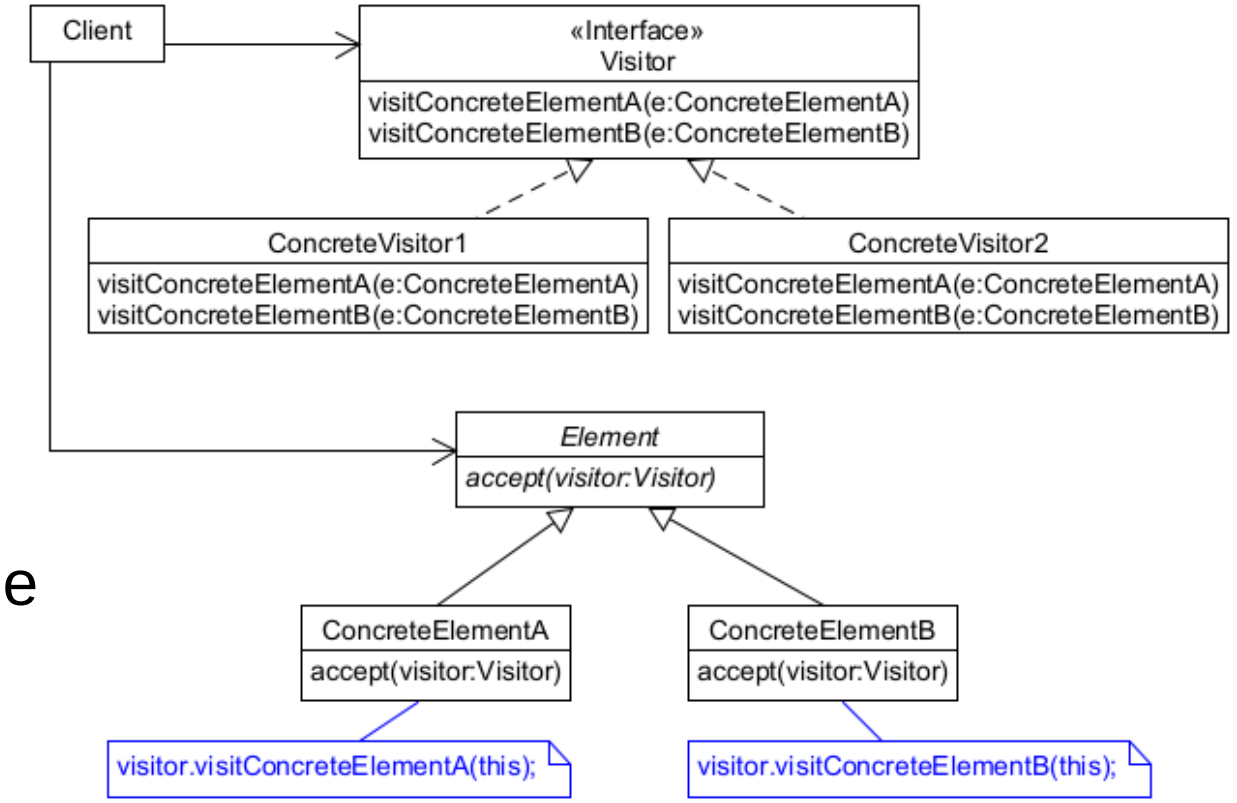
Visitor: Problem und Lösung

- Problem:
  - Eine Klassenhierarchie soll um (weniger wichtige) Verantwortlichkeiten erweitert werden, ohne dass viele neue Methoden hinzukommen.
- Lösung
  - Die Klassenhierarchie wird mit einer Visitor-Infrastruktur erweitert. Alle weiteren neuen Verantwortlichkeiten werden dann mit spezifischen Visitor-Klassen realisiert.



Visitor: Hinweise

- Hinweise
  - Widerspruch zum Information Expert. Daher wichtige Methoden weiterhin direkt der Klasse hinzufügen.
  - Oft werden Auswertungen an Visitor-Klassen delegiert.
  - Bei einer mehrstufigen Objekthierarchie stellt sich die Frage, wer die darin enthaltenen Elemente aufruft. Siehe Beispiel Schachprogramm.



Visitor: Beispiele (1/2)

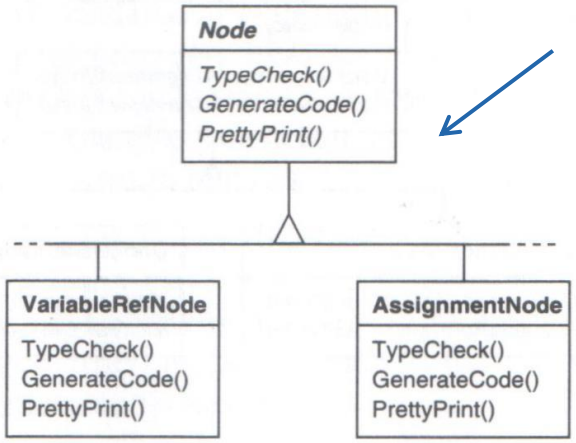
- JDK
  - java.nio.file.Files.walkFileTree (…, FileVisitor visitor)
- GoF Beispiel
  - Ein Beispiel aus dem Compilerbau: Der abstrakte Syntax Tree (AST) hat eine Visitor-Infrastruktur, die von einem Visitor für Typ-Kontrolle oder Code-Generation benutzt wird.

Visitor: Beispiele (2/2)

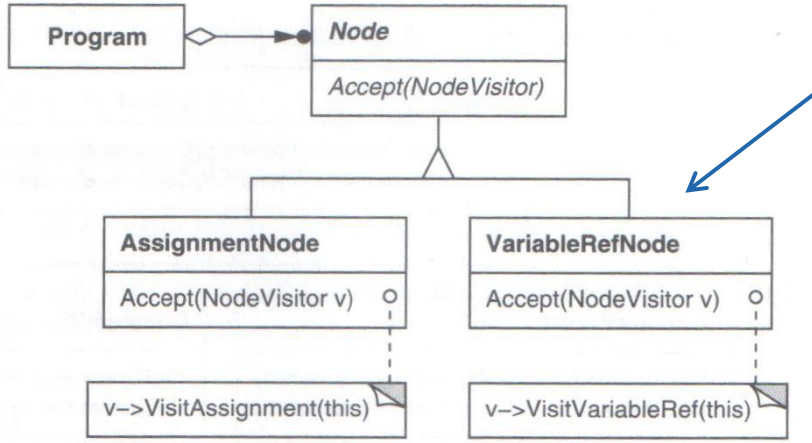
- Antlr
  - Antlr ist ein Parser-Generator, der aus einer formalen Syntax einen Parser mitsamt den Abstract Syntax Tree Nodes erzeugt. Dabei wird auch eine Visitor-Infrastruktur erzeugt für diese Nodes.
- Anwendung, wo die Daten der Domänenlogik direkt visualisiert werden, wie zum Beispiel Brettspiele, 2D/3D Echtzeitspiele, Simulationen
  - Der Teil der Domänenlogik, der auf dem Bildschirm dargestellt wird, stellt eine Visitor-Infrastruktur zur Verfügung. Der Teil des UI, der die Daten der Domänenlogik visualisiert, benutzt einen Visitor, um diese darzustellen.

Visitor: Beispiel GoF

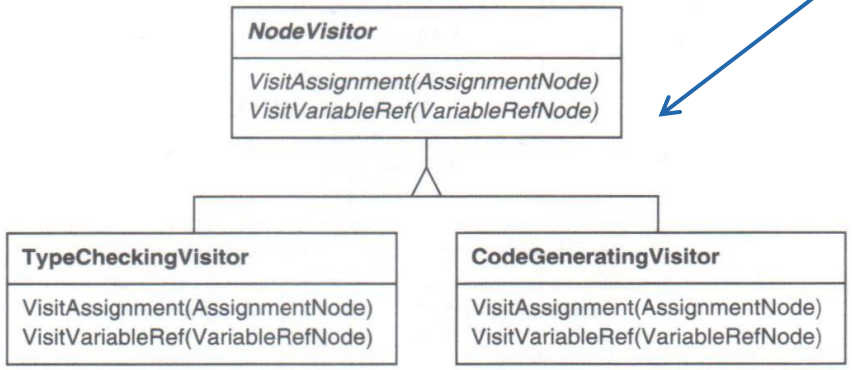
Ausgangslage ohne Visitor-Pattern



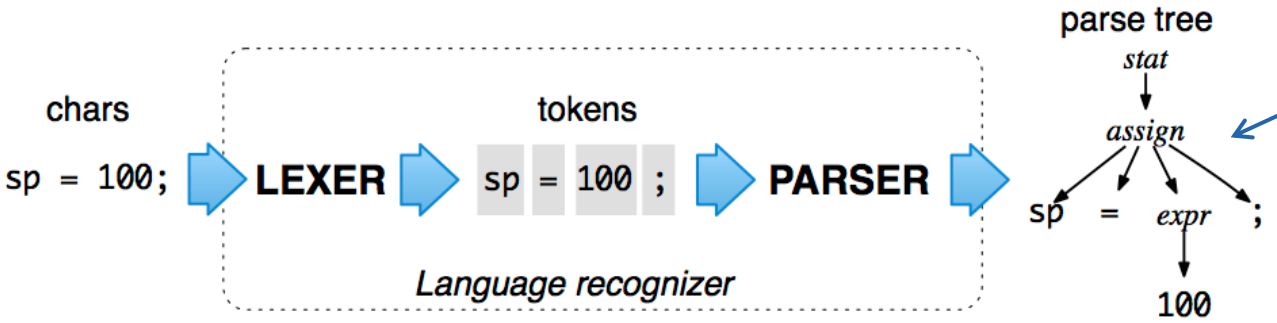
Visitor Infrastruktur



Visitors



Visitor: Beispiel Antlr (Parser Generator)



Abstract Syntax Tree (AST)
Klassen werden von Antlr Tools erzeugt, während die Objekte dann vom Parser basierend auf dem Programmcode erzeugt werden. Die Visitor-Infrastruktur wird ebenfalls automatisch erzeugt.

```
public interface JavaListener extends ParseTreeListener<Token> {
  void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx);
  void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx);
  void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx);
 ...
}
```

Visitor, der alle Nodes des AST hierarchisch besucht.

Visitor: Beispiel Schachprogramm, Figuren zeichnen (1)

- Für das UI eines Schachprogramms ist es die zentrale Aufgabe, die Figuren auf dem Spielfeld zu zeichnen
- Gemäss GRASP IE wäre eigentlich die spezifische Spielfigur (z.B. King) dafür zuständig. Aus Architekturgründen (Trennung UI – Domäne) kommt dies aber nicht in Frage.
- Als Alternative kann daher ein Visitor eingesetzt werden.

Visitor: Beispiel Schachprogramm, Figuren zeichnen (2)

- Piece Visitor Infrastruktur:
  - Visitor Interface
  - Aufruf des Visitors in den zu besuchenden Klassen

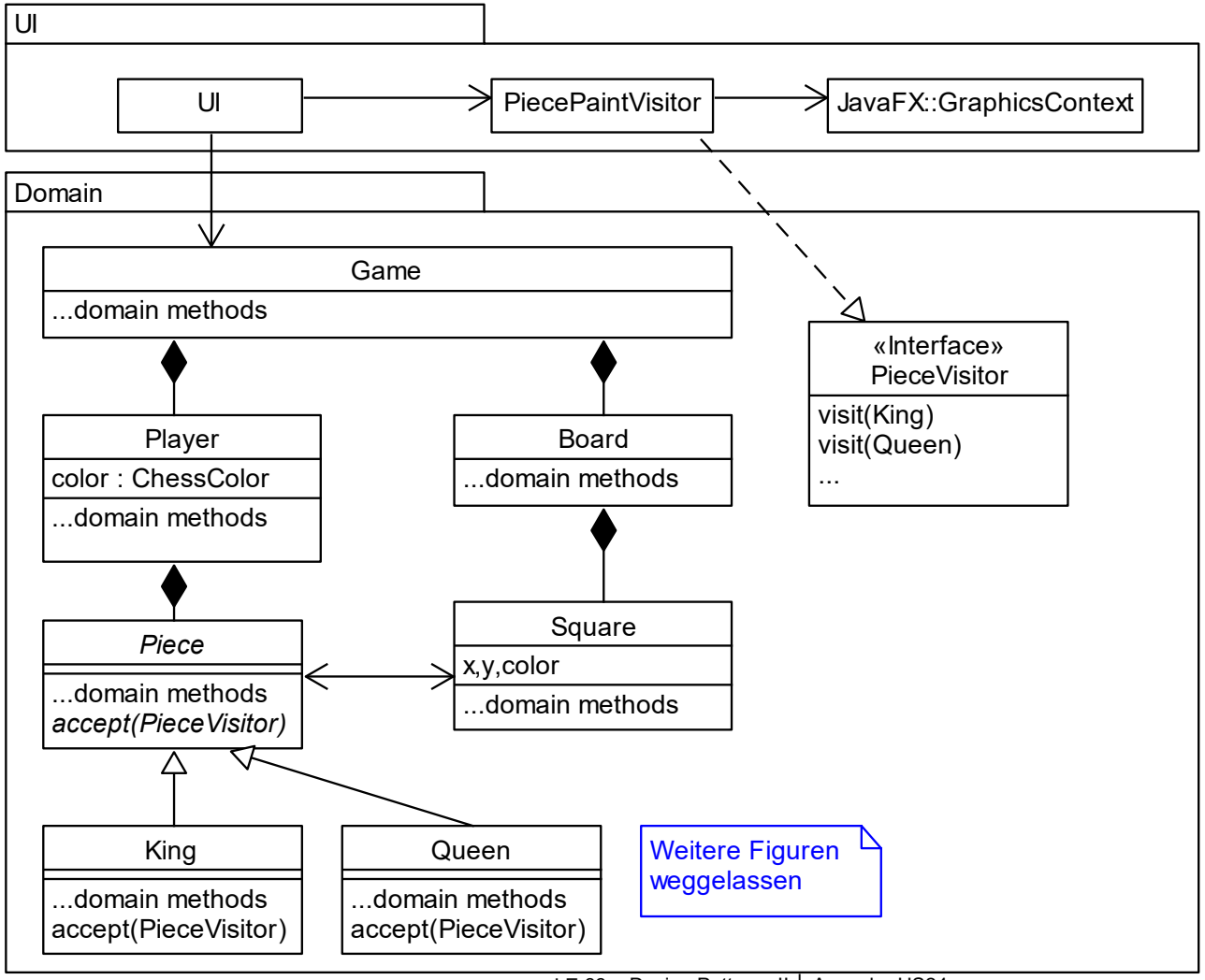
```
public interface PieceVisitor {
    void visit(King king);
    void visit(Queen queen);
    // weitere Methoden weggelassen
}

public class King extends Piece {
    public void accept(PieceVisitor visitor) {
        visitor.visit(this);
    }
}
```

Visitor: Beispiel Schachprogramm, Figuren zeichnen (3)

- Piece Visitor Implementation, um die Figuren über JavaFX GraphicsContext zu zeichnen

```
public class PiecePaintVisitor implements PieceVisitor {
    private GraphicsContext graphics;

    public PiecePaintVisitor(GraphicsContext graphics) {. . .}

    @Override
    public void visit(King king) {
        // draw and fill king symbol with the correct color.
        // if required draw contour of piece in opposite color.
        // use graphics instance variable
     }

    @Override
    public void visit(Queen queen) { . . . }
}
```

Visitor: Beispiel Schachprogramm, Figuren zeichnen (4)



Visitor: Beispiel Schachprogramm, Brett zeichnen (1)

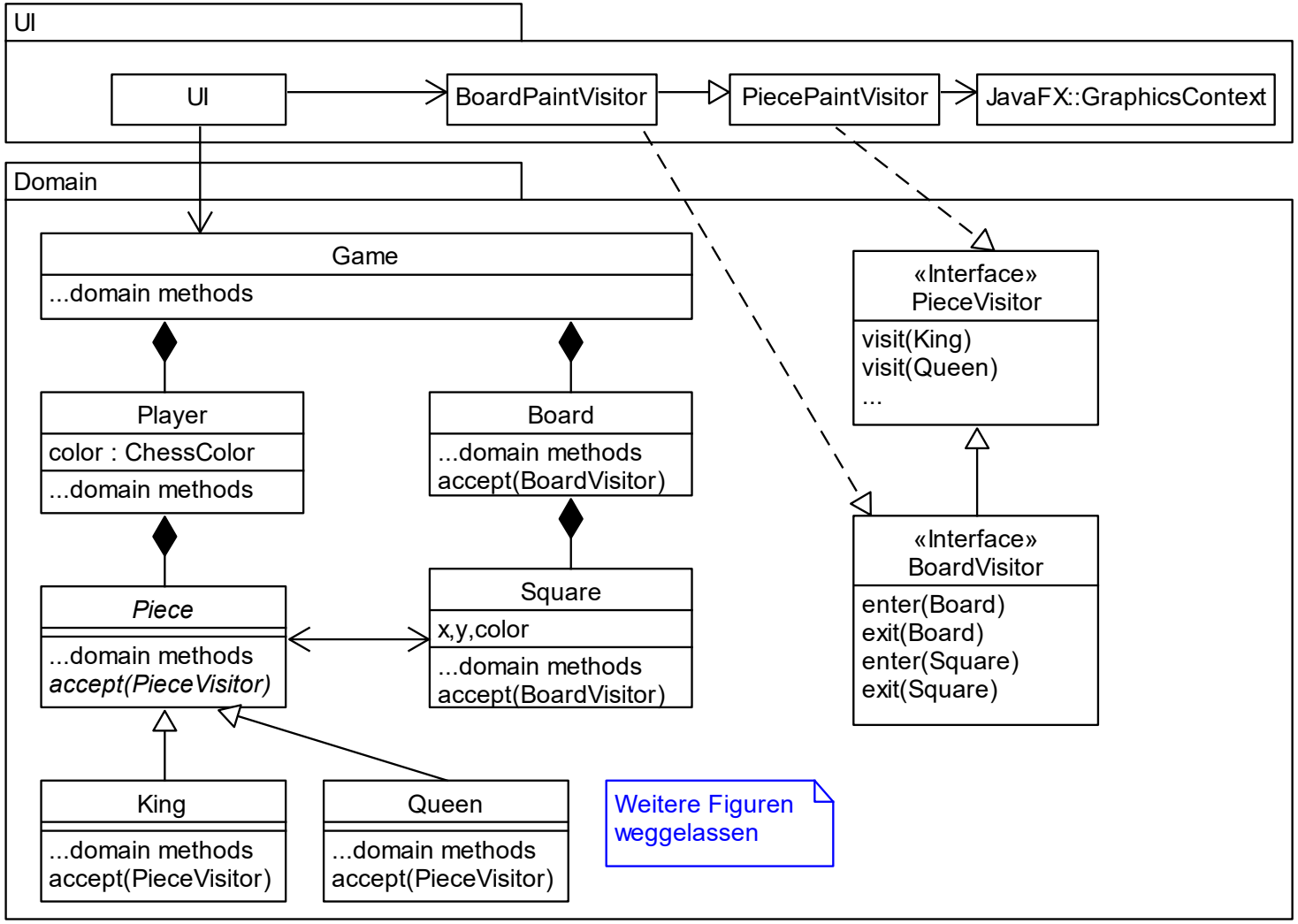
- Mit den Figuren alleine ist das Spielfeld aber noch nicht gezeichnet.
- Es müssen noch die leeren Spielfelder gezeichnet werden.
- Gemäss GRASP IE wären auch hier die Klassen Board und Square dafür zuständig, aber aus Architekturgründen (Trennung UI – Domäne) kommt dies nicht in Frage und wir setzen dafür auch einen Visitor ein.
- In diesem Fall kann ein hierarchischer Visitor für Board entworfen werden, der alle darin enthaltenen Felder besucht und von jedem Feld an die Spielfigur weitergeleitet wird, wenn es auf dem Feld eine hat.
- Der hierarchische Visitor hat anstelle der «visit» Methode je eine «enter» und eine «exit» Methode.

Visitor: Beispiel Schachprogramm, Brett zeichnen (2)

- Board Visitor Infrastruktur.
  - Der Board Visitor besucht auch enthaltene Elemente
  - Die enter Methode wird vor dem Besuch der Elemente aufgerufen
  - Die exit Methode wird nach dem Besuch der Elemente aufgerufen

```
public interface BoardVisitor extends PieceVisitor{
    void enter(Board board);
    void exit(Board board);
    void enter(Square square);
    void exit(Square square);
}

public class Square { // Attributes, constructor and other methods omitted.
    public void accept(BoardVisitor visitor) {
        visitor.enter(this);
        if (piece != null) piece.accept(visitor);
        visitor.exit(this);
    }
}
```

Visitor: Beispiel Schachprogramm, Brett zeichnen (3)



Visitor: Beispiel Schachprogramm, Brett zeichnen (4)

- Wieso werden 2 Visitor Interfaces definiert?
- Es wäre auch möglich, PieceVisitor zu BoardVisitor umzubenennen und dort die neuen Methoden hinzuzufügen.
- Der Nachteil dieser Lösung liegt darin, dass dann die Piece Klasse und alle davon abgeleiteten Klassen einen Visitor als Parameter haben, der Methoden enthält, die für Piece keinen Sinn machen.
- Dies ist gemäss dem «Interface-Segregation-Prinzip» der SOLID Prinzipien nicht zu empfehlen.

Für Interessierte: Artikel zu SOLID.

Denkpause

Aufgabe 9.3 (5’)

Diskutieren und bearbeiten Sie in Murmelgruppen folgende Fragen zum Beispiel Schachprogramm:

- Skizzieren Sie mit Java-/Pseudocode den Inhalt der Methode Board.accept(BoardVisitor) in Analogie zur Methode Square.accept(BoardVisitor)
- Was sind die Vor- und Nachteile von Visitor wie z.B. BoardVisitor, die einen Behälter von Objekten besuchen und darin die einzelnen Elemente?
- Wieso gibt es für die Behälter Klassen (Square, Board) jeweils eine enter- und eine exit Methode?

Denkpause

Aufgabe 9.3 – Musterlösung

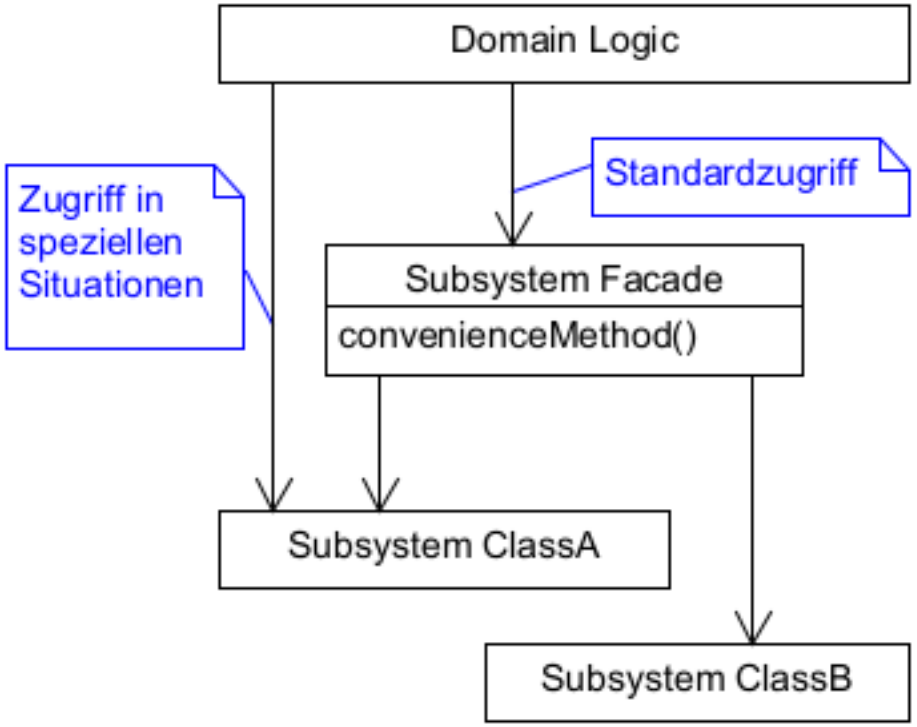
- Code siehe unten.
- Vorteil dieser Art von Visitor: Die Iteration über die Elemente muss nicht selber programmiert werden. Siehe auch die Stream Klassen und die Iterable.forEach(…) Methode.
- Nachteil: Es werden immer alle Elemente besucht, auch wenn es nicht notwendig wäre, weil zum Beispiel das gesucht Element gefunden wurde. Eine Lösung dafür wären Ergänzungen im Visitor, die angeben, ob der Visitor seine Aufgabe bereits erfüllt hat oder nicht.
- Die enter- und exit Methoden
  - dienen zuerst dazu, behälterspezifischen Code auszuführen (z.B. das Zeichnen der Brett Umrandung)
  - Vorbereitungen für die Elemente zu machen (z.B. kann Square in der enter Method die aktuelle Position im GraphicsContext setzen)
  - Nachbearbeitung zu machen (z.B. die aktuell selektierte Figur hervorzuheben)

```
public void accept(BoardVisitor visitor) {
    visitor.enter(this);
    visitSquares(visitor);
    visitor.exit(this);
}
```

```
protected void visitSquares(BoardVisitor visitor) {
    for(int row = 0; row < 8; row++) {
        for(int column = 0; column < 8; column++) {
            Square square = squares[row][column];
            square.accept(visitor);
        }
    }
}
```

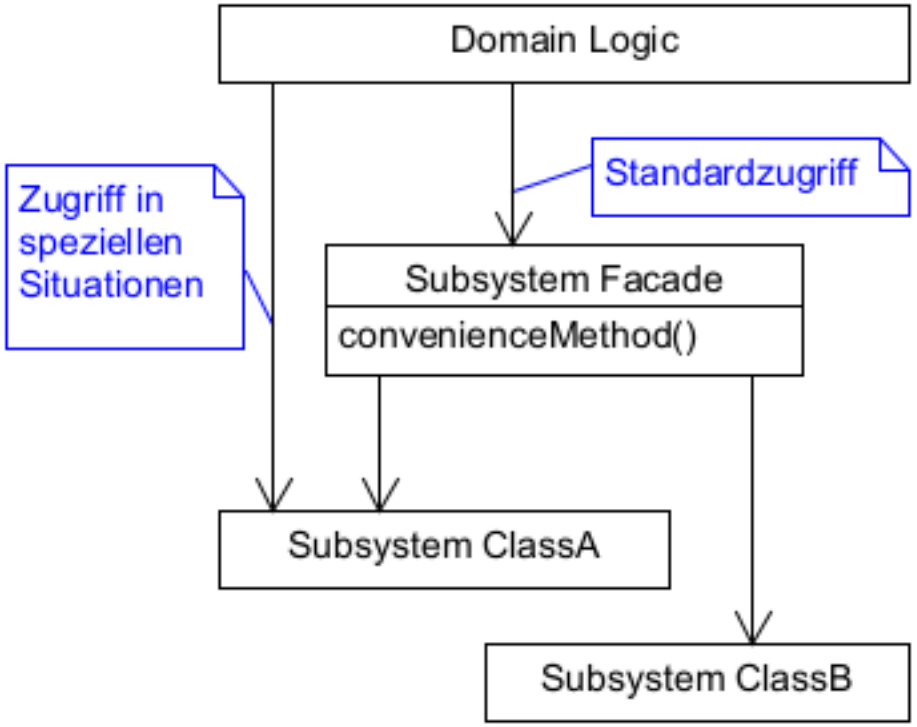
Facade: Problem und Lösung

- Problem:
  - Sie setzen ein ziemlich kompliziertes Subsystem mit vielen Klassen ein. Wie können Sie seine Verwendung so vereinfachen, dass alle Team-Mitglieder es korrekt und einfach verwenden können?
- Lösung
  - Eine Facade (Fassade) Klasse wird definiert, welche eine vereinfachte Schnittstelle zum Subsystem anbietet und die meisten Anwendungen abdeckt.



Facade: Hinweise

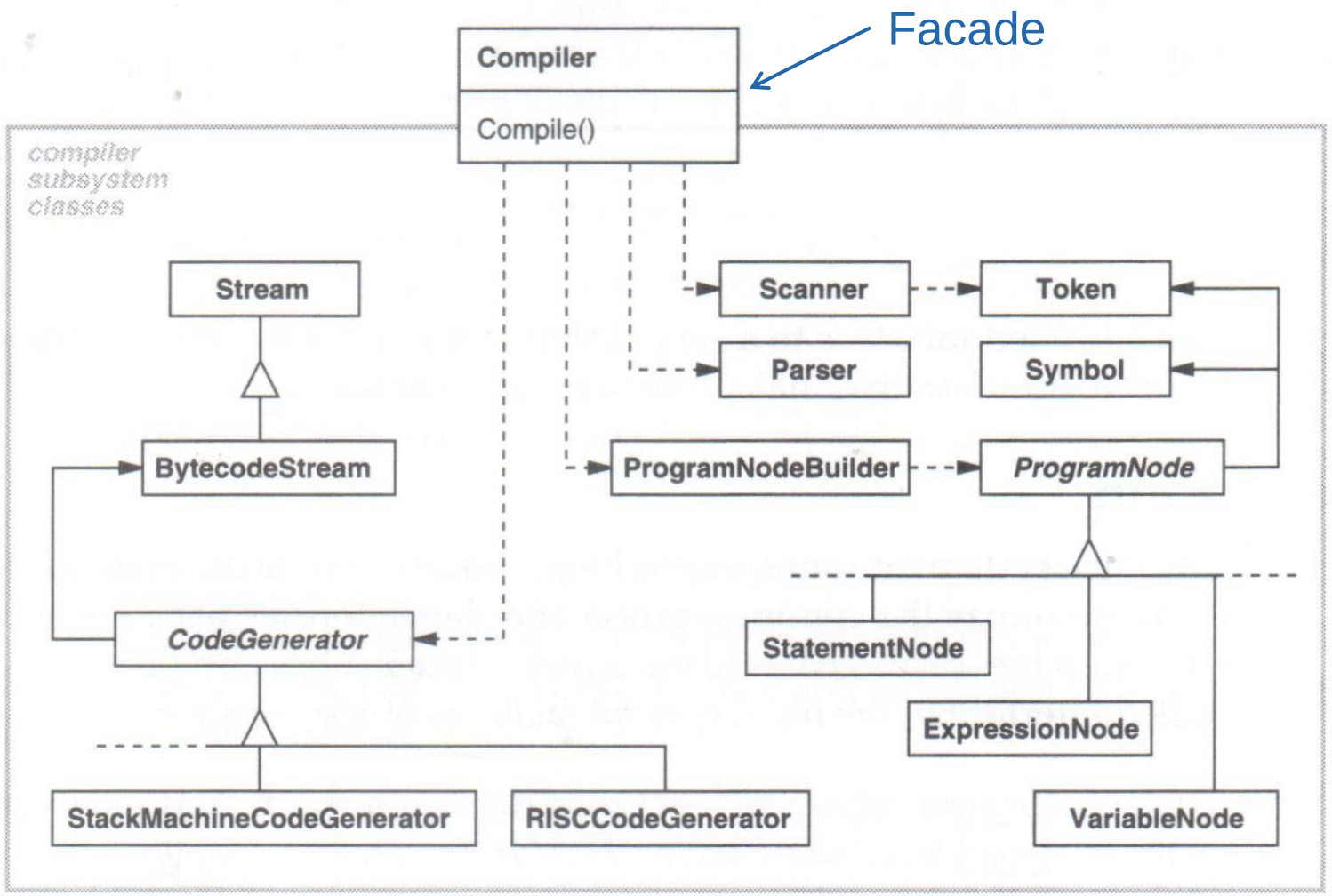
- Hinweise
  - Eine Facade kapselt, im Gegensatz zum Adapter, ein Subsystem nicht vollständig ab. Es ist erlaubt, dass die Methoden der Facade Parameter und Rückgabewerte haben, die Bezug auf das Subsystem nehmen.
  - Wird oft vom Ersteller eines Frameworks entwickelt.



Facade: Beispiele

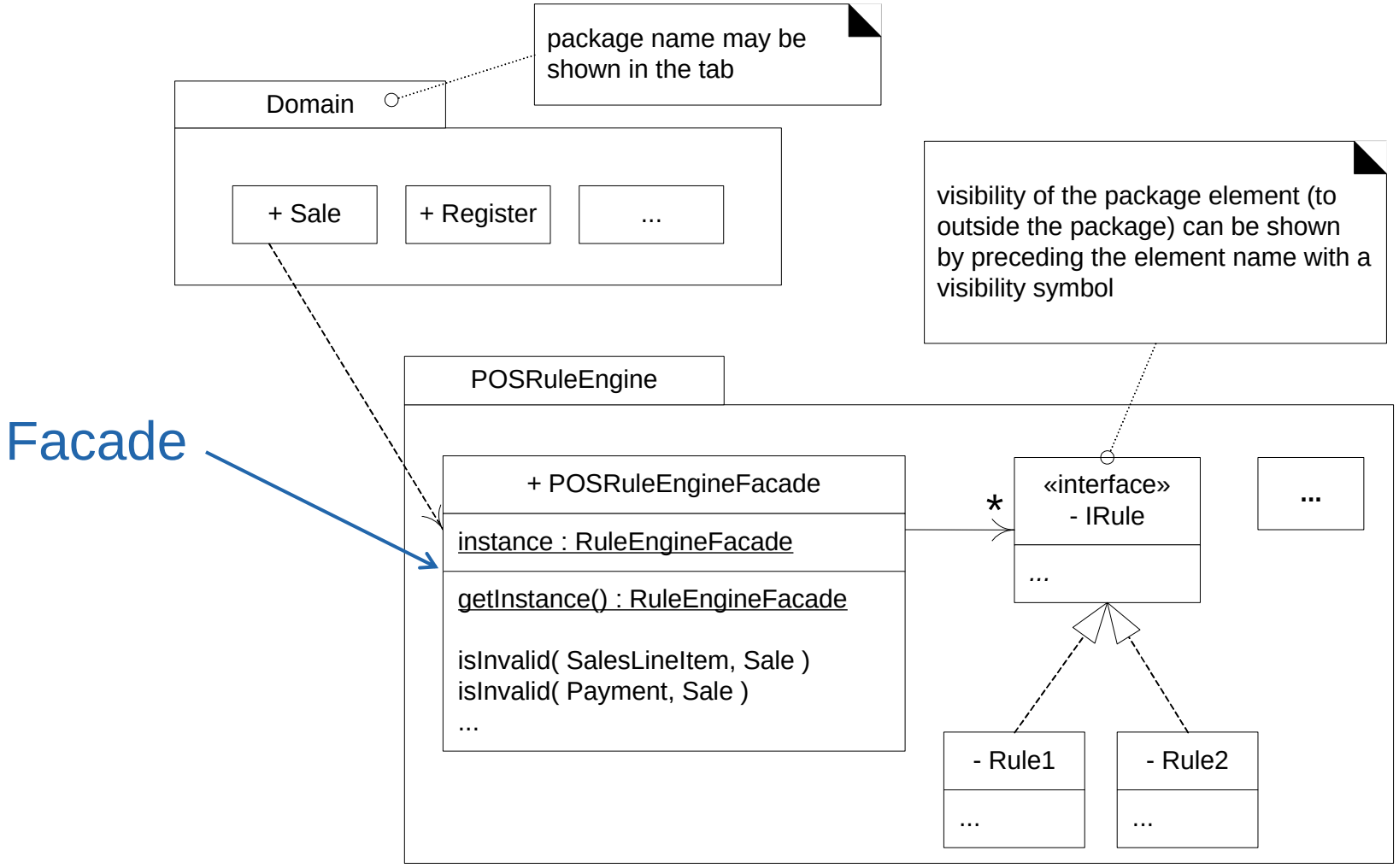
- JDK
  - javax.imageio.ImageIO besitzt Methoden, die direkt von einem Bildformat in ein anderes konvertieren.
- GoF Beispiel
  - Die Compiler Klasse bildet einen einfachen Einstiegspunkt in die Verwendung des Compiler Subsystems an.
- Point Of Sale Terminal
  - Eine Facade wird eingesetzt, um die Verwendung eines externen Regel-Frameworks zu vereinfachen.

Facade: Beispiele GoF



Andere Programme wie z.B. für eine Quelltextanalyse können trotzdem auf alle Elemente des Subsystems zugreifen

Facade: Beispiel POST Larman



Agenda

1. Repetition Aufbau von Design Patterns
2. Design Patterns
3. **Wrap-up und Ausblick**

Wrap-up

- Ein Decorator erweitert die Funktionalität eines Objekts (im Gegensatz zu Vererbung)
- Ein Observer beobachtet das Observable. Da der Observer dem Observable nur als Interface bekannt ist, wird Low Coupling unterstützt.
- Eine Strategy ist ein Klasse, die genau einen Algorithmus enthält. Über Polymorphismus kann dann die Strategy einfach ausgetauscht werden.
- Ein Composite beinhaltet Objekte, die dasselbe Interface wie das Composite implementieren. Viele Methoden werden dann auf diese Objekte weitergeleitet.
- Zustandsabhängiges Verhalten wird über ein State Objekt geleitet.
- Ein Visitor besucht Objekte, die dann die richtige Methode auf dem Visitor aufrufen.
- Ein Facade bietet für ein Teilsystem eine vereinfachte Benutzung an.

Ausblick

- In der nächsten Lerneinheit werden wir:
  - Einen Quiz zum Design durchführen.
  - Den Prozess des Refactorings genauer anschauen.

Quellenverzeichnis

[1] Larman, C.: UML 2 und Patterns angewendet, mitp Professional, 2005

[2] Seidel, M. et al.: UML @ Classroom: Eine Einführung in die objektorientierte Modellierung, dpunkt.verlag, 2012

[3] Martin, R. C.: Clean Architecture: A Craftsman's Guide to Software Structure and Design, mitp Professional, 2018

[4] Gamma, E et al.: Design Patterns: Elements of Reusable Object-Oriented Software Addison Wesley Longman, 1995

[5] McDonald, J: DZone Refcardz: Design Patterns, www.dzone.com, 2008