

Design Patterns

Wozu?

- ❖ bewährte Lösungswege für wiederkehrende Designprobleme in der Softwareentwicklung. Sie beschreiben die essenziellen Entwurfsentscheidungen (Klassen- und Objektarrangements). Durch den Einsatz von Design Pattern wird ein Entwurf flexibel, wiederverwendbar, erweiterbar, einfacher zu verwenden und änderungsstabil.

Verschiedene Muster

- ❖ Verhaltensmuster
 - ❖ Strategy, Observer, State
- ❖ Strukturmuster
 - ❖ Abstract Factory, Builder
- ❖ Erzeugungsmuster
 - ❖ Dekorator, Composite

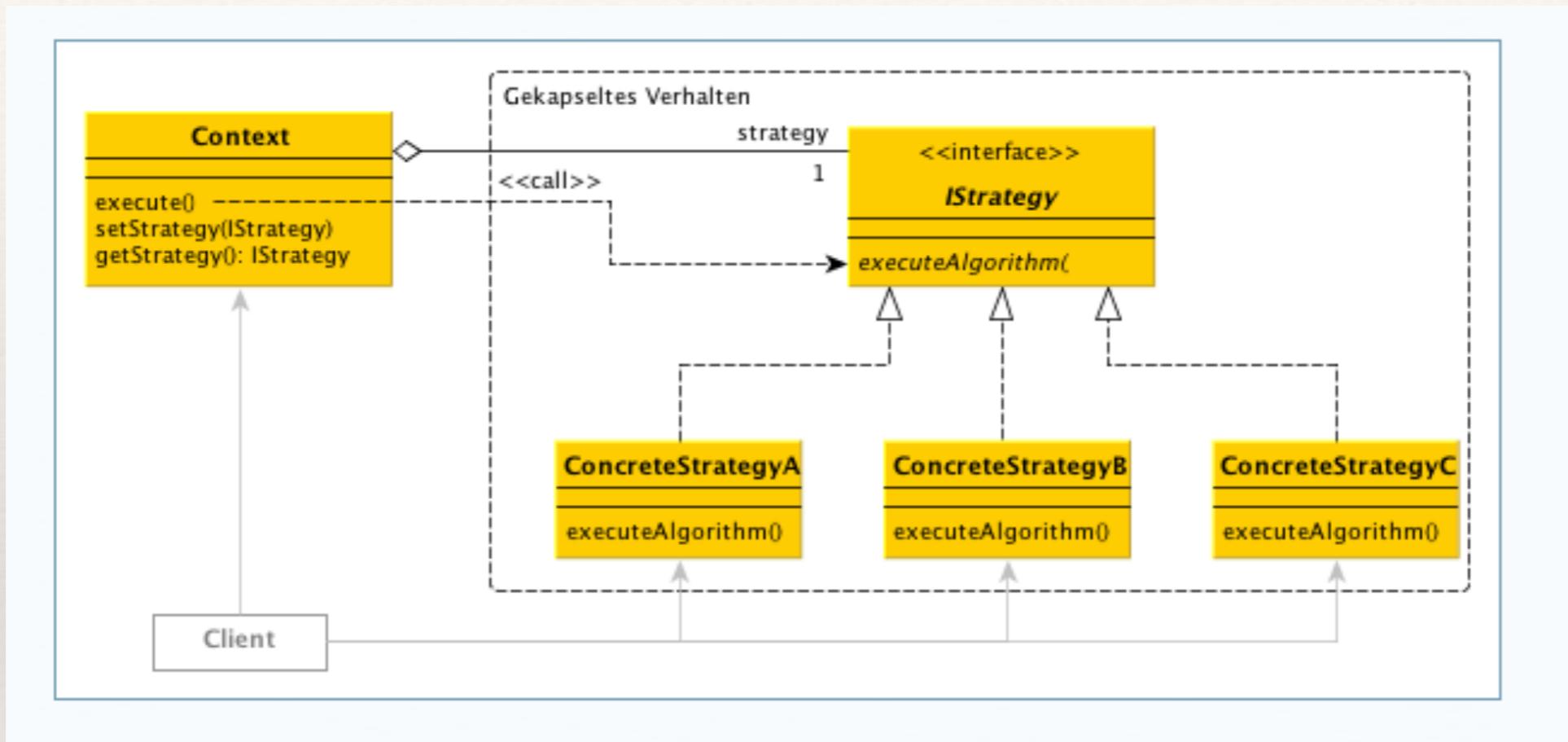
Strategy

- ❖ Kontext
 - ❖ Algorithmenfamilien verkörpern ein gemeinsames Konzept oder Prinzipien (bspw. Sortier- und Suchalgorithmen)
- ❖ Problem
 - ❖ Ein Algorithmus soll zur Laufzeit gegen einen alternativen Algorithmus aus derselben Familie ausgetauscht werden.
 - ❖ Ist der Algorithmus eng an ein Objekt gekoppelt, bspw. als eine von einer Vielzahl von Methoden, so ist der Wechsel des Algorithmus erschwert.

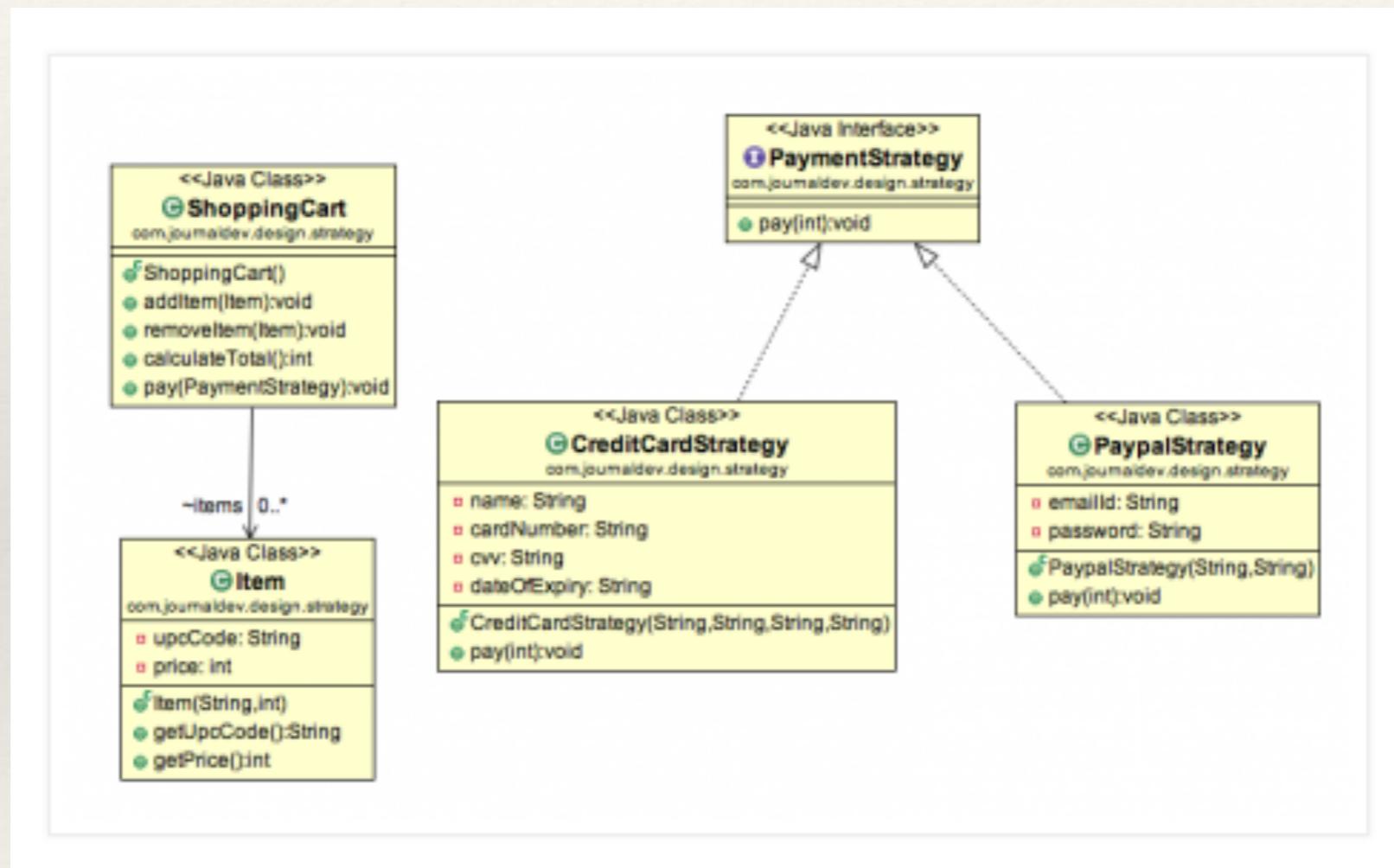
Strategy

- ❖ Lösung
 - ❖ Definiere eine Familie von Algorithmen, die durch eine abstrakte Oberklasse und konkrete Unterklassen realisiert wird.
- ❖ Schritte
 - ❖ Kapsele das algorithmische Konzept auf der abstrakten Ebene in einer Schnittstelle **Strategie**
 - ❖ Realisiere konkrete Algorithmen als Unterklassen (Konkrete Strategie).
 - ❖ Kontext-Klasse verfügt über eine Referenz auf ein Strategie-Objekt und delegiert die Ausführung des Algorithmus an dieses.
 - ❖ Kontext-Klasse wird mit einem geeigneten Strategie-Objekt konfiguriert.

Strategy



Beispiel



Beispiel

```
public class ShoppingCart {  
  
    // List of items  
    List<Item> items;  
  
    public ShoppingCart() {  
        this.items = new ArrayList<Item>();  
    }  
  
    public void addItem(Item item) {  
        this.items.add(item);  
    }  
  
    public void removeItem(Item item) {  
        this.items.remove(item);  
    }  
  
    public int calculateTotal() {  
        int sum = 0;  
        for (Item item : items) {  
            sum += item.getPrice();  
        }  
        return sum;  
    }  
  
    public void pay(PaymentStrategy paymentMethod) {  
        int amount = calculateTotal();  
        paymentMethod.pay(amount);  
    }  
}
```

Beispiel

```
public class Item {  
  
    private final String upcCode;  
    private final int price;  
  
    public Item(String upc, int cost) {  
        this.upcCode = upc;  
        this.price = cost;  
    }  
  
    public String getUpcCode() {  
        return upcCode;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
}
```

Beispiel

```
public interface PaymentStrategy {  
    public void pay(int amount);  
}
```

Beispiel

```
public class PaypalStrategy implements PaymentStrategy {  
  
    private final String emailId;  
    private final String password;  
  
    public PaypalStrategy(String email, String pwd) {  
        this.emailId = email;  
        this.password = pwd;  
    }  
  
    @Override  
    public void pay(int amount) {  
        System.out.println(amount + " paid using Paypal.");  
    }  
}
```

Beispiel

```
public class CreditCardStrategy implements PaymentStrategy {  
  
    private final String name;  
    private final String cardNumber;  
    private final String cvv;  
    private final String dateOfExpiry;  
  
    public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate) {  
        this.name = nm;  
        this.cardNumber = ccNum;  
        this.cvv = cvv;  
        this.dateOfExpiry = expiryDate;  
    }  
  
    @Override  
    public void pay(int amount) {  
        System.out.println(amount + " paid with credit/debit card");  
    }  
}
```

Vorteile

- ❖ Alternative zur Unterklassenbildung
- ❖ Wiederverwendbarkeit und Entkopplung von Context und Verhalten.
- ❖ Komposition statt Vererbung
- ❖ Dynamisches Verhalten
- ❖ Vermeidung von Bedingungen

Nachteile

- ❖ Enge Kopplung zwischen Client und StrategieImplementierung
- ❖ Unnötige Context Strategie-Kommunikation möglich.

Übung

- ❖ UebPattern.strategy.UebungStrategy.txt

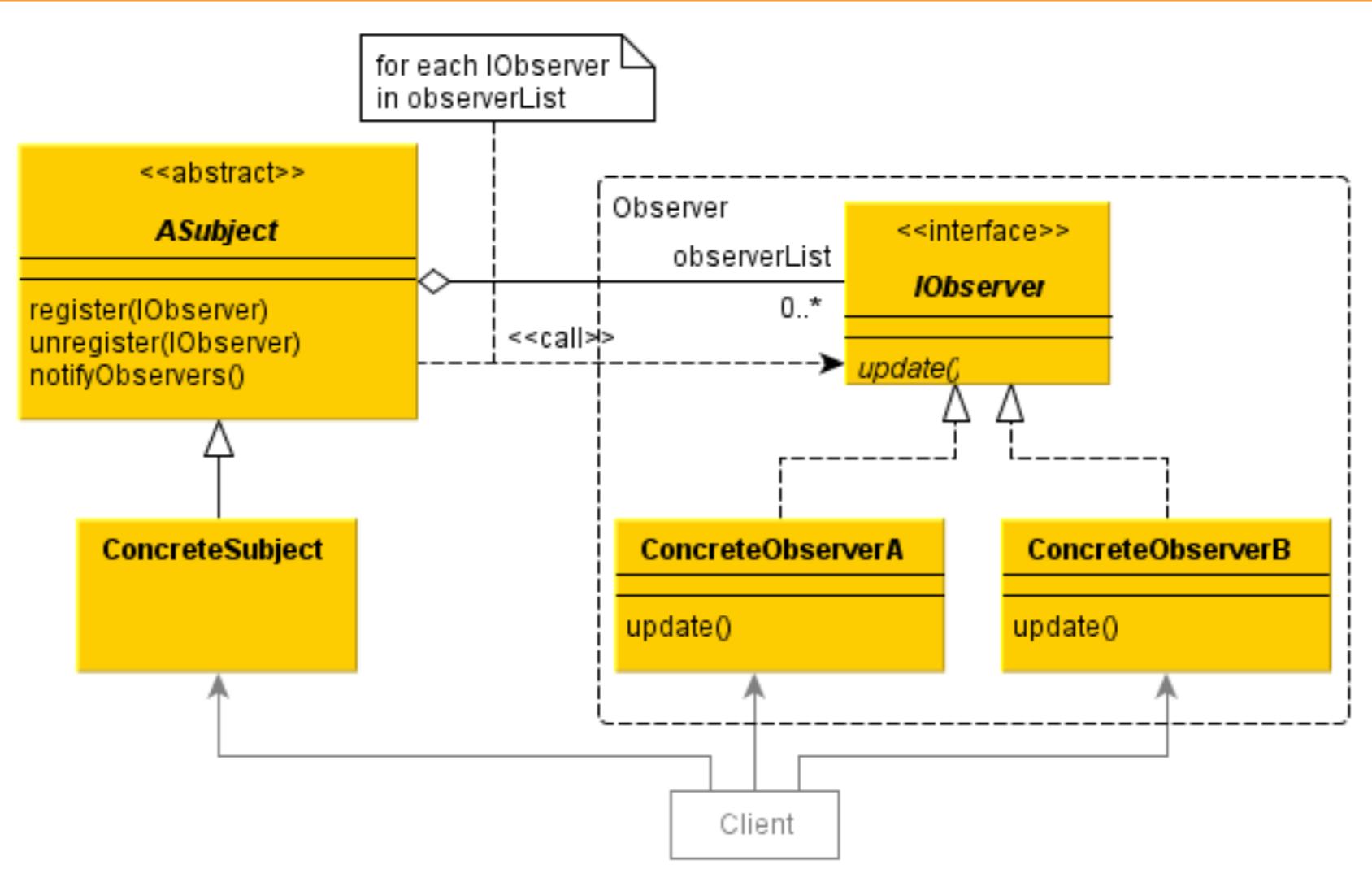
Observer

- ❖ Kontext
 - ❖ Auftreten bestimmter Ereignisse hat Auswirkungen auf Menge von Objekten in einem Subsystem
- ❖ Problem
 - ❖ Vermeidung von Zyklen und direkten Abhängigkeiten
 - ❖ Stetige Abfragen über Zustandsänderungen → *busy waiting*

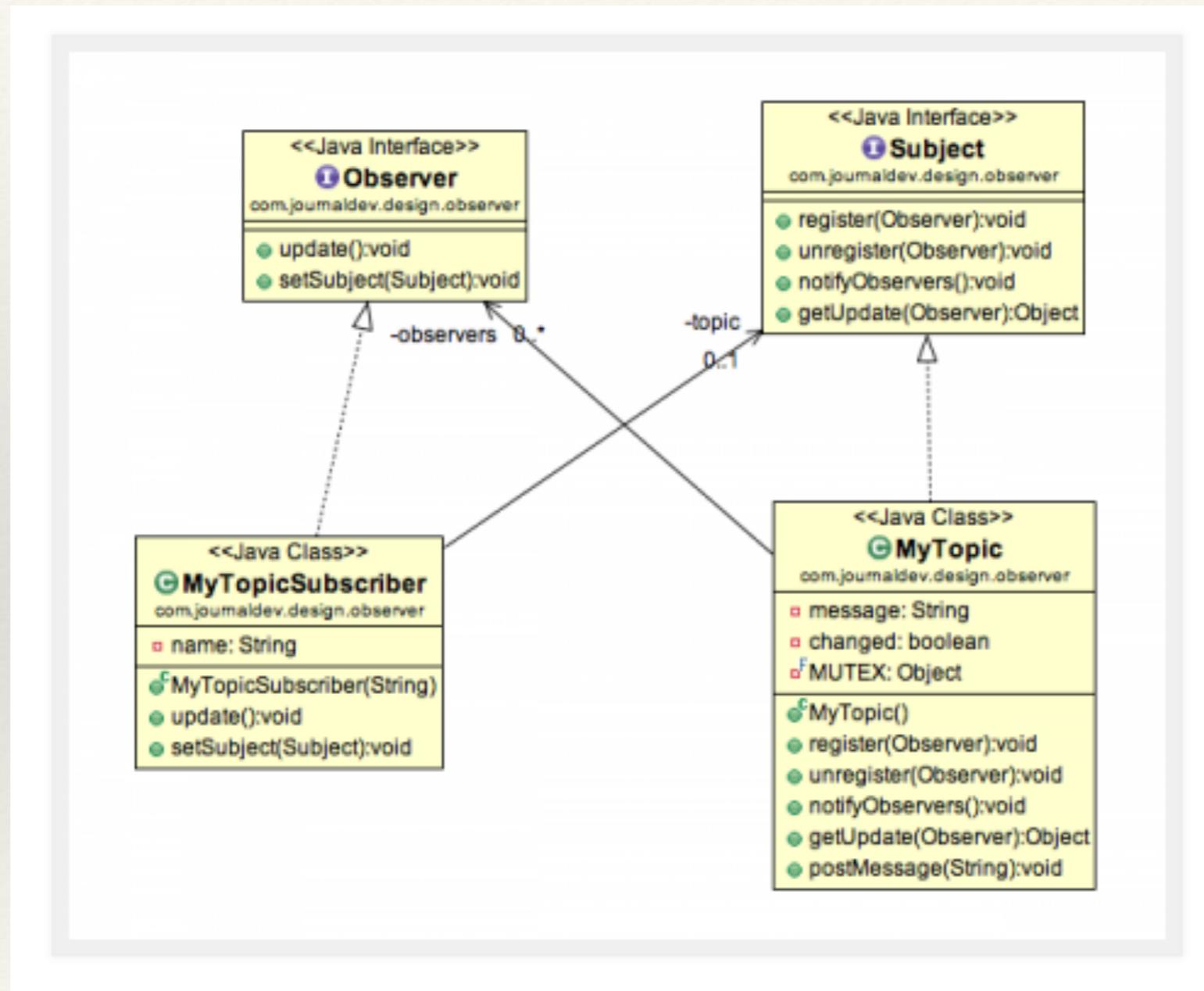
Observer

- ❖ Lösung
 - ❖ Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Observer



Observer



Beispiel

```
public interface Observer {  
    // method to update the observer, used by subject  
    public void update();  
  
    // attach with subject to observe  
    public void setSubject(Subject sub);  
}
```

Beispiel

```
public class MyTopicSubscriber implements Observer {

    private final String name;
    private Subject topic;

    public MyTopicSubscriber(String nm) {
        this.name = nm;
    }

    @Override
    public void update() {
        String msg = (String) topic.getUpdate(this);
        if (msg == null) {
            System.out.println(name + ":: No new message");
        } else
            System.out.println(name + ":: Consuming message::" + msg);
    }

    @Override
    public void setSubject(Subject sub) {
        this.topic = sub;
    }
}
```

Beispiel

```
public interface Subject {  
  
    // methods to register and unregister observers  
    public void register(Observer obj);  
  
    public void unregister(Observer obj);  
  
    // method to notify observers of change  
    public void notifyObservers();  
  
    // method to get updates from subject  
    public Object getUpdate(Observer obj);  
}
```

Beispiel

```
public class MyTopic implements Subject {  
  
    private final List<Observer> observers;  
    private String message;  
    private boolean changed;  
    private final Object MUTEX = new Object();  
  
    public MyTopic() {  
        this.observers = new ArrayList<Observer>();  
    }  
  
    @Override  
    public void register(Observer obj) {  
        if (obj == null)  
            throw new NullPointerException("Null Observer");  
        synchronized (MUTEX) {  
            if (!observers.contains(obj))  
                observers.add(obj);  
        }  
    }  
  
    @Override  
    public void unregister(Observer obj) {  
        synchronized (MUTEX) {  
            observers.remove(obj);  
        }  
    }  
}
```

Beispiel

```
@Override
public void notifyObservers() {
    List<Observer> observersLocal = null;
    // synchronization is used to make sure any observer registered after message is received is not notified
    synchronized (MUTEX) {
        if (!changed)
            return;
        observersLocal = new ArrayList<Observer>(this.observers);
        this.changed = false;
    }
    for (Observer obj : observersLocal) {
        obj.update();
    }
}

@Override
public Object getUpdate(Observer obj) {
    return this.message;
}

// method to post message to the topic
public void postMessage(String msg) {
    System.out.println("Message Posted to Topic:" + msg);
    this.message = msg;
    this.changed = true;
    notifyObservers();
}
}
```

Vorteile

- ❖ Zustandskonsistenz
- ❖ Flexibilität und Modalität
- ❖ Wiederverwendbarkeit
- ❖ Kompatibilität zum Schichtenmodell

Nachteile

- ❖ Aktualisierungskaskaden und -zyklen
- ❖ Abmeldung von Observer

Übung

- ❖ UebPattern.observer.UebungObserver.txt

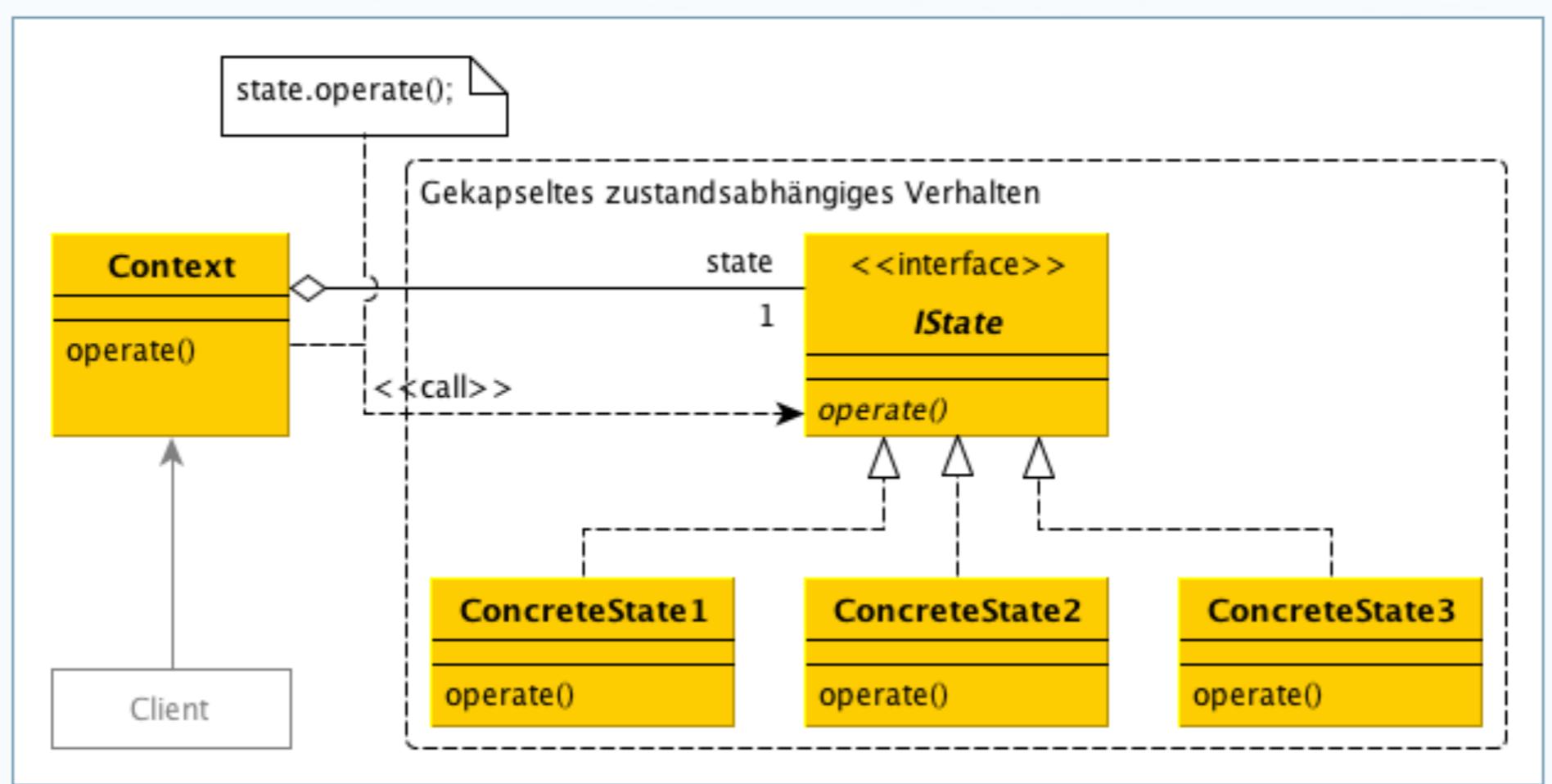
State

- ❖ Kontext
 - ❖ Objekte sollen häufig abhängig von ihrem jeweiligen internen Zustand unterschiedliches Verhalten zeigen
- ❖ Problem
 - ❖ Die zustandabhängige Variation des Verhaltens kann durch explizite Fallunterscheidungen durchgeführt werden. Bei umfangreichen bedingten Verzweigungen leidet jedoch die Lesbarkeit des resultierenden Programmcodes

State

- ❖ Lösung
 - ❖ Lagere das zustandsabhängige Verhalten in eine Klassenhierarchie aus, die durch eine abstrakte Oberklasse und konkrete Unterklassen realisiert wird.
- ❖ Im Detail
 - ❖ Kapsele das zustandsabhängige Verhalten auf der abstrakten Ebene in eine Schnittstelle **Zustand**
 - ❖ Realisiere das konkrete Verhalten für einen bestimmten Zustand in konkreten **Zustands-Klasse**.
 - ❖ Kontext-Klasse verfügt über eine Referenz auf ein Zustands-Objekt und delegiert die Ausführung an das jeweilige Objekt einer konkreten Zustands-Klasse
 - ❖ Kontext-Klasse wird mit einem geeigneten Zustands-Objekt initialisiert.
 - ❖ Bei einem Wechsel des Zustands wird das Zustands-Objekt ausgetauscht.

State



Beispiel

```
public interface State {  
    public void doAction();  
}
```

Beispiel

```
public class TVStartState implements State {

    @Override
    public void doAction() {
        System.out.println("TV is turned ON");
    }

}
```

Beispiel

```
public class TVStopState implements State {  
    @Override  
    public void doAction() {  
        System.out.println("TV is turned OFF");  
    }  
}
```

Beispiel

```
public class TVContext implements State {  
  
    private State tvState;  
  
    public void setState(State state) {  
        this.tvState = state;  
    }  
  
    public State getState() {  
        return this.tvState;  
    }  
  
    @Override  
    public void doAction() {  
        this.tvState.doAction();  
    }  
}
```

Beispiel

```
public class TVRemote {  
    public static void main(String[] args) {  
        TVContext context = new TVContext();  
        State tvStartState = new TVStartState();  
        State tvStopState = new TVStopState();  
  
        context.setState(tvStartState);  
        context.doAction();  
  
        context.setState(tvStopState);  
        context.doAction();  
    }  
}
```

Vorteile

- ❖ Erweiterbarkeit und Änderungsstabilität
- ❖ Intuitivität und Verständlichkeit
- ❖ Explizite Statusübergänge
- ❖ weniger fehlerträchtig als if-else Kaskaden.

Nachteile

- ❖ Erhöhte Klassenanzahl
- ❖ Weniger kompakt als eine einzige Klasse

Vergleich zwischen State & Strategy

- ❖ Kapselung zustandbasiertem Verhalten und Delegation des Verhaltens an den aktuellen Zustand *versus* Unterklassen entscheiden, wie Verhalten implementiert wird.
- ❖ Client hat oft Kenntnis von den Zuständen. Die Zustandswechsel werden intern und für den Client unsichtbar durchgeführt. Daher entsteht der Eindruck, der Context gehöre plötzlich einer anderen Klasse an, weil sich sein Verhalten ohne Einflussnahme des Clients geändert hat *versus* Der Client bestimmt häufig initial die zu verwendende Strategie(Algorithmus) und setzt das entsprechende Strategieobjekt einmalig selbst.

Vergleich zwischen State & Strategy

- ❖ Context oder Stateobjekt wechseln selbstständig den aktuellen Zustand. Der Zustandswechsel gehört mit zum Konzept des State Design Pattern *versus* Der Context oder das Strategyobjekt wechseln nicht selbstständig die aktuelle Strategie.
- ❖ Ein Contextobjekt wechselt zur Laufzeit sehr häufig seine Zustandsobjekte *versus* Häufig ist es *ein* Strategyobjekt das für ein Contextobjekt am besten passt und auch zur Laufzeit nicht mehr geändert wird

Vergleich zwischen State & Strategy

- ❖ Das State Design Pattern bietet eine Alternative zu einer großen Menge von strukturähnlichen Bedingungsanweisungen in Methoden *versus* Das Strategy Design Pattern bietet eine flexible Alternative zur Unterklassenbildung.

Übung

- ❖ UebPattern.state.UebungState.txt

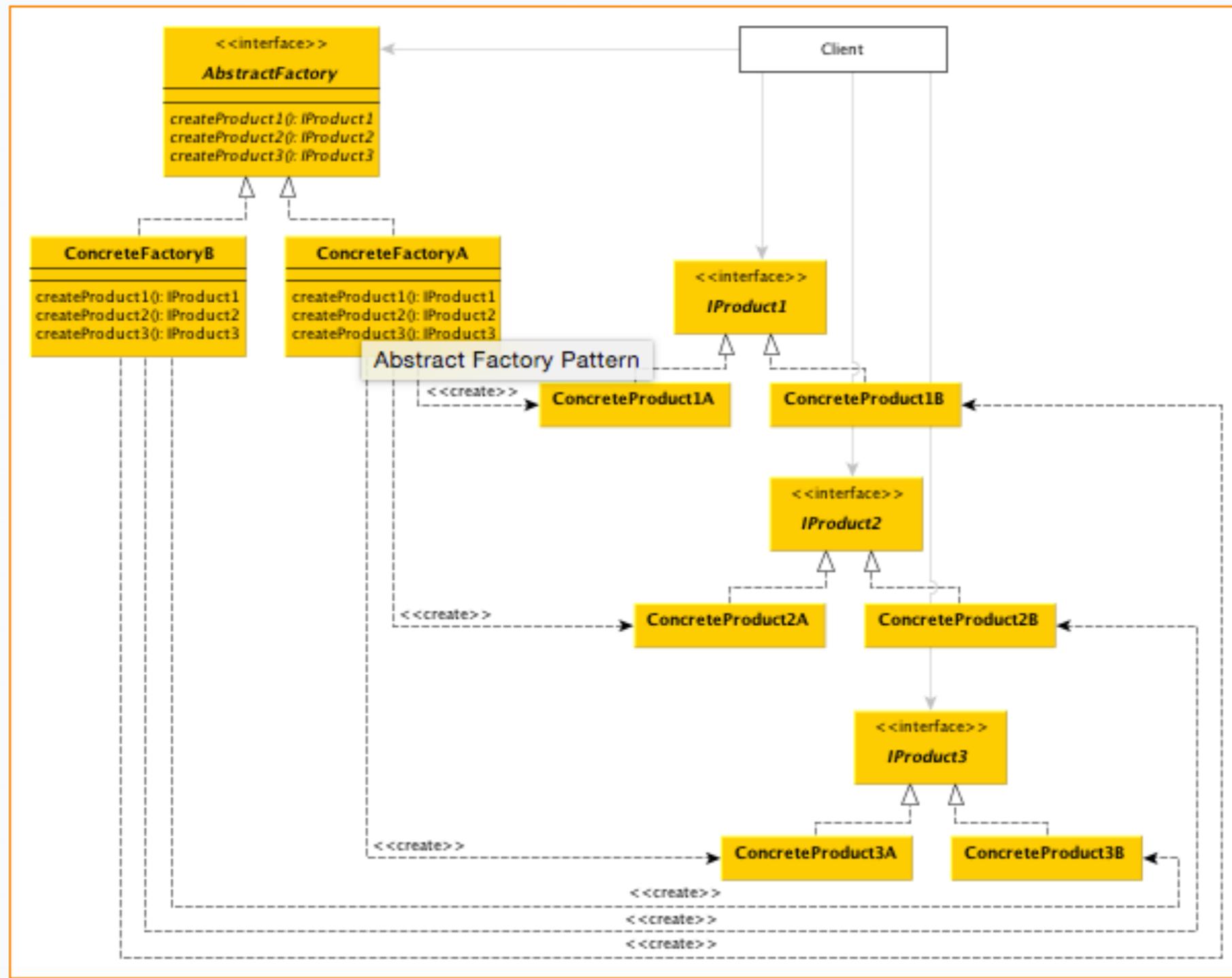
Abstract Factory

- ❖ Kontext
 - ❖ Objekte verschiedenen Typs treten in einem gemeinsamen Zusammenhang auf und sollen konsistent zueinander passen.
- ❖ Problem
 - ❖ Die Erzeugung einer Familie von Objekten soll nicht in der Verantwortlichkeit des Aufrufers liegen, sondern soll in einem Objekt gekapselt werden, das für eine konsistente Erzeugung von Produkten sorgt.
 - ❖ Die Änderung der Produktfamilie zur Laufzeit soll möglich sein.

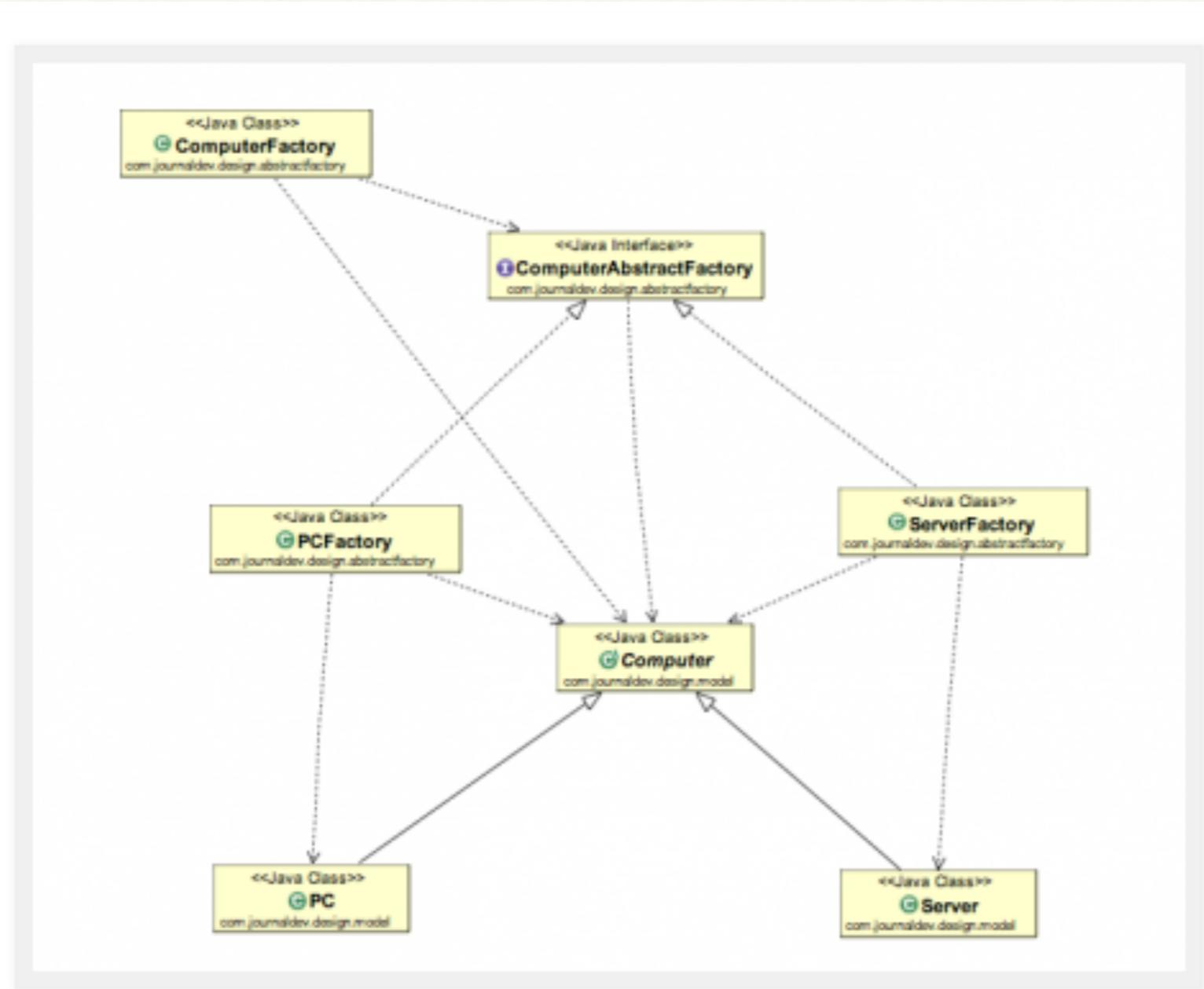
Abstract Factory

- ❖ Lösung
 - ❖ Kapsele die Erzeugung der Produkte in einem eigenen Objekt, einer konkreten Fabrik, die Methoden zur Erzeugung konkreter Produkte bereitstellt und die Produkte zueinander konsistent hält
 - ❖ Um Austauschbarkeit zur Laufzeit zu gewährleisten, führe abstrakte Oberklassen ein: Abstrakte Fabrik legt die Schnittstelle für alle konkreten Fabriken fest, abstrakte Produkte die für konkrete Produkte.
 - ❖ *Alternative* Produktfamilien wird durch Implementierung von weiteren konkreten Fabriken erreicht.
 - ❖ Es entstehen korrespondierende Hierarchien für die Fabriken und jedes in einer Fabrik erzeugte Produkt.

Abstract Factory



Beispiel



Beispiel

```
public class ComputerFactory {  
    public static Computer getComputer(ComputerAbstractFactory factory) {  
        return factory.createComputer();  
    }  
}
```

Beispiel

```
public interface ComputerAbstractFactory {  
    public Computer createComputer();  
}
```

Beispiel

```
public class PCFactory implements ComputerAbstractFactory {  
  
    private final String ram;  
    private final String hdd;  
    private final String cpu;  
  
    public PCFactory(String ram, String hdd, String cpu) {  
        this.ram = ram;  
        this.hdd = hdd;  
        this.cpu = cpu;  
    }  
  
    @Override  
    public Computer createComputer() {  
        return new PC(ram, hdd, cpu);  
    }  
}
```

Beispiel

```
public class ServerFactory implements ComputerAbstractFactory {  
    private final String ram;  
    private final String hdd;  
    private final String cpu;  
  
    public ServerFactory(String ram, String hdd, String cpu) {  
        this.ram = ram;  
        this.hdd = hdd;  
        this.cpu = cpu;  
    }  
  
    @Override  
    public Computer createComputer() {  
        return new Server(ram, hdd, cpu);  
    }  
}
```

Beispiel

```
public abstract class Computer {  
  
    public abstract String getRAM();  
  
    public abstract String getHDD();  
  
    public abstract String getCPU();  
  
    @Override  
    public String toString() {  
        return "RAM= " + this.getRAM() + ", HDD=" + this.getHDD() + ", CPU=" + this.getCPU();  
    }  
}
```

Beispiel

```
public class PC extends Computer {  
  
    private final String ram;  
    private final String hdd;  
    private final String cpu;  
  
    public PC(String ram, String hdd, String cpu) {  
        this.ram = ram;  
        this.hdd = hdd;  
        this.cpu = cpu;  
    }  
  
    @Override  
    public String getRAM() {  
        return this.ram;  
    }  
  
    @Override  
    public String getHDD() {  
        return this.hdd;  
    }  
  
    @Override  
    public String getCPU() {  
        return this.cpu;  
    }  
}
```

Beispiel

```
public class Server extends Computer {  
  
    private final String ram;  
    private final String hdd;  
    private final String cpu;  
  
    public Server(String ram, String hdd, String cpu) {  
        this.ram = ram;  
        this.hdd = hdd;  
        this.cpu = cpu;  
    }  
  
    @Override  
    public String getRAM() {  
        return this.ram;  
    }  
  
    @Override  
    public String getHDD() {  
        return this.hdd;  
    }  
  
    @Override  
    public String getCPU() {  
        return this.cpu;  
    }  
}
```

Vorteile

- ❖ Abschirmen der konkreten Klasse
- ❖ Konsistenz
- ❖ Flexibilität
- ❖ Einfache Erweiterung mit neuen Produktfamilien
- ❖ Wiederverwendbarkeit

Nachteile

- ❖ Unflexibilität hinsichtlich neuer Familienmitglieder

Übung

- ❖ UebPattern.abstractfactory.UebungAbstractFactory.txt

Builder

- ❖ kommt zum Einsatz, wenn ein Objekt viele optionale Variablen besitzt. Die bisherige, klassische Variante bei solchen Objekten ist das Schreiben von mehreren Konstruktoren ('telescoping constructor'), die über unterschiedliche Anzahl von Parametern verfügen. Allerdings sinkt die Lesbarkeit bei mehr als 6 Parametern gewaltig (vor allem wenn die Parameter vom gleichen Typ sind).

Builder

- ❖ Das “Builder-Pattern” verzichtet auf die ‘telescoping constructors’ und arbeitet stattdessen mit einer statischen inneren Klasse, die für das Erzeugen einer Instanz der äusseren Klasse zuständig ist. Der Konstruktor des Builder-Objekts enthält alle “Pflicht”-Variablen, die optionalen Variablen werden über (verkettbare) Methodenaufrufe hinzugefügt. Abschliessend wird die `build` Methode aufgerufen, die ein unveränderliches Objekt der äusseren Klasse erzeugt. Der Konstruktor der äusseren Klasse ist privat, d.h. nur ein Builder kann ein Objekt tatsächlich instanzieren:

Builder

```
public class Computer {  
  
    // required parameters  
    private final String HDD;  
    private final String RAM;  
  
    // optional parameters  
    private final boolean isGraphicsCardEnabled;  
    private final boolean isBluetoothEnabled;  
  
    public String getHDD() {  
        return HDD;  
    }  
  
    public String getRAM() {  
        return RAM;  
    }  
  
    public boolean isGraphicsCardEnabled() {  
        return isGraphicsCardEnabled;  
    }  
  
    public boolean isBluetoothEnabled() {  
        return isBluetoothEnabled;  
    }  
  
    private Computer(ComputerBuilder builder) {  
        this.HDD = builder.HDD;  
        this.RAM = builder.RAM;  
        this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;  
        this.isBluetoothEnabled = builder.isBluetoothEnabled;  
    }  
}
```

Builder

```
// Builder Class
public static class ComputerBuilder {

    // required parameters
    private final String HDD;
    private final String RAM;

    // optional parameters
    private boolean isGraphicsCardEnabled;
    private boolean isBluetoothEnabled;

    public ComputerBuilder(String hdd, String ram) {
        this.HDD = hdd;
        this.RAM = ram;
    }

    public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {
        this.isGraphicsCardEnabled = isGraphicsCardEnabled;
        return this;
    }

    public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {
        this.isBluetoothEnabled = isBluetoothEnabled;
        return this;
    }

    public Computer build() {
        return new Computer(this);
    }

}
```

Übung

- ❖ UebPattern.builder.UebungBuilder.txt

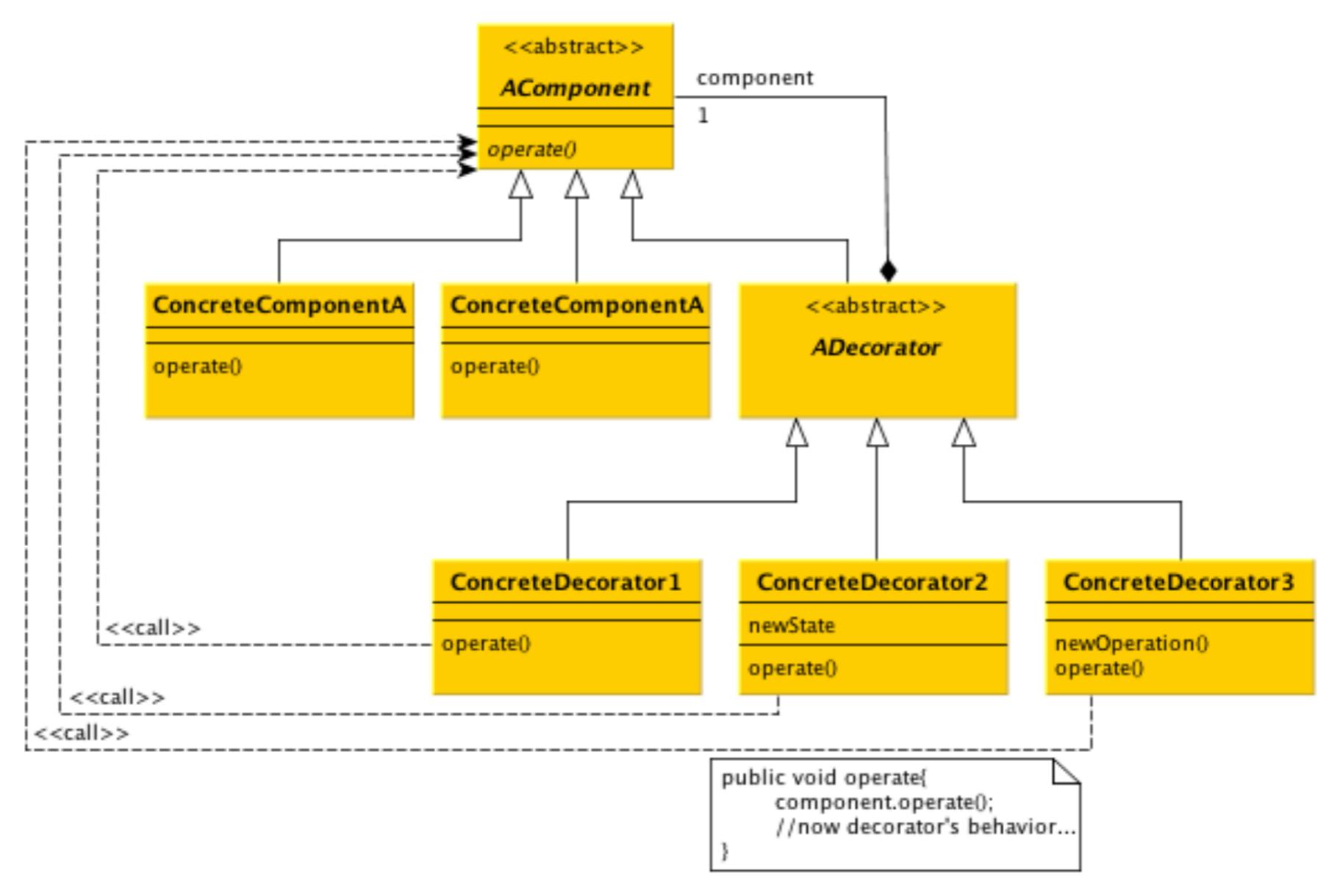
Decorator

- ❖ Kontext
 - ❖ Ein Objekt soll um zusätzliche Eigenschaften erweitert werden
- ❖ Problem
 - ❖ Direkte Implementierungsvererbung der bisherigen Eigenschaften und Hinzunahme der zusätzlichen Eigenschaft hat den Nachteil, dass die Klassenhierarchie zur Übersetzungszeit festgelegt sein muss.
 - ❖ Mehrere Eigenschaften können nicht getrennt voneinander ergänzt werden, da bei der Vererbung immer alle Operationen vererbt werden. Für jede Kombination der Eigenschaften würde also eine neue Unterklasse benötigt werden (exponentielles Wachstum).

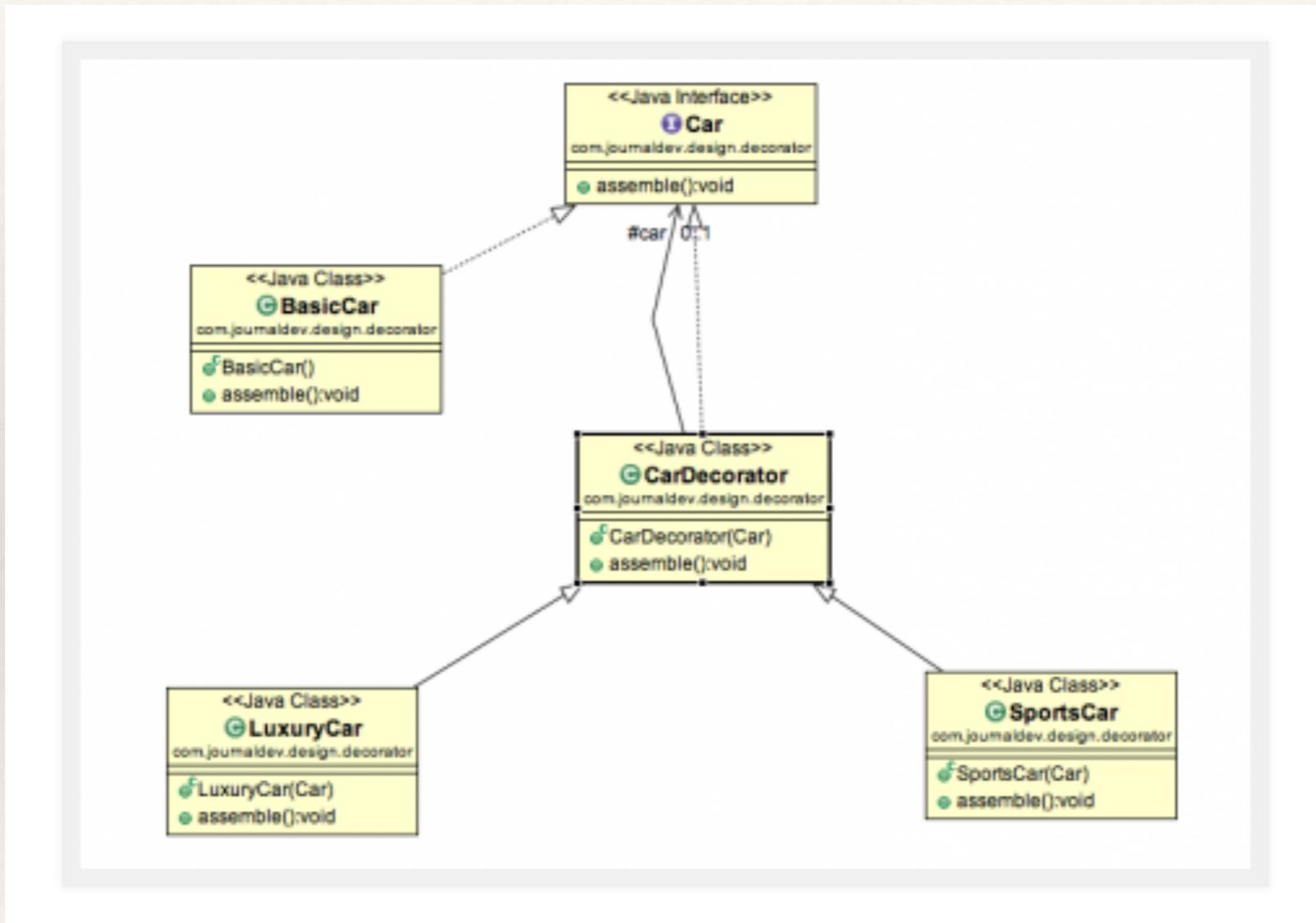
Decorator

- ❖ Lösung
 - ❖ Nutze Komposition und Delegation zur Verwendung bisheriger Eigenschaften und kapsele jeweils die zusätzliche Eigenschaft in eine eigene Klasse
- ❖ Schritte
 - ❖ Definiere wie beim Kompositum-Muster eine gemeinsame Schnittstelle **Komponente** für **einfache Komponenten** und für die Bereitsteller der zusätzlichen Eigenschaften, die **Dekoratoren**.
 - ❖ Ein Dekorator aggregiert (genau) eine Komponente und verwendet deren Operation durch Delegation.
 - ❖ **Konkrete Dekoratoren** bieten ihre zusätzliche Eigenschaft an und verwenden diese gegebenenfalls in einer Spezialisierung der Schnittstellenoperation.
 - ❖ Durch die Beschränkung auf eine Komponente weisen Dekorator-Objekte eine linear entartete Baumstruktur auf.

Decorator



Beispiel



Beispiel

```
public interface Car {  
    public void assemble();  
}
```

Beispiel

```
public class BasicCar implements Car {  
    @Override  
    public void assemble() {  
        System.out.print("Basic Car.");  
    }  
}
```

Beispiel

```
public class CarDecorator implements Car {  
    protected Car car;  
    public CarDecorator(Car c) {  
        this.car = c;  
    }  
    @Override  
    public void assemble() {  
        this.car.assemble();  
    }  
}
```

Beispiel

```
public class LuxuryCar extends CarDecorator {  
    public LuxuryCar(Car c) {  
        super(c);  
    }  
  
    @Override  
    public void assemble(){  
        car.assemble();  
        System.out.print(" Adding features of Luxury Car.");  
    }  
}
```

Beispiel

```
public class SportsCar extends CarDecorator {  
  
    public SportsCar(Car c) {  
        super(c);  
    }  
  
    @Override  
    public void assemble() {  
        car.assemble();  
        System.out.print(" Adding features of Sports Car.");  
    }  
}
```

Vorteile

- ❖ Dynamik und Flexibilität
- ❖ Transparenz
- ❖ Performance
- ❖ Wartbarkeit durch schlanke, kohäsive Klassen.

Nachteile

- ❖ Erschwerte Fehlerfindung
- ❖ Hohe Objektanzahl
- ❖ unkomfortable, wortreiche API
- ❖ Keine Objektidentität

Übung

- ❖ UebPattern.decorator.UebungDecorator.txt

Composite

- ❖ Kontext
 - ❖ Zur Strukturierung werden komplexe Objekte hierarchisch aus einzelnen Bestandteilen aufgebaut.
- ❖ Problem
 - ❖ Die Behandlung komplex zusammengesetzter Objekte führt oft zu unübersichtlichem Programmcode mit vielen Fallunterscheidungen und Typprüfungen

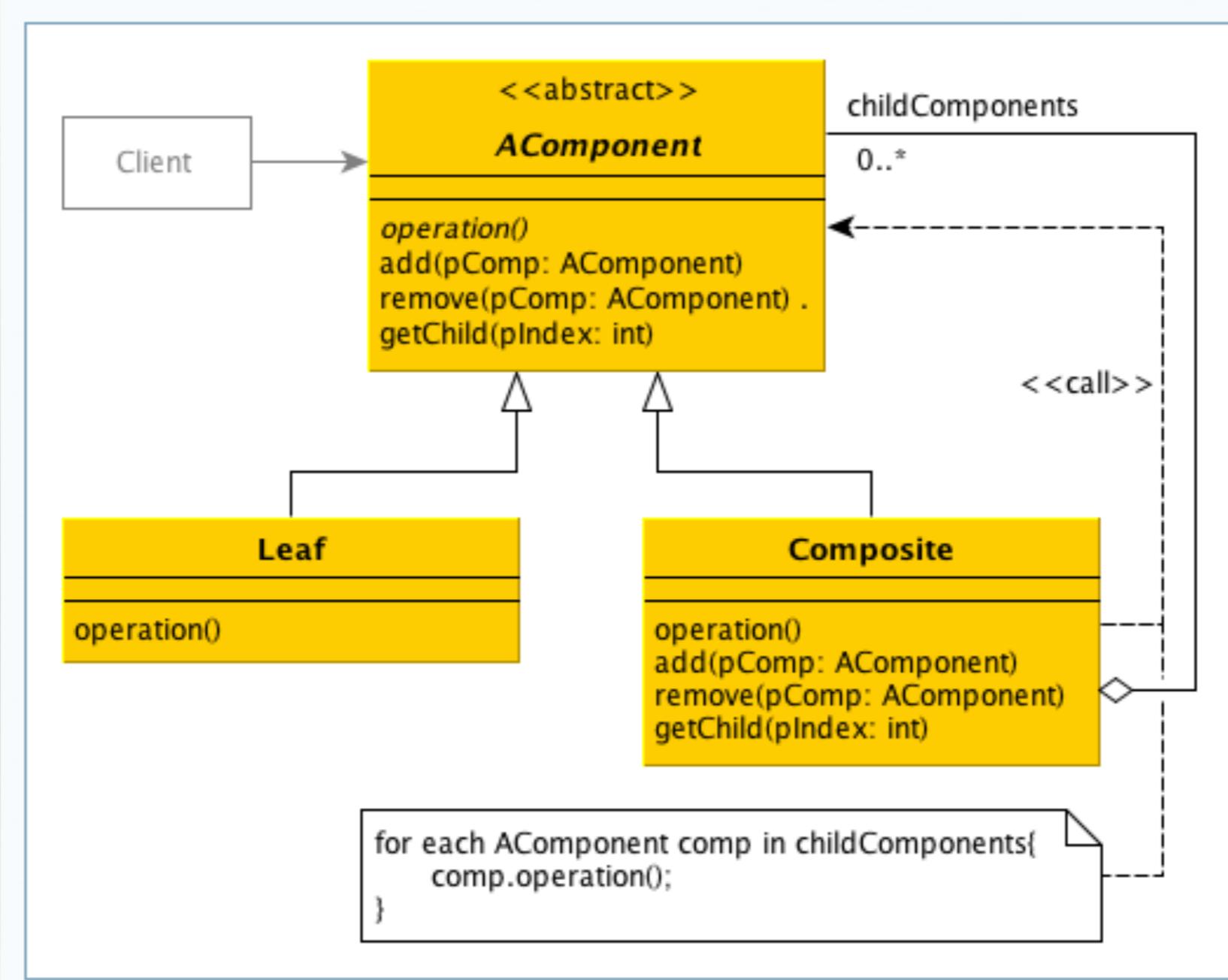
Composite

- ❖ Problem(Fortsetzung)
 - ❖ Besitzen komplexe und einfache Objekte gemeinsame Eigenschaften, wie z.B. kopieren, darstellen usw., so wäre es wünschenswert, den Aufruf unabhängig davon durchzuführen, wie komplex die Objekte sind.
 - ❖ Da komplexe Objekte hierarchisch aus einfacheren zusammengesetzt werden, sollten die Operationen der Bestandteile verwendet werden für die Operationen des Ganzen.

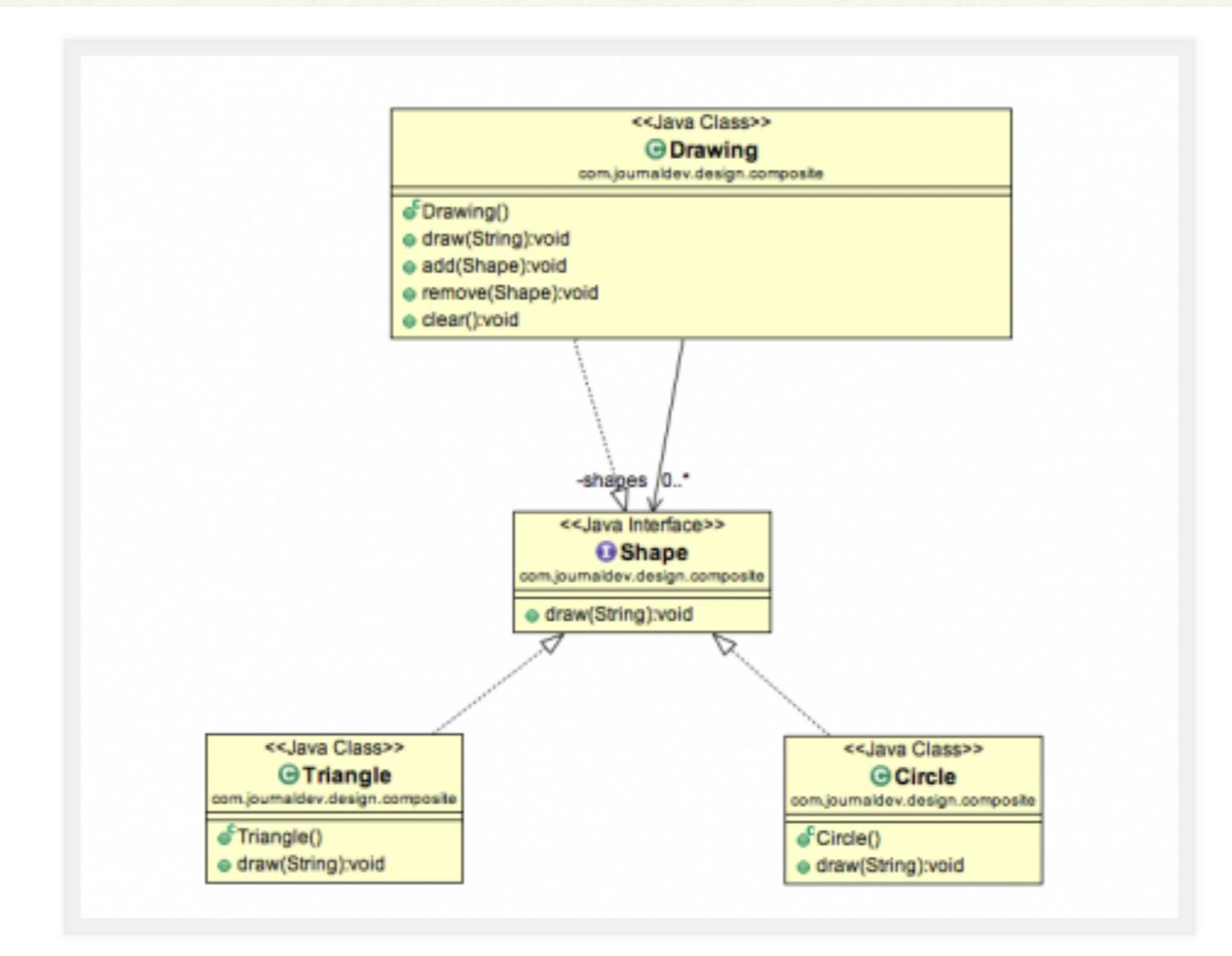
Composite

- ❖ Lösung
 - ❖ Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.
- ❖ Im Detail
 - ❖ Definiere eine Schnittstelle **Komponente**, die alle für einfache und komplexe Objekte gemeinsamen Operationen umfasst.
 - ❖ Einfache Objekte sind Instanzen einer oder mehrerer **Blatt-Klassen**.
 - ❖ Komplexe Objekte setzen sich aus mehreren Komponenten zusammen und können sich bei der Konkretisierung der Schnittstellen-Operationen auf die Operationen der Bestandteile abstützen (Komposition und Delegation).
 - ❖ Instanzen komplexer Objekte sind baumartig strukturiert.

Beispiel



Beispiel



Beispiel

```
public class Drawing implements Shape {  
  
    // collection of Shapes  
    private final List<Shape> shapes = new ArrayList<Shape>();  
  
    @Override  
    public void draw(String fillColor) {  
        for (Shape sh : shapes) {  
            sh.draw(fillColor);  
        }  
    }  
  
    // adding shape to drawing  
    public void add(Shape s) {  
        this.shapes.add(s);  
    }  
  
    // removing shape from drawing  
    public void remove(Shape s) {  
        shapes.remove(s);  
    }  
  
    // removing all the shapes  
    public void clear() {  
        System.out.println("Clearing all the shapes from drawing");  
        this.shapes.clear();  
    }  
}
```

Beispiel

```
public interface Shape {  
    public void draw(String fillColor);  
}
```

Beispiel

```
public class Circle implements Shape {  
    @Override  
    public void draw(String fillColor) {  
        System.out.println("Drawing Circle with color " + fillColor);  
    }  
}
```

Beispiel

```
public class Triangle implements Shape {  
    @Override  
    public void draw(String fillColor) {  
        System.out.println("Drawing Triangle with color " + fillColor);  
    }  
}
```

Vorteile

- ❖ Repräsentation von verschachtelten Strukturen.
- ❖ Vereinfachter Clientcode
- ❖ Elegantes Arbeiten mit der Baumstruktur
- ❖ Flexibilität und Erweiterbarkeit

Nachteile

- ❖ Schwierige Definition der allgemeinen Componentschnittstelle
- ❖ Spätere Einschränkungen der erlaubten Compositelemente ist diffizil.

Übung

- ❖ UebPattern.composite.UebungComposite.txt