Decorator Pattern

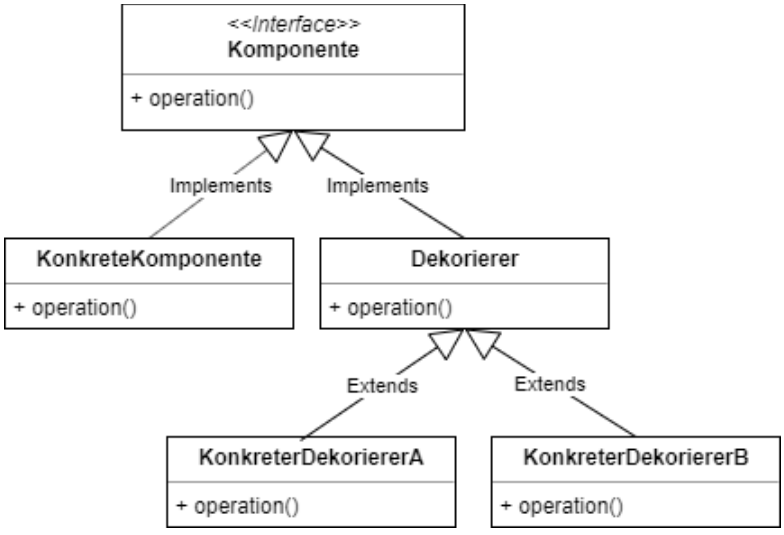
Zweck

Das Decorator Design Pattern ermöglicht es während der Laufzeit das Verhalten einer Komponente dynamisch zu erweitern. Dieser Vorgang nennt man dekorieren. Die Komponente kann mit beliebig vielen Verhalten erweitert werden. Fügt man der Komponente ein weiteres Verhalten hinzu, umhüllt man diese mit erweiterter Funktionalität, wobei die Schnittstelle gegen aussen nicht verändert wird.

Gewährleistet

- **Open-Closed-Prinzip.** Offen für Erweiterung, geschlossen für Veränderung. Fügt man der Komponente ein Verhalten hinzu, muss nichts an der bestehenden Komponente verändert werden. Es wird lediglich eine neue konkrete Decorator-Implementation hinzugefügt.
- **Flexibilität und Dynamik.** Beliebige Kombinationsmöglichkeiten (auch Mehrfachkombinationen) sind möglich und dies zur Laufzeit.
- **Wiederverwendbarkeit.** Komponenten und Dekoratoren können wiederverwendet werden.
- **Konsistenz und Wartbarkeit.** Wartung und Erweiterungen müssen nur an einer Klasse durchgeführt werden.
- **Übersichtlichkeit.** Lange und unübersichtliche Vererbungshierarchien können vermieden werden. Denn es muss nicht für jede Kombination eine neue Subklasse erstellt werden, sondern nur für jede weitere Zutat.

UML



Problem

Sie bieten in ihrem Restaurant Pizzen an. Der Kunde hat die Möglichkeit seine Pizza individuell zu gestalten. Er kann nach Wunsch Zutaten auf seiner Pizza hinzufügen. Die Pizzen können somit nicht vor der Bestellung gelistet werden, sondern müssen während der Bestellung zusammengestellt werden können. Der Gesamtpreis muss pro Pizza anhand der ausgewählten Zutaten berechnet werden. Die Küche erhält nach der Bestellung eine vollständige Beschreibung der vom Kunden zusammengestellten Pizza um diese zuzubereiten.

Lösung

Als Vorgabe für ihre Pizza-Implementation definieren Sie ein Interface, welches vorgibt, welche Methoden auf der Pizza aufgerufen werden können. Wir wollen «getPreis()» (Preis der Pizza) und «print()» (Beschreibung der Pizza) vorgeben.

Alle Kombinationsmöglichkeiten starten mit einer Basis-Komponente, welche mit Dekorationen erweitert wird. Die Basis-Komponente implementiert das Interface. In unserem Fall ist das die «BasePizza» mit dem «IPizza» Interface.

Jede Dekoration erbt von einer abstrakten Klasse, welche ebenfalls das Interface implementiert, damit bei der Erweiterung der Basis-Komponente die gleiche Schnittstelle zur Verfügung steh, wie nur mit der Base-Implementation ohne Dekoratoren. Zusätzlich hält jede Dekoration den Verweis auf das Parent. Damit am Schluss über die ganze Kette operiert und zusammengefasst werden kann.

Eine konkrete Implementation, wie zum Beispiel «Pilze» oder «Schinken», implementiert die Methoden der Pizza und erweitert den Rückgabewert des Parent's mit seinem spezifischen Verhalten. Dieses Verhalten ist bei den Pilzen einfach die Beschreibung «mit Pilzen» und der zusätzliche Preis von 2.50 CHF.

Beispiel in Java

BasePizza

```
// Basiskomponente bildet Grundlage zum Dekorieren (KonkreteKomponente)
public class BasePizza implements IPizza
{
	// Basispreis
	@Override
	public double getPreis() {
		return 8;
	}

	// Basisbeschreibung
	@Override
	public String print() {
		return "Pizza mit";
	}
}
```

IPizza

```
// Interface gibt vor, welche Methoden auf der Pizza vorhanden sind (Komponente)
public interface IPizza
{
	// Rückgabe des Preises anhand der Zutaten auf der Basiskomponente
	public double getPreis();
	// Beschreibung der Pizza mit den entsprechenden Zutaten
	public String print();
}
```

Zutat

```
// Abstrakte Klasse für die Dekoratoren (Wrapper) mit gleichen Schnitstellen (Dekorierer)
public abstract class Zutat implements IPizza {
	// Referenz zur Komponente, damit der Ketten-Operationen auf allen Hüllen möglich sind
	protected IPizza pizza;
	public Zutat(IPizza pizza)
	{
		this.pizza = pizza;
	}
	// Basiszusatzpreis für jede Zutat
	@Override
	public double getPreis() {
		return pizza.getPreis();
	}
	// Basiszusatzbeschreibung für jede Zutat
	@Override
	public String print() {
		return pizza.print();
	 }
}
```

Schinken

```
// Konkrete implementation einer Zutat (KonkreterDekorierer)
public class Schinken extends Zutat {
	public Schinken(IPizza pizza) {
		super(pizza);
	}
	// Erweiterung des Preises
	@Override
	public double getPreis() {
		return pizza.getPreis() + 2.50;
	}
	// Erweiterung des Beschreibung
	@Override
	public String print() {
		return pizza.print() + ", Schinken";
	}
}
```

Main

```
public class Main {
	// Demo
	public static void main(String[] args) {
		IPizza pizza = new Pilze(new BasePizza());
		System.out.println(pizza.print() + " für "+ pizza.getPreis() + " CHF");

		pizza = new Schinken( pizza);
		System.out.println(pizza.print() + " für "+ pizza.getPreis() + " CHF");
	}
}
```