



Decorator Muster

Der **Decorator** (auch **Dekorierer**) ist ein [Entwurfsmuster](#) aus dem Bereich der [Softwareentwicklung](#), das zur Kategorie der [Strukturmuster](#) (engl. *structural patterns*) gehört. Das Muster ist eine flexible Alternative zur Unterklassenbildung, um eine [Klasse](#) um zusätzliche Funktionalitäten zu erweitern.^[1] Es ist ein Entwurfsmuster der sogenannten [GoF-Muster](#).

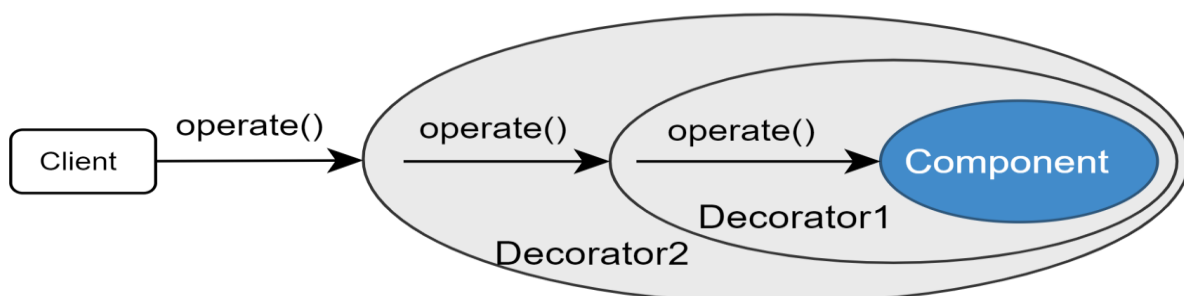
Decorator Pattern: Das Muster für dynamische Klassenerweiterungen

Wollen Sie eine bereits **bestehende Klasse** in einer objektorientierten Software um **neue Funktionalitäten** erweitern, haben Sie zwei verschiedene Möglichkeiten: Die einfache, aber auch schnell unübersichtliche Lösung ist, Unterklassen zu implementieren, die die Basisklasse in entsprechender Weise ergänzen. Als Alternative kann man eine **Dekorierer-Instanz** gemäß dem sogenannten Decorator Design Pattern verwenden.

Was ist das Decorator Pattern (Decorator-Entwurfsmuster)?

Das Decorator Design Pattern, kurz Decorator Pattern (dt. Decorator-Muster), ist eine 1994 veröffentlichte Musterstrategie für die **übersichtliche Erweiterung von Klassen** in objektorientierter Computersoftware. Nach dem Muster lässt sich jedes beliebige Objekt um ein gewünschtes Verhalten ergänzen, ohne dabei das Verhalten anderer Objekte derselben Klasse zu beeinflussen. Strukturell ähnelt das Decorator Pattern stark dem „Chain of responsibility“-Pattern (dt. Zuständigkeitskette), wobei Anfragen anders als bei diesem Zuständigkeitskonzept mit zentralem Bearbeiter von allen Klassen entgegengenommen werden.

Die Software-Komponente, die erweitert werden soll, wird nach dem Decorator-Entwurfsmuster mit einer bzw. mehreren **Decorator-Klassen** „dekoriert“, die die Komponente vollständig umschließen. Jeder Decorator ist dabei vom selben Typ wie die **umschlossene Komponente** und verfügt damit über die **gleiche Schnittstelle**. Dadurch kann er eingehende Methodenaufrufe unkompliziert an die verknüpfte Komponente delegieren, während er wahlweise zuvor bzw. anschließend das eigene Verhalten ausführt. Auch eine direkte Verarbeitung eines Aufrufs im Decorator ist grundsätzlich möglich.



Welchen Zweck erfüllt das Decorator Design Pattern?

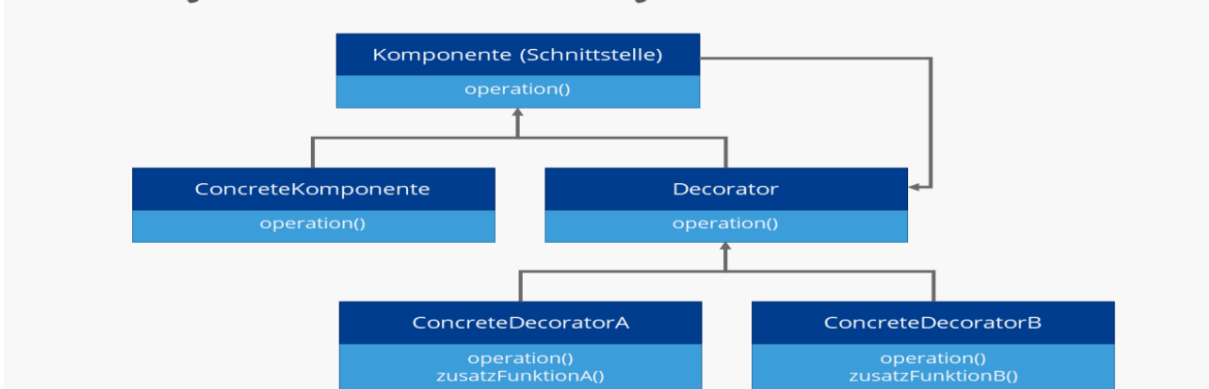
Wie andere GoF-Muster, etwa das [Strategy Pattern](#) oder das [Builder Pattern](#), verfolgt das Decorator Pattern das Ziel, Komponenten objektorientierter Software flexibler und einfacher wiederverwendbar zu gestalten. Zu diesem Zweck liefert der Ansatz die Lösung dafür, **Abhängigkeiten zu einem Objekt dynamisch** und – insofern erforderlich – **während der Laufzeit hinzufügen bzw. entfernen** zu können. Insbesondere aus diesem Grund stellt das Pattern eine gute Alternative zum Einsatz von Subklassen dar: Diese können eine Klasse zwar ebenfalls auf vielfältige Weise ergänzen, lassen jedoch keinerlei Anpassungen während der Laufzeit zu.

Decorator Pattern: UML-Diagramm zur Veranschaulichung

Der Decorator bzw. die Decorator-Klassen (ConcreteDecorator) verfügen über die **gleiche Schnittstelle** wie die zu dekorierende Software-Komponente (ConcreteKomponente) und sind vom gleichen Typ. Dieser Umstand ist wichtig für das **Handling der Aufrufe**, die entweder unverändert oder verändert weitergeleitet werden, falls der Decorator die Verarbeitung nicht selbst übernimmt. Im Decorator-Pattern-Konzept bezeichnet man diese elementare Schnittstelle, die im Prinzip eine abstrakte Superklasse ist, als „Komponente“.

Das Zusammenspiel von Basiskomponente und Decorator lässt sich am besten in einer **grafischen Darstellung der Beziehungen** in Form eines [UML-Klassendiagramms](#) verdeutlichen. In der nachfolgenden abstrakten Abbildung des Decorator Design Patterns haben wir daher die Modellierungssprache für objektorientierte Programmierung verwendet.

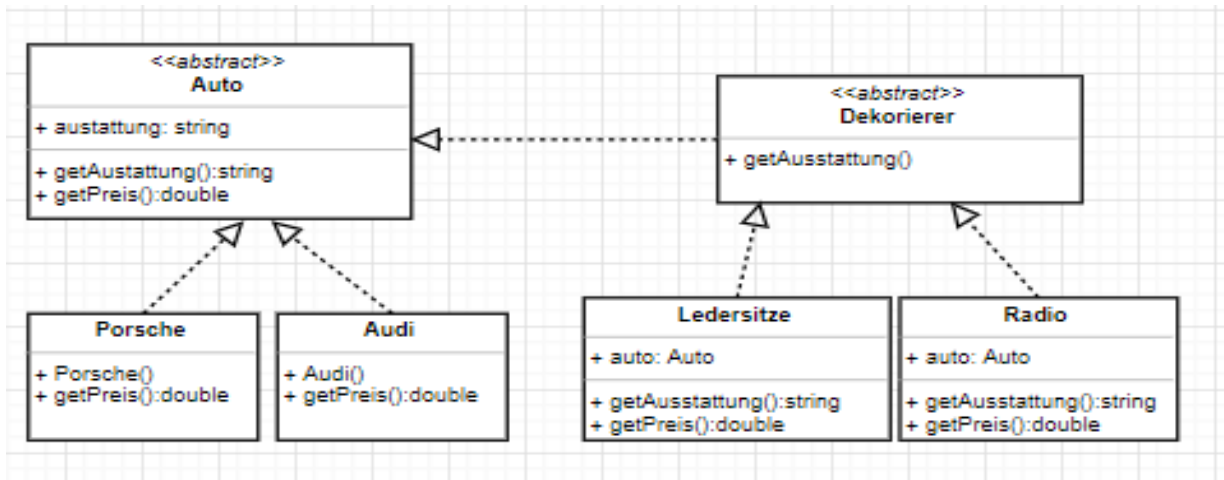
UML-Diagramm: Decorator Design Pattern



Die Vorteile und Nachteile des Decorator Patterns im Überblick

Vorteile	Nachteile
Hohes Maß an Flexibilität	Hohe Komplexität der Software (insbesondere der Decorator-Schnittstelle)
Funktionserweiterung von Klassen ohne Vererbung	Einsteigerunfreundlich
Gut lesbarer Programmcode	Hohe Anzahl an Objekten
Ressourcenoptimierte Funktionsaufrufe	Erschwerter Debugging-Prozess

Praxisbeispiel für die Umsetzung des Decorator Patterns



```
public abstract class Auto
{
    public string ausstattung;

    public virtual string getAusstattung()
    {
        return ausstattung;
    }

    public abstract double getPreis();
}
```

```
public class Porsche : Auto
{
    public Porsche()
    {
        ausstattung = " Porsche 911 ";
    }

    public override double getPreis()
    {
        return 135000;
    }
}
```

```
class Audi : Auto
{
    public Audi()
    {
        ausstattung = " Audi R8 ";
    }

    public override double getPreis()
    {
        return 190000;
    }
}
```

```
public abstract class Dekorierer : Auto
{
    public abstract override string getAusstattung();
}
```

```

public class Ledersitze : Dekorierer
{
    Auto auto;
    public Ledersitze(Auto my_AutoModel)
    {
        auto = my_AutoModel;
    }
    public override string getAustattung()
    {
        return auto.getAustattung() + " mit Ledersitze ";
    }
    public override double getPreis()
    {
        return auto.getPreis()+4500.50;
    }
}

```

```

public class Radio : Dekorierer
{
    Auto auto;
    public Radio(Auto my_AutoModel)
    {
        auto = my_AutoModel;
    }
    public override string getAustattung()
    {
        return auto.getAustattung() + " ein Super Radio ";
    }
    public override double getPreis()
    {
        return auto.getPreis() + 1520.50;
    }
}

```

```

static void Main(string[] args)
{
    Auto my_Porsche_911 = new Porsche();

    my_Porsche_911 = new Ledersitze(my_Porsche_911);

    my_Porsche_911 = new Radio(my_Porsche_911);
    Console.WriteLine("Ihre Porsche ist {0} und kostet {1} Euro", my_Porsche_911.getAustattung(), my_Porsche_911.getPreis());
    Console.WriteLine("-----");

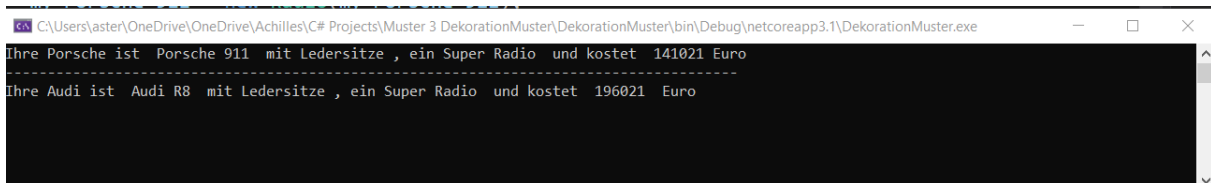
    Auto my_Audi_R8 = new Audi();

    my_Audi_R8 = new Ledersitze(my_Audi_R8);

    my_Audi_R8 = new Radio(my_Audi_R8);

    Console.WriteLine("Ihre Audi ist {0} und kostet {1} Euro", my_Audi_R8.getAustattung(),my_Audi_R8.getPreis());
    Console.ReadKey();
}

```



```

C:\Users\aster\OneDrive\OneDrive\Achilles\C# Projects\Muster 3 DekorationsMuster\DekorationsMuster\bin\Debug\netcoreapp3.1\DekorationsMuster.exe
Ihre Porsche ist  Porsche 911  mit Ledersitze , ein Super Radio  und kostet  141021 Euro
-----
Ihre Audi ist  Audi R8  mit Ledersitze , ein Super Radio  und kostet  196021 Euro

```