

Im 1. Band *Grundlagen* haben wir bereits Entwurfsmuster für die Entwicklung von Algorithmen kennengelernt. Das Ziel bestand darin, für bestimmte Problemklassen allgemeine Muster zu konstruieren, um diese geeignet für ein konkretes Problem anzuwenden. Diese Muster sind ein abstrakter Programmrahmen, der dann für das jeweilige Problem nur noch ausgefüllt werden muss. Wir haben diese allgemeinen Lösungsmethoden zunächst vorgestellt und sie anschließend an einem einfachen Beispiel gezeigt.

Im Bereich der objektorientierten Programmierung gehen wir nun auf ähnliche Weise vor. In der Softwareentwicklung gibt es für viele Entwicklungsschritte spezielle Muster zum Entwurf und der Implementierung von Programmen. Diese Entwurfsmuster sind bewährte Lösungsvorschläge für bestimmte Problemstellungen, die bereits in vielen Anwendungen erfolgreich verwendet wurden.

Ähnlich wie algorithmische Muster sind auch die objektorientierten Entwurfsmuster nicht auf eine bestimmte Programmiersprache beschränkt. Beim Erlernen der allgemeinen Prinzipien ist es jedoch von großem Vorteil, diese Muster an einem konkreten Beispiel zu implementieren. Genau diesen Weg werden wir in diesem Kapitel mithilfe der Programmiersprache Java verfolgen.

Das Ziel von objektorientierten Entwurfsmustern für den Softwareentwurf ist, bereits gewonnene Erkenntnisse wiederverwendbar zu machen, die Flexibilität der Softwarearchitektur zu erhöhen sowie einen verständlichen, erweiterbaren und wartungsfreundlichen Code zu schreiben. Durch das hohe Abstraktionsniveau dieser Entwurfsmuster lässt sich damit auch die Entwicklung großer Softwarepakete zielsicher und effizient planen.

In diesem Kapitel werden wir zunächst in einem einführenden Beispiel die Grundprinzipien der objektorientierten Programmierung wiederholen. Im Anschluss daran stellen wir zentrale und weiterführende Konzepte der objektorientierten Programmierung in Java vor. Die objektorientierten Entwurfsmuster beruhen vor allem auf den zwei Konzepten der Vererbung und der Polymorphie. Mit diesen Konzepten stellen wir die zentralen Entwurfsmuster objektorientierter Programmierung mithilfe von Beispielen in Java vor.

## 2.1 Einführendes Beispiel

Als einführendes Beispiel betrachten wir eine Aufgabenstellung zur Klassifizierung von Produkten in Form von Schrauben.

Ein Hersteller von Schrauben will seine Produkte nach dem folgenden Schema einordnen:

- Schrauben mit einem Durchmesser bis zu 3 mm und einer Länge bis zu 20 mm haben den Preis EUR 0,30.
- Schrauben mit einem Durchmesser zwischen 3 und 5 mm und einer Länge zwischen 20 und 30 mm haben den Preis EUR 0,40.
- Schrauben mit einem Durchmesser zwischen 5 und 6 mm und einer Länge zwischen 20 und 30 mm haben den Preis EUR 0,60.
- Schrauben mit einem Durchmesser zwischen 6 und 15 mm und einer Länge zwischen 30 und 50 mm haben den Preis EUR 0,80.
- Schrauben mit einem Durchmesser zwischen 15 und 20 mm und einer Länge zwischen 30 und 50 mm haben den Preis EUR 0,90.

Die Aufgabe besteht darin ein Programm zu erstellen, das den richtigen Preis einer Schraube ermittelt, wenn Durchmesser und Länge eingegeben werden. Sollte eine Schraube keiner der oben beschriebenen Kategorien angehören, soll die Meldung „Unbekannter Schraubentyp“ ausgegeben werden.

Wir stellen im Folgenden zwei Varianten zur Lösung dieser Problemstellung vor. Der erste Fall ist die herkömmliche strukturierte Programmierung, die wir bereits in vielen Fällen erfolgreich angewandt haben. Mit diesem Programmierparadigma können wir schnell eine Lösung für eine gegebene Problemstellung finden. Wesentlicher Nachteil dieses Ansatzes ist es, dass die Struktur der Daten bekannt sein muss und kein logischer Zusammenhang zwischen Daten und den darauf anwendbaren Operationen besteht. Das Ergebnis ist in vielen Fällen ein starrer, unflexibler und schwer erweiterbarer Code.

Im zweiten Fall verwenden wir einen objektorientierten Ansatz, bei dem zusammengehörige Daten und die darauf arbeitende Programmlogik zu einer Einheit zusammengefasst werden. Dieser Programmcode verlangt auf den ersten Blick vom Programmierer etwas mehr Arbeit ab, vor allem da ein Konzept von mehreren Klassen erstellt werden muss. Der große Vorteil davon ist, dass wir einen gut erweiterbaren, modularen und für andere ähnliche Problemstellungen wiederverwendbaren Programmcode erhalten.

### Strukturierte Programmierung

Die einfachste und kürzeste Implementierung erfolgt über eine Methode `bestimmePreis` mithilfe von `if-else`-Konstrukten:

```

public class Schrauben_Strukturiert
{
    // Bestimmung des Preises der Schrauben
    public static void bestimmePreis(double d, double l)
    {
        if (d<=3 && l<=20)
            System.out.printf("Der Preis beträgt 0.3 Euro.");
        else if (d>3 && d<=5 && l> 20 && l<=30)
            System.out.printf("Der Preis beträgt 0.4 Euro.");
        else if (d>5 && d<=6 && l>20 && l<=30)
            System.out.printf("Der Preis beträgt 0.6 Euro.");
        else if (d>6 && d<=15 && l>30 && l<=50)
            System.out.printf("Der Preis beträgt 0.8 Euro.");
        else if (d>15 && d<=20 && l>30 && l<=50)
            System.out.printf("Der Preis beträgt 0.9 Euro.");
        else
            System.out.println("Unbekannter Typ");
    }

    public static void main(String[] args)
    {
        // -----
        // --- 1. Durchmesser
        double durchmesser = 5.5;

        // --- 2. Laenge
        double laenge = 23;

        // -----
        bestimmePreis(durchmesser, laenge);
    }
}

```

Dieser Programmtext ist zwar schnell geschrieben, hat jedoch, wie wir gleich sehen werden, zahlreiche Nachteile.

### Objektorientierte Programmierung

In der realen Welt gibt es eine Preisgruppe für die Schraubentypen. Diese Preisgruppe bilden wir nun mit einer Klasse `Preisgruppe` nach. Die Attribute sind neben dem Preis die Unter- und Obergrenzen für den Durchmesser und die Länge.

```

public class Preisgruppe
{
    double uD, oD; // unterer und oberer Durchmesser
    double uL, oL; // untere und obere Länge
    double preis; // Preis
}

```

```

// Konstruktor zur Initialisierung einer Preisgruppe
public Preisgruppe(double uD, double oD, double uL, double oL, double preis)
{
    this.uD = uD;
    this.oD = oD;
    this.uL = uL;
    this.oL = oL;
    this.preis = preis;
}

// Prüfung des Preises des Produktes mit Durchmesser d und Laenge l
public boolean pruefePreis(double d, double l)

// Rueckgabe des Preises
public double getPreis()
}

```

Die Prüfung, ob ein gegebener Durchmesser und eine gegebene Länge zu einer Preisgruppe gehören, erfolgt mit der Methode `pruefePreis`. Diese bekommt die zwei Parameter des zu testenden Durchmessers und der zu testenden Länge übergeben. Anschließend vergleicht die Methode diese beiden Parameter mit ihren eigenen Attributen.

```

// Prüfung des Preises des Produktes mit Durchmesser d und Laenge l
public boolean pruefePreis(double d, double l)
{
    if (d > uD && d <= oD && l > uL && l <= oL)
        return true;
    else
        return false;
}

```

Diese Methode muss nur ein einziges Mal geschrieben werden, da einzelne Preisgruppenobjekte mit unterschiedlichen Parametern erzeugt worden sind. Auf die verschachtelten `if-else`-Schleifen können wir damit verzichten. Liegt die übergebene Größe im Bereich der Preisgruppe, dann gibt die Methode `true` zurück, andernfalls `false`.

Die Methode `getPreis` gibt den zugehörigen Preis als `double`-Wert zurück. Änderungen wie ein Steueraufschlag können leicht in dieser Methode noch ergänzt werden.

```

public double getPreis()
{
    return preis;
}

```

In der Klasse `TestSchrauben` definieren wir nun die 5 Preisgruppenobjekte in Form eines Feldes vom Typ `Preisgruppe`. Anschließend gehen wir der Reihe nach durch alle Preisgruppen durch und vergleichen, ob der zu testende Durchmesser und die zu testende Länge vorhanden sind. Am Ende wird das Ergebnis ausgegeben.

```

public class Schrauben
{
    public static void main(String[] args)
    {
        // -----
        // --- 1. Durchmesser
        double durchmesser = 5.5;

        // --- 2. Laenge
        double laenge = 23;

        // -----
        // --- 1. Definition der Preisgruppen
        Preisgruppe pg[] = new Preisgruppe[5];
        pg[0] = new Preisgruppe(0, 3, 0, 20, 0.30);
        pg[1] = new Preisgruppe(3, 5, 20, 30, 0.40);
        pg[2] = new Preisgruppe(5, 6, 20, 30, 0.60);
        pg[3] = new Preisgruppe(6, 15, 30, 50, 0.80);
        pg[4] = new Preisgruppe(15, 20, 30, 50, 0.90);

        // --- 2. Bestimmung des aktuellen Preises
        double preis = -1;
        for (int i = 0; i < pg.length; i++)
            if (pg[i].pruefePreis(durchmesser, laenge))
            {
                preis = pg[i].getPreis();
                break;
            }

        // --- 3. Ausgabe
        if (preis >= 0)
            System.out.printf("Der Preis beträgt %1.2f Euro.", preis);
        else
            System.out.printf("Der Preis ist unbekannt.");
    }
}

```

### Ausgabe

Der Preis beträgt 0,60 Euro.

Die objektorientierte Implementierung ist auf den ersten Blick etwas umfangreicher, dafür besitzt sie zahlreiche Vorteile. Die beiden Klassen sind übersichtlicher und gut nachvollziehbar. Änderungen wie beispielsweise durch Hinzufügen oder Weglassen von Gleichheitszeichen in der `if`-Abfrage benötigen nur 1 Änderung, statt 5 wie in der obigen Implementierung. Damit können problemlos neue Preisgruppen ergänzt werden, ohne Programmcode gegebenenfalls wieder fehlerhaft zu machen. Aufgrund der Tatsache, dass das Programm in zwei Teilklassen zerlegt ist, kann die Klasse `Preisgruppe` für andere Produkte leicht wiederverwendet werden.

## 2.2 Objektorientierte Analyse

Die Grundlage der objektorientierten Programmierung eines Softwaresystems ist die Modellierung der Aufgabenstellung durch kooperierende Objekte. Objekte sind besondere Daten- und Programmstrukturen, die Eigenschaften (Attribute) und Verhaltensweisen (Methoden) besitzen. Bei der objektorientierten Analyse sind die zu modellierenden Objekte zu finden, zu organisieren und zu beschreiben. Die einzelnen Schritte lassen sich in der folgenden Reihenfolge darstellen:

### 1. Finden der Objekte

In der gegebenen Problemstellung des zu modellierenden Softwaresystems sind die darin enthaltenen Objekte zu finden. Diese Objekte beschreiben eine Gruppe von interagierenden Elementen, um damit reale Objekte wie Autos, Kunden, Aufträge oder Artikel direkt in Software zu modellieren.

### 2. Organisation der Objekte

Bei einer großen Anzahl von beteiligten Objekten werden die zusammengehörigen Objekte in Gruppen zusammengesetzt. Diese Zusammenstellung ergibt sich aus den Beziehungen der einzelnen Objekte zueinander. Beispielsweise können Objekte andere Objekte enthalten (z. B. Bestellsystem enthält Artikel, Maschine enthält Komponenten). Die Gruppenzusammengehörigkeit ist umso größer, je mehr Beziehungen zwischen einzelnen Objekten bestehen.

### 3. Interaktion der Objekte

Die Interaktion bzw. Assoziation zweier Objekte beschreibt die Beziehung zwischen zwei Objekten. Die Aggregation beschreibt die Zusammensetzung eines Objekts aus anderen Objekten. Die Komposition ist ein Spezialfall einer Aggregation, bei der Abhängigkeiten zwischen den Objekten in der Form bestehen, dass ein beschriebenes Objekt nur durch gewisse Teilobjekte existiert (z. B. Maschine besteht aus Teilen, Container besteht aus Behältern, Behälter besteht aus Gegenständen).

### 4. Beschreibung der Attribute der Objekte

Die Attribute sind die individuellen Eigenschaften zur Definition des aktuellen Zustands des Objekts. Das Attribut ist ein Datenelement einer Klasse, das in allen Objekten vorhanden ist (z. B. Farbe eines Autos, Name eines Artikels, Länge eines Behälters).

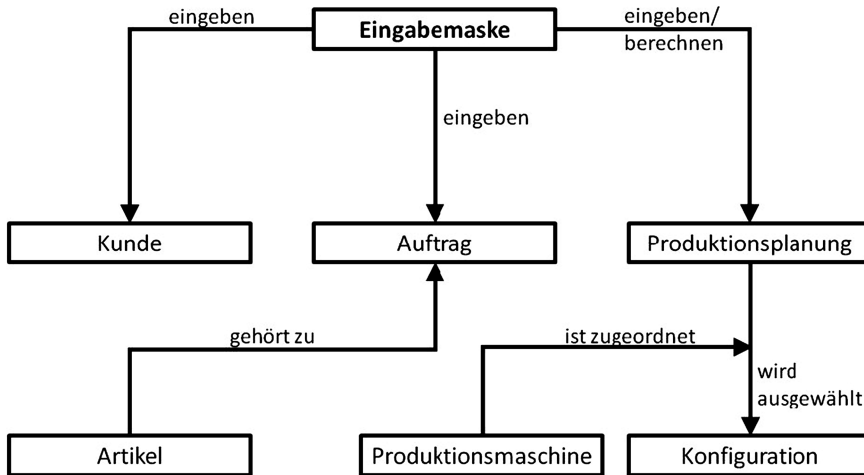
### 5. Beschreibung des Verhaltens der Objekte

Das Verhalten eines Objekts wird durch Methoden innerhalb einer Klasse definiert, die jeweils auf einem Objekt dieser Klasse operieren. Die Methoden definieren dabei eine Aufgabe in Form von Operationen auf der Instanz einer Klasse (z. B. Algorithmus steuert Maschine, Kalkulation berechnet Preis eines Artikels).

Das Ergebnis dieser Analyse ist die obige Beschreibung sowie die Erstellung von Abbildungen mit den einzelnen Klassen (Kästchen) und deren Beziehung untereinander (Linien und Text).

**Beispiel 2.1** Wir modellieren eine Software für die Produktionsplanung einer Firma. Das Softwaresystem besteht aus einer Eingabemaske, bei dem der Kunde die Auftragsdaten und die Produktionsplanungsdaten eingeben muss. Die Auftragsdaten bestehen aus den einzelnen Artikeln und die Produktionsplanungsdaten aus der Produktionsmaschine und deren Konfiguration.

1. **Finden der Objekte:** Eingabemaske, Kunde, Auftrag, Artikel, Produktionsplanung, Produktionsmaschine, Konfiguration
2. **Organisation der Objekte:** Die Objekte werden in 4 Gruppen aufgeteilt:
  - (a) Eingabemaske
  - (b) Kunde
  - (c) Auftrag, Artikel
  - (d) Produktionsplanung, Produktionsmaschine, Konfiguration
3. **Interaktion der Objekte:**
  - Kunde, Auftrag und Produktionsplanung werden über die Eingabemaske eingegeben.
  - Artikel gehört zu dem Auftrag
  - Produktionsmaschine ist Produktionsplanung zugeordnet
  - Konfiguration wird von Produktionsplanung ausgewählt
4. **Beschreibung der Attribute der Objekte:**
  - Kunde: Name, Adresse, Kundennummer
  - Artikel: Nummer, Name, Preis, Anzahl
  - Auftrag: Array von Artikeln
  - Konfiguration: Maschinenparameter
  - Produktionsmaschine: Maschinenbezeichner, Laufzeit
  - Produktionsplanung: Produktionsmaschine, Konfiguration
5. **Beschreibung des Verhaltens der Objekte:**
  - Kunde: getKundennummer, getKunde, ...
  - Auftrag: getAuftragswert, getAnzahlauftraege, ...
  - Artikel: getArtikelnummer, getArtikelanzahl, ...
  - Produktionsplanung: getKonfiguration, ...
  - Konfiguration: getParameter, setParameter, ...
  - Produktionsmaschine: getMaschinenbezeichner, ...



**Abb. 2.1** Objektorientierte Analyse einer Planungssoftware

Die Abb. 2.1 zeigt ein Übersichtsdiagramm der objektorientierten Analyse dieser Planungssoftware.

## 2.3 Objektorientierte Konzepte

Die objektorientierte Programmierung ist sehr gut geeignet, umfangreiche Programme in eine Gruppe von interagierenden Objekten aufzuteilen, um damit eine ganze Reihe von wichtigen Prinzipien der Programmierung zu erfüllen:

- **Flexibilität:** Klassen sind nicht starr auf ein vorliegendes Problem angepasst, sodass Änderungen problemlos möglich sind.
- **Wiederverwendbarkeit:** Einzelne Module können in neuen Anwendungen weiterverwendet werden, sodass die Kosten für die Neuentwicklung neuer Systeme sinken.
- **Erweiterbarkeit:** System kann gut um neue Funktionalität erweitert werden, um beispielsweise neue Bibliotheksklassen optimal hinzuzufügen.
- **Veränderbarkeit:** Veränderungen an einer bestimmten Stelle wirken sich nicht auf andere Teile des Systems aus.
- **Wartbarkeit:** System besteht nur aus wenigen Abhängigkeiten, sodass es über die Zeit kostengünstig weiterentwickelbar ist.
- **Redundanzfreiheit:** System besitzt keine mehrfache Verwendung von identischem oder ähnlichem Code.
- **Verständlichkeit:** System ist logisch durch einen gut lesbaren Code aufgebaut, damit Änderungen schnell und sicher umgesetzt werden können.



Das Ziel der objektorientierten Programmierung ist, die Abhängigkeiten der verschiedenen Module zu minimieren, die Korrektheit sicherzustellen sowie ein verständliches und gut dokumentiertes Programmpaket zu schaffen. Alle in diesem Kapitel vorgestellten Konzepte sind dazu da, diese Prinzipien in die Praxis umzusetzen.

Im Rahmen der objektorientierten Programmierung bezeichnet der Begriff des Refactoring die Veränderung des Programmcodes, ohne die Funktionalität zu erweitern. Das Refactoring eines Codes wird durchgeführt, um die Erweiterbarkeit und die Wartbarkeit sicherzustellen. Bei größeren Programmpaketen ist es oftmals sehr sinnvoll, in gewissen Entwicklungsschritten suboptimale Codeteile zu identifizieren, aufzuräumen und für eventuell weitere Erweiterungen vorzubereiten.

### 2.3.1 Innere Klassen

In Java ist es möglich eine Klasse in eine andere Klasse mit hineinzunehmen, um beispielsweise nur lokale Typdeklarationen zu definieren, die keine weitere Sichtbarkeit benötigen. Wir stellen nun einige verschiedene Varianten zur Definition von inneren Klassen vor. Innere Klassen sind oft sehr nützlich für den Entwurf von Hilfsdatenstrukturen, die nur in speziellen Klassen benötigt werden. Ebenso finden Sie in einigen Entwurfsmustern der Softwaretechnik Anwendung.

**Mitgliedsklasse** Eine Mitgliedsklasse einer äußeren Klasse ist ähnlich einem Attribut und wird wie folgt definiert:

```
class AeussereKlasse
{
    class InnereKlasse
    {
        ...
    }
}
```

Um ein Objekt der inneren Klasse zu erzeugen, muss ein Objekt der äußeren Klasse existieren:

```
AeussereKlasse out = new AeussereKlasse();
InnereKlasse in = out.new InnereKlasse();
```

Zum Zugriff von der inneren Klasse auf eine Variable `var` der äußeren Klasse, die von inneren Klassen überdeckt wird, verwenden wir den Befehl `AeussereKlasse.this.var`.

**Beispiel 2.2** Wir definieren eine äußere und eine innere Klasse mit einer Methode `ausgabe`:

```
public class AeussereKlasse
{
    String name = "Aussen";

    class InnereKlasse
    {
        String name = "Innen";
        public void ausgabe()
        {
            System.out.println(name);
            System.out.println(AeussereKlasse.this.name);
        }
    }

    public static void main( String[] args )
    {
        AeussereKlasse out = new AeussereKlasse();
        InnereKlasse in = out.new InnereKlasse();
        in.ausgabe();
    }
}
```

### Ausgabe

```
Innen
Aussen
```

### Allgemeine Erklärung

Zuerst wird ein Objekt `out` der äußeren Klasse `AeussereKlasse` angelegt. Anschließend kann ein Objekt `in` der inneren Klasse `InnereKlasse` mithilfe der Variable `out` definiert werden. Mithilfe der Objektvariable `in` kann dann auf die Methode `ausgabe` der inneren Klasse zugegriffen werden.

**Statische innere Klasse** Bei einer statischen inneren Klasse wird eine innere Klasse als eine Art statische Eigenschaft in der äußeren Klasse definiert. Der Unterschied zur Mitgliedsklasse ist das Schlüsselwort `static`:

```
class AeussereKlasse
{
    static class InnereKlasse
    {
        ...
    }
}
```

Programmieren für Ingenieure und  
Naturwissenschaftler  
Algorithmen und Programmiertechniken

Dörn, S.

2017, XIII, 376 S. 128 Abb., 22 Abb. in Farbe., Softcover

ISBN: 978-3-662-54175-3