# Programmierung 3

im Wintersemester 2024/25

## Praktikumsaufgabe 3

Technische Schulden und Clean Code

29. Oktober 2024

Mit diesen Aufgaben wandeln Sie "schmutzigen" Code in "sauberen" Code um. Durch dieses sogenannte Refactoring reduzieren Sie die technische Schuld im bestehenden Code, um die interne Qualität des Codes zu erhöhen. Um dies zu erreichen, verbessern Sie zunächst in Aufgabe 1 die Lesbarkeit und Verständlichkeit, wenden dann in Aufgabe 2 das DRY-Prinzip (Don't Repeat Yourself) und schließlich in Aufgabe 3 das SRP-Prinzip (Single Responsibility Principle) an.

In Aufgabe 4 haben Sie schließlich die Möglichkeit, moderne Sprachfeatures wie `Optional` und **`var`** zu nutzen, um den Code weiter zu verbessern.

Arbeiten Sie bei der Lösung der Aufgaben in einem Team von 3 bis 4 Personen zusammen, so dass Sie die Aufgaben kollaborativ lösen und Ihre Ergebnisse untereinander diskutieren können. Lösen Sie dabei die Programmieraufgaben im Pair Programming und bearbeiten Sie die anderen Aufgaben in einer für Sie geeigneten Weise.

# 1 Lesbarkeit und Verständlichkeit

**Lernziele**

- ☐ Sie brechen tief verschachtelten Code in kleinere, verständlichere Methoden auf.
- ☐ Sie vereinfachen die Logik durch die Einführung einer frühen Fehlerbehandlung mit Hilfe des Return Early Pattern.
- ☐ Sie verwenden prägnante und beschreibende Namen, um die Absicht des Codes klar zu vermitteln und damit die Verständlichkeit zu fördern.
- ☐ Sie fördern die Lesbarkeit des Codes durch einheitliche Formatierung und Einrückung.

Lesbarer und verständlicher Code ist eine Grundvoraussetzung für wartbaren und erweiterbaren Code. So kann z.B. ein schlecht strukturierter Code dazu führen, dass Entwickler:innen länger brauchen, um ihn zu verstehen und zu ändern, was zu technischen Schulden führt. Die folgende Aufgabe gibt Ihnen die Möglichkeit, Ihr Wissen anzuwenden und die Lesbarkeit von "smelly" Code durch sinnvolles Refactoring zu verbessern.

## 1.1 Ausgangssituation

Sie finden den folgenden Java-Code in einem bestehenden Projekt. Er ist funktional vollständig, aber schwer verständlich, was die Wartung und Weiterentwicklung erschwert. Der Code berechnet die Gesamtkosten einer Bestellung und gewährt einen Rabatt in Abhängigkeit vom Gesamtwert.

```
1  public class OrderManager {
2
3    public double calc(Order ord) {
4      double x = 0.0;
5
6      for (Item i : ord.getItems()) {
7        if (i != null && i.getQuantity() > 0) {
8          double y = i.getPrice() * i.getQuantity();
9          x += y;
10       } else {
11         System.out.println("Invalid item in order");
12       }
```

```
13      }
14
15      if (x > 500) {
16        x *= 0.9;
17        System.out.println("Applied 10% discount for high-value
            ↪  order");
18      } else if (x > 100) {
19        x *= 0.95;
20        System.out.println("Applied 5% discount for medium-value
            ↪  order");
21      }
22
23      if (ord.getShippingCost() != null) {
24        x += ord.getShippingCost();
25      }
26
27      return x;
28    }
29  }
```

## 1.2 Refactoring

Führen Sie ein geeignetes Refactoring durch, um den Code lesbarer und wartbarer zu machen. Dabei darf die ursprüngliche Funktionalität nicht verändert werden. Verbessern Sie die interne Qualität des Codes durch folgende Maßnahmen:

**Verschachtelungstiefe:** Vermeiden Sie zu tiefe Verschachtelungen.

**Separation of Concerns:** Extrahieren Sie einzelne Aufgaben in eigene, klar benannte Methoden.

**Namensgebung:** Überlegen Sie, ob alle Methoden-, Parameter- und Variablennamen den Zweck und den Kontext des jeweiligen Elements widerspiegeln. Ersetzen Sie unklare Namen durch aussagekräftigere, indem Sie prägnante Namen wählen, die die Funktion klar zum Ausdruck bringen.

**Fehlerbehandlung:** Fangen Sie ungültige Eingaben (z.B. Nullwerte oder negative Mengen) so früh wie möglich ab, um die Weiterverarbeitung zu vereinfachen.

**Formatierung:** Da Layout und Einrückungen die Qualität des Codes ebenfalls beeinflussen, sollten Sie darauf achten, dass Ihr Code einheitlich formatiert ist, um die Lesbarkeit zu erleichtern.

## 1.3 Reflexion

Begründen Sie abschließend im Team, warum Sie bestimmte Refactorings durchgeführt haben und diskutieren Sie, wie Ihre Maßnahmen die Lesbarkeit und Verständlichkeit des Codes verbessert haben. Beantworten Sie dabei auch die folgenden Fragen:

- Welche konkreten Änderungen haben am meisten zur Verbesserung der Lesbarkeit und Verständlichkeit beigetragen?
- Welche Schwierigkeiten gab es bei der Wahl aussagekräftiger Namen? Wie konnten sie gelöst werden?

# 2 DRY (Don't Repeat Yourself)

**Lernziele**

- ☐ Sie identifizieren sich wiederholende Logiken und erkennen Muster für eine effiziente Strukturierung
- ☐ Sie fassen redundanten Code durch sinnvolle Methoden und Konstanten zusammen und verbessern so die Wartbarkeit und Erweiterbarkeit.
- ☐ Sie setzen das DRY-Prinzip gezielt ein, um den Code modular und anpassungsfähig zu gestalten.

**Strecklernziele**

- ☐ Sie führen Enums ein, um die Robustheit und Lesbarkeit des Codes zu verbessern.

Das DRY-Prinzip (Don't Repeat Yourself) ist eine der zentralen Säulen von Clean Code. Es besagt, dass Redundanzen im Code vermieden werden sollen, um die Wartbarkeit und Erweiterbarkeit zu verbessern. Doppelte Logik führt zu unnötiger Komplexität und erhöht die Fehlerwahrscheinlichkeit, da Änderungen mehrfach durchgeführt werden müssen. In dieser Aufgabe wenden Sie das DRY-Prinzip an, um bestehenden „smelly" Code zu bereinigen.

## 2.1 Ausgangssituation

Der folgende Java-Code dient zur Berechnung von Rabatten für verschiedene Kundengruppen. Leider enthält der Code mehrere redundante Logikbereiche, die ihn schwer verständlich und fehleranfällig machen. Ziel ist es, diese Redundanzen durch eine geeignete Strukturierung des Codes zu beseitigen.

```
1  public class DiscountCalculator {
2
3    public double calculateDiscount(double amount, String
       ↪  customerType) {
4      double discount = 0.0;
5
6      if (customerType.equals("Regular")) {
7        if (amount > 100) {
```

```
8           discount = amount * 0.05;
9         } else if (amount > 500) {
10          discount = amount * 0.1;
11        } else if (amount > 1000) {
12          discount = amount * 0.15;
13        }
14      } else if (customerType.equals("VIP")) {
15        if (amount > 100) {
16          discount = amount * 0.1;
17        } else if (amount > 500) {
18          discount = amount * 0.15;
19        } else if (amount > 1000) {
20          discount = amount * 0.2;
21        }
22      } else if (customerType.equals("Premium")) {
23        if (amount > 100) {
24          discount = amount * 0.12;
25        } else if (amount > 500) {
26          discount = amount * 0.17;
27        } else if (amount > 1000) {
28          discount = amount * 0.25;
29        }
30      }
31
32      return discount;
33    }
34  }
```

## 2.2 Refactoring

Führen Sie ein weiteres Refactoring durch und konzentrieren Sie sich dabei insbesondere auf folgende Maßnahmen:

**Redundanz reduzieren:** Überarbeiten Sie den Code, um doppelte Logik zu vermeiden. Identifizieren Sie Muster und fassen Sie ähnliche Teile zusammen, um Wiederholungen zu

reduzieren.

Überlegen Sie, ob Sie z.B. zusätzliche Methoden verwenden können, um redundante Bedingungsblöcke zusammenzufassen.

Denken Sie auch daran, den Code so zu strukturieren, dass zukünftige Erweiterungen oder Anpassungen (z.B. neue Kundentypen) möglichst wenig Änderungen erfordern.

**Klare Konstanten und Methoden:** Erstellen Sie Hilfsmethoden und/oder Konstanten, die die Berechnung und Strukturierung des Rabatts klarer machen. Verwenden Sie stets aussagekräftige Namen, die den Zweck des Codes verdeutlichen. Beispielsweise können alle Rabattsätze als Konstanten definiert werden, um ihre Bedeutung zu verdeutlichen und die Anpassung des Codes zu erleichtern.

**Enumeration:** Überlegen Sie, ob die verschiedenen Kundentypen und die entsprechenden Rabattstufen als Enums dargestellt werden können, um die Anzahl der Bedingungsblöcke zu reduzieren und die Struktur des Codes zu vereinheitlichen. Implementieren Sie einen entsprechenden Enum-Typ, der die verschiedenen Kundentypen bzw. Rabattstufen repräsentiert.

## 2.3 Reflexion

Begründen Sie abschließend im Team, warum Sie bestimmte Refactorings durchgeführt haben und diskutieren Sie, wie Ihre Maßnahmen zur Vermeidung von Redundanzen beigetragen haben. Beantworten Sie dabei auch die folgenden Fragen:

- Welche Strategien haben Sie angewandt, um Redundanzen zu vermeiden?
- Welches sind die Änderungen, die sich am stärksten auf die Lesbarkeit und Wartbarkeit des Codes ausgewirkt haben?
- Wie würde sich Ihre Lösung auf zukünftige Anpassungen, wie z.B. das Hinzufügen weiterer Kundentypen, auswirken?
- Welche Vorteile hat die enum-basierte Lösung gegenüber der aktuellen Implementierung

hinsichtlich der Erweiterbarkeit (z.B. durch Hinzufügen neuer Kundentypen)?

# 3 Single Responsibility Principle (SRP)

**Lernziele**

- ☐ Sie erkennen und identifizieren Mehrfachverantwortungen in einer Klasse.
- ☐ Sie extrahieren eigenständige Klassen und Methoden, die sich klar auf eine einzelne Aufgabe konzentrieren, um Mehrfachverantwortungen aufzulösen und damit die Wartbarkeit zu verbessern.
- ☐ Sie nutzen das Single Responsibility Principle gezielt, um eine modularere Struktur zu schaffen und die langfristige Erweiterbarkeit zu unterstützen.

Das Single Responsibility Principle (SRP) besagt, dass eine Klasse oder Methode nur eine einzige Verantwortung haben sollte, um die Verständlichkeit und Wartbarkeit zu erhöhen.[1] In dieser Aufgabe werden wir den bestehenden Code so refaktorisieren, dass jede Klasse oder Methode nur eine spezifische Aufgabe erfüllt.

## 3.1 Ausgangssituation

Der folgende Java-Code verwaltet die Bestellungen, berechnet den Gesamtbetrag, prüft Rabatte und versendet Bestätigungsnachrichten. Dies führt zu einer unübersichtlichen und schwer zu wartenden Struktur.

```
1  public class OrderProcessor {
2
3    public double processOrder(Order order) {
4      double total = 0.0;
5
6      for (Item item : order.getItems()) {
7        if (item != null && item.getQuantity() > 0) {
8          double itemCost = item.getPrice() *
             ↪  item.getQuantity();
9          total += itemCost;
10       } else {
11         System.out.println("Invalid item in order");
```

[1] Das SRP gehört zu den SOLID-Prinzipien, wie sie von Robert C. Martin formuliert wurden.

```
12          }
13      }
14
15      if (total > 500) {
16          total *= 0.9;
17          System.out.println("Applied 10% discount for high-value
             ↪  order");
18      }
19
20      if (order.getCustomer() != null) {
21          System.out.println("Sending confirmation to: " +
             ↪  order.getCustomer().getEmail());
22      }
23
24      return total;
25    }
26  }
```

## 3.2 Refactoring

Führen Sie ein weiteres Refactoring unter Anwendung des Single Responsibility Prinzips durch. Überarbeiten Sie den Code so, dass jede Klasse oder Methode nur eine spezifische Verantwortung hat. Verwenden Sie das SRP, um die verschiedenen Aufgaben (z.B. Berechnung der Gesamtkosten, Prüfung der Rabatte, Versenden der Bestätigung) auf verschiedene Methoden oder Klassen zu verteilen. Jede Klasse sollte dabei so konzipiert sein, dass sie isoliert für eine Aufgabe verwendet und angepasst werden kann.

**Verantwortlichkeiten:** Untersuchen Sie die einzelnen Verantwortlichkeiten in der Klasse `OrderProcessor` und überlegen Sie, ob jede dieser Verantwortlichkeiten isoliert betrachtet werden kann.

**Tipp:** Eine Klasse, die nur eine Aufgabe erfüllt, lässt sich leichter anpassen und testen. Welche Aufgaben werden derzeit innerhalb der Klasse `OrderProcessor` erfüllt? Sind diese Aufgaben so unterschiedlich, dass sie in getrennten Klassen übersichtlicher und wartbarer wären? Wie würde sich die Pflege des Codes ändern, wenn jede Klasse nur

eine Aufgabe erfüllen würde?

**Idee:** Erwägen Sie die Einrichtung separater Klassen für Aufgaben wie die Berechnung der Gesamtkosten und das Versenden der Bestätigung.

**Klassen und Methoden:** Verwenden Sie aussagekräftige Namen für neue Klassen und Methoden, die den Zweck und die Funktionalität eindeutig beschreiben. Da jede Klasse oder Methode eine klar definierte Verantwortung haben sollte, sollten die Namen entsprechend präzise sein.

## 3.3 Reflexion

Begründen Sie abschließend im Team, warum Sie bestimmte Refactorings durchgeführt haben und diskutieren Sie, wie Ihre Maßnahmen zur Einhaltung des SRP beigetragen haben. Beantworten Sie dabei auch die folgenden Fragen:

- Welche Verantwortlichkeiten haben Sie in Form welcher Aufgaben in separate Klassen oder Methoden ausgelagert und warum?
- Welche Änderungen hatten den größten Einfluss auf die Verbesserung der Lesbarkeit und Verständlichkeit des Codes?
- Wie wirkt sich die neue Struktur auf die Wartung und zukünftige Änderungen aus, wie z.B. das Hinzufügen zusätzlicher Geschäftslogik (beispielsweise neue Rabattregeln, zusätzliche Bestätigungsoptionen, . . . )?

# 4 Modernes Java

**Strecklernziele**

- ☐ Sie verwenden das Pfadfinderprinzip, um den Code zu bereinigen, den Sie ohnehin ändern müssen.
- ☐ Sie verwenden Sprachfeatures wie `Optional` und **`var`**, um bestehenden Code zu modernisieren.

Sie haben sich bereits mit modernen Sprachfeatures wie `Optional` und **`var`** vertraut gemacht. In dieser Aufgabe können Sie diese zur weiteren Verbesserung des Codes verwenden.

## 4.1 Ausgangssituation

Der Java-Code, den Sie refaktorisiert und damit "gesäubert" haben, wurde zu einer Zeit geschrieben, als Java 8 noch nicht verfügbar war. Ihr Team hat beschlossen, nach dem Pfadfinderprinzip vorzugehen und den Code an den Stellen zu modernisieren, an denen Sie ihn ohnehin ändern.

## 4.2 Refactoring

Führen Sie ein abschließendes Refactoring durch und fügen Sie `Optional` und **`var`** an den Stellen ein, an denen Sie es für angebracht halten.

## 4.3 Reflexion

Begründen Sie abschließend im Team, wo Sie wofür `Optional` und **`var`** eingesetzt haben. Beantworten Sie dabei auch folgende Fragen:

- Welche Vorteile haben Sie durch den Einsatz von `Optional` und **`var`** erzielt? Inwiefern hat dies die Lesbarkeit und Verständlichkeit des Codes verbessert?

- Welche Auswirkungen hat dies auf zukünftige Änderungen am Code?