Zusammenfassung: Gute Klassenentwürfe & nichtfunktionale Aspekte

Dieses Kapitel dreht sich nicht darum, was ein Programm tut, sondern wie gut es aufgebaut ist. Es geht um Entwurf, Qualität, Erweiterbarkeit und Wartbarkeit von Klassen.

◇ 1. Kopplung `Couplin`

Beschreibt Abhängigkeiten zwischen Klassen.

Ziel: möglichst lose Kopplung

```
→ Klassen sollen unabhängig voneinander funktionieren.
→ Änderungen in einer Klasse sollen möglichst wenige andere Klassen betreffen.
```

Beispiel *schlechte Kopplung*:

```
class Bestellung {
    Kunde kunde;           // direkte Abhängigkeit
    Adresse lieferAdresse; // noch eine direkte Abhängigkeit
}
```

Beispiel *bessere Kopplung*:

```
class Bestellung {
    private KundenInfo info;  // nur eine abstrakte Schnittstelle
}
```

2. Kohäsion `Cohesion`

Beschreibt, wie gut eine Klasse oder Methode eine klar definierte Aufgabe erfüllt.

Ziel: hohe Kohäsion

```
→ Ein Modul = eine Aufgabe.
→ Ein klarer Fokus = bessere Verständlichkeit und Wartbarkeit.
```

Beispiel *schlechte Kohäsion*:

```
class Rechner {
    void berechnePreis() { ... }
    void speichereDatei() { ... }      // gehört nicht hierhin
    void sendeEmail() { ... }          // auch nicht
}
```

Beispiel *gute Kohäsion*:

```
class PreisRechner {
double berechnePreis(...) { ... }
}
```

3. Kapselung `Encapsulation`

- Interne Details verbergen `private Felder`, `öffentliche Methoden`.
- Hilft, Kopplung zu reduzieren → führt zu besseren Entwürfen.

Beispiel:

```
class Konto {
    private double saldo;     // versteckt
    public void einzahlen(double betrag) { ... }
}
```

4. Entwurf nach Zuständigkeiten

- Jede `Klasse` erhält eine klare Aufgabe / Verantwortung.
- Hilft, gute `Kohäsion` zu erreichen.
- Frage beim Design:
  - „Welche `Klasse` sollte wofür zuständig sein?“

5. Code-Duplizierung

- Duplizierter Code = Warnsignal.
- Führt zu Fehlern, höherem Wartungsaufwand, schlechterem Design.
- Immer vermeiden → durch:
  - Methoden extrahieren
  - Klassen neu strukturieren
  - Abstraktion einführen

Beispiel für Duplizierung *schlecht*:

```
if (alter >= 18) return true;
    else return false;
```

In beiden Methoden wiederholt → sollte ausgelagert werden.

6. Refactoring

Bestehenden Code restrukturieren, ohne das Verhalten zu ändern.

- Ziel:
  - `Kohäsion` verbessern
  - `Kopplung` reduzieren
  - Code lesbarer machen
  - zukünftige Änderungen vereinfachen
  - Typische Refactorings:
    - `Methoden` extrahieren
    - `Klassen` aufsplitten
    - redundante Felder entfernen

- bessere Namen vergeben

7. Implizite Kopplung

Versteckte Abhängigkeiten, z. B. über **globale Variablen**.

Besonders gefährlich, da schlecht sichtbar.

8. switch-Anweisung

Wählt aus mehreren Ausführungspfaden aus — Alternative zu vielen `if-else`.

```
switch (tag) {
    case "Montag": ...
    case "Dienstag": ...
}
```

Praktisches Beispiel

Schlechte Version `starke Kopplung`, `geringe Kohäsion`, `Duplizierung`:

```
class Bestellung {
Kunde kunde;
Produkt produkt;

    double berechnePreis() {
        double preis = produkt.getPreis();
        if (kunde.getAlter() > 65) preis *= 0.9;
        return preis;
    }

    void sendeBestaetigungsMail() {
        // hat hier nichts zu suchen
    }
}
```

Gute Version `lose Kopplung`, `klare Zuständigkeit`:

```
class PreisRechner {
    public double berechne(Produkt p, Kunde k) {
        double preis = p.getPreis();
        if (k.getAlter() > 65) preis *= 0.9;
            return preis;
        }
}

class Bestellung {
    private PreisRechner rechner = new PreisRechner();

    public double getEndpreis() {
        return rechner.berechne(produkt, kunde);
    }
}
```

🤯 Merksätze

Gute Software = lose Kopplung + hohe Kohäsion.

Kapselung schützt die `Klasse` und verbessert den Entwurf.

Zuständigkeiten klar verteilen → jede `Klasse` macht nur das, was sie soll.

Duplizierter Code ist ein Signal: „Bitte refactoren!“

Refactoring ist kein Luxus, sondern notwendig für langfristig guten Code.

Änderungen sollten lokal bleiben und nicht das ganze System beeinflussen.