

## **Web Engineering**

# **Häufige Fehler im Projekt vermeiden**

Adrian Herzog

# Teile die Zeit gut ein



*Es lohnt sich, das Projekt schon ein paar Tage vor der Frist abzuschliessen.*

Wer bis kurz vor Abgabefrist dran arbeitet, riskiert:

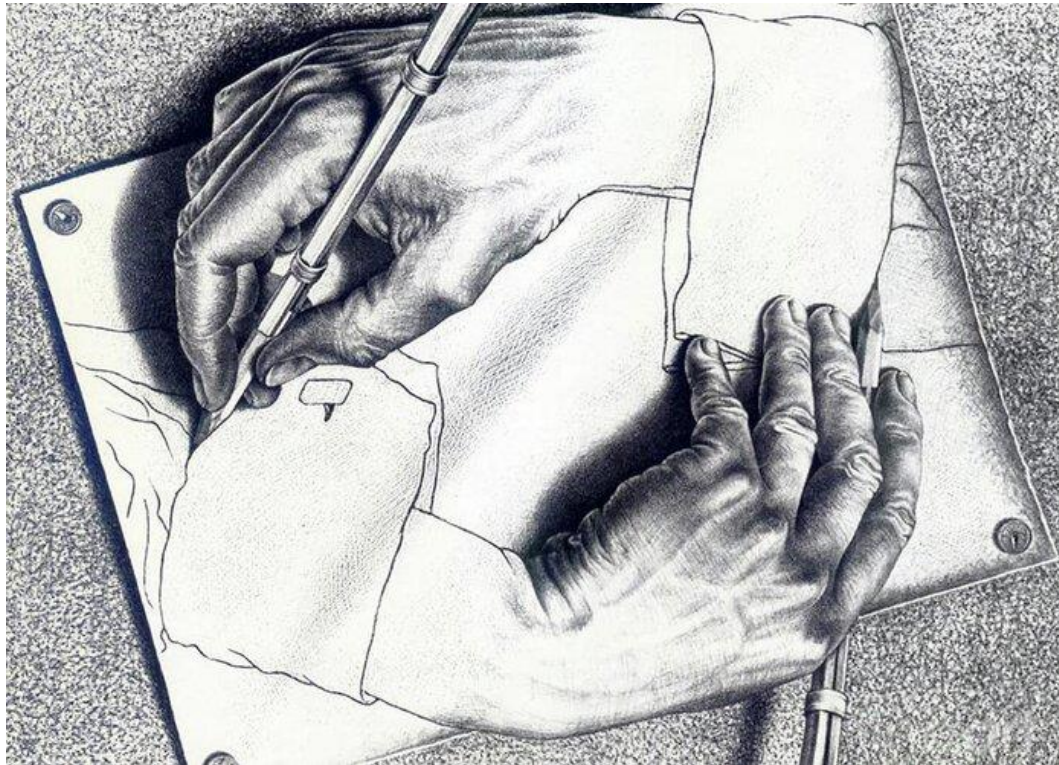
- Kompilierfehler
- Fehlschlagende Tests
- Defekte Features, weil nicht mehr getestet wurde

Wer schon etwas früher fertig ist:

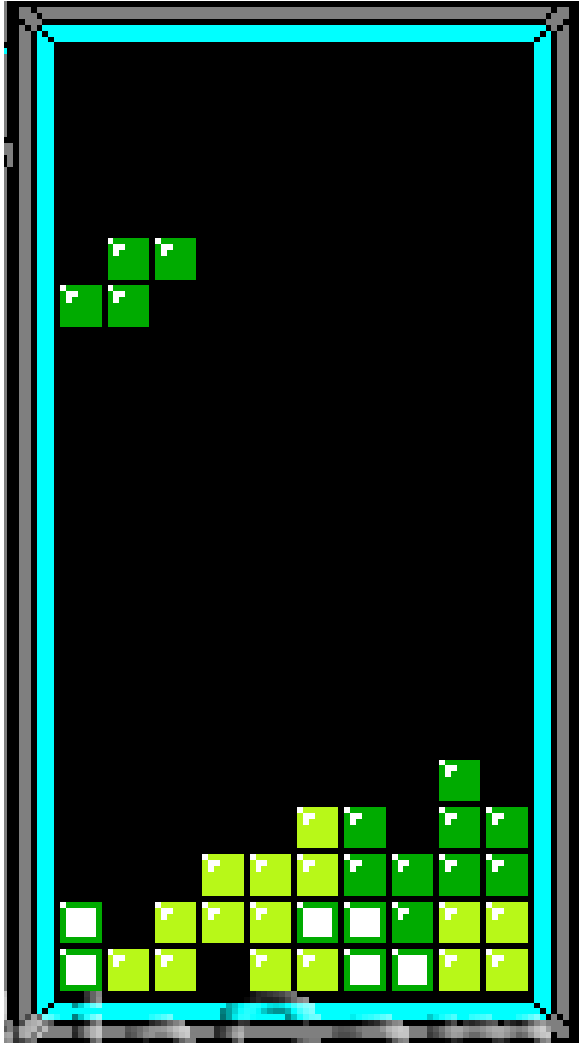
- Kann nochmals in Ruhe testen
- Kann nochmals die Bewertungskriterien prüfen
- Kann noch Feedback einholen (von Kollegen oder vom Dozenten)

# Prüfe die Bewertungskriterien

Bewerte dich am besten selbst. Wo würdest du dir wie viele Punkte geben? Wo würdest du dir Abzüge geben?



# Wähle die richtigen Lücken



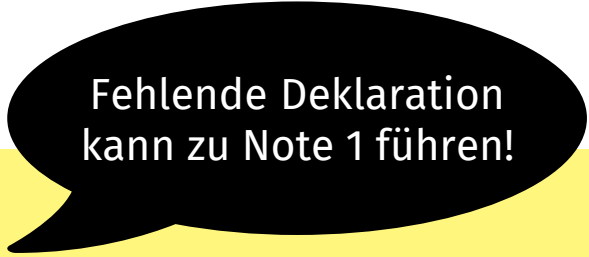
Wenn es knapp wird muss man konsequent priorisieren.

Hole möglichst die Punkte, welche du am effizientesten erreichen kannst. Aber wem sag ich das ;-)

# Deklariere externen Code

## Erlaubt:

- Kopieren von kleinen Codestücken von Stack Overflow, Tutorials, usw.
- Verwenden von Drittbibliotheken (siehe Erlaubte Technologien.xlsx)
- Diskutieren mit anderen Personen, Beratung durch Andere, Copilot und Chatbots



Fehlende Deklaration  
kann zu Note 1 führen!

## Kopierter oder generierter Code muss klar markiert sein!

- Kommentar in Code (oder Commit-Message), inkl. «EXTERNAL» und Quelle
- Ausnahme: single line code completion muss nicht deklariert werden

# Deklariere das Zusatzthema

- Auswahl eines Themas aus Liste von Ideen (siehe [Bewertungsraster.xlsx](#)).
- Alternativ sind auch eigene Ideen möglich, müssen aber von mir per Mail zusammen mit der Applikations-Idee genehmigt werden.
- Es wird nur ein Zusatzthema bewertet, auch wenn mehrere umgesetzt werden. ***Das zu bewertende Zusatzthema muss auf der Info-Seite der Applikation als solches angegeben werden.***

# Vermeide ineffiziente Queries

```
// EXTERNAL: Help from AI for this method and for inspiration of this method
@GetMapping
public String listWines(Model model) {
    List<Wine> wines = wineService.findAllWines();
    Map<Integer, Double> avgRatings = new HashMap<>();
    for (Wine w : wines) {
        Double avg = ratingService.calculateAverageRating(w.getId());
        if (avg == null) {
            avg = 0.0;
        }
        avgRatings.put(w.getId(), avg);
    }
    model.addAttribute("wines", wines);
    model.addAttribute("avgRatings", avgRatings);

    return "winelist";
}
```

```
public List<Rating> findAllByWineId(int wineId) {
    return ratingRepo.findAll()
        .stream()
        .filter(r -> r.getWine().getId() == wineId)
        .toList();
}

public Double calculateAverageRating(int wineId) {
    List<Rating> ratings = findAllByWineId(wineId);
    if (ratings.isEmpty()) {
        return null;
    }
    double sum = 0;
    for (Rating r : ratings) {
        sum += r.getRatingValue();
    }
    return sum / ratings.size();
}
```

# Vermeide ineffiziente Queries wirklich

findAll() mit nachträglichem Filtern ist fast immer falsch

```
public Optional<Recipe> findRecipeById(int id) {  
    return recipeRepository.findAll().stream()  
        .filter(r -> r.getId() == id ||  
            r.getExternalId() == id)  
        .findFirst();  
}
```



```
public Optional<Recipe> findRecipeById(int id) {  
    return recipeRepository.findById(id);  
}
```



# Gestaltung: einfach, aber sauber

- Fehlen offensichtliche Abstände oder sind sie zu gross / klein?
- Margin nicht global auf 0 setzen! Margin ist u.a. bei <p> absichtlich nicht 0.
- Gibt es eine konsistente Farbgestaltung mit einem einfachen Farbkonzept? (CSS Variablen können helfen)
- Nutze die “Gestaltgesetze” um Zusammengehörigkeit und Unterschiedlichkeit zu kommunizieren.
- Gibt es überflüssige Gestaltungselemente? (z.B. Überbleibsel, die mit KI generiert wurden?)
- Ist Umfang vom CSS-Code angemessen (ca. 100 bis 200 Zeilen)?

# E2E-Tests

- Verwende das Suffix IT, damit Maven den Test als Integration Test erkennt
- Vermeide es auf jeden Fall, eine fixe Zeit auf etwas zu warten
- Aktiviere “Implicit Waits” in Selenium (siehe Musterlösung E2E)
- Vermeide XPath-Ausdrücke als Selektoren
- Vergib explizite Test-IDs (z.B. data-test-id="submit-button") und verwende wenn immer möglich nur diese
- Test-Daten sollten nicht Teil der Page-Objects sein sondern von den Test-Methoden an die Page-Objects übergeben werden.

# Integration-Tests

- Verwende das Suffix IT, damit Maven den Test als Integration Test erkennt
- Ein Integration-Test muss mindestens zwei Dinge miteinander testen, z.B.:
  - Zwei Klassen
  - Controller zusammen mit Spring MVC
  - Service mit Repository

# Unit-Tests

- Tests für Methoden ohne jegliche Logik (“Durchlauferhitzer”) zählen nicht als “sinnvolle Tests” für die Bewertung

# Fehlerseiten / 404 testen

- Kommt eine 404 Seite wenn ich eine nicht existierende URL wie z.B. <http://localhost:8080/inexistent> aufrufe?
- Kommt eine 404 Seite wenn ich eine Entitäts-ID eingabe, die es in der DB nicht gibt, z.B. <http://localhost:8080/movie/999?>
- Kommt eine ansprechend gestaltete Fehlerseite, wenn eine Exception geschmissen wird?

# Code-Qualität

- Gibt es unnötige Duplizierung?
- Gibt es ein Basis-Layout Pebble-Template das überall verwendet wird?
- Gute Methoden und Variablennamen?
- Gute “Separation of Concerns”?
  - Z.B. Business-Logik in Services und nur Darstellungs-Logik im Controller.

# HTML / CSS

- Einigermassen schön formatiert? Der IDE-Support für automatische Formatierung von Pebble-Code ist leider nicht so toll.
- HTML und CSS valide? (CSS-Validator kann leider nicht mit CSS-Variablen umgehen → Validierung in IntelliJ anschauen)
  - HTMX-Attribute ohne data- sind erlaubt.
- Ein einziges CSS File für die App sollte reichen

# Letzter “Rundgang” vor Abgabe

- Ist alles eingecheckt und gepusht?  
→ Projekt in einen neuen Ordner auschecken und dann die untenstehenden Checks ausführen
- Testen, ob die wichtigsten Features noch funktionieren
- Alle automatisierten Tests laufen lassen
- Info-Seite nochmals prüfen