

## **Software Quality Concept:**

### **1. Entwicklungs- und Teamkultur:**

Grundsätzlich halten sich alle Teammitglieder an folgende Fragen:

- Verstehen meine Teammitglieder meinen Code mithilfe von JavaDoc?
- Entspricht mein Code der vereinbarten Programmiersprache?
- Gibt es bereits Code mit ähnlicher Funktionalität an anderer Stelle?

Wir arbeiten bevorzugt im Pair Programming mit zwei definierten Rollen: Eine Person schreibt den Code, die andere observiert und kommentiert ihn. Die Rollen werden regelmäßig getauscht, um Know-how gleichmäßig zu verteilen und die Codequalität konstant hoch zu halten.

Wir wollen zusätzlich mit Checklisten arbeiten und uns so oft wie möglich persönlich treffen. Zwischen den physischen Sitzungen wird über WhatsApp, Discord und Zoom kommuniziert.

### **2. Dokumentation mit JavaDoc:**

Wir setzen auf JavaDoc-basierte Dokumentation. Jede `public`-Klasse und -Methode muss mit JavaDoc beschrieben sein (Zweck, Parameter, Rückgabewert, Ausnahmen). Nur dokumentierter Code sollte gepusht werden – dies versuchen wir teamweit als Regel durchzusetzen.

Für die Dokumentation der Methoden und Klassen sind die Personen verantwortlich, die sie geschrieben haben.

### **3. Logging:**

Um die Fehlersuche zu vereinfachen, werden Logger eingesetzt. Die Idee dahinter ist, dass möglichst nach jeder Aktion eine Lognachricht erzeugt wird. Wir verwenden die Logger-Bibliothek Log4j2, um unseren Code zu debuggen und die Ausgaben in Log-Dateien zu speichern. Dies ist effizienter als der Ausdruck von Anweisungen auf der Konsole, da wir auf diese Fehler in einer separaten Textdatei zugreifen können, die viel mehr Informationen enthält als ein Standard-Terminal: den Schweregrad einer Anweisung (Fehler, Warnung, Info) und den genauen Zeitpunkt zu dem die Anweisung ausgegeben wurde. Dadurch können Fehler leichter erkannt und eingegrenzt werden.

### **4. Bug-Report:**

Wenn ein Bug auftritt, erstellen wir einen spezifischen Bug-Report, sodass alle Teammitglieder informiert sind und nachvollziehen können, was passiert ist. Bug-Reports können direkt in GitLab angelegt werden. Dies macht man in Gitlab unter Issues → newIssue. Sobald ein Bug ausfindig gemacht wird, sorgen die Autoren der Methoden oder Klassen mithilfe des Debuggers in IntelliJ dafür, dass der Programmfehler, isoliert und behoben wird.

## **5. Black Box Testing**

Wir möchten mithilfe von Black box testing die Benutzeroberfläche und Funktionen testen. Black Box testing überprüft die Benutzeroberfläche einer Anwendung, ohne den zugrunde liegenden Code zu analysieren. Es testet, ob Buttons, Formulare, Navigation und Interaktionen wie erwartet funktionieren. Dabei werden verschiedene Eingaben und Aktionen simuliert, um sicherzustellen, dass die Anwendung korrekt reagiert. Zudem werden Fehlerszenarien wie ungültige Eingaben oder unerwartete Nutzeraktionen getestet, um die Stabilität und Benutzerfreundlichkeit zu gewährleisten.

## **6. Metrik-Messungen:**

### **6.1 Lines of Code:**

Wir haben vereinbart, dass die Anzahl der Codezeilen pro Methode 150 Zeilen nicht überschreiten sollte. Einzelne Klassen sollten nicht mehr als 400 Zeilen Code beinhalten, ausser beinhalten z.B einen sehr grossen Teil der Spiellogik, dann kann es Sinn machen das diese Grenze überschritten wird. Wir haben diese Zahlen aus mehreren Gründen festgelegt: Größere Methoden und Klassen sind schwieriger zu lesen, schwerer zu ändern, fehleranfälliger und im Allgemeinen ohnehin vermeidbar sind. Um Lines of Code zu testen haben wir uns für das Plugin Metrics Reloaded für IntelliJ entschieden

### **6.2 Code Coverage:**

Unser Ziel ist es, JUnit-Tests durchzuführen und möglichst mehr als die Hälfte des Codes durch Tests abzudecken. Im Fokus stehen dabei die Spiel-Logik und die Client-Server-Interaktion, was auch das Encoding/Decoding des Protokolls sowie Lobby- und Spieler-Management einschließt. Als Tool verwenden wir das Jacoco-Plugin.

### **6.3 Complexity Metrik:**

Die zyklomatische Komplexität misst, wie komplex ein Stück Quellcode ist, indem sie die Anzahl der unabhängigen Pfade durch den Code zählt. Je höher der Wert, desto schwieriger ist der Code zu verstehen, zu testen und zu warten. Um die Komplexität zu bestimmen, wird ein Kontrollflussdiagramm erstellt, das zeigt, wie sich der Code basierend auf Bedingungen und Verzweigungen verhält. Ziel ist es, die zyklomatische Komplexität von Methoden unter 25 zu halten, damit der Code übersichtlich und wartbar bleibt. Für ganze Klassen ist ein höherer Wert akzeptabel. Zur Analyse nutzen wir das IntelliJ-Plugin Metrics Reloaded, das uns Messwerte zur zyklomatischen Komplexität liefert und hilft, problematische Stellen im Code frühzeitig zu erkennen.

## Messungen Lines of Code:

Stand: 5.04.25

