QS-Konzept für MS4

Syntax Syndicate - Casono

April 12, 2026

1 Einleitung

Dieses Dokument beschreibt die Software-Qualitätssicherung für unser Mehrspieler-Spiel *Casono*. Es wird als Teil der Programmierprojekt-Vorlesung (ehemals bekannt als cs108) an der Universität Basel von uns entwickelt.

Dieses Konzept basiert auf der *ISO 9126* Norm und ist daher in folgende Themenbereiche unterteilt:

- **Konstruktives Qualitätsmanagement** - Maßnahmen, die *während* der Entwicklung ergriffen werden, um die Qualität von Anfang an sicherzustellen. Einschließlich technischer Standards, Werkzeuge und der internen Organisation.
- **Analytisches Qualitätsmanagement** - Maßnahmen zur *Kontrolle und Untersuchung* des bestehenden Produkts, darunter automatisierte Analysen, Metrik-Schwellenwerte und Unit-Tests.

2 Konstruktives Qualitätsmanagement

Ein konstruktives Qualitätsmanagement umfasst alle Maßnahmen, die proaktiv Mängel verhindern, indem gemeinsame Standards, Prozesse und Werkzeuge festgelegt werden, die jedes Teammitglied während der Entwicklung anwendet.

2.1 Technische Maßnahmen

2.1.1 JavaDoc

Alle öffentlichen Klassen, Interfaces, Records und Methoden müssen durch JavaDoc dokumentiert werden.
Als Teil der GitLab-Build-Pipeline wird JavaDoc auf Richtigkeit überprüft. Werden strukturelle Fehler wie ungültige Referenzen erkannt, wird der Merge verhindert.
Unsere aktuelle Grundlage sind daher 0 Fehler. Für den kommenden Meilenstein ist aber unser Ziel, die Regelungen weiter zu verschärfen, sodass auch Warnungen einen Merge verhindern.

Unsere aktuelle implizite Form für JavaDoc-Kommentare ist wie folgt:

- Eine prägnante Zusammenfassung in einem Satz in der ersten Zeile
- Zusätzliche Informationen oder Beispiele als weitere Absätze.
- Dokumentation aller Parameter (`@param`), Rückgabewerte (`@return`) und Fehler (`@throws`).

Beginnend mit dem fünften Meilenstein soll diese Form ebenfalls durch einen Job in der CI-Pipeline überprüft werden.

2.1.2 Logging

Das Projekt verwendet Log4J 2 (`log4j-api` und `log4j-core` zur Laufzeit) zusammen mit Jansi für eine farbige Konsolenausgabe. Das Format der Ausgabe ist durch eine Konfigurationsdatei vereinheitlicht.

Die Protokollstufen werden wie folgt einheitlich verwendet:

- `DEBUG` für interne Zustandsänderungen
- `INFO` für Lebenszyklusereignisse wie Serverstart, Verbindungsaufbau/Trennung des Clients
- `WARN` für behebbare Anomalien
- `ERROR` für nicht behebbare Fehler

2.2 Organisatorische Maßnahmen

2.2.1 Teamkultur und die `CONTRIBUTORS`-Datei

Das Stammverzeichnis des GitLab-Repositories enthält eine Datei namens `CONTRIBUTORS.md`, die die Teamkonventionen dokumentiert:

- Die Verwendung von Branches - Feature-Branches werden in `main` zusammengeführt
- Merge-Richtlinie - CI muss bestanden werden. Keine ausdrückliche Genehmigung durch Kollegen erforderlich [1]
- Namenskonventionen für Branches und Commits
- Code-Stil - Google-Java-Format, AOSP-Variante, Einrückung mit vier Leerzeichen
- Jede Änderung beginnt mit einem Issue oder Task im GitLab, bevor mit der Umsetzung begonnen wird.
- Branch-Namen folgen dem Muster `<type>/<kurzbeschreibung>` gemäß Conventional Branch Standard
- Commit-Messages folgen dem Conventional Commits Standard
- Code-Style wird durch Linter (Checkstyle) und Formatter (Spotless, Google/AOSP Java Style) automatisiert überprüft
- Merge Requests müssen eine Beschreibung enthalten und auf das zugehörige Issue verweisen
- Zusammenarbeit und Kommunikation erfolgen bevorzugt über Issue-Kommentare, nicht über private Nachrichten

2.2.2 GitLab-Task- und Issue-Vorlagen für Fortschrittsverfolgung

GitLab-Vorlagen für Tasks und Issues werden für alle geplanten Arbeitselemente verwendet. Die Aufgabenvorlage sorgt für eine einheitliche Struktur, die die Fortschrittsverfolgung erleichtert und das Risiko vager oder unvollständiger Arbeitselemente verringert.
Durch Labels kann Tasks und Issues weiterer Kontext gegeben werden.

[1] Diese Regelung wird sich voraussichtlich ändern. Siehe dazu den Abschnitt zur `CODEOWNERS`-Datei.

3 Analytisches Qualitätsmanagement

3.1 Analytische Verfahren

3.1.1 Eigentumsrechte an Programmcode über GitLab `CODEOWNERS`

Beginnend mit dem fünften Meilenstein soll eine `CODEOWNERS`-Datei erstellt werden, welche Teammitgliedern die Eigentumsrechte an bestimmten Teilen des Programmcodes zuspricht. Wollen andere Änderungen an diesen Teilen vorgenommen werden, muss der Eigentümer sie freigeben.
Aktuell ist jedoch noch unklar, ob diese Funktion genutzt werden kann, da die Instanz mindestens die *Premium*-Stufe haben muss.

3.1.2 Automatisiertes CI-Linting und Build-Verifizierung

Jeder Push und jede Merge-Anfrage löst die CI-Pipeline aus.
Für Pushes und Merge-Requests setzt sich die Pipeline aus folgenden Phasen zusammen:

1. **Linting-Phase** [Push, MR] - Checkstyle und Spotless überprüfen, ob der Programmcode dem vereinbarten Stil entspricht.
2. **Build-Phase** [Push, MR] - `./gradlew assemble` prüft, ob das gesamte Projekt fehlerfrei kompiliert werden kann.
3. **Checkstyle-Report** [MR] - Ein zusätzlicher Checkstyle-Report wird für Merge-Requests erzeugt und als Code-Quality-Report bereitgestellt.
4. **Javadoc-Prüfung** [MR] - JavaDoc wird auf Korrektheit geprüft.
5. **Test-Phase** [Push, MR] - Automatisierte Tests werden ausgeführt und ein Testreport erzeugt.

Fehlschlagende Jobs verhindern einen Push nicht, jedoch wird ein Merge so lange verhindert, bis alle Jobs fehlerfrei abgeschlossen werden können.

3.2 Testverfahren

Unit-Tests werden mit *JUnit 5* geschrieben und in der Testphase der CI-Pipeline ausgeführt. Ein fehlgeschlagener Test blockiert den Merge-Request.
Als weiteres Werkzeug verwenden wir *JaCoCo* für die Ermittlung der Testabdeckung.
Die aktuelle Testabdeckung ist wie folgt:

Paketname (gekürzt)	**Instr. Cov.**	**Branch Cov.**
`client.chat`	0 %	0 %
`client.game`	0 %	0 %
`client.network`	3 %	1 %
`client.ui` (von Abdeckung ausgeschlossen)	0 %	0 %
`server.app.checks`	0 %	0 %
`server.app.commands`	0 %	0 %
`server.domain.game`	94 %	91 %
`server.domain.*`	45 %	37 %
`server.network.transport`	0 %	0 %
`server.network.protocol`	0 %	0 %
`server.network.*`	2 %	1 %
Summe	**12 %**	**9 %**

Unser Ziel ist es, realistische Anforderungen an unsere Testabdeckung zu stellen. Bis zum fünften Meilenstein wollen wir daher für wichtige Bereiche, wie die Game-Engine und interne Komponenten des Netzwerks, eine möglichst hohe Abdeckung von mindestens 50% zu erzielen.