QS-Report für MS6

Syntax Syndicate - Casono

May 17, 2026

1 Einleitung

Dieses Dokument ist die Fortsetzung des QA-Konzeptes für unser Mehrspieler-Spiel *Casono*. Es wurde von uns als Teil der Programmierprojekt-Vorlesung (ehemals bekannt als cs108) an der Universität Basel von uns entwickelt.
Dieser Report folgt dem selben *DIN ISO 9126*, und ist wie auch das Konzept für den ersten Teil in die selben Themenbereiche unterteilt:

- **Konstruktives Qualitätsmanagement** - Maßnahmen, die *während* der Entwicklung ergriffen werden, um die Qualität von Anfang an sicherzustellen. Einschließlich technischer Standards, Werkzeuge und internen Organisation.
- **Analytisches Qualitätsmanagement** - Maßnahmen zur *Kontrolle und Untersuchung* des bestehenden Produkts, durch Anwendung 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 einhält.

2.1 Technische Maßnahmen

2.1.1 JavaDoc

JavaDoc wurde genutzt um alle Öffentlichen Klassen, Interfaces, Records und Methoden zu dokumentieren. Als Teil der GitLab-Build-Pipeline wird JavaDoc auf Richtigkeit überprüft. Werden strukturelle Fehler wie ungültige Referenzen erkannt, schlägt die Pipeline fehl und der Merge wird verhindert.

Unsere implizite Form für JavaDoc-Kommentare ist wie folgt:

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

Leider konnten wir das in MS4 vorgenommene Ziel, das auch Warnungen zu einem Abbruch der Pipeline führen, aus Technischen Gründen nicht umsetzen.

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

Im Gegensatz zu MS4 wurden die Namen der Logger gekürzt und nicht mehr der gesamte Classpath verwendet. Ein Beispiel für die Ausgabe sieht so aus:

```
20:18:00.514 DEBUG    NetworkManager       Accepted connection from /127.0.0.1:51891
20:18:00.532 DEBUG    SessionReader-2f70cd1b-5088-44cb-aba1-3fb13c2c46d3 Recieved: RawPacket[requestId=1, payload=LOGIN USERNAME=Lars]
20:18:00.534 DEBUG    SessionReader-2f70cd1b-5088-44cb-aba1-3fb13c2c46d3 Parsed request to RawRequest[command=LOGIN, parameters=[Reque
stParameter[key=USERNAME, value=Lars]]]
20:18:00.537 DEBUG    SessionReader-2f70cd1b-5088-44cb-aba1-3fb13c2c46d3 Converted to PrimitiveRequest[context=RequestContext[sessionI
d=ch.unibas.dmi.dbis.cs108.casono.server.network.sessions.SessionId@6d06a05d, requestId=1], command=LOGIN, parameters=[RequestParamete
r[key=USERNAME, value=Lars]]]
20:18:00.542 INFO     UserFactory          Registered user 'Lars' with id 9962513e-546c-45d3-923f-5ae393da45aa for session 2f70cd1b-50
88-44cb-aba1-3fb13c2c46d3
```

Figure 1: Konsolenausgabe des Servers während dem Verbindungsaufbau und Anmeldung eines Nutzers

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 seit MS4 unveränderten Teamkonventionen dokumentiert:

- Die Verwendung von Branches - Feature-Branches werden in `main` zusammengeführt
- Merge-Richtlinie - CI muss bestanden werden. Eine Genehmigung durch Kollegen ist wünschenswert wenn Änderungen an dessen Code vorgenommen wurden.
- Namenskonventionen für Branches und Commits
- Code-Stil - Google-Java-Format, AOSP-Variante, Einrückung mit vier Leerzeichen
- Jede Änderung beginnt mit einem Issue oder Task in GitLab, bevor mit der Umsetzung begonnen wird
- Branch-Namen folgen dem Muster `<typ>/<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 Anfragen müssen eine Beschreibung enthalten und auf das zugehörige Issue oder den Task 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 reduziert.
Durch die Vergabe von Labels kann Tasks und Issues weiterer Kontext gegeben werden.

3 Analytisches Qualitätsmanagement

3.1 Analytische Verfahren

3.1.1 Eigentumsrechte an Programmcode über GitLab `CODEOWNERS`

In unserem Konzept haben wir uns vorgenommen, durch eine `CODEOWNERS` Datei den Teammitgliedern die Eigentumsrechte ihres Programmcodes zuzusprechen. Leider hat sich bestätigt das die von SciCORE betriebene Instanz nicht die Premium-Stufe hat, wodurch diese Funktion für uns nicht zur Verfügung steht.

Statt auf eine Automatisierte Grundlage zu vertrauen, haben wir uns für den nächstbesten Weg der Disziplinarität entschieden, wie am Beispiel von dieser Merge-Anfrage gesehen werden kann.

3.1.2 Automatisiertes CI-Linting und Build-Verifizierung

Jeder Push und jede Merge-Anfrage löst die CI-Pipeline aus.
Für Pushes und Merge-Anfragen 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-Anfragen 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.

Die Entscheidung das fehlgeschlagene Jobs keinen Push aber dafür eine Merge-Anfrage verhindern können, hat sich als vernünftig bewehrt. Über die Dauer des Projektes hat diese Pipeline mehrmals verhindert das Fehlerhafter oder sogar kaputter Programmcode in den main Branch gelangen konnte.

3.2 Testverfahren

Unit-Tests wurden mit *JUnit 5* geschrieben und in der Testphase der CI-Pipeline ausgeführt. Ein fehlgeschlagener Test blockiert die Merge-Anfrage.
Als weiteres Werkzeug verwendeten wir *JaCoCo* für die Ermittlung der Testabdeckung.
Die Testabdeckung für Meilenstein sechs ist wie folgt:

Paketname (gekürzt)	**Instr. Cov.**	**Branch Cov.**
`client.chat`	30 %	16 %
`client.game`	15 %	2 %
`client.network`	2 %	1 %
`client.ui` (von Abdeckung ausgeschlossen)	0 %	0 %
`server.app.checks`	0 %	0 %
`server.app.commands`	0 %	0 %
`server.domain.game`	68 %	43 %
`server.domain.*`	59 %	44 %
`server.network.transport`	0 %	0 %
`server.network.protocol`	67 %	55 %
`server.network.*`	67 %	55 %
Summe	**18 %**	**11 %**

Im vierten Meilenstein haben wir uns das Ziel gesetzt die Testabdeckung für die Game-Engine und wichtige Netzwerkkomponenten auf mindestens 50% zu erhöhen, dieses Ziel haben wir laut JaCoCo nicht erreicht.
Geht man jedoch nach der Abdeckung der "Test Runner for Java" Erweiterung in VSCode, dann beträgt die Abdeckung 51%. Sowohl Julian (Game-Engine) als auch Lars (Serverseitiges Netzwerk) haben sich nach diesen Angaben gereichtet, weshalb wir dachten wir hätten unser Ziel erreicht.

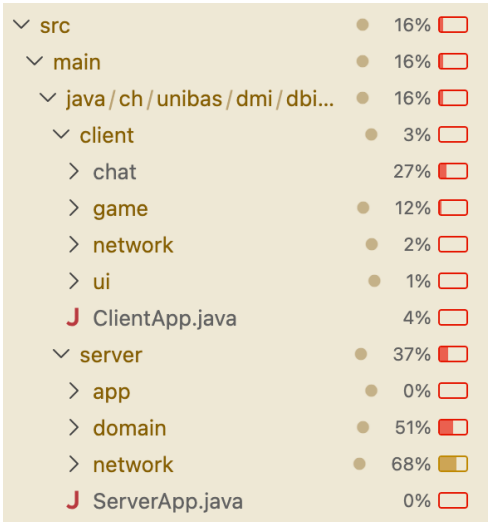

Figure 2: Abweichende Angaben bezüglich der Abdeckung von VSCode zu JaCoCo

Die Verwendung zwei unterschiedlicher Tools zur Beurteilung des Fortschritts ist definitiv etwas, was es in Zukunft zu vermeiden gilt.

4 Metriken

Die nachfolgenden Metriken wurden am 17. Mai 2026 um 20:58 Uhr erhoben, neuere Änderungen sind daher nicht berücksichtigt.

4.1 Lines of Code

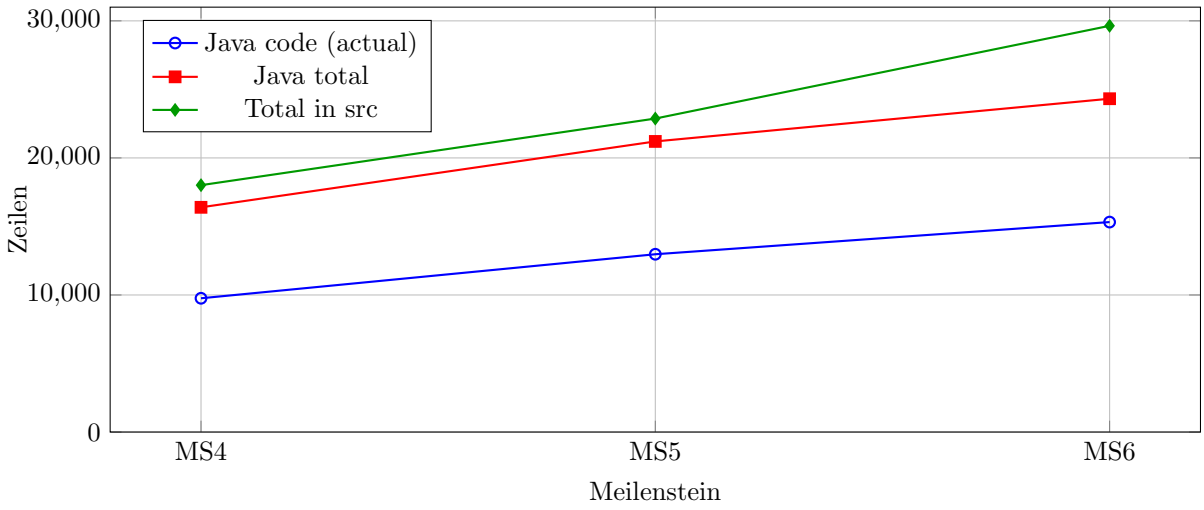

Figure 3: Entwicklung der Java-Zeilen (tatsächlicher Code vs. Gesamt) über die Meilensteine hinweg

4.2 Verteilung der Code-Zeilen auf verschiedene Dateitypen

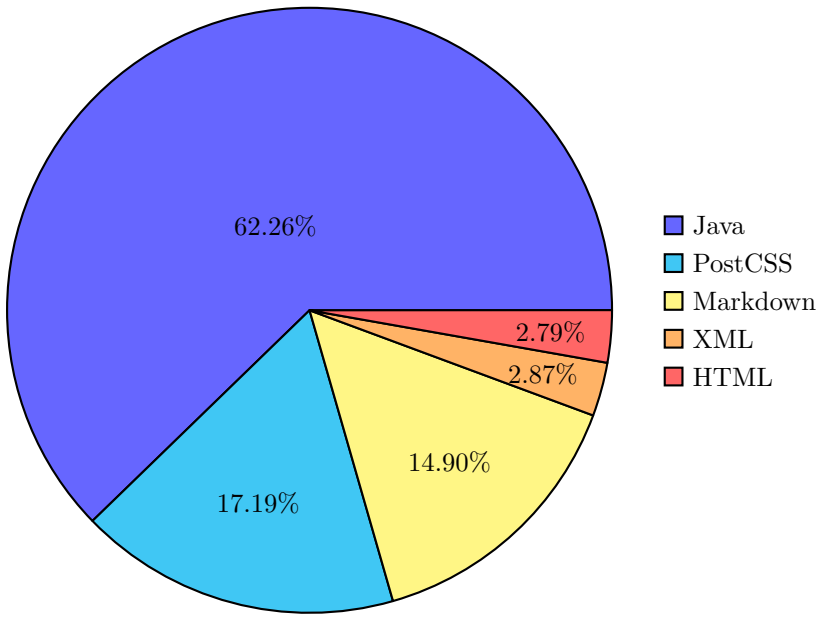

Figure 4: Verteilung der Code-Zeilen (Top 5 Dateitypen)