QA-concept

Grundsätzlich sollte sich das Team an folgenden Punkten, die zu nicht-funktionalen Qualitätsmerkmalen gehören orientieren:

- Verstehen meine Teammitglieder diesen Code mithilfe des JavaDocs und meiner Dokumentation?
- Hält sich der Code an den vereinbarten Programmierstil.Dabei haben wir uns für den Google Java Style Guide entschieden. Ausserdem wurde Checkstyle im Gradle konfiguriert, womit man sein Code selber testen kann.
- Hat es bereits Code mit ähnlicher Funktionalität an anderer Stelle?
- Würde ich diesen Code an dieser Stelle im Projekt erwarten?
- Lässt sich die Lesbarkeit dieses Codes erhöhen?
- Könnte ich diesen Code ändern oder ist der Code übertragbar?

JavaDoc

Durch das Dokumentieren und der Zusammenfassung der Methoden und Klassen wird der Aufwand für Teammitglieder, die Methoden und Klassen zu verstehen und weiter zu bearbeiten geringgehalten. Für die Dokumentation der Methoden und Classen sind die Personen, die die Klassen und Methoden schreiben zuständig. Um die Dokumentation zu vereinfachen wird der Plugin JavaDoc in IntelliJ verwendet.

Logging

Um Fehlersuche zu vereinfachen und zu beheben werden Logger verwendet. Die Idee dahinter ist, dass am Anfang jeder Aktion und am Ende jeder Aktion eine Lognachricht verschickt wird, wodurch man die Fehler einfacher hervorheben kann. Ausserdem werden die Lognachrichten in einem Logfile gespeichert. Wir haben uns für die SLF4J API mit logback-classic und core Konfiguration entschieden. Um unser Code manuell zu prüfen haben wir uns für informales Review in Sitzungstechnik entschieden. Dabei wird jede Methode und Klasse vom Autor vorgestellt und isoliert ausgeführt und getestet. Es findet all zwei bis drei Tage eine Teambesprechung über Zoom statt, dabei werden neue Fortschritte und aktuelle Schwierigkeiten besprochen. In der Besprechung werden die nächsten Aufgaben für die nächste Besprechung festgelegt und unter Teammitgliedern verteilt und zum Schluss werden neu geschriebene Methoden und Klassen erklärt und analysiert.

Ein Ausschnitt des LogFiles

```
12:28:10.394 [main] INFO  Start - Server started
12:30:05.051 [main] INFO  Start - Server started
12:30:32.514 [main] INFO  Start - ClientThread started
12:43:47.865 [main] INFO  Start - ClientThread started
12:43:47.886 [Thread-0] INFO  server.Client - Socket created
12:43:47.891 [Thread-0] INFO  server.Client - ClientreaderThread started
12:43:57.397 [Thread-0] INFO  server.Client - changing the name
12:44:14.402 [Thread-0] INFO  server.Client - created a new Lobby
12:44:51.420 [Thread-0] INFO  server.Client – Quitting
```

Bug-Reports

Programmfehler, welche bei individuellen Code Review oder bei Zoommeeting auftreten, müssen dokumentiert werden. Das Bug-Report muss folgende Informationen enthalten:

- Welche Version(en) des Programms (git commit hash) sind betroffen?
- In welcher Umgebung (Betriebssystem, Java Version, etc.) tritt das Problem auf?
- Wie genau äussert sich das Problem?
- Was sind Ausgaben/Logs vor, während und nach dem Problem?
- Welche Schritte sind nötig, um das Problem herbei zu führen?
- Ist das Problem zuverlässig reproduzierbar?

Sobald ein Bug ausfindig gemacht wird, sorgen die Autoren der Methoden oder Klassen mithilfe des Debuggers in IntelliJ dafür, dass der Programmfehler reproduziert, isoliert und behoben wird.

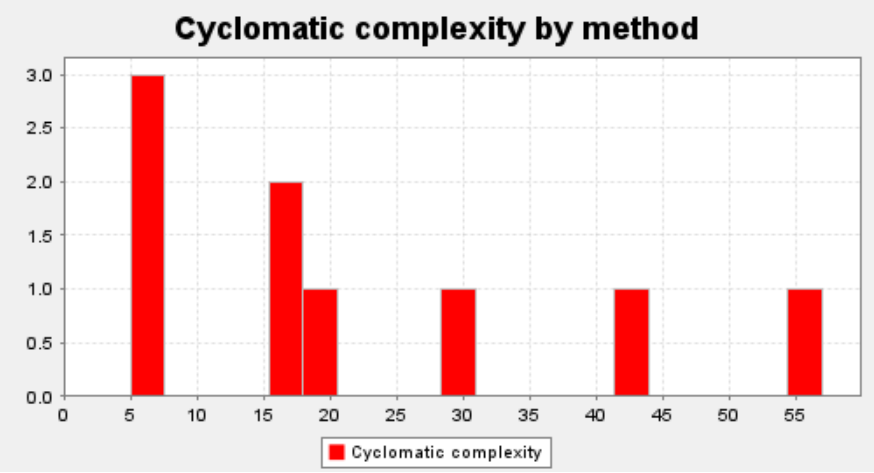
Code Coverage

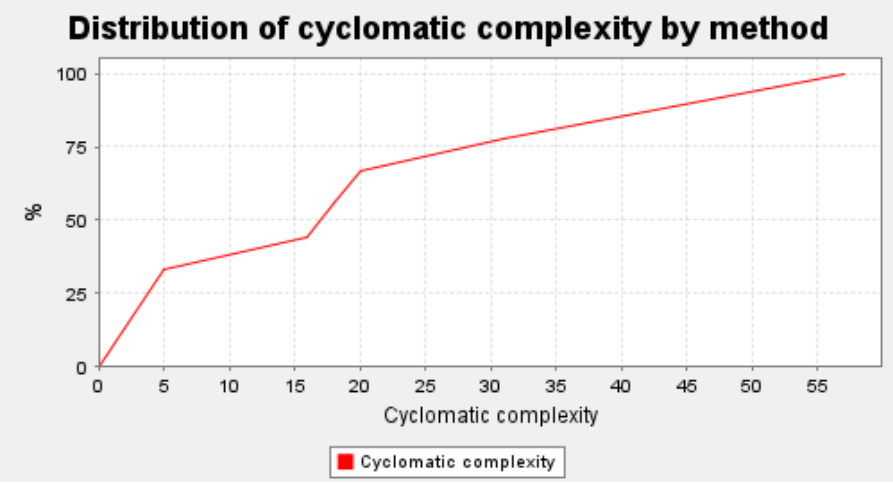
Unser Ziel für Meilenstein 4 ist es, JUnit Test durchzuführen und mehr als die hälfte des Codes mit Tests abzudecken. Zentral wird dabei auf die Gamelogik und Client-Server Interaktion geachtet, was wiederum auch Encoding-Decoding des Protokolls und Lobby-Player-Management beinhaltet. Als Tool wird dabei das Plugin JaCoCo benutzt.

Metrik-Messungen

Die drei Metriken, für die wir uns entschieden haben sind Lines of Code pro Klasse, Code Coverage für Tests und Cyclomatic Complexity. Vorerst haben wir uns entschieden Cyclomatic Complexity zu messen. Ziel für Meilenstein 4 ist es, Metrikmessungen für Code Coverage für Tests durchzuführen.
Um Messwerte über zyklomatische Komplexität von unserem Projekt zu bekommen, haben wir mithilfe von Metrics Reloaded von IntelliJ die Klassen Server, ServerThreadForClient, Client, ClientReaderThread, Board, und Lobby analysiert.
Diese Klassen wurden aufgrund ihrer Wichtigkeit und Komplexität ausgewählt.
Dabei haben wir folgende Resultate erhalten. (siehe nächste Seite)
Damit die Resultate übersichtlich sind, wurden nur Werte ausgewählt, die den metrischer Schwellenwert überschritten haben. Laut Metrics Reloaded sollte der metrische Schwellenwert bei Methoden maximal 10 sein. Wir haben allerdings Werte zwischen 5 bis 57 bekommen. Die Klassen ServerThreadForClient, Client und ClientReaderThread wiesen wie erwartet sehr hohe metrische Schwellenwerte zwischen 31 und 57 auf.

Cyclomatic complexity





server.ServerThreadForClient.run()	57.0
server.Client.run()	44.0
server.ClientReaderThread.run()	31.0
game.Lobby.run()	20.0
game.Board.floodBoard(int)	18.0
game.Board.Board(int,int)	16.0
Total	201.0
Average	22.32