Quality Assurance Concept

1. Organisatorische Massnahmen

Verantwortungen:

Pro Milestone wird eine **Responsibility Assignment Matrix** (RAM) erstellt. Jede Person weiss, für welche Aufgaben sie die Hauptverantwortung besitzt. Zusätzlich zum Hauptverantwortlichen wird auch eine zweite Person markiert, welche über diesen Teil des Projekts gut informiert ist.

Code Review:

Pro Woche sollte mindestens ein Treffen vereinbart werden, indem der neu geschriebene Code dem Rest des Teams vorgestellt wird. Weiter bekommt jeder eine Person zugeteilt, dessen Code geprüft werden soll. Es geht dabei darum, den Code zu lesen, auf Bugs zu prüfen und eine Rückmeldung zu geben. Wichtig dabei sind Dinge wie sinnvolle Dokumentation, eingehaltene Namenskonventionen und verständlicher, gut formatierter Code. Pro Woche sollte eine solche Rückmeldung an den jeweiligen Programmierer gelangen. Bugs werden zusätzlich per Issue Tracker auf GitHub für alle dokumentiert.

Pair-Programming:

Wann immer möglichst sollte **Pair-Programming** bevorzugt werden. Dies kann in Person oder über Discord geschehen. Mindestens zu den Zeiten der Programmierprojekt-Vorlesung am Donnerstag und Freitag sollte jedes Teammitglied erreichbar sein und wenn möglich vor Ort in der Spiegelgasse.

2. Technische Massnahmen

Werkzeuge:

Unser Projekt wird mit der Programmiersprache Java, der Markupsprache FXML und der Stylesheet-Sprache CSS in Intellij umgesetzt. Weiter benutzen wir auch folgende Werkzeuge:

- Logger (**Log4J**)
- IntelliJ-Plugin **MetricsReloaded**
- **Microsoft Excel**
- **JaCoCo** Code Coverage Library
- **SceneBuilder**
- **Mockito** Library

Richtlinien:

Folgende Richtlinien sollten alle Teilnehmer des Projektes befolgen:

- Eine **einheitliche Namensgebung** gemäss den typischen Java Namenskonventionen muss eingehalten werden (z.B. Klassen grossgeschrieben, Variablen in camelCase). Zudem sollten Namen immer aussagekräftig sein.
- Verständliche **JavaDocs** müssen während dem Programmieren geschrieben werden. Es sollte möglich sein, den Code mithilfe der Dokumentation zu verstehen.
- Sinnvolles **Exception-Handling** soll betrieben werden. Catch-Blöcke sollten immer bearbeitet werden.
- Bugs müssen per **Issue Reporter** auf GitHub notiert werden. Darin muss unbedingt enthalten sein, wie man den Bug reproduziert und in welchem Kontext er auftrat. Das Team muss auf neue Bug-Reports aufmerksam gemacht werden (z.B. per WhatsApp).

- Pro Woche sollte der **neu geschriebene Code** von einer anderen Person **überprüft** werden (siehe organisatorische Massnahmen). Die Reformat Code Option von IntelliJ sollte jeweils vorher auf den neuen Code angewendet werden.

Messungen:

Einmal pro Woche (Sonntags) werden folgende drei Code Metrics mithilfe von MetricsReloaded gemessen und mit Microsoft Excel visualisiert:

- **JavaDocs**
  Nicht jede Person ist gleich vertraut mit dem Code der anderen, eine effiziente Dokumentation ist für uns deshalb äusserst wichtig. Wir schauen uns das Ganze auf Klassenebene an, wobei wir uns für die Messwerte JLOC (Javadoc Lines of Code) und Jm (Method Coverage) interessieren. Mithilfe der Jm-Werte wird klar, in welchen Klassen es an Dokumentation mangelt. Die Klassen mit sehr hohen und sehr niedrige JLOC-Werten werden wir uns auch genauer anschauen um zu sehen, ob es eventuell mehr Dokumentation braucht oder ob vielleicht sogar zu ausführliche bzw. repetitive Erklärungen dabei sind.
- **Lines of Code**
  Um das Projekt übersichtlich zu halten, schauen wir uns die Anzahl Codezeilen pro Klasse an. Das Limit setzen wir bei 200 Linien Code, damit die Klassen so übersichtlich wie möglich bleiben. Ausnahmen können so bestimmt werden und müssen im Team abgeklärt werden. Wir möchten aber auch nicht Unmengen an Klassen mit nur wenig Zeilen Code haben, weshalb es auch sinnvoll ist, die eher spärlich besetzten Klassen im Blick zu behalten und zu besprechen, ob diese wirklich sinnvoll sind. Klassen wie der GUI-Launcher und Ähnliches dürfen natürlich entsprechend kurz sein und werden in dieser Messung nicht berücksichtigt.
- **Cyclic Complexity**
  Um im Blick zu behalten, wie komplex unser Projekt ist, messen wir pro Methode die zyklische Komplexität. Es sollten so wenig Methoden wie möglich über einem Wert von 20 sein. Sind sie es trotzdem, muss dies entweder geändert, oder zumindest immer im Hinterkopf behalten werden. Gewisse Methoden zur Verwaltung des Netzwerkprotokoll besitzen durch die vielen Switch-Cases sowieso eine hohe Komplexität, wessen wir uns allerdings bewusst sind.

Weiter werden die Anzahl Logging-Statements gemessen und per JaCoCo wird geprüft, wie viel Prozent des Codes von Tests abgedeckt sind.

Tests:

Geplant war, unsere Server-Client-Interaktion als zentrale Komponente mit JUnit-Tests zu testen. Allerdings sind wir in einige Probleme mit den Unit-Tests gerannt und testen schlussendlich “bloss” die korrekte Namensgebung, die Receive-Funktion des Protokoll, ob Lobbies und Users richtig erstellt werden und ob die send-Methoden funktionieren.

3. Durchführung

Bezüglich der Durchführung unseres Projektes war unsere grösste Erkenntnis die Wichtigkeit von Pair-Programming. Vor allem in den letzten paar Meilensteinen legten wir grossen Wert darauf und trafen uns jeden Mittwoch und teilweise auch Donnerstags und Freitags. Ausserdem gab es noch zusätzlich spontane Meetings per Discord. Auf diese Weise konnten wir Fragen direkt klären, Bugs besprechen und den weiteren Verlauf des Projektes planen. Das gemeinsame Programmieren war sicherlich auch ein Grund, weshalb wir die Meilensteinziele meist gut erreicht und kurz vor der

Abgabe nie gross Probleme mit schwerwiegenden Bugs hatten. Wegen diesen häufigen Treffen brauchten wir unser Programmiertagebuch weniger, wie erwartet, denn es kam nur sehr selten vor, dass nicht alle Teammitglieder anwesend waren. Einträge schrieben wir natürlich trotzdem. Weiter war auch die Stimmung im Team allgemein gut und die Zusammenarbeit klappte meist recht reibungslos. Zwar gab es teilweise Probleme bei der Kommunikation und vor allem bei den ersten beiden Meilensteinen war unsere Arbeitsteilung recht schlecht, doch während des Projektes verbesserte sich dies signifikant. Das wohl grösste Problem bei unserem Projekt war aber die Zeit. Zwar hatten wir, wie erwähnt, nie grosse Probleme vor den Abgaben, doch vor allem die Umsetzung unseres QA-Konzeptes hat gelitten. So führten wir zum Beispiel selten bis nie Code Review durch oder setzen uns zu wenig mit dem Logger auseinander, als dass alle Gruppenmitglieder ihn sinnvoll hätten nutzen können. Die fehlende Code Review war sicher auch ein Grund, weshalb wir teilweise den Überblick über den Code der anderen verloren hatten. Allerdings konnten wir dies gut durch unsere häufigen Treffen ausgleichen, da man so einfach kurz nachfragen konnte. Weiter haben wir auch nicht immer sauber geplant, wer was macht. Unsere Responsibility Assignment Matrix (RAM) und der Plan auf Gantt standen zwar, an mehr als an die grobe Zuteilung des Bereichs (z.B. GameLogic, GUI) hielten wir uns aber nicht. Allerdings war dies weniger ein Problem, wie erwartet, da wir bei unseren Treffen alle sofort informiert waren, wer neu welche Aufgaben zu erledigen hat. Ausserdem hatten wir auf Google Docs eine Datei mit allen Meilensteinen und legten dort fest, wer für welche Aufgabe verantwortlich ist. Eine solche "dynamische" Planung hat im Endeffekt sehr gut funktioniert. Ein weiteres Problem, was anzusprechen ist, sind die Unit-Tests. Wir hatten bloss eine Person für die Unit-Tests zugeteilt und erkannten zu spät, dass unser Code nur schlecht zum Testen ausgelegt war und es in Wahrheit eine Menge Arbeit gewesen wäre, sinnvolle Tests zu schreiben. Dies führte dazu, dass unsere Unit-Tests nur einen begrenzten Bereich unseres Projektes testen. Da alle anderen Personen schon selbst mit Aufgaben beschäftigt waren, entschieden wir uns schlussendlich, die JUnit-Test eher in den Hintergrund zu schieben. Im Nachhinein wissen wir aber zumindest, dass wir bei einem erneuten Projekt z.B. den Aufbau des Netzwerkprotokolls anders lösen müssen, damit Unit-Tests effizient laufen können. Schliesslich hatten wir, wie zu erwarten, auch einige frustrierende technische Probleme. Grund waren unter anderem Schwierigkeiten beim Mergen, verschiedene Intellij-Versionen, Betriebssystemunterschiede und vieles Weiteres. Schlussendlich fand sich aber immer eine Lösung auch wenn wir einiges an Zeit verloren. Da wir vor dem Projekt nie mit IntelliJ oder GitHub gearbeitet haben, war dies jedoch abzusehen und Teil des Lernprozesses.

4. Messungen

4.1 Logging Statements

warn	info	error	trace
14	8	5	2

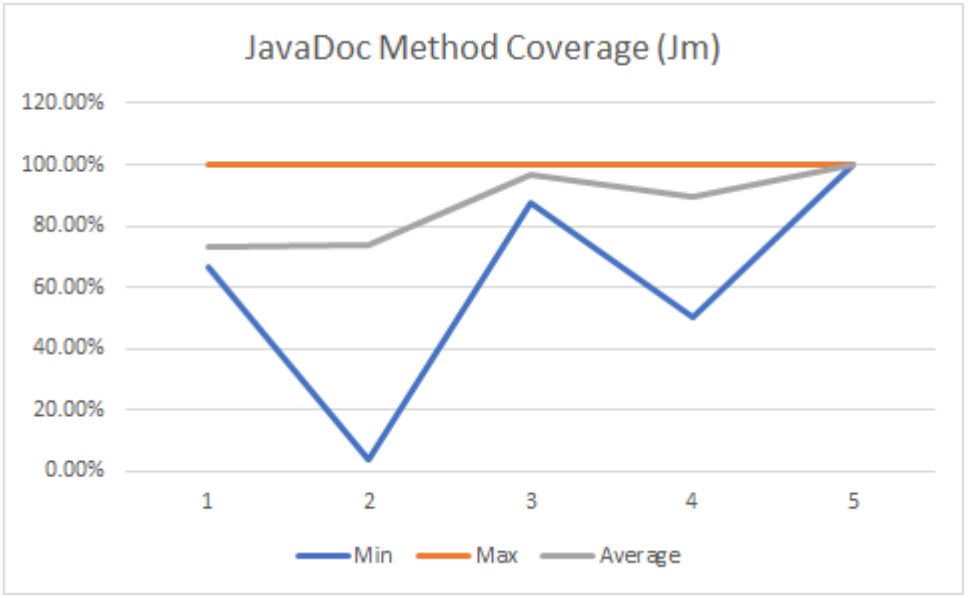
4.2 Code Coverage

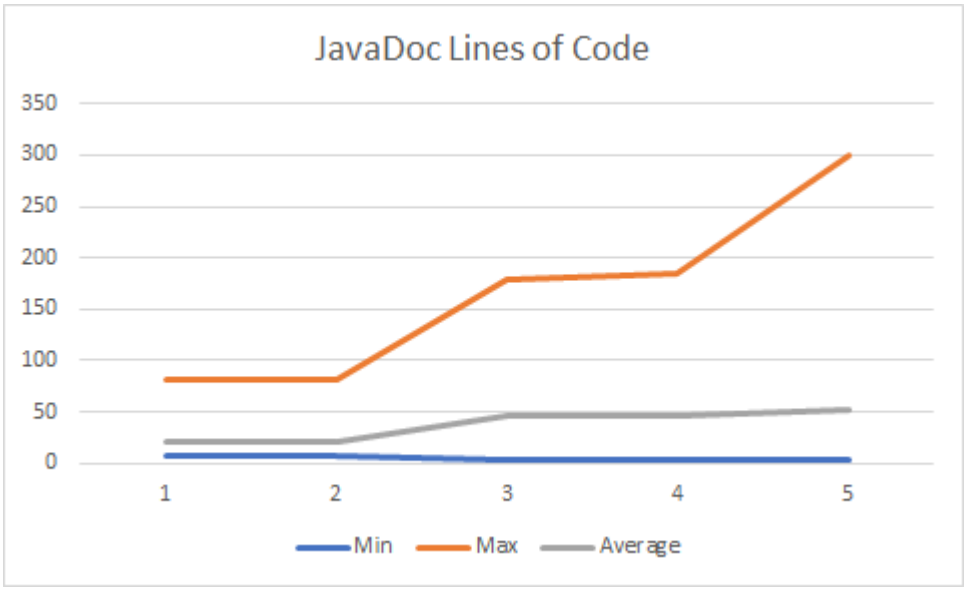
class	method	line
25%	12%	7%

4.3 Code Metrics

Wir haben fünf Messungen unserer drei Metrics vorgenommen, wobei die erste davon bei Meilenstein 3 und die letzte bei Meilenstein 5 war.

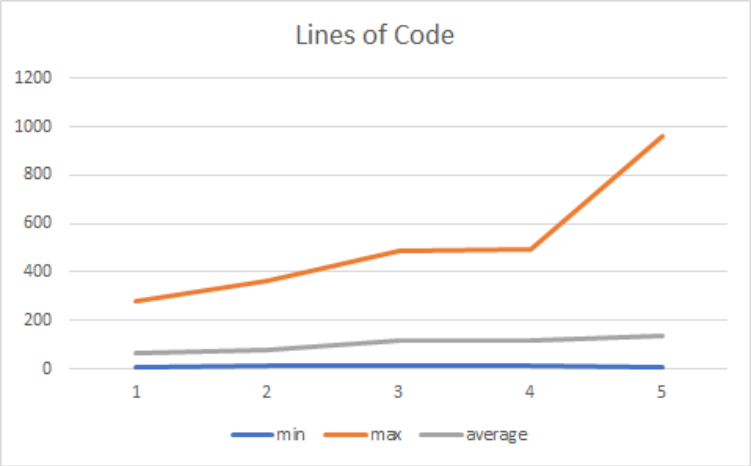
4.3.1 JavaDocs





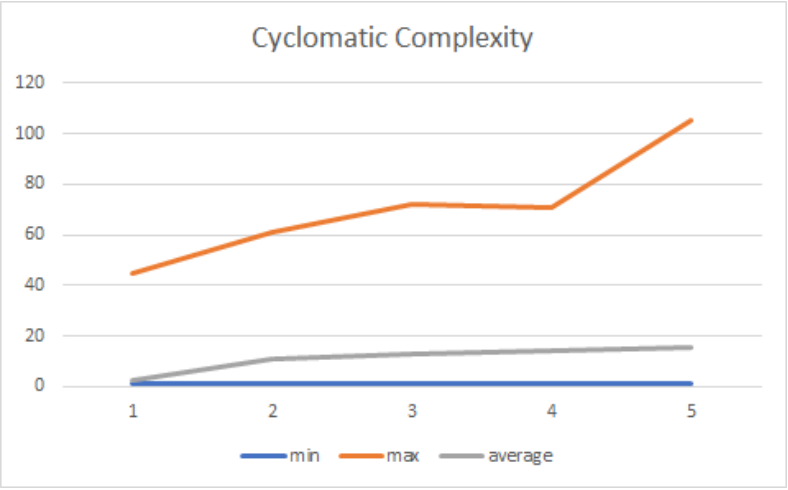
Anhand unserer JavaDoc-Messungen kann man ablesen, dass wir jeweils während dem Programmieren nur mittelmässig dokumentiert haben, was wir vor den Meilensteinen jeweils nachholen mussten. Da wir bei Meilenstein 3 die Punkte bezüglich Dokumentation nicht bekommen haben, stellten wir sicher, dass wir beim nächsten auch wirklich 100% aller Methoden und Feldvariablen per JavaDoc abgedeckt hatten. Wie schon gesagt, geschah dies aber meist erst kurz vor der nächsten Abgabe. Allerdings muss auch gesagt werden, dass wir oft selbsterklärende Methodennamen hatten, weshalb auch Teammitglieder, welche den Code nicht selbst geschrieben hatten, sofort wussten, was der Sinn der Methode ist. Durch unsere häufigen Treffen konnte jeder auch schnell nachfragen, was eine Methode tut bzw. wie sie aufgebaut ist. Dies war einerseits ein grosser Vorteil, andererseits auch ein Nachteil, da wir so die JavaDocs eher auf später verschoben. Bei Projekten mit mehr Teilnehmern und weniger häufigen Treffen müsste dies jedoch sicherlich anders gehandhabt werden.

4.3.2 Lines of Code



Bezüglich unseren Lines of Code sieht man, dass unser Ziel von 200 Zeilen Code pro Klasse klar nicht eingehalten wurde. Bereits bei Meilenstein 3 hatten wir mit der GameLogic Klasse einen Ausreisser, bei Meilenstein 5 nähern sich einige Klassen nun bereits 1'000 Zeilen Code an. Wahrscheinlich wäre es von Anfang an realistischer gewesen, die maximalen Zeilen pro Klasse auf 300 zu erhöhen und es hätte einer besseren Planung bezüglich Klassen und deren Methoden gebraucht. Teilweise hätte man sich mit neuen Klassen auch eine Menge Code sparen können, zum Beispiel hätte es für die GameController Klasse (zuständig für das GUI des Spiel-Screens) definitiv eine Hilfsklasse gebraucht, welche die Spieler-Tokens verwaltet. Da dies aber zu spät bemerkt wurde, gibt es in dieser Klasse viel unnötigen Code und einige zusätzliche Switch-Cases. Mit einer Hilfsklasse hätte man nicht nur weniger Zeilen Code, sondern auch eine niedrigere Komplexität, da viele Switch-Cases wegfallen würden. Zumindest die Klassen mit 400-500+ Zeilen hätte man überarbeiten müssen oder zumindest eine klare Gliederung einführen sollen, wie diese aufgebaut sind (zum Beispiel alle Setter am Schluss, Initialize am Anfang , etc.), da diese sehr unübersichtlich wurden. Wie bei vielen Punkten in unserem Projekt fehlte aber auch hier die Zeit.

4.3.3 Cyclomatic Complexity



Die Cyclomatic Complexity wurde für bestimmte Klassen sehr hoch mit der Zeit. Bereits bei Meilenstein 3 war sie für die GameLogic über unserer Grenze von 20, und die Komplexität wuchs von da an nur noch. In einigen Fällen war uns bewusst, dass wir solch hohe Zahlen erhalten werden. Ein Hauptgrund waren Switch-Cases im Netzwerkprotokoll, welche die Komplexität stark erhöhten. Dies ist für uns allerdings in Ordnung, da Switch-Cases eine übersichtliche und regelmässige Struktur

haben und wir sie somit nicht als übermässig komplex einstufen. Zum Netzwerkprotokoll legten wir auch ein Enum an, welches eine übersichtliche Dokumentation aller Cases liefert, so konnten wir den Überblick behalten. Allerdings gibt es teilweise sogar Switch-Cases in anderen Switch-Cases drin, was man wohl anders hätte lösen sollen. Auch gibt es in der GameLogic und in den Receives (Server Receive, Client Receive, etc.) einige Loops, welche das ganze unübersichtlicher und komplexer machen. Da wir viele komplexe Klassen haben, war das Testen mit Unit-Tests natürlich auch erschwert. Man muss allerdings auch sagen, dass das Programmierprojekt für uns alle unser erstes grosses Projekt war, und wir konzentrierten uns in erster Linie darauf, dass der Code überhaupt läuft. Würden wir unser Projekt noch einmal beginnen, ist klar, dass wir uns, vor allem bei absehbar komplexen Klassen, zuerst eine Struktur überlegen sollten und diese wenn nötig anpassen müssen, bevor viele andere Klassen von der komplexen abhängig sind und Änderungen nur zu Fehlern führen.