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 Gitlap 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 sie 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.

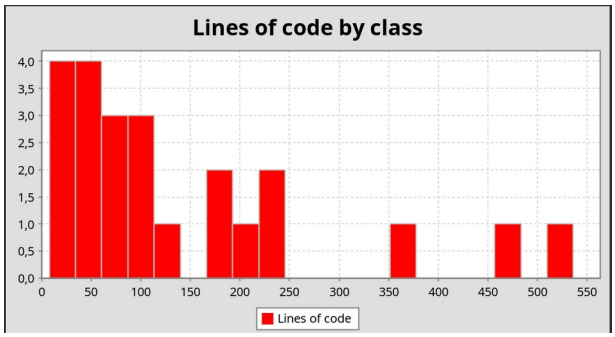
Durchführung und Messungen:

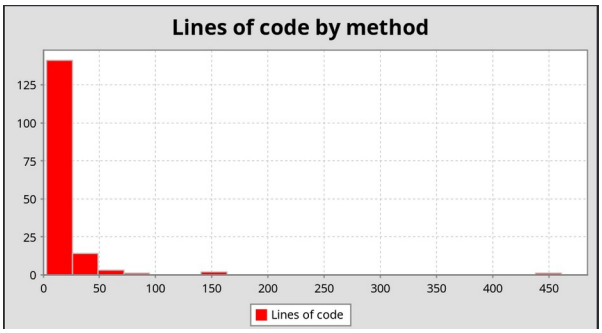
Wir haben uns entschieden, alle Messungen – bis auf "Lines of Code", die wir bereits vor MS3 einmal ausgewertet haben vor MS5 mithilfe von git checkout durchzuführen. Auf den folgenden Seiten werden unsere Resultate aufgelistet. Dabei listen wir jeweils pro Metrik alle Resultate aus den jeweiligen Meilensteinen auf. Auch wenn wir die finalen Messungen und Analysen erst vor MS5 durchgeführt haben, haben wir natürlich trotzdem versucht, unser Konzept einzuhalten, um so gut wie möglich unsere Codequalität zu sichern. Die Dokumentation mithilfe von JavaDoc hat bei uns sehr gut funktioniert, sodass nahezu unser gesamter Code mit JavaDoc kommentiert ist. Was wir hingegen nicht so häufig verwendet haben, waren die Bug Reports. Meistens haben wir unsere Bugs direkt über unseren Discord-Chat kommuniziert, da wir dort häufig in Kontakt miteinander standen. Eine andere Qualitätssicherungsmaßnahme, die sich sehr bewährt hat, war das Pair Programming. Es hat uns vor allem bei schwierigen Aufgaben wie dem Ping-Pong-Mechanismus und dem Aufbau der Client-Server-Interaktion geholfen.

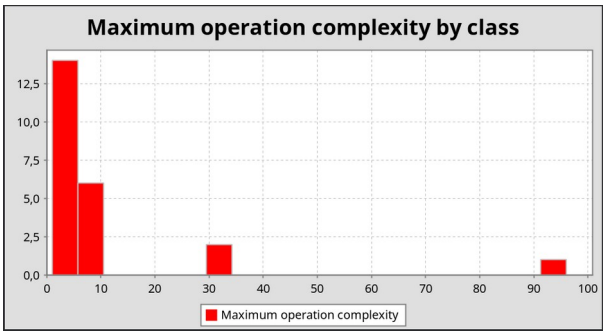
Darstellung Histogramme Lines of Code: Die x-Achse repräsentiert jeweils die Anzahl Code Linien und die y-Achse repräsentiert die Anzahl Klassen/ Methoden die diese Anzahl haben.

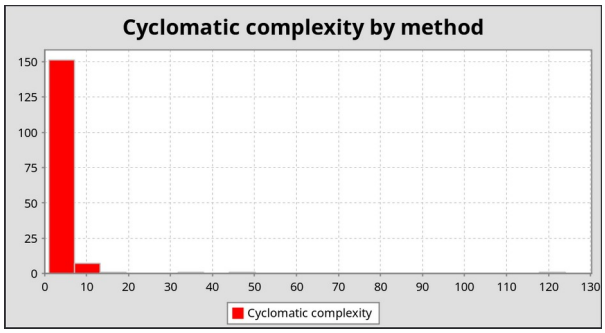
Darstellung Histogramme operation komplexität / zyklomatische Kompletität: Die x- Achse repräsentiert den Wert der zyklomatischen Komplexität und die y-Achse repräsentiert die Anzahl Methoden/Klassen, die diesen Wert haben.

Meilenstein 3 Messungen:



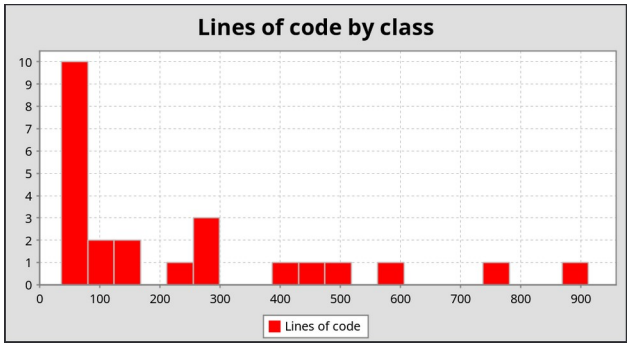


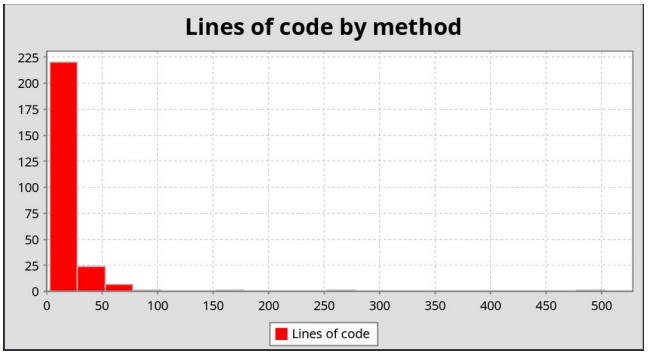


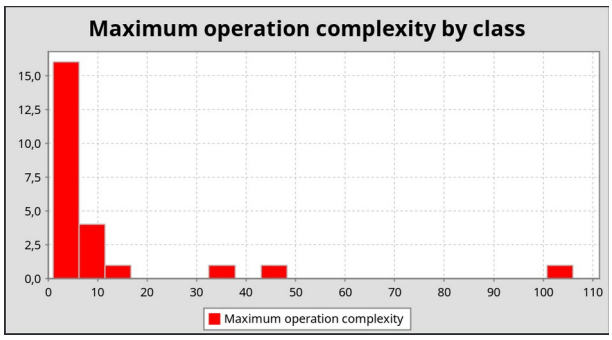


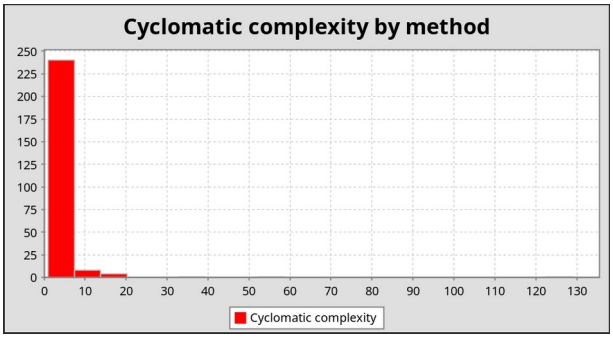
Mit den Messungen aus MS3 sind wir grundsätzlich zufrieden. Wir konnten unsere Vorsätze bis auf wenige Ausnahmen einhalten. Wie erwartet, hatten wichtige Klassen über 400 Zeilen Code, und eine zentrale Methode hatte weit mehr als 150 Zeilen. Dies empfinden wir jedoch nicht als problematisch, da es sich bei der Methode um den `readerServerLoop` handelt, über den ein Großteil der Client-Server-Interaktion läuft. Dasselbe lässt sich auch bei der zyklomatischen Komplexitätsmetrik beobachten. Uns ist bewusst, dass der Wert des `readerServerLoop` sehr hoch ist (er ist auf dem Diagramm kaum zu erkennen, da er eine deutliche Ausnahme darstellt).

Meilenstein 4 Messungen:









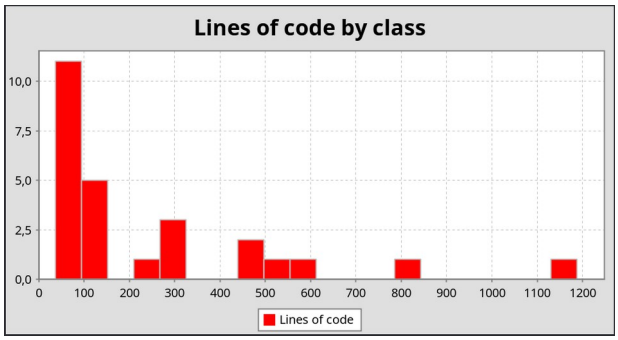
ColorRacer

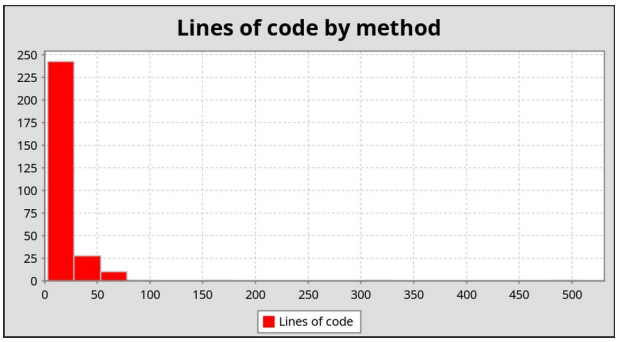
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ch.unibas.dmi.dbis.cs108.gui		0%		0%	219	220	931	932	141	142	8	9
ch.unibas.dmi.dbis.cs108.network		15%		5%	305	326	737	849	73	92	0	5
ch.unibas.dmi.dbis.cs108.server		58%		49%	147	264	386	880	26	97	1	7
ch.unibas.dmi.dbis.cs108.client		27%		2%	45	48	122	166	5	8	0	1
ch.unibas.dmi.dbis.cs108		0%		0%	6	6	21	21	3	3	1	1
ch.unibas.dmi.dbis.cs108.game		100%		96%	2	51	0	177	0	23	0	3
Total	7'977 of 12'120	34%	810 of 1'052	23%	724	915	2'197	3'025	248	365	10	26

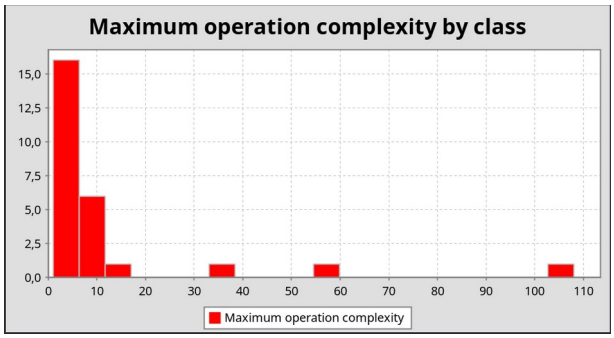
Zwischen MS3 und MS4 gab es bei den Lines of Code pro Klasse nochmals ein Wachstum. Dies lässt sich sehr gut mit den Anforderungen von MS4 erklären. Bis zu diesem Meilenstein sollten wir die GUI vollständig implementiert haben, und die Klasse `GameLobbyController` ist dabei stark gewachsen. Ansonsten sind die Klassen, die unser Ziel überschreiten, allesamt Klassen mit sehr wichtigen Funktionen – daher sind sie größer als unsere durchschnittlichen Klassen.Bei den Lines of Code pro Methode hat sich nichts Wesentliches verändert. Wir konnten unser Ziel von maximal 150 Zeilen pro Methode einhalten – mit Ausnahme der Methode `ProtokolReaderServer.loop`. Die meisten Methoden liegen weiterhin unter dem Wert von 25 Zeilen. Die Werte zur Komplexität unserer Klassen sind leicht angestiegen, bleiben jedoch insgesamt relativ stabil.

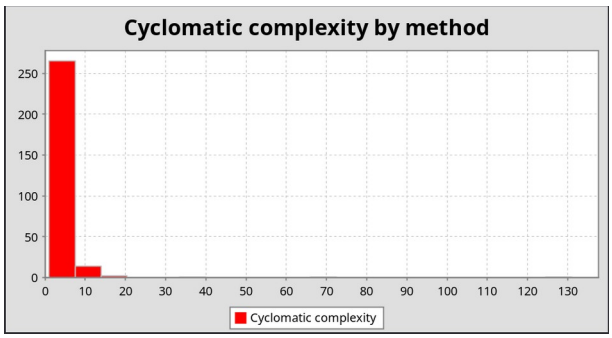
Mit diesem Meilenstein kommt nun auch die *Code Coverage*-Metrik hinzu. Wie auf dem Screenshot zu sehen ist, ist unser Code im Package `game` sehr gut durch JUnit-Tests abgedeckt, und das Package `server` weist eine Testabdeckung von fast 60 % auf.

Messungen Stand 9.5.25 :









ColorRacer

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ch.unibas.dmi.dbis.cs108.gui		0%		0%	219	220	931	932	141	142	8	9
ch.unibas.dmi.dbis.cs108.network		15%		5%	305	326	737	849	73	92	0	5
ch.unibas.dmi.dbis.cs108.server		58%		49%	147	264	386	880	26	97	1	7
ch.unibas.dmi.dbis.cs108.client		27%		2%	45	48	122	166	5	8	0	1
ch.unibas.dmi.dbis.cs108		0%		0%	6	6	21	21	3	3	1	1
ch.unibas.dmi.dbis.cs108.game		100%		96%	2	51	0	177	0	23	0	3
Total	7'977 of 12'120	34%	810 of 1'052	23%	724	915	2'197	3'025	248	365	10	26

Zwischen MS4 und MS5 mussten wir nicht mehr viel Code schreiben, daher sind die Werte – mit Ausnahme des `GameLobbyController` – weitgehend gleich geblieben. Insgesamt sind die Werte zwar etwas erhöht, jedoch nur minimal. Die Messung der *Code Coverage* ist identisch zu der aus MS4, da wir mit unserer Abdeckung bereits zufrieden waren und alle Tests in MS4 sinnvoll und erfolgreich „ grün“ durchgeführt wurden.

Fazit:

Im Großen und Ganzen sind wir mit dem Verlauf der Messungen zufrieden – insbesondere unter der Berücksichtigung, dass dies unser allererstes Projekt war und wir zum ersten Mal ein solches Konzept selbst schreiben, anwenden und evaluieren mussten.

Vieles hat gut funktioniert, wie z. B. die Verwendung von JavaDoc, Pair Programming, die Kontrolle der Lines of Code sowie das Black-Box-Testing. Andere Aspekte haben weniger gut funktioniert, etwa das Einhalten der zyklomatischen Komplexität. Wir hätten die `switch`-Case-Verzweigungen in einzelne Methoden oder Klassen aufteilen und so die Komplexität reduzieren können. Als Gruppe haben wir uns jedoch bewusst dafür entschieden, alle `switch`-Case-Verzweigungen in einer Klasse zu belassen, da uns dies die Entwicklung vereinfacht hat.

Rückblickend hätten wir das Logging konsequenter einsetzen sollen – wir haben es nur selten genutzt. Wahrscheinlich hätte es uns Zeit und Aufwand ersparen können. Auch hätten wir häufiger Bug Reports schreiben können. Innerhalb unserer Gruppe empfanden wir dies allerdings nicht als notwendig, da die Zahl der Bugs überschaubar blieb. Dennoch erkennen wir im Nachhinein, dass ein Bug-Tracking-System durchaus hilfreich sein kann.

Wenn wir das Projekt noch einmal durchführen würden, würden wir versuchen, bei der Kommunikation zwischen Server und Client von Anfang an stärker auf eine geringere zyklomatische Komplexität zu achten.

Besonders erwähnenswert ist, dass wir uns bei Klassen, die das Interface `Runnable` implementieren, aufgrund ihrer vergleichsweise hohen Komplexität große Mühe bei der Dokumentation gegeben haben. So ist es möglich, sich trotz vieler Codezeilen und komplexer Logik schnell einen Überblick zu verschaffen.