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-Prorgamming** 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 und der Markupsprache FXML 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 cameCase). Zudem sollten Namen immer aussagekräftig sein.
- Verständliche **JavaDocs** müssen während dem Programmieren geschrieben werden. Es sollte möglich sein, denn 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 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 die interessanten Messwerte JLOC (Javadoc Lines of Code) und Jm (Method Coverage) sind. 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 Code 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, 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 sobald wir JUnit Tests haben wird per JaCoCo geprüft, wie viel Prozent des Codes von Tests abgedeckt sind.

Tests:

Die Server-Client-Interaktion wird als zentrale Komponente mit JUnit-Tests getestet.

3. Aktuelle Messungen

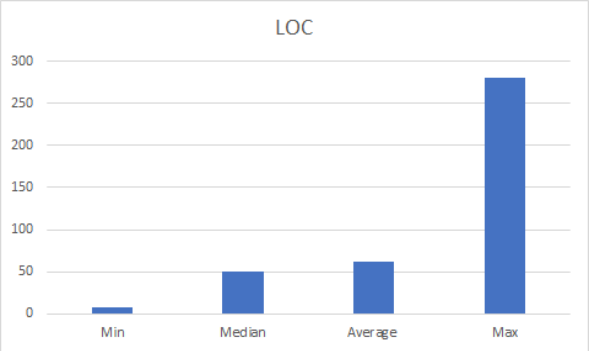
Logging Statements (als LOC)

error	warn	info	fatal
3	3	8	1

Code Coverage

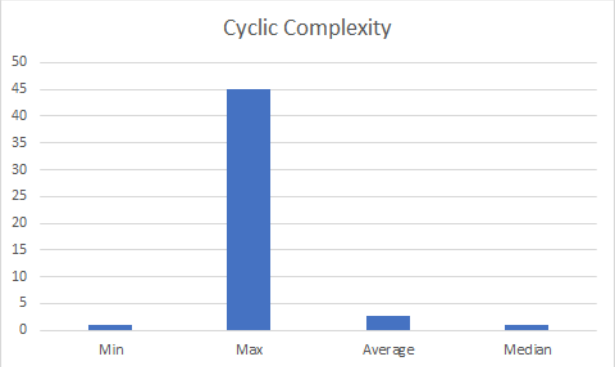
0% (noch keine JUnit-Tests vorhanden)

Lines of Code (16.04.2022)



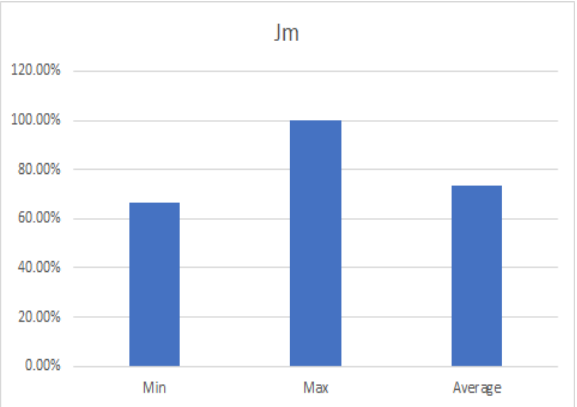
Min: 8 Zeilen (gameLogic.Node)
Max: 280 Zeilen (gameLogic.Game)
Median: 50 Zeilen (server.ConnectionToClientMonitor)
Average: 62.68 Zeilen

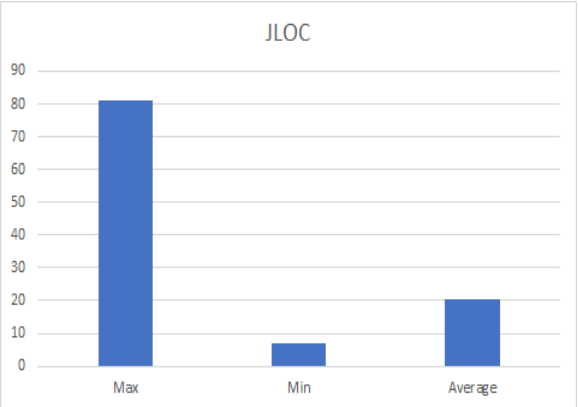
Cyclic Complexity (16.04.2022)



Min: 1 (100+ Methoden)
Max: 45 (server.ServerReceive.run())
Average: 2.69
Median: 1

JavaDoc (LOC und %)





Min: 66.67% (mehrere Klassen)
Max: 100.00% (mehrere Klassen)
Average: 73.30%

Min: 7 (gameLogic.Quiz, gui.Launcher)
Max: 81 (utility.io.CommandsToServer)
Average: 20.36