QA-Report Key-H(a)unt

Michel Romancuk, Lukas Schmidt, Yash Trivedi, Severin Memmishofer

May 2023

1 Konzept

1.1 Leitsätze der Qualitätssicherung

Die Qualitätssicherung unseres Projektes 'Key H(a)unt' ist aufgeteilt in vier Ziele:

1. Run-time Analyse und Debugging des Codes ist einfach.
2. Der Code lässt sich in wenigen Schritten testen.
3. Der Code ist verständlich und gut dokumentiert.
4. Änderungen im Code sind zielgerichtet an die Anforderungen

1.1.1 Run-time Analyse und Debugging des Codes

Die Verwendung eines zentral abstellbaren Loggers wie Log4j, der sowohl in Logfiles, als auch in Echtzeit die Nachverfolgbarkeit aller Abläufe im Code ermöglicht, soll das Debugging vereinfachen. Um genügend Verbosität zu erreichen, überprüfen wir die relative Anzahl Logging-Statements.

1.1.2 Testbarkeit des Codes

Kritische Stellen des Codes, wie die Game-Logik und die Netzwerkkommunikation werden durch Unit-Tests extensiv getestet. Bei Änderungen im Code müssen diese Tests (weiterhin) bestanden werden und die Code-Coverage dieser kritischen Stellen wird überprüft.

1.1.3 Verständlichkeit und Dokumentation des Codes

Die Dokumentation wird durch Javadoc umgesetzt. Ein einheitlicher Programmierstil wird in Anlehnung an die Sun Coding Conventions mittels dem Plugin Checkstyle überprüft, um die Lesbarkeit zu erhöhen. Cyclomatic complexity, relative Anzahl an Zeilen Javadoc und Zeilen Code pro Klasse werden gemessen, um die Code-Komplexität niedrig zu halten und Erweiterungen zu vereinfachen.

1.1.4 Erfüllung der Anforderungen

Gruppeninternes Code-Review unterstützt die oben genannten Ziele und garantiert, dass neue Features sowohl den vorgegebenen Meilensteinen, als auch den projektbezogenen Anforderungen an unser Produkt entsprechen.

2 Durchführung

Für die Messungen von Metriken wurden die Intellij-Plugins Metrics Reloaded sowie JaCoCo verwendet.

Während des Projekts haben wir meistens im Pair-Programming an Features gearbeitet. Dazu verwendeten die Gruppen jeweils eigene git-Branches. Dies erlaubte es uns, bei jedem Merge-Request mit den Anderen zusammen ein Code-Review zu machen und mögliche Anpassungen zu diskutieren. Wir benutzten auch auf der Seite von Gitlab die CI-Pipeline, um den Checkstyle durchzuführen und die Unit-Tests zu überprüfen.

3 Resultate

3.1 Meilenstein III

Im Meilenstein III wurden erstmals Messungen durchgeführt, die allerdings teilweise unvollständig waren, da wir zu jenem Zeitpunkt noch keine Unit-Tests hatten. Die Cyclomatic Complexity ist weitgehend unauffällig, zumindest im

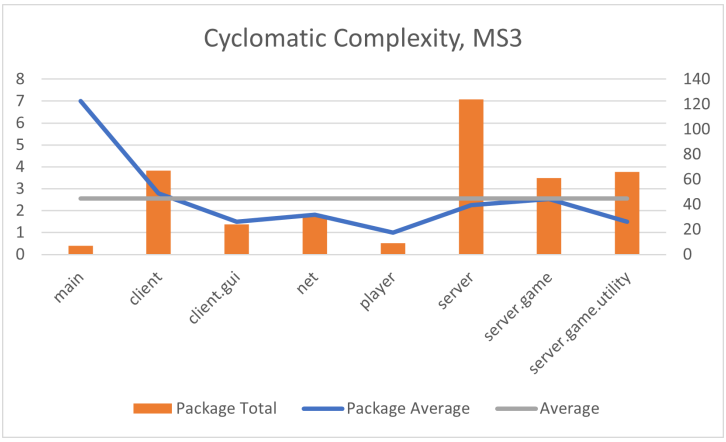

Abbildung 1: Cyclomatic complexity, Meilenstein III

Durchschnitt gemessen. Die Main-Einstiegsklasse ist hierbei eine Ausnahme, da von dort jeglicher Client- und Servercode gestartet werden kann und somit besonders viele Pfade durch den Code führen. Jedes andere Paket hat einen Durchschnitt unter 3; der Projektdurchschnitt (in Grau) liegt bei 2.55. Zum

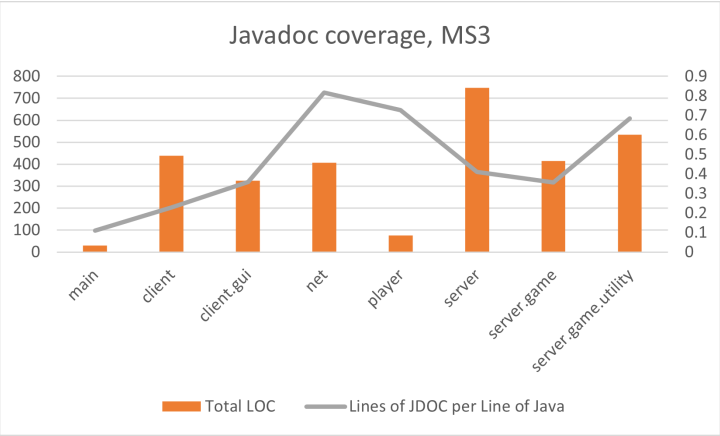

Abbildung 2: Javadoc Abdeckung, Meilenstein III

Meilenstein III waren von 2973 Zeilen Code 930 Zeilen Javadoc. Im Durchschnitt ergibt dies 0.45 Zeilen Javadoc pro Zeile Code. Besonders gut abgedeckt sind mit 92.3% die Methoden, und von den Paketen her das Netzwerkprotokoll. Hier werden Spitzenwerte von 0.81 Zeilen Javadoc pro Zeile Code erreicht.

Im Durchschnitt hatten die Klassen 79.25 Zeilen Code, im Minimum waren es 3 und im Maximum 291 auf Serverseite beim Task-Management.

Bei unseren ersten Messungen ist uns ein Fehler unterlaufen; wir hatten zum Stichtag 60 Logging-Statements, welches im Schnitt 0.029 Statements pro Zeile Javacode entspricht, also etwa jeder 35. Zeile. Der Fehler ist zurückzuführen darauf, dass bei der Analyse die Klein/Grossschreibung nicht ignoriert wurde.

3.2 Meilenstein IV

Im Meilenstein IV wurde hauptsächlich der Client auf GUI umgestellt, daher sank die Cyclomatic Complexity besonders des Client-Packages auf 2.1, was den Gesamt-Durchschnitt auf ansehnliche 2.08 senkte. Im Rest des Codes stieg die Komplexität höchstens minimal an.

Die Javadoc-Abdeckung sank minimal auf einen Durchschnitt von 0.43 Zeilen Javadoc pro Zeilen Java, was wiederum auf den Ausbau des GUI zurückzuführen ist. Insgesamt sind es hier 3792 Zeilen Code inklusive Javadoc.

Auf ähnliche Weise stieg die durchschnittliche Anzahl Zeilen pro Klasse auf 96.67, das Minimum ist immer noch bei drei aber das Maximum liegt nun bei 382 in der Controllerklasse des GUI. Der Logger wurde hier 72 Mal eingesetzt, im Schnitt waren dies 0.054 Statements pro Zeile, somit war etwa jede 20. Zeile Code ein Logger-Statement.

Im Meilenstein IV wurden unter Anderem auch Unit-Tests hinzugefügt. Getestet wurden die Netzwerk-Komponenten und die Gamelogik. Dies führt zu einer hohen Abdeckung in diesen Bereichen, was auf die Gesamtheit gesehen

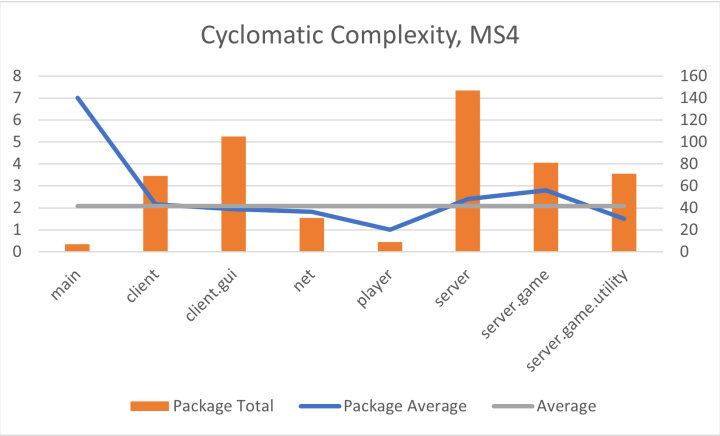

Abbildung 3: Cyclomatic complexity, Meilenstein IV

immer noch 33% aller Methoden abdeckt.

3.3 Meilenstein V - Stand 14.05.2023

Wir haben beschlossen, die Messungen schon jetzt zu machen, da der Code sich in der kommenden Woche nur noch minimal verändern wird. Alle unsere relevanten Features sind eingebaut und umgesetzt.

Im Meilenstein 5 veränderten sich die Metriken kaum. Einige Checks wurden auf Server-seite der Logik hinzugefügt und der Client minimal ausgebaut, da der Schwerpunkt für unsere Gruppe zu dieser Zeit definitiv auf ästhetischen Aspekten lag. Durch diese stieg die durchschnittliche Cyclomatic Complexity minimal auf 2.085. Wie zu erwarten war, veränderte sich die Javadoc Coverage nicht und die Gesamtzahl an Code-Zeilen stieg auf 3935. Die durchschnittliche Anzahl Zeilen pro Klasse stieg auf 100.48, was besonders auf den Feinschliff des GUI zurückzuführen ist - das Maximum liegt immernoch in der Controller-Klasse mit nun 408 Zeilen Code. Zum letzten Feinschliff gehörte auch die Ausbesserung der Logger-Verwendung: insgesamt sind nun 81 Logging-Statements verteilt und die Logging-Statements selber nochmal überholt. Somit sind wir bei 0.058 Statements pro Zeile Code, oder jeder 17. Zeile.

Die Unit-Test-Abdeckung blieb unverändert und unterlag lediglich den kleinen Additionen des Codes, so dass die Methoden-Abdeckung nun am Ende bei 34.59% liegt.

4 Diskussion

Über die Metriken, die wir gemessen haben, sind wir sehr zufrieden. Die durchschnittlich überall unter 2.5 gemessene Cyclomatic Complexity beweist uns,

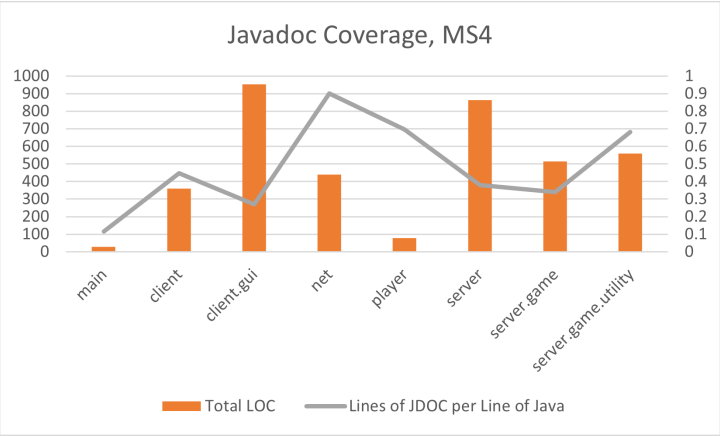

Abbildung 4: Javadoc Abdeckung, Meilenstein IV

dass wir durch unsere Arbeitsweise einen sehr nachvollziehbaren Code geschrieben haben - der Logik nach. Die Peer-Reviews sorgten (mindestens) intern dafür, dass dieser auch verständlich und lesbar ist. Dem wird auch durch die flächendeckende Javadoc-Dokumentation unter die Arme gegriffen - dreissig Prozent unserer Zeilen sind Javadoc. Dies ist natürlich in erster Linie nur eine Zahl, aber wir glauben, dass diese auch sinnvoll gefüllt sind. Die Klassengrösse ist zumindest im Durchschnitt mit etwa 100 auch überschaubar und ordentlich.

Zu bemängeln wäre einzig die Abdeckung unserer Codebase mit Unit-Tests. Zwar sind die Kernelemente Netzwerk und Game-Logik grösstenteils abgedeckt, wäre dies aber nun ein längerfristiges Projekt, mit viel Veränderung und Ausbau, dann genügten diese schon bald nicht mehr. Zu unseren Gunsten muss man aber auch sagen, dass GUIs, besonders mit FXML, sich nicht gut testen lassen und dies wohl den grössten Teil unser Client-Infrastruktur ausmacht. Dennoch gibt es hier bedeutend Luft nach oben.
Der Logger wurde zwar genügend implementiert und Log-Files zeitens auch für Debugging verwendet, allerdings zweifeln wir, dass wir ihn zu seinem vollen Potenzial eingesetzt haben. Wir vermuten, dass dieser unersetzlich wird bei Codes, die etwas grösser sind, doch von all den Zielen und Massnahmen der Qualitätssicherung war dieser am wenigsten relevant.

Wir sind zudem sehr froh, dass wir qualitätssichernden Massnahmen, wie Javadoc, Checkstyle und Peer-Reviews schon seit Meilenstein I benutzt haben. Dies hat den Prozess und Unterhalt hoher Standards massiv vereinfacht.

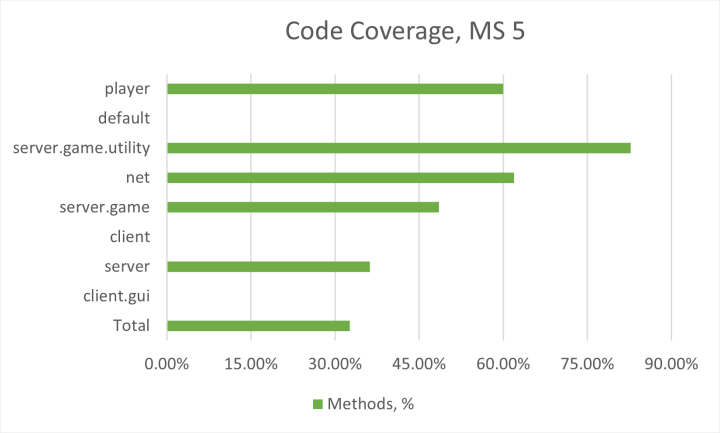

Abbildung 5: Code Coverage, Meilenstein V