QA-Report

Universität Basel FS 22

Programmierprojekt bei Prof. Dr. Heiko Schuldt

Tutorin: Flurina Fischer

Gruppe08 – Night Train to Budapest:

Sebastian Lenzlinger

Jonas Biedermann

Alexandr Sazonov

Seraina Schöb

Inhalt

1 Konzept

Um die Qualität der Software des Projekts sicherzustellen, werden wir uns an folgenden Qualitätsmerkmalen[1] orientieren:

Bedienbarkeit	**Testbarkeit**	**Verständlichkeit**	**Korrektheit**
- Erreichbarkeit	- Strukturiertheit	- Prägnanz	
- Barrierefreiheit	- Abgeschlossenheit	- Lesbarkeit	
- Mitteilungsgüte			

Die Sicherung dieser Merkmale wird in drei Prozessen stattfinden: Beim Coden selbst, in Testungen und in Messungen von Metriken und deren Auswertungen. In den folgenden Abschnitten werden alle drei Prozesse erläutert.

1.1 Coden

Die Qualität des Codes wird bereits beim Coden selbst durch einige Massnahmen sichergestellt. Hier zählen vor allem Lesbarkeit, Prägnanz und Strukturiertheit zu den Fokuspunkten aber auch die Abgeschlossenheit soll gefördert werden. Es werden folgende Massnahmen ergriffen:

(i) Verwendung des *Google styleguides; d*adurch wird die Einheitlichkeit des Codes sichergestellt und somit die Lesbarkeit und Strukturiertheit erhöht.
(ii) *Documentation-oriented-coding*; Es soll in einem ersten Schritt zuerst immer eine Dokumentation erstellt werden, in welcher die angestrebte Funktionalität der Klasse oder Methode beschrieben wird, dann wird implementiert und anschliessend die Dokumentation angepasst und ergänzt
(iii) Mittels Checkliste[2] wird sowohl eine sofortige Selbstprüfung nach der Codingsession als auch eine Prüfung im *Buddy-System* durchgeführt. Dabei wird der Code der Partner:in so schnell als möglich durchgeschaut und eine Rückmeldung gegeben.
(iv) Zweimal wöchentlich - an fixen Sitzungsterminen - wird der Fortschritt und der Code vorgestellt und das weitere Vorgehen besprochen.
(v) Die systematische Verwendung eines Loggers entsprechend den Konventionen.[3]
(vi) Schlussendlich werden schon während des Codingprozesses Ideen für Unittests gesammelt, so dass das Implementieren der Tests leichter fällt.

1.2 Testen

Um die Korrektheit und Bedienbarkeit der Applikation zu gewährleisten, werden Tests durchgeführt. Diese sind in 2 Kategorien eingeteilt:
1) White Box Tests: Mit Unittests wird die Funktionalität der zentralen Bestandteile systematisch überprüft. Damit wird sichergestellt, dass die Teile tun, was sie tun sollen. Die ersten Tests werden gegen Anfang des 4. Meilensteins durchgeführt um die ersten Funktionalitäten der Spiellogik zu Prüfen.
2) Black Box Tests: Vor Abgabe jedes Meilensteins wird die Applikation in ihrer Gesamtheit geprüft, um sicherzustellen, dass die Teile im Ganzen tun, was sie sollen. In Meilenstein V wird dieses

[1] Angelehnt an Boehm et al 1976, S. *595*
[2] Vgl. Appendix
[3] Ebd.

Testverfahren vermehrt angewendet werden, um die Erreichbarkeit und Mitteilungsgüte der Benutzeroberflache zu testen und zu verbessern. Falls der Zeitplan es erlaubt, soll in dieser Phase auch die Barrierefreiheit so gut als möglich gewährleistet werden.

1.3 Messen

Um einige der Qualitätsmerkmale quantifizieren zu können, verwenden wir verschiedene Metriken in regelmässigen Abständen (einmal wöchentlich) messen. Die Auswahl der Metriken orientiert sich an den genannten Qualitätszielen, namentlich Testbarkeit und Verständlichkeit.

a) Lines of Code per Class

Diese Metrik besteht aus einer einfachen Zählung von Codelinien im Quellcode. Wir fokussieren uns dabei auf zwei Messungen: (1) *Comment Lines of Code*; hier werden die Kommentarlinien in jeder Klasse gezählt, Leerzeilen werden nicht mitgezählt. (2) *Absolute Lines of Code*; dies zählt die Anzahl Codezeilen inklusive Kommentarzeilen in jeder Klasse ohne Leerzeilen.

Ausgewertet werden die Messergebnisse folgendermassen: Kommentarzeilen werden in ein Verhältnis zur Gesamtzahl der Zeilen gesetzt, was einen Rückschluss auf die Verständlichkeit und Abgeschlossenheit des Programmes zulässt. Weiter ergibt die Differenz der Gesamtzahl der Zeilen und der Kommentarzeilen einen Einblick in die Prägnanz der Codierung. Schlussendlich gibt uns der Unterschied zwischen zwei Messungen in zeitlichem Abstand eine grobe Einsicht in den Fortschritt des Projektes.

b) Zyklomatische Komplexität[4]

Um die Komplexität unseres Programms zu quantifizieren, soll sowohl die Zyklomatische Komplexität nach McCabe für Methoden als auch die *Weighted Methods per Class* nach Chidamber und Kemerer gemessen werden.
Dabei ist die Grundlegende Idee von McCabe, dass die Schwierigkeit eines Programmes von dessen möglichen Flüssen abhängt. Die Metrik hat dabei folgende Bedeutung: Bis 1o «niedrig», bis 20 «mittel» bis 50 «hoch» und ab 50 «undurchschaubar». Je kleiner die Zahl also ist, desto höher ist die Testbarkeit (weniger Fälle) und die Verständlichkeit.

c) Dependants per Class

Diese Metrik misst wie viele andere Klassen von einer Klasse abhängig sind und wie viele Abhängigkeiten jede einzelne Klasse hat. Dies gibt uns einen Einblick in das Abstraktionslevel einzelner Klassen und somit in die Verständlichkeit unseres Quellcodes. Dabei wird dieser Wert immer ins Verhältnis mit der Gesamtzahl aller Klassen gesetzt.

d) Code Coverage

Um die Wirksamkeit unserer Unittests zu prüfen und zu optimieren, wird gemessen, wieviel Code von den Tests abgedeckt wird. Die Abdeckung soll im möglichen Rahmen maximiert werden.

e) Number of Logging-Statements

Diese Metrik soll die Anzahl der Logging-Statements messen und im Verhältnis zu *Lines of Code* gesetzt werden. Dies gibt uns eine Quantifizierung der Verständlichkeit des Quellcodes.

[4] Vgl. Schneider 2021, Kapitel *4.3.2 Zyklomatische Komplexität von McCabe*

2 Messungen

Die Messungen werden mit den JavaPlugins JaCoCo und MetricsReloaded gemessen und anschliessend ausgewertet.

2.1 Messung 3. April 22

Es wurden die drei Metriken *Lines of Code*, *Cyclomatic Complexity* und *Dependency* sinnvoll gemessen und die Maxima, Minima und Mittelwerte errechnet, was in nachstehender Tabelle ersichtlich wird Es wurden noch keine Tests durchgeführt, weswegen die *Codecoverage* hier nicht aufgeführt ist:

Tabelle 1: Messdaten der ersten Messung. Dabei steht *LoC* für Lines of Code, die *Abs LoC* sind Codezeilen ohne Kommentare, die Cyklomatische Complexität wurde als Durchschnitt aller Methoden in einer Klasse berechnet und alle Messwerte beziehen sich auf Klassen.

Messung 1	Comment LoC	LoC	Abs LoC	Logg per 100 LoC	WMC	Cycl Compl.	Dependencies	Dependants
Max	42.0	155.0	113.0	0.0	21.0	7.0	5.0	20.0
Average	11.5	38.9	27.4	0.0	5.0	2.2	2.2	2.1
Min	0.0	4.0	2.0	0.0	0.0	0.0	0.0	0.0

2.2 Messung 30. April 22

Es wurden alle Metriken die in 1.3 beschrieben sind durchgeführt, ihre die Maxima, Minima und Mittelwerte berechnet. Die Logger-Statements wurden auf 100 Codezeilen normalisiert:

Tabelle 2: Messdaten der zweiten Messung. Analog zu Tabelle 1.

Messung 2	Comment LoC	LoC	Abs LoC	Logg per 100 LoC	WMC	Cycl Compl.	Dependencies	Dependants
Max	178.0	568.0	390.0	9.3	83.0	23.0	17.0	32.0
Average	24.1	75.4	51.3	1.6	9.8	2.3	3.5	3.3
Min	0.0	2.0	2.0	0.0	0.0	0.0	0.0	0.0

Ebenso wurde die *Code Coverage* bestimmt. Die verfügbaren *JUnit-Tests* beziehen sich nur auf einen Aspekt der Spiellogik und decken 10% der Klassen der Spiellogik, 5% aller Methoden der Spiellogik und 2% aller Codezeilen der Spiellogik ab.

2.3 Messung 18.Mai 22

Die Messungen wurden analog zu 2.2 durchgeführt und wo nötig zu weiterführenden Daten verrechnet:

Tabelle 3: Messdaten der dritten Messung. Analog zu Tabelle 1 und 2.

Messung 3	Comment LoC	LoC	Abs LoC	Logg per 100 LoC	WMC	Cycl Compl.	Dependencies	Dependants
Max	187.0	624.0	517.0	10.5	94.0	11.5	18.0	40.0
Average	23.9	95.5	71.6	1.7	13.4	1.7	4.0	3.8
Min	0.0	2.0	2.0	0.0	0.0	0.0	0.0	0.0

Wiederum wurde die *Code Coverage* bestimmt. Die Tests haben sich aber seit der letzten Messung leider nicht verändert, ebenso wenig die Spiellogik, weshalb auch die dritte Messung auf eine Codeabdeckung von 10% der Klassen der Spiellogik, 5% aller Methoden der Spiellogik und 2% aller Codezeilen der Spiellogik kommt.

3 Auswertung

Um die Messdaten aus Kapitel 2 auszuwerten, wurde jeweils ein Linien-Plot pro Metrik erstellt, und die Maxima, Minima und Mittelwerte zu jeder Messung dieser Metrik in diesem Plot dargestellt. Für die *Lines Of Code* Metrik wurden die Absoluten Codezeilen verwendet, für die Komplexität der Berechnete Durchschnitt jeder Methode einer Klasse und für die Abhängigkeit wurde nur die *Dependencies* nicht die *Dependants* der Klassen berücksichtigt. Dadurch ergibt sich nicht nur ein Überblick aller Messdaten, sondern auch ein Überblich über das Verhalten der Metriken über Zeit.

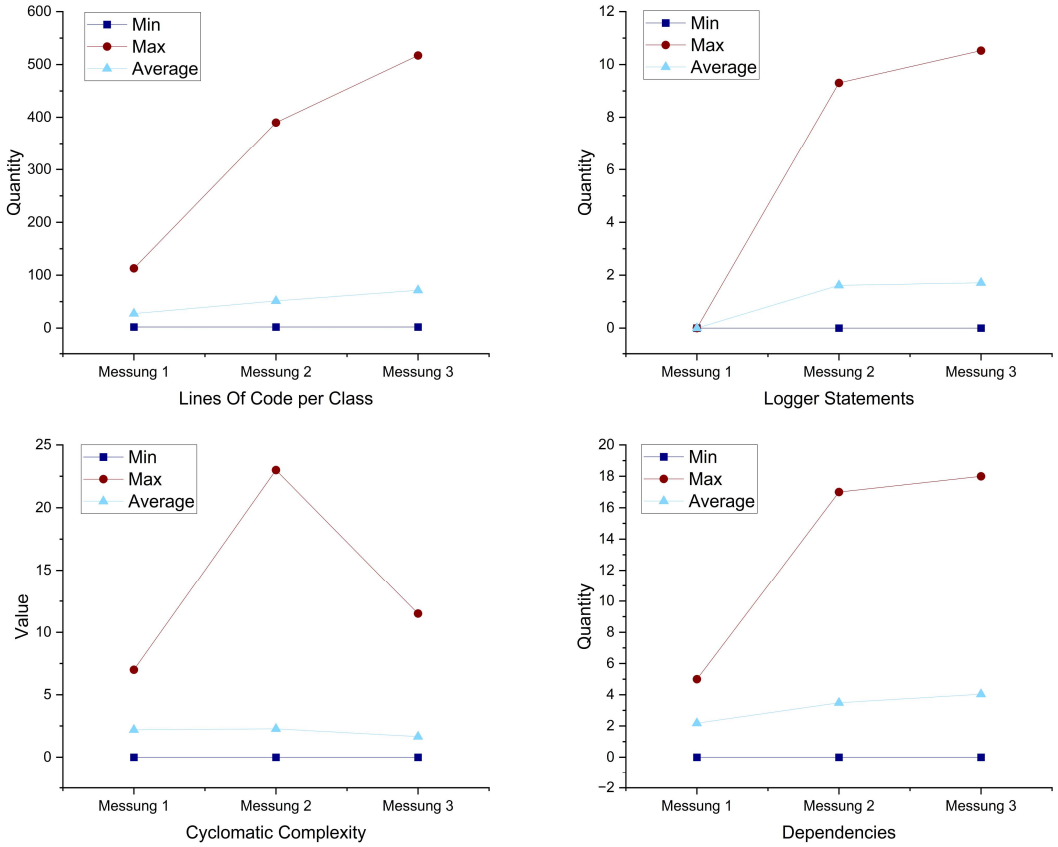

Abbildung 1: Messdaten aus Kapitel 2 dargestellt in Liniendiagrammen. Dabei sind die *Quantity* Achsen jeweils effektive Anzahl-Zählungen (z.B. 1 in *Lines of Code* bedeutet genau eine Codezeile). Die Achse der Komplexität verhält sich etwas anders, da die Komplexität selbst eine berechnete Grösse und nicht eine einfache Zählung ist (siehe 1.3 b).

Die Metriken für die Anzahl der Codezeilen, der Logger-Statements und der Abhängigkeit verhalten sich alle ähnlich. Die Maxima steigen stetig an, während Durchschnitt und Minimum auf ähnlichem Niveau verbleiben. Die scheinbare Parallelität zwischen Codezeilen und Logger-Statements ist naheliegend, da mehr Codezeilen in einer Klasse die Notwendigkeit von Loggern steigert, um die Übersichtlichkeit zu bewahren. Ein umgekehrter Trend wäre eher besorgniserregend. Auch ein leichter Anstieg der Abhängigkeiten der Klassen von anderen Klassen ist erwartungsgemäss. Die Maximalwerte dieser Metriken, vor allem aber der Codezeilen zeigen auf, dass es Klassen gibt denen ein gewisses Abstraktionslevel fehlt. Der Starke Anstiegt aber von Messung 1 und Messung zwei ist wohl auf die Implementation des GUIs zurückzuführen. Die Abstraktion von *Controller-Klassen* für das GUI fiel uns schwer.

Auch der Durchschnitt der Zyklomatischen Komplexität ist etwa gleichbleibend und mit einem Wert um zwei, ist diese im niedrigen Bereich. Der Maximalwert jedoch überstieg bei Messung 2 die zwanziger Marke und war deswegen als hoch einzustufen. Diese Komplexität ist unserem Protokoll-Parser geschuldet, welcher mittels Switch-Cases agiert. Bis zur Messung 3 wurde dieses Konstrukt bewusst vereinfacht und befindet sich nun in einem akzeptablen Mittleren Bereich.

Die Minimalwerte aller Metriken zeigen, dass es mindestens eine Klasse gibt, welche wahrscheinlich keinen Zweck erfüllt und deswegen gelöscht werden sollte.

Die Messungen der *Code Coverage* für unser Projekt sind leider völlig uninteressant. Da bis zum jetzigen Zeitpunkt nur ein Test erstellt wurde und dieser nur eine einzige Klasse abdeckt. Deshalb sind sie hier nicht aufgeführt. Die Abdeckung ist aber mit 2% aller Codezeilen der Spiellogik unzureichend.

3 Diskussion

Alles in allem ist das Ergebnis der Qualitätssicherung in Ordnung. Vor allem die Prinzipien aus 1.1 haben uns geholfen den Code übersichtlich zu gestallten. Die Dokumentation ist besser geworden und der Code besser lesbar. Trotzdem würden wir in einem neuen Projekt von Anfang an auf Stärkere Abstraktion setzten, wobei Klassen nicht völlig überladen werden. Dies würde die Lesbarkeit des Codes vor allem für Aussenstehende immens erhöhen.

Unzufrieden sind wir mit den Ergebnissen des Testen. Es hat sich als schwieriger herauskristallisiert als antizipiert. Bei einem Neubeginn begännen wir wohl früher mit der Fokussierung auf Tests. Das heisst ein Prinzip des *Testing-oriented-coding* würde weiter helfen um die Testbarkeit zu erhöhen.

Literaturverzeichnis

Schneider K.: *Abenteuer Softwarequalität - Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement,* dpunkt.verlag, Heidelberg, DE: 2012

Boehm B.W.: «Software Engineering» in *IEEE Transactions on Computers*, Vol. 25, Issue 12, pp. 1226 - 1241, Washington DC, USA: 1976

Boehm B. W., Brown J. R., and Lipow. M.: «Quantitative evaluation of software quality» in *Proceedings of the 2nd international conference on Software engineering.* IEEE Computer Society Press, pp. 592–605, Washington DC, USA,: 1976

Appendix[5]

Group-Conventions: Log4J Levels meaning:

FATAL	Is fatal for the application (i.e., crashes), needs to be addressed **right now.**
ERROR	Harmful but not fatal, needs to be addressed ASAP, but not everything needs to be stopped right away.
WARN	Needs to get looked at, at some point but is not the highest priority.
INFO	What is the code doing here (i.e., give a return value)
DEBUG	Wherever you would put a *System.out.println()* statement to test your code, instead log with debug.
TRACE	No assigned use so far.

Checklist Code Review:

- Kann ich diesen Code verstehen?
- Hält sich der Code an den *Google styleguide*?
- Ist alles Nötige in der *Dokumentation* vermerkt?
- Ist dieser Code sinnvoll platziert?
- Ist dieser Code redundant?
- Wie lässt sich dieser Code testen?
- Wie kann dieser Code noch verständlicher gemacht werden?
- …

[5] Es wird hier mit Foliensatz 6 *Software-Qualitätssicherung* Folie 65 davon ausgegangen, dass der Appendix mit Checkliste nicht zum Seitenlimit zählt.