

Softwarequalitätssicherungskonzept JDogs

Im Folgenden werden einige Massnahmen beschrieben, wie wir in Zukunft sicherstellen wollen, dass sich unsere Software unseren Vorstellungen entsprechend verhält und so wenig fehlerhaftes Verhalten wie möglich zeigt.

Konstruktives Qualitätsmanagement

- Wir treffen uns zusätzlich zur Übungsstunde jede Woche am Dienstag um 15:30 Uhr und am Freitag um 10 Uhr, um erledigte sowie anstehende Arbeiten zu besprechen. Wir erscheinen pünktlich zu den vereinbarten Treffen. Bei kurzfristigen Änderungen informieren wir im Gruppenchat auf WhatsApp.
- Wir verwenden – wie verlangt – Javadoc, um unseren Code zu dokumentieren. Dabei möchten wir uns an «Design by Contract» halten.
- Exceptions sollten, wenn möglich, sofort behandelt werden, oder es sollte zumindest vermerkt werden, von wem diese Fehlermeldung behandelt werden sollte.
- Wir verwenden einen WhatsApp-Chat, um uns über den Stand der Dinge auf dem Laufenden zu halten und um Probleme mitzuteilen.
- Wir besprechen die Anforderungen, die wir an unser Spiel stellen, detaillierter als bisher und halten diese in Checklisten fest, welche schliesslich auch zur Überprüfung der Software dienen. Damit es zu so wenig Missverständnissen wie möglich kommt, ist es zentral, dass die unterschiedlichen Vorstellungen kommuniziert werden und wir uns auf eine Version einigen, bevor wir mit der Umsetzung von Details beginnen.

Analytisches Qualitätsmanagement

- Wenn eine Person eine grössere Arbeit erledigt hat, erklärt sie den anderen ihren Code, sodass danach alle mit diesem Code weiterarbeiten können (Structured Walkthrough). Dazu ist es notwendig, dass zuvor die Aufgaben klar verteilt werden und allen klar ist, wer was macht.
- Zusätzlich führen wir Code Reviews durch, wenn eine Komponente fertig gestellt wurde. Dabei stützen wir uns auf die Checkliste aus der Vorlesung (siehe Anhang). Auch dazu ist es wichtig, dass wir im Vorhinein festlegen, wann eine Komponente soweit fertig ist, dass sie überprüft werden kann, bevor sie weiterentwickelt wird.
- Um den Black-Box- und den White-Box-Test durchzuführen, helfen die Listen mit den Anforderungen, die zuvor erstellt wurden. Zudem überlegt sich die zuständige Person entsprechende Äquivalenzklassen und erstellt eine Checkliste, die von den anderen Gruppenmitgliedern ggf. ergänzt werden kann. Der Black-Box- und der White-Box-Test sollte jeweils von einer Person durchgeführt werden, die den zu testenden Code nicht geschrieben hat.
- Bugs wollten wir ursprünglich auf Github mittels des Issue Trackers den anderen mitteilen. Stattdessen haben wir aber den WhatsApp-Chat benutzt. Unabhängig davon sollte der gefundene Fehler so genau wie möglich beschrieben werden (Voraussetzungen, Schritte zur Reproduktion, Fehlermeldungen, etc.).
- Wir wollen JUnit-Tests als Hilfsmittel benutzen, um korrekt funktionierenden und wartbaren Code zu schreiben.
- Wir verwenden einen Logger, Log4j2, um besser zu verstehen, wie das Programm abläuft und um auf Fehler aufmerksam zu werden.
- Wir wollen CPU-Usage-Tests einsetzen, um unsere Software zu optimieren.
- Zudem möchten wir Pair Programming (auch zu dritt) bei Gelegenheit einsetzen.
- Die Teammitglieder sollten ihren Code regelmässig einem Quietschentchen o.ä. erklären, um selbst auf allfällige Fehler aufmerksam zu werden.

Messungen

Wir führten für den Meilenstein 3 die folgenden Messungen durch.

Javadoc method coverage by class (Stand 18.04.2021)

Untenstehendes Histogramm zeigt, wie viele Methoden pro Klasse mittels Javadoc dokumentiert sind. Wir haben diese Metrik ausgewählt, da die Dokumentation mithilfe von Javadoc eine unserer Massnahmen ist, um die Qualität der Software sicherzustellen. Wir sehen, dass es ca 27 Klassen gibt, bei denen zwischen 0% und 5% der Methoden mittels Javadoc dokumentiert sind. Im Durchschnitt haben 26.82% der Methoden einer Klasse eine Javadoc-Dokumentation. Die Klasse MainGame nimmt mit 84.62% mittels Javadoc dokumentierten Methoden einen Spitzenplatz ein. Wir möchten in Zukunft mehr Methoden pro Klasse dokumentieren, um die Verständlichkeit des Codes zu erhöhen.

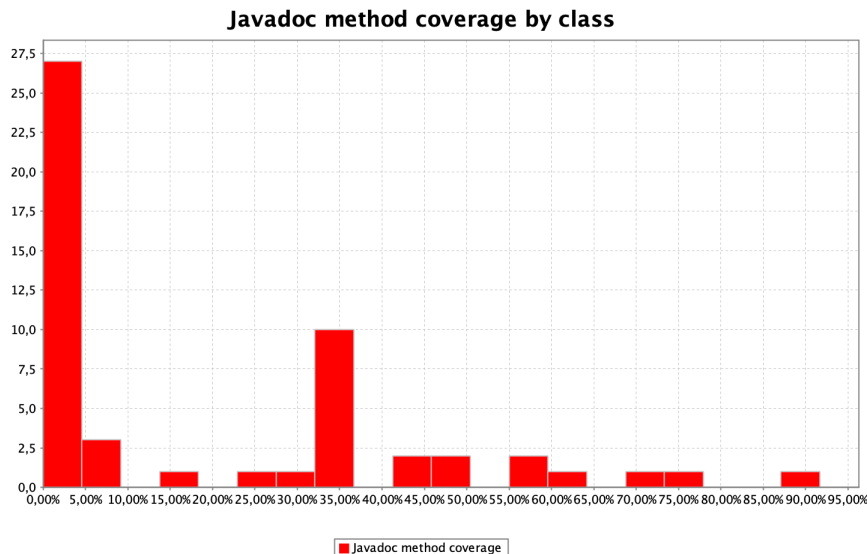


Abbildung 1: Javadoc method coverage by class mit Metrics Reloaded.

Lines of code by class (Stand 18.04.2021)

Dieses Histogramm zeigt die Verteilung der Anzahl Codezeilen über die Klassen. Man sieht, dass es 17 Klassen gibt, die zwischen 1 und 25 Zeilen Code beinhalten. Die meisten Klassen beinhalten zwischen 1 und 125 Zeilen Code. Die Klasse ServerGameCommand ist am längsten, sie verfügt über 629 Zeilen Code und bildet damit eine Ausnahme. Um den Code übersichtlicher zu halten, möchten wir diese Klasse auf mehrere Klassen aufteilen. Am wenigsten Zeilen hat die Klasse jDogs.Example mit 4 Zeilen, die jedoch lediglich zum Ausprobieren gedacht ist. Die Klasse mit den wenigsten Codezeilen, die verwendet wird, ist jDogs.board.TrackTile mit 5 Codezeilen. Im Durchschnitt hat eine Klasse 87.92 Zeilen Code.

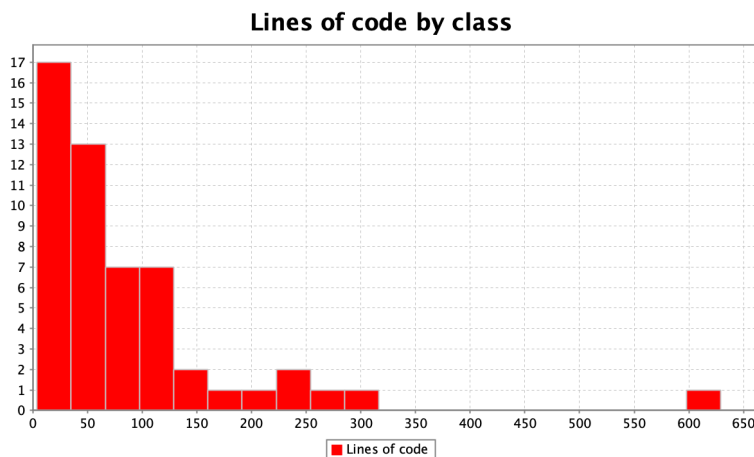


Abbildung 2: Lines of code by class mit Metrics Reloaded.

Lines of code by method (Stand 18.04.2021)

Dieses Histogramm zeigt die Anzahl Codezeilen pro Klasse. Darin enthalten sind auch die Kommentare und JavaDocs-Kommentare. Am meisten Codezeilen hat die Methode `ServerMenuCommand.execute(String)` mit total 152 Zeilen, wovon 9 Zeilen Kommentare sind. Wir möchten diese Methode in kleinere Methoden aufteilen, um den Code verständlicher zu gestalten. Am wenigsten Zeilen hat die Methode `LobbyController.displayOnGoingGamesInLobby(String)` mit lediglich 2 Codezeilen. Im Durchschnitt verfügt eine Methode über 11.72 Zeilen Code, wovon 9.66 Zeilen keine Kommentare sind.

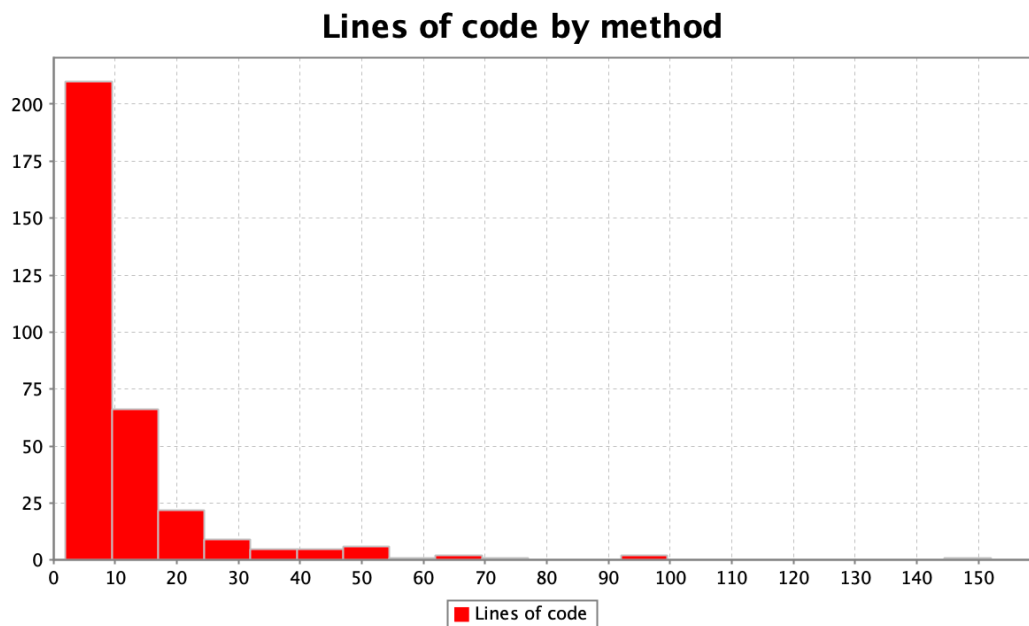


Abbildung 3: Lines of code by method mit Metric Reloaded.

Code Coverage

Code Coverage zeigt, in welchem Umfang der Code ausgeführt wurde. Die Analyse mit Jacoco ergab für das Package `jDogs.serverclient.clientside` folgendes Resultat.

jDogs.serverclient.clientside

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ClientGameCommand	<div><div></div></div>	14%	<div><div></div></div>	0%	11	13	27	35	1	3	0	1
ClientGameProtocol	<div><div></div></div>	0%	<div><div></div></div>	0%	4	4	13	13	2	2	1	1
ClientMenuCommand	<div><div></div></div>	57%	<div><div></div></div>	28%	10	15	17	39	0	3	0	1
SendFromClient	<div><div></div></div>	52%	<div><div></div></div>	75%	4	9	25	46	2	5	0	1
Client	<div><div></div></div>	79%	<div><div></div></div>	n/a	5	11	17	53	5	11	0	1
MessageHandlerClient	<div><div></div></div>	66%	<div><div></div></div>	41%	6	9	10	27	1	3	0	1
KeyboardInput	<div><div></div></div>	42%	<div><div></div></div>	16%	4	6	11	18	1	3	0	1
ReceiveFromServer	<div><div></div></div>	65%	<div><div></div></div>	83%	2	6	12	27	1	3	0	1
ConnectionToServerMonitor	<div><div></div></div>	73%	<div><div></div></div>	50%	1	3	2	10	0	2	0	1
ClientMenuProtocol	<div><div></div></div>	98%	<div><div></div></div>	75%	1	4	1	20	0	2	0	1
Total	498 of 1.147	56%	44 of 69	36%	48	80	135	288	13	37	1	10

Abbildung 4: Codecoverage mit Jacoco.

Anhang

Checkliste Code Review

1. Verstehe ich diesen Code?
2. Hält sich der Code an den vereinbarten Programmierstil?
3. Würde ich diesen Code an dieser Stelle im Projekt erwarten?
4. Hat es bereits Code mit ähnlicher Funktionalität an anderer Stelle?
5. Lässt sich die Lesbarkeit dieses Codes erhöhen?
6. Könne ich diesen Code warten und Änderungen vornehmen?
7. Ist der Code ausreichend dokumentiert?

Checkliste Milestone3

1. Lobby
 - a. Spielstatistik kann aufgerufen werden.
 - b. Andere Spieler werden angezeigt.
 - c. Public Chat funktioniert in der Public Lobby.
 - d. Public Chat funktioniert, in und von der GameLobby.
 - e. WhisperChat funktioniert, wenn sich der Absender und Empfänger in der Public Lobby befinden.
 - f. WhisperChat funktioniert nicht, wenn sich der Absender oder Empfänger in der GameLobby befinden.
2. Spiel starten
 - a. Ein Spiel eröffnen mit OGAM.
 - b. Einem Spiel beitreten mit JOIN.
 - c. Das Spiel startet nur, wenn genügend Spieler beigetreten sind.
 - d. Nur der Host kann das Spiel starten mit STAR.
3. Spielen (Allgemein)
 - a. Die Reihenfolge der Spieler wird eingehalten.
 - b. Es wird ein zufälliger Beginner ermittelt.
 - c. Ein Spieler kann nur spielen, wenn er am Zug ist.
 - d. Die Spieler erhalten entsprechende Anzahl Karten.
 - e. Spieler erhalten vier Murmeln auf HomeTiles.
 - f. Vom Ende des Spielbretts gelangt man wieder an den Anfang.
 - g. Wenn die Spieler keine Karten mehr haben, erhalten sie neue.
4. Spielzug
 - a. Spieler kann gewünschte Karte aus seiner Hand wählen.
 - b. Wenn eine Karte gespielt wird, verschwindet sie von der Hand des Spielers.
 - c. Spieler kann nur Aktionen durchführen, die mit der gespielten Karte erlaubt sind.
 - d. Ein Spieler kann nur seine Murmeln bewegen (richtige Farbe).
 - e. Wenn ein Spieler auf ein besetztes Feld gelangt, wird die Murmel, die vorher auf dem Feld war, in den Zwinger geschickt.
5. Spiel beenden
 - a. Der Gewinner wird korrekt ermittelt
 - b. Spieler gelangen zurück in die Lobby
 - c. Das Spiel kann vorzeitig mit QUIT oder EXIT beendet werden