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 Vornherein 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

Bei Meilenstein 3 hatten wir ca. 27 Klassen, bei denen zwischen 0% und 5% der Methoden mittels Javadoc dokumentiert waren. Im Durchschnitt hatten 26.82% der Methoden einer Klasse eine Javadoc-Dokumentation. Die Klasse MainGame nahm mit 84.62% mittels Javadoc dokumentierten Methoden einen Spitzenplatz ein. Wir nahmen uns vor mehr Methoden pro Klasse zu dokumentieren, um die Verständlichkeit des Codes zu erhöhen.

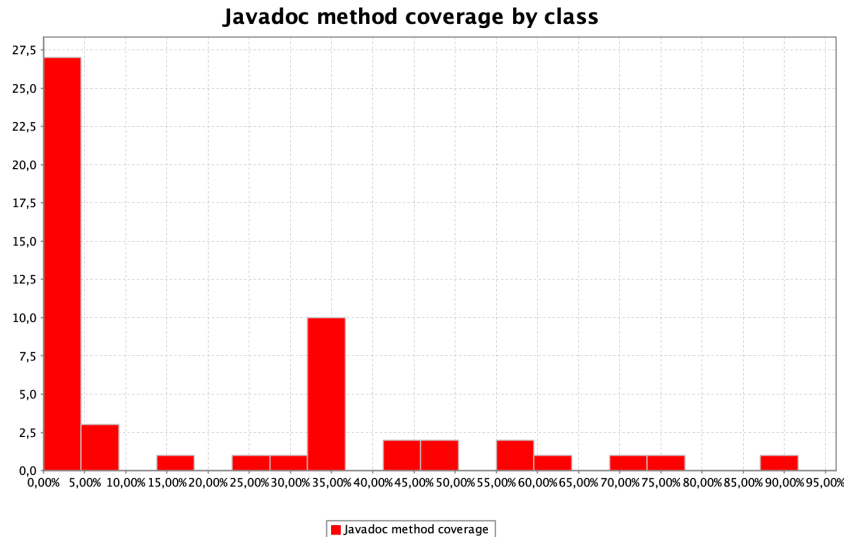

Abbildung 1: Javadoc method coverage by class mit Metrics Reloaded (Stand 18.04.2021).

Dass wir dieses Ziel erreicht haben, zeigt untenstehendes Histogramm von Meilenstein 4. Nun sind es lediglich noch 5 Klassen, bei denen zwischen 0% und 5% der Methoden dokumentiert sind. Bei 30 Klassen sind zwischen 95% und 100% der Methoden dokumentiert.

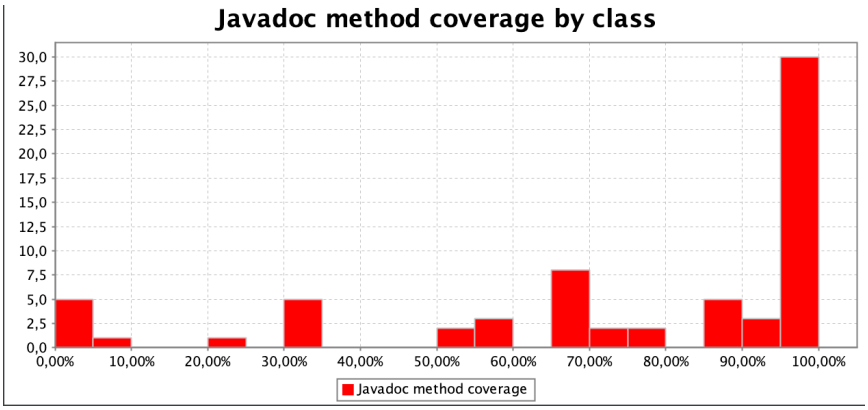

Abbildung 2: Javadoc method coverage by class mit Metrics Reloaded (Stand 09.05.2021).

Lines of code by class

Bei Meilenstein 3 hatten wir 17 Klassen, die zwischen 1 und 25 Zeilen Code beinhalteten. Die meisten Klassen verfügten über 1 und 125 Zeilen Code. Die Klasse ServerGameCommand war mit 629 Zeilen Code am längsten. Im Durchschnitt hatte eine Klasse 87.92 Zeilen Code. Um den Code übersichtlicher zu halten, wollten wir die Klasse ServerGameCommand aufteilen.

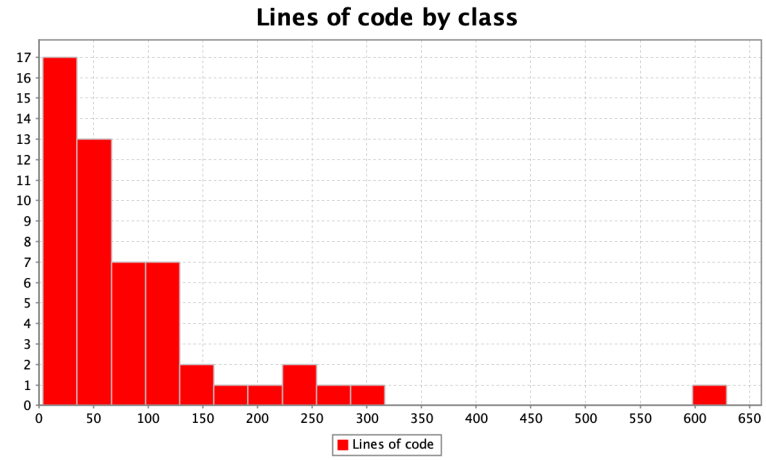

Abbildung 3: Lines of code by class mit Metrics Reloaded (Stand 18.04.2021).

Betrachten wir nun das Histogramm von Meilenstein 4, sehen wir, dass es nun eine Klasse gibt (GameWindowController), die mit 910 Codezeilen die Klasse ServerGameCommand von Meilenstein 3 bei Weitem übertrifft. Am wenigsten Codezeilen hat die Klasse TrackTile mit 8 Zeilen. Die durchschnittliche Anzahl Codezeilen pro Klasse erhöhte sich von 87.92 auf 132.33.

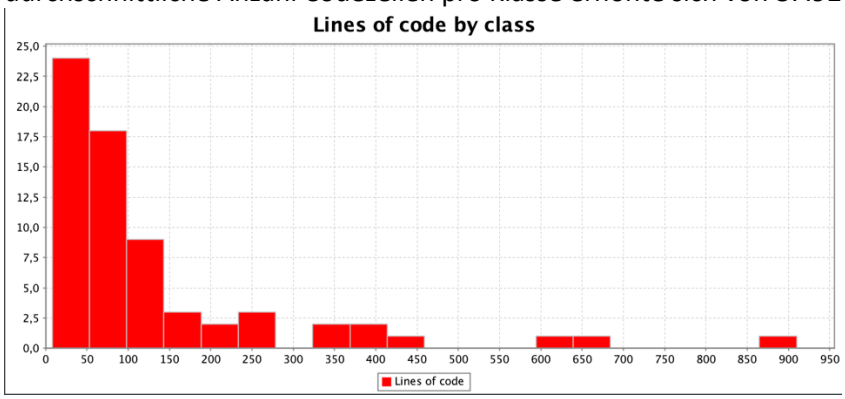

Abbildung 4: Lines of code by class mit Metrics Reloaded (Stand 09.05.2021).

Lines of code by method

Dieses Histogramm zeigt die Anzahl Codezeilen pro Klasse. Darin enthalten sind auch die Kommentare und JavaDocs-Kommentare. Am meisten Codezeilen hatte bei Meilenstein 3 die Methode ServerMenuCommand.execute(String) mit total 152 Zeilen, wovon 9 Zeilen Kommentare waren. Wir wollten diese Methode in kleinere Methoden aufteilen, um den Code verständlicher zu gestalten. Im Durchschnitt verfügte eine Methode über 11.72 Zeilen Code, wovon 9.66 Zeilen keine Kommentare waren.

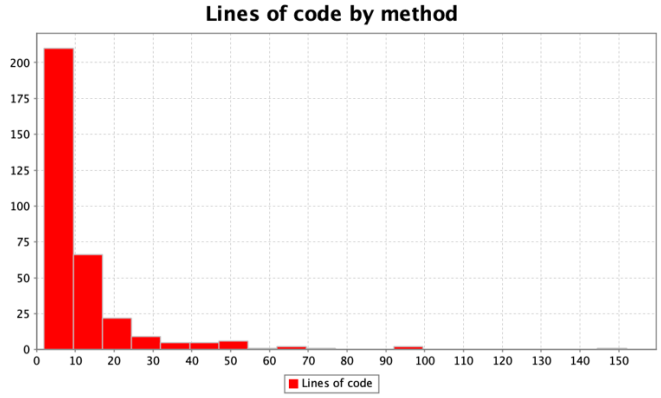

Abbildung 5: Lines of code by method mit Metric Reloaded (Stand 18.04.2021).

Bei dieser Metrik haben wir unser Ziel nicht erreicht. Die Methode ServerGameCommand.execute verfügt nun mit 205 Codezeilen sogar noch über mehr Zeilen Code als bei Meilenstein 3. Davon sind lediglich 12 Zeilen Kommentare. Am wenigsten Codezeilen hat die Methode RulesCheckHelper.RulesCheckHelper(MainGame) mit 3 Zeilen. Die durchschnittliche Anzahl Codezeilen pro Methode hat sich von 11.72 auf 15.42 Zeilen erhöht. Davon sind durchschnittlich 10.95 Zeilen keine Kommentare.

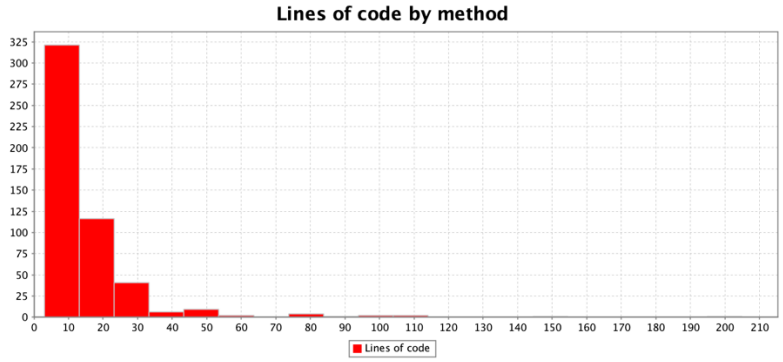

Abbildung 6: Lines of code by method mit Metric Reloaded (Stand 09.05.2021).

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		14%		0%	11	13	27	35	1	3	0	1
ClientGameProtocol		0%		0%	4	4	13	13	2	2	1	1
ClientMenuCommand		57%		28%	10	15	17	39	0	3	0	1
SendFromClient		52%		75%	4	9	25	46	2	5	0	1
Client		79%		n/a	5	11	17	53	5	11	0	1
MessageHandlerClient		66%		41%	6	9	10	27	1	3	0	1
KeyboardInput		42%		16%	4	6	11	18	1	3	0	1
ReceiveFromServer		65%		83%	2	6	12	27	1	3	0	1
ConnectionToServerMonitor		73%		50%	1	3	2	10	0	2	0	1
ClientMenuProtocol		98%		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 7: Codecoverage mit Jacoco (Stand 18.04.2021).

Bei Meilestein 4 ergibt sich folgendes Bild.

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	54,5% (36/66)	37,3% (194/520)	31% (1086/3505)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
jDogs	100% (3/3)	85,7% (30/35)	81,4% (114/140)
jDogs.board	100% (5/5)	88,2% (15/17)	96,8% (61/63)
jDogs.gui	81,2% (13/16)	58,6% (82/140)	59,2% (592/1000)
jDogs.player	100% (2/2)	37,5% (15/40)	38,9% (42/108)
jDogs.serverclient.clientside	100% (10/10)	86,3% (44/51)	74,4% (250/336)
jDogs.serverclient.helpers	100% (3/3)	100% (8/8)	96,4% (27/28)
jDogs.serverclient.serverside	0% (0/27)	0% (0/229)	0% (0/1830)

Abbildung 8: Codecoverage mit Jacoco (Stand 09.05.2021).

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 Milestone4

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 Button Start
    b. Einem Spiel beitreten mit Button JOIN.
    c. Das Spiel startet nur, wenn genügend Spieler beigetreten sind.
    d. Nur der Host kann das Spiel starten mit dem Start-Button.
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.
    f. Jack funktioniert korrekt (Murmeln auf Tracktile tauschen).
    g. Spezialkarte 7 funktioniert korrekt:
        i. Alle von der 7 überholten Murmeln werden nach Hause geschickt.
        ii. 7 kann auf mehrere eigene Murmeln aufgeteilt werden.
        iii. 7 kann auf eigene und Murmeln des Partners aufgeteilt werden.
    h. Nur mit der Karte 4 kann man rückwärts laufen.
    i. Eine Murmel, die erst gerade aus dem Zwinger gekommen ist, blockiert den Durchgang für die anderen Murmeln.
    j. Eine Murmel, die erst gerade aus dem Zwinger gekommen ist, darf nicht direkt ins Ziel laufen.

    k. TeamMode: Wenn alle Murmeln eines Spielers im Ziel sind, darf dieser Spieler mit den Murmeln seines Teampartners fahren.
5. Spiel beenden
    a. SingleMode: Der Gewinner wird korrekt ermittelt
    b. TeamMode: Das Gewinnerteam wird korrekt ermittelt
    c. Spieler gelangen zurück in die Lobby
    d. Das Spiel kann vorzeitig mit beendet werden.
        i. Zurück in die Lobby gehen
        ii. Das Spiel komplett verlassen (Verbindung zum Server trennen)