Universität Basel
Programmierprojekt
FS21

QM-Report

jDOGS

Gruppe 13

Ruben Hutter, Gregor Bachmann, Johanna Meyer

Inhaltsverzeichnis

1. Einleitung 2
2. Softwarequalitätssicherungskonzept 2
   2.1 Konstruktives Qualitätsmanagement 2
   2.2 Analytisches Qualitätsmanagement 2
3. Durchführung 3
4. Resultate 3
   4.1 Javadoc method coverage by class 3
   4.2 Lines of code by class 4
   4.3 Lines of code by method 5
   4.4 Code Coverage 6
5. Diskussion 7
6. Anhang 8
   6.1 Checkliste Code Review 8
   6.2 Checkliste Milestone2 8
   6.3 Checkliste Milestone3 8
   6.4 Checkliste Milestone4 9
   6.5 Checkliste Milestone4 10

1. Einleitung
Eine der Vorgaben beim Programmierprojekt war es, ein Softwarequalitätssicherungskonzept zu erstellen, durchzuführen, zu bestimmten Zeitpunkten Messungen vorzunehmen und die Resultate zu diskutieren. In diesem Bericht dokumentieren und reflektieren wir unsere Qualitätssicherungsmassnahmen.

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

2.1 Konstruktives Qualitätsmanagement
Konstruktives Qualitätsmanagement betrifft die Entwicklung des Produkts. Wir haben folgende Massnahmen festgelegt:

- 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, so detailliert wie möglich 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.
- Wir dokumentieren die Sitzungen im Tagebuch, so dass alle Gruppenmitglieder bei Bedarf nachlesen können, was besprochen wurde und was die zu erledigenden Aufgaben sind.

2.2 Analytisches Qualitätsmanagement
Analytisches Qualitätsmanagement betrifft die Analyse der Software. Wir haben folgende Massnahmen festgelegt:

- Wenn eine Person eine grössere Arbeit erledigt hat, erklärt sie den anderen ihren Code, sodass danach alle mit diesem Code weiterarbeiten können (Structured Walkthrough).
- Zusätzlich führen wir Code Reviews durch, wenn eine Komponente fertig gestellt wurde. Dabei stützen wir uns auf die Checkliste aus der Vorlesung (siehe Anhang). 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 dies jedoch auf WhatsApp getan, da wir dort schneller reagieren und

nachfragen konnten. Der gefundene Fehler sollte so genau wie möglich beschrieben werden (Voraussetzungen, Schritte zur Reproduktion, Fehlermeldungen, etc.).
- Wir benutzen JUnit-Tests als Hilfsmittel, 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.

3. Durchführung

Wir hielten uns im Grossen und Ganzen an unser Qualitätssicherungskonzept und passten das Konzept nach Meilenstein 3 und 4 auf die aktuellen Bedürfnisse an. Beispielsweise führten wir nach Meilenstein 3 ein zweites regelmässiges Treffen ein, da uns ein Treffen pro Woche (zusätzlich zur Übungsstunde) als zu wenig erschien. Wir verwendeten ab Meilenstein 2 den Logger Log4j. Seit Beginn des Projekts achteten wir uns darauf, den Code mittels Javadoc zu dokumentieren, mehr dazu in Kapitel 4.1.
Die Messungen führten wir wie verlangt bei den Meilensteinen 3 und 4 sowie kurz vor dem fünften Meilenstein durch. Die Resultate der Messungen bei Meilenstein 3 und 4 nahmen wir zum Anlass, unseren Code zu verbessern.

4. Resultate

Im Qualitätssicherungskonzept für Meilenstein 3 legten wir die folgenden Metrics fest:

- Javadoc method coverage by class
- Lines of code by class
- Lines of code by method

Im Folgenden werden die Resultate der festgelegten Metrics zu den drei Messpunkten präsentiert und erläutert.

4.1 Javadoc method coverage by class

Diese Metrik gibt an, wie viele Methoden einer Klasse mittels Javadoc dokumentiert sind. Wir entschieden uns für diese Metrik, da ein ausreichend dokumentierter Code die Verständlichkeit desselben erhöht.

Meilenstein	Min	Max	Average
3	0% (bei 15 Klassen)	100% (bei 4 Klassen)	51.44%
4	0% (bei 6 Klassen)	100% (bei 25 Klassen)	85.47%
5	0% (bei 5 Klassen)	100% (bei 24 Klassen)	85.84%

Tabelle 1: Prozentualer Anteil Methoden einer Klasse, die mittels Javadoc dokumentiert sind.

Die Tabelle zeigt, dass wir von Meilenstein zu Meilenstein immer mehr Methoden einer Klasse mit Javadoc dokumentierten. Ein grosser Sprung lässt sich zwischen Meilenstein 3 und 4 feststellen, wo sich die durchschnittliche Anzahl Methoden einer Klasse, die mit Javadoc dokumentiert sind, von 51.44% auf 85.47% steigerte.
Diese Entwicklung lässt sich auch anhand der Plots feststellen. Während bei Meilenstein 3 noch bei der Mehrheit der Klassen zwischen 0% und 10% der Methoden mittels Javadoc dokumentiert waren, hat sich das Bild bei Meilenstein 4 und 5 verschoben.

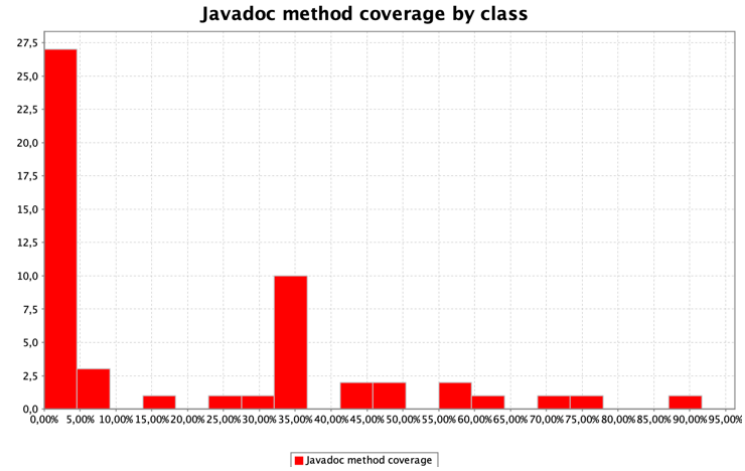

Abbildung 1: Javadoc method coverage by class Milestone 3.

Ab Meilenstein 4 sind deutlich mehr Methoden mit Javadoc dokumentiert, da wir uns speziell darauf geachtet haben, den Code ausreichend zu kommentieren. Die Grafik zeigt, dass unsere Bemühungen erfolgreich waren, denn nun sind durchschnittlich über 85% der Methoden einer Klasse dokumentiert. Dass nicht alle Methoden kommentiert sind, liegt unter anderem daran, dass wir es nicht als sinnvoll erachteten, jede toString-, get- und set-Methode zu kommentieren, da diese Methoden mehrheitlich selbsterklärend sind.

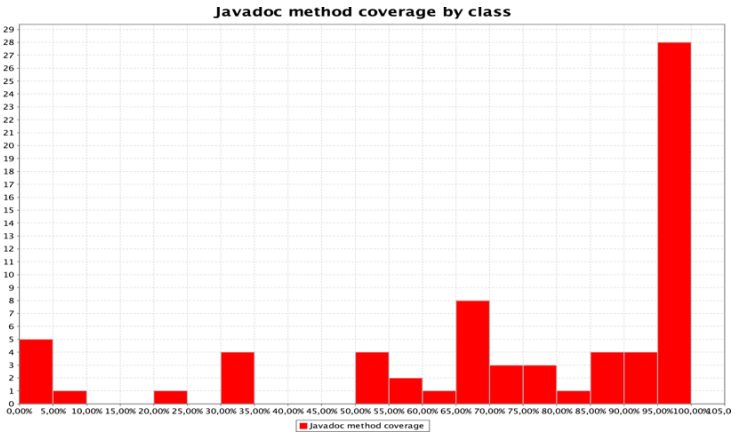

Abbildung 2: Javadoc method coverage by class Milestone 5.

4.2 Lines of code by class

Diese Metrik gibt die Anzahl Codezeilen pro Klasse an, Wir entschieden uns für diese Metrik, da lange Klassen die Verständlichkeit und Übersichtlichkeit des Codes beeinträchtigen.

Meilenstein	Min	Max	Average
3	8 (board.TrackTile)	781 (ServerGameCommand)	99.91
4	8 (board.TrackTile)	916 (RulesCheck)	132.1
5	8 (board.TrackTile)	916 (GameWindowController)	140.81

Tabelle 2: Anzahl Codezeilen pro Klasse.

Wie die Tabelle zeigt, stieg die Anzahl Codezeilen pro Klasse von Meilenstein zu Meilenstein. Dies liegt daran, dass unser Code immer komplexer wurde, da wir beispielsweise die Spielregeln verfeinerten oder bei Meilenstein 4 die GUI hinzugekommen ist.
Wie anhand untenstehender Grafik ersichtlich, bilden die extrem langen Klassen jedoch eine Ausnahme. Es gibt nur eine Klasse, die über 900 Codezeilen aufweist und nur zwei mit mehr als 600 Zeilen Code.

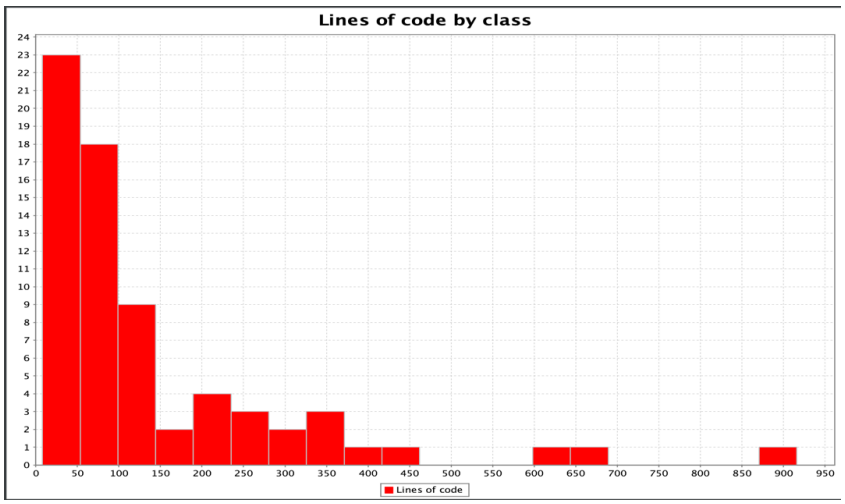

Abbildung 3: Lines of code by class, Milestone 5.

4.3 Lines of code by method

Diese Metrik gibt die Anzahl Codezeilen pro Methode an, Wir entschieden uns für diese Metrik, da lange Methoden die Verständlichkeit und Übersichtlichkeit des Codes beeinträchtigen.

Meilen-stein	Min	Max	Average
3	3 (z.B. Alliance_4.Alliance_4(int))	162 (ServerMenuCommand.execute(String)	13.24
4	3 (z.B. Alliance_4.Alliance_4(int))	205 (ServerMenuCommand.execute(String)	15.35
5	3 (z.B. Alliance_4.Alliance_4(int))	212 (ServerMenuCommand.execute(String)	15.35

Tabelle 3: Anzahl Codezeilen pro Methode.

Im Gegensatz zu LOC by class ändert sich die Methode mit den meisten Codezeilen nicht bei den Messpunkten. Die Methode ServerMenuCommand.execute(String) weist zu allen drei Messzeitpunkten am meisten Codezeilen auf.
Die Verteilung der Klassen bezüglich Anzahl Codezeilen ist jedoch ungefähr vergleichbar mit LOC by class, wie untenstehende Grafik zeigt. Denn auch by LOC by method gibt es nur einzelne Methoden, die äusserst viele Codezeilen aufweisen. Bei LOC by method gibt es beispielsweise nur eine Methode, die über 210 Codezeilen aufweist, die nächste Methode weist lediglich ca. 140 Codezeilen auf.

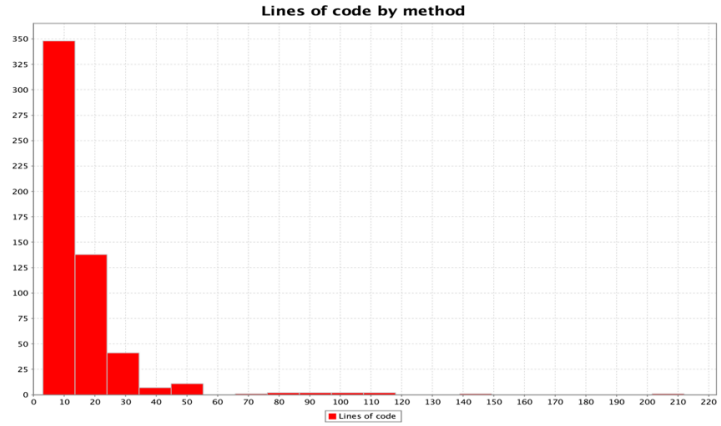

Abbildung 4: LOC bei Meilenstein 5

4.4 Code Coverage

Obwohl wir uns vorgenommen hatten, den TDD-Ansatz zu verwenden, schrieben wir zuerst den Code und erst danach die Unit-Tests. Bis zu Meilenstein 3 hatten wir keine nennenswerten Unit-Tests geschrieben, wie auch auf untenstehender Abbildung erkenntlich.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
jDogs.serverclient.serverside		0%		0%	498	498	1.418	1.418	152	152	16	16
jDogs.gui		0%		0%	139	139	414	414	79	79	12	12
jDogs.serverclient.clientside		0%		0%	82	82	297	297	37	37	10	10
jDogs		0%		0%	47	47	114	114	27	27	4	4
jDogs.player		0%		0%	49	49	91	91	36	36	2	2
jDogs.board		0%		0%	29	29	63	63	17	17	5	5
jDogs.serverclient.helpers		0%		0%	13	13	30	30	8	8	3	3
jDogs.serverClientEnvironment.serverSide		100%		n/a	0	4	0	4	0	4	0	1
Total	10.258 of 10.264	0%	935 of 935	0%	857	861	2.427	2.431	356	360	52	53

Abbildung 5: Code Coverage. Übersicht über die Packages, Meilenstein 3.

Erst bei Meilenstein 4 konnten wir sinnvolle Unit-Tests vorweisen, nachdem wir einiges im Code ändern mussten, um ihn testbar zu machen. Wir schrieben ausschliesslich Unit-Tests zur Spielregelüberprüfung. Diese wird in den Klassen RulesCheck und RulesCheckHelper behandelt. Zu anderen Klassen schrieben wir keine Tests, da es beispielsweise relativ aufwändig ist, die Client-Server-Kommunikation mit Unit-Tests zu testen.
Ein Vergleich der Code Coverage von Meilenstein 3 und Meilenstein 4 zeigt, dass nun einige Tests vorhanden sind, beinahe ausschliesslich im Package serverclient.serverside, wo die Regelüberprüfung lokalisiert ist.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
jDogs.serverclient.serverside		39%		27%	529	716	1.445	2.371	160	273	19	28
jDogs.gui		0%		0%	421	421	1.466	1.466	198	198	17	17
jDogs.serverclient.clientside		0%		0%	122	122	369	369	62	62	10	10
jDogs		9%		0%	58	61	134	143	31	34	2	3
jDogs.player		65%		66%	17	52	42	108	11	40	0	2
jDogs.board		0%		0%	29	29	63	63	17	17	5	5
jDogs.serverclient.helpers		0%		0%	13	13	28	28	8	8	3	3
jDogs.serverClientEnvironment.serverSide		0%		n/a	4	4	4	4	4	4	1	1
Total	15.705 of 19.936	21%	1.203 of 1.447	16%	1.193	1.418	3.551	4.552	491	636	57	69

Abbildung 6: Code Coverage, Übersicht über die Packages, Meilenstein 4.

Die meisten Tests überprüfen Code aus den Klassen RulesCheck und RulesCheckHelper. Bei Meilenstein 4 zeigt die Analyse mit JaCoCo in der Klasse RulesCheck eine Instruction Coverage von 75% und eine BranchCoverage von 49%. Dies zeigt, dass wir noch deutlich mehr Tests schreiben müssten, um alle Fälle zu überprüfen. Ein Blick auf die Cyclomatic Complexity zeigt, dass die Klasse RulesCheck mit dem Wert 137 ziemlich komplex ist und demnach auch eine hohe Anzahl Tests benötigt, um vollständig getestet zu sein. Dies ist wiederum ein Hinweis darauf, dass einige unserer Methoden sehr lang sind und kürzere Methoden nicht nur einfacher zu verstehen, sondern auch einfacher zu testen wäre.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
RulesCheck		75%		49%	88	137	92	381	0	12	0	1
Server		0%		0%	67	67	188	188	28	28	1	1
MainGame		35%		22%	42	65	120	191	13	29	0	1
RulesCheckHelper		74%		56%	24	51	35	135	1	14	0	1

Abbildung 7: Code Coverage, Übersicht über einige Klassen, Meilenstein 4.

5. Diskussion

Wir haben festgestellt, dass es sich lohnt, ein Qualitätssicherungskonzept zu erstellen und uns an die Massnahmen zu halten. Dank der Vorgaben von Seiten der Vorlesung haben wir interessante Tools und Techniken kennen- und anwenden gelernt, die wir sicherlich auch bei weiteren Projekten einsetzen können.

Die manchmal zeitaufwändige Dokumentation mittels Javadoc erwies sich in diesem Gruppenprojekt als sehr nützlich. Denn eine ausführliche Dokumentation erleichtert es enorm den Code anderer Personen nachzuvollziehen. Die Messungen zeigen auch, dass wir unser Ziel, den Code ausreichend zu dokumentieren, erreicht haben. Besonders zwischen Meilenstein 3 und 4 nahm der Anteil dokumentierter Methoden pro Klasse stark zu.

Wir stellten fest, dass Klassen und Methoden schnell sehr lang und komplex werden. Dies zeigen auch die Messungen zu LOC by class, LOC by method und der Cyclomatic complexity. Lange und komplexe Klassen und Methoden sind wohl nicht sehr übersichtlich, jedoch fragten wir uns, ob viele kurze Klassen und Methoden den Code zwingend verständlicher machen, da dann jeweils zwischen den Klassen und Methoden hin- und hergesprungen werden muss. Doch kürzere und weniger komplexe Klassen wären nicht nur übersichtlicher, sondern auch besser testbar mittels Unit-Tests. Deshalb würden wir bei zukünftigen Projekten darauf achten, dass die Klassen und Methoden nicht zu lang und komplex werden.

Bei weiteren Projekten möchten wir von Anfang an darauf achten, dass der Code mit Unit-Tests zu testen ist. Wir hatten uns dies zwar auch bei jDogs vorgenommen, jedoch war uns zu Beginn nicht bewusst, welche Eigenschaften der Code aufweisen muss, damit Unit-Tests geschrieben werden können. Die Vorgabe, Unit-Tests zu integrieren, zwang uns dann dazu, bereits geschriebenen Code umständlich abzuändern. JUnit und JaCoCo lernten wir als nützliche Werkzeuge kennen, um die Qualität einer Softwarekomponente sicherzustellen. Auch die übersichtlichen Grafiken, die sich mit Metrics Reloaded erstellen lassen, um verschiedenste Metriken zu visualisieren, waren eine grosse Unterstützung.

Ein weiteres Tool, das sich als überraschend nützlich erwies, um Fehler aufzuspüren, war der Logger Log4j2. Nachdem wir zu Beginn nicht recht wussten, wozu und wie wir einen Logger einsetzen sollten, wollten wir schon bald nicht mehr darauf verzichten.

Neben den eher technischen Qualitätssicherungsmassnahmen ist jedoch eine funktionierende Kommunikation äusserst wichtig. Insbesondere die regelmässigen Treffen und der Austausch über WhatsApp waren sehr hilfreich, damit alle Gruppenmitglieder wussten, was der Stand ist. Bei weiteren Projekten würden wir von Beginn weg auf eine klarere Aufgabenverteilung mit fixen Deadlines achten.

6. Anhang

6.1 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?

6.2 Checkliste Milestone2

1. Verbindung Client-Server
    a. Richtige Daten (IP-Adresse, Portnummer) eingeben.
    b. Falsche Daten (IP-Adresse, Portnummer) eingeben.
    c. Client will sich mit Server verbinden, aber der Server ist noch nicht gestartet.
    d. Client ist mit Server verbunden, Server wird abgeschaltet.
2. Login
    a. Dem Client wird ein Username vorgeschlagen, basierend auf dem Systemnamen
    b. Client wählt bereits vergebenen Namen.→Neuer Name wird vorgeschlagen.
    c. Client kann seinen Namen ändern.
    d. Client kann sich ausloggen.
3. Chat
    a. Nachricht wird allen aktiven Clients angezeigt.
    b. Es ist ersichtlich, wer die Nachricht geschrieben hat.
    c. Protokollbefehle werden entsprechend behandelt.
    d. Sonderzeichen in Nachrichten werden korrekt angezeigt.
    e. Emojis in Nachrichten werden korrekt angezeigt.

6.3 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

6.4 Checkliste Milestone4

1. Lobby
    d. Spielstatistik kann aufgerufen werden.
    e. Andere Spieler werden angezeigt.
    f. Public Chat funktioniert in der Public Lobby.
    g. Public Chat funktioniert, in und von der GameLobby.
    h. WhisperChat funktioniert, wenn sich der Absender und Empfänger in der Public Lobby befinden.
    i. 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.