

# Konzept zur Qualitätssicherung

Buddler Joe

May 19, 2019

## Inhalt

<b>1</b>	<b>Einleitung</b>	<b>4</b>
1.1	Zielsetzung . . . . .	4
<b>2</b>	<b>Massnahme: Code Standards</b>	<b>5</b>
2.1	Eingliederung der Massnahme . . . . .	5
2.2	Wahl des Standards . . . . .	5
2.3	Ergänzende Standards . . . . .	5
2.3.1	Information Hiding . . . . .	6
2.3.2	APIs und Schnittstellen . . . . .	6
2.3.3	Zuständigkeiten . . . . .	6
2.4	Konkrete Umsetzbarkeit im Projekt . . . . .	6
2.4.1	Ziele und Resultate für Meilenstein 3 . . . . .	6
2.4.2	Ziele und Resultate für Meilenstein 4 und 5 . . . . .	7
2.5	Kontrolle der Massnahmen . . . . .	8
2.5.1	Kontrolle des Code Styles . . . . .	8
2.5.2	Kontrolle der erweiterten Standards . . . . .	8
<b>3</b>	<b>Massnahme: Issue Tracker</b>	<b>9</b>
3.1	Eingliederung der Massnahme . . . . .	9
3.2	Formattierung eines Issues . . . . .	9
3.3	Konkrete Umsetzbarkeit im Projekt . . . . .	9
3.3.1	Ziele und Resultate für Meilenstein 3 . . . . .	9
3.3.2	Ziele und Resultate für Meilenstein 4 und 5 . . . . .	12
3.4	Kontrolle der Massnahmen . . . . .	12

<b>4</b>	<b>Massnahme: Kommunikation</b>	<b>13</b>
4.1	Eingliederung der Massnahme . . . . .	13
4.2	Wahl des Standards . . . . .	13
4.3	Ergänzende Kommunikation . . . . .	13
4.4	Konkrete Umsetzbarkeit im Projekt . . . . .	13
4.4.1	Ziele und Resultate . . . . .	13
4.5	Kontrolle der Massnahmen . . . . .	14
<b>5</b>	<b>Massnahme: GIT</b>	<b>15</b>
5.1	Eingliederung der Massnahme . . . . .	15
5.2	Konkrete Umsetzbarkeit im Projekt . . . . .	15
5.2.1	Ziele und Resultate . . . . .	15
5.3	Kontrolle der Massnahmen . . . . .	15
<b>6</b>	<b>Massnahme: Unit Tests</b>	<b>16</b>
6.1	Eingliederung der Massnahme . . . . .	16
6.2	Konkrete Umsetzbarkeit im Projekt . . . . .	16
6.2.1	Ziele und Resultate . . . . .	16
6.3	Kontrolle der Massnahmen . . . . .	20
<b>7</b>	<b>Massnahme: Javadoc</b>	<b>21</b>
7.1	Eingliederung der Massnahme . . . . .	21
7.2	Konkrete Umsetzbarkeit im Projekt . . . . .	21
7.2.1	Ziele und Resultate für Meilenstein 3 . . . . .	21
7.2.2	Ziele und Resultate für Meilenstein 4 und 5 . . . . .	21
7.3	Kontrolle der Massnahmen . . . . .	21
<b>8</b>	<b>Massnahme: Logging</b>	<b>22</b>
8.1	Eingliederung der Massnahme . . . . .	22
8.2	Wahl des Standards . . . . .	22
8.3	Konkrete Umsetzbarkeit im Projekt . . . . .	22
8.3.1	Ziele und Resultate für Meilenstein 3 . . . . .	22
8.3.2	Ziele und Resultate für Meilenstein 4 und 5 . . . . .	23
8.4	Kontrolle der Massnahmen . . . . .	23
<b>9</b>	<b>Massnahme: Continuous Integration (CI)</b>	<b>24</b>
9.1	Eingliederung der Massnahme . . . . .	24
9.2	Konkrete Umsetzbarkeit im Projekt . . . . .	24

9.2.1	Ziele und Resultate für Meilenstein 5 . . . . .	24
9.3	Kontrolle der Massnahmen . . . . .	25

# 1 Einleitung

Wie in der Vorlesung vom 22.03.2019 vorgezeigt, kann eine fehlerhafte Software schnell zu grossen Schäden und Komplikationen führen. Natürlich wird sich der Schaden bei unserem Projekt in etwas kleinerem Rahmen als die Millionenschäden bewegen, jedoch ist es auch für uns wichtig, eine starke Qualitätssicherung zu haben.

Dafür werden wir uns selber hohe Qualitätsanforderungen stellen, welche wir mit starken Methoden überprüfen und einhalten werden. Dafür haben wir ein umfangreiches Qualitätssicherungskonzept erarbeitet.

In den folgenden Seiten werden wir uns mit unseren Massnahmen zur Qualitätssicherung befassen. Dazu gehören unsere Code Standards, der Issue Tracker, unsere Kommunikationsstrategie, unser Umgang mit GIT, die Unit Tests, das Javadoc, der Logger und abschliessend die CI.

Wir werden uns in diesem Dokument primär mit unseren Umsetzungsstrategien befassen und unseren Plänen, wie wir die Qualität unseres Spielprojekts auf einem hohen Niveau halten können.

## 1.1 Zielsetzung

Wir haben uns zur Zielsetzung genommen, so viel wie möglich aus diesem Softwareprojekt zu lernen. Dies heisst vor allem, dass wir regelmässig alle unsere Strategien überprüfen werden und diese bei Bedarf anpassen werden. Weiterhin werden wir uns achten, so viele effektive Massnahmen wie möglich einzusetzen, damit wir als Gruppe den grössten Lerneffekt haben. Dadurch werden wir mit einem effektiven und geplanten Ablauf den Qualitätsstandard hoch halten.

Zusätzlich werden wir in der Kommunikation auf klare Verantwortlichkeiten setzen. Dadurch ist allen ihre Rolle klar definiert und jeder kann seine Aufgaben pflichtbewusst umsetzen.

Dies alles sollte unserem Ziel eines möglichst fehlerfreien Programmes helfen und auch helfen, reflektierend und überlegt zu arbeiten.

## 2 Massnahme: Code Standards

### 2.1 Eingliederung der Massnahme

Coding Standards sind Teil des konstruktiven Qualitätsmanagements und dort den organisatorischen Massnahmen angehängt. Coding Standards setzen Richtlinien und Standards für den Umgang mit Quellcode und Entwicklungsumgebungen fest. Coding Standards haben viele Vorteile, welche direkt zur Qualität einer Software beitragen. Die Vorteile lassen sich grob in ästhetische und nicht-ästhetische Kategorien unterteilen.

Ästhetische Vorteile ergeben sich durch die vereinfachte und konsistente Lesbarkeit des Quellcodes über das ganze Projekt hinweg. Dies macht überprüfen von nicht selbst geschriebenem Code (oder vor einer Weile geschriebenen Codes) im Projekt einfacher und effizienter. Eine gute Struktur erleichtert ausserdem die Navigation im Quellcode.

Nicht-Ästhetische Vorteile sind etwa, dass sich weniger Fehler im Code einschleichen können, wenn man sich an Standards hält (z.B. immer Klammern setzen, auch wenn das If-Statement nur eine Zeile ist).

### 2.2 Wahl des Standards

Aufgrund grosser Beliebtheit und auf Anraten unseres Tutors werden wir den **Google Java Style Guide** für unser Projekt verwenden:

<https://google.github.io/styleguide/javaguide.html>

Das Dokument ist kurz, leicht verständlich und praxis-erprobt.

### 2.3 Ergänzende Standards

Zusätzlich zu den Standards im Google Java Style Guide machen wir uns Gedanken zu Standards welche für unser konkretes Projekt und unseren Lernerfolg sinnvoll sind. Diese ergänzenden Standards sind bisher an Sitzungen und im Discord besprochen worden, sollen aber in Zukunft zentral niedergeschrieben werden. Eine nicht abschliessende Liste mit Standards und Konzepten, welche wir bisher besprochen haben und an welche wir uns zu halten versuchen.

### 2.3.1 Information Hiding

Wir legen sehr viel Wert auf das Prinzip des Information Hiding. Jede Klassenvariable ist entweder privat oder es liegt ausführliche Dokumentation mit guten Gründen vor, falls die Variable nicht privat ist.

### 2.3.2 APIs und Schnittstellen

Wird ein Package mit mehreren Klassen erstellt, sollte die API oder Schnittstelle dafür - falls benötigt - sich auf so wenig Klassen wie möglich beschränken. Wenn immer möglich sollte die API in einer "Master-Klasse" zusammengefasst sein.

### 2.3.3 Zuständigkeiten

Jede Klasse hat eine eigene Zuständigkeit und Rolle, welche sich nicht mit einer anderen Klasse überschneiden sollte. Die Logik einer Methode sollte immer in der Klasse geschrieben werden, bei welcher die Zuständigkeit am besten gegeben ist. Beispiel: Die Methode `sendToLobby()` aus der abstrakten `Packet` Klasse fällt in den Zuständigkeitsbereich der `ServerLogic`. Folglich wird die Methode dort implementiert und aus der `Packet` Klasse nur aufgerufen.

## 2.4 Konkrete Umsetzbarkeit im Projekt

### 2.4.1 Ziele und Resultate für Meilenstein 3

**Ziele** Alle Teammitglieder arbeiten mit IntelliJ, welches Code Standards erzwingen kann bzw. es kann Verletzungen des Standards als Warnungen anzeigen. Bis zum 3. Meilenstein wollen wir folgende Ziele erreicht haben:

- Alle Mitglieder haben das den Google Java Style Guide gelesen und verstanden
- Alle Mitglieder haben das *Checkstyle* Plugin für IntelliJ installiert
- Sämtlicher Code ist so umgeschrieben, dass er sich an den Code Standard hält
- Neuer Code wird direkt im richtigen Stil geschrieben

- Wir haben die ergänzenden Standards ausformuliert und in einem eigenen Dokument niedergeschrieben
- An mindestens einer Sitzung werden Code Standards besprochen und potentiell erweitert

**Resultate** Die für den dritten Meilenstein gesetzten Ziele sind erreicht. Das *Checkstyle* wurde regelmässig von allen Teammitgliedern eingesetzt. Wie man der Graphik entnehmen kann, bewegt sich die Fehlerzahl fast im Nullbereich. Vor dem Anwenden des *Checkstyle* Plugins befanden sich knapp 9'000 Fehler im Quellcode, welche korrigiert wurden. Abbildung 1 illustriert die Rentabilität des *Checkstyle* Plugins. Die Fehlerzahl sank in kurzer Zeit.

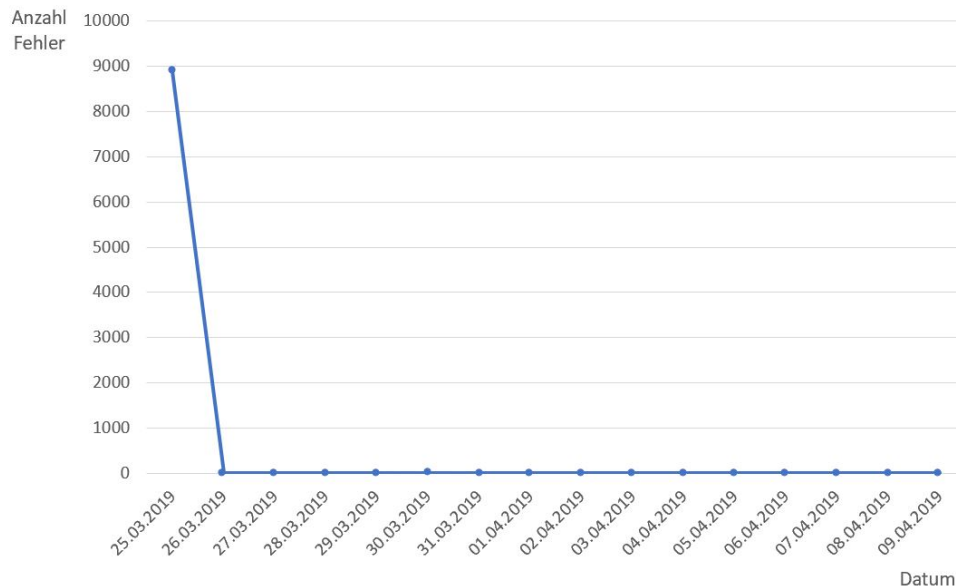


Figure 1: Anzahl Fehler im Quellcode

#### 2.4.2 Ziele und Resultate für Meilenstein 4 und 5

**Ziele** Für den vierten wie auch den fünften Meilenstein möchten wir unseren Quellcode möglichst fehlerfrei beibehalten und natürlich vor jedem Merge eines Branches in den Master einen fehlerfreien Code bereitstellen. Dafür werden wir für den vierten Meilenstein eine Continuous Integration auf Gitlab einbauen, welche bis zum 5. Meilenstein genutzt werden sollte. Auf die

CI wird im 9. Kapitel genauer eingegangen.

**Resultate** Der Checkstyle wurde in den Testzyklus der CI eingebaut. Dadurch wurde uns versichert, dass jeder Code nach dem Auflösen der Mergerequests auch die entsprechende Qualität hat, um in den Master gemerged zu werden.

Das stetige Einsetzen des *Checkstyle* Plugins ermöglichte uns eine schnelle Prüfung unseres Programmierstils. Ist eine Prüfung fehlgeschlagen, wurden die Fehler als Warnungen herausgegeben. Dadurch konnte man die Fehler sehen und somit sich den korrekten Programmierstil schneller einprägen.

## 2.5 Kontrolle der Massnahmen

### 2.5.1 Kontrolle des Code Styles

Die Kontrolle hier ist simpel: Je weniger Warnungen das *Checkstyle* Plugin anzeigt, desto besser ist die Massnahme umgesetzt; mit dem Ziel den kompletten Code ohne undokumentierte Style Warnungen zu haben. Das Ausführen des *Checkstyle* Plugins wird ebenfalls durch die Continuous Integration erzwungen, auf welche im 9. Abschnitt eingegangen wird.

### 2.5.2 Kontrolle der erweiterten Standards

Wir haben keine automatisierten Methoden um eine quantitative Kontrolle durchzuführen. Stattdessen vertrauen wir auf gegenseitige Kontrolle im Team. Wir machen uns gegenseitig auf potentielle Verletzungen unserer Standards aufmerksam und suchen bei Bedarf gemeinsam eine Lösung. Im Discord ist die Diskussion für alle, auch nachträglich, ersichtlich und sie dient gleichzeitig als ein Archiv für ähnliche Situationen in der Zukunft.



## 3 Massnahme: Issue Tracker

### 3.1 Eingliederung der Massnahme

Issue Tracking und Bug Reporting sind Teile des analytischen Qualitätsmanagements und dort den analysierenden Verfahren angehängt. Sie dienen dazu, im Programm entdeckte Fehler für alle Entwickler übersichtlich und standardisiert abzulegen. Der Issue Tracker erlaubt ebenfalls das gruppieren ("taggen") der Reports und ein Zuweisen von Team Mitgliedern an die Fehler.

### 3.2 Formattierung eines Issues

Folgende Informationen sollten in jedem Issue vorhanden sein:

**Branch und Version.** Falls nicht auf dem master, auf welchem Branch und auf welcher Version des Branchs das Problem entdeckt wurde.

**Beschreibung** Wie genau äussert sich das Problem? Welche Teile des Programms sind betroffen?

**Logs** Logs und Konsolenoutput vor, während und nach dem Problem, wenn vorhanden.

**Reproduktion** Welche Schritte sind nötig, um das Problem herbeizuführen und wie zuverlässig ist die Reproduktion?

**Erwartung** Was ist das erwartete Verhalten?

### 3.3 Konkrete Umsetzbarkeit im Projekt

#### 3.3.1 Ziele und Resultate für Meilenstein 3

**Ziele** Wir haben oft relativ klar aufgeteilte Zuständigkeiten, wer welchen Code schreibt. Issues können so in unserem Projekt direkt an die verantwortliche Person zugeteilt werden. Bis zum 3. Meilenstein wollen wir folgende Ziele erreicht haben:

- Sinnvolle Labels für Issues sind erstellt
- Jedes Mitglied hat mindestens einen Issue nach Vorlage erstellt

Open

Opened 1 minute ago by 

WWZ-Nadler Matthias

Close issue

New issue

Beispiel Issue

Branch und Version

Master Branch, Hash: `a1856fc8`

Beschreibung

Lobbynamen können leer sein oder am Anfang/Ende Leerzeichen enthalten. " " wird zum Beispiel als gültiger Lobbyname angenommen.

Logs

Konsolen output, da wir noch keine Logger haben:

```
create      a
Lobby-Creation Successful
-----
Available Lobbies:
Name:      a      , LobbyId: 1, Spieler: 0
-----
```

Reproduktion

Einloggen auf dem Server mit beliebigem Benutzernamen. Lobby erstellen mit "create a ". Sollte 100% reproduzierbar sein.

Erwartung

Lobby name sollte vor der validierung getrimmt werden (Leerzeichen am Anfang und am Ende entfernen).

Edited right now by WWZ-Nadler Matthias

👍

0

👎

0

😊

Show all activity

Create merge request

Figure 2: Beispiel eines vollständigen Issue Reports

- Der Issue Tracker ist mit unserem Discord verbunden

**Resultate** Für den dritten Meilenstein hat jeder schon mindestens ein Issue nach Vorlage erstellt und auch mit sinnvollen Labels vermerkt.

**Die Spielrunde wird beim Ausscheiden des letzten Spielers nicht beendet.**  
 #6 · opened 15 minutes ago by Popovic Sanja - xudyla72 To Do

**Username remain displayed on the sceen after disconnection from e player**  
 #5 · opened 3 days ago by Würth Moritz - legody29 To Do

**Tried to load texture lobbyOverviewWood\_norm.png , didn't work**  
 #4 · opened 5 days ago by WWZ-Nadler Matthias 📅 Apr 8, 2019 To Do

**Back-Button beendet Programm**  
 #3 · opened 5 days ago by Schlachter Sebastian - cyvazu20 📅 Apr 7, 2019 To Do

**Issue mit dem Main programm und nullpointer Konflikt**  
 #2 · opened 6 days ago by Gsteiger Viktor - gelody40 📅 Apr 3, 2019 To Do

**Beispiel Issue**  
 #1 · opened 2 weeks ago by WWZ-Nadler Matthias Non Critical Bug

Figure 3: open issues

### 3.3.2 Ziele und Resultate für Meilenstein 4 und 5

**Ziele** Für den vierten und fünften Meilenstein möchten wir den Issue Tracker aktiver benutzen.

**Resultate** Der Issuetracker wurde weiterhin eingesetzt. Die Bugs wurden durch den Report übersichtlich dargestellt (Abb. 4). Die Reports stellen eine effizientere Arbeit aller Teammitglieder sicher, indem jeder genau weiss, woher er sich die nötigen Informationen bezüglich der Reproduktion etc. beschaffen soll. Struktur und Ordnung wurde gewährleistet.

## Highscore File wird auch bei frühzeitig abgebrochenen Spielen aktualisiert

---

**Branch und Version**

master [22e7e53d](#)

---

**Beschreibung**

Eigentlich sollten nur Spieler in den Highscore kommen, welche 3000 Gold erreicht haben. Im Moment kommen aber alle, die gewinnen in den Highscore. Dadurch ist der Highscore nicht aktuell.


---

**Logs**

keine

---

**Reproduktion**

Ein Spiel vorzeitig gewinnen durch sterben, dadurch kommt man hoch in den Highscore 

---

**Erwartung**

Korrektur, dass nur Spieler, welche das Spiel gewinnen in den Highscore kommen.

Figure 4: Issuereport

## 3.4 Kontrolle der Massnahmen

Wird ein Issue nicht ordnungsgemäss formatiert, kann dies direkt dem Mitglied mitgeteilt werden. Jedem Issue kann eine Person und ein Datum zugeteilt werden, was die Chance, dass ein Issue vergessen geht, sehr stark reduziert.

## 4 Massnahme: Kommunikation

### 4.1 Eingliederung der Massnahme

Die Kommunikation ist in solch einem Projekt ein wichtiger Teil des konstruktiven Qualitätsmanagements. Da die Gruppe aus fünf Personen besteht, ist die Kommunikation untereinander ein wichtiger Bestandteil. Durch eine gründliche Kommunikation kann viel Zeit eingespart werden. Durch fehlende Absprache mit anderen Gruppenmitgliedern können Fehler generiert werden, welche viel Zeit kosten, um sie wieder zu beheben.

### 4.2 Wahl des Standards

Unsere Standardkommunikationsmittel sind Discord und Whatsapp.

### 4.3 Ergänzende Kommunikation

Natürlich sollte ein Projekt nicht nur über Textnachrichten und Sprachchat verlaufen. Deshalb sind gemeinsame Treffen sehr wichtig, um einen Überblick über das Projekt zu behalten. So können weitere Schritte koordiniert geplant werden. Kommende Aufgaben werden besprochen und auf die Gruppenmitglieder verteilt.

### 4.4 Konkrete Umsetzbarkeit im Projekt

#### 4.4.1 Ziele und Resultate

**Ziele** In unserem Projekt legen wir viel Wert darauf, eine gute und stabile Kommunikation zu führen. Wir möchten uns mindestens ein Mal pro Woche als Gruppe treffen, um den Stand der Dinge zu prüfen. Dabei sollen Protokolle geschrieben werden, die die besprochenen Themen dokumentieren. Auf Discord soll mit Hilfe von Kanälen Ordnung geschaffen und Diskussionen, Besprechungen oder Fragen im entsprechenden Kanal ausgeführt werden.

**Resultate** Indem wir per Discord in den entsprechenden Kanälen diskutierten, wurde in unserem Projekt viel Struktur und Ordnung geschaffen. Es wurde viel Zeit eingespart beziehungsweise effizienter gearbeitet, da man genau Bescheid wusste, wo man nach bestimmten Unterhaltungen zu suchen hat. Gerade bei einer Zusammenarbeit von mehreren Personen ist die Übersicht ein wichtiger Faktor.

Neben den Protokollen wurde seit dem dritten Meilenstein nach jedem Treffen im "to-dos"-Kanal eine detaillierte Liste hochgeladen, wer welche Aufgaben übernimmt und welche beim letzten Treffen verteilten Aufgaben erledigt wurden. Indem wir während dem Programmieren die Discordapp, welche auch als App für den PC zur Verfügung steht, stets offen hielten, resultierte ein Protokoll in Nachrichtenförm als eine schnellere und effizientere Zugriffsmöglichkeit als eine PDF-Datei in einem separaten Ordner. Ab dem zweiten Meilenstein konnten wir mit Hilfe der Voice Chat Funktion in Discord vor jedem Meilenstein zu fünf telefonieren, dabei unser Programm testen und wichtige Punkte, welche uns beim Testen aufgefallen sind, besprechen.

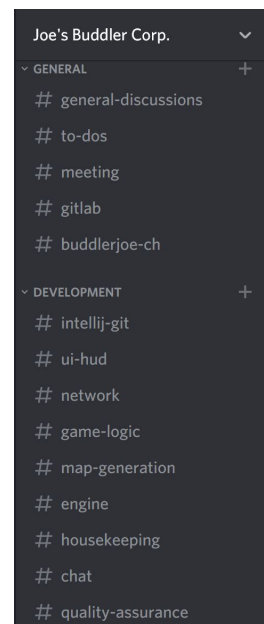


Figure 5: Kanäle in Discord

## 4.5 Kontrolle der Massnahmen

Eine Kontrolle fällt wesentlich einfach aus. Wir halten uns an unsere Vereinbarung uns jede Woche zu treffen und den Kontakt auf Discord weiterhin aufrecht zu halten. Die Protokolle sind datiert in einem Ordner zu finden.

## 5 Massnahme: GIT

### 5.1 Eingliederung der Massnahme

GIT ermöglicht das gleichzeitige, koordinierte Arbeiten aller Teammitglieder an der Software. Durch Branching kann man effizienter arbeiten und ist bis zu einem gewissen Punkt unabhängig von den anderen Teammitgliedern, weil jeder auf seinem eigenen Branch arbeiten kann. Trotzdem bleibt die Interaktion erhalten, denn es wird allen Mitgliedern ermöglicht, Einblick in die Arbeit seines Teamkollegen zu erhalten. Erachtet man seinen Branch als einsatzbereit, so kann man ein Mergerequest erstellen und den Quellcode gegenlesen lassen. Ordnung und Struktur werden gewährleistet, indem man den Quellcode im Master nicht bearbeiten darf.

### 5.2 Konkrete Umsetzbarkeit im Projekt

#### 5.2.1 Ziele und Resultate

**Ziele** Gitkraken soll bei jedem Teammitglied installiert sein. Jedes Teammitglied soll bei Bedarf an Modifizierung des Codes im Master einen eigenen Branch erstellen und dort die Änderungen vornehmen. Der Code im Master darf nie direkt bearbeitet werden. Jeder Commit soll mit einer ausschlaggebenden Nachricht versehen werden.

**Resultate** Jedes Teammitglied benutzte seit Anfang des Projekts die graphische Oberfläche GitKraken und arbeitete mit Branches. Die schöne Visualisierung der Branches vereinfachte das Arbeiten und man konnte sofort sehen, wer an welchem Branch arbeitet und welche Branches existieren. Alle Teammitglieder versahen jeden Commit mit einer ausschlaggebenden Nachricht. Das Kommentieren der letzten Modifizierungen verschaffte Übersicht und man musste nicht den ganzen Code lesen, um sich in Erinnerung zu rufen, was man im Code zuletzt verändert hatte.

### 5.3 Kontrolle der Massnahmen

Die Visualisierung von GitKraken stellt uns eine detaillierte Dokumentation über die Fortschritte des Projekts wie auch über den Einsatz jedes Teammitglieds zur Verfügung. Diese sind für alle Projektteilnehmer einsehbar.

## 6 Massnahme: Unit Tests

### 6.1 Eingliederung der Massnahme

Unit Tests sind Teil des analytischen Qualitätsmanagements. Durch Unit Tests können einzelne Klassen und Module auf Fehler getestet werden. Durch solche Tests versucht man Fehler von Grund auf zu finden, indem man eine Klasse als kleinste Einheit testet. Auf diese Art und Weise kann man prophylaktisch bei Interaktion aller Klassen grössere Fehler in der Software beheben und somit auch ein ewiges Suchen des Entstehungsorts des Fehlers im Quellcode bis zu einem gewissen Punkt vermeiden.

### 6.2 Konkrete Umsetzbarkeit im Projekt

#### 6.2.1 Ziele und Resultate

##### Ziele

- Das Team soll mit JUnit und einer weiteren Bibliothek arbeiten.
- Bis zum fünften Meilenstein sollen 40% der Klassen mit Tests abgedeckt sein.
- Wichtigste Komponenten sollen getestet werden.
- Die Tests sollen grün sein.

**Resultate** Für den vierten Meilenstein haben wir nebst JUnit auch mit Mockito Unit Tests geschrieben. Wir haben eine Klassen Coverage von über 40% (Abb. 6) mit 218 Unit Tests geschafft.

Package	Class, %	Method, %	Line, %
all classes	43.7% (86/ 197)	27.1% (432/ 1593)	21.5% (1958/ 9089)

Figure 6: Coverage durch Unit Tests

Wir empfanden das net Package als eine core component, welche am tiefgründigsten getestet werden musste. Auf der unteren Graphik werden die Abhängigkeiten visualisiert. Gerade bei solch einem verstrikten System möchten wir, dass die einzelnen Komponenten auch wirklich funktionieren.



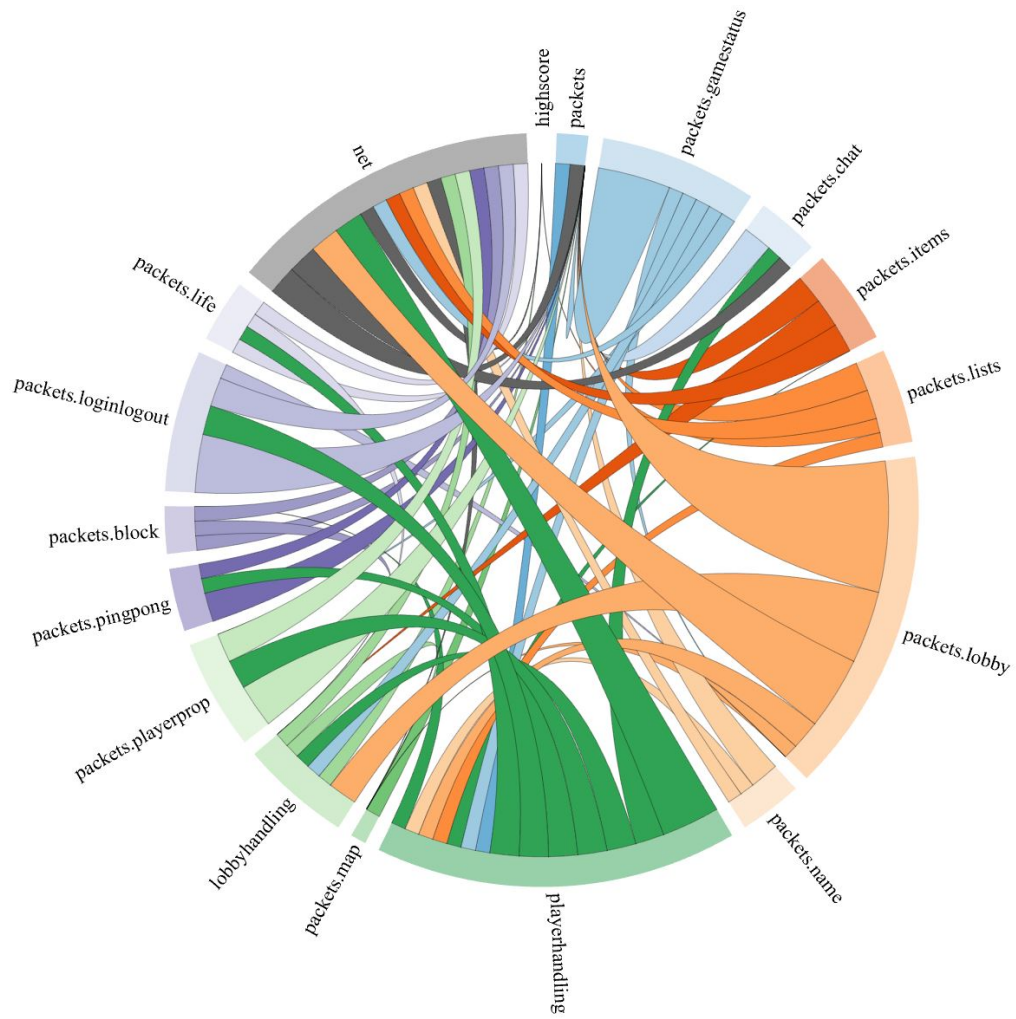


Figure 7: Abhängigkeiten im net Package

In der unteren Abbildung befindet sich ein Unit Test für die PacketLobbyOverview-Klasse, welche sich im net.packets.lobby Package befindet. Analog wie bei den anderen Klassen des net Packages wird getestet, ob die Konstruktoren die korrekten Parameter erhalten. Alle net.packets Klassen enthalten die validate()-Methode, die die Parameter testet und eine fehlerhafte Verarbeitung stoppt, falls diese nicht stimmen. Der Konstruktor der PacketLobbyOverview-Klasse erwartet beispielsweise einen String als Parameter. Durch korrekte Validierung sollte eine Error Message erfolgen, falls die Einzelkomponenten des Strings nicht den zu erwartenden String bilden. Die drei Tests vergewissern uns, dass mit falscher Stringzusammensetzung korrekt umgegangen wird.

```
@Test
public void checkDataNotAscii() {
    PacketLobbyOverview p = new PacketLobbyOverview( data: "0");
    Assert.assertEquals( expected: "ERRORS: Invalid characters, only extended ASCII.", p
.createErrorMessage());
}

@Test
public void checkNoOpenLobbiesAvailable() {
    PacketLobbyOverview p = new PacketLobbyOverview( data: "Test||No open Lobbies");
    Assert.assertEquals( expected: "ERRORS: ", p.createErrorMessage());
}

@Test
public void checkDataFormatError() {
    PacketLobbyOverview p = new PacketLobbyOverview( data: "Test||No open Lobbies");
    Assert.assertEquals( expected: "ERRORS: ", p.createErrorMessage());
}
```

Figure 8: Ausschnitt des Unit Tests für PacketLobbyOverview-Klasse

Mithilfe der Unit Tests sind wir auf viele Stellen im Code gestossen, welche noch modifiziert werden mussten. In den net.chat Klassen wurde beispielsweise nie abgefragt, ob der Stringparameter des Konstruktors null ist. Dabei sind wir auf NullPointerExceptions gestossen. Diese konnten mithilfe einer Abfrage behoben werden. Die gelbmarkierten Stellen wurden hinzugefügt.

```

public PacketChatMessageToClient(String chatmsg) {
    super(PacketTypes.CHAT_MESSAGE_TO_CLIENT);
    try {
        this.chatmsg = chatmsg.trim();
    } catch (NullPointerException e) {
        addError("No Message found.");
    }
    setData(chatmsg);
    validate();
}

```

Figure 9: Veränderung im Konstruktor von PacketChatMessageToClient

Weiter sind wir auf nicht vollständig prüfende oder ressourcenverschwendende validate()-Methoden gestossen. Es wurde in manchen Fällen nicht getestet, ob der String nur aus ASCII Zeichen besteht.

Die validate()-Methode in der unteren Abbildung testet, ob der String nur aus Zahlen besteht. Es wurde kein frühzeitiger Abbruch beim ersten Auftauchen eines non-Digits eingebaut, welches die Validierung unnötigerweise verlängerte.

```

public void validate() {
    if (getData() == null) {
        addError("Empty message");
    } else {
        char[] temp = getData().toCharArray();
        for (int i = 0; i < getData().length(); i++) {
            if (48 > temp[i] && temp[i] > 57) {
                addError("Not a digit.");
                return;
            }
        }
    }
}

```

Figure 10: Veränderung in validate()-Methode der PacketPong-Klasse

Durch die Unit Tests konnte unnötiger Code entfernt und Lücken im Code gefüllt werden. Durch sie konnte viel Code rekapituliert werden, denn bei einem grossen Projekt wie diesem, wo die Übersicht schnell verloren gehen kann, helfen die Tests die Qualität des Codes zu steigern.

### **6.3 Kontrolle der Massnahmen**

Die geschriebenen Klassen sind wie der Quellcode im main Ordner unter Test zu finden. Spätestens durch die CI werden die Unit Tests auf den betroffenen Branch angewandt. Schlägt ein Test fehl, so wird dies in der Pipeline angezeigt. Der Code muss die Tests bestehen.

## 7 Massnahme: Javadoc

### 7.1 Eingliederung der Massnahme

Javadoc ist ein Teil des konstruktiven Qualitätsmanagements. Jedem Teammitglied wird es durch das Dokumentieren des Quellcodes ermöglicht, sich schnell in den Quellcode einzulesen und mit wenig Zeitaufwand die Funktion der jeweiligen Klasse oder der jeweiligen Methode zu erfahren. Auf diese Art und Weise wird auch Ordnung im Quellcode geschaffen.

### 7.2 Konkrete Umsetzbarkeit im Projekt

#### 7.2.1 Ziele und Resultate für Meilenstein 3

**Ziele** Der Autor der jeweiligen Methode oder der jeweiligen Klasse ist für deren Dokumentation zuständig, da dieser selbst die Funktion am besten kennt.

**Resultate** Bisher wurde der Stand des Javadocs nicht sofort von allen Teammitgliedern aktualisiert. Die Aktualisierung erfolgte erst nach ein paar Tagen.

#### 7.2.2 Ziele und Resultate für Meilenstein 4 und 5

**Ziele** Bis zum vierten Meilenstein möchten wir Disziplin schaffen und spätestens kurz vor jedem Merge in den Master ein aktuelles Javadoc haben. Deshalb möchten wir das Javadoc in den CI-Testzyklus integrieren.

**Resultate** Durch die CI (Kapitel 9) wurde das Erfüllen dieser Massnahme auch erzwungen. Jede Methode, welche ein Javadoc braucht, ist nun mit einer Beschreibung versehen.

### 7.3 Kontrolle der Massnahmen

Das Javadoc ist jederzeit in einem Ordner für alle Gruppenmitglieder einsehbar und vollständig, da die Tests nicht fehlschlagen.

## 8 Massnahme: Logging

### 8.1 Eingliederung der Massnahme

Logging ist eine Massnahme zur erleichterten Fehlerbehebung. Durch verschiedene Log Levels ermöglicht ein effektives Logging Tool eine schnelle Fehlersuche.

Durch eine Echtzeitverfolgung lassen sich ausserdem bei einem aktiven Review des Codes schnell Fehler entdecken. Zusätzlich gibt es Tools zum Filtern, Suchen, Hervorheben und viele mehr, welche die Fehlersuche noch einfacher machen.

### 8.2 Wahl des Standards

Aufgrund grosser Beliebtheit und auf Hinweis in der Vorlesung vom 22.3.2019 werden wir das Tool **slf4j** für unser Projekt verwenden:

<https://www.slf4j.org/>

Das Tool ist einfach einsetzbar, Open Source, modular und einfach bedienbar.

### 8.3 Konkrete Umsetzbarkeit im Projekt

#### 8.3.1 Ziele und Resultate für Meilenstein 3

**Ziele** Für regelmässige Code Reviews soll das Tool eingesetzt werden, um isolierte Teile des Codes zu überprüfen. Dafür sollte vor den jeweiligen Reviews das Logging eingerichtet werden.

**Resultate** Der Logger (slf4j) ist in unserem Projekt eingebaut und wird zum Debuggen eingesetzt. In einem Ordner namens log sind die Textdateien zu finden, welche dort für 30 Tage lokal gespeichert bleiben.

```

22:50:06.073 [Thread-0] DEBUG net.playerhandling.PingManager - Number of unanswered pings: 1
22:50:07.075 [Thread-0] DEBUG net.playerhandling.PingManager - Number of unanswered pings: 2
22:50:08.076 [Thread-0] DEBUG net.playerhandling.PingManager - Number of unanswered pings: 3
22:50:09.076 [Thread-0] DEBUG net.playerhandling.PingManager - Number of unanswered pings: 4
22:50:10.078 [Thread-0] DEBUG net.playerhandling.PingManager - Number of unanswered pings: 5
22:50:11.079 [Thread-0] DEBUG net.playerhandling.PingManager - Number of unanswered pings: 6
22:50:12.079 [Thread-0] DEBUG net.playerhandling.PingManager - Number of unanswered pings: 7
22:50:13.081 [Thread-0] DEBUG net.playerhandling.PingManager - Number of unanswered pings: 8
22:50:13.081 [Thread-0] DEBUG net.ServerLogic - Removing client no. 1

```

Figure 11: Ausschnitt eines Logfiles

### 8.3.2 Ziele und Resultate für Meilenstein 4 und 5

**Ziele** Bis zum 5. Meilenstein wollen wir folgende Ziele erreicht haben:

- Alle zentralen Code Stücke sind mit einem Logging versehen, welches die Fehlersuche vereinfacht.
- In regelmässigen code Reviews wird das Logging Tool angewendet.

**Resultate** Der Logger wurde in den zentralen Code Stücken eingebaut. Durch den Logger wurde das Debuggen deutlich vereinfacht und Fehler konnten somit schneller gefunden werden.

## 8.4 Kontrolle der Massnahmen

Der Logger ist im Quellcode und die Logfiles in einem Ordner einsehbar.

## 9 Massnahme: Continuous Integration (CI)

### 9.1 Eingliederung der Massnahme

Die Continuous Integration ist ein Teil des konstruktiven Qualitätsmanagements. Die CI wird bei jedem Merge-Request auf den betroffenen Branch angewandt. Die Anwendung führt mehrere Tests durch, die sogenannte Pipeline, und stellt sicher, dass der Code im Branch den Anforderungen entspricht. Der zu mergende Code muss drei Stages bestehen: Build, Unit Tests, Javadoc und Checkstyle. Würde der Branch bei einem Test durchfallen, so wird dies visualisiert und der Code könnte nicht in den Master gemergt werden. Mit Hilfe der CI kann man stets einen fehlerfreien Code im Master gewährleisten.

### 9.2 Konkrete Umsetzbarkeit im Projekt

#### 9.2.1 Ziele und Resultate für Meilenstein 5

**Ziele** Vor dem vierten Meilenstein haben wir es uns als Ziel gesetzt, die CI in Gitlab einzurichten und sie dementsprechend einzusetzen. Alle Teammitglieder sollen diszipliniert vorgehen: Jeder soll eine bestandene Pipeline anstreben, wenn man einen Branch in den Master mergen möchte.

**Resultate** Die CI wurde vor dem vierten Meilenstein erfolgreich eingeführt und eingesetzt. Schon wenige Tage nach der Einführung wurden ca. 115 Pipelines ausgeführt. Durch die CI wurden somit drei sehr wichtige Massnahmen sichergestellt und deren Kontrolle vereinfacht.













































	#8345 by 	FurtherUnitT...  59d574c0 Checkstyle   	⌚ 00:05:12 📅 19 hours ago
	#8344 by 	master  a228496d fixed the button to abo...   	⌚ 00:04:58 📅 19 hours ago
	#8343 by 	FurtherUnitT...  8516da18 Minor changes to mac...   	⌚ 00:04:59 📅 19 hours ago
	#8341 by 	FurtherUnitT...  5cf25477 Checkstyle   	⌚ 00:05:06 📅 20 hours ago
	#8340 by 	master  3b9c9f3f Merge branch 'adjustFr...   	⌚ 00:05:15 📅 20 hours ago
	#8339 by 	WinnerDeterm...  98e97de2 Javadoc and Checkstyle   	⌚ 00:05:10 📅 20 hours ago
	#8338 by 	adjustFreeze...  b942982b Adjusted the Freeze ov...   	⌚ 00:05:07 📅 20 hours ago

Figure 12: Ausschnitt der Pipelinesammlung

### 9.3 Kontrolle der Massnahmen

Die CI wird automatisch bei einem Mergerequest ausgeführt. Auf Gitlab sind die Pipelines aller Mergerequests einsehbar. Man kann die Pipelines den entsprechenden Personen zuteilen, die Stages und die Uhrzeit sehen.

## List of Figures

1	Anzahl Fehler im Quellcode . . . . .	7
2	Beispiel eines vollständigen Issue Reports . . . . .	10
3	open issues . . . . .	11
4	Issuereport . . . . .	12
5	Kanäle in Discord . . . . .	14
6	Coverage durch Unit Tests . . . . .	16
7	Abhängigkeiten im net Package . . . . .	17
8	Ausschnitt des Unit Tests für PacketLobbyOverview-Klasse . .	18
9	Veränderung im Konstruktor von PacketChatMessageToClient	19
10	Veränderung in validate()-Methode der PacketPong-Klasse . .	19
11	Ausschnitt eines Logfiles . . . . .	23
12	Ausschnitt der Pipelinesammlung . . . . .	25