

Fernfachhochschule Schweiz

—
Informatik

—
Bachelor of Science in Informatik

Projektarbeit im Modul Fortgeschrittene Technologien in der objektorientierten Programmierung (FTOOP):

Multiplayer-Quiz



Autoren

Chris Wüthrich
Robert Zelder

Dozent Christian Heitzmann

Gruppe

GEP9

Abgabe

20. Juni 2021

Inhaltsverzeichnis

1	Aufgabenstellung und Planung	1
1.1	Anforderungen.....	1
1.1.1	Muss-Anforderungen	2
1.1.2	Feature-Liste / Kann-Anforderungen	3
1.2	Planung	3
1.3	Vorgehensmodell.....	4
1.3.1	Story	5
1.3.2	Definition of Done	6
1.3.3	Coding Conventions.....	6
1.4	Objektorientierte Analyse.....	7
1.4.1	Substantiv-Verb-Analyse.....	7
1.4.2	CRC-Karten	8
1.4.3	UML-Diagramme.....	9
2	Umsetzung im Team	13
3	Funktionalität und Aufbau des Spiels	14
3.1	Start des Spiels	14
3.2	Erste Schritte.....	15
3.3	Lobby	16
3.3.1	/chat.....	16
3.3.2	/quit.....	16
3.3.3	/who	16
3.3.4	/stats	17
3.3.5	/play	17
3.4	Im Spiel	17
3.5	Handhabung von Client-Server Timeouts	19
4	Funktionsweise der Klassen	20
4.1	Architektur	20
4.2	Kommunikation.....	21
4.3	Serverattribute	21
4.4	Dateiimport	22
4.5	Gameloop.....	22
4.6	Parsing	23
4.7	Ausgabe	24
4.8	Login	24

4.9	GameMatching	25
4.10	Quizsession.....	25
4.11	Ergebnis berechnen	26
5	Kurzvorstellung der Unit Tests	27
5.1	Vorgehensweise der Testerstellung.....	27
5.2	Probleme durch Klassenabhängigkeiten.....	27
5.3	Erkenntnisse und Verbesserungspotential.....	28
6	Soll-Ist-Vergleich.....	29
7	Feedback	29
	Literaturverzeichnis.....	29

Abbildungsverzeichnis

Abbildung 1: Kanban-Board in Trello	4
Abbildung 2: Backlog Eintrag aus Trello.....	5
Abbildung 3: Backlog Eintrag mit Fortschrittsanzeige ²	6
Abbildung 4: Subjekt-Verb Analyse angewendet an Spieleanforderungen	7
Abbildung 5: Subjekt-Verb Analyse übertragen in Liste ³	7
Abbildung 6: Fotografie der CRC-Karten.....	8
Abbildung 7: Use-Case Diagramm.....	9
Abbildung 8: Aktivitätsdiagramm	10
Abbildung 9: UML Klassendiagramm frühes Stadium	11
Abbildung 10: UML Klassendiagramm späteres Stadium.....	12
Abbildung 11: Auszug aus GitLab Branches	13
Abbildung 12: Zustand nach Serverstart	14
Abbildung 13: Clientzustand nach fehlgeschlagener Serververbindung	14
Abbildung 14: Ausgegebenes Banner nach erfolgreicher Server-Client Verbindung	14
Abbildung 15: Willkommens-Nachricht des Servers	15
Abbildung 16: Login error nach Eingabe eines ungültigen Benutzernamens.....	15
Abbildung 17: Server-Ausgabe nach erfolgreicher Anmeldung	15
Abbildung 18: Server-Ausgabe der möglichen Befehle in der Lobby.....	15
Abbildung 19: Beispiel einer Chat-Nachricht	16
Abbildung 20: Server-Ausgabe nach Abmeldung	16
Abbildung 21: Server-Ausgabe nach Eingabe des Befehls "/who"	16
Abbildung 22: Ausgabe der Spieler-Statistik	17
Abbildung 23: Server-Ausgabe während der Gegnersuche.....	17
Abbildung 24: Server-Ausgabe nach erfolgreicher Gegnersuche.....	17
Abbildung 25: Ausgabe der Quizfrage	18
Abbildung 26: Eigene Ausgabe nach korrekter Antwort	18
Abbildung 27: Ausgabe nach korrekter Antwort des Gegners	18
Abbildung 28: Eigene Ausgabe nach falscher Antwort.....	18
Abbildung 29: Server-Ausgabe nach beenden des Spiels.....	19
Abbildung 30: Server-Ausgabe nach Timeout.....	19
Abbildung 31: Abschliessendes Klassendiagramm des Multiplayer-Quiz	20
Abbildung 32: Codeausschnitt des ServerWorker Konstruktors	27
Abbildung 33: Codeausschnitt der Testklasse ServerWorkerTest.....	28

1 Aufgabenstellung und Planung

Es soll ein Multiplayer-Quiz mit zeilenbasierter Ein- und Ausgabe in Java programmiert werden. Die Vorgaben der FFHS zur Umsetzung lassen viel Spielraum für eigene Anforderungen und Lösungswege offen. Die Grundanforderungen lauten jedoch wie folgt:

- «Es handelt sich um eine reine Kommandozeilen-Applikation (Command Line Interface, CLI) ohne grafische Benutzeroberfläche (Graphical User Interface, GUI). Ein- und Ausgaben erfolgen in der Regel direkt im entsprechenden IDE-Fenster.
- Sie dürfen den gesamten Unicode-Zeichensatz verwenden.
- Sowohl auf den Clients als auch auf dem Server erscheinen saubere Ausgaben auf der Kommandozeile.
- Die Kommunikation zwischen den Clients und dem Server erfolgt direkt über Sockets.
- Der Server liest die Textdatei mit den Quizfragen ein. Eine Quizfrage besteht aus einer Frage mit jeweils vier Antwortmöglichkeiten A bis D (wie bei «Wer wird Millionär?»). Diese Textdatei wird Ihnen von Ihrem Dozenten gestellt.
- Die Spieler/Clients bekommen vom Server jeweils gleichzeitig eine Frage mit den vier Antwortmöglichkeiten zugesandt.
- Der Spieler/Client mit der schnellsten richtigen Antwort bekommt einen Punkt.
- Jeder Spieler/Client wird über den aktuellen Spielstand informiert.
- Sie dürfen kein Java Remote Method Invocation (RMI) und keine Drittbibliotheken verwenden, mit Ausnahme von JUnit5.7.0.
- Sicherheitsaspekte und die Behandlung von Verbindungsunterbrüchen können Sie ausser Acht lassen.»

(s. FFHS, 2021)

1.1 Anforderungen

Zu Beginn analysieren wir die Aufgabenstellung und geben sie auf hoher abstraktionsebene mit eigenen Worten wieder. Dabei definieren wir, was das zu entwickelnde Programm unter Einhaltung der Aufgabenstellung leisten soll. Die von der FFHS eröffneten Freiheiten des Spieldesigns müssen entschieden und beschrieben werden. Dabei teilen wir die Anforderungen in zwei Bereiche auf. Als erstes beschreiben wir die Muss-Anforderungen, welche von der Applikation zwingend implementiert werden müssen, damit das Spiel in unseren

Augen als vollständig angesehen werden kann. Diese heben sich bereits von den Grundanforderungen ab und sollen ein solides Fundament für mögliche Erweiterungen bieten. Im Anschluss folgt eine Feature-Liste, die als «nice-to-have» anzuschauen ist. Die in der zweiten Liste definierten Weiterentwicklungen werden von uns nicht als notwendige Implementierungen angesehen, dienen aber bereits dem Codedesign. So soll verhindert werden, dass wir uns mögliche Weiterentwicklungsmöglichkeiten verbauen.

1.1.1 Muss-Anforderungen

2-Spieler-Variante, Best of 9

- Es werden 9 Fragen gestellt, der Spieler mit den meisten Punkten gewinnt.
- Spieler starten die Applikation und geben ihren Namen ein.
- Spieler werden vom Server begrüsst und landen in einer Lobby.
- Die Begrüssung enthält alle Befehle, die der Spieler eingeben kann (z.B. /help, 1, 2, 3, /quit, /play /stats).
- Gibt ein Spieler /play ein, wird «Suche gleich starken Gegner» angezeigt.
 - Im Abstand von ein paar Sekunden werden lustige Sprüche bzgl. der Spielersuche angezeigt.
 - Sobald ein zweiter Spieler die Spielersuche gestartet hat, startet ein Spiel.
 - Es startet ein Countdown von 5 Sekunden nachdem das Spiel beginnt.
- Die Spieler bekommen vom Server jeweils gleichzeitig eine Frage mit den zugehörigen Antwortmöglichkeiten zugesandt.
- Die Spieler haben 30 Sekunden Zeit eine Antwort einzugeben. Falls keiner eine Antwort eingegeben hat, wird die richtige Antwort einblendet, was keine Punkte gibt.
- Wenn beide falsch antworten, gibt es ebenfalls keinen Punkt.
- Antworten beide Spieler falsch wird der Timer gestoppt und es wird die nächste Frage ausgegeben.
- Wird eine richtige Antwort gegeben, wird der Sieger ausgegeben und es erscheint die nächste Frage.
- Nur der Spieler mit der schnellsten richtigen Antwort bekommt einen Punkt.
- Ein Unentschieden ist möglich, es gibt keine Stichfragen.
- Allen Spielern wird ein Zwischenstand ausgegeben und der Spieler, welcher zu langsam war oder falsch geantwortet hatte, erfährt die richtige Antwort.
- Nach einer Wartezeit von 2-3 Sekunden wird die nächste Frage gestellt.

- Nachdem das Spiel vorbei ist, landen beide Spieler wieder in der Lobby.
- Nach dem Spiel bekommen die Spieler angezeigt, wie viele Punkte sie im momentanen Spiel erreicht haben und wie sie abgeschnitten haben.
- Es gibt keine Begrenzung von gleichzeitigen Spielern, es müssen also mehrere Spiele parallel stattfinden können.
- Ein Befehl, um anzuzeigen wer alles online ist (/who).

1.1.2 Feature-Liste / Kann-Anforderungen

Anbei stehen Features, welche von uns als zusätzlich und „nice-to-have“ eingestuft wurden. Unser Ziel ist jedoch eine Implementierung nach Abgabe der Projektarbeit, um wichtige Themen der späteren Präsenzveranstaltungen des FTOOP-Moduls zu vertiefen.

- Ein Quizrunde mit mehr als zwei Spielern.
- Eine Quizrunde gegen einen bestimmten Spieler.
- Eine Quizrunde, in der Spieler in Gruppen gegeneinander antreten können.
- Spieler können miteinander chatten.
- Ein Matchmaking aufgrund der Spielerstatistik.
- Ein Schwierigkeitsgrad, bei dem die möglichen Antworten nicht gleichzeitig angezeigt werden.
- Ein Single-Player-Modus auf Zeit.
- Auswahl von Themengebieten (mit aktuellen Daten wäre nur das Jahr der Fragestellung möglich).
- Eine Benutzerverwaltung mit Historie.
- Eine Anzeige von Werbung nach jeder Quiz-Runde.
- Premiumfeature zum Ausschalten der Werbung.

1.2 Planung

Bevor mit der Programmierung begonnen wurde, erfolgte eine Planung des Projekts. Die Planung beinhaltet die Auswahl des Vorgehensmodells (agile Methodik mit Scrum-Elementen) sowie eine Durchführung der objektorientierten Analyse. Die verwendeten Technologien und Tools waren entweder durch die FFHS vorgegeben oder wurden gemeinschaftlich zum Zeitpunkt der Notwendigkeit entschieden.

1.3 Vorgehensmodell

Als Vorgehensmodell wurde ein agiler Ansatz gewählt, um die Projektarbeit in kleinere Teile zerlegen und iterativ entwickeln zu können. Um unser Kanban Board darzustellen, verwenden wir die Software Trello von Atlassian (Abb.1). Sie beinhaltet auch gleichzeitig das Backlog für das Projekt. In unseren zwei-wöchentlichen Sprint-Meetings entschieden wir, welche Aufgaben wir in dem Sprint umsetzen wollten und übernahmen diese in die Spalte «To Do». Stories an denen wir arbeiten sind in der Spalte «Dev» wobei nur zwei Stories gleichzeitig in «Dev» sein dürfen. Dies soll verhindern, dass nicht zu viele Dinge angefangen und nicht abgeschlossen werden. Ist die Story abgeschlossen, wird sie in die Spalte «Done» verschoben.

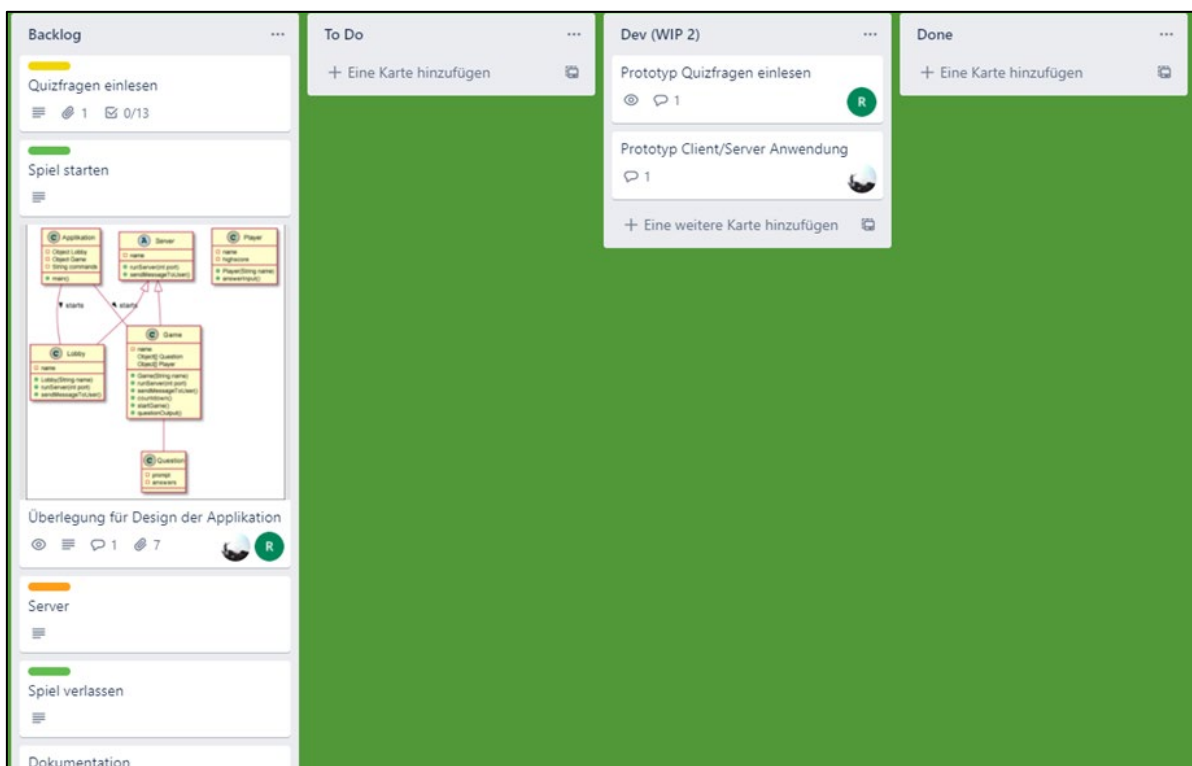


Abbildung 1: Kanban-Board in Trello¹

¹ Quelle: Eigene Darstellung aus «www.trello.com»

1.3.1 Story

Um einen besseren Überblick über die Stories zu erhalten, zeigt Abbildung 2 ein Beispiel einer Story. Die Stories werden im Aufwand geschätzt, wobei S, M und L verwendet werden, um es möglichst einfach zu halten. Dann folgt eine Beschreibung, was mit der Story umgesetzt werden soll. Das kann entweder auf Nutzersicht erfolgen oder wie in diesem Fall etwas technischer ablaufen.

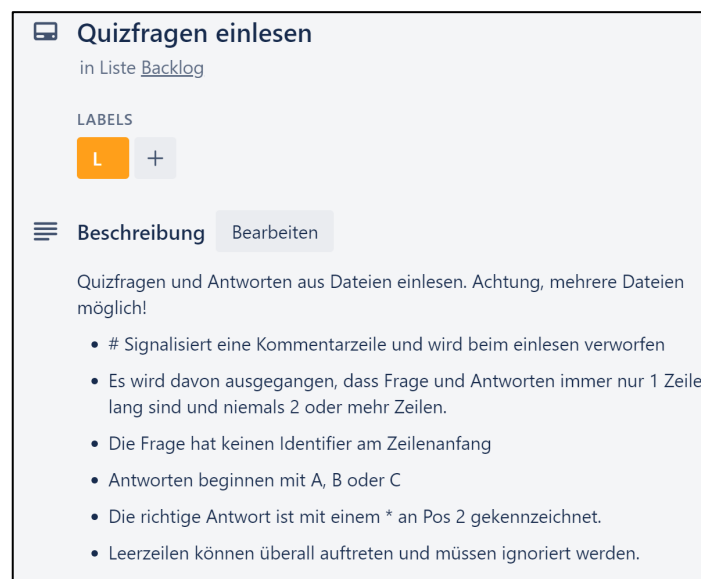


Abbildung 2: Backlog Eintrag aus Trello²

Ausserdem kann eine Story auch detaillierte Informationen enthalten, zum Beispiel welche verschiedenen Zustände es bei Fragen mit Antworten geben kann, falls wir diese im Sprint-review bereits besprochen haben (Abb.3).

² Quelle: Eigene Darstellung aus «www.trello.com»

☒ Fehler in der Datei erkennen und darauf reagieren Löschen

0%

- ☐ Frage ohne Antwort
- ☐ Antwort ohne Frage
- ☐ Mehrere Antworten einer Frage mit * als korrekt definiert
- ☐ Mehr als 3 Antworten, zum Beispiel D oder zwei Mal A
- ☐ Weniger als 3 Antworten
- ☐ Programmierung dynamisch genug gestalten, um eine beliebige Anzahl an Antwortmöglichkeiten zu ermöglichen

Element hinzufügen

Abbildung 3: Backlog Eintrag mit Fortschrittsanzeige²

1.3.2 Definition of Done

Um ein einheitliches und eindeutiges Verständnis davon zu bekommen, was das «Done» einer Story in unserem Projekt bedeutet, legten wir eine «Definition of Done» fest.

- Die Funktion macht, was in der Story beschrieben wurde.
- Es gibt keine Meldungen in SonarLint (Warning-Free).
- Es sind aussagenkräftige Unit-tests vorhanden.
- Mindestens alle öffentlichen Teile sind mit javadoc dokumentiert.
- Code Review mit Partner wurde durchgeführt.

1.3.3 Coding Conventions

Um einen möglichst einheitlichen Quellcode zu erhalten, legen wir unserer Entwicklung Coding Conventions zu Grunde, an die wir uns bestmöglich bei der Implementierung halten.

- Objektorientierte Prinzipien berücksichtigen (z.B. Solid).
- Die Ausgabe erfolgt nicht in der Businesslogik, so dass das Frontend beliebig getauscht werden kann.
- Keine technischen Schulden anhäufen.
- Google Java Style Guides wird versucht zu berücksichtigen.

1.4 Objektorientierte Analyse

In der objektorientierten Analyse versuchten wir, von unseren definierten Anforderungen über eine Substantiv-Verb-Analyse und mithilfe von UML-Diagrammen, eine Grobplanung der Implementierung vorzunehmen.

1.4.1 Substantiv-Verb-Analyse

Bei dieser Analyseform werden alle Substantive und Verben der Anforderungen in jeweils unterschiedlichen Farben markiert (Abb.4). Anschliessen werden die Wörter gruppiert und im ersten Schritt geprüft, ob es sich um Synonyme handelt. Als zweiter Schritt erfolgt eine Sortierung, ob es sich bei den Substantiven um mögliche Objekte oder Objekteigenschaften handelt. Bei den Verben wird geprüft, ob es sich um Aktionen der Objekte handelt (Abb.5).

Verben und **Substantive**

Spezifikation zum **Quiz**, Anonyme 2-Player-Variante, Best of 9

- Es werden 9 **Fragen** gestellt, der **Spieler** mit den meisten richtigen **Antworten** **gewinnt**
- Spieler** **startet** die **Applikation** und gibt seinen **Namen** ein
- Spieler** werden vom **Server** **begrüsst**

Abbildung 4: Subjekt-Verb Analyse angewendet an Spieleanforderungen³

Substantive		Verben
Quiz	Abschnitt im Spiel	stellt Fragen an Spieler
Fragen		
Antworten		
Spieler		gibt Name ein
		gewinnt, verliert, spielt unentschieden
		Bekommen Fragen angezeigt
		kann Befehle/Antworten eingeben
		bekommt Punkte
Gegner	Synonym für Spieler	
Name	Attribut von Spieler	

Abbildung 5: Subjekt-Verb Analyse übertragen in Liste³

³ Quelle: Eigene Darstellung aus OneNote

Um dabei ein gutes Ergebnis zu erhalten und den Lerneffekt zu steigern, haben wir die Subjekt-Verb-Analyse getrennt durchgeführt und anschliessend unsere Resultate verglichen.

1.4.2 CRC-Karten

Anschliessend wurden sogenannte CRC-Karten «Class-Responsibility-Collaboration-Karten» erstellt (Abb.6). Dabei stellten wir allerdings fest, dass uns dieses Hilfsmittel nicht in dem gewünschten Masse weiterbrachte. Die gewonnenen Erkenntnisse waren zu gering und letztendlich nur noch mal eine andere Darstellung der bereits in der Subjekt-Verb-Analyse ermittelten Informationen.

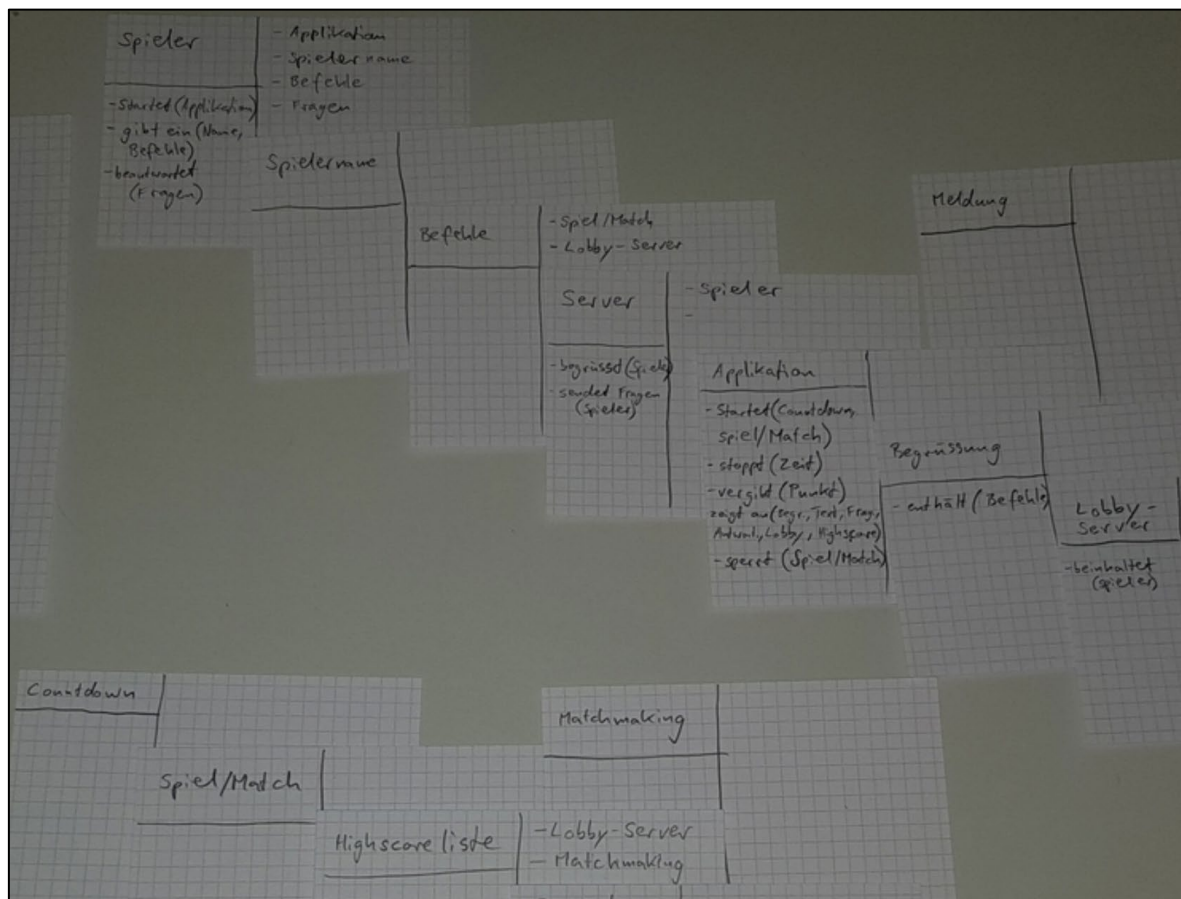


Abbildung 6: Fotografie der CRC-Karten⁴

⁴ Quelle: Eigene Fotografie

1.4.3 UML-Diagramme

Um weitere Informationen über das zu erstellende Programm zu ermitteln, entschlossen wir uns UML-Diagramme zu erstellen.

UML - Use-Case-Diagramm

Als erstes erschufen wir Use-Case-Diagramme, welche alle Anwendungsfälle unserer Software abdecken sollen (Abb.7).

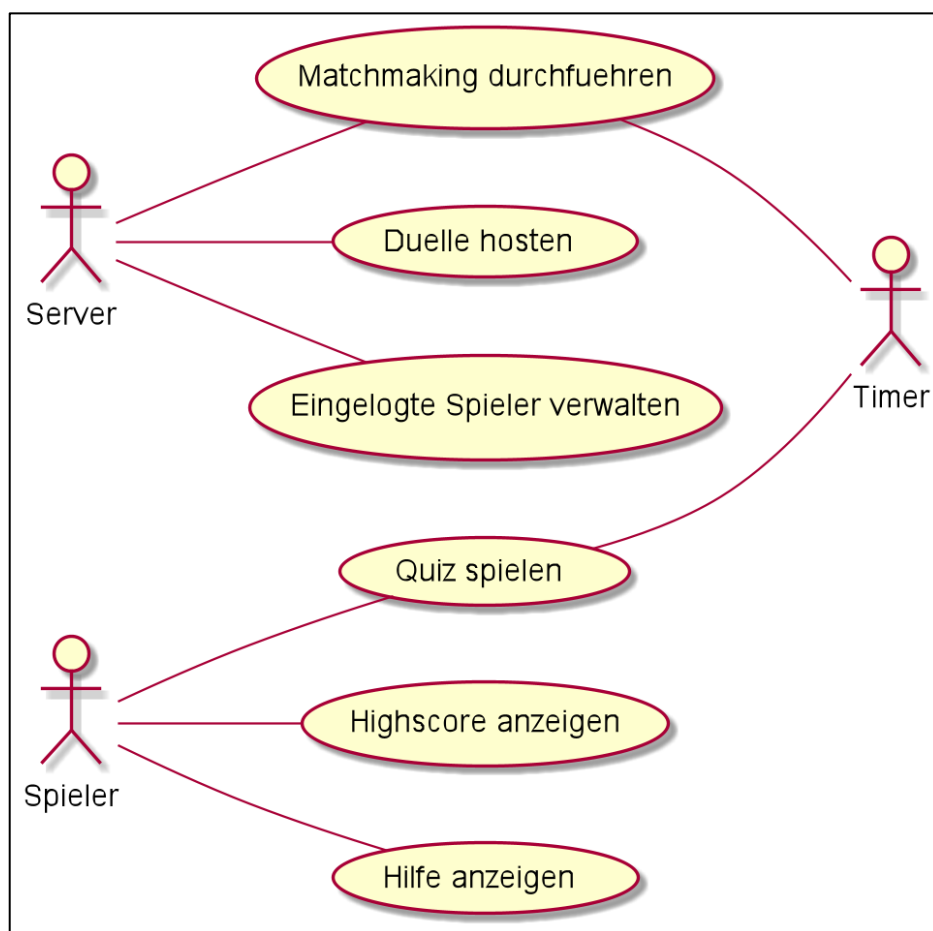


Abbildung 7: Use-Case Diagramm⁵

⁵ Quelle: Eigenes UML Diagramm, erzeugt via PlantUML Plugin in IntelliJ

UML - Aktivitätsdiagramm

Der daraus resultierende Spielablauf haben wir anschliessend in einem Aktivitätsdiagramm dokumentiert. Dieser soll die möglichen Navigationsmöglichkeiten des Spiels aufzeigen (Abb.8).

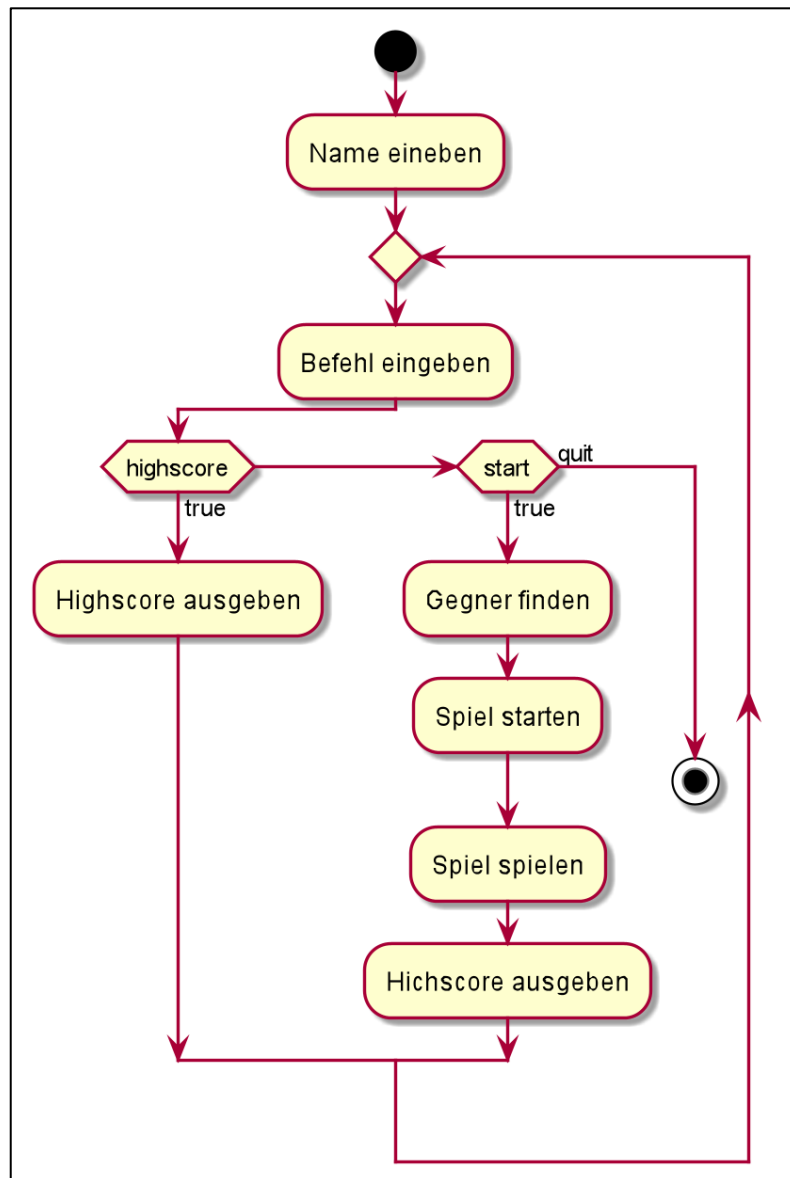


Abbildung 8: Aktivitätsdiagramm⁶

⁶ Quelle: Eigenes UML Diagramm, erzeugt via PlantUML Plugin in IntelliJ

UML - Klassendiagramm

Zum Schluss entwickelten wir ein erstes UML-Klassendiagramm (Abb.9). Während wir im Laufe des Projekts immer wieder kleinere Prototypen ausprobiert haben, um Teilfunktionalitäten zu prüfen, hat sich auch unser Klassendiagramm immer wieder verändert. Stellvertretend hier zwei Versionen aus der Entwicklungshistorie.

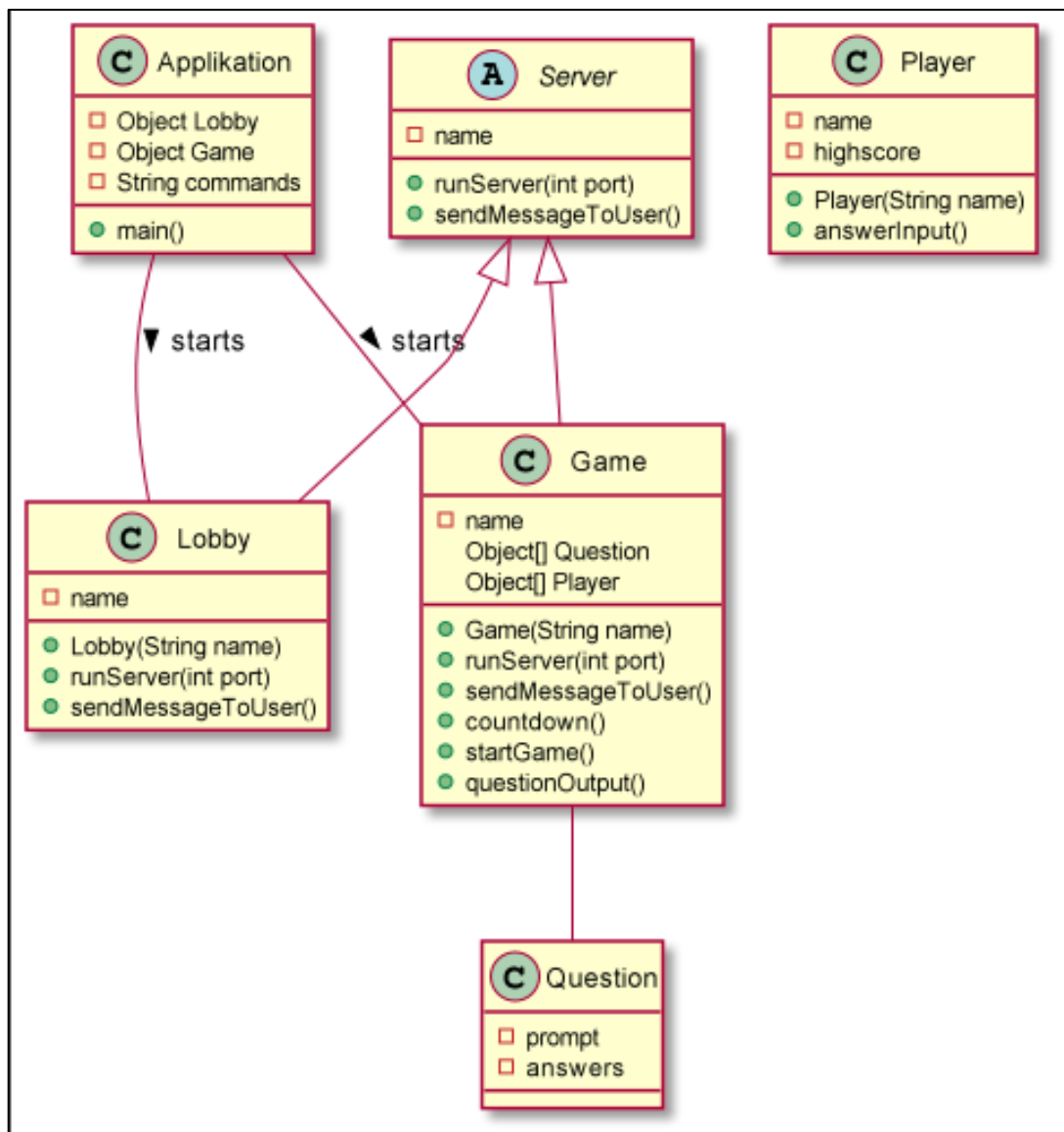


Abbildung 9: UML Klassendiagramm frühes Stadium⁷

⁷ Quelle: Eigenes UML Diagramm, erzeugt via PlantUML Plugin in IntelliJ

Wenn man die anfänglichen Diagramme mit der tatsächlich erstellten Software (Abb.10) vergleicht, ist eine steile Lernkurve erkennbar. Einiges konnte wie geplant umgesetzt werden, anderes wurde gänzlich anders implementiert bzw. ist späteren Refactorings zum Opfer gefallen. Trotzdem hat die Planung unsere Umsetzung gut unterstützt

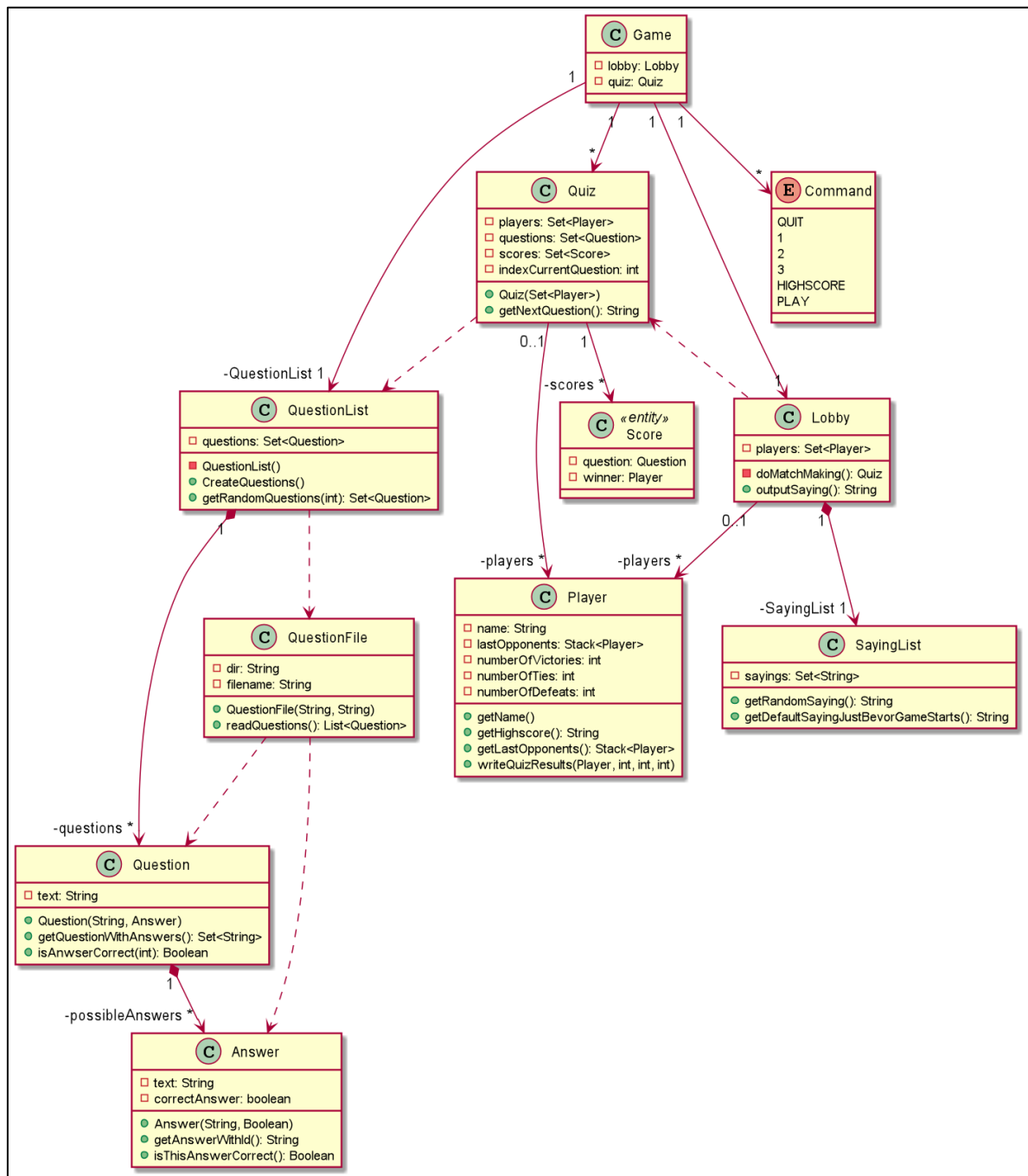


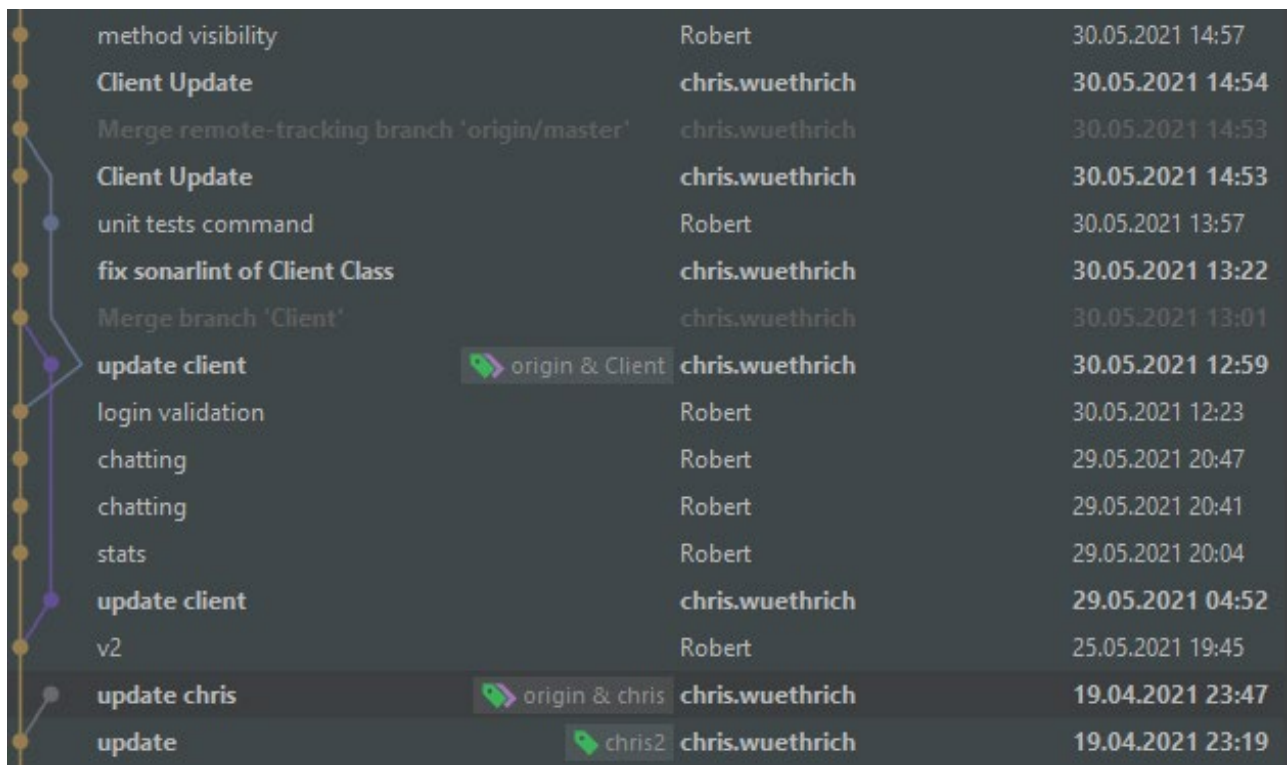
Abbildung 10: UML Klassendiagramm späteres Stadium⁸.

⁸ Quelle: Eigenes UML Diagramm, erzeugt via PlantUML Plugin in IntelliJ

2 Umsetzung im Team

Bereits vor der ersten Zeile Code, hatten wir konkrete Vorstellungen unseres Programms. Dies hat uns zu Beginn etwas Zeit gekostet, jedoch sind wir sicher, dass sich dieser Aufwand gelohnt hat. Durch den regelmässigen Austausch und der Zuhilfenahme diverser Entwicklertools wurde während dem Projekt so manche Fehlerquelle umgangen. Um der Praxis noch etwas näher zu kommen, entschieden wir uns auch aktiv mit GitLab zu arbeiten und mit den dadurch erhaltenen Möglichkeiten zu spielen. In diesem Projektumfang wäre dies bestimmt nicht zwingend notwendig gewesen, jedoch bot es sich an, das umfangreiche Tool noch besser kennenzulernen.

Angegangene Probleme oder Weiterentwicklungen konnten dadurch in separaten Branches entwickelt werden. Dadurch erhielten wir eine saubere Fortschrittsübersicht während des gesamten Projekts (Abb.11).



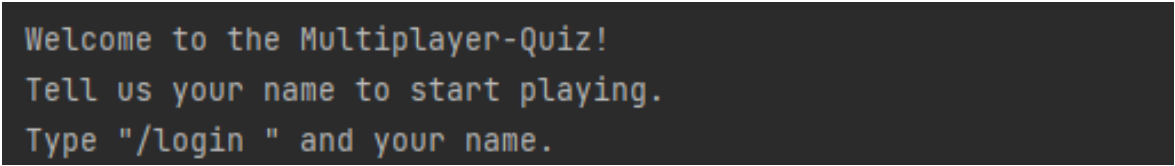
method visibility	Robert	30.05.2021 14:57
Client Update	chris.wuethrich	30.05.2021 14:54
Merge remote-tracking branch 'origin/master'	chris.wuethrich	30.05.2021 14:53
Client Update	chris.wuethrich	30.05.2021 14:53
unit tests command	Robert	30.05.2021 13:57
fix sonarlint of Client Class	chris.wuethrich	30.05.2021 13:22
Merge branch 'Client'	chris.wuethrich	30.05.2021 13:01
update client	chris.wuethrich	30.05.2021 12:59
login validation	Robert	30.05.2021 12:23
chatting	Robert	29.05.2021 20:47
chatting	Robert	29.05.2021 20:41
stats	Robert	29.05.2021 20:04
update client	chris.wuethrich	29.05.2021 04:52
v2	Robert	25.05.2021 19:45
update chris	chris.wuethrich	19.04.2021 23:47
update	chris.wuethrich	19.04.2021 23:19

Abbildung 11: Auszug aus GitLab Branches⁹

Dabei wurden teils Ideen übernommen, verworfen oder gemeinsam an einem Sprint-Review weiterentwickelt.

⁹ Quelle: Eigene Darstellung aus IntelliJ

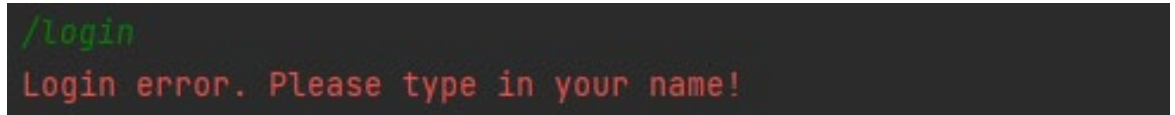
3.2 Erste Schritte



```
Welcome to the Multiplayer-Quiz!  
Tell us your name to start playing.  
Type "/login " and your name.
```

Abbildung 15: Willkommens-Nachricht des Servers¹²

Zusätzlich zum begrüßenden Banner des Servers wird man aufgefordert, sich initial mit einem Benutzernamen anzumelden. Dazu kann direkt in der Konsole der Befehl «/login » gefolgt von einem Namen eingetragen werden. Der Benutzername kann frei gewählt werden, muss jedoch einigen Anforderungen entsprechen. Beispielsweise darf der Name nicht weggelassen werden, da die Anmeldung sonst abgelehnt wird.




```
/login  
Login error. Please type in your name!
```

Abbildung 16: Login error nach Eingabe eines ungültigen Benutzernamens¹²

Ausserdem werden noch weitere Mindestanforderungen an den Benutzernamen kontrolliert. Der Name muss zum Beispiel von 3-20 Zeichen haben und darf nicht mit einem Leerzeichen beginnen. Bei nicht Beachtung dieser Anforderungen wird ebenfalls ein «Login error» ausgegeben, welcher die Anforderungen an den Namen beinhaltet.


Ein gültiger Benutzername wäre beispielsweise «Player1», mit welchem wir in dieser Demo auch fortfahren werden. Nach erfolgreicher Anmeldung wird uns dies bestätigt und die Anzahl aller bereits angemeldeten Spieler ausgegeben.



```
/login Player1  
Login successful!  
Players online: 1
```

Abbildung 17: Server-Ausgabe nach erfolgreicher Anmeldung¹²

Nun befinden wir uns in der Lobby, was uns einige weitere Befehlsmöglichkeiten bietet. Um einen Überblick über die möglichen Befehle zu erhalten, können wir «/help» eintippen und mit Enter bestätigen, was uns eine Liste aller Befehle ausgibt.



```
/help  
Your command words are:  
/chat /quit /help /who /play /stats
```

Abbildung 18: Server-Ausgabe der möglichen Befehle in der Lobby¹²

¹² Quelle: Eigene Darstellung aus der IntelliJ-Konsole des eigenen Programms

3.3 Lobby

In folgendem Abschnitt werden alle möglichen Befehle der Lobby aufgezeigt und kurz erläutert.

3.3.1 /chat

Die Lobby beinhaltet eine Chatfunktion, wodurch die Kommunikation der Spieler untereinander möglich ist. Dafür kann vor dem mitzuteilenden Text der Befehl «/chat» gefolgt von einem Leerschlag eingegeben werden. Die Nachfolgenden Textzeichen werden dann an alle Spieler ausgegeben, welche momentan Online sind und sich in der Lobby befinden.

```
/chat Hallo Player1, wie geht es dir?  
Player2: Hallo Player1, wie geht es dir?  
Player1: Hallo Player2, mir geht es gut danke :)
```

Abbildung 19: Beispiel einer Chat-Nachricht¹³

3.3.2 /quit

Mit dem Befehl «/quit» kann der Spieler sich aus dem Spiel abmelden. Dadurch wird der Spieler nicht mehr als online angezeigt und seine Spielerdaten werden entfernt. Somit werden auch alle gespeicherten Spielerdaten, wie beispielsweise die Spielstatistiken, entfernt und der Spielernamen steht für einen neuen Client wieder zur Verfügung.

```
/quit  
Thank you for playing. You can close the window. Good bye.
```

Abbildung 20: Server-Ausgabe nach Abmeldung¹³

3.3.3 /who

Der Befehl «/who» zeigt alle Spieler an, welche angemeldet sind. Ausserdem wird angezeigt in welchem Bereich (Lobby oder Ingame) sich der Spieler momentan befindet.

```
/who  
Player1 (LOBBY)  
Player2 (INGAME)  
Player3 (INGAME)
```

Abbildung 21: Server-Ausgabe nach Eingabe des Befehls "/who"¹³

¹³ Quelle: Eigene Darstellung aus der IntelliJ-Konsole des eigenen Programms

3.3.4 /stats

Im Spielerobjekt werden nach jedem Spiel Informationen über das Spielergebnis gespeichert. Diese Informationen können in der Lobby über den Befehl «/stats» abgerufen werden.

```
/stats  
Overall Game Stats:  
1 game(s) won  
1 game(s) drawn  
0 game(s) lost
```

Abbildung 22: Ausgabe der Spieler-Statistik¹⁴

3.3.5 /play

Der wichtigste aller Befehle ist wohl «/play». Mit dieser Eingabe startet die Suche nach einem würdigen Gegner. Während der Wartezeit werden den suchenden Spielern einige lustige Sprüche ausgegeben. Sobald jedoch ein anderer Spieler ebenfalls den Befehl «/play» eingegeben hat, wird ein Spiel gestartet.

```
/play  
Finding Opponent  
Nose and mouth breather  
Who Wants to Be a Millionaire Expert  
Gourd critic  
Cheater McCheaterson  
Really slow guy  
Worthy opponent found!
```

Abbildung 23: Server-Ausgabe während der Gegnersuche¹⁴

3.4 Im Spiel

Nach der erfolgreichen Spielersuche wird nun das Spiel gestartet. Kurz davor werden den Spielern noch die Eingabemöglichkeiten aufgelistet.

```
Worthy opponent found!  
Game starts in 5 Seconds! Prepare yourself!  
Use 1, 2 or 3 followed by enter to give an answers.
```

Abbildung 24: Server-Ausgabe nach erfolgreicher Gegnersuche¹⁴

¹⁴ Quelle: Eigene Darstellung aus der IntelliJ-Konsole des eigenen Programms

Daraufhin wird zufällig die erste Multiple Choice Frage ausgegeben, welche nun eine Eingabe von 1, 2 oder 3 der Spieler erwartet. Die Spieler haben maximal 10 Sekunden Zeit eine Antwort zu geben.

```
Weshalb warnen Ärzte vor der unkontrollierten Einnahme von Aktivkohle?  
(1) Die Partikel der Kohle reichern sich im Organismus an.  
(2) Medikamente wie die Antibabypille können unwirksam werden.  
(3) Die erhöhte Kohlenstoffdosis kann Bluthochdruck hervorrufen.
```

Abbildung 25: Ausgabe der Quizfrage¹⁵

Sobald ein Spieler seine Antwort eingegeben hat, wird diese unmittelbar ausgewertet. Hat ein Spieler die Frage als erstes richtig beantwortet, erfahren dies beide Spieler sofort nach der Eingabe und es wird die nächste Frage ausgegeben.

```
Weshalb warnen Ärzte vor der unkontrollierten Einnahme von Aktivkohle?  
(1) Die Partikel der Kohle reichern sich im Organismus an.  
(2) Medikamente wie die Antibabypille können unwirksam werden.  
(3) Die erhöhte Kohlenstoffdosis kann Bluthochdruck hervorrufen.  
  
2  
Great! Your answer was right. One Point for you!
```

Abbildung 26: Eigene Ausgabe nach korrekter Antwort¹⁵

Dem Gegner ist es somit nicht mehr möglich eine Antwort zu geben, da er leider zu spät war. Dies wird ihm fairerweise noch mitgeteilt, bevor die nächste Frage ausgegeben wird.

```
Player1 gave the correct answer: 2
```

Abbildung 27: Ausgabe nach korrekter Antwort des Gegners¹⁵

Wenn ein Spieler die Frage falsch beantwortet hat, erfährt er dies auch unmittelbar, jedoch wird dann noch die Antwort des Gegners abgewartet.

```
Weshalb warnen Ärzte vor der unkontrollierten Einnahme von Aktivkohle?  
(1) Die Partikel der Kohle reichern sich im Organismus an.  
(2) Medikamente wie die Antibabypille können unwirksam werden.  
(3) Die erhöhte Kohlenstoffdosis kann Bluthochdruck hervorrufen.  
  
1  
You know nothing, Jon Snow! More luck next question!
```

Abbildung 28: Eigene Ausgabe nach falscher Antwort¹⁵

¹⁵ Quelle: Eigene Darstellung aus der IntelliJ-Konsole des eigenen Programms

Nachdem alle Fragen beantwortet wurden oder die Zeit abgelaufen ist, wird beiden Spielern das Ergebnis ausgegeben. Im Hintergrund werden zudem die Spielergebnisse in der Spielerstatistik beider Spieler gespeichert. Nach Abschluss des Spiels befinden sich beide Spieler automatisch wieder in der Lobby.

```
The result:
Player1: 2
Player2: 0
Player1 won the game!
The Quiz is over. You are back in the Lobby.
```

Abbildung 29: Server-Ausgabe nach beenden des Spiels¹⁶

3.5 Handhabung von Client-Server Timeouts

Um den Server zu entlasten, wurde zudem eine Timeout Funktion implementiert. Diese Funktion meldet jeden Spieler vom Spiel ab, der länger als 10 Minuten keine Eingabe getätigt hat. Dies soll verhindern, dass Spieler die Netzwerkverbindung unnötig belasten. Ausserdem wird beim Schliessen der Clientverbindung auch der Spieler und dessen gesamte Statistik gelöscht. Dies ermöglicht es einem nächsten Spieler, sich mit demselben Benutzernamen anzumelden und zu spielen.

```
Timeout at 2021/06/15 04:01:59!
```

Abbildung 30: Server-Ausgabe nach Timeout¹⁶

¹⁶ Quelle: Eigene Darstellung aus der IntelliJ-Konsole des eigenen Programms

4 Funktionsweise der Klassen

Bevor die Funktionsweise der einzelnen Komponenten inkl. der dazugehörigen Designentscheidungen beschrieben wird, erfolgt eine kurze Erläuterung der Architektur, um einen besseren Einstieg in den Quellcode zu ermöglichen. Abgerundet wird jedes Kapitel durch eine Reflektion, in der weitere Implementierungsmöglichkeiten und Verbesserungen aufgeführt werden. Verbesserungen, die aufgrund des ohnehin schon sehr grossen Aufwandes, nicht mehr umgesetzt wurden.

4.1 Architektur

Das Klassendiagramm in Abbildung 31 wurde nach Abschluss der Programmierarbeiten erstellt und spiegelt den aktuellen Zustand wider. Es bietet einen Überblick über alle erstellten Klassen und stellt ausgewählte Beziehungen zwischen den Klassen dar. Es lässt sich grob in vier Bereiche unterteilen:

- Dateiimport: QuestionList, QuestionFile, Question, Answer
- Ausgabe: IOutput, OutputToStream, ColorCommandLines
- Parsing: Parser, CommandWords, Command, CommandWord
- Spiellogik und Ablauf: Alle restlichen Klassen

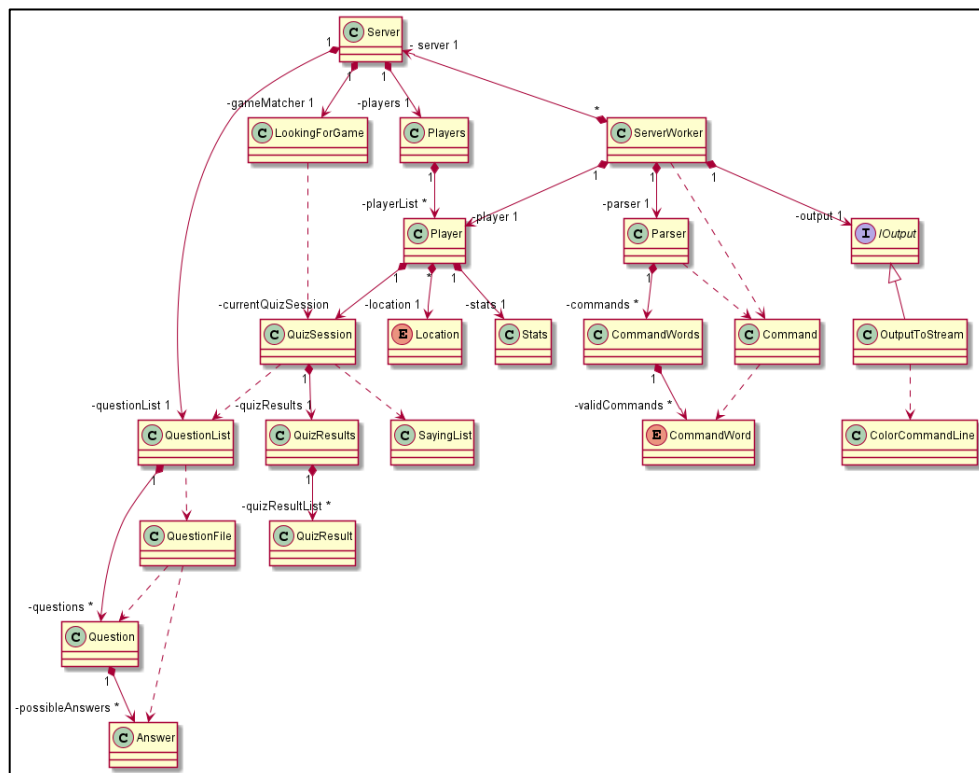


Abbildung 31: Abschliessendes Klassendiagramm des Multiplayer-Quiz¹⁷

¹⁷ Quelle: Eigenes UML Diagramm, erzeugt via PlantUML Plugin in IntelliJ

4.2 Kommunikation

Die `run()` Methode der Klasse `Server` wird in einem eigenen Thread ausgeführt und wartet innerhalb einer While-Schleife auf eingehende Verbindungen. Verbindet sich ein Client auf dem im Server definierten Port mittels der eigens dafür geschriebenen Client-Klasse oder über Telnet, wird die Verbindung ohne weitere Prüfungen akzeptiert. Die Klasse `ServerWorker` spiegelt eine Clientverbindung wider und für jede Clientverbindung wird ein eigener Thread erzeugt. Dem `ServerWorker` wird die Instanz des Servers und der `ClientSocket` im Konstruktor mitgegeben, so dass im Server keine Liste aller erstellten `ServerWorker` vorgehalten werden muss. Gerne hätten wir noch eine Konfigurationsdatei für den Server erstellt und diese beim Start ausgelesen, um den Server übersichtlich parametrisieren zu können. Da dies für die Kernfunktionalität aber nicht erforderlich ist, haben wir darauf verzichtet.

4.3 Serverattribute

Die Attribute des Servers sind gering an ihrer Zahl. Es wird eine Instanz vom Typ `java.util.logging.Logger` erstellt. Die Instanz wird an die anderen Klassen weitergereicht, um ein einheitliches logging in der gleichen Instanz zu erreichen. Auf eine Persistierung des Logs in eine Datei wurde aus zeitlichen Gründen verzichtet.

Im Server werden alle Spieler gespeichert. Diese Aufgabe übernimmt die Klasse `Players`. `Players` ist dafür verantwortlich einen eindeutigen Zeiger auf alle erzeugten `Player`-Instanzen zu halten. Die Spieler-Instanzen werden auch über die Klasse `Players` erzeugt, um die Eindeutigkeit des Namens sicherstellen zu können. Die Klasse `Players` sollte den Zugriff auf die Klasse `Player` kapseln und als Controller fungieren, leider haben wir das nicht konsequent durchgehalten und im Programm später direkt den Status der Klasse `Player` geändert. Die Aufgaben und Verantwortlichkeiten von `Players` und `Player` sollten in einem Refactoring überprüft werden. Dabei gehört auch die nicht lose Kopplung auf den Prüfstand.

Der Server hält eine Instanz auf die Klasse `QuestionList`. `QuestionList` bietet Zugriff auf die importierten Fragen und liefert über eine öffentliche Methode eine gewünschte Anzahl an unterschiedlichen Fragen zurück.

Zuletzt hält der Server einen Zeiger die `LookingForGame`-Instanz. Diese Klasse ist für die Zusammenstellung des Erzeugens der Quizsessions zuständig und entscheidet, wer gegen wen spielt.

4.4 Dateiimport

Der Dateiimport und die Verwaltung der Fragen und Antworten wurde unabhängig vom restlichen Spiel entwickelt und umgesetzt. So wurde es möglich, durch nur eine Methode in `QuestionList` alle benötigten Fragen für eine Quizsession zur Verfügung zu stellen. Das Vorgehen führte auch dazu, dass Informationen, die man in der Klasse `Frage` speichern könnte, zum Beispiel ob eine Frage bereits richtig beantwortet wurde oder ob alle Spieler zu dieser Frage eine Antwort gegeben haben, woanders gespeichert wurden.

Der Vorteil dabei ist, dass es sich bei diesen Informationen um einen anderen bounded context handelt. Die Informationen gehören zwar zum Objekt `Frage` im Sinne eines Quizes, aber nicht zum Objekt `Frage`, welches gerade importiert wurde. Diese strikte Trennung ermöglichte, die Klassen `Question` und `Answer` immutable zu implementieren. Der Nachteil dabei ist, dass es versäumt wurde, für den zweiten bounded context eine eigene Klasse zu erstellen, so dass Informationen zur Frage und Antwort direkt in anderen Klassen gespeichert wurden.

Das Einlesen der Daten erfolgt zeilenweise mittels `BufferedReader`. Da `Question` immutable ist, die Fragen und Antworten aber über mehrere Schleifendurchgänge hinweg eingelesen werden, wurde der Klasse `Question` ein Builder hinzugefügt. Evtl. hätte man diesen auch `Factory` nennen können. In der Schleife zum Einlesen der Daten (`QuestionFile.proces-Line()`) wird `IllegalArgumentException` abgefangen, die vom Builder beim Aufrufen vom `build()` geworfen werden kann. Der Builder reicht mögliche Exception von `Question` und `Answer` weiter. So wird im Konstruktor von `Question` zum Beispiel geprüft, ob eine Frage auch eine Antwort hat und ob mindestens eine Antwort als korrekt markiert ist. Die Exceptions werden an der Stelle abgefangen, um fehlerhafte Fragen einfach auslassen zu können, aber nicht den ganzen Importprozess abbrechen zu lassen.

4.5 Gameloop

Mit dem Gameloop ist die Hauptschleife des gesamten Spiels gemeint. Die Schleife wird so lange ausgeführt, bis das Spiel beendet wird. Der Loop wurde in den `ServerWorker` in die Methode `run` implementiert. Abbruchbedingung der Schleife ist ein boolean, der beim Eingeben von «/quit» auf `true` gesetzt wird und dadurch die Schleife beendet. Inhaltlich besteht der Gameloop nur aus zwei Zeilen. Der vom User eingegeben Text wird an einen Parser geschickt, der den Text in ein Kommando umwandelt, falls der Text einem gültigen Befehl entsprochen hat. Die zweite Zeile ist die Prozessierung des Commandos.

4.6 Parsing

Das Parsing wurde auf drei Klassen und ein Enum aufgeteilt. Jeder ServerWorker hält eine eigene Instanz der Klasse Parser. Dies ist notwendig, da der Parser den Zustand der Befehle verwaltet, ob der Befehl in der aktuellen Situation des Spiels verfügbar ist, oder nicht. Dies geschieht in der Klasse CommandWords, auf die der Parser eine Referenz hält. Befehle bzw. Kommandos selbst sind als Enum CommandWord gespeichert.

Ein Befehl besteht in unserem Spiel aus maximal zwei Wörtern. Das erste Wort ist der Befehl selbst, das zweite Wort ein optionales Argument. So gibt es Befehle, die kein zweites Wort benötigen («/play», «/help», «/quit»). «/login hans» ist hingegen ein Befehl, der ein zweites Wort verlangt.

Die öffentliche Methode `getCommand()` der Klasse Parser war ursprünglich recht übersichtlich. Die Eingabe des InputStreams wird auf einem String gespeichert, mittels `split()` wird das erste Wort vom Rest der Eingabe abgetrennt. `getCommandWord()` der Klasse CommandWords wird aufgerufen und man bekommt immer einen Enum zurück, der auch `unknown` sein kann, wenn das eingegeben Kommando nicht gefunden wurde. Mit dem Enum wird eine Instanz von Command erstellt, welche vom Parser zurückgegeben wird.

Leider ist die Methode `getCommand()` nun etwas unübersichtlicher, da für den Chat eine Ausnahme eingebaut werden musste. Der User soll nicht vor jeder Nachricht «/chat» eingeben müssen.

Diese Aufteilung auf mehrere Klassen und den Enum haben wir gewählt, um die Befehle vollständig von dem Spiel zu entkoppeln. Werden neue Befehle dem Spiel hinzugefügt, muss dies lediglich an zwei Stellen erfolgen: Das Enum muss um den neuen Befehl erweitert werden und der zentrale Switch-Case muss um den zusätzlichen Fall erweitert werden. Die Auslagerung als Enum bietet die Möglichkeit, an einer zentralen Stelle das Befehlswort zu ändern. Falls nicht mehr «/play» vom Anwender eingegeben werden soll, sondern stattdessen «/start», kann das einfach im Attribut vom Enum geändert werden. Im Enum ist auch die Bezeichnung des Befehls hinterlegt, so dass diese zentral geändert werden kann.

4.7 Ausgabe

Für die Ausgabe wurde das Interface `IOutput` angelegt. Das Interface ist für unsere Implementierung des Spiels nicht zwingend notwendig, soll aber aufzeigen, dass neben der Ausgabe auf der Kommandozeile andere UserInterfaces implementiert werden können. Möchte man das Spiel um ein GUI erweitern, muss lediglich das Interfaces `IOutput` neu implementiert werden und die Instanzierung des Interfaces im `ServerWorker` injectable gestaltet werden. Auf diesem Weg wäre auch eine Mehrsprachigkeit realisierbar, da die bestehende Klasse `OutputToStream` einfach ein weiteres Mal implementiert werden könnte. Für Mehrsprachigkeit gibt es aber sicher bessere Konzepte in Java, dies war aber nicht Teil des Projekts.

Alle Ausgaben erfolgen in der Klasse `OutputToStream`. `IOException` wird herausgereicht, um nicht den Logger hineingeben zu müssen. Eine Entscheidung, die hinterfragt werden könnte, wenn eine zusätzliche GUI-Implementierung erfolgt.

Die Klasse `ColorCommandLine` stellt statische Methoden zur Verfügung, um die Ausgabe der Buchstaben und deren Hintergrund farbig darstellen zu können. Da nicht alle Farben benötigt werden, aber alle Farben trotzdem für spätere Anpassungen verfügbar sein sollen, musste die Warnung bzgl. ungenutzter Attribute für die Klasse deaktiviert werden.

4.8 Login

Die Methode `handleLogin()` in `ServerWorker` wird aufgerufen, wenn der Spieler «/login» nach dem Start des Spiels eingibt. Die Validierung des Logins ist auf die Auswahl eines noch nicht vergebenen Namens beschränkt. Eine Userverwaltung inkl. Speicherung hätten den Rahmen des Projekts gesprengt. Zuerst wird geprüft, ob der User «/login» mit einem zusätzlichen Argument aufgerufen hat. Das zweite Argument ist sein gewünschter Spielername. Wurde ein Name eingegeben, wird geprüft, ob dieser den formalen Anforderungen genügt, also mind. 3 Zeichen und maximal 20 Zeichen lang ist und nicht mit einem Leerzeichen beginnt.

Zuletzt wird geprüft, ob der Name bereits in Verwendung ist. Diese Prüfung ist ausgelagert in die Klasse `Players`. Es wird der gewünschte Name in die Methode `createPlayerIfPossible()` übergeben und zurück erhält man eine gültige Instanz eines der Klasse `Player`. Wird der Name bereits von einem Spieler verwendet, wird null zurückgegeben.

Diese Prüfungen sind direkt im ServerWorker eingebaut. Im Rahmen eines Refactorings wären diese Aufgaben wahrscheinlich besser in der Klasse Player oder Players aufgehoben, um dort alle Entscheidungen zu einem Spieler zu bündeln.

4.9 GameMatching

Das GameMatching erfolgt in der Klasse LookingForGame. Die Klasse wird beim Hochfahren des Servers instanziiert. Sobald ein Spieler «/play» eingibt, um ein Spiel zu starten, wird ein Thread der Klasse LookingForGame gestartet. Dieser Thread wird sich nun um alle Spieler kümmern, die ein Quiz spielen wollen und erst wieder geschlossen, wenn kein Spieler mehr auf der Suche ist.

Die Spieler haben die Eigenschaft Location, ein Enum der angibt, wo der Spieler sich gerade befindet. Gibt der Spieler «/play» ein, wird die Location auf «LOOKING_FOR_GAME» gesetzt. So weiss der GameMatcher, um welche Spieler er sich kümmern muss.

Um eine Logik für das GameMatching zu programmieren, die das reine verfügbarsein von Spielern übersteigt, zum Beispiel Punkteschnittvergleich aus den letzten Games oder letzten Gegner vermeiden, hat leider nicht die Zeit gereicht und wurde in den Anforderungen auch nicht definiert. Die Klasse ist allerdings so vorbereitet, dass sich diese Erweiterungen einfach implementieren lassen sollten.

4.10 Quizsession

Die Quizsession wird von der Klasse LookingForGame instanziiert. Pro Quizsession wird ein Thread der Klasse QuizSession gestartet, da mehrere Sessions von unterschiedlichen Spielern parallel ausgetragen werden können.

In den Attributen von QuizSession wird gespeichert, welche Fragen zu stellen sind, wie das Ergebnis pro Frage ist und welche Spieler gegeneinander spielen. Zusätzlich werden Attribute verwaltet, die sich auf den aktuellen Zustand des Quiz beziehen, also auf die gerade gestellt Frage. Zu jeder Frage wird ein eigenes Ergebnisobjekt erstellt (QuizResult) und in einem boolean gespeichert, ob die Frage abgeschlossen (von beiden Spielern beantwortet oder von einem Spieler korrekt beantwortet) ist, um zur nächsten Frage springen zu können. Das sind wie bereits erwähnt Informationen, die weiter gekapselt werden können.

Die QuizSession gliedert sich in drei Teile. Einen vorbereitenden Teil prepareQuiz(), der einleitende Informationen zur Quizrunde an alle teilnehmenden Spieler schickt. Die

Methode `postQuestions()` beinhaltet die Hauptschleife der Quizrunde. Es werden in zeitlichen Abständen die Fragen ausgegeben. Um bereits vor Ablauf der eigentlichen Antwortfrist von 30 Sekunden eine neue Frage stellen zu können, wurde für das Warten eine eigene Schleife gebaut. Die Schleife wartet pro Durchgang 1 Sekunde und läuft 30 Mal. Bei jedem Durchgang wird geprüft, ob der zu Beginn erwähnte boolean inzwischen `true` ist. So kann das Warten vorzeitig unterbrochen werden.

Geben Spieler einen Antwortversuch ab, wird im `SeverWorker` die im `Player` gespeicherte Instanz von `QuizSession` verwendet, um die Methode `checkAnswer()` aufzurufen. Die Antwort wird geprüft und der Zustand von `QuizSession` und `QuizResult` aktualisiert.

Abgeschlossen wird eine Quizsession durch die Methode `closeQuiz()`. Zur Ergebnisberechnung wird eine Klasse der Methode `QuizResults` aufgerufen. Der Punktestand wird nun noch ausgegeben und der Gewinner gekürt.

Die Vermischung mit anderen Klassen ist sehr stark. Aufrufe wie `player.getWorker().getParser().switchToLobbyCommands()` zeugen von Problemen im Design. Leider hat die Zeit nicht ausgereicht, hier weiteres Refactoring durchzuführen.

4.11 Ergebnis berechnen

Für jede gestellte Frage wird ein Objekt von `QuizResult` erzeugt. `QuizResult` speichert, welche Spieler bereits einen Antwortversuch abgegeben haben, um mehrfache Versuche zu verhindern. Sobald ein Spieler die korrekte Antwort gibt, wird die Instanz des Spielers ebenfalls gespeichert.

Die Instanz `QuizResult` wird der Klasse `QuizResults` übergeben. `QuizResults` hat eine `Collection`, um alle `QuizResult`-Objekte innerhalb einer `QuizSession` zu speichern. Ausserdem bietet die Klasse eine Methode an, um das Ergebnis einer Quizsession zu berechnen und die Stats der Spieler zu aktualisieren. Dazu werden die `QuizResult`-Objekte iteriert und der Gewinner pro Frage erhält einen Punkt, falls es einen Spieler mit einer richtigen Antwort gab. Unentschieden sind somit möglich.

5 Kurzvorstellung der Unit Tests

Die Testklassen unseres Multiplayer Quiz wurden mit der externen Bibliothek «JUnit5.7.0» erstellt. Dabei haben wir vor allem Wert auf die wesentlichen und komplexeren Code Bestandteile gelegt. Nachfolgend wird kurz auf die Vorgehensweise der Testerstellung und die daraus gewonnenen Erkenntnisse eingegangen.

5.1 Vorgehensweise der Testerstellung

Das Schreiben der Codebestandteile wurde, wie in Abschnitt 3.1 bereits angedeutet, in verschiedene Blöcke eingeteilt. Dabei wurde vor allem die Logik der Fragen-Antwort Verarbeitung und die der Socket-Kommunikation voneinander gekapselt entwickelt. Das Schreiben der Unit Tests hatte ebenfalls diese Unterteilung, was sich im Nachhinein als schwieriger herausstellte, da sich die Klassenabhängigkeit der Socket-Kommunikation als sehr komplex erwies.

Für die Klassen, welche die Logik der Frage-Antwort Verarbeitung beinhalten, war die Erstellung von reinen Modultests ohne Probleme realisierbar. Jedoch mussten wir bei den Klassen der Socket-Kommunikation eine Mischung aus Modultests und Integrationstests vornehmen, da die dort inbegriffenen Klassen starke Abhängigkeiten aufwiesen und wir mit der Erstellung von Mock-Objekten nicht vertraut waren. Trotz der erwähnten Hybridvariante, erwiesen sich unsere Unit Tests als sehr hilfreich und wir konnten durch geschickt implementierte Tests mehrere Fehler finden und beheben.

5.2 Probleme durch Klassenabhängigkeiten

Die Komplexität der Socket-Kommunikation fiel bei der Erstellung der Unit Tests ins Gewicht, indem einige Tests nicht vollumfänglich gekapselt erstellt werden konnten. Um ein konkretes Beispiel zu nennen und um unseren Designentscheid etwas zu konkretisieren, sehen wir uns den Konstruktor der Klasse `ServerWorker` etwas genauer an.

```
public ServerWorker(final Server server, final Socket clientSocket) {
    Objects.requireNonNull(server, "text should not be null");
    Objects.requireNonNull(clientSocket, "text should not be null");

    this.logger = server.getLogger();
    this.players = server.getPlayerController();
    this.clientSocket = clientSocket;
    this.server = server;

    //.....
}
```

Abbildung 32: Codeausschnitt des `ServerWorker` Konstruktors¹⁸

¹⁸ Quelle: Eigene Darstellung aus eigenem Projektcode

Die Erzeugung eines ServerWorker Objektes benötigt ein Server und ein Socket Objekt als Übergabeparameter. Wir entschieden uns daher in der Testklasse des ServerWorkers eine setUp-Methode zu implementieren, welche die benötigten Objekte zur Verfügung stellt. Um von allen Testmethoden innerhalb der Testklasse auf die Instanzen zuzugreifen, wurden globale Variablen deklariert, welche durch die setUp-Methode instanziiert werden.

```
class ServerWorkerTest {
    private static Server server;
    private static Client client;
    private static final int SERVER_PORT = 60900;
    private static final int SAFESLEEP_MILLISECONDS = 100;

    private ServerWorker cut;

    @BeforeAll
    static void setUp() {
        server = new Server(SERVER_PORT);
        var serverThread = new Thread(server);
        serverThread.start();
        SleepUtils.safeSleep(TimeUnit.MILLISECONDS, SAFESLEEP_MILLISECONDS);
        client = Client.createClient("localhost", SERVER_PORT);
    }
}
```

Abbildung 33: Codeauschnitt der Testklasse ServerWorkerTest¹⁹

Um den ClientSocket für die Erzeugung des ServerWorkers zu erhalten, musste zusätzlich eine Socket-Kommunikation simuliert werden. Dazu mussten wir in der Hilfsmethode vor der Erzeugung des Client Objektes den ServerThread starten, welcher dann den Client erwartete. Aufgrund der Nebenläufigkeit terminierte dies nicht immer in der richtigen Reihenfolge, weshalb wir auf unsere safeSleep Methode der SleepUtil Klasse zurückgriffen. Dies gewährte dem serverThread etwas Zeit sich auf die Clientverbindung vorzubereiten und erzeugt somit erfolgreich die Client-Server Verbindung für unseren Test. Uns ist bewusst, dass ein so kompliziertes Setup nicht unbedingt einer Musterlösung für einen Unit Test gleicht, jedoch erfüllt es unsere Bedingungen, den Konstruktor des ServerWorkers zu testen.

5.3 Erkenntnisse und Verbesserungspotential

Um oben erwähnte Hybridvarianten im Unit Test zu vermeiden, hätten die Klassen der Socket Kommunikation mehr gekapselt werden sollen. Aufgrund des fortgeschrittenen Projektstandes entschieden wir uns jedoch, die Unit Tests auf unseren Projektcode anzupassen und nicht umgekehrt. Die Problematik, welche hierbei entstand, diente uns jedoch sehr zum Verständnis der Wichtigkeit von Kapselung einzelner Klassen.

¹⁹ Quelle: Eigene Darstellung aus eigenem Projektcode

6 Soll-Ist-Vergleich

Da es uns gelungen ist, alle unsere Muss-Anforderungen zu erfüllen, gehen wir hier nicht detailliert auf diese ein. Viel wichtiger und erfolgreicher sind jedoch die Kann-Anforderungen, welche bereits einen Platz in unserem Programm gefunden haben. Dabei handelt es sich vor allem um die Chatfunktion, welche in der Lobby bereits zur Verfügung steht.

Nachfolgend sind einige Funktionen aufgelistet, welche erst in der Feature-Liste aufgeführt waren oder sogar komplett neu hinzugekommen sind:

- Spieler können miteinander chatten.
- Ein Matchmaking aufgrund der Spielerstatistik wurde vorbereitet.
- Eine Timeout-Funktion meldet inaktive Benutzer ab.
- Die Spielerstatistik kann jederzeit in der Lobby abgerufen werden.
- Eine Benutzerverwaltung wurde vorbereitet.

7 Feedback

Die Erarbeitung dieser Projektarbeit erwies sich als äusserst wertvolles Instrument für die Festigung des im Kurs erlernten Stoffes. Da die Übungsaufgaben im Moodle teilweise etwas viel Vorstellungsvermögen voraussetzten, um den Zusammenhang in der Praxis zu sehen, half dieses Projekt umso mehr die Verknüpfungen der einzelnen Modulbestandteile zu verstehen.

Die Zusammenarbeit im Team hat gezeigt, dass eine detaillierte Vorbereitung und kontinuierlicher Austausch unabdingbar sind, um ein funktionierendes und designtechnisch sauberes Programm zu schreiben. Ausserdem hat dies uns zum ersten Mal ermöglicht, Git in der Praxis einzusetzen und von dessen Vorteilen zu profitieren.

Als besonders anspruchsvoll erwies sich die Erarbeitung der Server-Socket Kenntnisse, welche für dieses Projekt eine tragende Rolle spielten. Dieser Faktor erfüllt uns umso mehr mit Stolz, ein solides und vollwertiges Spiel entwickelt zu haben.

Literaturverzeichnis

FFHS (2021); *Projektarbeiten; Thema 1 | Multiplayer-Quiz*; URL:

<https://moodle.ffhs.ch/mod/page/view.php?id=3784957> (besucht am 25.04.2021)

Trello (2021), *Trello | Board*; Login und Berechtigung auf Board nötig; URL:

<https://trello.com/b/5IBJRhef/board> (besucht am 25.04.2021)