Super-Superhirn

Projektbeschreibung

Software Engineering 2
WS 2023/2024

Eingereicht von:
Louis Beul [588411]
Till Giesa [584373]
Ole Gleß [564153]
Philipp Röber [584672]
Marc Sabrautzki [584673]

Inhaltsverzeichnis

1. Einleitung

Im Zuge des Moduls Software-Engineering 2 wurde das Spiel Super-Superhirn, eine erweiterte Version des Brettspiels Superhirn, in Deutschland auch unter dem Namen Mastermind bekannt, in Python implementiert. Während des Semesters wurden mehrere Kundengespräche durch den Dozenten Professor Dr. Salinger simuliert, woraus die Gruppe Anforderungen erheben und auf Anforderungsänderungen eingehen musste. Im Vordergrund steht die richtige Umsetzung des Spiels und das Arbeiten mit den, im Semester erlernten, Qualitätssicherungsmaßnahmen. Diese Projektdokumentation hält dabei unsere Arbeit fest und reflektiert unseren Umgang mit der Aufgabenstellung.

1.1. Aufgabe

Ziel ist die vollständige Implementierung des Brettspiels Superhirn und die Erweiterung des Spiels zu Super-Superhirn. Das Grundprinzip des Spiels bleibt bestehen, ein Rater versucht einen Farbcode in einer festgelegten Anzahl von Zügen aufzudecken. Der Codierer beantwortet jeden Zug mit farblichem Feedback. Eine Anzahl von weißen Stiften, sollte der Versuch die richtige Farbe enthalten und eine Anzahl von schwarzen Stiften, sollte die Farbe und Position an einer Stelle richtig sein. Der Rater versucht sich so, von Runde zu Runde, an den gesuchten Code anzunähern und schließlich zu erraten. Gelingt es dem Ratenden nicht in der vorgegebenen Anzahl von Zügen den Code zu erraten, gewinnt der Codierer das Spiel. In der hier umgesetzten Version Super-Superhirn kann mit verschiedenen Parametern gespielt werden. Der Code kann aus vier oder fünf Stellen bestehen und wird aus zwei bis acht Farben zusammengesetzt. Die Anzahl der Züge orientiert sich mit zehn Rateversuchen am Original. Das Spiel selbst wird in einem Terminal gespielt, die Farben werden durch Zahlen von 1 bis 8 dargestellt.

1.2. Team

- Louis Beul [588411]
- Till Giesa [584373]
- Ole Gleß [564153]
- Philipp Röber [584672]
- Marc Sabrautzki [584673]

1.3. Software-Prozess

Bei der Wahl des Prozesses fiel unsere Entscheidung auf das Agile Framework Scrum. Der entscheidende Faktor hierfür war vor allem die Ankündigung unseres Kunden, dass es zu Anforderungsänderungen kommen wird. Mit einem dynamischen und iterativen Modell wie Scrum kann man diese Änderungen leicht umsetzen, zudem ermöglicht dieses Framework eine frühzeitige Anfertigung eines Prototypens, welcher dem Kunden zum Einholen von Feedback präsentiert werden kann. Dabei haben wir,

für Scrum typische Werkzeuge eingesetzt, wie etwa Sprints und den Einsatz von User Stories. Die typische Sprintzeit von zwei Wochen konnte hier optimal auf die, alle zwei Wochen stattfindenden, Zwischenpräsentationen angewandt werden und die User Stories konnten den Spielablauf gut darstellen. Im Gegensatz zu den genannten Werkzeugen haben wir jedoch nicht alle Vorgehensweisen, wie Daily Standups oder die klassischen internen Teamrollen wie Product Owner oder Scrum-Master übernommen. Grund für das Aussetzen der Daily Standups war, dass in unseren individuellen Terminplanungen ein tägliches Treffen praktisch nicht umsetzbar war. Zudem hätte ein tägliches Meeting auch zu keinem erheblichen Mehrwehrt geführt. Wir entschieden uns für die schnelle Kommunikation per Instant Messaging und für die Einführung eines Weekly Cycles, durchgeführt immer am Sonntagabend. Zu diesen Terminen wurden die aktuellen Aufgaben besprochen und zugewiesen. Die klassische Scrum-Rollenverteilung wurde nicht strikt umgesetzt, da in unserer eher kleinen Gruppe kein Vorteil von einem Scrum Master oder einem Product Owner gesehen wurde. Stattdessen wurden die Verantwortlichkeiten in gleichen Teilen auf alle Mitglieder verteilt und auf deren Eigenverantwortung gesetzt. Zusätzlich haben wir den agilen Prozess um ein Kanban Board ergänzt, um eine bessere Visualisierung unseres Fortschrittes zu erhalten und Probleme bei zugewiesenen Aufgaben schnell ausfindig machen zu können.

2. Analyse

Der folgende Abschnitt erläutert die dem Super-Superhirn zugrundeliegende Analysephase. Das genaue Herausarbeiten von Anforderungen und ihr Umfang wurde mithilfe der folgenden Struktur durchgeführt.

2.1. Analyseprozess

Der erste Schritt der objektorientierten Analyse ist die Erhebung von Anforderungen, sie bilden die Basis für die spätere Überführung in Use Cases und die daraus resultierenden Modelle. Zu Beginn des Projekts haben wir verschiedene Quellen gesichtet und Anforderungen herausgarbeitet, diese werden im Folgenden näher beleuchtet.

2.1.1. Sichtung existierender Dokumente

Das erste uns vorliegende Anforderungsartefakt war eine vom Kunden bereitgestellte Projektbeschreibung. Diese liefert die Spielregeln, einen exemplarischen Spielablauf, sowie weitere gewünschte Aspekte [vgl. 1]. Sie beinhaltet zusätzlich den zeitlichen Rahmen für die Umsetzung des Projekts. Da das gewünschte Produkt auf dem Spiel Superhirn aufbaut, wurde anschließend eine Spielanleitung der Firma Hasbro gesichtet [2]. Anhand dieser Anleitung ließen sich gewisse Unschärfen klären, zudem konnte sich auch im weiteren Verlauf hieran orientiert werden. Denn die Vorgaben des Kunden weisen zwar gewisse Abweichungen zum Spiel auf, trotzdem bleibt der ungefähre Spielablauf erhalten.

2.1.2. Introspektion

Nachdem der Spielablauf und die Spielregeln definiert wurden, wurde ein Spiel mit Hilfe von Stift und Papier in der Gruppe durchgeführt. Eine Person hat die Rolle des Raters eingenommen, eine weitere die Rolle des Codierers. Die anderen Gruppenmitglieder haben den Spielablauf beobachtet. Ziel war es das Spielverständnis aller Gruppenmitglieder auf den gleichen Stand zu bringen und verschiedene Spielabläufe und Szenarien zu simulieren.

2.1.3. Interview

In einem auf 15 Minuten begrenzten Interview mit dem Kunden konnten Fragen zu Rahmenbedingungen, zur gewünschten Spiellogik und zur Benutzung erörtert und geklärt werden. Zur effektiven Nutzung der Zeit wurden im Voraus verschiedene Themengebiete aufgeteilt:

- Rahmenbedingungen (L. Beul)
- Spiellogik (T. Giesa)
- Ablauf und Benutzung (P. Röber)
- Detailfragen (O. Gleß)
- Dokumentation (M. Sabrautzki)

Durch die vorherige Aufteilung konnten im Interview alle Fragen strukturiert abgearbeitet werden, die Mitschrift des Interviews hat zusammen mit den vorliegenden

Dokumenten die Grundlage für die anschließend ausformulierten Use Cases gebildet.

2.2. Analyseergebnisse

Die Analyseergebnisse wurden zunächst mit Use-Cases schriftlich ausformuliert und im Anschluss durch Objektmodelle und dynamsiche Modelle erweitert. Zur Veranschaulichung zeigen wir in den nächsten Abschnitten Auszüge aus diesen Arbeitsschritten.

2.2.1. Anwendungsfälle

Auf Grundlage der vorliegenden Dokumente, der Interviewmitschrift und unseren Spielerfahrungen wurden durch alle Gruppenmitglieder Use-Cases geschrieben. Diese wurden bei einem Zusammentreffen gegenseitig vorgestellt, um Gemeinsamkeiten, aber noch wichtiger, Differenzen zu erkennen. Verschiedene Interpretationen, welche im späteren Verlauf zu Problemen führen könnten, wurden so ausgeschlossen. Im Anschluss wurden gemeinsam sechs elementare Use-Cases definiert, vier von ihnen werden im Folgenden tabelarisch aufgeführt. Ergänzend dazu finden sich im Anhang die weiteren Use Cases zum Nachlesen A.1.1, A.1.2.

Spielstart mit Rollenwahl

Name	Spielstart mit Rollenwahl
Primärakteur	Spieler
Anwendungsbereich	GUI
Stakeholder	Spieler
Vorbedingung	• Das Programm wurde auf dem Rechner installiert • Das Programm wurde erfolgreich gestartet
Mindestzusicherung	• Spieler erhält entsprechende Fehlermeldung, sollte es zu Fehlern bei der Rollenauswahl kommen
Zusicherung im Erfolgsfall	• Das Spiel startet mit der vom Spieler gewählten Rolle
Haupterfolgsszenario	• GUI bittet um Angabe der gewünschten Rolle • Spieler gibt die gewünschte Rolle an • Spieler erhält visuelles Feedback zur erfolgreichen Rollenwahl • Gegenrolle wird vom automatisch gesetzt • Beginn des ersten Spielzugs
Erweiterungen	• Fehlermeldung bei einer falschen Eingabe

Erster Zug Codierer

Name	Erster Zug aus menschlicher Codierer-Perspektive
Primärakteur	Spieler
Anwendungsbereich	GUI / Spiellogik
Stakeholder	Spieler
Vorbedingung	• Spieler hat die Rolle des Codierers belegt und Spiel wurde erfolgreich gestartet
Mindestzusicherung	• Spieler erhält im Fehlerfall eine entsprechende Fehlermeldung
Zusicherung im Erfolgsfall	• Der Spieler hat einen validen Code gesetzt, der für die Dauer des aktuellen Spiels persistiert wird, danach ist der Rater am Zug
Haupterfolgsszenario	• GUI bittet um Eingabe eines Codes • User gibt einen frei gewählten (validen) Code ein • GUI bestätigt Validität des Codes • GUI bestätigt Persistierung für den Verlauf des laufenden Spiels • Der Zug wird automatisch beendet • Zugwechsel wird eingeleitet
Erweiterungen	• Fehlermeldung bei einer falschen Eingabe

Spielzug Rater

Name	Spielzug aus menschlicher Rater-Perspektive
Primärakteur	Spieler
Anwendungsbereich	GUI / Spiellogik
Stakeholder	Spieler
Vorbedingung	• Das Spiel wurde erfolgreich gestartet • Der Spieler hat die Rolle des Raters belegt • Der Codierer hat einen Code für das laufende Spiel festgelegt • Bisherige Rateversuche und deren Bewertungen werden angezeigt • Restrateversuche größer als Null
Mindestzusicherung	• Spieler erhält im Fehlerfall eine entsprechende Fehlermeldung
Zusicherung im Erfolgsfall	• Der Spieler hat einen validen Rateversuch unternommen und bekommt ein entsprechendes Feedback vom Computer
Haupterfolgsszenario	• GUI bittet um Eingabe eines Rateversuches in Form eines Codes • User gibt einen validen Code ein • GUI bestätigt Validität des Codes und damit den Rateversuch • GUI gibt eine Bewertung des Versuchs als visuelles Feedback zurück • GUI gibt bekannt, dass nächster Zug nun startet • Der nächste Zug als Rater wird gestartet
Erweiterungen	• Fehlermeldung bei Eingabe eines nicht validen Codes • Vorzeitiges Spielende im Gewinnfall • Spielende bei Ausschöpfung der Maximalversuche

Normaler Spielzug Codierer

Name	(Nicht-erster) Spielzug aus menschlicher Codierer-Perspektive
Primärakteur	Spieler
Anwendungsbereich	GUI / Spiellogik
Stakeholder	Spieler
Vorbedingung	• Spieler hat das Spiel als Codierer gestartet und im ersten Zug einen Code gesetzt • Der Rater hat mindestens einen Rateversuch unternommen
Mindestzusicherung	• Spieler erhält im Fehlerfall eine entsprechende Fehlermeldung
Zusicherung im Erfolgsfall	• Der Spieler hat einen validen Rateversuch unternommen und bekommt ein entsprechendes Feedback
Haupterfolgsszenario	• GUI bittet um Eingabe eines Rateversuches in Form eines Codes • User gibt einen validen Code ein • GUI bestätigt Validität des Codes und damit des Rateversuchs • GUI gibt eine computergenerierte Bewertung des Versuchs als visuelles Feedback zurück • GUI gibt bekannt, dass nächster Zug nun startet • Der nächste Zug als Rater wird gestartet
Erweiterungen	• Fehlermeldung bei Eingabe eines nicht validen Codes • Vorzeitiges Spielende im Gewinnfall • Spielende bei Ausschöpfung der Maximalversuche

2.2.2. Objektmodell

Zusätzlich zu den ausformulierten Use Cases wurden verschiedene UML Diagramme erstellt. Die nachfolgende Abbildung zeigt alle Akteure des Spiels.

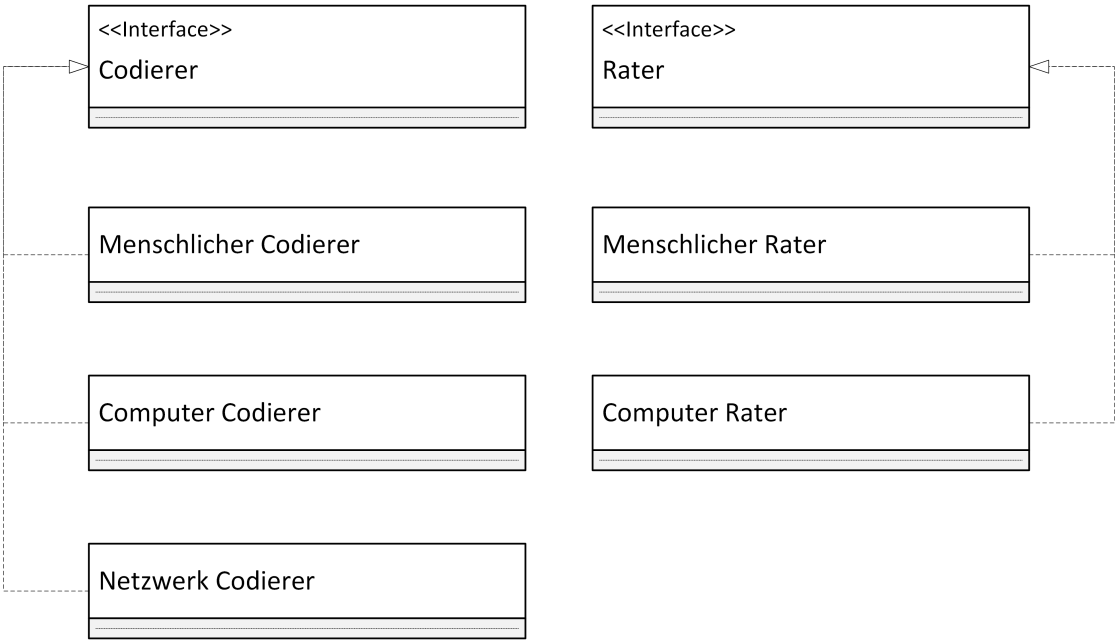

Abbildung 2.1.: *Akteure*

Zu sehen ist hier, dass die beiden Hauptrollen im Spiel Rater und Codierer sein sollen. Diese werden durch Interfaces dargestellt, da jeweils der Codierer und der Rater in einen menschlichen und einen Computer-Akteur aufgeteilt werden können. Im Laufe des Projekts wünschte sich der Kunde auch die Anbindung an einen Netzwerk-Codierer. Entscheidet sich der Spieler beispielsweise für die Rolle des Codierers, so nimmt der Computer automatisch den Gegenpart des Raters ein. Da aber beide Rollen, sowohl des Raters als auch des Codierers, ähnliche Grundzüge haben, bietet sich das Interface hier an. Als nächstes wurde eine Spielrunde und deren Inhalte dargestellt.

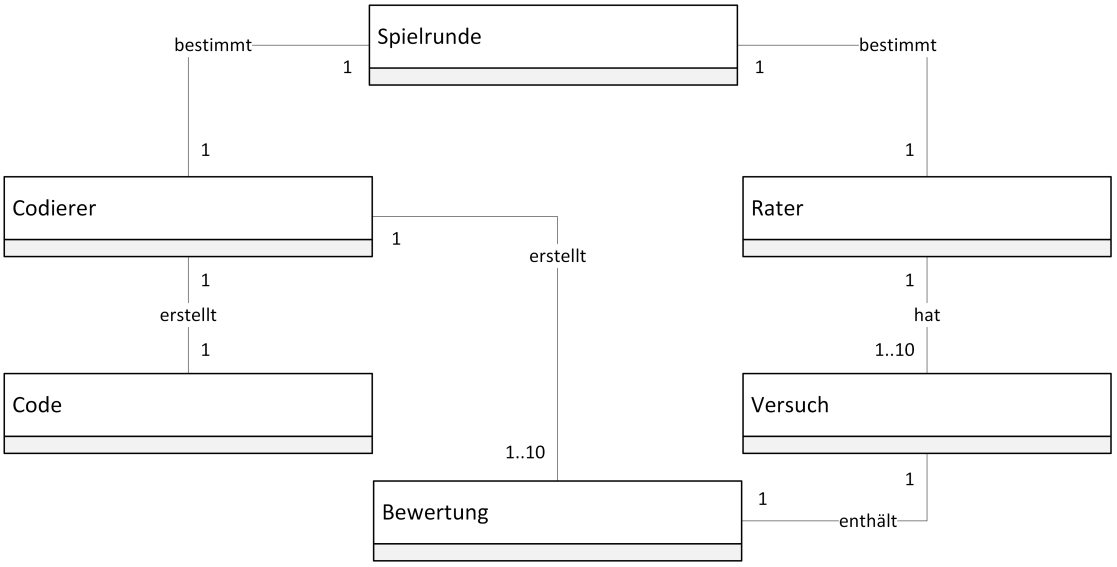

Abbildung 2.2.: *Spielrunde*

In diesem Entwurf ist zu sehen, dass eine Spielrunde aus jeweils einem Codierer und einem Rater besteht, wie auch in der vorherigen Grafik, jedoch werden diese Akteure hier präzisiert. Ein Rater hat nämlich bis zu zehn Versuche, um den richtigen Code des Codierers zu erraten. Dabei kann der Codierer einmalig diesen Code erstellen, und im weiteren Spielverlauf dann jeweils die Versuche des Raters mit einer Bewertung versehen. Ein Code enthält vier oder fünf Positionen, welche wiederum durch Farben dargestellt werden. Bei diesen Farben bietet sich eine Enumeration an. Ein Versuch enthält immer eine Bewertung, welche für die vier bzw. fünf gesetzten Steine jeweils entweder kein Feedback oder ein Feedback in Form eines weißen bzw. schwarzen Stiftes gibt.

2.2.3. Dynamisches Modell

Zur Darstellung, wie genau ein Zug abläuft, eignet sich ein dynamisches Modell. Die nachfolgende Grafik zeigt einen Zug des Rates.

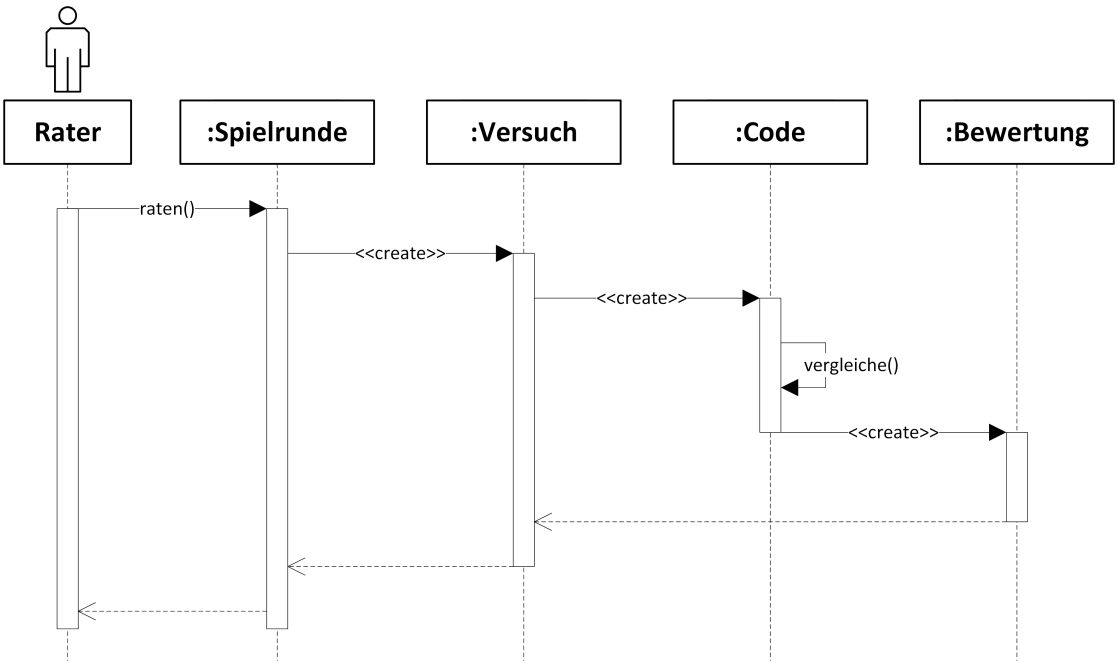

Abbildung 2.3.: *Dynamisches Modell*

Hier abgebildet sieht man, dass der Rater durch den Vorgang des Ratens sich im Objekt der Spielrunde befindet. Konkreter erstellt er dann einen Versuch, der wiederum durch einen erstellten Code repräsentiert wird. Durch Vergleichen dieses geratenen Code mit dem gesetzten Code wird eine Bewertung erstellt, wie gut dieser Versuch ist. Nach dieser Bewertung geht der Zug zurück an den Rater, der wieder von vorne beginnt, bis er das Spiel beendet, seine maximale Zugzahl überschreitet oder richtig rät.

3. Design

Im Folgenden wird die Designphase des Projekts vorgestellt. Es wird auf die gewählte Architektur eingegangen und wie diese umgesetzt werden soll. Zur Veranschaulichung wird zunächst der Grobentwurf vorgestellt, die einzelnen Komponenten werden anschließend im Feinentwurf näher beleuchtet.

3.1. Architekturstil

Zur Umsetzung wurde der Architekturstil der Drei-Schicht Architektur gewählt. Die drei Schichten sind der Client, die Logik und die Spieldaten. Die Client-Schicht bildet die Schnittstelle zum Benutzer, sorgt für die Oberflächen und nimmt Eingaben entgegen. Der Spielablauf selbst wird in der Logik-Schicht gesteuert. Alle für das Spiel benötigten Werte werden in der Daten-Schicht gespeichert.

Da sich im Gespräch mit dem Kunden Änderungen und Erweiterungen abzeichneten, haben wir uns für eine modulare Architektur entschieden. Dieser Architekturstil wurde gewählt, weil die einzelnen Module für sich erweiterbar oder sogar komplett austauschbar sind.

Während der späteren Implementierung wurde schnell deutlich, dass es zu mehreren recht starken Kopplungen zwischen Data- und Logic-Schicht kommen könnte. Deshalb entscheiden wir uns dazu, die bis dahin bestehende Architektur insofern zu optimieren, alle funktionalen Komponenten der Daten-Schicht in die Logik-Schicht zu extrahieren, wo sie unter anderem auch Utility-Methoden zur Typkonvertierung und beispielsweise Vergleichsoperationen zur Verfügung stellen können. Im Zuge dessen wurde die Daten-Schicht zu einem rein passiven Datenspeicher umgeformt, an dessen Speicherständen sich die übrigen Schichten bedienen können. Somit entwickelte sich die initiale Schichtenarchitektur zu einer Mischung aus Schichten- und Repository-Architektur, bei der die Data-Schicht als Repository und die übrigen zwei Schichten als Agenten fungieren.

3.2. Grobentwurf

Im Folgenden ist der Grobentwurf für unser Spiel abgebildet. Besonders auffällig ist dabei zunächst, dass die drei Schichten einen inhomogenen Umfang haben. So bestehen die Module View und Data jeweils nur aus einer einzelnen Klasse. Das Logic Modul enthält alle Klassen und Methoden zur eigentlichen Spiellogik. Dieser Aufbau spiegelt auch den Arbeitsaufwand wider, den wir für die Implementierung als notwendig erachten. Die visuelle Ausgabe auf dem Terminal und das Speichern der Spielparameter sind im Vergleich weniger umfangreich.

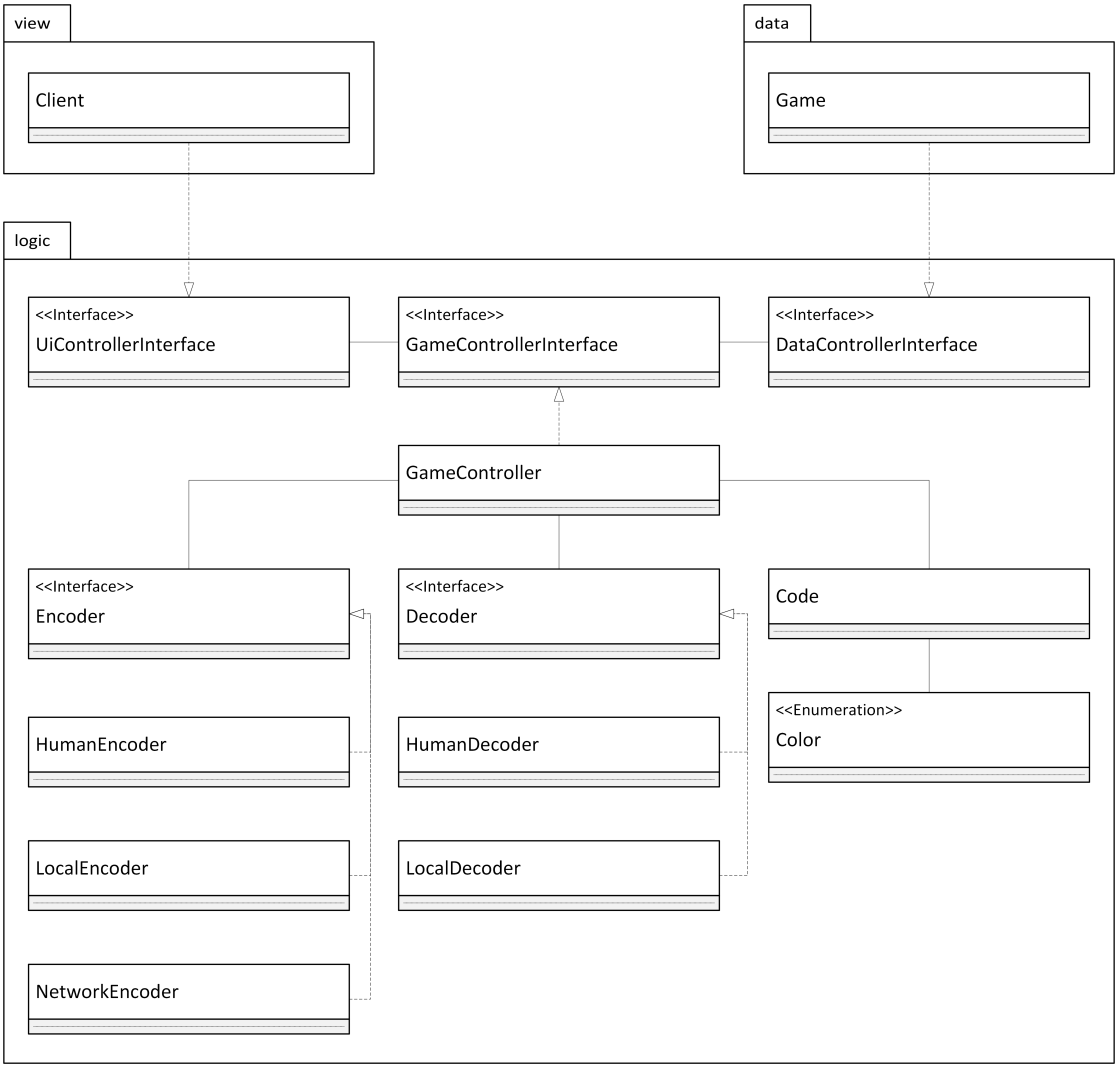

Abbildung 3.1.: *Grobentwurf*

3.3. Feinentwurf

Als letzte Entwurfszeichnung wurde der Feinentwurf angefertigt, um den einzelnen Klassen und Interfaces auch Attribute und Methoden zuzuweisen. Dieser Feinentwurf kann als Vorgabe für die nächste Phase, die Implementierung, genutzt werden. Durch den Entwurf der Codierer-Schnittstelle „EncoderInterface“ war die nachträgliche Ergänzung der Netzwerkfunktionalität ohne größere Umstände realisierbar. Die bereits bestehenden Signaturen „generate_code“ und „rate“ mussten nicht geändert werden. Es wurde lediglich eine neue Codierer-Implementierung realisiert, welche den Kontakt zum Server herstellt und die Rate-Versuche und Bewertungen von Python-Datentypen in JSON bzw. aus Python-Code umwandelt. Um die Lesbarkeit des Feinentwurfs zu verbessern, werden die Schichten einzeln abgebildet. Die Abhängigkeiten zwischen ihnen sind im Grobentwurf dargestellt.

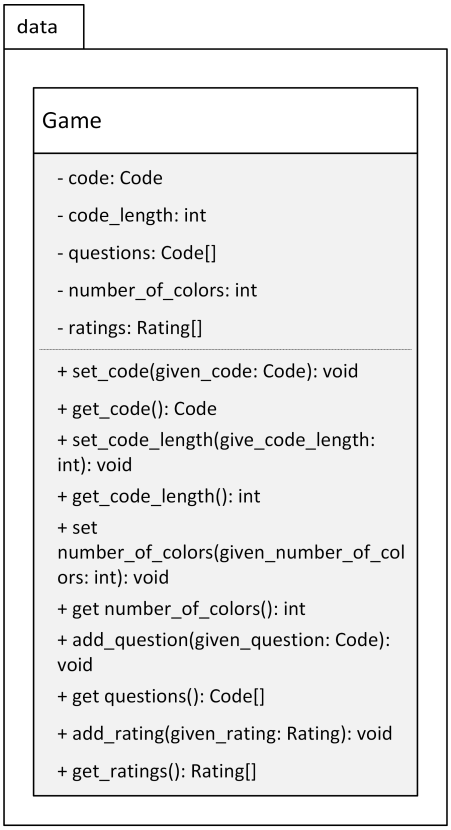

Abbildung 3.2.: *Feinentwurf: Data*

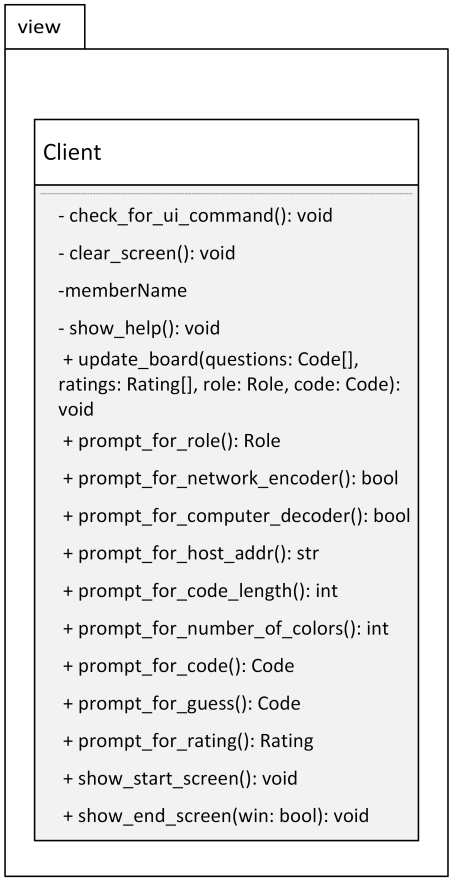

Abbildung 3.3.: *Feinentwurf: View*

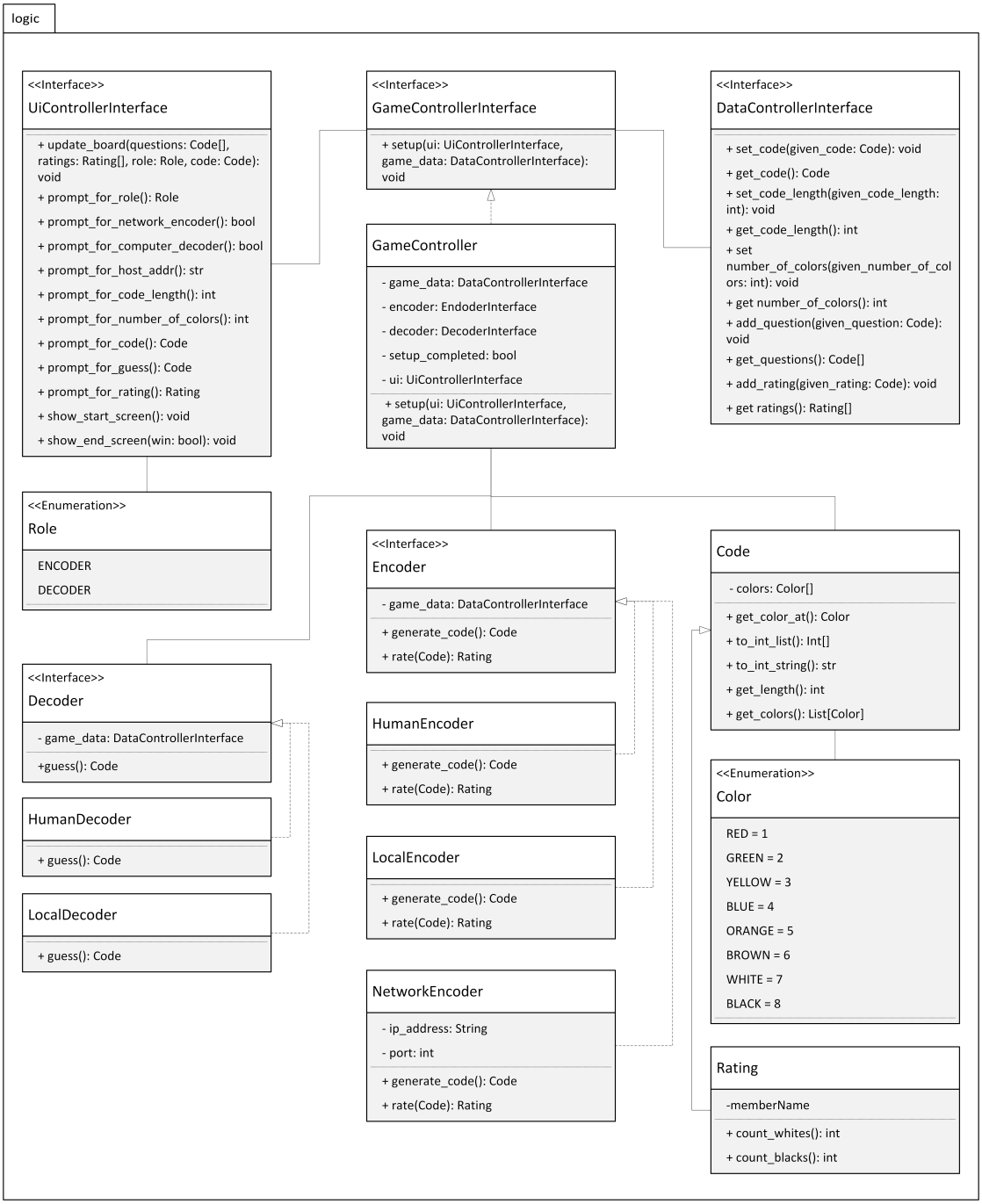

Abbildung 3.4.: *Feinentwurf: Logic*

4. Implementierung

In der Implementierungsphase wurden dann die Ergebnisse aus dem Feinentwurf in Code überführt. Dazu wurden in unserem Projekt die folgende Ordner und Dateistruktur angelegt, an welcher sich dann orientiert wurde:

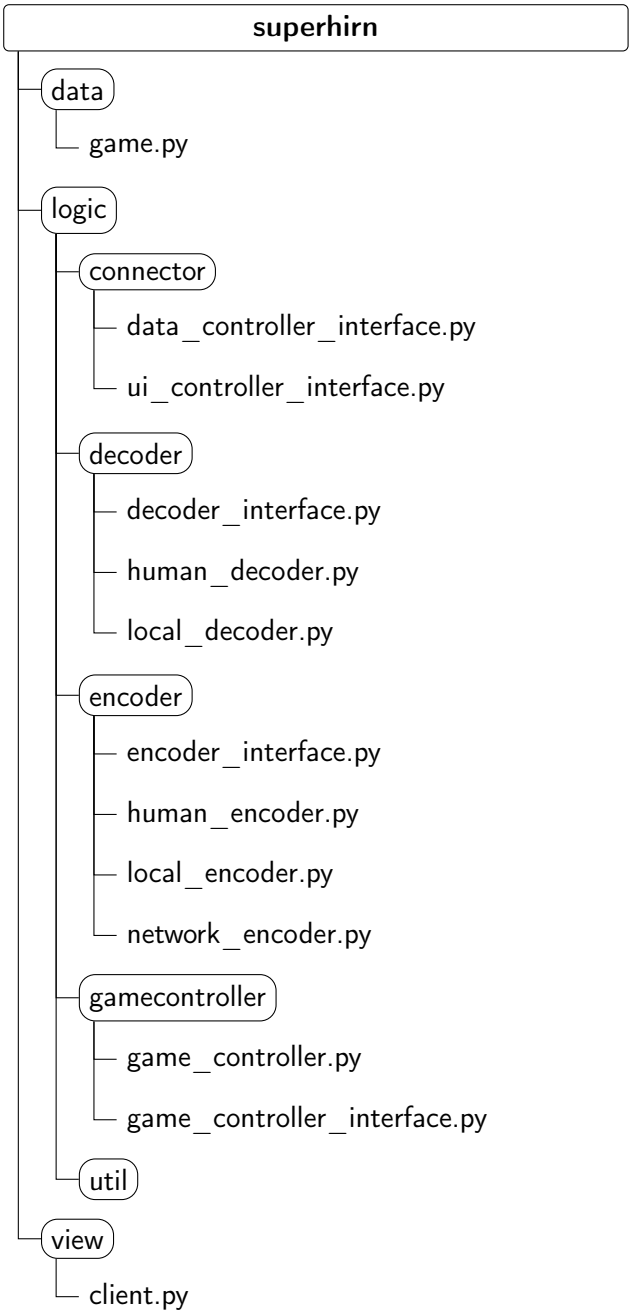

Abbildung 4.1.: *Ordnerstruktur*

Dabei ist auch zu bemerken, dass die Tests zu den jeweiligen Units direkt gespiegelt angelegt wurden, um eine bessere Übersicht zu garantieren.

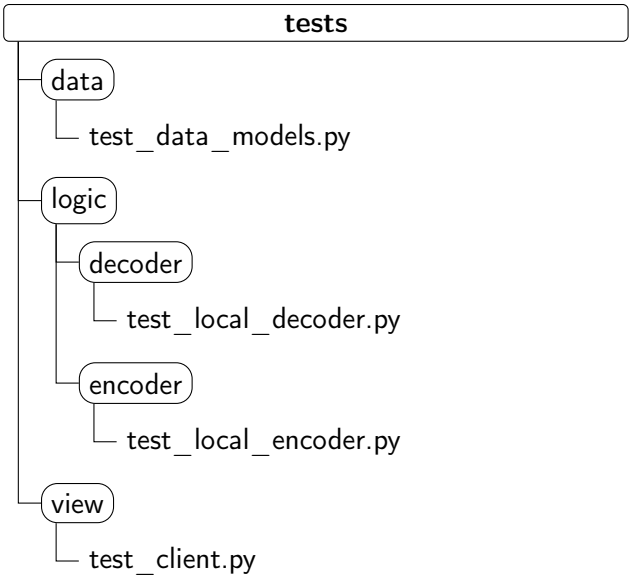

Abbildung 4.2.: *Ordnerstruktur Test*

Interessante Implementierungsmerkmale waren hier vor allem die Verwendung von generalistischen Interfaces für die Decoder und Encoder. Wie in dem Kapitel Feinentwurf 3.3 bereits erwähnt, konnten durch die Interfaces Anforderungsänderungen wie die Netzwerkkomponente schnell umgesetzt werden. Die Modularisierung ist dadurch enorm verbessert worden.

Eine weitere besondere Herausforderung in der Implementierungsphase war die Lösungsstrategie des Lokalen Codierers. Hierbei sollte man sich im Spiel einen Code ausdenken, der dann vom Computer erraten wird. Da die Vorgabe hier war, dass der Computer methodisch lösen soll, wurde ein reines zufallsbasiertes Ratesystem ausgeschlossen. In der Implementierung orientieren wir uns an der Lösung von Informatiker Donald E. Knuth, welcher ein Spiel mit sechs Farben und vier Steinen immer in maximal fünf Zügen lösen kann [vgl. 3]. Dabei wird zunächst ein Set aus allen möglichen Bewertungen erstellt und nach der Eingabe des Feedbacks aus diesem Set die nicht mehr möglichen Varianten eliminiert. Jedoch verwendet Knuth hier das Verfahren MinMax, um nicht immer den ersten Set Eintrag zu nehmen. Anzumerken ist, dass es Algorithmen gibt welche im Durchschnitt mit weniger Versuchen gewinnen können. Allerdings wurde der sogenannte “Five-Guess-Algorithm” von Knuth so entworfen, dass dieser für verschiedene Mastermind-Regelvarianten anwendbar ist.

Das letzte erwähnenswerte Implementierungsmerkmal war die Einführung des Color Enums, welches oftmals bei der Datenweitergabe über verschiedene Schichten hinweg zunächst schwer umzusetzen war. Jedoch war dieser etwas höhere Aufwand dadurch begründet, dass so spätere potentielle Erweiterungen des Spiels, wie beispielsweise eine GUI, leichter umgesetzt werden könnten.

5. Qualitätssicherung

Im gesamten Projektablauf wurden verschiedene Maßnahmen ergriffen, auf einige, für uns wichtige, möchten wir einzeln eingehen. Andere eingesetzte Methoden aus Scrum und Extreme-Programming beleuchten wir hingegen nicht ausführlich. Sie sind Teil unserer Strategie zur Qualitätssicherung während des Projekts. Sie basiert darauf, dass sich alle Gruppenmitglieder jederzeit über den aktuellen Stand informieren können, Änderungen nicht ohne Rücksprache übernommen werden können und Probleme offen und direkt angesprochen werden.

5.1. Strategie

Neben den Werkzeugen und Methoden, auf die wir im nächsten Kapitel näher eingehen, hat unser Git-Repository die Grundlage für unsere Qualitätssicherung gebildet. Es hat als sogenannte Single Point of Truth gedient. Das interne Wiki hat alle projektspezifischen Begriffe definiert, ein Rateversuch ist beispielsweise eine Frage und die Bewertung des Encoders ist eine Antwort. Der Aktivitäten Tab stellt dazu sicher, dass alle Änderungen einsehbar und zurückverfolgbar sind. Der Main-Branch wurde geschützt, sodass Änderungen an ihm nur bei Zustimmung von mindestens zwei weiteren Gruppenmitglieder übernommen werden. Die Teams haben sich beim Pair-Programming immer einen neuen Branch erstellt und aus den Commits ist ersichtlich, wer an ihnen gearbeitet hat. Ergänzt wurde das Git-Repository durch ein Kanbanboard für offene Aufgaben.

5.2. Maßnahmen

Eine Projektanforderung, die klar zu Qualitätssicherung zu zählen ist, ist das Pair-Programming. In diesem Projekt haben wir das Pair-Programming zum “Pair-Pair-Programming“ erweitert. Da unsere Main-Branch protected ist, muss der Code bei einer Merge-Request von einem weiteren Team überprüft und freigegeben werden. Dadurch ist sichergestellt, dass jede Änderung von mindestens vier Gruppenmitgliedern bearbeitet wurde. Im Verlauf des Projekts hat sich gezeigt, dass Änderungen durch das zweite Team auch abgelehnt wurden und somit nicht auf den Main-Branch übernommen wurden. Die Teams wurde jede Woche neu zusammengesetzt, um feste Strukturen zu verhindern und die Stärken einzelner Gruppenmitglieder sinnvoll einzusetzen und immer wieder neue Perspektiven auf die Bausteine zu schaffen.

Als wichtige Maßnahme zum Aufrechterhalten unserer Strategie hat sich der Weekly Cycle erwiesen, die wöchentlichen Treffen führten zu einer kontinuierlichen Arbeit am Projekt und einem gemeinsamen aktuellen Wissensstand. Probleme mit zugewiesenen Aufgaben aus der letzten Woche wurden so zeitnah angesprochen. Ergänzend zum Weekyl Cycle ist jedoch auch der Austauch über Instant Messaging zu erwähnen. Detailfragen konnten so auch vor den wöchentlichen Terminen besprochen werden, wodurch eine Verschleppung einer Aufgabe, aufgrund von Rückfragen, in die nächste Woche verhindert wurde.

Das alleinige Abhalten eines wöchentlichen Termins garantiert keinen Erfolg. Entscheidend für uns war vielmehr, wie diese Termine gestaltet wurden. Für uns war es wichtig die Treffen so zugestalten, dass genügend Raum für alle Gruppenmitglieder besteht, Probleme und Anmerkungen offen anzusprechen. Gleichzeitig wurde zu Beginn mithilfe von Timeboxing ein fester Rahmen für jedes Treffen definiert. Die Meetings wurden von einen Person moderiert und Themen, welche nicht für alle Gruppenmitglieder von Bedeutung sind wurden außerhalb dieser Treffen besprochen. Die Treffen wurden auf 45 Minuten begrenzt, wurden nicht alle Themen in dieser Zeit behandelt werden können, wurde ein weiterer Termin geplant.

Der gesamte Code wurde mithilfe von automatisierter Unit-Tests getestet. Für jede Komponente wurde eine Test-Suite eingerichtet, welche die spezifischen Testfälle bündeln. Der nächste Schritt, die Tests mit einer CI-Pipeline automatisch durchzuführen, hat sich als für uns nicht umsetzbar erwiesen. Diese Pipeline hätte die CI/CD in unserem Projekt wesentlich verbessert, jedoch ist diese Art der Automatisierung auf Gitlab nur mit Hilfe eines externen Servers möglich, auf dem der Code deployed werden kann. Da wir aber zu keinen Zugang zu einem solchen Server hatten, fehlt diese Art der Qualitätssicherung in diesem Projekt. Trotzdem haben sich alle Gruppenmitglieder mit dieser Art der CI/CD befasst und werden diese für zukünftige Projekte anwenden. Alternativ Plattformen wie GitHub bieten diese Tools beispielsweise leichter zugänglich an, was die Einbindung in Projekte vereinfacht.

6. Fazit

6.1. Fazit zum Ergebnis

Das Resultat ist unserer Ansicht nach äußerst positiv einzuschätzen. Wir waren in der Lage, alle erforderlichen Meilensteine des Projekts termingerecht zu präsentieren und konnten schließlich eine voll funktionsfähige Software erfolgreich an unseren Kunden übergeben.

6.2. Fazit zum Prozess

Auch der gewählte Arbeitsprozess, der verschiedene Elemente aus Scrum und Kanban integrierte, war für uns als Team hilfreich und zielführend. Es hat eine gewisse Zeit gedauert, bis alle Teammitglieder mit den eingesetzten Werkzeugen auf dem gleichen Stand waren. Gegen Ende des Projekts entwickelte sich jedoch eine äußerst produktive Arbeitsumgebung, die wir gerne positiv hervorheben wollen. Außerdem empfindet das gesamte Team abschließen, dass eine steile Lernkurve zurückgelegt wurde. Arbeiten und Entwürfe, die zu Beginn des Projekts erstellt wurden, wurden gegen Ende oft aufgrund der verbesserten Kenntnisse kritisch betrachtet und hinterfragt. Dies liegt unserer Ansicht nach teilweise daran, dass dieses Projekt für die meisten von uns das erste umfangreiche Softwareprojekt darstellt und einige anfängliche Fehler gemacht wurden, die im Nachhinein überflüssig erschienen. Dies lässt sich aber auch auf den Verlauf der Vorlesungen und die dadurch erst später im Semster gewonnenen Werkzeuge und Fähigkeiten zurückführen, die vor den Lernveranstaltungen unbekannt waren. Ein Beispiel hierfür sind bestimmte Konzepte der Qualitätssicherung, wie zum Beispiel CI/CD, von denen uns erst gegen Ende des Projekts in der Vorlesung Kenntnis vermittelt wurde.

6.3. Was wir anders machen würden

Trotz des erfolgreichen Prozesses und Ergebnisses gibt es natürlich immer Verbesserungspotential, welches in zukünftigen Projekten von uns umgesetzt werden wird. Hier wäre beispielsweise das Einhalten von internen Deadlines zu nennen, welche einen zeitlichen Puffer zu den Projektdeadlines schaffen sollten. Diese wurden nicht ausreichend berücksichtigt. So gab es kurz vor den Zwischenpräsentationen noch späte Änderungen. Eine weitere Verbesserung für die zukünftige Projekte ist, dass wir aufgrund der Ankündigung von Anforderungsänderungen oft zu weit gedacht haben, obwohl diese Änderung schließlich nie stattgefunden hat. Der modulare Aufbau unseres Programms ist zwar wichtig und wurde gut umgesetzt, aber wie im Kapitel Implementierung 4 bereits erklärt, rechneten wir beispielsweise mit einer GUI, welche vom Kunden jedoch nicht gefordert wurde. Auf Erweiterbarkeit zu achten ist also wichtig, aber Zeit und Energie in potentielle Änderungen zu stecken, die dann nicht passieren, kosten Zeit und damit in der Zukunft auch immer Geld. Zuletzt ist als Verbesserungspotential zu nennen, dass wir uns in zukünftigen Projekten mehr am

Feinentwurf orientieren sollten. Hier haben einzelne Gruppenmitglieder Module implementiert und dabei um Funktionen und Attribute ergänzt, die im Feinentwurf so nicht aufgetaucht definiert waren. Sollten man diese Änderungen tätigen, muss es direkt an alle anderen Gruppenmitglieder kommuniziert werden und der Feinentwurf muss dementsprechend angepasst werden.

All diese Erkenntnisse werden wir dank der Dokumentation während des Projekts für zukünftige Softwareentwicklungen berücksichtigen und somit noch effektiver und effizienter zum gewünschten Ergebnis kommen.

Quellenverzeichnis

[1] *SOFTWARE ENGINEERING 2: Übung 4*. 5. Nov. 2023. URL: https://moodle.htw-berlin.de/pluginfile.php/1841326/mod_resource/content/3/04_%C3%9Cbung_Software_Engineering_II.pdf (besucht am 05. 11. 2023).

[2] *Mastermind. Spielanleitung*. 3. Nov. 2023. URL: https://www.bernstein-stuttgart.de/wp-content/uploads/Mastermind-PDF.pdf (besucht am 03. 11. 2023).

[3] Donald E. Knuth. *The Computer As Master Mind*. Baywood Publishing Co.,Inc., 1976. URL: https://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf (besucht am 14. 12. 2023).

A. Appendix

A.1. Weitere Use Cases

A.1.1. Hilfe aufrufen

Name	Hilfe wird angefordert
Primärakteur	Spieler
Anwendungsbereich	GUI / Spiellogik
Stakeholder	Spieler
Vorbedingung	• Das Programm wurde gestartet
Mindestzusicherung	• Spieler erhält GUI Ausgabe von verfügbaren Zugmöglichkeiten
Zusicherung im Erfolgsfall	• Spieler bekommt die Spielanleitung durch das GUI angezeigt
Haupterfolgsszenario	• Spieler ist am Zug (egal welche Rolle) • „Hilfe“ wird in Kommandozeile eingetippt und mit Enter bestätigt • Hilfemenü wird aufgerufen mit gesamtem Spielablauf den den entsprechenden möglichen Kommandos • Spieler nutzt eines der vorgeschlagenen Kommandos um valide Eingabe zu machen und nächsten Zug auszulösen
Erweiterungen	• Fehlermeldung bei Eingabe eines unbekannten Befehls

A.1.2. Spiel beenden

Name	Spiel wird vorzeitig beendet
Primärakteur	Spieler
Anwendungsbereich	GUI / Spiellogik
Stakeholder	Spieler
Vorbedingung	• Spieler hat das Spiel mit einer beliebigen Rolle gestartet
Mindestzusicherung	• Spieler erhält die Möglichkeit das laufende Spiel zu beenden
Zusicherung im Erfolgsfall	• Spiel wird beendet
Haupterfolgsszenario	• Der Spieler hat ein Spiel begonnen • Der Spieler tippt während seines Zuges „exit“ • Das Spiel wird beendet
Erweiterungen	• Fehlermeldung bei Eingabe eines unbekannten Befehls