Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Projekt: Super Superhirn

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Angewandte Informatik*

Modul: B42 Software Engineering 2
Prof.Dr. Stephan Salinger

02.01.2024

Eingereicht von:

Md Abdullah Al Mamun	582558
Maximilian Ziemann	554205
Waindja Bukam Kamseu	584779
Gizem Biliktü	584591
Linus Wittrin	584983

Gliederung

1. Einleitung

1.1 Projektbeschreibung

Herr Prof. Dr. Salinger, der Auftraggeber im Rahmen des Moduls B42 Software Engineerings II, hat der Projektgruppe, dem Auftragnehmer, den Auftrag erteilt, ein Softwareprojekt umzusetzen. Sämtliche Anforderungen stammen vom Auftraggeber. Es gab verschiedene schriftliche und mündliche Hinweise sowie die Möglichkeit, Rückfragen an den Auftraggeber zu stellen. Es sollte das Spiel Super Superhirn implementiert werden. Weitere Details zu Details des Spiels sind unter anderem im Abschnitt zur Analyse vorzufinden. Nachstehend ist die Dokumentation des Projektes dargestellt. Dabei wird insbesondere auf die für das Software Engineering typischen Projektphasen eingegangen. Diese sind: Analysephase, Designphase und Implementierungsphase. Es ist Vorgabe des Auftraggebers, dass der Qualitätssicherung während des gesamten Prozesses besonderes Gewicht beigemessen wird. Alle Qualitätssicherungsmaßnahmen sind am Ende der Dokumentation zusammenfassend dargestellt.

1.2 Projektstart

Der Projektstart erfolgte durch den Auftraggeber. Nach der Teambildung musste sich das Team gemeinsam für eine Vorgehensweise entscheiden und das gesamte Projekt planen. Nach der Vorstellung der Teammitglieder wurde geklärt, wie die Programmierkenntnisse im Team verteilt sind. Im Anschluss daran hat sich das Team Gedanken über die Arbeitsweise gemacht. Da der Auftraggeber bereits angekündigt hatte, dass mit sich ändernden Anforderungen zu rechnen ist, haben wir uns für das agile Vorgehensmodell entschieden. So kann flexibel auf auftretende Änderungen reagiert werden. Das Team einigte sich zum Projektstart auf ein Projektmanagement Tool und einen digitalen Kommunikationsweg. Das Projektmanagement Tool ist Asana, da bereits Erfahrung im Team vorlag. Hauptkommunikationsweg ist der serverbasierte Kommunikationsdienst Discord. In beiden Fällen verfügen die Teammitglieder bereits über die notwendigen Accounts und Erfahrung. Somit ist eine einfache Kollaboration möglich. Diese Formalitäten markieren den Beginn des Projekts und den Start der Analysephase. Als Qualitätsmodell haben wir und der Vorlesung im Modul entsprechend standardmäßig ISO 9126 gewählt. Anhand der ISO 9126 wird die Gesamtqualität der Software schlussendlich bewertet.

2. Analyse

2.1 Beschreibung des gewählten Analyseprozesses

Ziel des Analyseprozesses war es, die funktionalen und nicht funktionalen Anforderungen der zu entwickelnden Software genau zu bestimmen. Hierzu wurden vom Auftraggeber Informationen bereitgestellt. Zudem wurde ein Interview mit dem Auftraggeber durchgeführt. Der Grundstein für die gesamte Designphase Implementierungsphase wurde während der Analysephase gelegt. Der Analyseprozess beinhaltete zunächst eine Sichtung der vom Auftraggeber zur Verfügung gestellten Materialien. Danach traf sich das gesamte Team für eine Brainstorming Session, um offene Fragen zu klären. Das Spielen eines physischen Prototypen des Spiels Mastermind hat erste wichtige Erkenntnisse über den Spielablauf gegeben. Während der zweiten Projektwoche gab es die Möglichkeit, ein Interview mit dem Auftraggeber durchzuführen. Zur Vorbereitung wurden zentrale Interviewfragen in einem Dokument gesammelt. Die Ergebnisse des Interviews wurden notiert und sind selbstverständlich in die Anforderungserhebung eingeflossen. Zudem gab es regelmäßig Änderungen und Erweiterungen der Anforderungen, so ist beispielsweise die Netzwerkkomponente hinzugekommen. Die nachstehenden Analyseergebnisse spiegeln die endgültig vom Auftraggeber gestellten Anforderungen wider.

Die Analysephase ist im Prozess vor der Designphase und soll diese optimal vorbereiten. Als Ergebnis der Analysephase wurden die wichtigsten Nicht-Funktionalen und Funktionalen Anforderungen herausgearbeitet.

2.2 Nicht funktionale Anforderungen

Dem Auftraggeber ist Softwarequalität von besonderer Bedeutung. In Absprache mit dem Auftraggeber fokussieren wir uns deshalb bei den nicht-funktionalen Anforderungen in besonderem Maße auf die im Rahmen von ISO 9126 vorgeschlagenen Qualitätsmerkmale. Zudem hat der Auftraggeber nachstehende Qualitätsmerkmale als besonders wichtig bezeichnet:

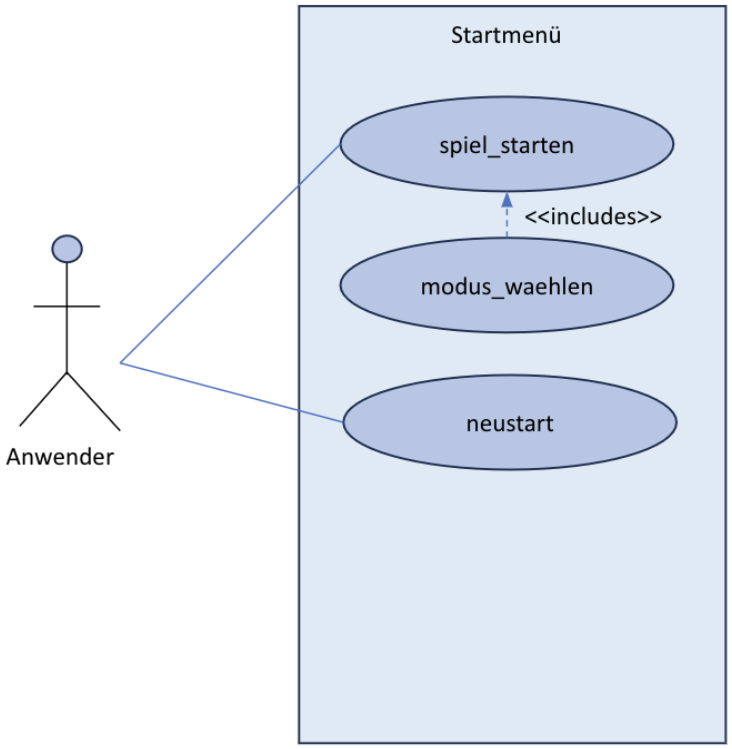
- Änderbarkeit/Wartbarkeit: Analysierbarkeit, Modifizierbarkeit, Stabilität, Testbarkeit
- Übertragbarkeit: Installierbarkeit
- Zuverlässigkeit: Reife
- Funktionalität: Angemessenheit

Die Gewährleistung dieser Qualitätsmerkmale muss insbesondere während der Designphase besonders berücksichtigt werden.

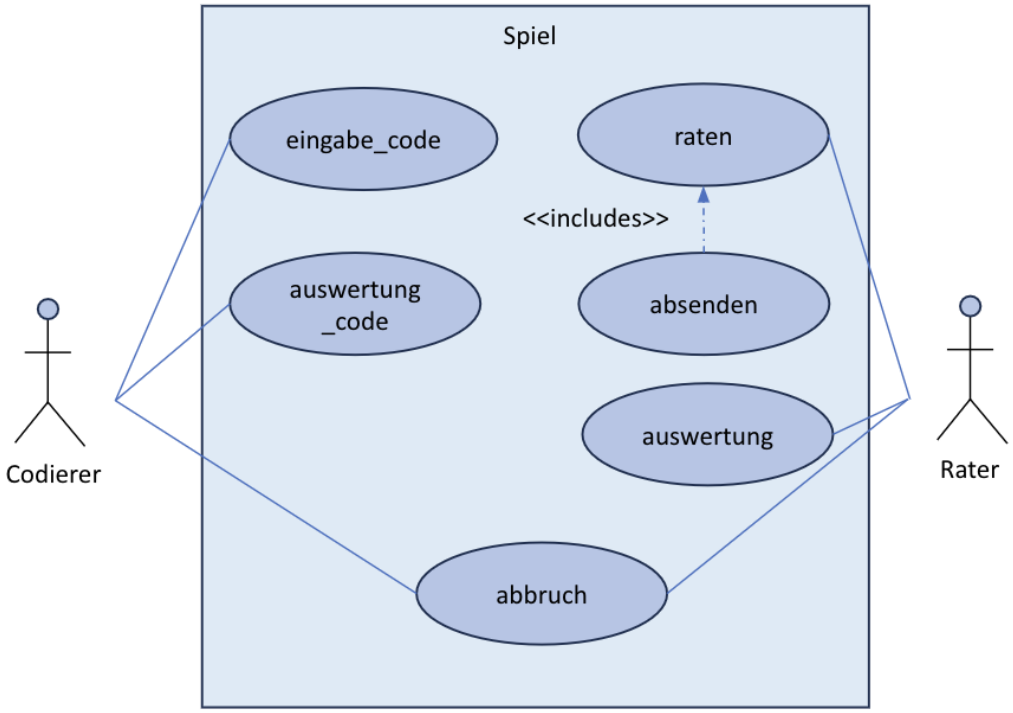
2.3 Funktionale Anforderungen

Die funktionalen Anforderungen wurden zunächst aus den gesammelten Informationen in Use Case Diagrams überführt und dann in einzelne Use Cases. Das gesamte Spiel ist in 2 Phasen unterteilt. Die Spiel Vorbereitungsphase und die Spielphase.

2.3.1 Use Case Diagrams



[A1] Phase 1: Spiel Vorbereitungsphase



[A2] Phase 2: Spielphase

2.3.2 Use Cases tabellarisch

Name	spiel_starten
Primärakteur	Anwender
Niveau	Überblick
Stakeholder und Interessen.	Der Anwender will das Spiel starten.
Vorbedingung	Anwendung gestartet
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Eine Instanz des Spiels wurde im Terminal gestartet
Haupterfolgsszenario	1.Anwender startet Spiel 2.Bereitstellung der Spielinstanz

Name	modus_waehlen
Primärakteur	Anwender
Niveau	Überblick
Stakeholder und Interessen.	Der Anwender will die Rolle im Spiel wählen.
Vorbedingung	Spielinstanz bereitgestellt.
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Das Spiel beginnt und Phase 2 wird eingeläutet.
Haupterfolgsszenario	1.Anwender wählt die gewünschte Rolle aus 2.In der Spielinstanz wird dem Anwender die Rolle Rater oder Codierer zugewiesen 3.Übergang zu Phase 2, das Spiel

Name	eingabe_code
Primärakteur	Codierer
Niveau	Überblick
Stakeholder und Interessen.	Der Codierer will den zu erratenden Zielcode platzieren
Vorbedingung	Spielinstanz bereitgestellt und Codierrolle ausgewählt
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Start des Rateprozesses
Haupterfolgsszenario	1.Codierer setzt Code 2.Vom Codierer gesetzer Code setzt die Erfolgsbedingung 3.Start des Rateprozesses

Name	auswertung
Primärakteur	Codierer
Niveau	Überblick
Stakeholder und Interessen.	Der Codierer wertet den vom Systeme eingehenden Zielcode aus
Vorbedingung	codierung_eingabe erfolgreich
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Auswertungsprozess
Haupterfolgsszenario	1.Rateversuch des Systems wird Codierer angezeigt 2.Codierer wertet den Code aus

Name	raten
Primärakteur	Rater
Niveau	Überblick
Stakeholder und Interessen.	Rater möchte einen Rateversuch erstellen
Vorbedingung	Anwender hat Modus Rater ausgewählt
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Spiel ist nicht beendet
Haupterfolgsszenario	1.Rateprozess wird gestartet 2.Rater gibt seine Kombinationen an

Name	absenden
Primärakteur	Rater
Niveau	Überblick
Stakeholder und Interessen.	Der Rater möchte den erstellten Rateversuch absenden
Vorbedingung	Der Rater hat Code erstellt
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Rateversuch wird in Spielinstanz aktualisiert
Haupterfolgsszenario	1.Rateversuch wird abgesandt 2.Rateversuch wird in Spielinstanz aktualisiert

Name	auswertung
Primärakteur	Rater
Niveau	Überblick
Stakeholder und Interessen.	Der Rater möchte wissen, ob er den korrekten Code eingegeben hat
Vorbedingung	Rater hat Code abgesandt
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Rateversuch wird ausgewertet
Haupterfolgsszenario	1.Code wird auf Korrektheit überprüft 2.Rater bekommt Auswertung in Form von schwarzen und weißen Pins ausgegeben

Name	abbruch
Primärakteur	Rater/Codierer
Niveau	Überblick
Stakeholder und Interessen.	Rater oder Codierer möchten das Spiel abbrechen
Vorbedingung	Rater oder Codierer befinden sich in einer Spielinstanz
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Das Spiel wird abgebrochen
Haupterfolgsszenario	1.Rater oder Codierer befinden sich innerhalb eines Spiels. 2.Das Spiel wird abgebrochen 3.Rater oder Codierer befinden sich wieder im Startmenü

Name	neustart
Primärakteur	Anwender
Niveau	Überblick
Stakeholder und Interessen.	Der Anwender kann eine erneute Partie spielen
Vorbedingung	ende_erfolg oder ende_bedingung erfolgreich durchlaufen
Mindestzusicherung	Fehlermeldungen
Zusicherung im Erfolgsfall.	Neustart der Anwendung
Haupterfolgsszenario	1.Der Anwender wählt Neustart aus 2.Der Anwender gelangt ins Hauptmenü

2.4 Zwischenergebnis

Die Hauptanforderungen wurden herausgearbeitet. Die Use Cases stellen eine solide Basis dar, um die Designphase zu starten. Um eine hohe Qualität sicherzustellen, wurden die Anforderungen sowohl in nicht funktionaler Hinsicht, als auch in funktionaler Hinsicht genau ermittelt.

3. Design

3.1. Einführung und Zielsetzung des Objektorientiertes Designs

Im Kontext unseres Projekts "Superhirn" steht das objektorientierte Design (OOD) im Mittelpunkt unserer Entwicklungsmethodik. Das Fundament des OOD liegt in der präzisen Modellierung des Projekts anhand von Objekten und deren Interaktionen.

Das Ziel unseres OOD-Ansatzes war es, das Spiel Superhirn in kleinere, wiederverwendbare Module zu zerlegen. Diese Modularität verschaffte uns eine erhöhte Flexibilität und erleichterte die Erweiterung des Spiels. Die objektorientierte Gestaltung verbesserte die Übersichtlichkeit des Codes und trug maßgeblich zur Gesamtqualität unseres Projekts bei. Durch die Modularisierung wird zudem Erweiterbarkeit gewährleistet.

Alle vorgestellten Diagramme wurden mit Hilfe der Online Diagram Software Drawio[1] erstellt.

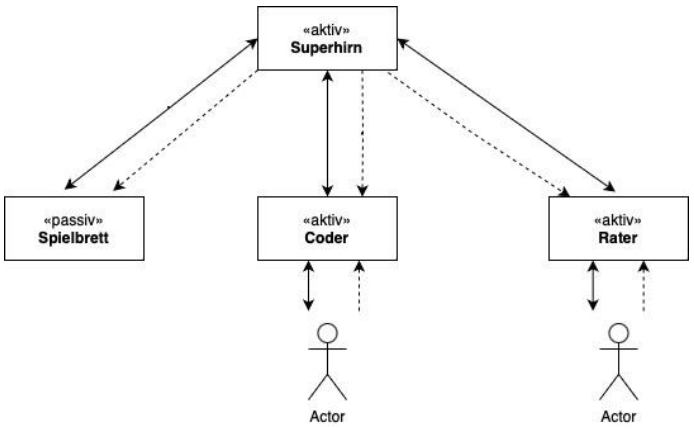
3.2 Technische und Funktionale Details

3.2.1 Architektur und -Designmuster

Ursprünglich einigten wir uns auf das Repository-System als Architekturstil. Dabei bestimmten wir die 'Superhirn'-Komponente als aktive Spiellogik, die als zentrale Steuerung fungiert und sämtliche Interaktionen der Komponenten koordiniert. Das 'Spielbrett' übernahm die Verantwortung für die Darstellung und Verwaltung des Spielfelds und agierte passiv. Sowohl der 'Coder' als auch der 'Rater' wurden als aktive Komponenten definiert, von jeweils einem Akteur bzw. Benutzer genutzt oder gesteuert. Diese Struktur schuf eine deutliche zentrale Bündelung und klare Rollenverteilung der Komponenten.

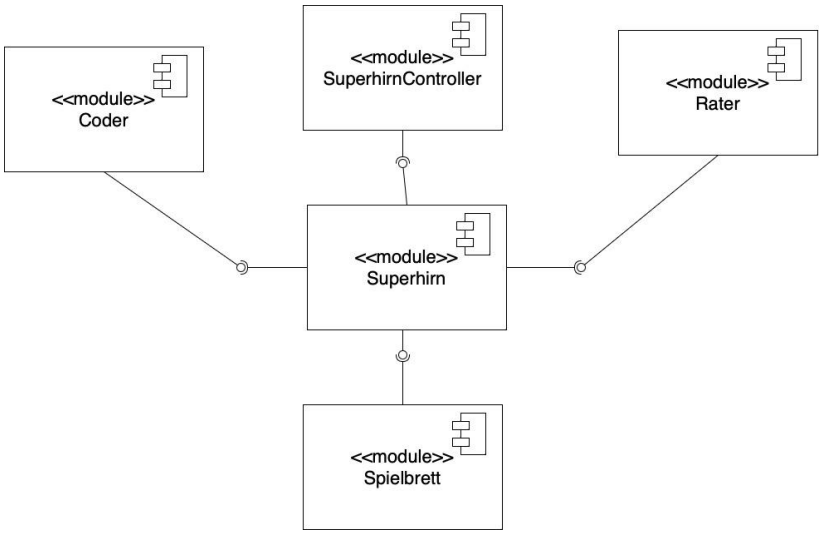
Später wurde uns jedoch mitgeteilt, dass das Repository-System nur dann sinnvoll ist, wenn eine Datenbank erforderlich ist, die in unserem Fall jedoch nicht existiert. Daher entschieden wir uns letztendlich für den komponentenbasierten Architekturstil [B1]. Die Präsentationsschicht wurde dem 'Spielbrett' zugewiesen, das als Benutzeroberfläche des Spiels fungiert. Hier kann der Benutzer Optionen auswählen und Eingaben tätigen. Die Logikschicht beinhaltet alle anderen Aspekte wie Spielregeln, die Überprüfung von Spielzügen und die Bewertung der Spieler-Aktionen, einschließlich der Überprüfung, ob der geratene Farbcode korrekt ist.

[1] https://app.diagrams.net/



[B1] komponentenbasierten Architekturstil

Hier präsentieren wir unser Moduldiagramm [B2] mit lediglich vier Modulen. Es veranschaulicht die Hauptkomponenten, ihre Interdependenzen sowie ihre jeweiligen Funktionen. Die "Superhirn"-Komponente agiert als Spiellogik und dient als Bindeglied zwischen dem Rater, dem Coder und dem Spielbrett.

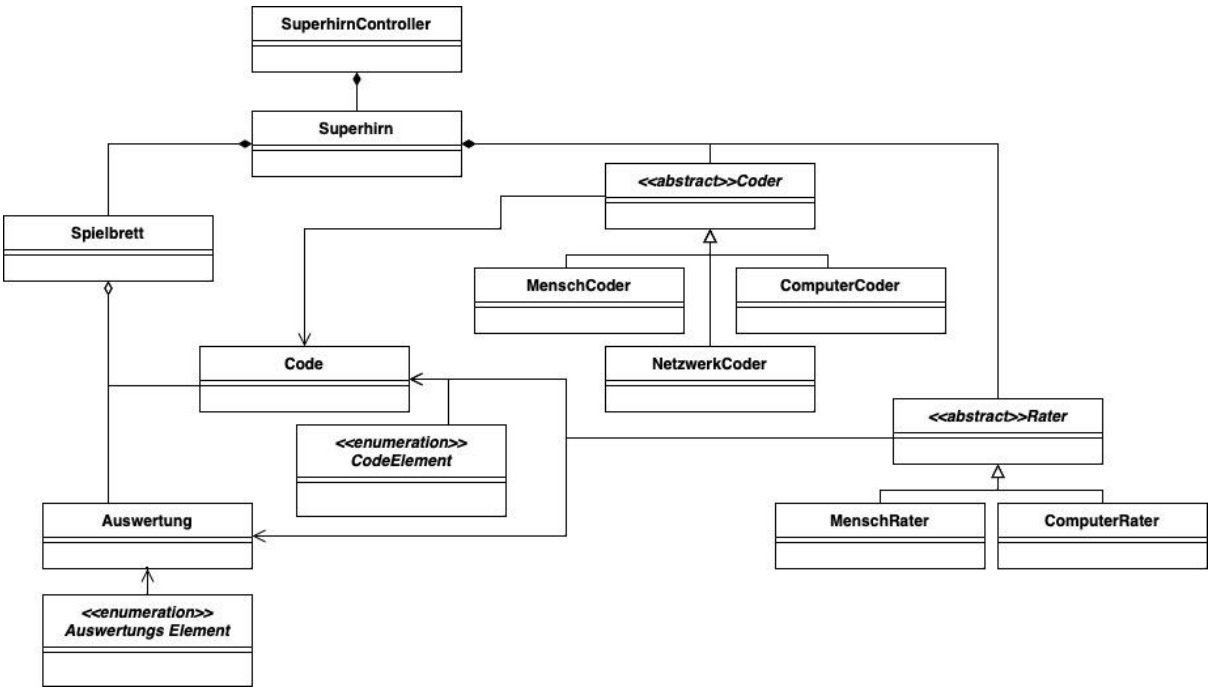


[B2] Moduldiagramm

3.2.2 Datenmodellierung

Das Klassendiagramm [B3] wurde vereinfacht, um die essentiellen Komponenten und ihre Interdependenzen klar darzustellen. Wir haben abstrakte Klassen als Schnittstelle für die Spielmodi genutzt, um konkrete Methoden zu implementieren und dadurch die Code-Wiederverwendung zu erleichtern. Dadurch entsteht Polymorphie, da derselbe Methodenname für jedoch verschiedene Implementierungen verwendet wird.

Die Auswertung und Strukturierung des Codes wurden mithilfe von Enums vereinfacht. Dies verbessert die Lesbarkeit und gewährleistet eine sichere, begrenzte Auswahl der Werte. Durch diese konstanten Werte werden Tippfehler oder fehlerhafte Eingaben vermieden. Die Superhirn-Klasse nutzt den Controller, um Aufgaben der Steuerung und Verarbeitung zu trennen.



[B3] Vereinfachtes Klassendiagramm

Dadurch wird die Testbarkeit erleichtert, da der Controller nur für die Logik zuständig ist und somit leichter isoliert und getestet werden kann. Das Spielbrett wird von der Superhirn-Klasse aufgerufen, um das Spiel zu initiieren, was zur verbesserten Strukturierung des Codes beiträgt. Diese Trennung hilft auch dabei, die Verantwortlichkeiten des Spiels von seinen Regeln zu separieren.

Um die neue Anforderung zu erfüllen, haben wir eine zusätzliche Komponente namens “NetzwerkCoder“ hinzugefügt, dessen Aufgabe es ist, über das Netz auch mit anderen Spielern spielen zu können. Diese ermöglicht es, dass Spieler über das Netzwerk mit anderen Teilnehmern spielen können. Hierbei erhält der Rater seine Auswertung für seinen Code von einem Codierer, der über das Internet verbunden ist.

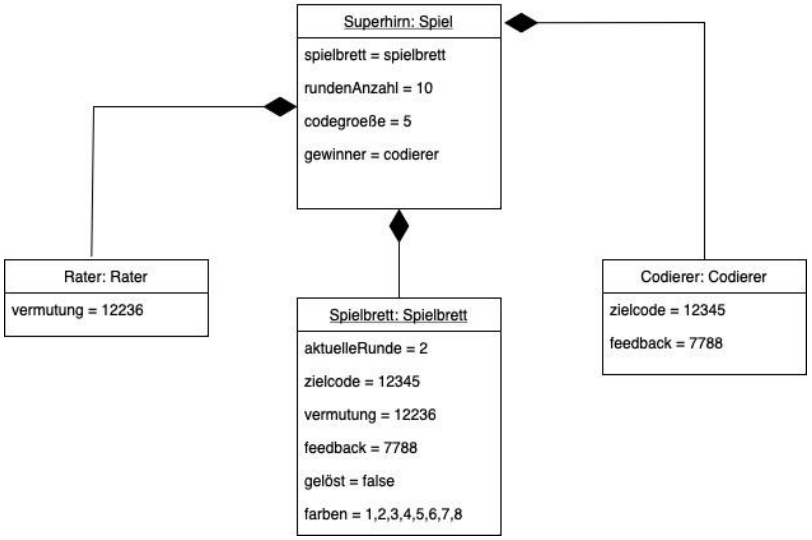
Komponente	Aufgabe
Superhirn	Gehirn des Spiel
Spielbrett	Ablauf / Ausführung des Spiels
Coder	Abstrakte Klasse mit MenschenCoder und ComputerCoder
Rater	Abstrakte Klasse mit MenschenRater und ComputerRater
Code	Enthält die Code-Elemente als eine Liste

Komponente	Aufgabe
Auswertung	Enthält die Auswertungselemente als eine Liste
CodeElement	Enum, das die Farbkodierung enthält (1=Rot, 2=Grün, 3=Gelb, 4=Blau, 5=Orange, 6=Braun, 7=Weiß)
AuswertungsElement	Enum, das die Auswertung enthält (8=Schwarz, 7=Weiß)
NetzwerkCoder	Codierer aus dem Netz wertet den Code des Raters aus

Insgesamt zeigt sich hier eine klare Aufgabentrennung, Verantwortlichkeiten und deren Abhängigkeiten, was die Struktur und Wartbarkeit des Codes verbessert.

Das Klassendiagramm im Detail mit allen jeweiligen öffentlichen Methoden ist im Anhang zu sehen. [B14]

Das Objektdiagramm bietet eine Momentaufnahme der tatsächlichen Instanzen der Klassen während der Laufzeit. Es zeigt, wie die jeweiligen Klassen und Objekte in Beziehung zueinander stehen und Daten austauschen, was eine Visualisierung der Datenstruktur des Spiels ermöglicht.

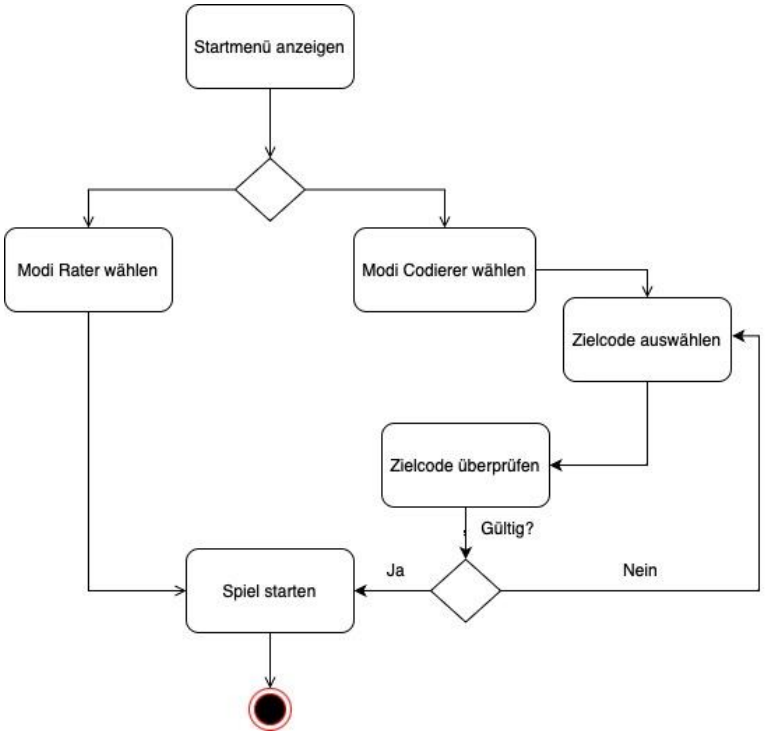


[B4] Objektdiagramm

Hier ist ein ungefährer Ausschnitt aus dem laufenden Spiel, in dem der Rater im zweiten Zug seinen Code eingibt und der Codierer diesen bewertet. Dabei sind die Codelänge und die verfügbaren Farboptionen bereits bekannt, und der Code wurde noch nicht gelöst [B4].

3.2.3 Schnittstellen und Interaktionen

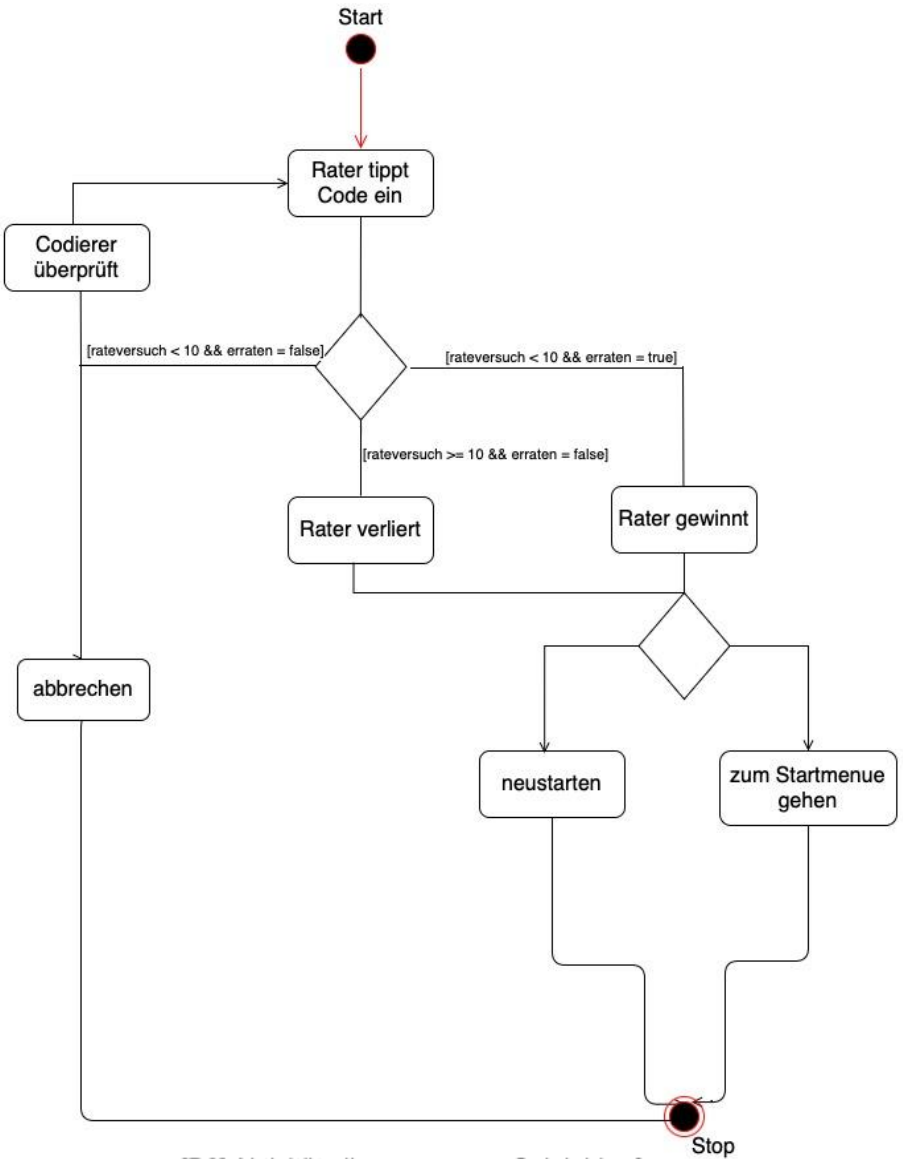
Hier präsentiert sich ein Aktivitätsdiagramm [B5], das die Spielmodi des Spiels anschaulich darstellt. Zu Beginn wird das Hauptmenü angezeigt, das dem Benutzer die Wahl der Spielmodi ermöglicht. Bei Auswahl des "Rater"-Modus startet das Spiel direkt. Im Gegensatz dazu erfordert die Auswahl des "Codierer"-Modus die Eingabe eines Zielcodes. Es sei angemerkt, dass aufgrund neuer Anforderungen seitens der Stakeholder der Zielcode eine Farbauswahl von 2-8 Farben sowie eine feste Länge von 4-5 Ziffern aufweisen muss. Sollte die Eingabe nicht diesen Kriterien entsprechen, wird der Codierer aufgefordert, einen neuen Zielcode einzugeben, bis die Voraussetzungen erfüllt sind. Erst dann kann der Codierer das Spiel beginnen.



[B5] Aktivitätsdiagramm zur Auswahl des Spielmodus

Diese Darstellung bietet nicht nur eine visuelle Übersicht der Spielmodi, sondern auch eine klare Erfassung der Anforderungen an die Benutzereingabe. Dadurch wird die Verständlichkeit des Codes erhöht und die Einhaltung der Spielregeln sichergestellt.

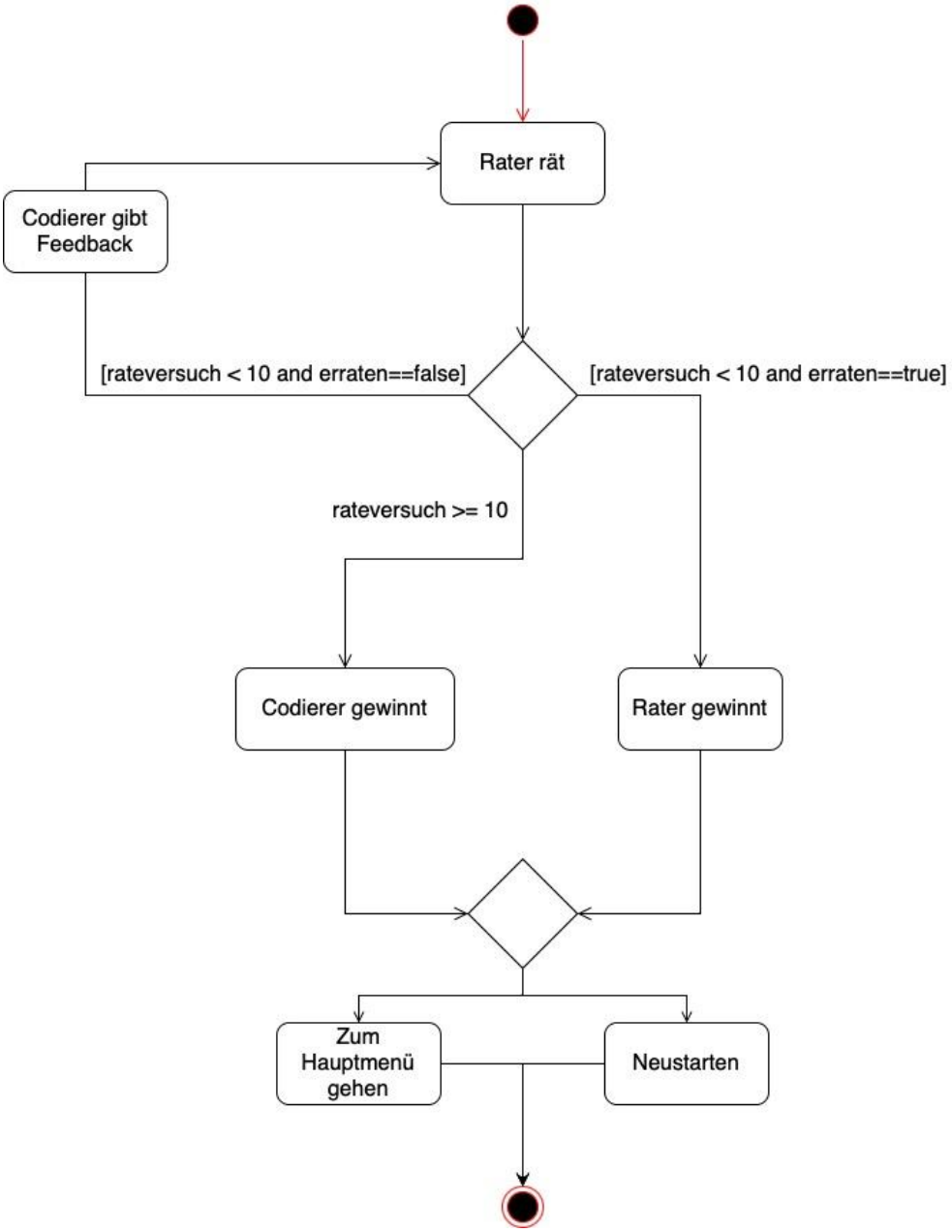
Das vorliegende Aktivitätsdiagramm [B6] veranschaulicht die Spielabläufe, wodurch komplexe Prozesse auf verständliche Weise dargestellt werden. Es verdeutlicht klare Entscheidungspunkte und Verzweigungen, sowie die Interaktionen zwischen dem Rater und dem Codierer.



[B6] Aktivitätsdiagramm zum Spielablauf

Der Rater tippt den Code ein, wodurch drei mögliche Bedingungen entstehen. Sollte der Rateversuch unter zehn liegen und der Zielcode nicht korrekt erraten worden sein, wird der Codierer aufgefordert, den Code des Raters auszuwerten. Innerhalb dieser Bedingung besteht die Möglichkeit, das Spiel jederzeit abzubrechen. Der zweite Entscheidungspunkt tritt ein, wenn der Rateversuch weniger als zehnmal erfolgt, jedoch der Code korrekt erraten wurde. In diesem Fall gewinnt der Rater. Falls der Code innerhalb der zehn Versuche nicht richtig erraten wurde, gewinnt der Codierer. Beiden Spielern wird die Option geboten, das Spiel neu zu starten oder zum Hauptmenü zurückzukehren.

Analog dazu wird im Aktivitätsdiagramm [B7] auch das Spielende des Spiels beschrieben, wo verdeutlicht wird, dass es maximal einen Gewinner gibt – entweder der Rater oder der Codierer.



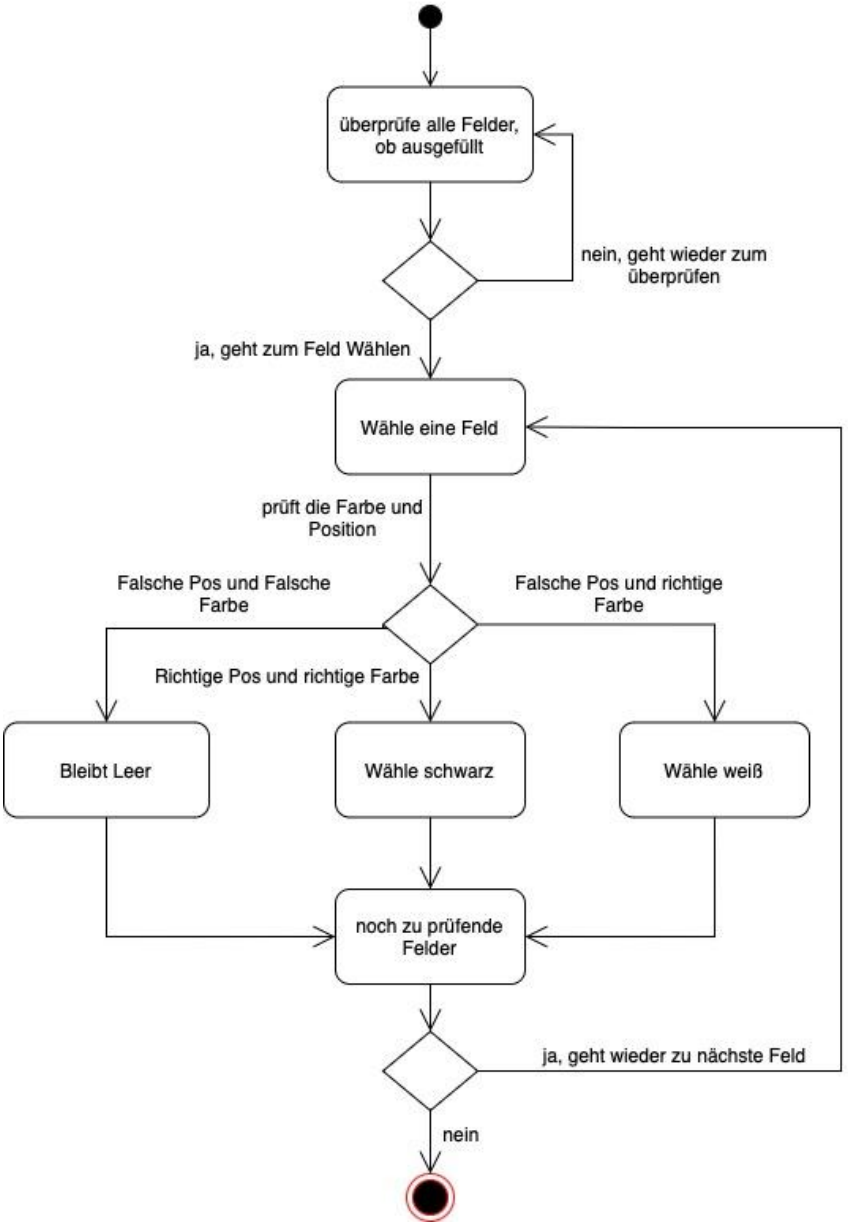
Somit wurde eine vereinfachte Visualisierung der Interaktionen zwischen den beiden Spielmodi präsentiert, was die Spiellogik und die Bedingungen für einen Spielgewinn transparent macht.

Als nächstes betrachten wir ein detailliertes Aktivitätsdiagramm [B8] unter Berücksichtigung der zuvor erwähnten Bedingungsknoten.

[B8] Aktivitätsdiagramm zu den Bedingungsknoten

Der vom Codierer eingegebene Code wird anhand zweier Parameter überprüft: der Code-Länge und der Farbauswahl. Falls diese Kriterien nicht erfüllt sind, werden entsprechende Exceptions erfasst und angezeigt. Erst nach erfolgreicher Erfüllung dieser Bedingungen wird der Code anerkannt und ein entsprechendes Objekt erstellt.

Nun kommen wir zum Auswerten des Codes anhand eines weiteren Aktivitätsdiagramms [B9]. Man beginnt mit der Überprüfung aller ausgefüllten Codefelder. Hierbei wird jedes Feld einzeln ausgewählt und anhand seiner Position und Farbe überprüft. Wenn die Farbe übereinstimmt, wird das Feld weiß markiert. Stimmen sowohl Farbe als auch Position überein, wird das Feld schwarz markiert. Wenn keine Übereinstimmung vorliegt, bleibt das Feld leer. Falls nicht alle Felder überprüft wurden, geht der Prozess zum nächsten Feld über. Sobald alle Felder geprüft sind, wird die Auswertung aufgenommen und weiterverarbeitet.

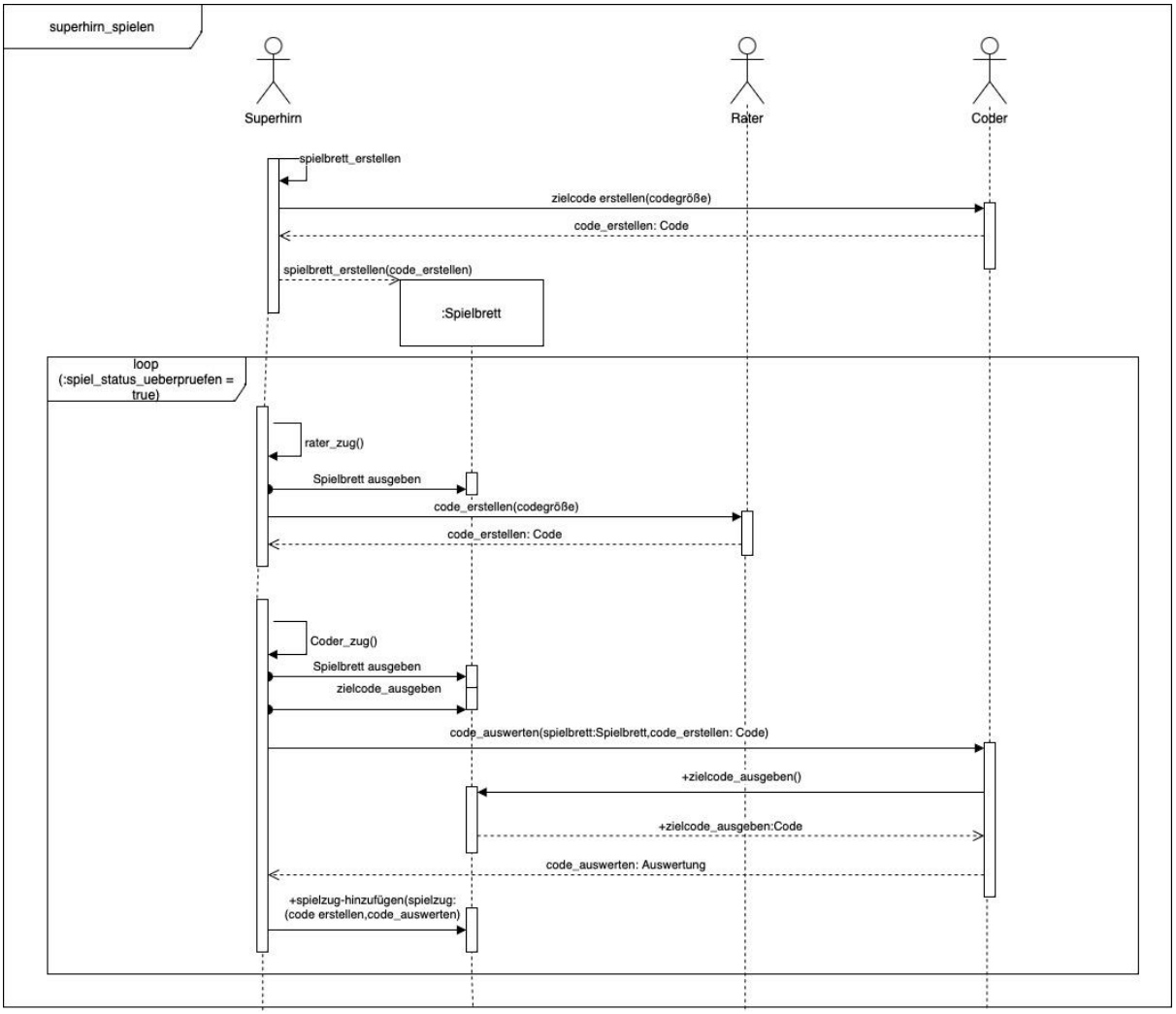


[B9] Aktivitätsdiagramm zur Codeauswertung

Sequenzdiagramme sind ein mächtiges Werkzeug, um die zeitlichen Abläufe und Interaktionen eines Systems, insbesondere in Bezug auf das Spiel mit seinen vorher definierten Bedingungen, darzustellen. Auch hier werden die Abläufe und Interaktionen deutlich und das alles in einer bestimmten Reihenfolge.

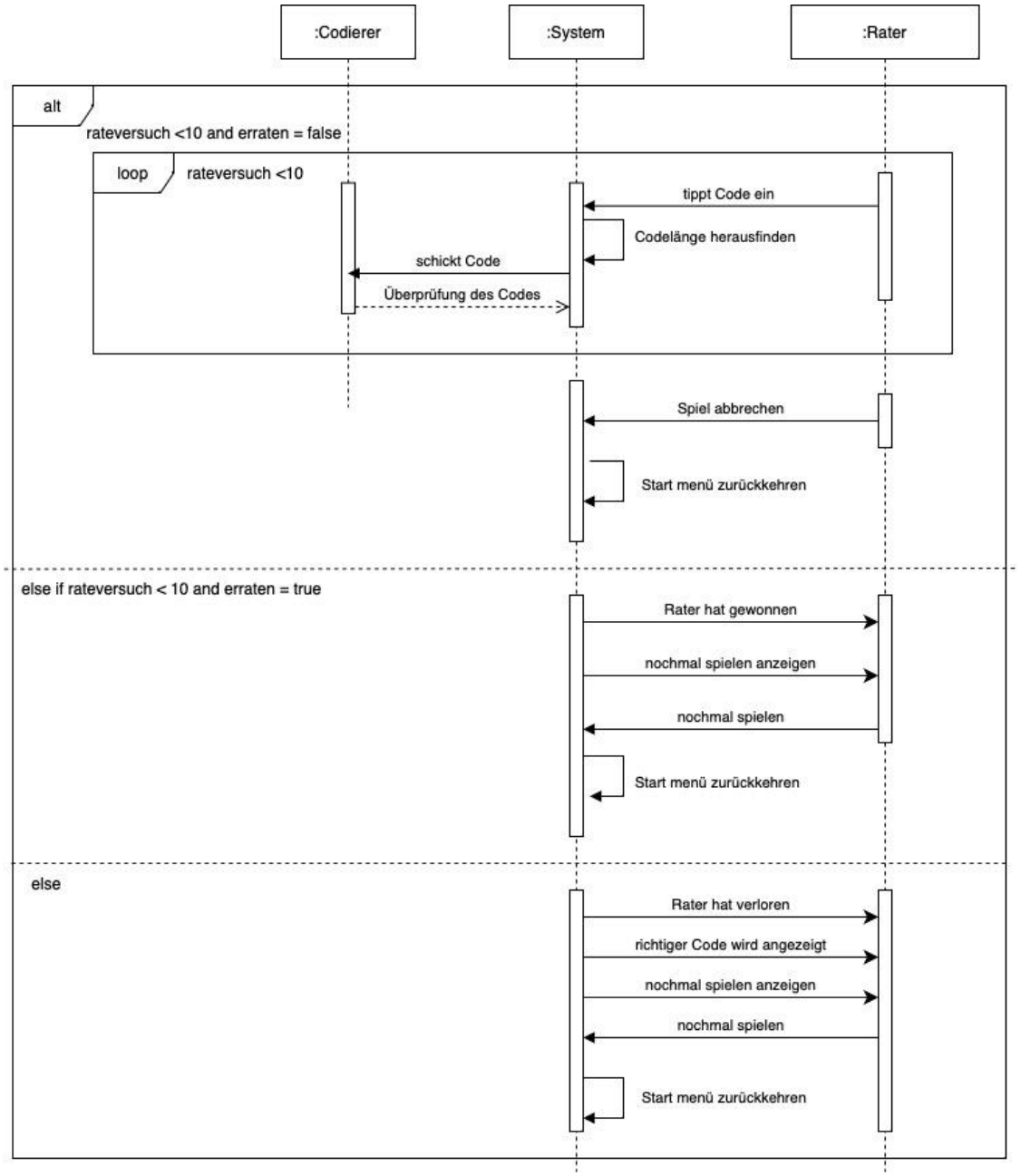
Es werden nicht nur die Reihenfolge der Nachrichtenübermittlung und Methodenaufrufe gezeigt, sondern auch die zeitliche Anordnung dieser Ereignisse. Dabei wird deutlich, wie die verschiedenen Teile des Systems miteinander agieren, und auch in welcher Abfolge diese Aktionen erfolgen.

Das Sequenzdiagramm [B10] verdeutlicht die Erstellung des Spielbretts nach der gültigen Eingabe des Zielcodes durch den Codierer. Innerhalb einer Schleife, die so lange andauert, bis der Code erraten wird, bleibt das Spielbrett funktionsfähig. Der Rater beginnt mit seinem ersten Zug und gibt seinen ersten Rateversuch ein. Anschließend ist der Codierer an der Reihe, der den Code des Raters auswertet. Die Klasse "Superhirn" ruft in jedem Zug die Ausgabe des Spielbretts auf. Nach jedem Zug wird die Rundenanzahl erhöht. Dieser Prozess wiederholt sich, bis der Gewinner ermittelt wird.



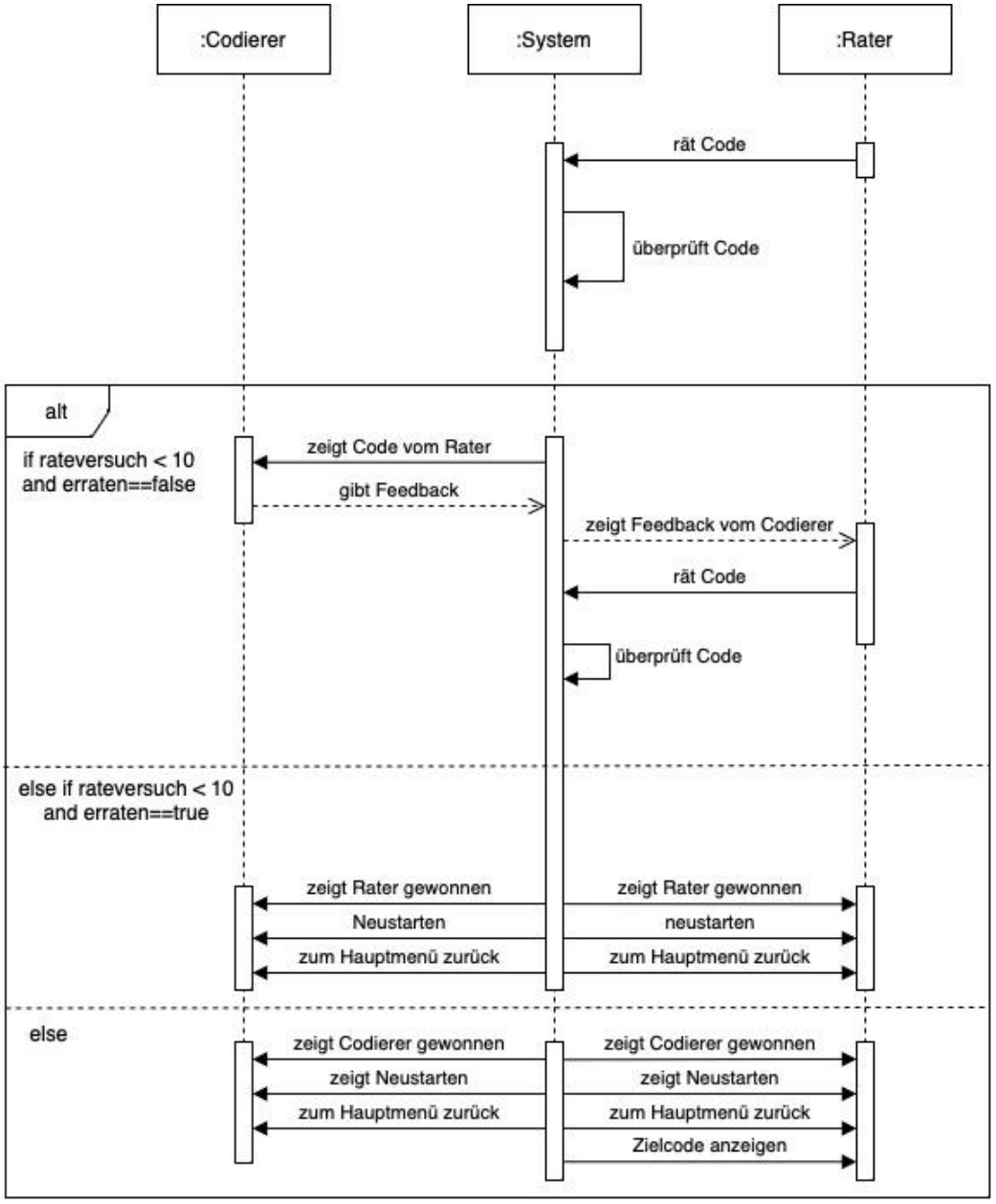
[B10] Sequenzdiagramm zum Spielbrett

Das Sequenzdiagramm [B11] illustriert die Interaktion eines einzelnen Benutzers, wobei der Fokus ausschließlich auf den Interaktionen und Handlungen des Raters liegt. Der Rater hat die Hauptaufgabe, den Zielcode zu erraten. Dabei besteht die Option, das Spiel jederzeit abzubrechen, solange der Code noch nicht korrekt erraten wurde.



[B11] Sequenzdiagramm zum Rater

In der Abbildung [B12] wird zudem auch das Spielende aus beiden Perspektiven dargestellt. Unabhängig davon, ob der Rater gewonnen oder verloren hat, besteht die Möglichkeit, das Spiel erneut zu starten oder zum Hauptmenü zurückzukehren. Gewinnt der Rater, wird dies durch eine entsprechende Erfolgsmeldung angezeigt, analog zum Codierer, wobei dem verlorenen Rater der Zielcode angezeigt wird.

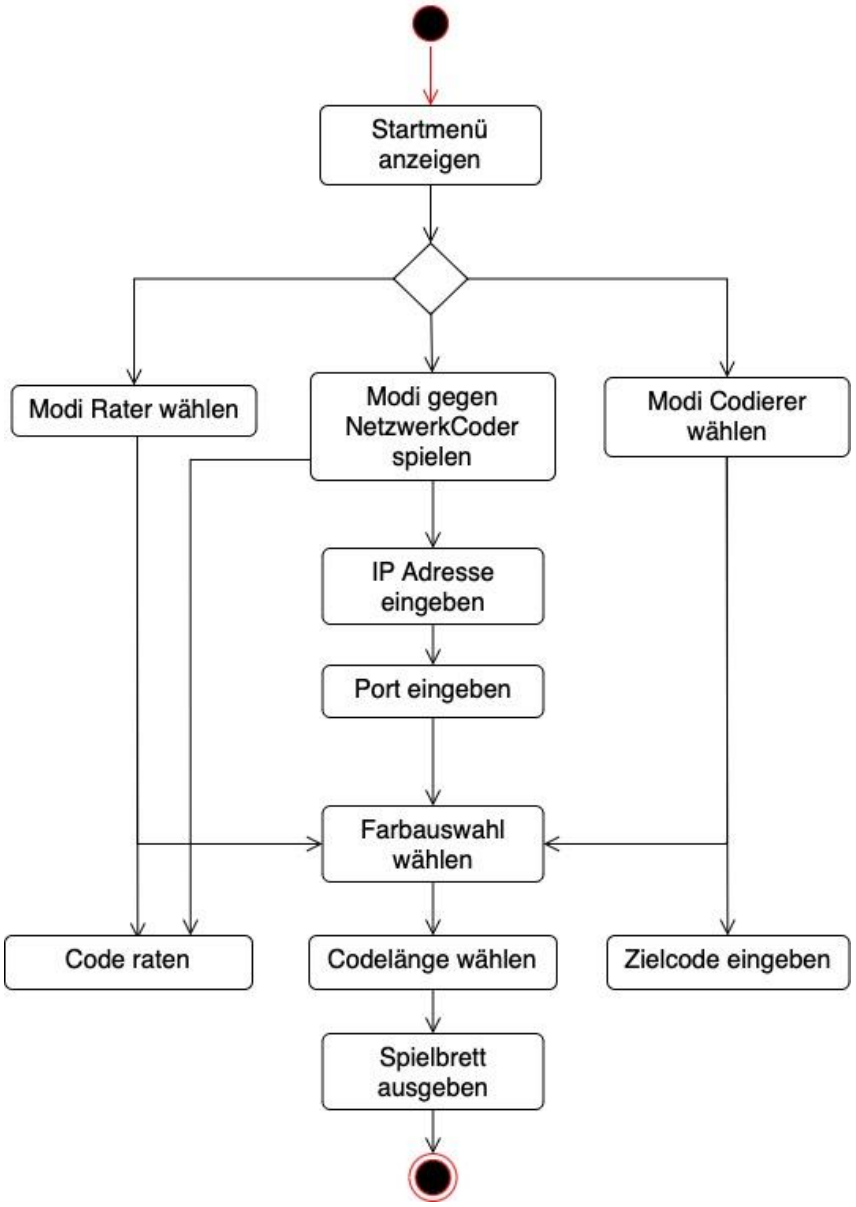


[B12] Sequenzdiagramm zum Spielende

Die Vorteile liegen auch in der Fähigkeit, Engpässe, mögliche Verzögerungen oder unerwartete Verhaltensweisen während der Interaktionen aufzudecken. Dadurch unterstützen Sequenzdiagramme nicht nur das Verständnis, sondern auch die Analyse und Verbesserung des Systemverhaltens.

Um der neuen Anforderung, gegen einen Codierer aus dem Netz spielen zu können, gerecht zu werden, haben wir ein Aktivitätsdiagramm erstellt, das den dritten Modus vorstellt [B13].

Hier sehen wir die drei möglichen Spieloptionen zur Auswahl. Wenn die Wahl auf die Option fällt, mit einem realen Spieler über das Netzwerk zu spielen, müssen zunächst die IP- und Port-Adressen eingegeben werden. Anschließend kann wie bei den anderen Modi die Farbauswahl und die Codelänge festgelegt werden. Beim Codierer wird der Zielcode festgelegt, während die Rater ihren Rate-Code eingeben.



[B13] Aktivitätsdiagramm zum NetzwerkCoder

3.3 Validieren der Anforderungen

Zusätzlich legten wir großen Wert auf Frameworks für Tests, um die Funktionalität und Robustheit unseres Systems sicherzustellen. Diese Frameworks bieten eine strukturierte Umgebung für die Testautomatisierung und ermöglichen die Implementierung der Entwurfsprinzipien.

Unsere Teststrategie umfasst verschiedene Testarten. Die Unit-Tests, durchgeführt mit 'PyTest', überprüfen isoliert jede einzelne Code-Komponente und gewährleisten deren korrekte Funktionalität. Zusätzlich verwenden wir 'unittest.mock', um Mocking und Stubbing zu ermöglichen, wodurch Abhängigkeiten abgegrenzt und getestet werden können. Integrationstests spielen ebenfalls eine wichtige Rolle, um sicherzustellen, dass die verschiedenen Systemkomponenten effektiv interagieren und nahtlos zusammenarbeiten. Dabei setzen wir auf strukturierte Ansätze und bewährte Testkonzepte wie Test-Driven-Development (TDD). Die Zuverlässigkeit und Robustheit des Systems wird durch funktionale Tests gewährleistet, welche sicherstellen, dass das System gemäß den funktionalen Anforderungen arbeitet. Nach jeder Code-Modifikation führen wir außerdem Regressionstests durch, um sicherzustellen, dass neue Implementierungen keine bestehenden Funktionen beeinträchtigen.

Die Verwendung dieser Frameworks unterstützt nicht nur die Qualitätssicherung, sondern fördert auch bewährte Entwurfspraktiken in der objektorientierten Entwicklung. Sie erleichtern die Umsetzung von Test-Driven-Development (TDD) und tragen dazu bei, die Funktionalität und Stabilität unserer Software zu gewährleisten.

4. Implementierung

In der Implementierungsphase war unser Hauptziel die Gewährleistung der Sicherheit und Robustheit des Superhirn Spiels. Zudem die Gewährleistung der nicht funktionalen Anforderungen sowie der - während der Designphase weiter definierten - funktionalen Anforderungen.

Konstruktive Qualitätssicherungsmaßnahmen kamen während des gesamten Entwicklungsprozesses zum Einsatz. Darunter Test-Driven Development (TDD), Code Reviews, und die Verwendung separater Branches pro Komponenten Bereich. Auch die Analytischen Qualitätssicherungsmaßnahmen, darunter Bottom-Up Test, automatisierte Tests und Integrationstests, wurden durchgeführt, um die korrekte Funktionalität und Zusammenarbeit der Module sicherzustellen.

Wir legten besonderen Fokus auf drei Module: **Superhirn, Superhirn Controller** und **Spielbrett**, da diese das Spiel größtenteils fungieren lassen und den Coder und den Rater zum Spiel aufrufen.

Im Folgenden werden relevante Implementierungsmerkmale der jeweiligen Module erläutert.

4.1 Modul: Superhirn Controller

Das Modul ist zuständig für die Steuerung des gesamten Spielflusses und legt einige grundlegende Faktoren für das Spiel fest. Das wären zum Beispiel Spielerrollen, Anzahl der Farben und die Codelänge.

Funktionen:

Methoden	**Funktion**
superhirn_spielen()	startet das Spiel steuert den Spielfluss
superhirn_erstellen()	erstellt ein Objekt der Klasse Superhirn durch auswahl der Spielerrollen, Anzahl der Farben und Codelänge
get_code_auswahl()	ruft die hilfsmethode helper_get_code_auswahl() wiederholend auf bei falscher eingabe
get_codelaenge()	ruft die hilfsmethode helper_get_codelaenge() wiederholend auf bei falscher eingabe
helper_getplayer_roll()	hilfsmethode um spielerrollen (Coder Rater

	zu wählen)
helper_get_code_auswahl()	hilfsmethode um auswahl der farben zu bekommen
helper_get_codelaenge()	hilfsmethode um länge des Codes als Ganzzahl (int) zu bekommen

Tests:

****helper_get_codelaenge**

Testname	**Testinhalt**
_helper_test_get_codeleange(self, input_value, expected_result):	Überprüft die Länge des Zielcodes
def test_get_codeleange_zulaessiger_input	Testet,ob die Codelänge akkurat ist
def test_get_codeleange_minimum_fail	Testet, ob die Exception ausgeworfen wird, wenn die Codelänge unter der Grenze ist
def test_get_codeleange_minimum_fail_buchstaben(self):	Testet, ob die eine Exception ausgeworfen wird wenn Buchstaben eingegeben werden
def test_get_codeleange_maximum_fail(self):	Testet ob eine Exception ausgeworfen wird wenn die Codelänge über der Grenze ist

****helper_get_code_auswahl**

Testname	**Testinhalt**
def _helper_test_get_code_auswahl(self, input_value, expected_result):	Testet, ob die Codeauswahl korrekt ausgegeben wird
def test_get_code_auswahl_zulaessiger_input	Testet, ob die Codeauswahl korrekt eingegeben wird
def test_get_code_auswahl_maximum_fail	Testet, ob eine Exception rausgeworfen wird, wenn Codeauswahl überschritten wird
def test_get_code_auswahl_maximum_fail	Testet, ob eine Exception rausgeworfen wird, wenn Codeauswahl unterschritten wird
def test_get_code_auswahl_minimum_fail_buchstaben	Testet, ob eine Exception rausgeworfen wird, wenn Codeauswahl mit Buchstaben eingegeben wird

****helper_getplayer_roll**

Testname	**Testinhalt**
test_getplayer_roll_mensch_coder	Testet, ob der User ein Codierer ist

test_getplayer_roll_computer_coder	Testet, ob der Computer Codierer ist
test_getplayer_roll_minimum_fail	Gibt eine Exception aus, wenn es keine Spieler gibt

4.2 Modul: Superhirn

Das Modul ist verantwortlich für die Umsetzung des kompletten Superhirn- Spiels und stellt die Spiellogik bereit. Das Modul kommuniziert ausgiebig mit dem Rater und den Coder Klassen sowie das Spielbrett.

Funktionen:

Methoden	**Funktion**
spielzug()	führt einen Spielzug durch
spiel_status_ueberpruefen()	überprüft Spielstatus ob ein Spieler gewonnen hat oder ob das Rundenlimit erreicht wurde
zielcode_erstellen()	erstellt ein Zielcode für das Spiel mit hilfe der code_erstellen() vom Codierer
_coder_zug()	führt den Zug des Codierers aus, aktuelle Code auswertung
_rater_zug()	führt den Zug des Raters aus, rät Code
helper_coder_zug()	hilfsmethode zur Überprüfung, ob Coder gewonnen hat, rundenlimit erreicht
helper_rater_zug()	hilfsmethode zur Überprüfung, ob Rater gewonnen hat, alle Auswertungselemete sind schwarz
spielbrett_erstellen()	eine statische Methode die die Erstellung eines leeren Spielbretts für das Superhirn Spiel macht

Tests:

Testname	**Testinhalt**
test_initialisierung()	Testet die Initialisierung der Klasse Superhirn
test_codelaenge_ueberschritten()	Testet eine Exception, wenn Codelaenge

	(größer 5, kleiner 4)
test_spielstatus_ueberpruefen()	Testet den Spielstatus
test_max_runden_erreicht()	Testet, dass Rundenanzahl erhöht wird, wenn ein Zug getätigt wurde
test_zielcode_erstellen()	Codierer erstellt ein Zielcode
test_spielbrett_erstellen()	Testet Spielbrett erstellen

4.3 Modul: Spielbrett

Dieses Modul ist in drei drei Klassen unterteilt: **Code, Auswertung** und **Spielbrett.** Die Klassen arbeiten miteinander zusammen, um die Spielzüge und Auswertungen im Superhirn-Spiel zu verwalten.

Klassen:

4.3.1 Code

Die Code Klasse zeigt den Spielzug oder den Zielcode. Sie speichert eine Liste von CodeElement(Enum) Objekten.

Funktionen:

Methoden	**Funktion**
get_code()	gibt eine Liste der CodeElemente(Enum) zurück
get_code_to_string()	gibt den Code als String zurück

Tests:

Testname	**Testinhalt**
test_get_code()	überprüft ob das Code Element richtig zurückgeliefert wird

4.3.2 Auswertung

Die Auswertung Klasse zeigt die Auswertung eines Spielzuges. Sie speichert eine Liste von AuswertungsElement Objekten.

Funktionen:

Methoden	**Funktion**
get_auswertung()	gibt eine Liste der AuswertungsElemente zurück (schwarz, weiß)

Tests:

Testname	**Testinhalt**
test_get_auswertung()	überprüft die Auswertung den Zielcodes

4.3.3 Spielbrett

Die Spielbrettklasse verwaltet die Spielzüge und die Auswertungen und enthält eine Liste von Tupeln, die jeweils Spielzug und Auswertung zeigen, sowie der Zielcode.

Funktionen:

Methoden	**Funktion**
spielbrett_ausgeben()	zeigt das Spielbrett auf der Kommandozeile in einer textuellen darstellung
spielzug_hinzufuegen()	fügt dem Spielbrett einen Spielzug hinzu
zielcode_hinzufuegen()	fügt dem Spielbrett einem Zielcode hinzu
zielcode_ausgeben()	gibt den Zielcode auf der Kommandozeile aus

Tests:

Testname	**Testinhalt**
test_zielcode_hinzufuegen()	überprüfung ob ein Zielcode richtig zum Spielbrett hinzugefügt wird

Zusätzlich zu den 3 Modulen wurden auch die ‘Coder’ und 'Rater' -Module implementiert. Die Module setzen die Logik für den menschlichen und computergesteuerten Spieler um.
Wir haben eine wichtige Erweiterung in der Coderklasse vorgenommen, um den neuen Anforderungen an Netzwerkfähigkeiten gerecht zu werden. Die Integration ermöglicht uns, das Spiel Superhirn über ein Netzwerk zu spielen.

4.4 Modul: Coder

Das Modul Coder stellt die Logik für Spieler bereit, um ihren Zielcode festzulegen und die Eingaben des Raters auszuwerten.Es umfasst eine **Abstrakte Klasse Coder** sowie spezifische Implementierungen wie **Computer Coder und Mensch Coder.** Zusätzlich gibt es eine spezielle Implementierung für die Netzwerkfähigkeit des Spiels, die durch die Klassen **SuperhirnDTO** und **NetzwerkCoder** repräsentiert sind.

Klassen:

4.4.1 SuperhirnDTO

Die Klasse zeigt ein Datenübertragungsobjekt. Sie dient dazu, Informationen zwischen dem Netzwerk Coder und dem Server zu übermitteln. Diese Klasse wandelt Daten empfangen in ein JSON Datenformat um.

Funktionen:

Methoden	**Funktion**
to_dict()	wandelt Attribute der Klasse SuperhirnDTO in ein Dictionary. Mit der Klasse SuperhirnDTO als schlüssel und den Werten

4.4.1 NetzwerkCoder

Die Klasse ist wie MenschCoder und ComputerCoder, eine Unterklasse der abtrakten Coder Klasse. Sie implementiert die Funktionalität des NetzwerkCoders für das Spiel.

Funktionen:

Methoden	**Funktion**
code_erstellen()	erstellt einen Code für den Netzwerk_coder
code_auswerten()	sendet ein POST-Request über HTTP an den Server um den Code auszuwerten
helper_sendrequest()	hilfmethode für das senden das POST Request an den Server. Umwandlung rückgabe ist text
helper_response	hilfmethode für konvertieren der Server antwort von JSON in ein SuperhirnDTO-Objekt

5. Qualitätssicherung

5.1 Allgemein

Der Auftraggeber hat zu Beginn des Projektes mitgeteilt, dass der effektive Einsatz von Methoden der konstruktiven sowie der analytischen Qualitätssicherung von besonderer Bedeutung ist. Die Wahl der Methoden der Qualitätssicherung muss in einem Qualitätssicherungskonzept dargestellt werden. Dieses Qualitätssicherungskonzept ist nachstehend vorzufinden. Das Qualitätssicherungskonzept soll die Softwarequalität maximieren. Zur Definition der Softwarequalität orientieren wir uns an Balzert: "Unter Softwarequalität versteht man die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen."[2] Dieser Definition folgend, war es von Anfang an Ziel, eine möglichst hohe Qualität zu erreichen. Im Rahmen der Qualitätssicherung wollen wir mithilfe der Prozesse sicherstellen, dass die Beschaffenheit unseres Endproduktes den Anforderungen unseres Auftraggebers genügt. Dies versuchen wir durch Methoden der Konstruktiven und Analytischen Qualitätssicherung zu erreichen. Am Anfang stand die Qualitätsplanung, in der wir die Rahmenbedingungen festgelegt haben und uns auf bestimmte Maßnahmen der Qualitätssicherung verständigt haben. Während der gesamten Planungs- und Implementierungsphase kam es zur Qualitätssteuerung, um die Konzepte der Qualitätssicherung auch tatsächlich umzusetzen. Die Qualitätssicherungsmaßnahmen wurden phasenübergreifend eingesetzt, um in jeder Phase das Endziel, eine hohe Softwarequalität, zu fördern.

5.2 Konstruktive Qualitätssicherung

Unter konstruktiver Qualitätssicherung versteht man all die Methoden und Techniken, die Qualitätsmängel von vornherein durch passendes Vorgehen vermeiden sollen. (Dirk W. Hoffmann: Software-Qualität (2008)). Diese Definition sollte als Grundlage dienen. Qualitätsmängel durch die Wahl des korrekten Vorgehens direkt vermeiden.

5.2.1 QS Prozess / QS Strategie

Der Prozess der konstruktiven Qualitätssicherung startete beim Projektstart. Er umfasst vor allem die Wahl des Vorgehensmodells. Wir haben uns für das agile Modell entschieden. Der Grund liegt darin begründet, dass immer wieder neue Anforderungen von Seiten des Auftraggebers hinzukommen können. Dadurch kann jede Iteration neu geplant werden. Zudem erleichtert die agile Vorgehensweise auch die Abstimmung innerhalb des Teams, da so nicht das gesamte Projekt geplant werden muss, sondern lediglich der nächste Schritt und der gesamte Projektablauf steht etwas im Hintergrund. Die Wahl dieses Vorgehensmodells hat sich schlussendlich als fehlerhaft herausgestellt. Gegen Ende geriet das gesamte Projekt unter Zeitdruck. Hier wäre Scrum besser geeignet gewesen, da dort

[2] Helmut Balzert: Lehrbuch der Softwaretechnik. Band 2: Softwaremanagement, Software-Qualitätssicherung, Unternehmensmodellierung.

zwar auch einzelne Sprints vorliegen. Insgesamt gibt es jedoch eine bessere Rahmenplanung mit klaren Deadlines. Neben der Wahl des Vorgehensmodells setzten wir unterschiedliche Maßnahmen der konstruktiven Qualitätssicherung ein. Die Strategie zur Auswahl der Maßnahmen beinhaltete einerseits Vorgaben des Auftraggebers, andererseits aber auch Recherche und Erfahrungen, die bereits im Team vorlagen. Als Leitlinie der konstruktiven Qualitätssicherung galt über das gesamte Projekt hinweg: Vorbeugen ist besser als Heilen.

5.2.2 QS Maßnahmen konstruktive Qualitätssicherung

Vertragsbasierte Programmierung (Design by Contract)
Wir orientierten uns an vertragsbasierter Programmierung. Nachdem innerhalb der Designphase das Objektmodell finalisiert wurde, wurden Interfaces definiert. Diese Interfaces galten als Contracts für die weitere Implementierung.

5.2.2.1 Verwaltung des Quellcodes

Zur Verwaltung des Quellcodes wurde, wie vom Auftraggeber vorgeschrieben, ein Gitlab Code Repository eingesetzt. Die Einrichtung erfolgte problemlos. Wie unter Design bereits beschrieben, gab es verschiedene Hauptklassen. Jede dieser Hauptklassen erhielt innerhalb des Repositorys eine eigene Branch. Jeder Branch wurde eine verantwortliche Person zugeteilt, die für die Implementierung wie für das Testen dieser Branch verantwortlich war. Um eine hohe Codequalität zu gewährleisten, entschieden wir uns dafür, dass für einen Commit in die Main Branch ein Review Prozess stattfinden muss. Dieser Review Prozess umfasst das Review der Implementierung und der Tests durch mindestens zwei weitere Teammitglieder. Bei erfolgreichem Review kann der Merge der Branch in die Main Branch erfolgen. Diese Maßnahme hat sich als sehr hilfreich erwiesen, da so fehlerhafte Implementierungen oder fehlerhafte Tests erkannt und verbessert worden sind, bevor sie auf die Main Branch gelangten.

5.2.2.2 Pair Programming

Eine vorgegebene Maßnahme der konstruktiven Qualitätssicherung war das Pair Programming. Diese Praxis, welche aus dem extreme Programming stammt, soll bewirken, dass Fehler direkt beim Entstehen entdeckt werden. Im Kern bedeutet Pair Programming, dass das Programmieren im Team stattfindet. Ein Driver lenkt die Programmierung und den Navigator, der den Ablauf permanent kontrolliert. Wir nutzten zur Umsetzung einerseits den Client der IDE andererseits die Streaming Möglichkeiten innerhalb von Discord um die Kommunikation während der Pair Programming Sessions zu ermöglichen. Das Pair Programming wurde so organisiert, dass sich immer ein erfahrener Python Programmierer mit einem weniger erfahrenen Python Programmierer zusammen schloss. Insgesamt hat sich der Ansatz des Pair Programmings als hilfreich erwiesen. Der Austausch während der Implementierung half über viele Verständnisschwierigkeiten hinweg. Insbesondere kam es zu einer genaueren Differenzierung der Anforderungen, die das Programm vollbringen muss. Behindert hat das Pair Programming die Fertigstellung des Endproduktes dahingehend, dass es immer zwei Personen zur Programmierung benötigte. Es kam also zu Verzögerungen aufgrund von Schwierigkeiten bei der Terminfindung. Insgesamt überwogen die Vorteile und im Qualitäts Zuwachs durch Pair Programming die Nachteile durch die geringere Implementierung Geschwindigkeit.

5.2.2.3 Informative Workspace

Eine weitere Maßnahme der konstruktiven Qualitätssicherung stellt der Informative Workspace dar. Der Informative Workspace stammt aus dem Extreme Programming und soll alle relevanten Informationen zum Projekt enthalten. Hierdurch sind alle an der Entwicklung beteiligten Personen informiert, welche Schritte als nächstes erfolgen, und wie es um das Projekt insgesamt steht. Der informative Workspace war innerhalb von Asana organisiert. Die Verantwortung wurde klar einem Teammitglied zu gewiesen, dieses hatte die Aufgabe, den Workspace während der gesamten Projektdauer aktuell zu halten. In Asana gab es einerseits ein an Kanban orientiertes Board zur Aufgabenverwaltung und andererseits auch eine detaillierte Projektübersicht mit Terminen und Verantwortlichkeiten. Insgesamt hat der Informative Workspace sehr weitergeholfen, da es eine zentrale Anlaufstelle zur Informationsbeschaffung gab. Dennoch gab es immer wieder Rückfragen zu einzelnen Themen, obwohl diese eigentlich innerhalb des Workspaces erschöpfend behandelt worden sind. In zukünftigen Projekten müsste vorab klarer kommuniziert werden, wie die Kommunikation sowie die Informationsbeschaffung abläuft.

5.2.2.4 Softwarerichtlinien

Um hohe Softwarequalität mittels konstruktiver Qualitätssicherung zu gewährleisten, kamen Softwarerichtlinien zum Einsatz. Diese beinhalten hauptsächlich Namenskonventionen sowie Vorgaben zur Code Kommentierung. Namen von Funktionen, Klassen oder Variablen sollen aufzeigen, welche Funktion sie erfüllen. Es müssen eindeutige Namen verwendet werden, um Verwechslungen und Verständnisprobleme von vornherein zu vermeiden. Die Kommentierung erfolgt in Deutsch. Darüber hinaus, und auch um weitere Details festzulegen, wurde Python Standard PEP 8 gefolgt. Die Richtlinien zu Namenskonventionen sowie zur Code Kommentierung sind innerhalb des Informative Workspace niedergeschrieben und somit für jedes Teammitglied immer sichtbar und zugänglich während des gesamten Projektes. Eine weitere Software Richtlinie im Bereich der konstruktiven Qualitätssicherung stellt die Modularisierung dar. Unser Ziel war es, eine möglichst hohe Kohäsion und geringe Kopplung zu erreichen. Dies gelang uns weitestgehend. Das zeigt sich dadurch, dass die einzelnen Branchen eigenständig entwickelt werden konnten und wenig Interdependenzen zwischen den einzelnen Modulen vorlagen. Mittels Modularisierung der Klassen wollte das Team vor allem einen "Separation of concerns” erreichen. Jeder der Module soll sich also mit einem separaten Problem befassen. Dies ist im Abschnitt Design umfassend dargestellt.

5.3 Analytische Qualitätssicherung

Im Rahmen der analytischen Qualitätssicherung ging es darum, konkrete Qualitätsmängel aufzudecken und nachzubessern.

5.3.1 QS Prozess / QS Strategie

Die Qualitätssicherungsstrategie im Rahmen der analytischen Qualitätssicherung sah vor, Maßnahmen der analytischen Qualitätssicherung als Bestandteil aller Aktivitäten vorzunehmen. Der Prozess beinhaltete die Planung aller Maßnahmen der analytischen Qualitätssicherung zu Beginn einer neuen Aktivität. Die analytische Qualitätssicherung innerhalb der ersten beiden Phasen, also der Analyse und dem Design, war vor allem auf das Feedback des Auftraggebers angewiesen. Er teilte am Ende jedes Abschnitts mit, ob er mit der Arbeit zufrieden war und was noch fehlte. So wurde beispielsweise am Ende der Designphase ein grober Fehler im Architekturstil vom Auftraggeber aufgedeckt. Dies hat uns sehr geholfen, unseren Architekturentwurf nochmals zu überdenken.

5.3.2 QS Maßnahmen

5.3.2.1 statische Maßnahmen

In statischer Hinsicht wurden Durchsichten und Inspektionen durchgeführt, die am Ende einer jeden Phase durchgeführt wurden. Während der Implementierungsphase, wurden diese Durchsichten und Inspektionen vor jedem Commit in die Main Branch unserer Code Basis durchgeführt.

5.3.2.2 Dynamische Maßnahmen

In dynamischer Hinsicht entwickelten wir bereits während der Designphase ein Testkonzept für unser System. Unser Testkonzept beinhaltet hauptsächlich Unit Tests, um die einzelnen Komponenten jeweils auf ihre Funktionalitäten zu testen. Zudem wurde das gesamte System in manueller Weise einem Systemtest unterzogen. Dabei wurde überprüft, ob das System die einzelnen Anforderungen erfüllt. Die Basis für diese Anforderungen stellten die Use Cases dar. Wenn ein Test einen Use Case erfüllte, dann galt dieser als bestanden. Unsere Test Strategie folgte einem Bottom-Up Ansatz. Jede Methode und jede Komponente sollen einzeln getestet werden. Wenn Komponenten zusammengeführt werden, folgen Regressionstests, um zu erkennen, ob es zu Fehlern beim Zusammenführen der einzelnen Komponenten kommt. Dieses Testkonzept konnte vom Team nur bedingt vollständig umgesetzt werden. Es gibt Codeanteile, die nicht von Tests abgedeckt sind. Somit bestand zwar ein taugliches Testkonzept und damit auch ein taugliches Konzept der analytischen Qualitätssicherung, dieses wurde jedoch nicht komplett umgesetzt.

6. Fazit

6.1 Beurteilung des Ergebnisses

Während der Designphase wählte das Team einen grundlegend falschen Architekturstil. Dies führte dazu, dass der gesamte Designprozess noch einmal überdacht werden musste. Das agile Vorgehensmodell erlaubte jedoch eine Korrektur. Somit konnte diese Fehlentwicklung am Ende der Analysephase rasch korrigiert werden. Ansonsten wurden alle Hauptanforderungen erkannt und implementiert. Lediglich die Netzwerkkomponente wurde nicht ausgiebig genug getestet.

6.2 Beurteilung des Prozesses

Die Beurteilung des Prozesses erfolgt auf verschiedenen Ebenen, darunter die Zusammenarbeit über alle Phasen hinweg. Die Zusammenarbeit war größtenteils positiv, jedoch traten Probleme bei der Terminfindung für die Pair Programming Sessions auf. Dies führte regelmäßig zu Verzögerungen im Projektverlauf. Das gewählte Vorgehensmodell Agile erwies sich über große Teile des Projektes hinweg als tauglich. Lediglich die übergreifende Planung und die Betrachtung der Bearbeitungszeit waren nicht optimal. Hier wäre, wie oben bereits beschrieben, ein anderes Planungskonzept wie beispielsweise Scrum sehr wahrscheinlich zielführender gewesen. Den Prozess der konstruktiven Qualitätssicherung bewerten wir positiv, da die anfangs geplanten Maßnahmen alle umgesetzt wurden und auch zu einer erhöhten Qualität geführt haben. Den Prozess der analytischen Qualitätssicherung muss man neutral bewerten, da die Anwendung nicht in dem von uns anfangs angepeilten Maß getestet worden ist.

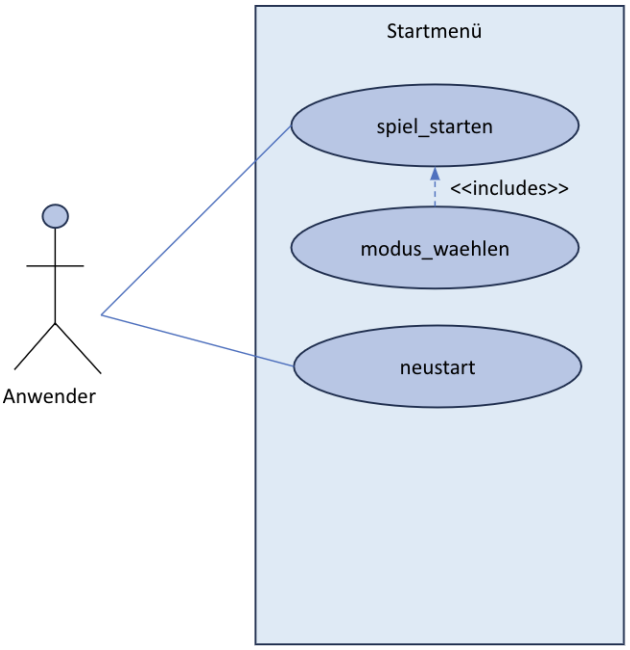
6.3 Raum für Verbesserungen

Es gibt deutlichen Verbesserungsbedarf. Insbesondere das Zeitmanagement des gesamten Projektteams war nicht optimal. In Zukunft müssen hier zu Projektstart strikte Deadlines festgelegt werden, um Verzögerungen zu vermeiden. Dies führt zum zweiten Kritikpunkt: Umfassendere Planung. Eine umfassendere Planung hätte die zeitlichen Engpässe verhindert und zu einer deutlichen Steigerung der Qualität geführt, da somit mehr Zeit für die analytische Qualitätssicherung in Form von Tests vorliegen würde. Zudem erkennen wir jetzt, dass ein früherer Start der Implementierung durchaus ratsam gewesen wäre. Zum Präsentationstermin der Implementierung hätte bereits ein funktionsfähiger Prototyp vorliegen müssen. Gegen Ende konnte der Test Driven Development Ansatz nicht durchgehend eingehalten werden, da nicht mehr genug Zeit war, um alle Methoden und Funktionalitäten unserem Testkonzept gemäß zu testen. Insgesamt wäre also die Unterteilung in kleinere Arbeitspakete hilfreich gewesen.

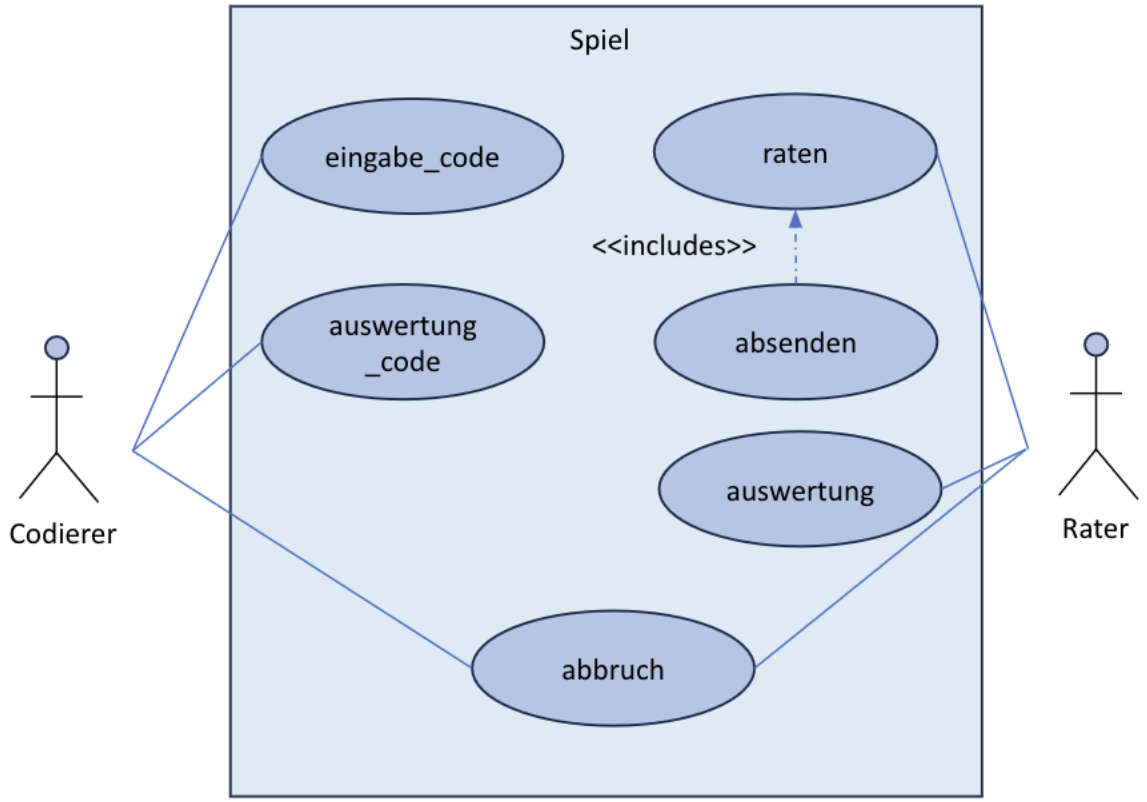
Anhang

Abbildungsverzeichnis

[A] Use cases

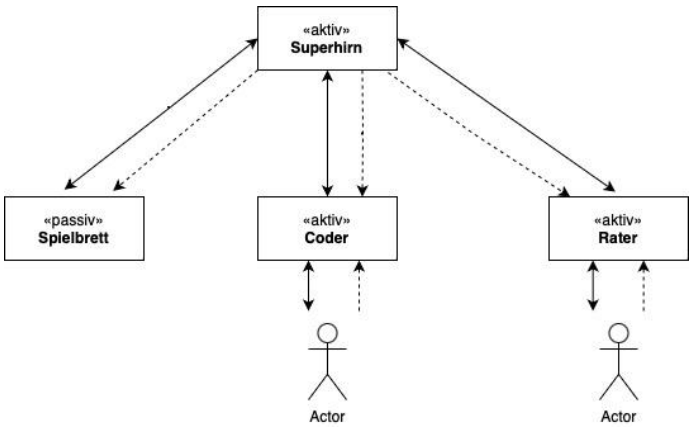


[A1] Phase 1: Spiel Vorbereitungsphase

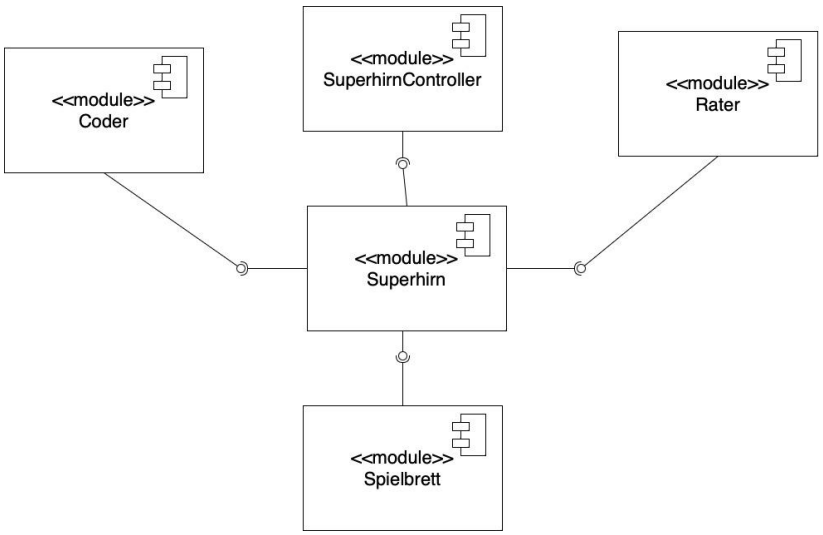


[A2] Phase 2: Spielphase

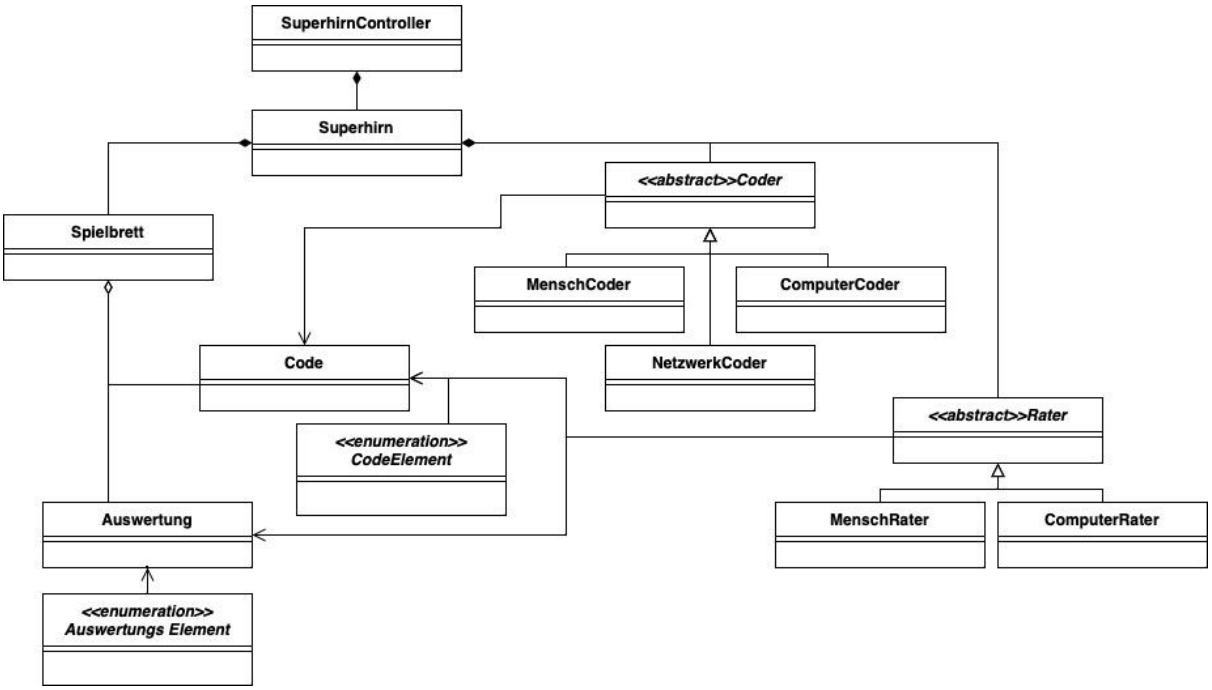
[B] Diagramme



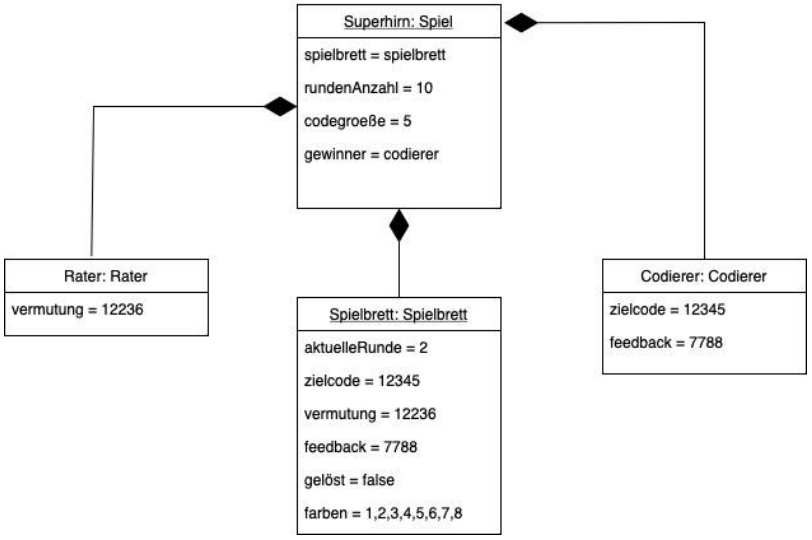
[B1] komponentenbasierter Architekturstil



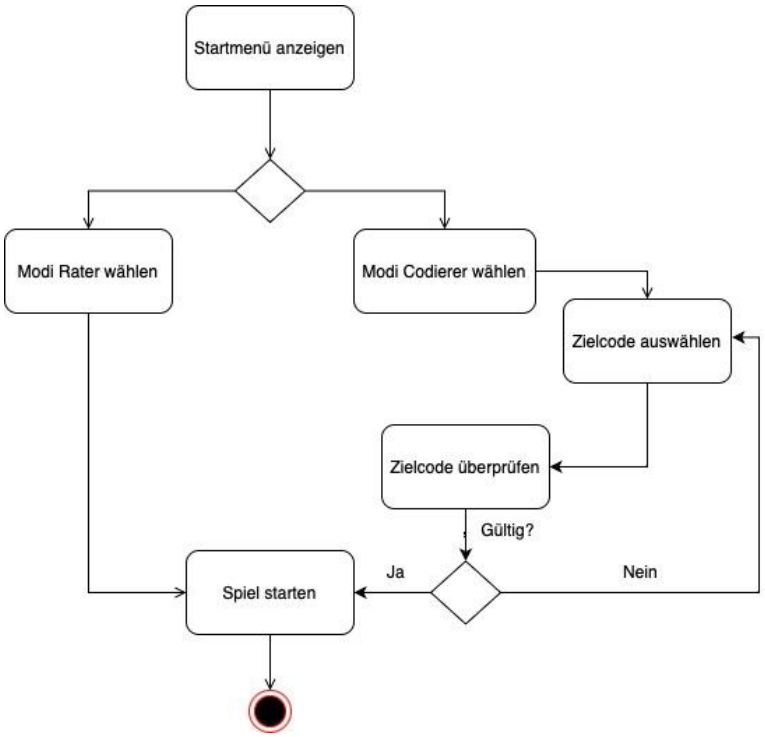
[B2] Moduldiagramm



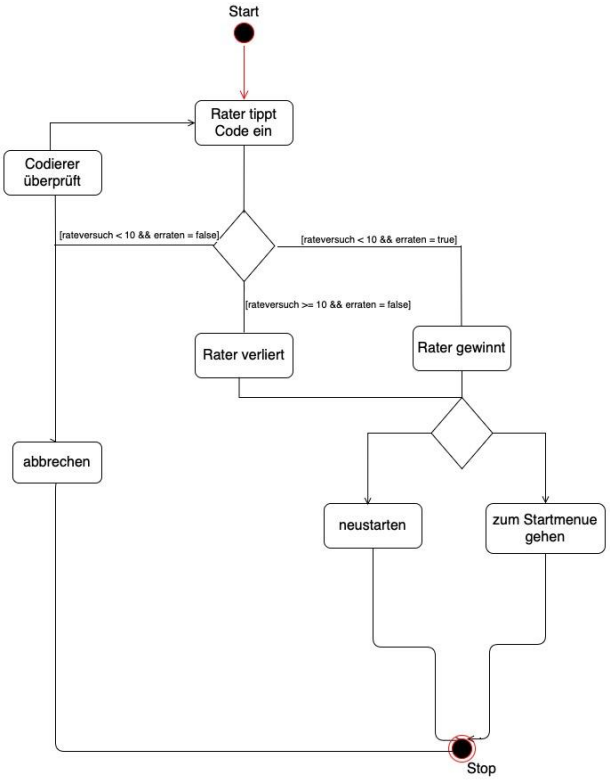
[B3] Vereinfachtes Klassendiagramm



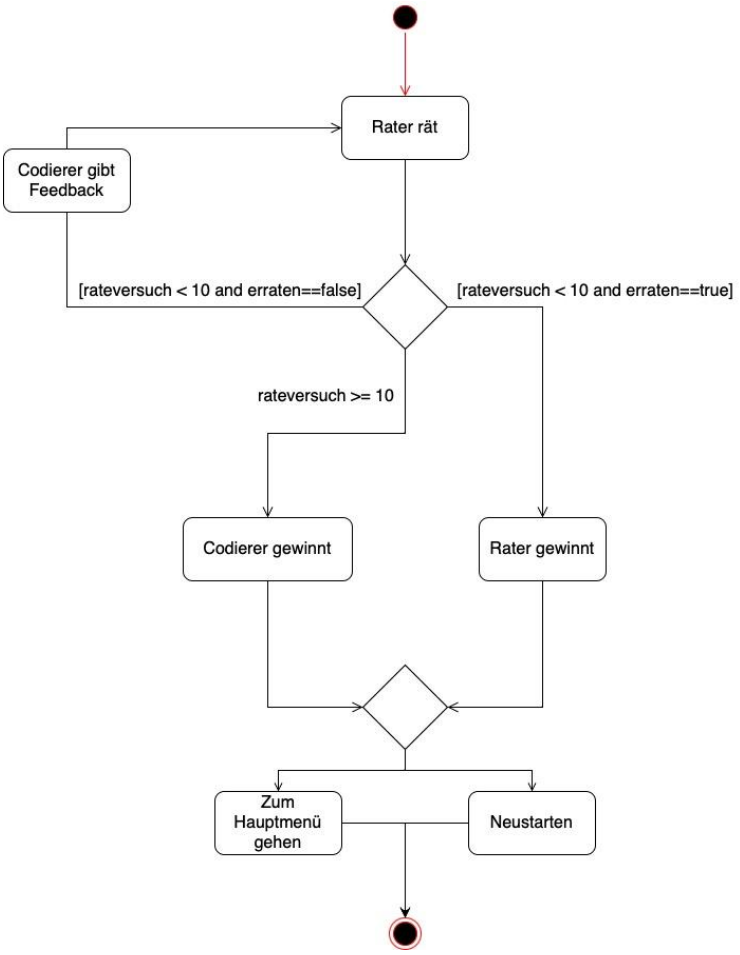
[B4] Objektdiagramm



[B5] Aktivitätsdiagramm zur Auswahl des Spielmodus

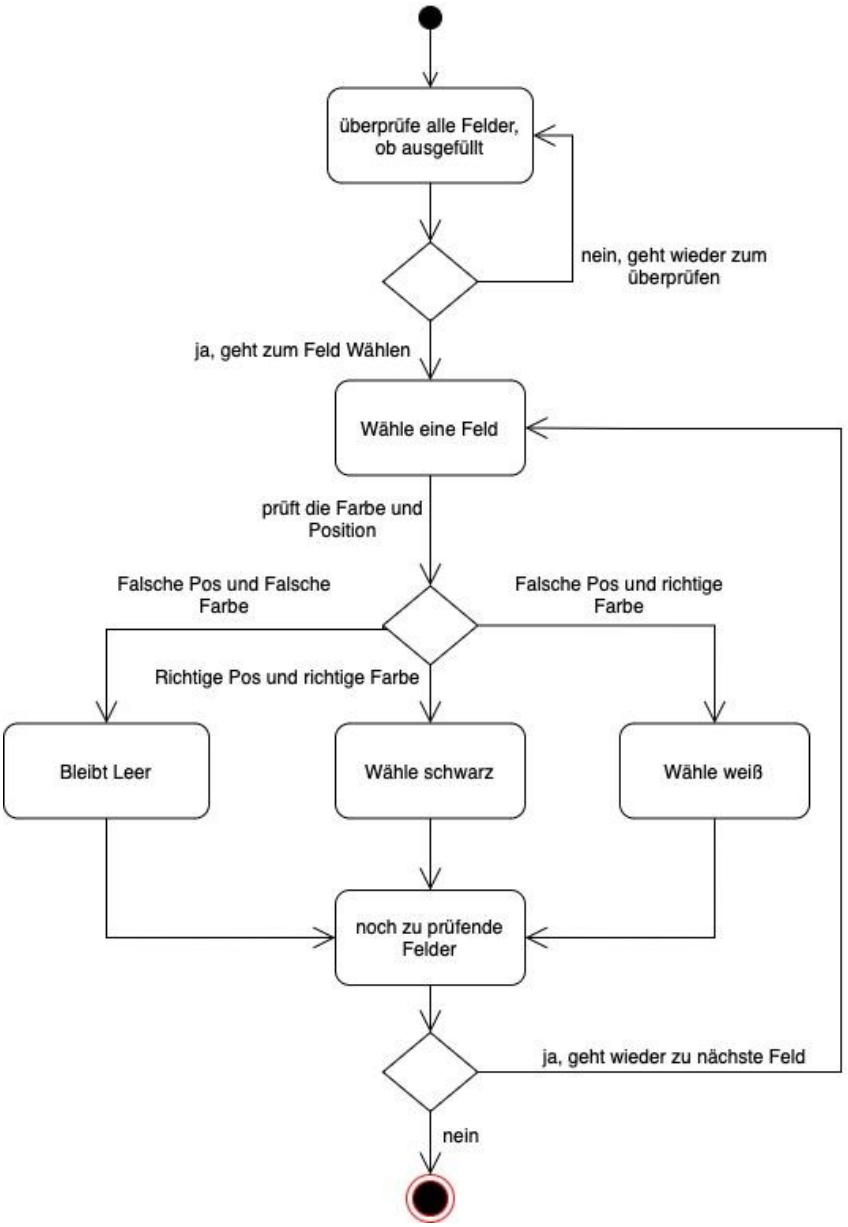


[B6] Aktivitätsdiagramm zum Spielablauf

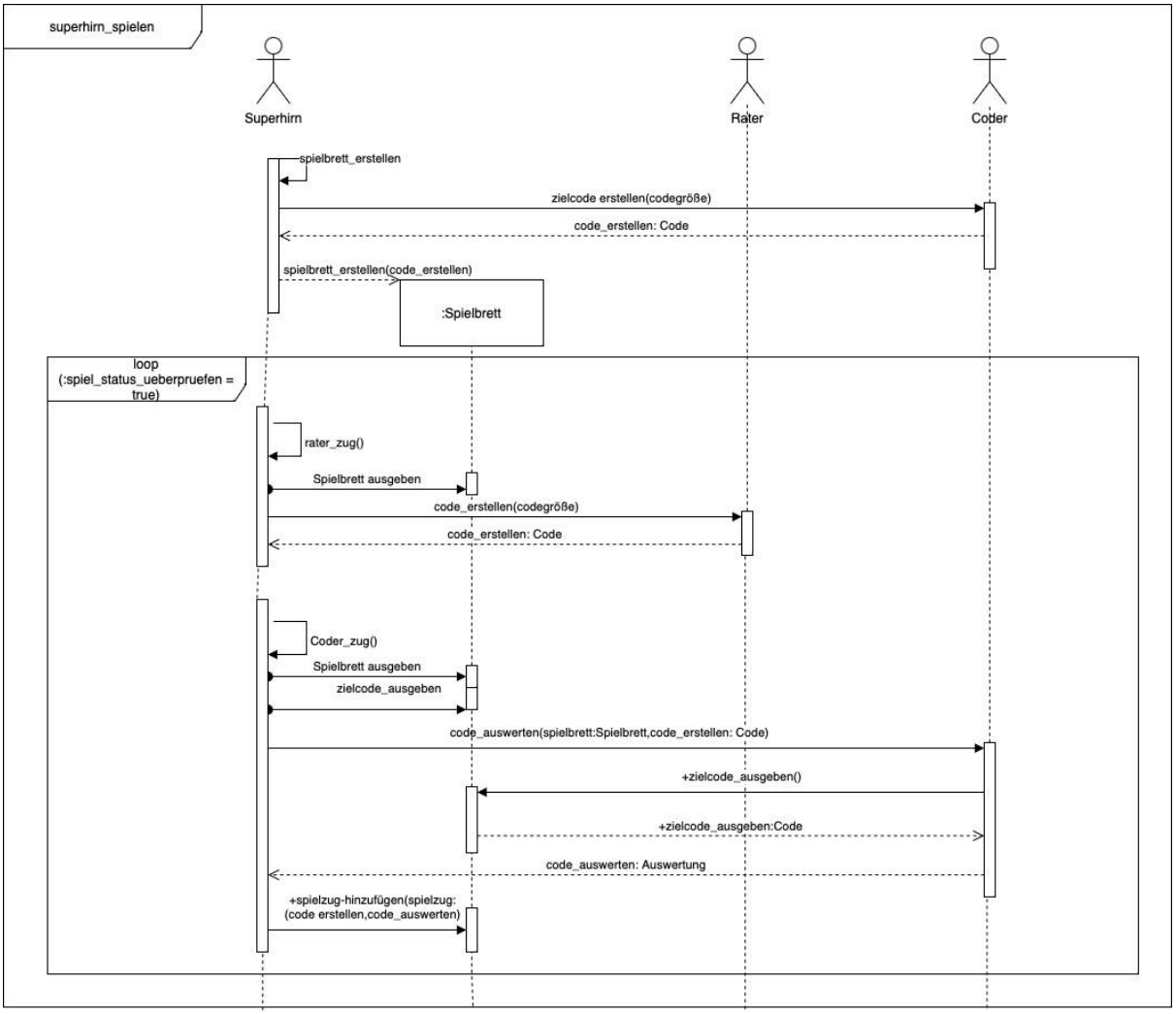


[B7] Aktivitätsdiagramm zum Spielende

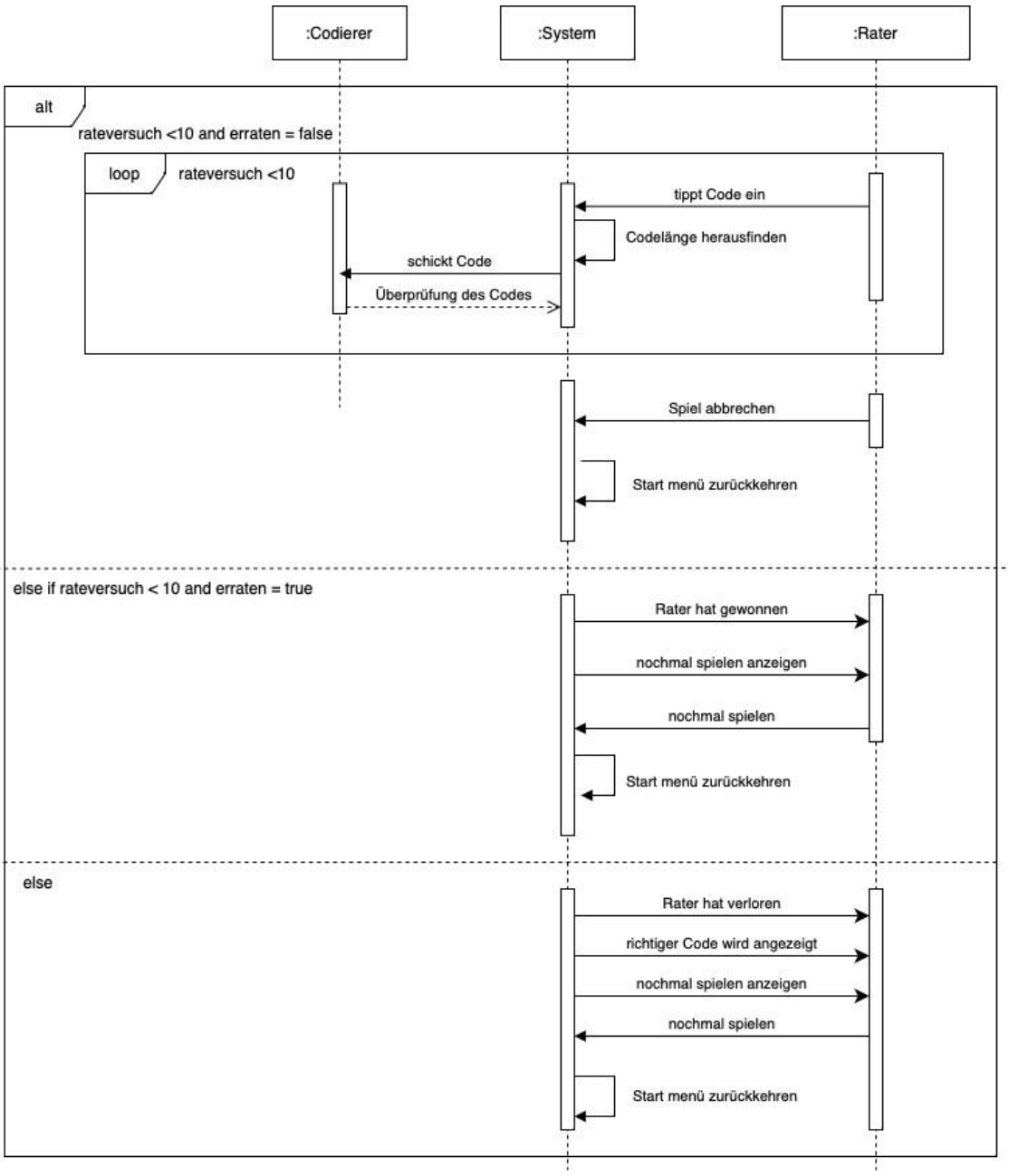
[B8] Aktivitätsdiagramm zu den Bedingungsknoten



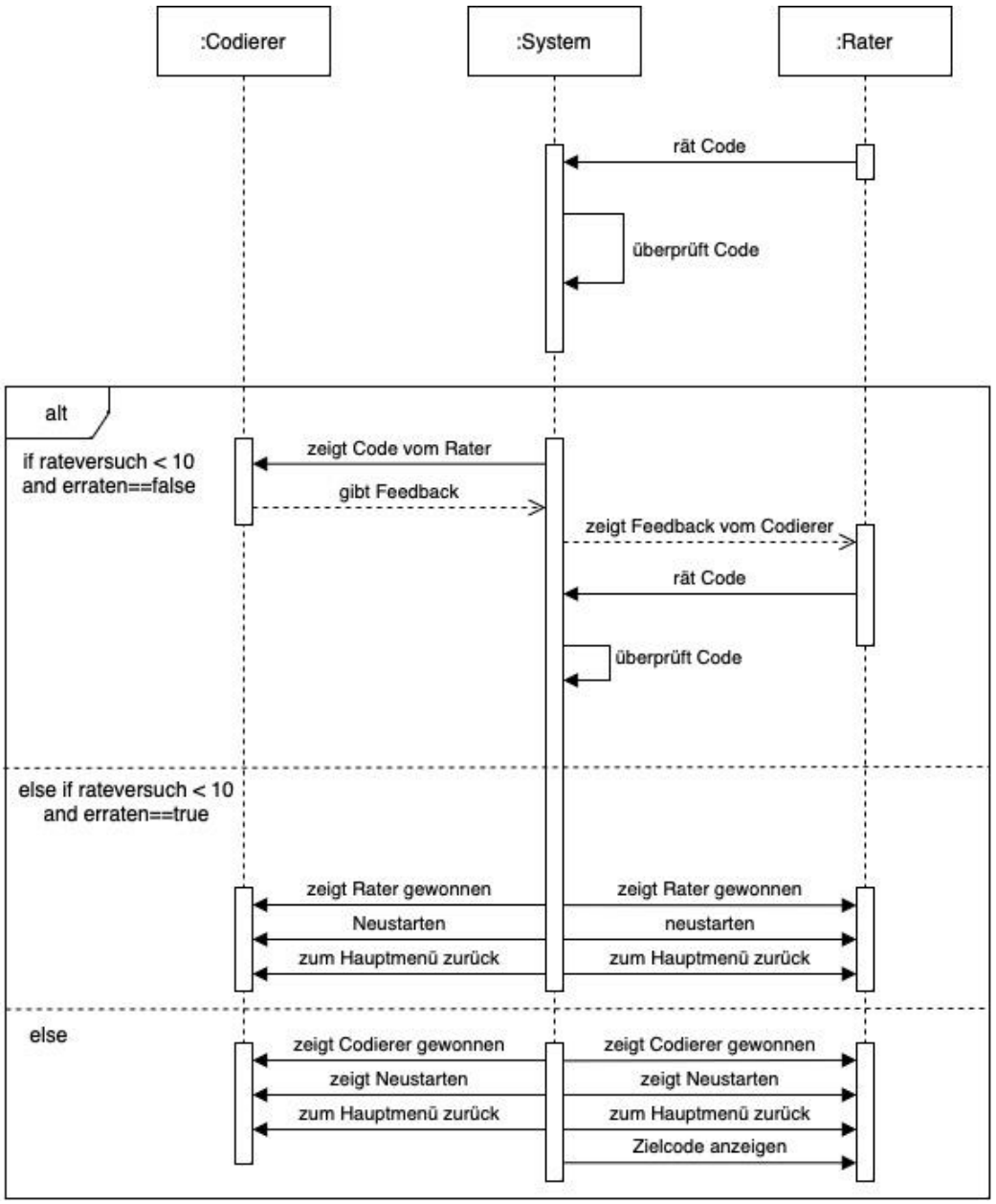
[B9] Aktivitätsdiagramm zur Codeauswertung



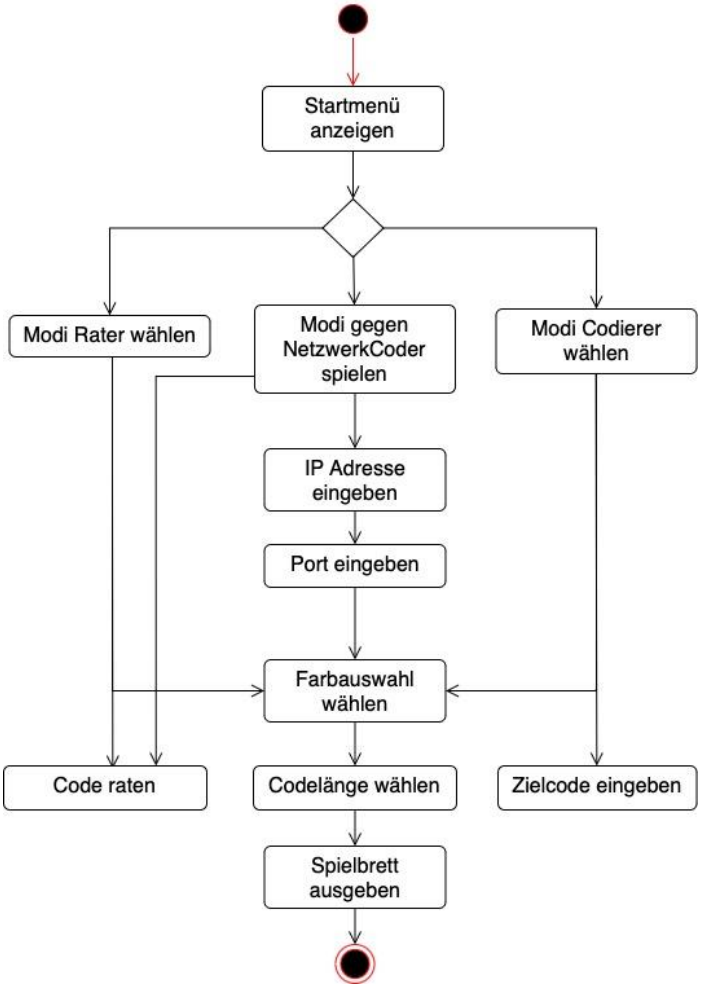
[B10] Sequenzdiagramm zum Spielbrett



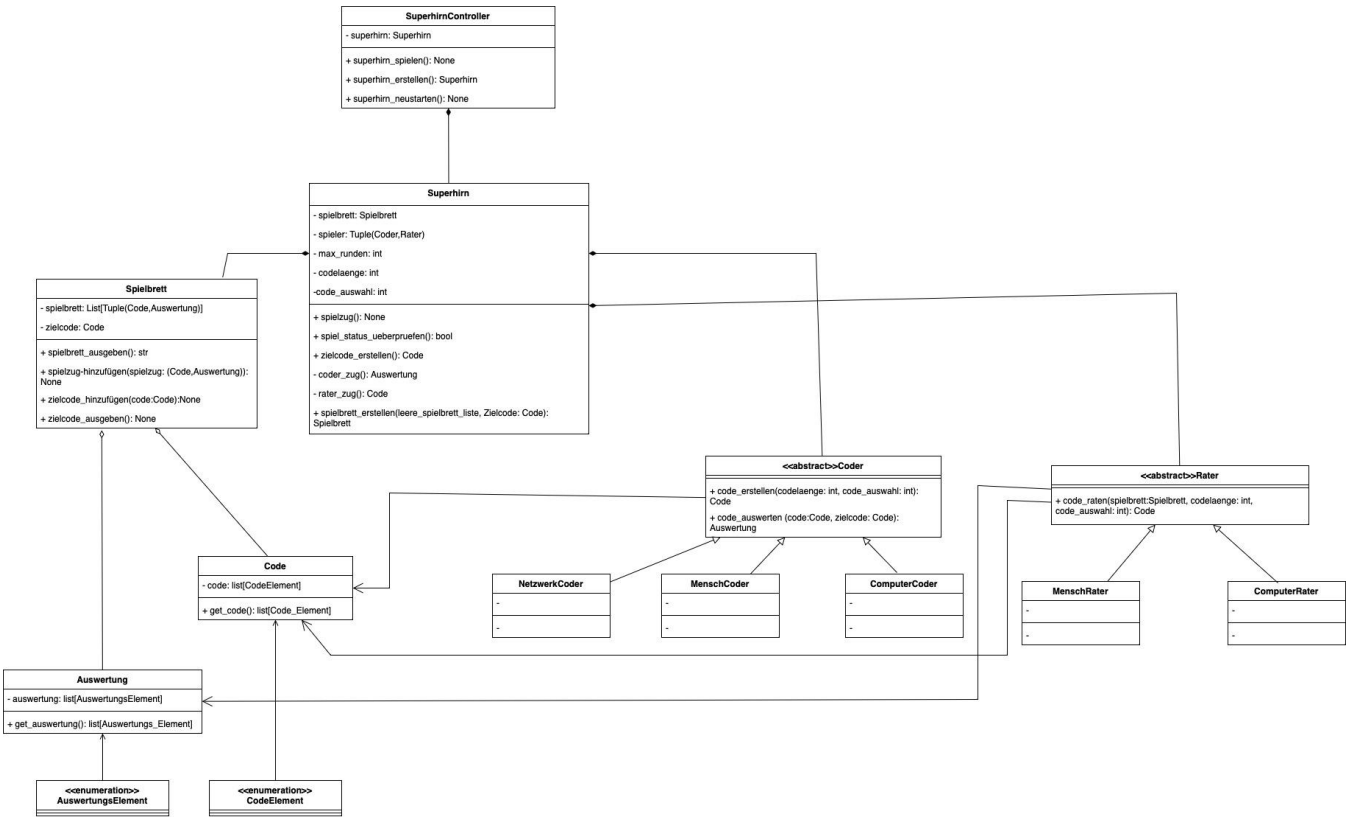
[B11] Sequenzdiagramm zum Rater



[B12] Sequenzdiagramm zum Spielende



[B13] Aktivitätsdiagramm zum NetzwerkCoder



[B14] Detailliertes Klassendiagramm

Quellenverzeichnis

Helmut Balzert: Lehrbuch der Softwaretechnik. Band 2: Softwaremanagement, Software-Qualitätssicherung, Unternehmensmodellierung