InformatiCup 2021: Dokumentation

Team Chillow
(Florian Trei, Jonas Hellmann)

Wintersemester 2020/21

Spiel-Implementierung:
spe_ed

Stand:
1. Januar 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

KI	Künstliche Intelligenz
PR	Pull Request
MVC	Model-View-Controller
UI	User Interface
ER	Entity-Relationship
DB	Datenbank

Kapitel 1

Einleitung

In dieser Dokumentation wird der Lösungsentwurf vom Team Chillow der Universität Oldenburg für den InformatiCup 2021, der von der Gesellschaft für Informatik organisiert wird, beschrieben. Das Team besteht aus den Mitgliedern Florian Trei und Jonas Hellmann, die zum Zeitpunkt des Wettbewerbs im fünften Semester in den Studiengängen B. Sc. Informatik bzw. B. Sc. Wirtschaftsinformatik eingeschrieben sind. Das Repository mit dem Quellcode zu der hier beschriebenen Lösung ist unter folgendem Link abrufbar: `https://github.com/jonashellmann/informaticup21-team-chillow`.

Die Aufgabenstellung des InformatiCup 2021 [1] sieht eine Implementierung des Spiels `spe_ed` vor. Hierbei steuern bis zu sechs Spieler rundenbasiert eine Figur, die besuchte Felder markiert und bei einer Kollision mit einem bereits markierten Feld oder beim Verlassen des Spielfelds verliert. Ziel ist es, eine eigenständig spielende KI zu programmieren, die möglichst viele Spiele gewinnt.

Kapitel 2

Einstieg in das Projekt

Bevor mit der Implementierung der KI begonnen werden konnte, mussten zuerst noch einige andere Punkte geklärt bzw. erledigt werden. Diese ersten Schritte werden im Folgenden erläutert.

2.1 Auswahl der Programmiersprache

Zu Beginn war zu klären, mit welcher Programmiersprache dieser Lösungsvorschlag umgesetzt werden soll. Die Wahl ist dabei schnell auf Python gefallen, obwohl beide Gruppenmitglieder hiermit noch keinerlei Erfahrung aufweisen konnten. Der Grund für diese Entscheidung liegt neben dem starken Interesse an dem Kennenlernen einer neuen Programmiersprache auch an der bereits sehr hohen und immer noch steigenden Popularität der Programmiersprache und der damit verbundenen zukünftigen Wichtigkeit. [2] [3] Hinzu kommt, dass wir vor Beginn der Implementierung anhand der Aufgabenstellung das Potenzial für den Einsatz von Machine Learning gesehen haben und Python in diesem Bereich oft empfohlen wird. [4] [5]

2.2 Erstellung eines lauffähigen Projekts

Damit unsere zu implementierende Lösung auch ausgeführt und manuell getestet werden konnte, war es zunächst notwendig, ein minimales, ausführbares Projekt aufzusetzen.

2.2.1 Einsatz von Poetry als Build-Tool

Ein einfacher Weg, um Abhängigkeiten in Python zu verwalten, ist, eine Datei mit dem Namen `requirements.txt` zu verwenden und in dieser die eingesetzten Bibliotheken aufzulisten. [6] Wir haben uns allerdings für die Verwendung von Poetry als vollständiges Build-Tool entschieden, da dieses zum einen sehr einfach zum Einstieg ist und simple Kommandozeilen-Befehle bereitstellt, zum anderen aber auch weitere Funktionen wie das Ausführen von Tests und Erstellen eines fertigen Pakets anbietet und die Auflösung komplexerer Abhängigkeiten von Paketen durch dieses Tool sehr gut funktioniert. [7]

2.2.2 Entwicklung eines Dockerfile

Zum Start haben wir ein minimales Python-Skript erstellt, das lediglich die an den Docker-Container übergebenen Parameter für die Server-URL und den API-Key ausgibt. Zwar mussten bis hierhin noch keine Abhängigkeiten hinzugefügt werden, aber bei der Konzeption des Dockerfiles sollte bereits die Installation zusätzlicher Bibliotheken berücksichtigt werden. Mit der Nutzung des Standard-Python-Containers von Docker Hub [8] wird bereits ein vorgefertigter Container bereitgestellt, in dem Python und Pip installiert sind. Bei Pip handelt es sich um „ein rekursives Akronym für Pip Installs Python und ist das Standardverwaltungswerkzeug für Python-Module“ [9]. Mittels diesem Tool wird die Installation von Poetry durchgeführt. Poetry wiederum bietet anschließend die Möglichkeit, die verwalteten Abhängigkeiten in Form einer `requirements.txt`-Datei zu exportieren, welche dann von Pip eingelesen werden kann, um die Bibliotheken zu installieren.

Letztendlich wurde ein Dockerfile entworfen, welches die in *Listing 2.1 (Dockerfile zum Erstellen eines lauffähigen Containers)* dargestellten Befehle enthält. Die Umgebungsvariable `TERM` wird gesetzt, um eine farbliche Ausgabe in der Konsole zu ermöglichen. Auf die Parameter zum Start der Anwendung wird in *Kapitel 9 (Benutzerhandbuch)* eingegangen. Bei der Erstellung wurden die von Docker vorgeschlagenen Best Practices [10] umgesetzt.

```
1  FROM python
2
3  COPY . /app
4  WORKDIR /app
5
6  ENV TERM=xterm-256color
7
8  RUN python -m pip install --upgrade pip \
9      && pip install poetry \
10     && poetry export -f requirements.txt | pip install -r /dev/stdin
11
12 ENTRYPOINT [ "python", "./main.py", "--deactivate-pygame=TRUE" ]
13 CMD [ "--play-online=TRUE" ]
```

Listing 2.1: Dockerfile zum Erstellen eines lauffähigen Containers

2.3 Nachstellung des Spiels

Um die zu entwickelnde KI ohne die Server-Verbindung manuell testen zu können, haben wir bei der Implementierung damit begonnen, die Spiellogik nachzustellen. Dazu haben wir nach Erhalt des API-Keys anhand der bereitgestellten Dokumentation und der Möglichkeit, das Spiel in einer Online-Version im Browser testen zu können, begonnen, die Spiellogik zu analysieren.

Auf diese Analyse wird im *Kapitel 4.3 (Implementierung des Offline-Spiels)* genauer eingegangen. Zum Testen haben wir zwei verschiedene Oberflächen entwickelt. Zum einen kann das Spielfeld nach jeder Runde auf der Konsole ausgegeben werden. Dazu wird das Spielfeld als Tabelle dargestellt und in den Zellen, in denen sich ein Spieler befindet, wird dessen Spieler-ID angezeigt. Diese Darstellung wird in der *Abbildung 2.1 (Darstellung der Oberfläche in der Konsole)* verdeutlicht.

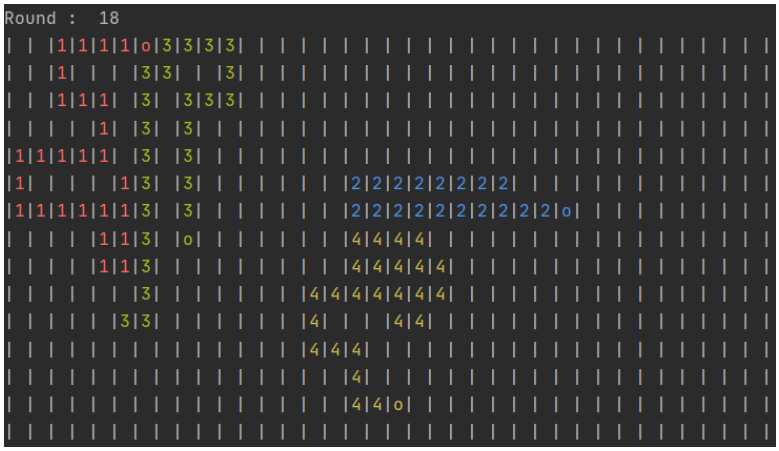

Abbildung 2.1: Darstellung der Oberfläche in der Konsole

Zum anderen gibt es auch die Möglichkeit, das Spiel auf einer grafischen Oberfläche zu spielen, die mittels PyGame [11] entworfen wurde. Jedem Spieler wird dabei zu Spielbeginn eine Farbe zugeordnet, um die verschiedenen Spieler auseinanderhalten zu können. Ein Spiel mit grafischer Oberfläche wird in *Abbildung 2.2 (Darstellung der grafischen Oberfläche mit PyGame)* gezeigt.

Abbildung 2.2: Darstellung der grafischen Oberfläche mit PyGame

Kapitel 3

Lösungsansatz

3.1 Selbstlernende KI mithilfe von Trainingsdaten

Zunächst war eine Überlegung, ob eine KI mithilfe generierter Daten trainiert werden kann. Je nach vorliegender Spielsituation würde die KI bspw. unter Einsatz von neuronalen Netzen eigene Entscheidungen treffen können. Dieser Ansatz bringt jedoch das Problem mit sich, dass „präzise Trainingsdaten benötigt werden, die den Algorithmen helfen, bestimmte Muster oder Ergebnisreihen zu verstehen, wenn eine bestimmte Frage gestellt wird“ [12]. Bei einer maximalen Anzahl von sechs Spielern im Spiel `spe_ed` und fünf unterschiedlichen Aktionen gibt es sehr viele Möglichkeiten, wie ein Spiel verlaufen kann. Pro Spielzug ergeben sich jeweils $5^6 = 15625$ Varianten, welche sich mit jedem weiteren Spielzug potenzieren. Dabei entsteht das Problem, beurteilen zu müssen, welche Spielsituationen und welche Spielverläufe als Trainingsdaten gut geeignet sind, sodass die gesamte Komplexität des Spiels in den Trainingsdaten abgebildet wird.

Ist es nicht möglich, eine ausreichend umfangreiche Menge an Daten generieren zu können, besteht die Gefahr, dass eine damit trainierte KI deutlich schlechtere Entscheidungen trifft als eine Implementierung, die auf simpleren und programmatisch vorgegebenen Heuristiken basiert. Dies wird unter anderem in der Arbeit von Erik Bohnsack und Adam Lilja deutlich, die ein selbstlernendes Programm für eine ähnliche Problemstellung entwickelt haben, jedoch zu dem Schluss gekommen sind, dass „jede Art von taktischer Vorstellung fehlt, was es bestenfalls zu einem durchschnittlichen Gegner machte“ [13]. Zwar könnten wir eigene Daten durch manuelles Spielen des Spiels erstellen, allerdings nimmt dies sehr viel Zeit in Anspruch und gewährleistet darüber hinaus nicht, dass diese Daten repräsentativ für die Strategien anderer Spieler sind.

Aufgrund dessen fiel die Entscheidung, zunächst Lösungsansätze ohne eine selbstlernende KI zu implementieren. Diese Strategie hatte für uns den Vorteil, mithilfe einfacherer Lösungsansätze die Komplexität und auftretenden Probleme im Spielverlauf besser kennenlernen zu können.

3.2 Lösungsansätze ohne selbstlernende KIs

Bei dem Ansatz, Strategien fest im Code zu implementieren, hatten wir mehrere unterschiedliche Ideen, die nachfolgend und aufeinander aufbauend beschrieben werden. Ziel war es hierbei, Teilprobleme zu erkennen und zu lösen, mit der Intuition diese unterschiedlichen KIs kombinieren zu können. Die nachfolgend beschriebenen KIs und die Abhängigkeiten dieser können in *Anhang A.1: (UML-Klassendiagramm der KIs)* nachvollzogen werden. Die Mehrfachvererbungen, wie sie im UML-Klassendiagramm dargestellt werden, konnte in der Implementierung durch die Verwendung von Python als Programmiersprache genau so umgesetzt werden.

3.2.1 RandomAI

Die `RandomAI` ist unsere erste lauffähige KI gewesen und unser Maß für die einfachste KI. Diese stellt zwar keinen wirklichen Lösungsansatz dar, diente jedoch als Einstieg und um erste Probleme zu erkennen. Durch die `RandomAI` ist uns das grundlegende Problem aufgefallen, dass sich die KI selber tötet. Folglich wird die Minimal- und Maximalgeschwindigkeit überschritten, das Spielfeld verlassen oder in vorhandene Spuren gefahren, mit denen eine Kollision vermeidbar gewesen wäre.

3.2.2 NotKillingItselfAI

Aufgrund des beschriebenen Problems der `RandomAI` haben wir uns dafür entschieden, die `NotKillingItselfAI` zu implementieren. Die KI wählt aus allen Aktionen eine zufällige Aktion aus, die sie nicht direkt verlieren lässt. Dazu wird für jede mögliche Aktion die Spur berechnet, die entstehen würde und auf Kollisionen überprüft. Aktionen, die eine Kollision hervorrufen, werden nicht ausgeführt. Hierbei bleiben mögliche Aktionen der Gegenspieler zunächst unberücksichtigt.

Durch die Implementierung der `NotKillingItselfAI` fiel auf, dass weiterhin schlechte Entscheidungen bei Geschwindigkeiten größer 1 getroffen werden. Die erhöhte Geschwindigkeit kann dafür sorgen, dass die KI im nächsten Zug durch die Geschwindigkeit keine verbleibende Aktion hat, bei der sie nicht verlieren wird. In vielen Spielsituation ist dies vermeidbar, indem die Geschwindigkeit nicht erhöht oder sogar verringert wird. In der *Abbildung 3.1 (`NotKillingItselfAI` Fehlentscheidung bei Geschwindigkeit größer eins)* kann das Problem nachvollzogen werden. Links ist die Ausgangssituation des Beispiels zu sehen und im mittleren Bild wurde dann anstelle der ebenfalls möglichen Aktionen change_nothing und turn_right, die Aktion speed_up gewählt und dadurch die Geschwindigkeit zwei erreicht. Die `NotKillingItselfAI` hat nun zwar eine Aktion gewählt, die sie nicht direkt in diesem Zug verlieren lässt, jedoch im direkt darauf folgenden Spielzug.
Dieses Problem haben wir bei der Implementierung berücksichtigt und der `NotKillingItselfAI` kann optional eine Tiefe übergeben werden, mit der die Anzahl Spielzüge festgelegt wird, in denen die gewählte Aktion nicht zum Verlieren führen soll. Das sorgt dafür, dass die KI entsprechend der Tiefe in die Zukunft vorausschaut und nicht verlieren wird. Dabei bleiben die

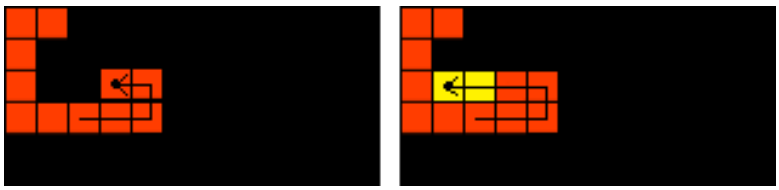

Abbildung 3.1: `NotKillingItselfAI` Fehlentscheidung bei Geschwindigkeit größer eins

gegnerischen Aktionsmöglichkeiten jedoch unberücksichtigt.

Außerdem hat die `NotKillingItselfAI` das Problem, dass sie sich zwar nicht mehr im unmittelbar folgenden Zug tötet, jedoch häufig in Sackgassen läuft. Dieses Problem wird in der *Abbildung 3.2 (`NotKillingItselfAI` unten rechts in einer Sackgasse)* verdeutlicht.

Abbildung 3.2: `NotKillingItselfAI` unten rechts in einer Sackgasse

3.2.3 PathfindingAI

Folglich haben wir einen Lösungsansatz gesucht, welcher das Betreten von Sackgassen zu vermeiden versucht. Genutzt wurde als Grundlage die `NotKillingItselfAI`, sodass nur zwischen Aktionen gewählt wird, die den eigenen Spieler überleben lassen. Um Aktionen zu finden, die nicht in eine Sackgasse führen, haben wir uns für einen Lösungsansatz zum Finden von Pfaden entschieden. Dazu wird eine konfigurierbare Anzahl zufälliger Koordinaten auf dem Spielfeld generiert, auf denen sich bisher kein Spieler befindet. Anschließend wird für jede Aktion geprüft, zu wie vielen dieser Koordinaten nach der Ausführung der Aktion noch ein möglicher Pfad

existiert. Die Aktion, die die höchste Anzahl Koordinaten erreichen kann, führt mit höchster Wahrscheinlichkeit nicht in eine Sackgasse und wird ausgewählt. In der *Abbildung 3.3 (Auswirkung unterschiedlicher Aktionen auf die erreichbaren Pfade)* wird dies grafisch verdeutlicht. Die gelben Quadrate stellen die zu erreichenden Koordinaten dar und somit ist nach der Aktion turn_left nur noch ein Pfad erreichbar und nach Ausführung der Aktion turn_right hingegen drei Pfade. Folglich wählt die `PathfindingAI` die Aktion turn_right.

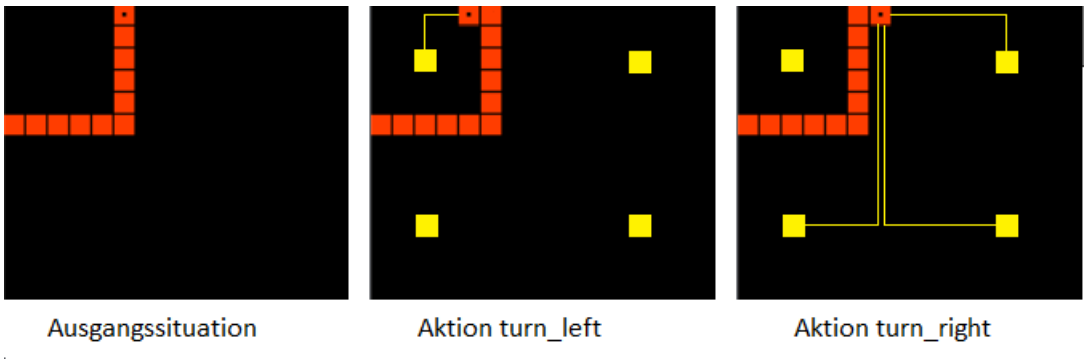

Abbildung 3.3: Auswirkung unterschiedlicher Aktionen auf die erreichbaren Pfade

Bei diesem Lösungsansatz arbeiten wir lediglich mit einer Wahrscheinlichkeit, dass wir nicht in eine Sackgasse laufen. Je höher allerdings die Zahl der zu prüfenden Pfade in Relation zu den freien Feldern gewählt wird, umso unwahrscheinlicher wird eine schlechte Entscheidung. Der Grund für dieses Verhalten wird deutlich, wenn man ein Szenario betrachtet, in dem zu jeder freien Zelle ein Pfad gesucht wird. Hier wird offensichtlich die Aktion genommen, die tatsächlich die meisten Pfade erreichen kann und in den Bereich führt, der die meisten freien Felder hat. Aufgrund der steigenden Berechnungsdauer mit steigender Anzahl Pfade, die kontrolliert werden müssen, musste eine Pfadanzahl kleiner als die der freien Felder gewählt werden. In der Evaluation in *Kapitel 5.3.1 (Ergebnisse der ersten Evaluation)* wurde jedoch ersichtlich, dass die Anzahl Pfade deutlich unter der Anzahl freier Felder liegen kann, ohne offensichtlich schlechte Entscheidungen zu treffen.
Wir haben die Python-Bibliothek `pathfinding` [14] verwendet, die bereits unterschiedliche Algorithmen zum Finden von Pfaden in einem Graphen bietet. Die Entscheidung, welcher Algorithmus zum Finden eines Pfades verwendet wird, hat eine große Auswirkung auf die Performance der `PathfindingAI`. Der Algorithmus stellt den Flaschenhals dar, da dieser je nach Konfiguration mehrere hundert bis tausend mal pro Zug ausgeführt werden muss, wie es bspw. in *Kapitel 3.2.3 (Best-First-Algorithmus)* beschrieben wird. Wir haben unterschiedliche Algorithmen, die die Bibliothek zur Verfügung stellt, hinsichtlich ihrer Ausführungsdauer verglichen und uns folglich für die Verwendung vom Best-First-Algorithmus entschieden. Bei dem Vergleich wurde in 50 unterschiedlichen Spielsituationen jeweils 200 Pfade für jede mögliche Aktion gesucht, um den schnellsten Algorithmus zu finden. Die durchschnittlichen Ausführungszeiten auf unserem Testrechner können der *Tabelle 3.1 (Ausführungsdauer unterschiedlicher Algorithmen zum Finden von Pfaden)* entnommen werden.

Algorithmus	Durchschnittliche Ausführungsdauer in Sekunden
BestFirst	0.740099930763244
BiAStarFinder	0.994705820083618
AStarFinder	1.257498860359192
BreadthFirstFinder	1.376600646972656
DijkstraFinder	2.440193486213684

Tabelle 3.1: Ausführungsdauer unterschiedlicher Algorithmen zum Finden von Pfaden

Best-First-Algorithmus

Der Best-First-Algorithmus „ist ein Algorithmus zum Durchsuchen eines Graphen, bei dem in jeder Iteration der vielversprechendste Knoten gewählt wird, bewertet nach einer gewissen Heuristik. Damit zählt er zu den informierten Such-Algorithmen“ [15]. Der Dokumentation zur von uns genutzten Klasse `BestFirst` der Bibliothek `pathfinding` kann entnommen werden, dass für den Best-First-Algorithmus ein ähnlicher Ansatz wie beim A*-Algorithmus genutzt wird und zusätzlich eine Heuristik für die Bewertung der Nachbarknoten angegeben werden kann. [16] Dieser A*-Algorithmus ist nach Stout „der am besten etablierte Algorithmus für die allgemeine Suche nach optimalen Pfaden“, welcher „garantiert, den kürzesten Weg zu finden, solange die heuristische Schätzung zulässig ist“ [17]. Als Heuristik zur Berechnung der Distanz zwischen zwei Punkten wird bei diesem Algorithmus die Manhattan-Distanz genutzt, die die Summe der absoluten Differenzen der x- und y-Koordinaten darstellt. [18]
Bei jedem Nachbarknoten des aktuell betrachteten Knotens wird die Manhatten-Distanz zum Start- sowie Endknoten berechnet. Summiert ergeben diese beiden Distanzen dann eine heuristische Bewertung des Nachbarknotens. Die Suche wird dann mit dem Knoten fortgesetzt, für den die beste Bewertung errechnet wurde und die umliegenden Knoten hinsichtlich der Distanzen aktualisiert. Dies wird solange fortgeführt, bis der beste Pfad gefunden oder alle erreichbaren Knoten des Graphen besucht wurden. [19]

3.2.4 SearchTreeAI

Sowohl bei der `NotKillingItselfAI` als auch der `PathfindingAI` gibt es weiterhin ein Problem. Die KI errechnet zwar Aktionen, die sie für die nächste Runde überleben lassen, allerdings werden die möglichen Aktionen der Gegenspieler nicht beachtet und es ist keine Prognose möglich, wie gut diese Aktion in den kommenden Runden sein könnte. Ziel der `SearchTreeAI` ist es, dass nur Aktionen berücksichtigt werden, bei denen bereits alle möglichen gegnerischen Aktionen und dessen Auswirkungen berücksichtigt werden und der eigene Spieler trotzdem überlebt.

Unser Lösungsansatz für dieses Problem ist es, mithilfe einer Baumstruktur alle möglichen gegnerischen und eigenen Aktions-Kombinationen eine bestimmte Anzahl Spielzüge in die Zukunft zu simulieren. Ein solcher Baum besteht aus einer Menge von Knoten, die einem Vorgänger-Knoten zugeordnet sind und Nachfolger haben können. Eine Ausnahme stellt der Wurzelknoten dar, der den Startpunkt markiert und keinen Vorgänger hat. [20] In unserem Fall repräsentiert jeder

Knoten einen Zustand des Spiels, nachdem von eigenen oder allen anderen Spielern Aktionen durchgeführt wurden. Es wird dann nach einem Teilbaum gesucht, bei dem vorhergesagt werden kann, dass der eigene Spieler unabhängig der von allen anderen Spielern ausgewählten Aktionen nicht in den nächsten Zügen stirbt. Für diese Simulation konnte viel von der in *Kapitel 4.3 (Implementierung des Offline-Spiels)* beschriebenen Implementierung wiederverwendet werden.

Diese Idee wird in *Abbildung 3.4 (Visualisierung der Idee der* `SearchTreeAI`*)* visualisiert, wobei es in diesem Beispiel neben dem eigenen Spieler noch die beiden weiteren Spieler mit den IDs 1 und 2 gibt und eine Simulation lediglich für den nächsten Schritt demonstriert wird. Die Buchstaben an den Pfaden sollen jeweils die simulierte Aktion kennzeichnen. Der Baum wird von links nach rechts aufgebaut und besteht pro Zug aus zwei Ebenen. Zuerst wird ausgehend vom aktuellen Spielstand die eigene Aktion des Spielers berechnet und abgebrochen, wenn hier schon kein Überleben mehr möglich ist. Im zweiten Schritt wird für jede mögliche Aktion des eigenen Spielers jede mögliche Kombination der gegnerischen Aktionen simuliert und wiederum das Überleben geprüft.

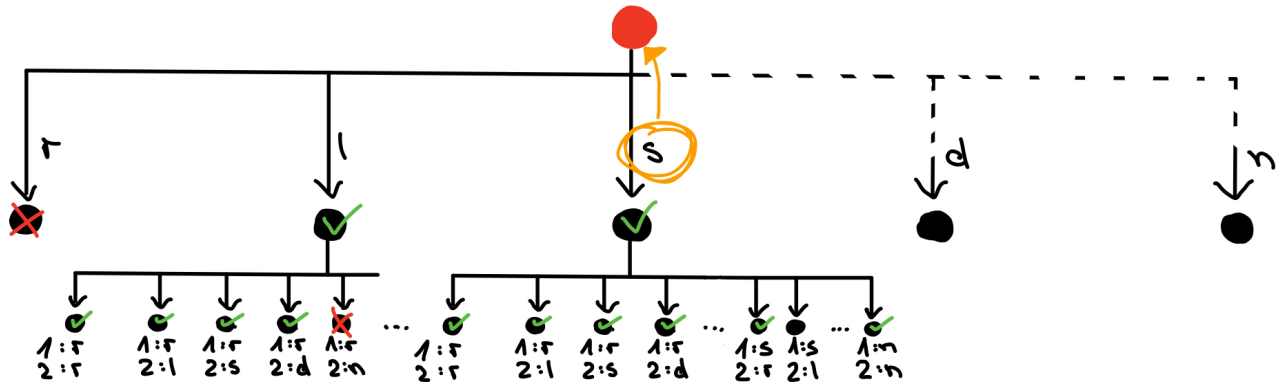

Abbildung 3.4: Visualisierung der Idee der `SearchTreeAI`

Wurde eine Ausgangsaktion gefunden, bei der alle nachfolgenden Pfade das Überleben des Spielers garantieren, wird diese Aktion gewählt. Ein Problem stellt hierbei allerdings die für jeden Zug extrem stark steigende Anzahl zu berechnender Aktionen dar. Während bei einem Spiel mit der Maximalzahl von sechs Spielern und fünf möglichen Aktionen die Berechnung von drei Zügen im schlimmsten Fall $(6^5)^3 = 470.184.984.576$ Kombinationen umfasst, sind es bei vier Zügen schon $(6^5)^4 = 3.656.158.440.062.976$.

Da diese Anzahl an Berechnungen in der Zeit bis zum Ablaufen der Deadline nicht durchgeführt werden kann, wurde diese Anzahl reduziert, indem in die Simulation nicht immer alle Spieler einberechnet werden, sondern nur noch solche, die sich in einer gewissen Distanz zu dem eigenen Spieler befinden. Um die Entfernung zwischen dem eigenen und den anderen Spielern unter der Berücksichtigung der bereits belegten Felder zu errechnen, konnte der bereits beschriebene Algorithmus zum Finden von Pfaden wiederverwendet werden. Ist dann bspw. nur noch ein

weiterer Spieler überhaupt in unmittelbarer Nähe, reduziert sich die Zahl von 470.184.984.576 auf lediglich $(2^5)^3 = 32.768$ Kombinationen.

3.2.5 SearchTreePathfindingAI

Die `SearchTreePathfindingAI` kombiniert die Stärken der `SearchTreeAI` und der `PathfindingAI`. Hierbei wird die Entscheidung der `SearchTreeAI` priorisiert. Dazu werden der `PathfindingAI` zur Berechnung der besten Aktionen nur noch die Aktionen zur Verfügung gestellt, die durch die `SearchTreeAI` ermittelt wurden. Dadurch erreichen wir, dass die Aktion mit den meisten Pfaden gewählt wird, die in jedem Fall die nächsten Züge entsprechend der gewählten Tiefe des Suchbaums überleben wird.
Ein Nachteil dieser Variante ist, dass die `SearchTreePathfindingAI` sich durch andere gegnerische Spieler in seltenen Spielsituationen in Sackgassen zwingen lassen kann.

3.2.6 PathfindingSearchTreeAI

Die `PathfindingSearchTreeAI` kombiniert ebenso wie die `PathfindingSearchTreeAI` die Stärken der `SearchTreeAI` und der `PathfindingAI`, priorisiert jedoch die Entscheidungsgrundlage der `PathfindingAI`.
Die `PathfindingAI` erstellt zunächst eine Liste der Aktionen mit den erreichbaren Pfaden. Die Liste wird absteigend nach der Aktion sortiert, die die meisten Pfade erreichen kann. Die `SearchTreeAI` berechnet ebenfalls eine Liste möglicher Aktionen. Aus diesen beiden Listen muss dann eine Aktion ausgewählt werden, die den beiden Ergebnissen der KIs genügt.
Die Priorisierung hinsichtlich der `PathfindingAI` ist so umgesetzt, dass der `PathfindingSearchTreeAI` bei der Initialisierung ein Parameter übergeben wird, der eine Toleranz zwischen 0 und 1 darstellt. Eine mögliche Aktion der `SearchTreeAI` muss mindestens die maximal mögliche Anzahl an Pfaden multipliziert mit der Toleranz (3.1) erreichen können.

$$erreichbare\_Pfade >= maximal\_erreichbare\_Pfade * Toleranz \tag{3.1}$$

Sofern keine Aktion gefunden wird, die in beiden Aktions-Listen vorhanden ist und dieser Formel entspricht, wird die beste Aktion der `PathfindingAI` gewählt. Diese Strategie bewirkt, dass sich die `PathfindingSearchTreeAI` nicht in zu kleine Gebiete durch die Gegner zwingen lässt. Jedoch besteht das Risiko, eine vermeidbare Kollision zu verursachen.

3.2.7 Weitere nicht umgesetzte Lösungsansätze

Alle bisher beschriebenen Lösungsansätze haben die Eigenschaft, dass sie so lange wie möglich überleben wollen. Dabei wird jedoch nicht der Ansatz verfolgt, die anderen Mitspieler gezielt dazu zu bringen, kürzer zu überleben als man selbst. Nachfolgend werden derartige Lösungsideen beschrieben und erklärt, warum wir uns gegen die Implementierung dieser entschieden haben.

Als ein möglicher Lösungsansatz stand zur Diskussion, die gegnerischen Spieler an den Spielfeldrand zu drängen und ihn somit zum Verlieren zu zwingen. Bei diesem Ansatz muss die eigene KI dicht an die gegnerische KI fahren, damit diese zu Entscheidungen gezwungen wird, die nicht optimal sind und ein vorzeitiges Verlieren bedeuten.
Ein Problem, dass uns bei einer zu kleinen Distanz zu den gegnerischen Spielern aufgefallen ist, ist jedoch, dass man den Aktionen der Mitspieler vertrauen muss. Es ist notwendig, dass der Gegner kein Risiko eingeht und immer nur Aktionen wählt, die unter Berücksichtigung der möglichen Aktionen unserer KI nicht zum Verlieren führen. Ansonsten kann es passieren, dass es aufgrund der Nähe zum anderen Spieler schnell zu einer Kollision kommen kann. Würde die KI nicht dicht an den Gegner heranfahren und einen Sicherheitsabstand einhalten, wäre diese Strategie unwirksam, denn es bleiben zu viele gute Optionen für den Gegner übrig.

Resultierend aus dem Problem, nicht absichtlich zu dicht an Mitspieler heranfahren zu wollen, entstand der Lösungsansatz, einen Gegner zu umkreisen und somit in ein kleines Gebiet einzusperren. Dadurch würde man dazu beitragen, dass der Gegner entweder nur noch wenige Züge zur Verfügung hat und den verbleibenden Platz nutzen muss oder einen sechsten Zug mit einer Geschwindigkeit größer zwei nutzen muss, um aus dem begrenzten Feld mittels eines Sprungs wieder heraus zu kommen. Die Berechnung, mit einem sechsten Zug über die umkreisende Spur zu kommen, ist entsprechend eigener Lösungsversuche nicht trivial und zudem existiert dazu in einigen Spielsituationen keine mögliche Option. Diese Vorteile sprechen zunächst für diesen Lösungsansatz, jedoch bringt er wiederum auch eine erhöhte Gefahr für das Ausscheiden der eigenen KI mit sich.
Um einen gegnerischen Spieler umkreisen zu können, muss eine solche aggressive KI in den meisten Fällen schneller sein als der Gegenspieler, da dieser eine solche Umkreisung relativ einfach verhindern kann. Dies bedeutet aber auch, dass Geschwindigkeiten erreicht werden müssen, die Löcher in der Spur in jedem sechsten Zug hinterlassen, wodurch das Umkreisen an Wirkung verlieren würde. Außerdem hat sich eine hohe Geschwindigkeit bei der Evaluation unserer implementierten Lösungsansätze bereits als riskant herausgestellt. Die erhöhte Geschwindigkeit sorgt dafür, dass häufig nur wenige oder keine Aktion zur Auswahl stehen, die im nächsten Spielzug nicht zum Verlieren führen. Diese Erkenntnis kann dem *Kapitel 5.3.1 (Ergebnisse der ersten Evaluation)* entnommen werden.
Daher steigt die Wahrscheinlichkeit, dass die eigene KI durch ihr aggressives Verhalten frühzeitig ausscheidet. Dieses Risiko gepaart mit dem Problem der Löcher in der Umkreisung hat dafür gesorgt, dass auch dieser Lösungsansatz nicht implementiert wurde.

Aufgrund der genannten Probleme haben wir uns dagegen entscheiden, eine KI zu implementieren, die proaktiv für das Ausscheiden anderer Spieler sorgt, da sich das Risiko für das eigene Ausscheiden zu stark erhöht.

Kapitel 4

Implementierung

Den Einstieg in das Programm stellt die Datei `main.py` dar. Hier wird entschieden, ob ein Online- oder Offline-Spiel, wie bereits in den vorherigen Kapiteln beschrieben, gestartet werden soll. Die Implementierungen dieser beiden Spielvarianten sollen in diesem Kapitel beschrieben werden, wobei der Fokus auf die Online-Verbindung gerichtet ist, da es sich hierbei um die Umsetzung der eigentlichen Aufgabenstellung handelt.

4.1 Modellierung des Spiels

Um eine Grundlage zu haben, auf der die Implementierung aufgebaut werden konnte, wurde zunächst die Modellierung des Spiels vorgenommen. Dazu haben wir geschaut, welche Informationen benötigt und vom Server bereitgestellt werden und wie man diese dann mithilfe eines objektorientierten Ansatzes abbilden kann.

Das Ergebnis der Modellierung ist in *Abbildung 4.1 (UML-Klassendiagramm des Modells)* zu sehen. Das `Game` hat Zugriff auf die eigenen Eigenschaften, kennt aber auch alle `Player`, die an diesem Spiel teilnehmen und weiß, welcher von diesen der eigene Spieler ist. Zudem besteht ein `Game` aus einem zweidimensionalen `Cells`-Array, der das Spielfeld repräsentieren. In einer `Cell` befinden sich dann alle Spieler, die dieses Spielfeld bereits besucht haben.

Da ein Spieler in eine bestimmte Anzahl an Richtungen gedreht sein kann, wird diese Ausrichtung über die Enumeration `Direction` abgebildet. Ebenso ist die Auswahl der möglichen Aktionen begrenzt, sodass diese in der Enumeration `Action` festgelegt worden sind.

4.2 Implementierung des Online-Spiels

Um eine Verbindung zu dem `spe_ed`-Server aufbauen zu können, müssen die URL und ein gültiger API-Key vor dem Start der Anwendung als Umgebungsvariablen gesetzt worden sein. Diese Websocket-URL wird entsprechend modifiziert auch als Endpunkt zur Abfrage der Server-

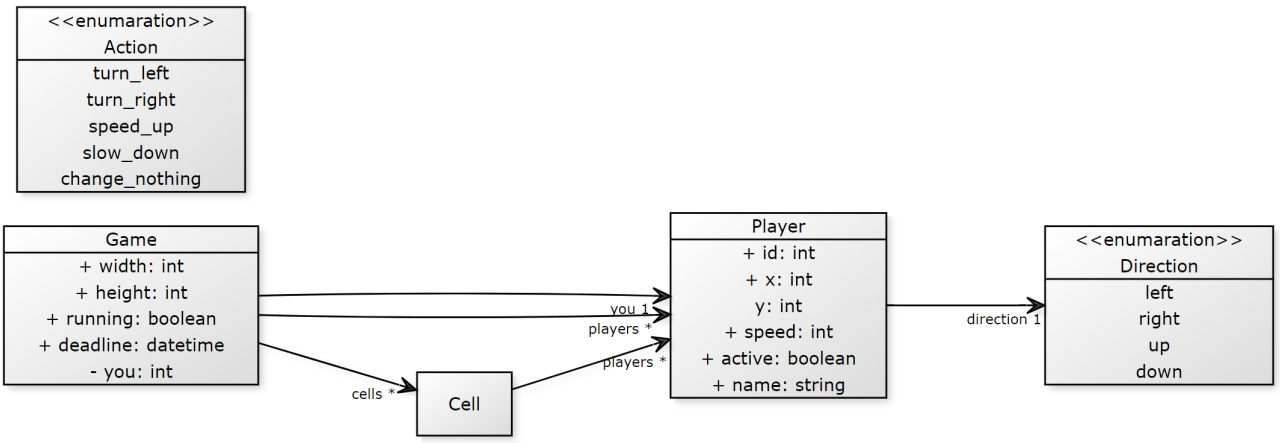

Abbildung 4.1: UML-Klassendiagramm des Modells

Zeit verwendet, auf deren Nutzung nachfolgend noch eingegangen wird.

Die zum Start des Spiels öffentlich bereitgestellte Methode `play()` ist in *Listing 4.1 (`play()`-Methode des `OnlineControllers`)* dargestellt. Diese ist sehr simpel und ruft lediglich die private Methode `__play()` auf. Hierbei handelt es sich um eine asynchrone Methode. Es wird mittels der Bibliothek `asyncio` sichergestellt, dass diese asynchrone Methode vollständig verarbeitet wurde, bevor der Kontrollfluss im Programm weiterläuft. Sobald dies der Fall ist, ist das Spielende eingetreten und die Oberfläche wird beendet.

```
1 def play(self):
2     asyncio.get_event_loop().run_until_complete(self.__play())
3     self.monitoring.end()
```

Listing 4.1: `play()`-Methode des `OnlineControllers`

In der im vorherigen Absatz bereits erwähnten Methode `__play()` in dem `OnlineController` wird eine Websocket-Verbindung zum Server aufgebaut. Anschließend wird in einer Endlosschleife die Logik zur Ausführung eines einzelnen Spielzugs ausgeführt. Wie ein solcher Spielzug abläuft, ist in *Anhang A.2: (Sequenzdiagramm zur Implementierung eines Spielzug)* in Form eines Sequenzdiagramms nachvollziehbar und die Umsetzung im Python-Code kann zusammen mit dem Verbindungsaufbau dem *Anhang A.1: (`__play()`-Methode des `OnlineControllers`)* entnommen werden.

4.2.1 Einlesen des Spiel-Zustands

Der Beginn einer neuen Spielrunde wird damit eingeläutet, dass neue Daten über die Websocket-Verbindung vom Server versendet werden. Dabei handelt es sich jeweils um den aktuellen Zustand des Spiels mit allen notwendigen Daten. Zur Serialisierung wird das Spiel in ein JSON-Format übersetzt. Ein Beispiel zur Veranschaulichung dieses Formats ist in *Anhang A.2: (JSON-Repräsentation eines Spiel-Zustands)* abgebildet.

Dieser eingelesene String wird an ein Objekt der Klasse `JSONDataLoader` übergeben, welches die Übersetzung des JSON-Formats in ein `Game`-Objekt – das wie in *Kapitel 4.1 (Modellierung des Spiels)* beschrieben modelliert ist – zur Aufgabe hat.

Sobald das Spiel fertig aufgebaut wurde, wird ein `GET`-Request an den Server geschickt, auf welchen die aktuelle Server-Zeit als Antwort erwartet wird und ebenfalls durch den `JSONDataLoader` aus einem String in ein `datetime`-Objekt [21] geparst wird. Da die Server- und die System-Zeit abweichen können und die Deadline-Zeit des Spiels auf der Server-Zeit basiert, wird die Abweichung zwischen dem Server und dem eigenen System berechnet und diese dann auf die Deadline angewendet, sodass diese anschließend mit dem eigenen System synchronisiert ist.

4.2.2 Ermittlung der besten Aktion

Anschließend ist das Spiel in seinem aktuellen Zustand korrekt abgebildet. Als Nächstes steht an, dass eine Entscheidung für die nächste Aktion getroffen werden muss. Dies ist allerdings nur notwendig, wenn der eigene Spieler noch aktiv ist.

Die Berechnung und Festlegung auf die nächste Aktion wird von dem `OnlineController` an ein Objekt einer Subklasse der abstrakten Basisklasse `ArtificialIntelligence` mit dem Aufruf der Methode `create_next_action(game: Game)` delegiert. Im ersten Schritt wird von einer vergleichsweise schnellen, aber eher schwächeren KI eine Standard-Aktion für den nächsten Zug berechnet, durch die der Spieler nicht direkt sterben wird. Erst im Anschluss wird die Berechnung der eigentlich ausgewählten KI in einem neuen Prozess gestartet.

Hierbei kann die Berechnung länger dauern, als in der Spielrunde zeitlich zur Verfügung steht. Daher wird dieser Prozess eine Sekunde vor Ablauf der Deadline abgebrochen, falls die Berechnung der KI dann noch nicht beendet ist. Bei einem Abbruch wird die errechnete Standard-Aktion an den Server geschickt, andernfalls wird die Berechnung der stärkeren KI verwendet. Es kann allerdings sein, dass eine KI schon Zwischenergebnisse bereitstellt, welche in dem Fall dann den Rückgabewert darstellen. Dies ist im Speziellen bei der PathfindingSearchTreeAI und der SearchTreePathfindingAI der Fall. Welche Art der im *Kapitel 3 (Lösungsansatz)* beschriebenen KIs gewählt wird, steht dem Anwender grundsätzlich vollkommen frei.

4.2.3 Übergabe der Aktion an die Web-API

Sobald eine Aktion ausgewählt wurde, muss diese dem Server mitgeteilt werden. In der Dokumentation für die diesjährige Aufgabenstellung [1] wurde festgehalten, dass auch hier wieder das JSON-Format verwendet werden soll. Daher wird die Aktion an die Klasse `JSONDataWriter` überreicht, die einen String folgender Form erstellt: {`"action": "speed_up"`}. Dieser wird anschließend über die Websocket-Verbindung an den Server gesendet.

4.3 Implementierung des Offline-Spiels

Bei der Offline-Version des Spiels ging es darum, lokal die eigenen KIs gegeneinander spielen zu lassen und als menschlicher Spieler gegen die KIs antreten zu können, ohne eine Verbindung zum spe_ed-Server herstellen zu müssen. Dies ermöglichte uns, lokal zu testen und unsere verschiedenen Lösungsansätze (siehe *Kapitel 3 (Lösungsansatz)*) gegeneinander auszuprobieren, um zu beurteilen, welche die beste Strategie ist. Die Implementierung des Offline-Spiels erfolgte in der Klasse `OfflineController`, die entsprechend der Ober-Klasse `Controller` die Methode `play()` implementiert.

Die Logik, wie die Antworten mit den Aktionen der Spieler auf dem Server verarbeitet werden, ist relativ einfach nachzuvollziehen und lässt sich wie folgt zusammenfassen: Nachdem alle Spieler ihre Aktion an den Server gesendet haben, wird jede Aktion eines Spielers in einer Runde zeitgleich ausgeführt. Trifft der Spieler während seiner Aktion auf ein bereits belegtes Feld, so verliert dieser das Spiel, bewegt sich aber noch so viele Felder weiter vorwärts, wie es seine Geschwindigkeit und Richtung normalerweise bewirkt hätten. Verhindert werden können solche Kollisionen mithilfe eines Sprungs, der bei entsprechendem Tempo automatisch in jeder sechsten Runde des Spiels ausgeführt wird.

Da wir bei der Implementierung keine Verbindung zum spe_ed-Server herstellen, erhalten wir keine Aktualisierung des Spiels. Auch müssen die errechneten Aktionen der KI nicht versendet, sondern lokal verarbeitet werden. Daher haben wir die Spiellogik in der Klasse `GameService` nachgebildet. Diese Klasse manipuliert das übergebene Spiel und ist somit der Ersatz zum spe_ed-Server in der Offline-Variante. Die Klasse `GameService` muss folglich in der oben genannten Methode `play()` des `OfflineController`s initialisiert werden. Dazu müssen zunächst `Player`-Objekte und das entsprechende `Game`-Objekt erzeugt werden, welches dem `GameService` übergeben wird. Dieses `Game`-Objekt wird dann im Spielverlauf durch den `GameService` manipuliert.
Zusätzlich werden die KIs erzeugt und exklusiv einem Spieler zugeordnet, der sich im Spiel befindet. Solange das Spiel läuft, werden der Reihe nach die nächsten Aktionen der Spieler/KIs abgefragt und an den `GameService` weitergeleitet. Die Berechnung einer KI findet in einem eigenen Prozess statt. Falls mehr Zeit gebraucht wird, als in der Runde zur Verfügung steht, wird der Prozess abgebrochen, um ein Fortführen des Spiels zu gewährleisten.

4.3.1 Implementierung des GameService

Die Ausführung einer Spieler-Aktion, die Verwaltung der Spielzüge sowie das Manipulieren des `Game`-Objekts sind die Hauptaufgabe des `GameService`. Im *Anhang A.3: (Aktivitätsdiagramm zur Durchführung einer Spieler-Aktion durch den `GameService`)* kann der Ablauf einer Spieler-Aktion durch den `GameService` und die damit verbundenen Änderungen des Spiels nachvollzogen werden. Diese Abfolge der Aktivitäten diente als Vorlage zur Implementierung des `GameService`.

Die daraus entstandene Implementierung wurde wie nachfolgend beschrieben umgesetzt.

Bei dem Aufruf des Konstruktors wird dem `GameService` ein `Game`-Objekt übergeben, welches abgespeichert wird. Zusätzlich erzeugt der `GameService` eine Instanz der Klasse `Turn`, die einen Spielzug repräsentiert. In einem `Turn`-Objekt ist die aktuelle Nummer des Spielzugs und alle Spieler des Spiels enthalten. Außerdem wird eine Liste mit den Spielern gepflegt, die in diesem Zug noch eine Aktion durchführen müssen. Damit ein Spieler seine Aktion durchführen kann, ruft er die Methode `action(player: Player)` auf. Das Turn-Objekt prüft dann, ob dieser Spieler bereits eine Aktion gemacht hat und wirft in dem Fall eine `MultipleActionByPlayerException`, sodass dieser Spieler durch den `GameService` aus dem Spiel ausscheidet. Sollte der Spieler seine erste Aktion in diesem Zug machen, wird er aus der Liste der Spieler mit den ausstehenden Aktionen ausgetragen und geprüft, ob ein neuer Zug initialisiert werden muss, weil dies die letzte erwartete Aktion in dieser Spielrunde war.

Wenn ein Spieler eine Aktion durchführen möchte, verschickt er in der Offline-Variante keine Nachricht an den spe_ed-Server, sondern ruft am `GameService` die Methode `do_action(player: Player, action: Action)` auf. Die Methode manipuliert dabei das `Game`-Objekt, sodass dies eine äquivalente Aktualisierung zum Erhalt des Spiels im JSON-Format durch den spe_ed-Server ist.

Der grobe Ablauf dieser Methode ist wie nachfolgend beschrieben. Bei dem `Turn`-Objekt wird die Methode `action(player: Player)` aufgerufen, um zu prüfen, ob der Spieler eine Aktion machen darf und ob ein neuer Spielzug nach dieser Aktion beginnt. Nachfolgend wird dann die Aktion mit der Methode `get_and_visit_cells(player: Player, action: Action)` simuliert. Der Code der Methode befindet sich im *Anhang A.3: (`get_and_visit_cells(player, action)`-Methode des `GameService`)*. Dabei wird der Spieler, der die Ausführung der Aktion eingeleitet hat, bezüglich seiner x- und y-Koordinaten, seiner Geschwindigkeit und Ausrichtung aktualisiert und im `Game` in die `Cells` eingetragen, die er durch die Aktion neu besucht hat. Sollte ein neuer Spielzug durch die Aktion entstanden sein, werden Spieler mit Kollisionen inaktiv geschaltet und geprüft, ob das Spiel beendet ist.

Mithilfe dieser Logik des `GameService` können wir Spiele ohne Nutzung des spe_ed-Servers durchführen. Bei der Implementierung der Logik wurde darauf geachtet, dass diese auch bei der Implementierung unserer KIs helfen. Dadurch konnte der `GameService` in den KIs häufig genutzt werden, um beispielsweise Züge vorherzusagen oder zu prüfen, ob Aktionen zum Verlieren führen.

4.4 Bereitstellung einer Oberfläche

Unabhängig davon, ob ein Spiel online oder offline ausgeführt wird, gibt es die Möglichkeit, den Spielverlauf in zwei verschiedenen Formen dargestellt zu bekommen. Zum einen lässt

sich das Spiel auf der Konsole darstellen und zum anderen als grafische Oberfläche mittels PyGame. Darüber hinaus wird ein weiteres Dummy-User Interface (UI) bereitgestellt, das jegliche Ausgaben zu dem Spiel-Zustand unterdrückt. Alle drei Darstellungsvarianten implementieren das Interface `View`.

4.4.1 View-Interface

Das Interface `View` deklariert 3 abstrakte Methoden, die durch die Unterklassen implementiert werden müssen. Die Methode `update(game: Game)` ist dafür gedacht, dem Benutzer das Spielgeschehen fortlaufend mithilfe des übergebenen `Game`-Objekts darzustellen, sodass immer der aktuelle Stand des Spiels angezeigt wird. Mit der Methode `read_next_action()` wird die nächste Aktion eines menschlichen Spielers abfragt, eingelesen und verarbeitet. Die dritte Methode `end()` ist dafür gedacht, notwendige Schritte zum Beenden der Oberfläche auszuführen.

4.4.2 Darstellung des Spiels in der Konsole

Die Umsetzung der konsolenbasierten View geschieht durch die Klasse `ConsoleView`. Zur Darstellung des Spiels auf der Konsole haben wir uns für das Package `tabulate` [22] entschieden. Dadurch war es leicht, strukturierte Tabellen in der Konsole auszugeben, was in der Methode `update(game: Game)` geschieht. Durch das Attribut `cells` in dem Objekt eines Spiels haben wir bereits die Belegung der Felder auf dem Spielfeld durch die `Cell`-Objekte. Dies wird dann in ein zweidimensionales Array mit Strings überführt. Wenn sich kein Spieler auf dem Feld befindet, wird ein leerer String ausgegeben, andernfalls die ID des Spielers. Die Darstellung kann in *Kapitel 2.3 (Nachstellung des Spiels)* eingesehen werden.

Zur Implementierung der Methode `read_next_action()` wurde die Eingabe der Konsole durch die Methode `input` genutzt und entsprechend der Eingabe wird die passende Aktion zurückgegeben.

4.4.3 Nutzung von PyGame als grafische Oberfläche

Die grafische Oberfläche wurde in der Klasse `GraphicalView` umgesetzt. Bei der Darstellung des Spiels haben wir uns für die Nutzung der Bibliothek PyGame entschieden. Der Grund für die Entscheidung ist die leichte Implementierung einer zweidimensionalen Oberfläche, mit der das Spiel spe_ed abgebildet werden kann. Außerdem benötigt PyGame einen geringen Aufwand bei der Initialisierung, um ein Spiel rendern zu können. [11]

Zur Darstellung mittels PyGame durch die Methode `update(game: Game)` wird dann in einem initialisierten Fenster jeder Bereich in Form von Rechtecken farblich bestimmt und aktualisiert. Befindet sich kein Spieler auf dem Feld, wird es schwarz gezeichnet und andernfalls entsprechend einer festgelegten Farbe, die dem Spieler zugeordnet ist, befüllt. Die Darstellung kann in *Kapitel 2.3 (Nachstellung des Spiels)* eingesehen werden. Die Implementierung der Methode

befindet sich im *Anhang A.4: (`update(game: Game)`-Methode der `GraphicalView`).*

Bei der Implementierung der Methode `read_next_action()` für die Eingabe der menschlichen Spieler haben wir die Keylistener von PyGame genutzt. In einem Event sind alle zurzeit gedrückten Tasten gespeichert und somit können wir filtern, welche Aktion vom Spieler gewünscht ist und geben diese zurück.

Die Methode `end()` wird in der grafischen Oberfläche dazu genutzt, um das PyGame ordnungsgemäß zu beenden. Außerdem warten wir zehn Sekunden, sodass die letzte ausgeführte Runde des Spiels nachvollzogen werden kann und die View nicht sofort geschlossen wird.

Kapitel 5

Evaluation des besten Lösungsansatzes

Aufbauend auf den beiden vorangegangenen Kapiteln, in denen Lösungsansätze entworfen und die Implementierung beschrieben wurde, konnte ein Vorgehen zur Auswahl der besten KI-Strategie entwickelt werden, auf die in diesem Kapitel eingegangen wird.

5.1 Vorgehen zur Auswahl der besten Strategie

Die Überlegung hierbei war es, basierend auf der in *Kapitel 4.3 (Implementierung des Offline-Spiels)* beschriebenen Möglichkeit, Spiele auch ohne Zugriff auf den spe_ed-Server mit mehreren unserer KIs ausführen zu können, eine automatisierte Simulation möglichst vieler Spiele durchführen zu können. Die daraus resultierenden Ergebnisse sollten in einer Form gespeichert werden, die es im Anschluss ermöglicht, daraus Informationen über die Stärke einer bestimmten KI mit ihren unterschiedlichen Konfigurations-Möglichkeiten gewinnen zu können.

Zu diesem Zweck wurde die Klasse `AIEvaluationController` erstellt, der vom bereits erläuterten `OfflineController` erbt. Dieser ruft bei der Ausführung in einer Schleife die `play`-Methode des `OfflineController` auf, wobei bei jedem Aufruf ein neues Spiel generiert wird. Hierbei mussten allerdings einige Annahmen getroffen werden, die potenziell einen Einfluss auf das Ergebnis haben könnten:

- Ein Spielfeld besitzt bei der Simulation eine zufällige Höhe und Breite zwischen 30 und 70 Feldern.
- Den KIs wird für die Auswahl in einer Runde eine zufällige Zeit zwischen 3 und 15 Sekunden eingeräumt.
- Es nehmen an einem Spiel immer eine zufällige Anzahl an Spielern teil, die zwischen 3 und 6 liegen kann.

Nach jeder Berechnung einer KI wird die Berechnungsdauer abgespeichert. Gleiches wird nach dem Ende eines jeden Spiels für das Spiel selber und die teilnehmenden Spieler durchgeführt, wobei auch die Information über den Sieger eines Spiels erhalten bleibt.

5.2 Entwurf einer DB zum Speichern der Simulationen

Um die beschriebenen Daten leicht abfragen zu können, fiel die Entscheidung auf die Nutzung einer relationalen DB. In diesem Fall wurde dafür SQLite3 ausgewählt, da die gesamte DB in einer Datei gespeichert wird und somit das Aufsetzen eines Datenbank-Managements-Systems entfällt. [23] Die Verbindung konnte dann mit der Python-Bibliothek `sqlite3` [24] hergestellt werden. Zuerst aber war es notwendig, die DB-Struktur zu entwerfen, um anschließend entsprechende SQL-Statements im Code auf die DB anwenden zu können.

5.2.1 Erstellen des Entity-Relationship-Modells

Im ersten Schritt wurde dazu ein ER-Modell basierend auf den zugrunde liegenden Anforderung entworfen. Wie bereits beschrieben, wollen wir primär Informationen erhalten, wie oft eine bestimmte KI mit welchen Parametern in den simulierten Spielen gewonnen hat. Ziel dieser Evaluation war es nicht, das Verhalten der KI in bestimmten Spielsituationen zu beurteilen, sondern die Qualität der Folge aller Entscheidungen einer KI in einem Spiel zu beurteilen. Die KI, die die beste Folge von Entscheidungen getroffen hat sollte gewonnen haben, sodass wir die Anzahl gewonnener Spiele als Maßzahl für die Stärke der KI gewählt haben. Ein weiterer wichtiger Wert stellt die durchschnittliche Ausführungsdauer einer KI dar und diesen u. a. mit der Spielfeldgröße und der Anzahl der am Spiel teilnehmenden Gegnern in Korrelation bringen zu können. Das daraus resultierende ER-Modell ist in *Abbildung 5.1 (ER-Modell der Evaluations-DB)* dargestellt.

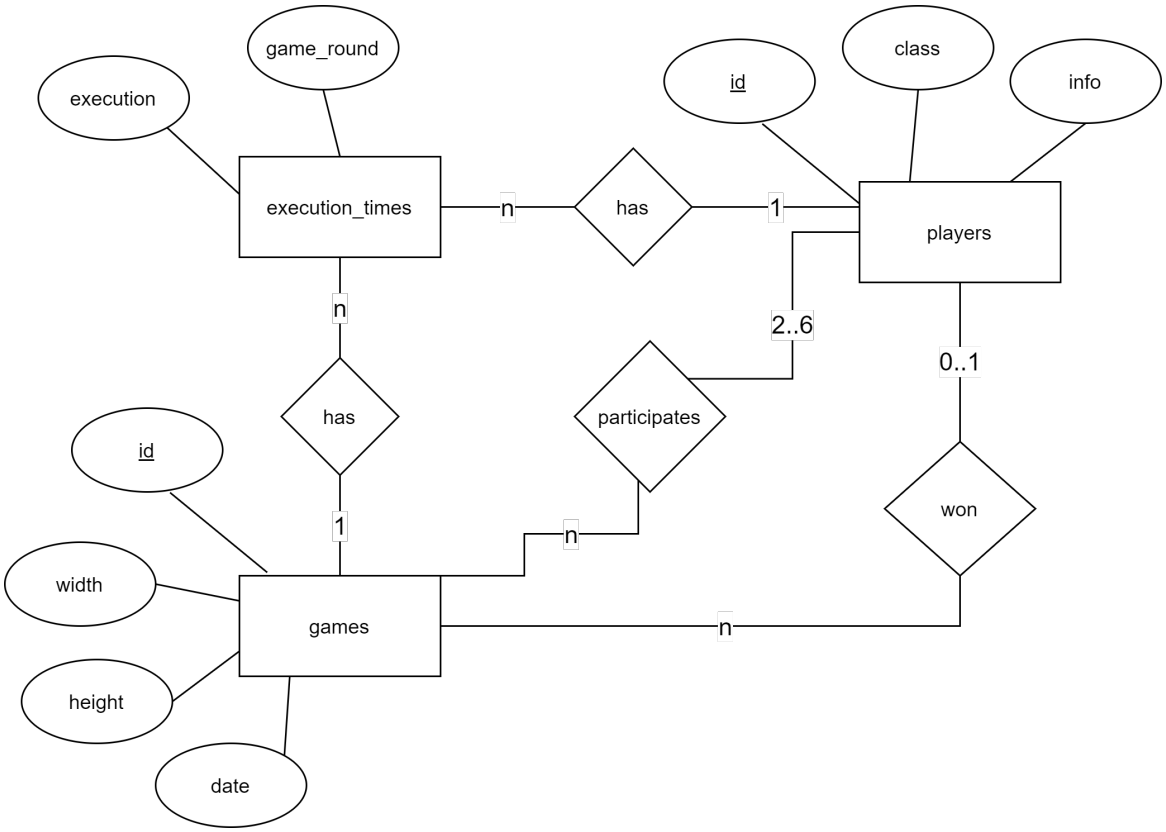

Abbildung 5.1: ER-Modell der Evaluations-DB

5.2.2 Überführen des ER-Modells in ein relationales Datenbank-Modell

Das ER-Modell wurde anschließend in ein relationales Modell überführt, welches die tatsächlichen Relationen in der Datenbank darstellt. Dazu wurden Primärschlüssel festgelegt und über Fremdschlüssel Verknüpfungen zwischen den Relationen hergestellt. Bei der Modellierung wurde darauf geachtet, die dritte Normalform der DB-Normalisierung einzuhalten, um Anomalien und Redundanzen zu verhindern. [25] In *Abbildung 5.2 (Relationales Datenbankschema der Evaluations-Datenbank)* ist das entworfene relationale Modell zu sehen.

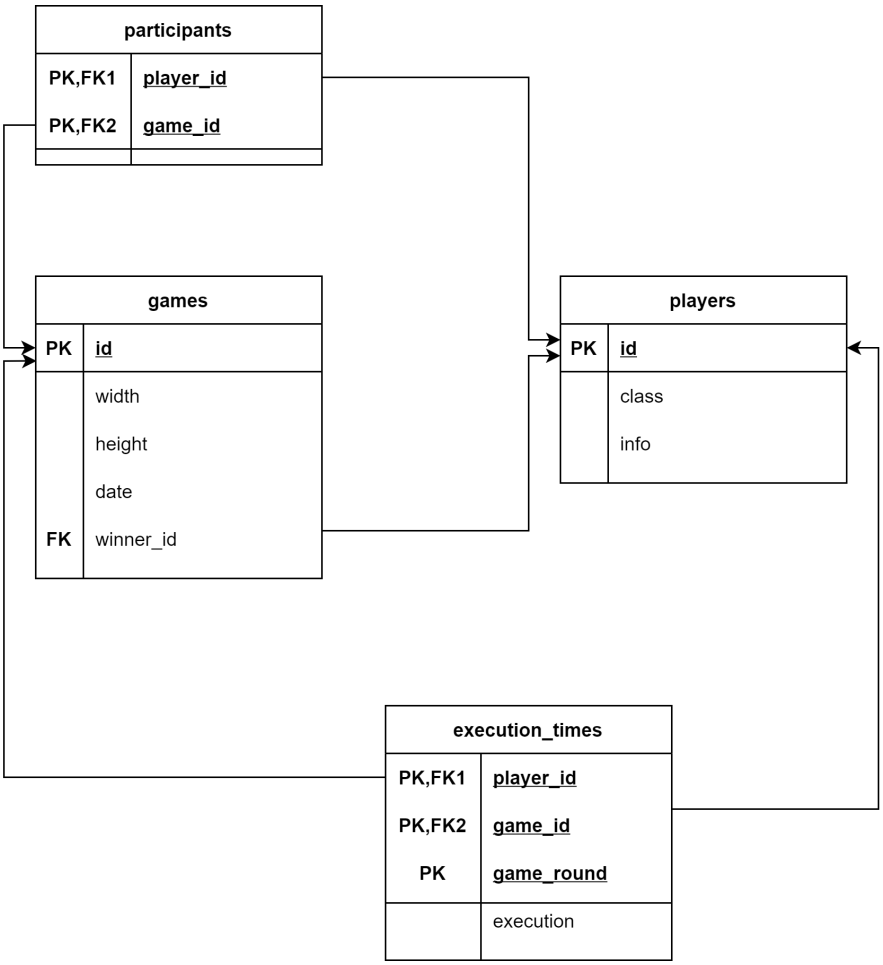

Abbildung 5.2: Relationales Datenbankschema der Evaluations-Datenbank

5.3 Ergebnis der Evaluation

Die Evaluation wurde in zwei Etappen durchgeführt. Für den ersten Teil der Evaluation wurden alle implementierten KIs in verschiedenen Konfigurationen für die Simulation betrachtet. Aufbauend auf diesen Ergebnissen wurde eine zweite Simulation gestartet, in der nur die besten KIs berücksichtigt wurden.

5.3.1 Ergebnisse der ersten Evaluation

Mit dem oben beschriebenen `AIEvaluationController` wurden in einem Zeitraum von 16 Tagen insgesamt 1000 Spiele mit zufälligen KI-Konstellationen simuliert und dabei insgesamt 638685 Spielzüge berechnet. Um aus den gespeicherten Daten der DB die besten KIs herauszufiltern, wurde insbesondere jeweils die Siegquote in den Spielen betrachtet, an denen sie teilgenommen haben. Mit dem in *Anhang A.5: (Auswertung der besten KI-Klasse)* abgebildetem SQL-Statement wird eine solche Quote gruppiert nach der KI-Klasse berechnet. Das Ergebnis kann der *Tabelle 5.1 (Auswertung der besten KI-Klasse)* entnommen werden.

Klasse	**Wins**	**Plays**	**Gewinn-rate (%)**	**Max Zeit (Sek)**	**Avg Zeit (Sek)**	**Avg Zeit oD (Sek)**	**Deadline eingehal-ten (%)**
NotKillingItselfAI	149	491	30.35	1.5989	0.3621	0.3568	100
PathfindingAI	303	1000	30.3	60.001	4.3782	2.1739	96.19
PathfindingSearch-TreeAI	297	1000	29.7	60.002	9.3117	2.6884	88.44
SearchTreePath-findingAI	238	1000	23.8	60.002	6.6962	2.24	92.28
SearchTreeAI	6	733	0.82	60.002	4.2471	0.3472	93.46
RandomAI	0	257	0	0.0251	0.0078	0.0078	100

Tabelle 5.1: Auswertung der besten KI-Klasse

Etwas überraschend hat die `NotKillingItselfAI` demnach am besten abgeschnitten, obwohl andere Taktiken wie bereits erläutert teilweise hierauf aufbauen und die Ermittlung der besten Aktion um zusätzliche Berechnungen erweitern. Um ein differenzierteres Ergebnis zu erhalten, wurde die Abfrage aus *Anhang A.6: (Auswertung der besten KI-Konfiguration)* ausgeführt, welche nicht nur die Art der KI, sondern zusätzlich auch ihre jeweiligen Konfigurationsparameter betrachtet. In *Tabelle 5.2 (Auswertung der besten KI-Konfiguration)* werden die ersten Einträge reduziert auf die relevantesten Spalten abgebildet.
Hierbei wird deutlich, dass die `NotKillingItselfAI` schlechter abschneidet als es das erste SQL-Statement vermuten ließ. Das liegt daran, dass die anderen Implementierungen abhängig von der Konfiguration deutlich besser, aber eben auch deutlich schlechter abschneiden können. Dies kann an den komplexeren Berechnungen liegen, die je nach Zustand des Spiels nicht innerhalb der Deadline abgeschlossen werden können und dann eine schlechte Bewegung durch eine fehlende Rückmeldung an den Server zur Folge haben.

Weitere Erkenntnisse, die aus dem Ergebnis gezogen werden konnten, waren u. a. , dass die Anzahl der Felder, die von der `PathfindingAI` geprüft werden, sehr deutlich unter der Anzahl aller Felder liegen kann. Die beste Konfiguration prüft nur zu 50 Feldern die Pfade, während ein Spielfeld aus mindestens $30 * 30 = 900$ Feldern besteht. Außerdem ist zu sehen, dass besonders KIs mit einer niedrigen Maximal-Geschwindigkeit von 1 oder 2 gute Ergebnisse erzielen.

Klasse	**Info**	**Wins**	**Plays**	**Gewinnrate (%)**
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 50, depth=2, distance_to_check =30	12	18	66.67
SearchTreePathfindingAI	max_speed=1, count_paths_to_ check=25, depth=2, distance_ to_check=20	17	26	65.38
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 25, depth=2, distance_to_check =10	10	16	62.50
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 75, depth=3, distance_to_check =10	7	12	58.33
PathfindingSearchTreeAI	max_speed=2, paths_tolerance =0.75, count_paths_to_check= 75, depth=3, distance_to_check =20	8	15	53.33
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 75, depth=2, distance_to_check =10	9	17	52.94
PathfindingAI	max_speed=1, count_paths_to_ check=50	63	120	52.50
...	...	...	...	...

Tabelle 5.2: Auswertung der besten KI-Konfiguration

5.3.2 Ergebnisse der zweite Evaluation

Anschließend wurden die besten 25 Konfigurationen, die wir als Ergebnis in der *Tabelle 5.2 (Auswertung der besten KI-Konfiguration)* erhalten haben, hart in einem Array codiert. Die Spielerstellung wurde im Vergleich zur ersten Evaluation dahingehend angepasst, dass nicht mehr zufällige KIs für die Spieler gewählt wurden, sondern die zufällige Auswahl auf die Einträge dieses Arrays beschränkt wurde. Im zweiten Teil sollte sichergestellt werden, dass nur noch starke KIs gegeneinander antreten und dadurch eine bessere Vergleichbarkeit der Siegquoten möglich ist.

Am Ende von 500 simulierten Spielen mit 508407 berechneten Spielzügen wurde erneut die SQL-Abfrage aus *Anhang A.6: (Auswertung der besten KI-Konfiguration)* verwendet, um insbesondere anhand des Verhältnisses von gewonnen zu gespielten Spielen den Gewinner der zweiten Evaluation zu ermitteln. Das Ergebnis der Evaluation wird in *Tabelle 5.3 (Ergebnis der zweiten Evaluation)* gezeigt. Es wird deutlich, dass die Kombination aus der `PathfindingAI` und der `SearchTreeAI`, die in *Kapitel 3.2.6 (PathfindingSearchTreeAI)* erläutert wurde, im

Vergleich zu unseren anderen Lösungsansätzen die besten Ergebnisse in den Spielen erzielt, sodass diese schlussendlich auch für die Abgabe des Projektes ausgewählt wurde.

Klasse	**Info**	**Wins**	**Plays**	**Gewinnrate (%)**
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 50, depth=3, distance_to_check =10	33	99	33.33
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 50, depth=2, distance_to_check =20	31	96	32.29
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 50, depth=2, distance_to_check =30	25	88	28.41
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 75, depth=2, distance_to_check =10	27	99	27.27
SearchTreePathfindingAI	max_speed=1, count_paths_to_ check=75, depth=2, distance_ to_check=10	25	93	26.88
PathfindingSearchTreeAI	max_speed=1, paths_tolerance =0.75, count_paths_to_check= 75, depth=3, distance_to_check =10	23	89	25.84
SearchTreePathfindingAI	max_speed=1, count_paths_to_ check=50, depth=2, distance_ to_check=10	23	90	25.56
...	...	...	...	...

Tabelle 5.3: Ergebnis der zweiten Evaluation

Kapitel 6

Software-Qualität

Um sicherzustellen, dass die Software erwartungsgemäß funktioniert und eine Weiterentwicklung vereinfacht wird, wurden verschiedene Aspekte berücksichtigt, die für eine verbesserte Qualität der Software sorgen. Die Wartbarkeit der Anwendung wird in *Kapitel 7.1 (Erweiterbarkeit des Codes)* noch genauer betrachtet.

6.1 Architektur der Software

Beim Aufbau der Software haben wir uns für eine dreischichtige Architektur und den Einsatz des Model-View-Controller (MVC)-Patterns [26] entschieden. Die Architektur ist in *Abbildung 6.1 (Schichtenarchitektur der Software)* dargestellt und beinhaltet neben der Schicht zur Anzeige des Spiels in einer Oberfläche die Logik-Schicht, in der u. a. implementiert wurde, wie ein Spiel abläuft, wie die Daten zur Kommunikation mit dem Server übersetzt werden können und auch wie die KIs funktionieren sollen. Im Prinzip wird hier das Zusammenspiel der Klassen aus dem Modell umgesetzt. Die Modell-Schicht stellt die letzte und unterste Schicht dar und bietet die Klassen zur Datenhaltung und deren interne Logik an.

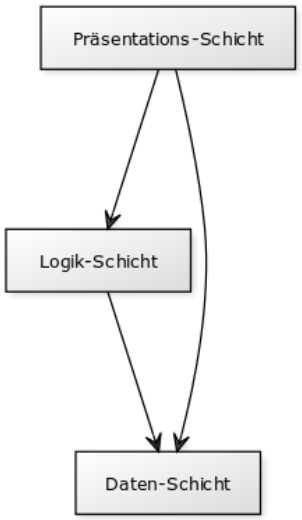

Abbildung 6.1: Schichtenarchitektur der Software

Ein wichtiger Aspekt ist hierbei, dass ein Zugriff nur auf eine untere Schicht erlaubt ist. Somit soll verhindert werden, dass die Logik-Schicht bspw. abhängig von der Art der Darstellung ist.

6.1.1 Package-Struktur

Aufbauend darauf haben wir das MVC-Pattern umgesetzt und dies in dem Aufbau der Package-Struktur verdeutlicht, welche auch in *Abbildung 6.2 (Package-Struktur des Projekts)* zu sehen ist. Das View-Package stellt in der Schichten-Architektur die Präsentations-Schicht dar und beinhaltet Klassen, die sich um die Anzeige kümmern. Dabei ist, wie durch das Schichtenmodell erlaubt wird, ein Zugriff auf das Modell möglich. Die Logik-Schicht wird zum einen Teil durch das Controller-Package realisiert, in welchem insbesondere die Verknüpfung zwischen der Geschäftslogik und der Oberfläche geschieht. Die eigentliche Logik befindet sich dann im Service-Package, welches sich in der gleichen Schicht befindet. Allerdings soll ein Zugriff von Services zu Controllern unterbunden werden. Zuletzt bildet das Model-Package die oben beschriebene Modell-Schicht ab.

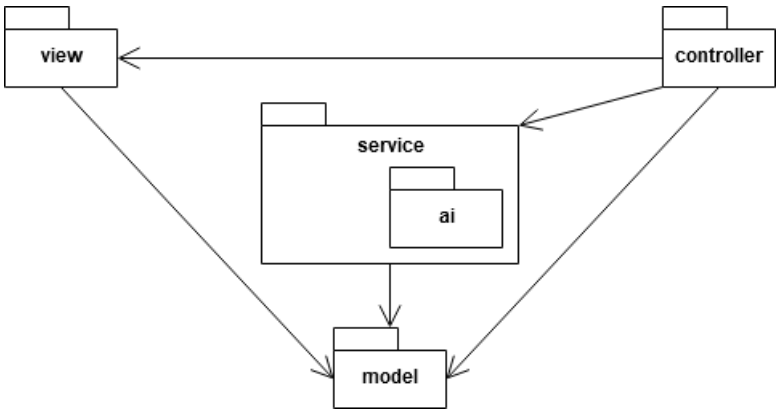

Abbildung 6.2: Package-Struktur des Projekts

6.2 Automatisierte Tests

Zur Sicherstellung der Korrektheit der Software wurden automatisierte Unit-Tests [27] implementiert. Diese dienen während der Implementierung dazu, ein gewünschtes Szenario abzubilden und anschließend den Code so zu implementieren, dass der Test erfolgreich durchläuft. Anschließend ist eine Refaktorisierung möglich, d. h. der Code wird verändert, die Funktionalität soll aber gleich bleiben. Ein Beispiel dafür kann eine Verbesserung der Lesbarkeit oder Wartbarkeit durch eine Auslagerung einer großen in mehrere kleinere Methoden sein. Diesen testgetriebenen Ansatz [28] haben wir zwar nicht für alle Komponenten verwendet, an einigen Stellen war dies aber hilfreich.

Ein weiterer Vorteil von automatisierten Tests ist, dass bei einer Weiterentwicklung sichergestellt werden kann, dass durch eine Änderung keine ungewollten Nebeneffekte eintreten, sondern

der Entwickler sicher sein kann, dass bei erfolgreich durchgelaufenen Tests alles weiterhin wie gewünscht funktioniert.

Ein Beispiel für einen Test ist in *Listing 6.1 (Beispiel für einen Unit-Test)* zu sehen. Grundsätzlich folgt der Ausbau eines Testfalls den drei Schritten Arrange, Act und Assert. Im ersten Schritt wird der Testfall also vorbereitet, anschließend der zu testende Use-Case aufgerufen und abschließend geprüft, ob das gewünschte Resultat erzielt wurde. [29]

```
def test_convert_json_to_running_game(self):
    json = tests.read_test_file("service/game.json")
    player1 = Player(1, 2, 2, Direction.up, 1, True, "")
    player2 = Player(2, 1, 0, Direction.down, 3, True, "")
    player3 = Player(3, 4, 3, Direction.left, 2, False, "Name 3")
    players = [player1, player2, player3]
    cells = [
        [Cell(), Cell([player2]), Cell(),          Cell(), Cell()],
        [Cell(), Cell(),          Cell(),          Cell(), Cell()],
        [Cell(), Cell([player1]), Cell([player1]), Cell(), Cell()],
        [Cell(), Cell(),          Cell(),          Cell(), Cell([player3])]
    ]
    time = datetime(2020, 10, 1, 12, 5, 13, 0, timezone.utc)
    expected = Game(5, 4, cells, players, 2, True, time)

    result = self.sut.load(json)

    self.assertEqual(expected, result)
```

Listing 6.1: Beispiel für einen Unit-Test

6.3 Coding Conventions

Um sicherzustellen, dass der Code unabhängig vom Autor einheitlich und wartbar aufgebaut ist, wurde auf die Einhaltung von Konventionen beim Schreiben von Quell-Code beachtet. [30] Viele solcher Vorgaben werden bereits durch die Programmiersprache oder von etablierten Institutionen bereitgestellt und können daher automatisch überprüft werden. [31] Dabei war Pylint [32] als Tool hilfreich, das sich als Plugin direkt in PyCharm-IDE von JetBrains [33] integrieren ließ und somit bereits bei der Arbeit am Code unmittelbar Verbesserungsvorschläge macht. Daraus konnten wir einen großen Mehrwert ziehen, da wir wie eingangs beschrieben noch keine Erfahrung mit Python hatten und durch diese Anmerkungen einige Dinge lernen konnten. Außerdem wurde in einem automatischen Prozess, der nachfolgend noch in *Kapitel 7.3 (Nutzung von Github Actions)* beschrieben wird, das Tool Flake8 [34] eingesetzt, sodass eine Überprüfung durch zwei verschiedene Programme erfolgte.

Darüber hinaus haben wir auch eigene Punkte abgesprochen. Dazu zählt bspw., dass wir bei Parametern und dem Rückgabewert an Methoden die jeweilige Typangabe ergänzt haben,

obwohl dies in Python optional ist. Der Vorteil, den wir dadurch bemerkt haben, war eine bessere Unterstützung bei der Autovervollständigung und Anzeige von möglichen Fehlern durch die IDE. Darüber hinaus wurde insbesondere für die Dokumentation des Codes mithilfe von Docstrings der Python-Styleguide von Google [31] verwendet. Dieser beschreibt unter anderem, dass alle öffentlich sichtbaren und nicht trivialen Methoden sowie alle Klassen dokumentiert werden sollen und gibt vor, wie die Formatierung der Dokumentation auszusehen hat. Durch den oben beschriebenen Einsatz von Pylint wurden automatisiert viele Konventionen aus diesem Styleguide überprüft, die über die Code-Dokumentation hinausgehen.

Kapitel 7

Weiterentwicklung

Wichtig war bei dem Projekt auch, stets sicherzustellen, dass das Projekt möglichst leicht weiterentwickelt werden kann. Was wir dazu unternommen haben, soll in diesem Kapitel erläutert werden. Dazu betrachten wir sowohl die Implementierung des Quell-Codes als auch den Workflow, der zur Umsetzung dieses Projektes verwendet wurde.

7.1 Erweiterbarkeit des Codes

An vielen Stellen im Code haben wir durch die Nutzung allgemeiner Oberklassen eine leichte Austauschbarkeit gewährleistet. Die Verwendung von abstrakten Oberklassen durch Nutzung der `abc`-Python-Bibliothek [35] war notwendig, da Python als Programmiersprache nicht unmittelbar Interfaces oder abstrakte Klassen anbietet.

Durch diese Entscheidung war es mithilfe von Dependency Injection möglich, zentral in der Datei `main.py` festzulegen, welche spezifischen Implementierungen der allgemeinen Oberklassen verwendet werden sollen. „Dependency Injection überträgt die Verantwortung für das Erzeugen und die Verknüpfung von Objekten an eine eigenständige Komponente, wie beispielsweise ein extern konfigurierbares Framework. Dadurch wird der Code des Objektes unabhängiger von seiner Umgebung. Das kann Abhängigkeiten von konkreten Klassen beim Kompilieren vermeiden und erleichtert besonders die Erstellung von Unit-Tests.“ [36] Angewendet wurde dieses Vorgehen an oberster Stelle bei den `Controllern` und der zu verwendenden Oberfläche, aber auch beim Konvertieren zwischen einer String-Repräsentation und dem Modell im `DataLoader` und `DataWriter` ließe sich prinzipiell sehr leicht ein Umstieg von JSON auf ein beliebiges anderes Format ermöglichen. Auch ein Austausch der zu verwendenden KI-Klasse lässt sich sehr einfach konfigurieren.

7.2 Einsatz von PRs im Git-Workflow

Wir haben uns dazu entschieden, als Versionsverwaltungs-Tool Git [37] einzusetzen und Github als Plattform zu benutzen. Github ermöglicht die Konfiguration, das Pushen auf den Haupt-

Branch, welcher in unserem Fall der `main`-Branch war, zu unterbinden. [38] So kann kein Code durch einen versehentlichen Push den aktuellen Produktiv-Code in seiner Funktionalität stören. Stattdessen können Änderungen in diesen Branch nur durch Pull Request (PR)s in den Haupt-Branch gemergt werden.

Es gab somit für jede logische Einheit für Code-Änderungen einen neuen Branch, auf dem diese entwickelt und getestet wurden, bevor eine Überführung in `main` möglich war. [39] Es wäre möglich gewesen, die Aufgaben als sogenannte Issues zu pflegen und jeden Branch mit einem Issue zu verknüpfen, allerdings haben wir dies nicht genutzt, da wir uns durch regelmäßige Absprachen auch so einen Überblick über die als Nächstes zu erledigenden Aufgaben ohne ein Ticket-System machen konnten.

Für PRs wurden dann Kriterien festgelegt, die erfüllt sein müssen, um einen Merge durchführen zu können. Es wird automatisch von Github kontrolliert, ob mögliche Konflikte beim Mergen auftreten können. Falls dies so sein sollte, ist es notwendig, diese zuerst manuell zu beheben. Weiterhin haben wir eingestellt, dass mindestens ein Code Review notwendig ist. Da wir als Zweiergruppe an dem Projekt gearbeitet haben, konnten wir so sicherstellen, dass jeder zu jeder Zeit einen Überblick über den aktuellen Stand hat und jeder Code einem Review unterzogen wurde. Solche Code Reviews haben bspw. im Ansatz des Extreme Programming eine zentrale Bedeutung. [40]

7.3 Nutzung von Github Actions

Als letzter Aspekt, der einen Merge potenziell verhindern konnte, wurde eine sogenannte Github Action bei dem Öffnen eines PRs und bei dem Pushen auf einen Branch mit einem bereits geöffneten PR ausgeführt.

Eine solche Aktion wird nicht manuell in den Einstellungen hinterlegt, sondern nach dem Configuration-as-Code-Paradigma in einer Text-Datei verwaltet. Dazu muss lediglich in einem Unterordner `./.github/workflows` ausgehend vom Hauptverzeichnis des Repositorys eine Datei im YAML-Format abgelegt werden. [41] [42].

Die für unser Projekt verwendete Konfiguration wird in *Anhang A.7: (YAML-Konfiguration der Github Action)* gezeigt. Hier läuft nach der Installation aller notwendigen Packages eine Kompilierung und Code-Analyse über den Quellcode gefahren, wobei dieser auf Probleme hin untersucht wird. Im Anschluss werden alle Tests ausgeführt. Bei Kompilier-Fehlern oder fehlschlagenden Tests schlägt auch die Ausführung fehl und verhindert einen Merge.

So wird durch Continuous Integration automatisch eine Kontrolle vollzogen, die sicherstellt, dass durch Änderungen keine bestehende Logik beschädigt wird. Dies gab uns als Entwickler eine zusätzliche Sicherheit und verringerte die Risiken eines Merges.

7.4 Entwickler-Dokumentation

Um für die Weiterentwicklung einen Einstieg in das Projekt zu ermöglichen, wurden wie in *Kapitel 6.3 (Coding Conventions)* beschrieben Docstrings verwendet, um den Quellcode zu dokumentieren. Für eine bessere Übersichtlichkeit wurde das Tool PDoc eingesetzt, welches aus den Kommentaren im Code automatisiert HTML-Seiten generieren kann. [43] Diese Dateien wurden dann in ein eigenes Repository auf Github gepusht, für welches Github Pages [44] aktiviert wurden. Hiermit ist es möglich, die Entwickler-Dokumentation unter folgender URL bereitzustellen: `https://jonashellmann.github.io/informaticup21-team-chillow-doc/`.

Kapitel 8

Fazit

8.1 Einschätzung unserer Lösung

Insgesamt sind wir mit unserem Ergebnis nach Abschluss des Projektes sehr zufrieden. Es wurde eine Software entwickelt, die leicht erweitert werden kann und deren Schichten voneinander sauber getrennt sind. Bei der Entwicklung wurde auf die Berücksichtigung bewährter Standards wie den Einsatz von Versionsverwaltung oder Continuous Integration geachtet, es konnte vorher in der Universität erlerntes Wissen eingebracht werden und neue Erkenntnisse gewonnen werden.

Im Hinblick auf die Stärke der abgegebenen KI sind wir ebenfalls davon überzeugt, eine solide Leistung erzielen zu können. Mit der Kombinationen verschiedener Ansätze ist eine Vorausberechnung gegnerischer Züge und entsprechendes Reagieren darauf genauso möglich wie das Verhindern von Sackgassen durch das Finden von Pfaden auf dem Spielfeld zu beliebig ausgewählten Punkten. Allerdings wurde in einem internen Turnier der Mannschaften der Universität Oldenburg auch deutlich, dass andere Taktiken möglicherweise trotzdem stärker sein könnten und unsere im Vergleich lediglich mittelmäßig abgeschnitten hat. Unsere KI wurde entwickelt, möglichst sichere Entscheidungen für die nächsten x Spielzüge zu treffen, was durch aggressive KIs im Turnier ausgenutzt werden konnte. In dem Turnier wurden jedoch nur drei Spiele ausgeführt, sodass die dort erzielte Gewinnquote nicht repräsentativ ist. Des Weiteren waren wir zu der Zeit noch nicht am Ende der Entwicklung, weswegen wir trotzdem optimistisch sind und auf das Ergebnis vom InformatiCup sehr gespannt sind.

8.1.1 Umsetzung von optionalen Erweiterungen

Der geforderte Umfang der Aufgabenstellung [1] wurde in unserem Lösungsvorschlag an einigen Stellen durch zusätzliche Erweiterungen ergänzt. Diese wurden in der Dokumentation in vorangegangenen Abschnitten bereits erläutert, sollen an dieser Stelle aber noch einmal zusammenfassend erwähnt werden.

Es wurden zwei verschiedene UIs für eine konsolenbasierte Ausgabe und eine grafische Darstellung des Spiels in einer Desktop-Anwendung implementiert, auf deren Entwicklung in *Kapitel 4.4.2 (Darstellung des Spiels in der Konsole)* und *Kapitel 4.4.3 (Nutzung von PyGame als grafische Oberfläche)* eingegangen wurde. Diese Oberflächen erlauben neben der automatischen Berechnung der nächsten Aktion durch eine KI mithilfe der im *Kapitel 9 (Benutzerhandbuch)* genannten Konfigurationen auch die Interaktion eines menschlichen Spielers mittels Benutzereingaben.

Zusätzlich zu der geforderten Aufgabe, „ein Programm zu schreiben, das selbstständig (also ohne menschliche Unterstützung) das Spiel spe_ed spielen kann“ [1], welches lediglich als Client fungiert und mit einem Server kommuniziert, wurde die gesamte Logik des Servers wie in *Kapitel 4.3 (Implementierung des Offline-Spiels)* beschrieben nachgebildet. Dadurch wird ermöglicht, in den bereitgestellten UIs auch offline Spiele der KIs durchzuführen.

Auf diese Möglichkeit wurde aufgebaut, indem eine Erweiterung bereitgestellt wird, die es ermöglicht, eine beliebige Anzahl an Spielen mit zufällig generierten Spieleinstellungen zu erzeugen. Hierauf wurde in *Kapitel 5.1 (Vorgehen zur Auswahl der besten Strategie)* eingegangen. Um auf die Ergebnisse dieser Simulationen zugreifen zu können, wurde ein Entity-Relationship-Modell entworfen, das in ein relationales Datenbank-Modell überführt wurde. Auf Grundlage dieses Modells wird bei der Simulation eine SQLite3-Datenbank verwendet und befüllt, aus der mithilfe von SQL-Statements Informationen zu der Stärke einer KI gewonnen werden können.

8.2 Reflexion des Wettbewerbs

Der Wettbewerb war aus unserer Sicht eine sehr gute Möglichkeit, sich im Rahmen des Studiums mit interessanten Themen auseinanderzusetzen und diese teilweise auch praktisch umzusetzen. Dazu zählen u. a. Python als Programmiersprache, eine Einarbeitung in maschinelles Lernen und die Konzeption einer guten, prädiktiven Strategie für ein Spiel mit einem grundsätzlich einfach zu verstehendem Regelwerk, das allerdings bei der Vorhersage von Spielzügen durch die exponentiell schnell steigende Anzahl von Möglichkeiten sehr komplex wird.

Auch der kompetitive Gedanke dieser Ausgabe des InformatiCups, dass einen Lösungsvorschlag entwickelt werden sollte, der mit den anderen eingereichten Projekten mithalten bzw. gegen diese gewinnen kann, war sehr interessant.

Kapitel 9

Benutzerhandbuch

Das Benutzerhandbuch soll eine Anleitung darstellen, wie die eingereichte Lösung installiert und ausgeführt werden kann. Dafür wird zwischen der Verwendung von Docker oder einer manuellen Installation unterschieden.

9.1 Installation

Zur Verwendung muss das Projekt lokal heruntergeladen werden, entweder durch Klonen des Repositorys oder durch einen Download als ZIP-Datei. Das Projekt kann unter folgendem Link eingesehen werden: `https://github.com/jonashellmann/informaticup21-team-chillow`

9.1.1 Docker

Falls Sie Docker auf Ihrem Rechner installiert haben, lässt sich für das Projekt aufgrund des vorhandenen `Dockerfile`s mit folgendem Befehl ein neuer Container erstellen:

```
docker build -t informaticup21-team-chillow .
```

Dieser Container kann mit folgendem Befehl gestartet werden, bei dem die URL zum spe_ed-Server, der API-Key und die URL zur Abfrage der Server-Zeit entsprechend angepasst werden müssen, wobei TIME_URL optional ist:

```
docker run -e URL=SERVER_URL -e KEY=API_KEY \
    -e TIME_URL=TIME_URL informaticup21-team-chillow
```

In der Konsole des Docker-Containers lässt sich dann der Spiel-Verlauf nachvollziehen.

9.1.2 Manuelle Installation

Neben der Docker-Installation kann das Projekt auch eigenständig gebaut werden. Dafür ist erforderlich, dass neben Python in der Version ¿=3.8 auch Poetry als Build-Tool installiert ist. Die erforderlichen Abhängigkeiten lassen sich anschließend mittels `poetry install` installieren.

Um ein Spiel mit einer simplen grafischen Oberfläche zu starten, in dem gegen die implementierte KI gespielt werden kann, genügt der Befehl `python ./main.py`. Wenn gegen andere KI-Konstellationen gespielt werden soll, muss dies im `OfflineController` bei der Erstellung des initialen Spiels manuell angepasst werden.
Um ein Online-Spiel der KI auf dem Server zu starten, müssen folgende Umgebungsvariablen verwendet werden, die im Docker-Container automatisch gesetzt bzw. als Parameter übergeben werden:

- `URL=[SERVER_URL]`
- `KEY=[API_KEY]`
- `TIME_URL=[TIME_URL]` (optional)

Mittels dem Kommandozeilen-Parameter `--deactivate-pygame` kann entschieden werden, ob eine grafische Oberfläche benutzt werden soll oder die Ausgabe wie im Docker-Container über die Konsole erfolgt. Wenn die Python-Bibliothek PyGame nicht vorhanden ist, muss dieser Wert entweder auf `False` gesetzt werden oder es ist eine manuelle Installation von PyGame bspw. mittels Pip notwendig.

9.2 Benutzung

Wenn das Programm im Online-Modus gestartet wird, ist keine weitere Eingabe des Benutzers zu tätigen. Sobald der Server das Spiel startet, kann entweder auf der Konsole oder in der grafischen Oberfläche der Spielverlauf nachvollzogen werden. Hier muss der Parameter `--play-online` auf `TRUE` gesetzt werden.
Bei einer Ausführung im Offline-Modus wird – je nach manueller Anpassung im `Offline-Controller` - auf eine Eingabe von keinem, einem oder mehreren Spielern gewartet, bis die nächste Runde des Spiels gestartet wird. Der *Tabelle 9.1 (Steuerung der Oberflächen)* kann entnommen werden, mit welchen Eingaben eine Aktion ausgeführt werden kann. Der Parameter `--play-online` muss für diesen Modus auf `FALSE` gesetzt werden.

	turn_right	**turn_left**	**speed_up**	**slow_down**	**change_nothing**
Konsole	r	l	u	d	n
Grafische Oberfläche	→	←	↑	↓	Leertaste

Tabelle 9.1: Steuerung der Oberflächen

Darüber hinaus ist eine Offline-Simulation mehrerer Spiele hintereinander möglich, in dem KIs mit zufälliger Konfiguration auf einem Spielfeld mit zufälliger Größe gegeneinander antreten, um die bestmögliche KI zu ermitteln. Dazu ist es notwendig, dass zusätzlich zum normalen Offline-Spiel dem Parameter `--ai-eval-runs` auf eine Zahl größer als Null gesetzt wird. Mit dem Parameter `--ai-eval-db-path` kann statt dem Standardwert auch individuell der Pfad zu

einer SQLite3-Datenbank festgelegt werden. Weiterhin steuert `--ai-eval-type`, welche Art der Evaluation ausgeführt werden soll. Bei Wert 1 werden alle KIs betrachtet und jeweils maximal eine zufällige Konfiguration von einer Klasse zu einem Spiel hinzugefügt. Bei Wert 2 hingegen sind die nach unserer Evaluation aus dem ersten Lauf heraus ermittelten KI-Konfigurationen hinterlegt und es werden nur aus diesen möglichen Konfigurationen KIs für ein Spiel ausgewählt. Andere Werte als 1 und 2 sind für diesen Parameter ungültig.

Anhang A

Anhang

A.1 Lösungsansatz

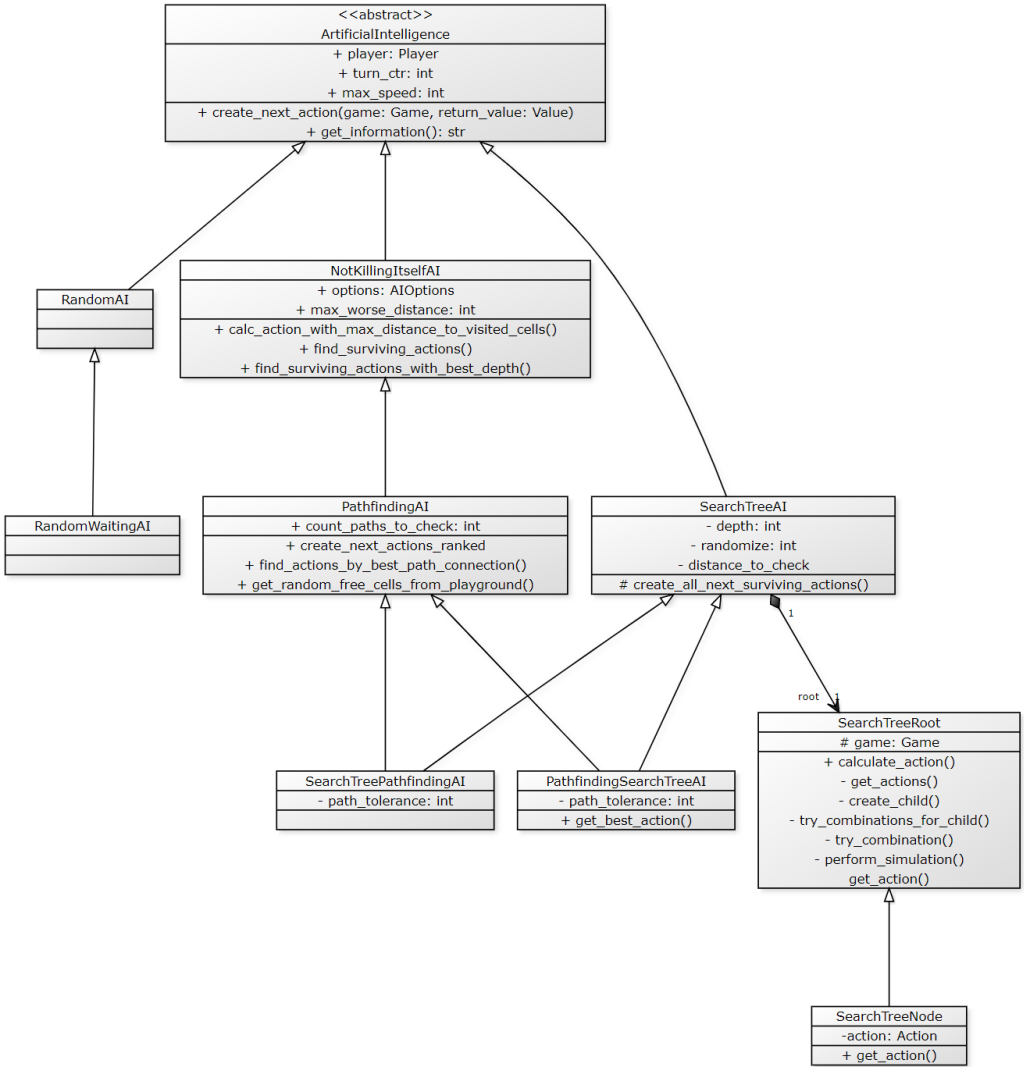

Abbildung A.1: UML-Klassendiagramm der KIs

A.2 Implementierung

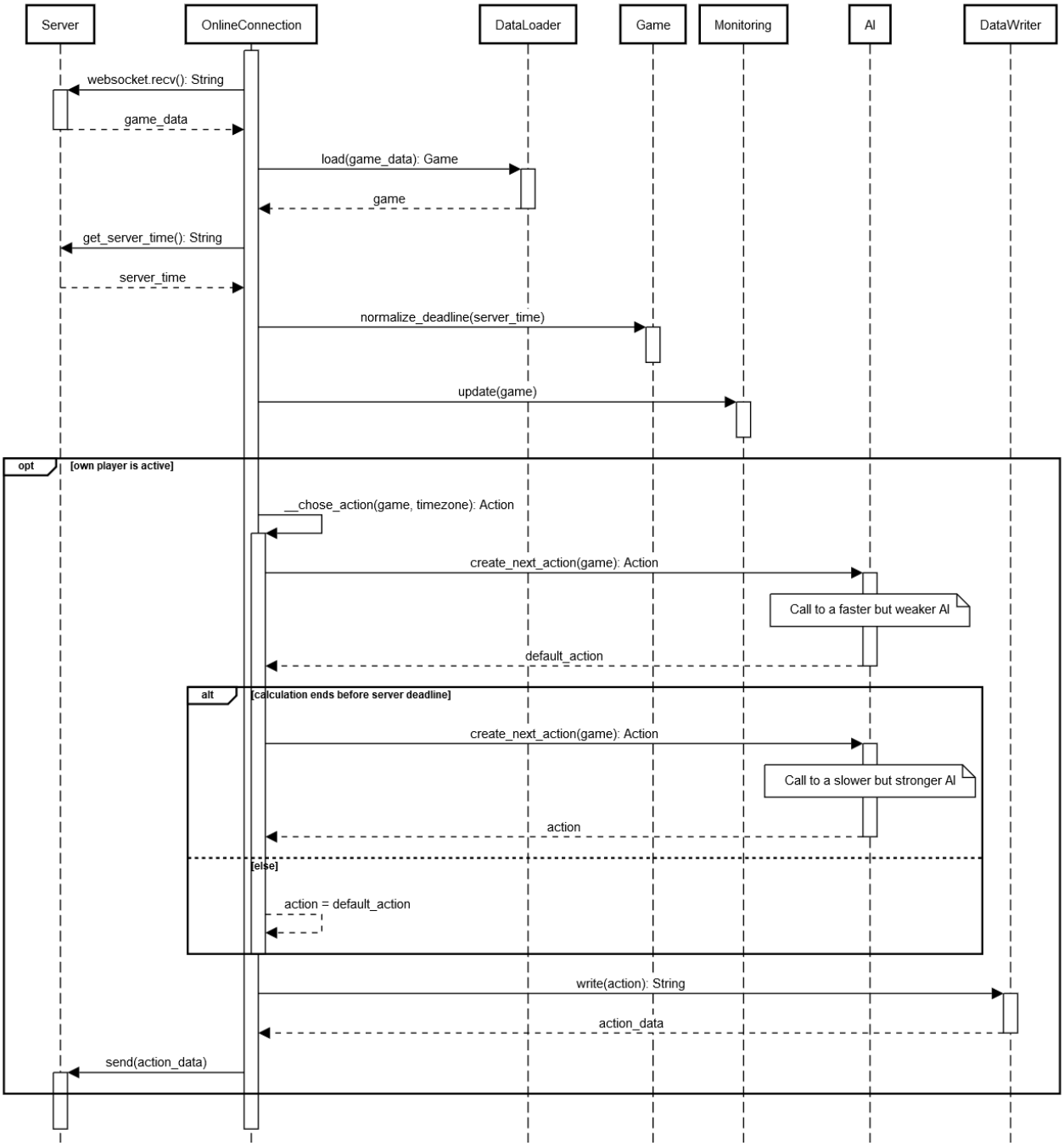

Abbildung A.2: Sequenzdiagramm zur Implementierung eines Spielzug

```
async def __play(self):
    async with websockets.connect(f"{self.url}?key={self.key}") as websocket:
        while True:
            game_data = await websocket.recv()
            game = self.data_loader.load(game_data)

            self.monitoring.update(game)

            if not game.running:
                break

            time_data = requests.get(self.time_url).text
            server_time = self.data_loader.read_server_time(time_data)
            own_time = datetime.now(server_time.tzinfo)
            game.normalize_deadline(server_time, own_time)

            if self.ai is None:
                self.ai = globals()[self.ai_class](game.you, *self.ai_params)
                self.default_ai = NotKillingItselfAI(game.you, [], 1, 0)

            if game.you.active:
                action = self.__choose_action(game, server_time.tzinfo)
                data_out = self.data_writer.write(action)
                await websocket.send(data_out)

def __choose_action(self, game: Game, timezone: datetime.tzinfo) -> Action:
    return_value = multiprocessing.Value('i')
    self.default_ai.create_next_action(game, return_value)

    own_time = datetime.now(timezone)
    seconds_for_calculation = (game.deadline - own_time).seconds

    process = multiprocessing.Process(target=OnlineController.call_ai,
                                      args=(self.ai, game, return_value,))
    process.start()
    process.join(seconds_for_calculation - 1)

    if process.is_alive():
        process.terminate()

    return Action.get_by_index(return_value.value)

@staticmethod
def call_ai(ai: ArtificialIntelligence, game: Game,
            return_value: multiprocessing.Value):
    ai.create_next_action(game, return_value)
```

Listing A.1: `__play()`-Methode des `OnlineControllers`

```
{
  "width": 10,
  "height": 8,
  "cells": [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
    [0, 0, 1, 0, 2, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 2, 0, 0, 3, 0, 0],
    [0, 0, 0, 0, 2, 0, 0, 3, 0, 0],
    [0, 0, 0, 0, 0, 0, 3, 3, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ],
  "players": {
    "1": {
      "x": 6,
      "y": 2,
      "direction": "right",
      "speed": 1,
      "active": true
    },
    "2": {
      "x": 4,
      "y": 3,
      "direction": "up",
      "speed": 1,
      "active": true
    },
    "3": {
      "x": 6,
      "y": 6,
      "direction": "left",
      "speed": 2,
      "active": true
    }
  },
  "you": 2,
  "running": true,
  "deadline": "2020-10-01T12:05:13Z"
}
```

Listing A.2: JSON-Repräsentation eines Spiel-Zustands

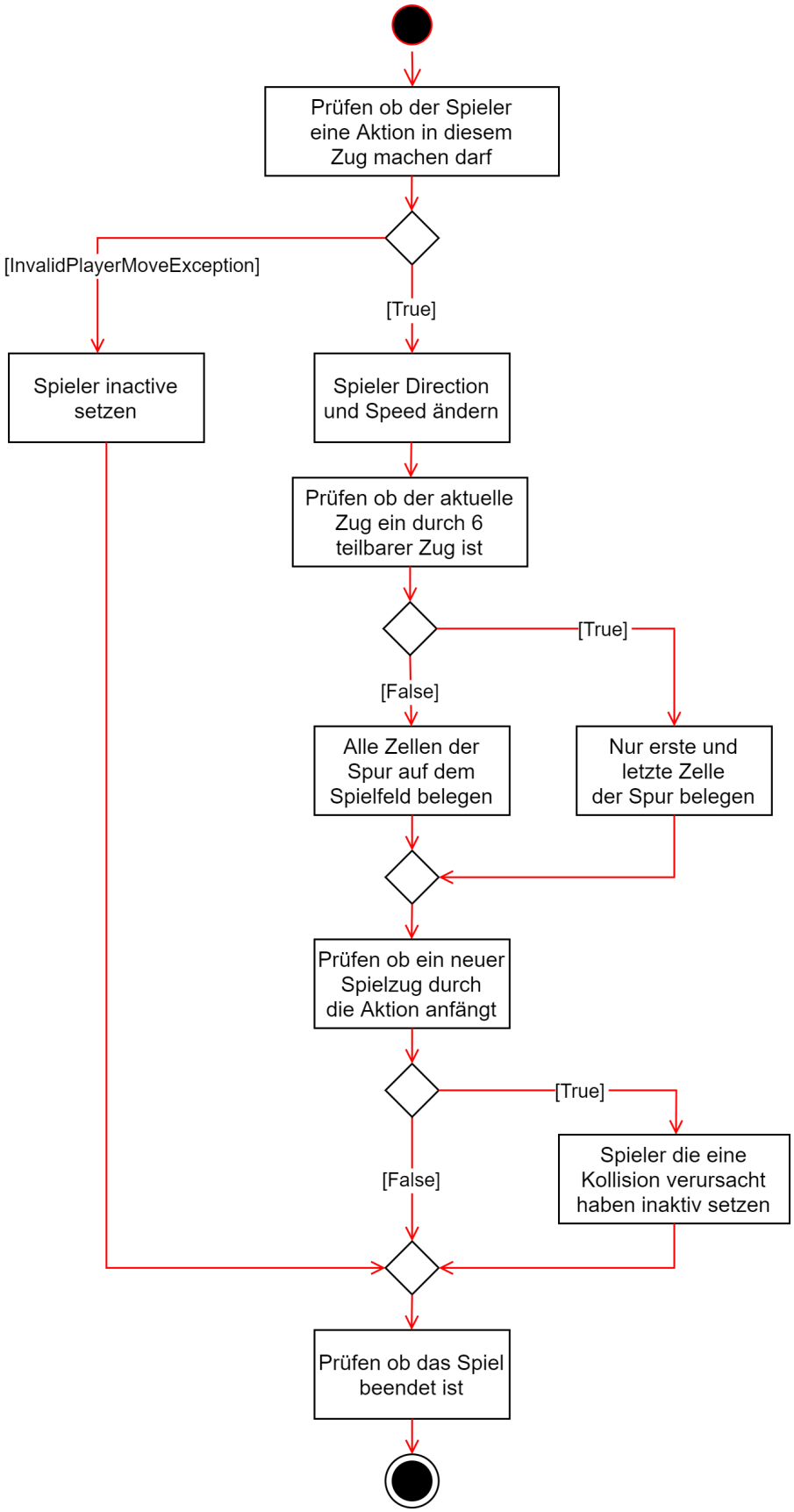

Abbildung A.3: Aktivitätsdiagramm zur Durchführung einer Spieler-Aktion durch den `GameService`

```
def get_and_visit_cells(self, player: Player, action: Action)
        -> List[Tuple[int, int]]:
    visited_cells = []
    GameService.change_player_status_by_action(player, action)
    horizontal_multiplier, vertical_multiplier = \
        GameService.get_horizontal_and_vertical_multiplier(player)

    for i in range(1, player.speed + 1):
        visited_cells.append((player.x + i * horizontal_multiplier,
                              player.y + i * vertical_multiplier))

    # Gap every sixth move, so take only first and last coordinate
    if self.turn.turn_ctr % 6 == 0 and len(visited_cells) > 1:
        visited_cells = [visited_cells[0], visited_cells[-1]]

    visited_cells_result = []
    for (x, y) in visited_cells:
        if x not in range(self.game.width) or y not in range(self.game.height):
            self.set_player_inactive(player)
            break
        player.x = x
        player.y = y
        visited_cells_result.append((x, y))
        if self.game.cells[y][x].players is None \
                or len(self.game.cells[y][x].players) == 0:
            self.game.cells[y][x].players = [player]
        else:
            self.game.cells[y][x].players.append(player)

    return visited_cells_result
```

Listing A.3: `get_and_visit_cells(player, action)`-Methode des `GameService`

```
def update(self, game: Game):
  """See base class."""
  if not self._interface_initialized:
    self._initialize_interface(game)

  if not game.running:
    player = game.get_winner()
    if player is None:
      print("No winner in game.")
    else:
      print("Winner: Player " + str(player.id) + " (" +
          player.name + "). Your player ID was " +
          str(game.you.id) + ".")

  self.__screen.fill((0, 0, 0))  # black background
  for row in range(game.height):
    for col in range(game.width):
      self.__pygame.draw.rect(self.__screen,
                  self.__get_player_color(game.cells[row][col]),
                  (col * self.RECT_SIZE + col,
                   row * self.RECT_SIZE + row,
                   self.RECT_SIZE,
                   self.RECT_SIZE))
      if game.cells[row][col].get_player_id() != 0:
        player = game.get_player_by_id(game.cells[row][col].get_player_id())
        if player.x == col and player.y == row:  # print head
          border_width = 2
          if player == game.you:
            border_width = 4  # head of the own player has a smaller dot
          self.__pygame.draw.rect(self.__screen,
                      self._player_colors[0],
                      (col * self.RECT_SIZE + col + border_width,
                       row * self.RECT_SIZE + row + border_width,
                       self.RECT_SIZE - (2 * border_width),
                       self.RECT_SIZE - (2 * border_width)))
  self.__pygame.display.update()
  self.__clock.tick(60)
```

Listing A.4: `update(game: Game)`-Methode der `GraphicalView`

A.3 Evaluation

```
SELECT
    class AS Klasse,
    Gewonnene_Spiele AS "Wins",
    Gespielte_Spiele AS "Plays",
    ROUND((CAST(Gewonnene_Spiele AS DOUBLE) * 100 /
           CAST(Gespielte_Spiele AS DOUBLE)), 2) AS "Gewinnrate (%)",
    Max_Ausfuerungszeit_in_Sek AS "Max Zeit (Sek)",
    Avg_Ausfuerungszeit_in_Sek AS "Avg Zeit (Sek)",
    Avg_Ausfuerungszeit_in_Sek_ohne_deadline "Avg Zeit oD (Sek)",
    ROUND((CAST(Anzahl_Deadline_nicht_ueberschritten AS DOUBLE) * 100 /
           CAST(Anzahl_Executions AS DOUBLE)), 2) AS "Deadline eingehalten (%)"
FROM (
    SELECT
        p.class,
        (SELECT COUNT(winner_id) FROM games g
            JOIN players ssp ON ssp.id = g.winner_id
            WHERE ssp.class = p.class) AS Gewonnene_Spiele,
        (SELECT COUNT(class) FROM participants pp
            JOIN players ssp ON ssp.id = pp.player_id
            WHERE ssp.class = p.class) AS Gespielte_Spiele,
        (SELECT ROUND(MAX(execution), 4) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.class = p.class) AS Max_Ausfuerungszeit_in_Sek,
        (SELECT ROUND(AVG(execution), 4) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.class = p.class) as Avg_Ausfuerungszeit_in_Sek,
        (SELECT ROUND(AVG(execution), 4) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.class = p.class AND execution < 60) AS
    Avg_Ausfuerungszeit_in_Sek_ohne_deadline,
        (SELECT COUNT(class) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.class = p.class) AS Anzahl_Executions,
        (SELECT COUNT(class) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.class = p.class
            AND execution < 60) AS Anzahl_Deadline_nicht_ueberschritten
    FROM players p
    GROUP BY p.class
) AS result
ORDER BY "Gewinnrate (%)" DESC
```

Listing A.5: Auswertung der besten KI-Klasse

```
SELECT class AS "Klasse", info AS "Info",
    Gewonnene_Spiele AS "Wins",
    Gespielte_Spiele AS "Plays",
    ROUND((CAST(Gewonnene_Spiele AS DOUBLE) * 100 /
           CAST(Gespielte_Spiele AS DOUBLE)), 2) AS "Gewinnrate (%)",
    Max_Ausfuerungszeit_in_Sek AS "Max Zeit (Sek)",
    Avg_Ausfuerungszeit_in_Sek AS "Avg Zeit (Sek)",
    (SELECT ROUND(AVG(execution), 4) FROM execution_times e
        JOIN players ssp ON ssp.id = e.player_id
        WHERE ssp.class = result.class
        AND execution < 60) AS "Avg Zeit oD (Sek)",
    ROUND((CAST(Anzahl_Deadline_nicht_ueberschritten AS DOUBLE) * 100 /
           CAST(Anzahl_Executions AS DOUBLE)), 2) AS "Deadline eingehalten (%)"
FROM (
    SELECT p.class, p.info,
        (SELECT COUNT(winner_id) FROM games g
            JOIN players ssp ON ssp.id = g.winner_id
            WHERE ssp.info = p.info
            AND ssp.class = p.class) as Gewonnene_Spiele,
        (SELECT COUNT(info) FROM participants pp
            JOIN players ssp ON ssp.id = pp.player_id
            WHERE ssp.info = p.info
            AND ssp.class = p.class) AS Gespielte_Spiele,
        (SELECT ROUND(MAX(execution), 4) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.info = p.info
            AND ssp.class = p.class) AS Max_Ausfuerungszeit_in_Sek,
        (SELECT ROUND(AVG(execution), 4) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.info = p.info
            AND ssp.class = p.class) AS Avg_Ausfuerungszeit_in_Sek,
        (SELECT COUNT(class) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.info = p.info
            AND ssp.class = p.class) AS Anzahl_Executions,
        (SELECT COUNT(class) FROM execution_times e
            JOIN players ssp ON ssp.id = e.player_id
            WHERE ssp.info = p.info
            AND ssp.class = p.class
            AND execution < 60) AS Anzahl_Deadline_nicht_ueberschritten
    FROM players p
    GROUP BY p.class, p.info
) AS result
ORDER BY "Gewinnrate (%)" DESC
```

Listing A.6: Auswertung der besten KI-Konfiguration

A.4 Weiterentwicklung

```yaml
name: Python Application

on:
  pull_request:
    branches: [ main ]

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.8
      uses: actions/setup-python@v2
      with:
        python-version: 3.8
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest poetry
        poetry export -f requirements.txt | pip install -r /dev/stdin
    - name: Lint with flake8
      run: |
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        flake8 . --count --exit-zero --max-complexity=10 --statistics
    - name: Test with pytest
      run: |
        pytest --junit-xml pytest.xml
    - name: Publish Unit Test Results
      if: always()
      uses: EnricoMi/publish-unit-test-result-action@v1.3
      with:
        check_name: Unit Test Results
        github_token: ${{ secrets.GITHUB_TOKEN }}
        files: pytest.xml
    - name: Upload Unit Test Results
      if: always()
      uses: actions/upload-artifact@v2
      with:
        name: Unit Test Results (Python 3.8)
        path: pytest.xml
```

Listing A.7: YAML-Konfiguration der Github Action

Literaturverzeichnis

[1] Gesellschaft für Informatik, "informatiCup 2021-Aufgabe." https://github.com/informatiCup/InformatiCup2021/blob/master/spe_ed.pdf, November 2020. [Online; Zugriff am 27. November 2020].

[2] TIOBE - the software quality company, "TIOBE Index for October 2020." https://www.tiobe.com/tiobe-index/, Oktober 2020. [Online; Zugriff am 01. November 2020].

[3] P. Carbonnelle, "PYPL PopularitY of Programming Language." https://pypl.github.io/PYPL.html, 2020. [Online; Zugriff am 01. November 2020].

[4] Springboard India, "Best language for Machine Learning: Which Programming Language to Learn." https://in.springboard.com/blog/best-language-for-machine-learning/, August 2020. [Online; Zugriff am 01. November 2020].

[5] Developer Economics, "What is the best programming language for Machine Learning?." https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7, Mai 2017. [Online; Zugriff am 01. November 2020].

[6] PyPA, "User Guide - pip 20.3 documentation." https://pip.pypa.io/en/stable/user_guide/#requirements-files, November 2020. [Online; Zugriff am 01. Dezember 2020].

[7] T. Birchard, "Package Python Projects the Proper Way with Poetry." https://hackersandslackers.com/python-poetry-package-manager/, Januar 2020. [Online; Zugriff am 01. November 2020].

[8] Docker Inc., "python - Docker Hub." https://hub.docker.com/_/python, November 2020. [Online; Zugriff am 27. November 2020].

[9] ubuntuusers, "pip > Wiki > ubuntuusers.de." https://wiki.ubuntuusers.de/pip/, Februar 2020. [Online; Zugriff am 27. November 2020].

[10] Docker Inc., "Best practices for writing Dockerfiles." https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. [Online; Zugriff am 27. November 2020].

[11] PyGame, "About - pygame wiki." https://www.pygame.org/wiki/about, Oktober 2020. [Online; Zugriff am 27. November 2020].

[12] Buzz Blog Box, "What is training data its types and why it is important?." `https://becominghuman.ai/what-is-training-data-its-types-and-why-it-is-important-f998424c3c9`, Februar 2020. [Online; Zugriff am 27. November 2020].

[13] E. Bohnsack and A. Lilja, "Mastering AchtungDieKurve with Deep Q-Learning using OpenAI Gym." `http://www.adamlilja.com/images/achtung.pdf`. [Online; Zugriff am 27. November 2020].

[14] A. Bresser, "python-pathfinding." `https://pypi.org/project/pathfinding/`. [Online; Zugriff am 27. November 2020].

[15] Wikipedia contributors, "Best-first search — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Best-first_search&oldid=989554272`, 2020. [Online; Zugriff am 01. Dezember 2020].

[16] A. Bresser, "python-pathfinding." `https://github.com/brean/python-pathfinding/blob/master/pathfinding/finder/best_first.py`. [Online; Zugriff am 27. November 2020].

[17] B. Stout, "Smart moves: Intelligent pathfinding," in *Game Developer*, vol. 3 no. 10, Game Developer, Oktober 1996.

[18] B. Luderer, *Wie misst man Entfernungen?*, pp. 26–28. Wiesbaden: Springer Fachmedien Wiesbaden, 2017.

[19] Xiao Cui and Hao Shi, "A*-based pathfinding in modern computer games." `https://www.researchgate.net/profile/Xiao_Cui7/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games/links/54fd73740cf270426d125adc.pdf`, 2011. [Online; Zugriff am 19. November 2020].

[20] TK, "Everything you need to know about tree data structures." `https://www.freecodecamp.org/news/all-you-need-to-know-about-tree-data-structures-bceacb85490c/`, November 2017. [Online; Zugriff am 27. November 2020].

[21] Python Software Foundation, "datetime — Basic date and time types." `https://docs.python.org/3/library/datetime.html`, November 2020. [Online; Zugriff am 01. Dezember 2020].

[22] S. Astanin, "python-tabulate." `https://pypi.org/project/tabulate/`. [Online; Zugriff am 27. November 2020].

[23] sqlite.org, "Features Of SQLite." `https://www.sqlite.org/features.html`, 2020. [Online; Zugriff am 28. November 2020].

[24] Python Software Foundation, "sqlite3 — DB-API 2.0 interface for SQLite databases." `https://docs.python.org/3/library/sqlite3.html`, November 2020. [Online; Zugriff am 01. Dezember 2020].

[25] Wikipedia contributors, "Database normalization — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Database_normalization&oldid=987846461`, 2020. [Online; Zugriff am 01. Dezember 2020].

[26] Wikipedia contributors, "Model–view–controller — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=990104986`, 2020. [Online; Zugriff am 01. Dezember 2020].

[27] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[28] D. Astels, *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

[29] Microsoft, "Unit test basics." `https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019#write-your-tests`, August 2019. [Online; Zugriff am 27. November 2020].

[30] T. Okubo and H. Tanaka, "Secure software development through coding conventions and frameworks," in *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pp. 1042–1051, 2007.

[31] Google, "Google Python Style Guide." `https://google.github.io/styleguide/pyguide.html`, November 2020. [Online; Zugriff am 26. November 2020].

[32] Python Code Quality Authority, "Pylint." `https://pypi.org/project/pylint/`. [Online; Zugriff am 27. November 2020].

[33] JetBrains, "PyCharm." `https://www.jetbrains.com/pycharm/`. [Online; Zugriff am 27. November 2020].

[34] T. Ziade, "flake8." `https://pypi.org/project/flake8/`. [Online; Zugriff am 27. November 2020].

[35] Python Software Foundation, "abc — Abstract Base Classes." `https://docs.python.org/3/library/abc.html`, November 2020. [Online; Zugriff am 01. Dezember 2020].

[36] Wikipedia, "Dependency injection — wikipedia, die freie enzyklopädie." `https://de.wikipedia.org/w/index.php?title=Dependency_Injection&oldid=204514178`, 2020. [Online; Zugriff am 01. Dezember 2020].

[37] Git, "Git." `https://git-scm.com/`. [Online; Zugriff am 27. November 2020].

[38] GitHub, Inc., "Enabling branch restrictions." `https://docs.github.com/en/free-pro-team@latest/github/administering-a-repository/enabling-branch-restrictions`, 2020. [Online; Zugriff am 27. November 2020].

[39] J. Vallandingham, "Feature Branches and Pull Requests : Walkthrough." `https://gist.github.com/vlandham/3b2b79c40bc7353ae95a`, 2019. [Online; Zugriff am 27. November 2020].

[40] R. Jeffries and K. Beck, "Extreme Programming Code Reviews." `https://wiki.c2.com/?ExtremeProgrammingCodeReviews`, Juni 2004. [Online; Zugriff am 27. November 2020].

[41] GitHub, Inc., "Quickstart for GitHub Actions." `https://docs.github.com/en/free-pro-team@latest/actions/quickstart`, 2020. [Online; Zugriff am 08. November 2020].

[42] GitHub, Inc., "Workflow syntax for GitHub Actions." `https://docs.github.com/en/free-pro-team@latest/actions/reference/workflow-syntax-for-github-actions`, 2020. [Online; Zugriff am 08. November 2020].

[43] pdoc, "pdoc – Auto-generate API documentation for Python projects." `https://pdoc3.github.io/pdoc`, November 2020. [Online; Zugriff am 27. November 2020].

[44] GitHub, Inc., "GitHub Pages." `https://pages.github.com`, November 2020. [Online; Zugriff am 27. November 2020].