InformatiCup 2021: Dokumentation

Team Chillow
(Florian Trei, Jonas Hellmann)

Wintersemester 2020/21

Spiel-Implementierung:
spe_ed

Stand:
25. November 2020

Inhaltsverzeichnis

Abbildungsverzeichnis **3**

Tabellenverzeichnis **4**

Listing-Verzeichnis **5**

Abkürzungsverzeichnis **6**

1 Einleitung **7**

2 Einstieg in das Projekt **8**
2.1 Auswahl der Programmiersprache . . . 8
2.2 Erstellung eines lauffähigen Projekts . . . 8
2.2.1 Einsatz von Poetry als Build-Tool . . . 8
2.2.2 Entwicklung eines Dockerfile . . . 8
2.3 Nachstellung des Spiels . . . 9

3 Lösungsansatz **11**
3.1 Selbstlernende KI mithilfe von Trainingsdaten . . . 11
3.2 Lösungsansätze ohne selbstlernende KIs . . . 11
3.2.1 RandomAI . . . 12
3.2.2 NotKillingItselfAI . . . 12
3.2.3 PathfindingAI . . . 13
3.2.4 SearchTreeAI . . . 15
3.2.5 SearchTreePathfindingAI . . . 16
3.2.6 PathfindingSearchTreeAI . . . 16
3.2.7 Weitere nicht umgesetzte Lösungsansätze . . . 17
3.3 Vorgehen zur Auswahl der besten Strategie . . . 18

4 Implementierung **19**
4.1 Modellierung des Spiels . . . 19
4.2 Implementierung des Online-Spiels . . . 20
4.2.1 Einlesen des Spiel-Zustands . . . 20
4.2.2 Ermittlung der besten Aktion . . . 21
4.2.3 Übergabe der Aktion an die Web-API . . . 21
4.3 Implementierung des Offline-Spiels . . . 22
4.3.1 Implementierung des GameService . . . 22
4.4 Bereitstellung einer Oberfläche . . . 23
4.4.1 View-Interface . . . 23
4.4.2 Darstellung des Spiels in der Konsole . . . 24
4.4.3 Nutzung von PyGame als grafische Oberfläche . . . 24

5 Software-Qualität **25**
5.1 Architektur der Software 25
5.1.1 Package-Struktur 26
5.2 Automatisierte Tests 26
5.3 Coding Conventions 27

6 Weiterentwicklung **28**
6.1 Erweiterbarkeit des Codes 28
6.2 Einsatz von PRs im Git-Workflow 28
6.3 Nutzung von Github Actions 29

7 Fazit **30**
7.1 Einschätzung unserer Lösung 30
7.2 Reflexion des Wettbewerbs 30

8 Benutzerhandbuch **31**
8.1 Installation 31
8.1.1 Docker 31
8.1.2 Manuelle Installation 31
8.2 Benutzung 32

9 Anhang **33**
9.1 Lösungsansatz 33
9.2 Implementierung 34
9.3 Weiterentwicklung 39

Literaturverzeichnis **41**

Abbildungsverzeichnis

2.1 Darstellung der Oberfläche in der Konsole 10
2.2 Darstellung der grafischen Oberfläche mit PyGame 10

3.1 `NotKillingItselfAI` Fehlentscheidung bei Geschwindigkeit größer eins 12
3.2 `NotKillingItselfAI` unten rechts in einer Sackgasse 13
3.3 Auswirkung unterschiedlicher Aktionen auf die erreichbaren Pfade 14
3.4 Visualisierung der Idee der `SearchTreeAI` 16

4.1 UML-Klassendiagramm des Modells 19

5.1 Schichtenarchitektur der Software 25
5.2 Package-Struktur des Projekts 26

9.1 UML-Klassendiagramm der Künstliche Intelligenz (KI)s 33
9.2 Sequenzdiagramm zur Implementierung eines Spielzug 34
9.3 Aktivitätsdiagramm zur Durchführung einer Spieler-Aktion durch den `GameService` 40

Tabellenverzeichnis

3.1 Ausführungsdauer unterschiedlicher Algorithmen zum Finden von Pfaden 14

8.1 Steuerung der Oberflächen 32

Listings

2.1 Dockerfile zum Erstellen eines lauffähigen Containers 9
4.1 `play()`-Methode des `OnlineControllers` 20
5.1 Beispiel für einen Unit-Test 27
9.1 `__play()`-Methode des `OnlineControllers` 35
9.2 JSON-Repräsentation eines Spiel-Zustands 36
9.3 `get_and_visit_cells(player, action)`-Methode des `GameService` 37
9.4 `update(game: Game)`-Methode der `GraphicalView` 38
9.5 YAML-Konfiguration der Github Action 39

Abkürzungsverzeichnis

KI	Künstliche Intelligenz
PR	Pull Request
MVC	Model-View-Controller

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[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.

[1] Eine genauere Beschreibung lässt sich in folgendem PDF-Dokument finden: `https://github.com/informatiCup/InformatiCup2021/blob/master/spe_ed.pdf`

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 konnte. Der Grund für diese Entscheidung liegt neben dem starken Interessen 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. [1] [2] 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. [3] [4]

2.2 Erstellung eines lauffähigen Projekts

2.2.1 Einsatz von Poetry als Build-Tool

Der einfachste Weg, um Abhängigkeiten in Python zu verwalten, ist, eine Datei mit dem Namen `requirements.txt` zu verwenden und in dieser die eingesetzten Bibliotheken aufzulisten. 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, aber auch weitere Funktionen wie das Ausführen von Tests und Erstellen eines fertigen Pakets anbietet. Zudem funktioniert die Auflösung komplexerer Abhängigkeiten von Paketen durch dieses Tool sehr gut. [5]

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 wird bereits ein vorgefertigter Container bereitgestellt, in dem Python und Pip installiert ist. Anschließend wird mittels Pip die Installation von Poetry durchgeführt. Dieses Tool wiederum bietet 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. Später sind noch Umgebungsvariablen zur Steuerung des Programm-Ablaufs hinzugekommen, sodass das Dockerfile letztendlich wie in *Listing 2.1* dargestellt aussieht.

```
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  RUN pip install poetry
10 RUN poetry export -f requirements.txt | pip install -r /dev/stdin
11
12 CMD [ "python", "./main.py", "--deactivate-pygame=TRUE", "--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.

Diese Logik ist relativ einfach nachzuvollziehen. 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 und bewegt sich nicht weiter vorwärts, auch wenn seine Aktion dies normalerweise noch bewirkt hätte. Verhindert werden können solche Kollisionen mithilfe eines Sprungs, der bei entsprechendem Tempo automatisch in jeder sechsten Runde des Spiels ausgeführt wird.

Wir haben diese Logik, auf die wir später noch genauer eingehen werden und die bei der Implementierung der KIs wiederverwendet werden konnte, in einer Klasse `GameService` implementiert. 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* verdeutlicht.

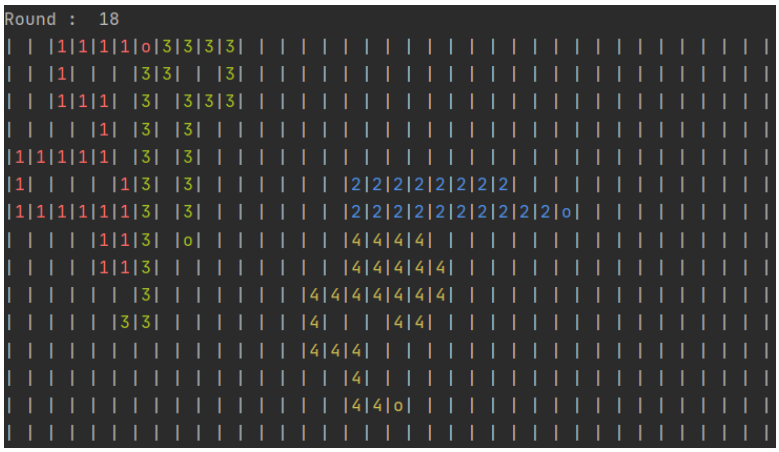

Abbildung 2.1: Darstellung der Oberfläche in der Konsole

Zum anderen gibt es auch die Möglichkeit, das Spiel auf einer grafischen Oberfläche zu spielen, die mittels PyGame 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* 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, gute Trainingsdatensätze besitzen zu müssen. 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. 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. 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 in Code zu implementieren, hatten wir mehrere unterschiedliche Ideen, die nachfolgend 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 9.1: (UML-Klassendiagramm der KIs)* nachvollzogen werden. Die Mehrfachvererbungen, wie sie im UML-Klassendiagramm dargestellt werden, wurden in der Implementierung durch die Verwendung von Python als Programmiersprache genau so umgesetzt.

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 Problemes der `RandomAI` haben wir uns dafür entschieden, die `NotKillingItselfAI` zu implementieren. Die KI soll aus allen Aktionen eine zufällige Aktion auswählt, 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* 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.

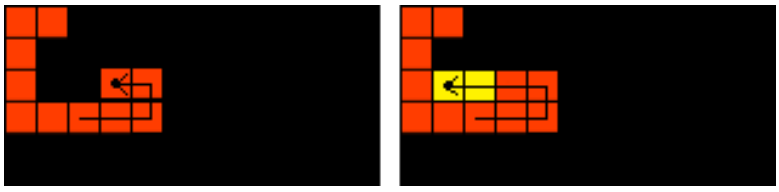

Abbildung 3.1: `NotKillingItselfAI` Fehlentscheidung bei Geschwindigkeit größer eins

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 führt. Das sorgt dafür, dass die KI entsprechend der Tiefe in die Zukunft vorausschaut und nicht verlieren wird. Dabei bleiben die gegnerischen Aktionsmöglichkeiten jedoch unberücksichtigt.

Außerdem hat die `NotKillingItselfAI` das Problem, dass sie sich zwar nicht mehr im unmittelbar folgenden Zug tötete, jedoch häufig in Sackgassen läuft. Dieses Problem wird in der *Abbildung 3.2* verdeutlicht.

Abbildung 3.2: `NotKillingItselfAI` unten rechts in einer Sackgasse

3.2.3 PathfindingAI

Folglich der `NotKillingItselfAI` haben wir einen Lösungsansatz gesucht, welcher das Betreten von Sackgassen verhindert. Genutzt wurde die Basis der `NotKillingItselfAI`, sodass nur zwischen überlebenden Aktionen gewählt wird. Um die Aktionen zu finden, die nicht in eine Sackgasse läuft, haben wir uns für einen Lösungsansatz zum Finden von Pfaden entschieden. Dazu wird eine konfigurierbare Anzahl zufälliger Koordinaten generiert, auf denen sich bisher kein Spieler befindet. Anschließend wird für jede Aktion geprüft, zu wie vielen der 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* 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.
Bei diesem Lösungsansatz arbeiten wir lediglich mit der Wahrscheinlichkeit, dass wir nicht in eine Sackgasse laufen. Dies liegt dem Problem zugrunde, dass wir die Implementierung eines Algorithmus zum tatsächlichen Erkennen von Sackgassen oder abgesperrten Gebieten als

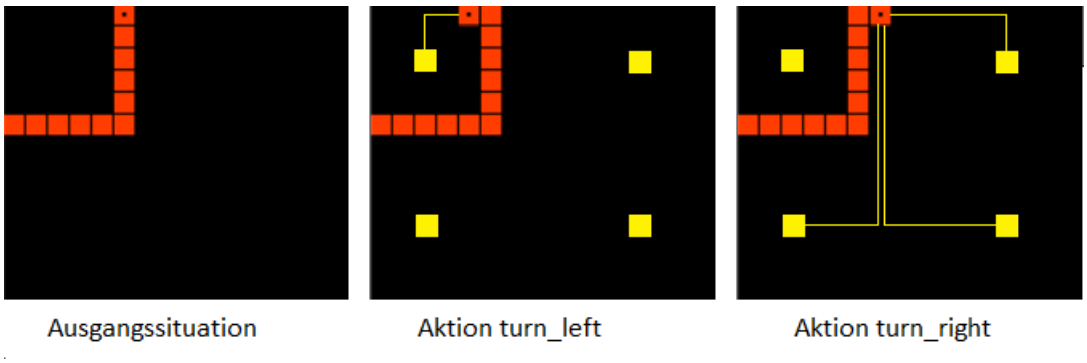

Abbildung 3.3: Auswirkung unterschiedlicher Aktionen auf die erreichbaren Pfade

schwieriger erachteten.

Wir haben die Python-Bibliothek `pathfinding` 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. 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

Bei dem Best-First-Algorithmus handelt es sich um eine Bestensuche, bei der ein gegebener Graph mit einem Start- und einem Endpunkt nach einem Pfad durchsucht wird. Die Strategie der Bestensuche ist es, den vielversprechendsten Knoten als Nächstes zu wählen und von dort aus dann wiederum die Suche fortzusetzen. Der Dokumentation zur von uns genutzten Klasse `BestFirst` der Bibliothek `pathfinding` kann entnommen werden, dass für den Best-First-Algorithmus der A*-Algorithmus genutzt wird und zusätzlich die Heuristik für die Bewertung der Nachbarknoten geändert wurde. 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. [6]
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. [7]

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 eines Suchbaums alle möglichen gegnerischen und eigenen Aktions-Kombinationen eine bestimmte Anzahl Spielzüge in die Zukunft zu simulieren. Dabei wird dann nach einem Teilbaum gesucht, bei dem vorhergesagt werden kann, dass der eigene Spieler unabhängig der von allen anderen Spielern ausgewählten Aktionen nicht im nächsten Zug 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* visualisiert, wobei es in diesem Beispiel neben dem eigenen Spieler noch die beiden weiteren Spieler mit den IDs 1 und 2 gibt. 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.

Gibt es einen Pfad bis zur vorletzten Ebene, nach dem bei der Kombination aller gegnerischen Aktionen immer ein Überleben auf der untersten Ebene folgt, ist eine Ausgangsaktion gefunden, die in dem aktuellen Zug ausgeführt werden kann, um in jedem Fall die Anzahl der nächsten berechneten Züge überleben zu können.

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 $(6^5)^3 = 470.184.984.576$

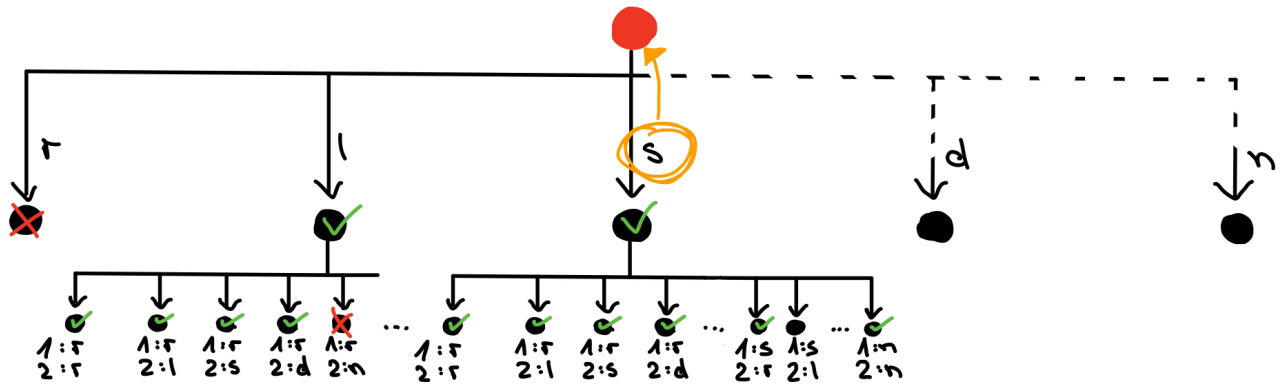

Abbildung 3.4: Visualisierung der Idee der `SearchTreeAI`

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.
Problematisch bei einer zu kleinen Distanz zu den gegnerischen Spielern 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. 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 nutzt oder einen sechsten Zug mit einer Geschwindigkeit größer zwei nutzen muss, um aus dem begrenzten Feld wieder heraus zu kommen. Die Berechnung, mit einem sechsten Zug über die umkreisende Spur zu kommen, ist 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. 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.
Daher steigt die Wahrscheinlichkeit, dass die eigene KI durch ihr agressives 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.

3.3 Vorgehen zur Auswahl der besten Strategie

Kapitel ausformulieren

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* zu sehen. Das `Game` hat Zugriff auf die eigenen Eigenschaften, kennt aber auch alle `Player`, die an diesem Spiel teilnehmen. Zudem besteht ein `Game` aus einem zweidimensionalen Array aus `Cells`, die das Spielfeld repräsentieren. In einer `Cell` befinden sich dann alle Spieler, die dieses Spielfeld bereits besucht haben.

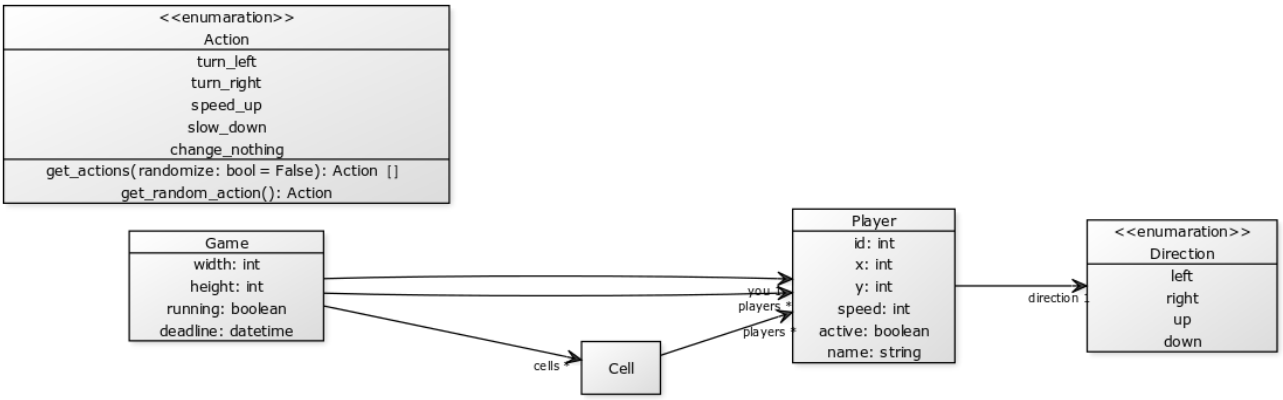

Abbildung 4.1: UML-Klassendiagramm des Modells

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-Zeit verwendet, auf deren Nutzung nachfolgend noch eingegangen wird.

Die zum Start des Spiels öffentlich bereitgestellte Methode `play()` ist in *Listing 4.1* dargestellt. Diese ist sehr simpel und ruft lediglich die private Methode `__play()` auf. Hierbei handelt es sich um eine asynchrone Methode, wie schon in der Methoden-Signatur deutlich wird. 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 9.2: (Sequenzdiagramm zur Implementierung eines Spielzug)* in Form eines Sequenzdiagrammes nachvollziehbar und die Umsetzung im Python-Code kann zusammen mit dem Verbindungsaufbau dem *Anhang 9.1: (`__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 9.2: (JSON-Repräsentation eines Spiel-Zustands)* abgebildet. Anzumerken ist, dass die exakte Formatierung des Strings abweichen kann.

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 geparst wird. Diese Server-Zeit wird dann verwendet, um durch einen Abgleich mit der Zeit des eigenen Systems gegebenenfalls die Deadline für die Festlegung auf die nächste Aktion zu verschieben, da diese sich immer nach der Server-Zeit richtet. Es wird hierbei eine Toleranz von drei Sekunden eingebaut, da je nach Internet-Verbindung die Anfrage an den Server etwas dauern kann und so die Aktion besser zu früh als zu spät übermittelt werden sollte.

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 noch nicht beendet ist. Bei einem Abbruch wird dann die Standard-Aktion an den Server geschickt, andernfalls wird die Berechnung der stärkeren KI verwendet. 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 noch mitgeteilt werden. In der Dokumentation für die diesjährige Aufgabenstellung 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"`}, welcher dann über die Websocket-Verbindung an den Server gesendet wird.

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 KI 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.

Da wir 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.

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 9.3: (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 9.3: (`get_and_visit_cells(player, action)`-Methode des `GameService`)*. Dabei wird der Spieler aktualisiert (x- und y-Koordinate, speed), der die Ausführung der Aktion eingeleitet hat und er wird 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 autark 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. Beide 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()` soll die Funktionalität bereitgestellt werden, als menschlicher Spieler eine Aktion durchführen zu können. Die dritte Methode `end()` ist dafür gedacht, möglicherweise 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` 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 eines zweidimensionalen Spiels, wie es das Spiel spe_ed ist. Des Weiteren benötigt PyGame einen geringen Aufwand bei der Initialisierung, um ein Spiel rendern zu können.

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 9.4: (`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

Software-Qualität

Um sicherzustellen, dass Software erwartungsgemäß funktioniert und eine Weiterentwicklung vereinfacht wird, ergibt es Sinn, verschiedene Aspekte zu berücksichtigen, die für eine verbesserte Qualität der Software sorgen. Die Wartbarkeit der Anwendung wird in *Kapitel 6.1 (Erweiterbarkeit des Codes)* noch genauer betrachtet.

5.1 Architektur der Software

Beim Aufbau der Software haben wir uns für eine dreischichtige Architektur und den Einsatz des Model-View-Controller (MVC)-Patterns entschieden. Die Architektur ist in *Abbildung 5.1* 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.

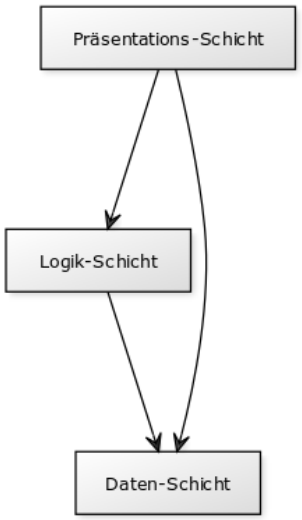

Abbildung 5.1: Schichtenarchitektur der Software

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.

5.1.1 Package-Struktur

Aufbauend darauf haben wir das MVC-Pattern umgesetzt und dies in dem Aufbau der Package-Struktur verdeutlicht, welche auch in *Abbildung 5.2* 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 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 aus Service nach Controller unterbunden werden. Es handelt sich um eine Erweiterung des klassischen MVC-Patterns. Zuletzt bildet das Modell-Package die oben beschriebene Modell-Schicht ab.

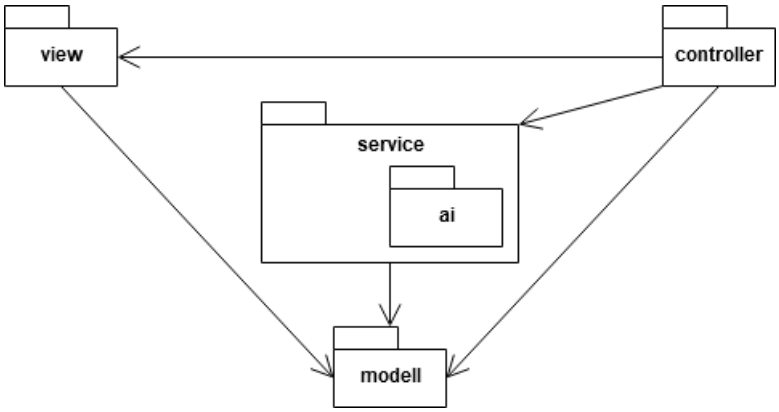

Abbildung 5.2: Package-Struktur des Projekts

5.2 Automatisierte Tests

Zur Sicherstellung der Korrektheit der Software wurden automatisierte Tests 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 haben wir zwar nicht für alle Komponenten verwendet, an einigen Stellen war dies aber durchaus 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 funktioniert.

Ein Beispiel für einen Test ist in *Listing 5.1* 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.

```
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 5.1: Beispiel für einen Unit-Test

5.3 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. Viele solcher Vorgabe werden bereits durch die Programmiersprache geliefert und können daher automatisch überprüft werden. Dabei war Pylint als Tool hilfreich, das sich als Plugin direkt in PyCharm-IDE von JetBrains integrieren ließ und somit bereits bei der Arbeit am Code unmittelbar Verbesserungsvorschläge macht. Außerdem wurde in einem automatischen Prozess, der nachfolgend noch in *Kapitel 6.3 (Nutzung von Github Actions)* beschrieben wird, das Tool Flake8 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, der sich daraus ergeben hat, war eine bessere Unterstützung bei der Autovervollständigung und Anzeige von möglichen Fehlern durch die IDE.

Kapitel 6

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.

6.1 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 war notwendig, da Python als Programmiersprache keine Interfaces 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. Angewendet wurde dieses Vorgehen an oberster Stelle bei den `Controller`n 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.

6.2 Einsatz von PRs im Git-Workflow

Wir haben uns dazu entschieden, als Versionsverwaltungs-Tool Git 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. 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. 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 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.

6.3 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. [8] [9].

Die für unser Projekt verwendete Konfiguration wird in *Anhang 9.5: (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 diese Aktion fehl und verhindert einen Merge.

So wird durch Continious 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.

Kapitel 7

Fazit

7.1 Einschätzung unserer Lösung

Kapitel ausformulieren

7.2 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, prädiktive 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 war sehr interessant, sodass man immer auch im Hinterkopf hatte, einen Lösungsvorschlag zu entwickeln, der gegen die anderen eingereichten Projekte mithalten bzw. diese schlagen kann.

Kapitel 8

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.

8.1 Installation

Zur Verwendung dieses Projektes muss es 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`

8.1.1 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, wobei die URL zum spe_ed-Server und der API-Key entsprechend angepasst werden müssen:

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

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

8.1.2 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 eine andere

KI gespielt werden soll als die, für die wir uns am Ende entschieden haben, 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]`

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.

8.2 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 `OfflineController` - auf eine Eingabe von einem oder mehreren Spielern gewartet, bis die nächste Runde des Spiels gestartet wird. Der *Tabelle 8.1 (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 8.1: 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.

Kapitel 9

Anhang

9.1 Lösungsansatz

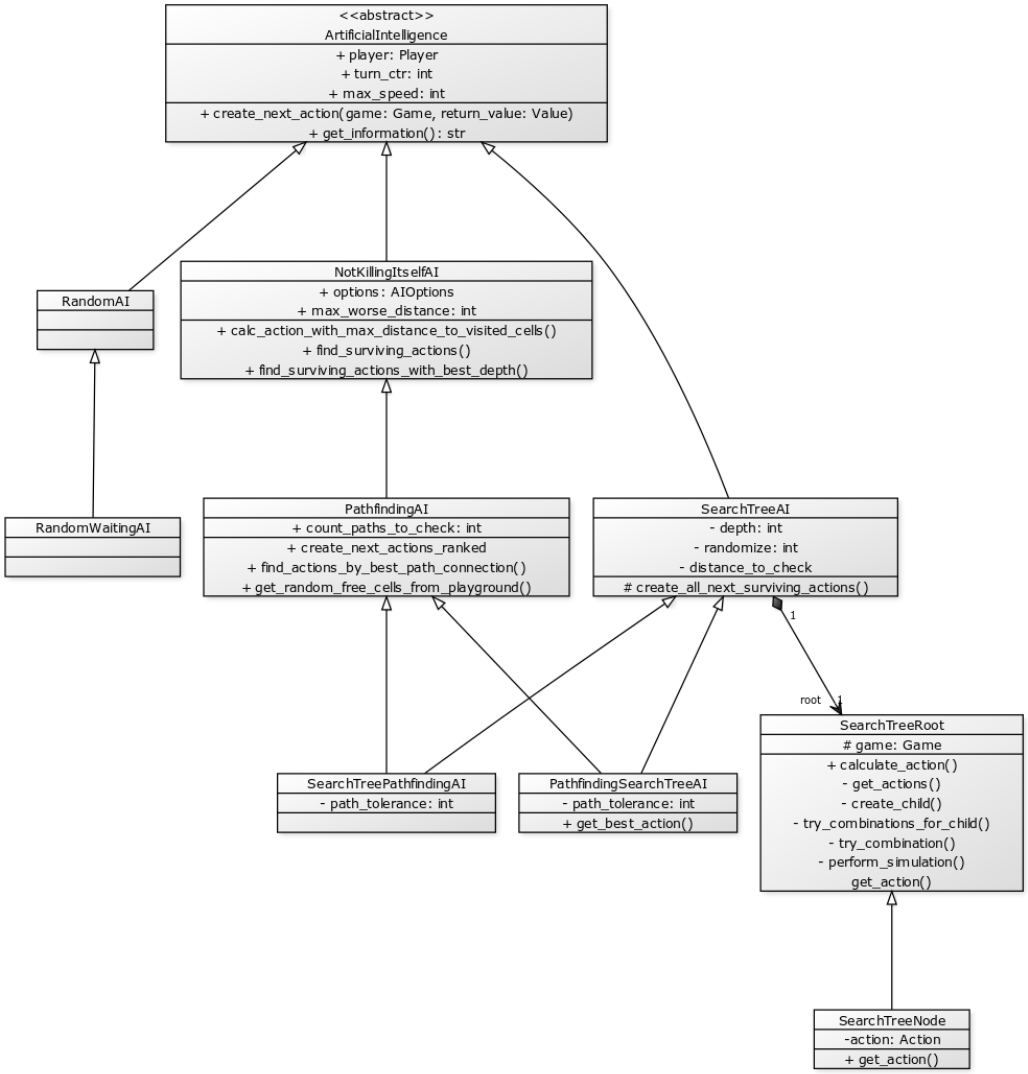

Abbildung 9.1: UML-Klassendiagramm der KIs

9.2 Implementierung

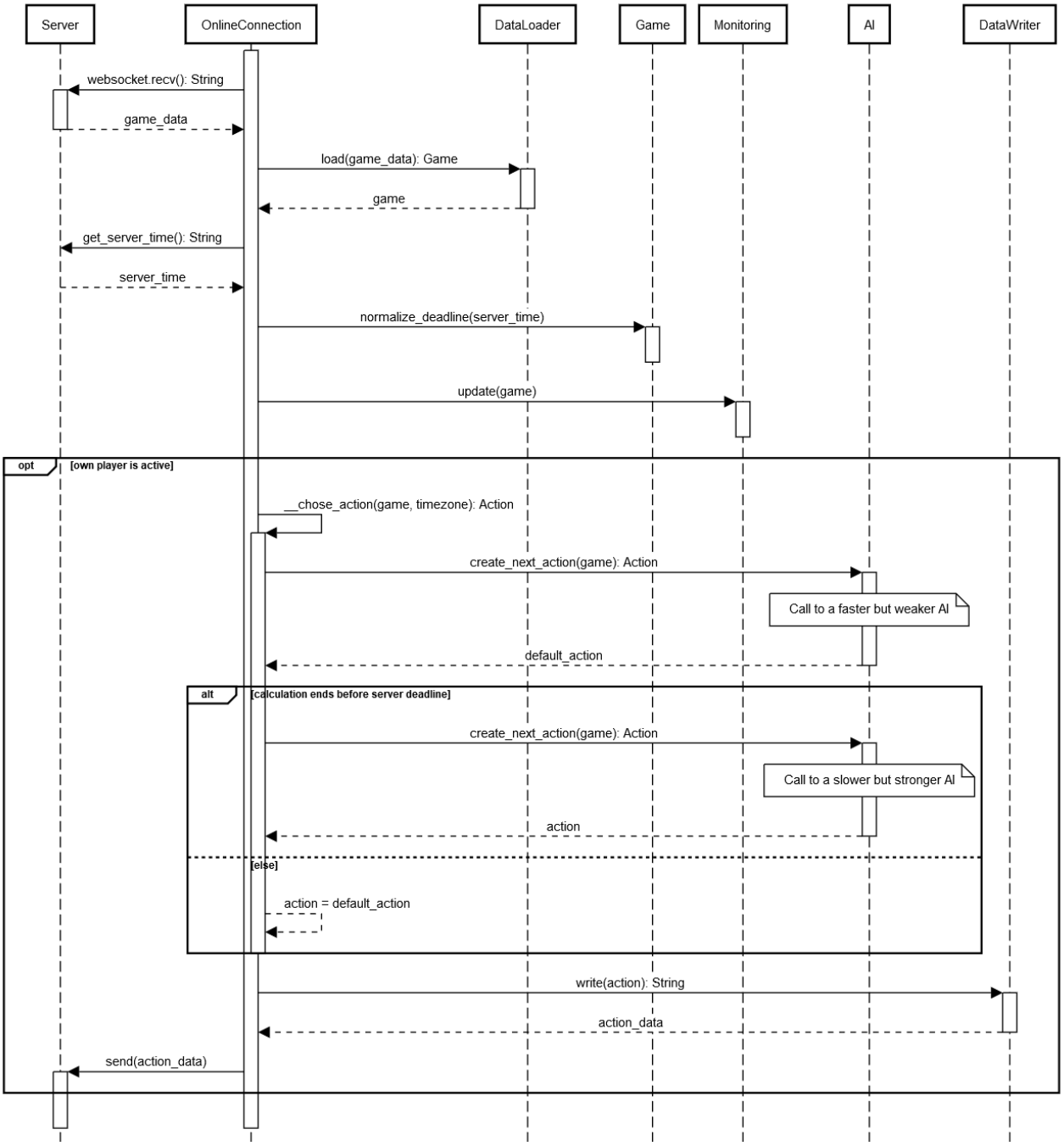

Abbildung 9.2: Sequenzdiagramm zur Implementierung eines Spielzug

```
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 9.1: `__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 9.2: JSON-Repräsentation eines Spiel-Zustands

```
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))

    if self.turn.turn_ctr % 6 == 0 and len(visited_cells) > 1:  # L cke, also nur ersten und letzten Punkt nehmen
        visited_cells = [visited_cells[0], visited_cells[-1]]

    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)
            return visited_cells
        player.x = x
        player.y = 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
```

Listing 9.3: `get_and_visit_cells(player, action)`-Methode des `GameService`

```
def update(self, game: Game):
    if not self._interface_initialized:
        self._initialize_interface(game)

    if not game.running:
        player = game.get_winner()
        print("Winner: Player " + str(player.id) + " ("
              + player.name + "). Your player ID was " + str(game.you.id))

    self.__screen.fill((0, 0, 0))
    for row in range(game.height):
        for col in range(game.width):
            pygame.draw.rect(self.__screen,
                             self._player_colors[game.cells[row][col].
    get_player_id()],
                             (col * self.RECTANGLE_SIZE + col,
                              row * self.RECTANGLE_SIZE + row,
                              self.RECTANGLE_SIZE,
                              self.RECTANGLE_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
                    pygame.draw.rect(self.__screen,
                                     self._player_colors[0],
                                     (col * self.RECTANGLE_SIZE + col +
    border_width,
                                      row * self.RECTANGLE_SIZE + row +
    border_width,
                                      self.RECTANGLE_SIZE - (2 * border_width),
                                      self.RECTANGLE_SIZE - (2 * border_width)))
    pygame.display.update()
    self.__clock.tick(60)
```

Listing 9.4: `update(game: Game)`-Methode der `GraphicalView`

9.3 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: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --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 9.5: YAML-Konfiguration der Github Action

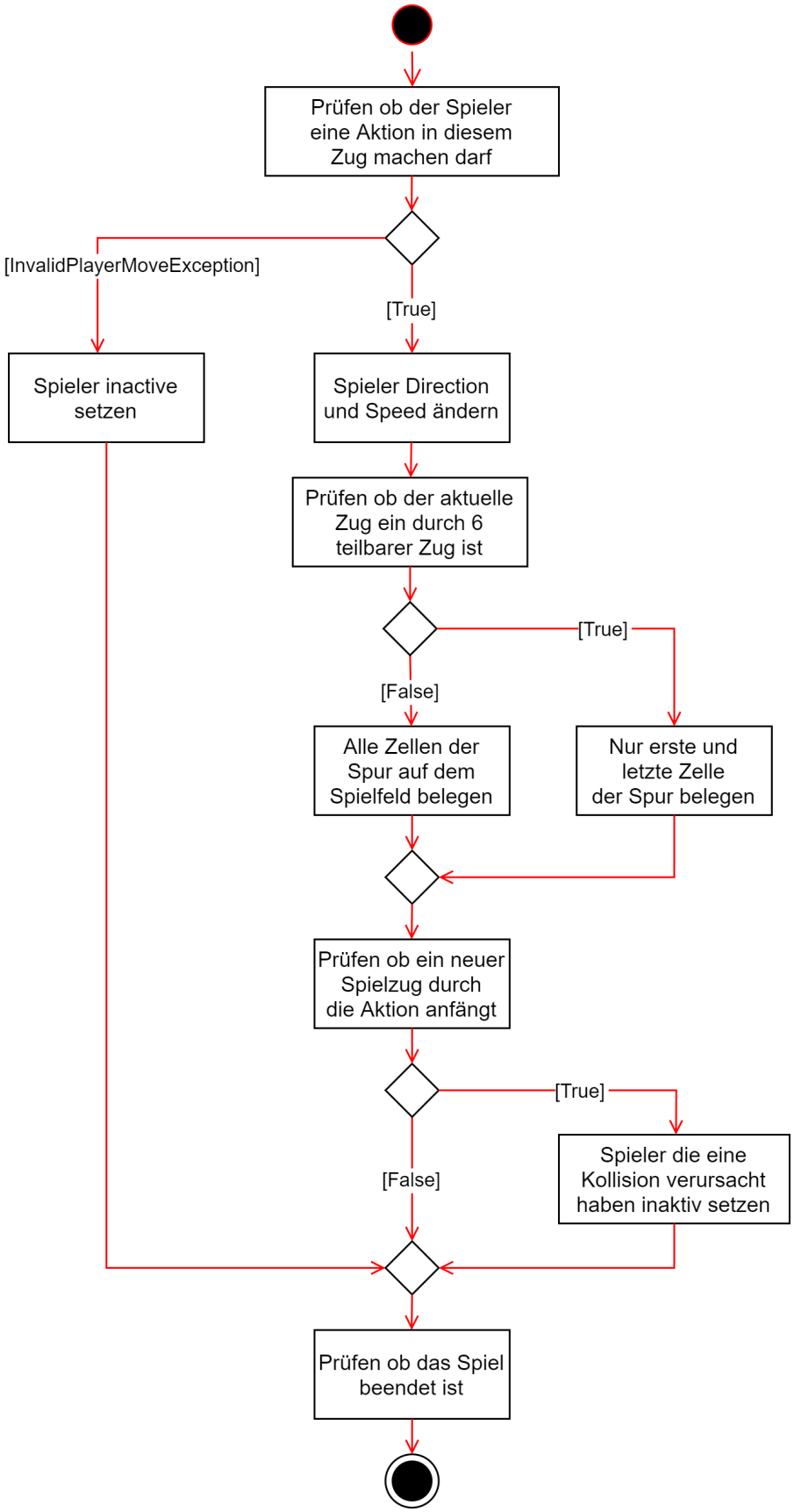

Abbildung 9.3: Aktivitätsdiagramm zur Durchführung einer Spieler-Aktion durch den `GameService`

Literaturverzeichnis

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

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

[3] 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].

[4] 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].

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

[6] Luderer, Bernd, "Wie misst man Entfernungen?." https://link.springer.com/chapter/10.1007%2F978-3-658-19188-7_10, 2017. [Online; Zugriff am 08. November 2020].

[7] Cui, Xiao and Shi, Hao, "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].

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

[9] 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].