Report Tic Tac Toe

1. Einführung

Aufgabe des Leistungsnachweises für das Modul Programmierung 2 war es, in Python als Konsolenanwendung das Spiel Tic Tac Toe zu implementieren und dazu einen Report anzufertigen. Nachfolgend wird die Funktionsweise des Programms anhand verschiedenster Überpunkte näher beschrieben.

2. Architektur

Als Architektur wurde die Model View Controller Architektur verwendet, wie in den Vorgaben beschrieben. Allgemein übernimmt die View die Interaktionen mit dem Benutzer. Abgesehen von der Eingabe und der Ausgabe über die Konsole gibt es in diesem Projekt noch die Methode *clear*. Diese bereinigt die Konsole und somit wird die Übersicht der Ausgaben angenehmer gestaltet.

Die View wird vom Controller und vom Model manipuliert. Die Klasse Model übernimmt dabei die Logik des Programms und ist in der *main* Datei als Klasse *App* zu finden. Dabei ist die genannte Logik in verschiedenen Klassen wie *AI* und *Player* ausgelagert.

Der Controller prüft Benutzereingaben und manipuliert anhand der Daten die View oder schickt eine Nachricht an Model. Dabei benutzt der Controller einige Methoden, die die Benutzereingaben prüfen und auch ohne den Model einfache Aufgaben, wie das Anzeigen des Menüs, übernehmen.

3. Speicher- und Ladefunktion

Die Funktionen *save_score* und *load_score* sind für die Verwaltung der Speicherdaten verantwortlich. Allgemein wird der Fortschritt in die Datei *saves.dat* gespeichert.

- *save_score*

  Die Funktion *save_score* speichert alle Spielrelevanten Variablen, wie z. B. die IDs, Symbole und Namen der Spieler in die Datei *saves.dat*. Trennzeichen ist hierbei die *(*. Wenn schon eine Sicherungsdatei existiert, wird diese zuerst gelöscht.

- *load_score*

  Die Funktion *load_score* lädt alle notwendigen Variablen aus der Datei *saves.dat.* Außerdem erzeugt die Funktion noch alle nötigen Objekte, wie z. B. das Spielbrett oder ggf. die KI. Eine Herausforderung hierbei war das Spielbrett wieder in den gespeicherten Zustand zu setzten, da dieses aus einem Dictionary besteht und das Lesen einer Datei lediglich einen String zurückgibt.

4. Spiel-KI

Ein weiterer nennenswerter Aspekt des Projektes ist die KI, die beim Einspieler Modus die Rolle des Gegners übernimmt. Grundlegend ist die KI in drei Schwierigkeitsstufen aufgeteilt, die der Spieler im Menü auswählen kann.

- Schwierigkeitsstufe 1

  Die erste Schwierigkeitsstufe wird über die Funktion *move_*weak gesteuert und ist lediglich ein Zufallsgenerator der anschließend prüft, ob der generierte Zug gültig ist oder nicht.

- Schwierigkeitsstufe 2

  Die zweite Schwierigkeitsstufe wird über *move_middle* gesteuert und prüft zuerst, ob der Spieler oder die KI mit einem nächsten Zug gewinnen können. Wenn der Spieler gewinnen könnte, setzt die KI ihren Zug auf das Feld, sodass der Spieler nun nicht mehr gewinnt. Kann die KI mit einem nächsten Zug gewinnen, setzt sie diesen auch so. Wenn keine der beiden Partien mit dem nächsten Zug gewinnen kann, wird die Funktion *move_weak* aufgerufen.

- Schwierigkeitsstufe 3

  In der finalen Schwierigkeitsstufe wird zum Handeln die Funktion *move_hard* verwendet. Da Tic Tac Toe ein gelöstes Spiel ist, gibt es Strategien, mit denen man niemals verlieren kann. Diese Strategien in Form von Algorithmen verwendet die KI in der schwerstes Schwierigkeitsstufe. Dabei wird unterschieden, ob sie im *Attack-Mode* oder im *Defense-Mode* ist:

  - *Attack-Mode*

    Wenn die KI das Spiel eröffnet, ist diese im *Attack-Mode*. Hier wird beim ersten Zug des Spielers eine von drei Angriffsstrategien gewählt, je nachdem, welchen Zug der Spieler setzte. Die Chance, dass die KI gewinnt anstatt Unentschieden spielt, liegt bei 2:3.

  - *Defense-Mode*

    Wenn der Spieler das Game eröffnet, ist die KI im *Defense-Mode*. Beim ersten Zug der KI wird eine von drei Strategien gewählt, welche die größte Siegeschance bietet. Die Wahl hängt auch hier von den Zügen des Spielers ab und je nachdem, wie der Spieler im Laufe des Spieles zieht, kann die Strategie beibehalten oder angepasst werden. Es besteht die Möglichkeit, dass der Spieler Züge setzt, die nicht in den Strategien geplant wurden. In diesem Fall ist es jedoch nicht mehr möglich, dass die KI oder der Spieler gewinnen kann und so spielt die künstliche Intelligenz mit der Funktion *move_middle*.

5. Tests

Das Erstellen von Tests eines Programmes ist für die Entwicklung und Wartung eines jeden Projektes notwendig, da so Fehler frühzeitig gefunden und behoben werden können. So ist das Projekt Tic-Tac-Toe mit Unittests geprüft worden. Diese sind mit der klassischen Schule implementiert worden, jeder Unittest ist also von den anderen isoliert. Allgemein rufen die Unittests jede Methode des Projektes auf und testen deren Logik mit unterschiedlichsten Werten auf Korrektheit.

Eine weitere Anforderung an das Programm war eine Hundert Prozentige Line Coverage, welche jedoch nur mit dem Kommentar *# pragma: no cover* realisierbar war. Durch diesen Kommentar wurde das Line Coverage angewiesen, bestimmten Code zu ignorieren. Grund dafür sind in der *main.py* Datei die letzten beiden Zeilen Code. Diese starten normalerweise das gesamte Programm, da beim Aufruf der *main.py* Datei die Variable *__name__* auf *_main__* gesetzt ist. Wenn das Programm getestet wird, ist der Inhalt der Variable *main*. Die Erweiterung der *if-Bedingung* löst dieses Problem jedoch nicht, da beim Testen das Programm nicht gestartet werden soll. Die Line Coverage bleibt also nur auf hundert Prozent, wenn man diese Zeilen des Programms ignoriert.

6. Performance

Das Analysieren der Laufzeit ist ein weiterer Bestandteil, der in diesem Report hervorgehoben wird. Die verwendete IDE „PyCharm“ bietet ein sogenanntes Profiling, welches die Performance des Codes analysiert und anschließend eine Übersicht über langsame sowie schnelle Codeteile bietet. Nach einigen Durchführungen des Profilers wird deutlich, dass die Funktion *input* die meiste Zeit benötigt. Dies liegt jedoch daran, dass das Programm wartet, bis der Benutzer eine Eingabe durchführt. Ein weiterer Codeabschnitt, der überdurchschnittlich lange benötigt um ausgeführt zu werden, ist die Funktion *time.sleep*. Diese wird benötigt, um dem Benutzer beim Spielen mit der KI die Meldung *KI überlegt* zwei Sekunden lang anzuzeigen. Beide Abschnitte lassen sich folglich nicht optimieren, sind jedoch beim Analysieren der Performance des Codes zu vernachlässigen.

Wenn man beide Funktionen außenvor lässt, sticht noch die statische Funktion *clear* durch die lange Laufzeit hervor. Diese wird benötigt, um die Konsole auf Windows, Linux und MacOS zu leeren. Die Laufzeit dieser Funktion hängt vor allem an *os.system*, da die Ausführung deutlich länger als der Rest des Codes benötigt. *os.system* wird jedoch benötigt, um Konsolenbefehle wie *cls* und *clear* auszuführen.

7. Aussichten

Insgesamt kann das Projekt mit allen vorgeschriebenen Implementationen und ohne bekannte Bugs im vorgeschriebenen Zeitraum abgegeben und somit als erfolgreich bezeichnet werden.

Einige Features, die nicht in der Aufgabenstellung gefordert waren, sind bereits entwickelt und getestet worden: jeder Spieler kann einen individuellen Namen und ein Symbol auswählen, die KI besitzt verschiedene Schwierigkeitsgrade und bei Beendigung werden Teile des Spielfeldes farblich hervorgehoben, falls eine der beiden Parteien gewonnen hat.

Jedoch gibt es auch Klassen und Funktionen, die künftig noch überarbeitet werden können. Die KI hat beispielsweise noch Entwicklungsbedarf, da sie immer denselben Zug macht, wenn sie anfängt. Auch das Menü kann optisch besser gestaltet werden und hat so Wartungsbedarf.

In dieser Version des Spiels ist auch der Sicherheitsaspekt pflegebedürftig, da die Sicherungsdatei ungeschützt bearbeitet werden kann und so eine Manipulation des Spieles möglich ist. Auch die Manipulation des Arbeitsspeichers ist möglich und so kann ein laufendes Spiel beeinflusst werden.