

Nim-Spiel-Simulation

-

Dokumentation

von
Andreas Maertens
für die Abschlussprüfung Winter 2012

Prüflingsnummer: 183 05662
Prüfungsausschuss: 6511-1

Inhaltsverzeichnis

1 Einführung	3
1.1 Motivation	3
1.2 Das Nim-Spiel	3
2 Aufgabenanalyse	4
2.1 Problembeschreibung	4
2.2 Lösungsansatz	4
3 Entwicklerdokumentation	5
3.1 verbale Verfahrensbeschreibung	5
3.1.1 Datenstruktur	5
3.1.2 Lösungsalgorithmus	6
3.2 Konzeption UML-Klassendiagramm	7
3.3 Klassen	8
3.3.1 Controller	8
3.3.2 FeldBuilder	9
3.3.3 OutputBuilder	9
3.3.4 Spieler	9
3.3.4.1 Spieler1	9
3.3.4.2 Spieler2	9
3.3.5 Turnier	10
3.3.6 Spiel	10
3.3.7 Log	11
3.3.8 UsrError	11
3.3.9 UsrNeedHelp	11
3.4 Code Snippets	12
3.5 Ausführliche Dokumentation	19
4 Benutzerhandbuch	19
4.1 Systemvoraussetzung	19
4.2 Programmstart	19
4.3 Bedienung	19
5 Beispiele	20
6 Testprotokoll	31
7 Schlusswort	32
7.2 Ausblick	32
8 Anhang	32
8.1 Quellcode	32
8.2 Juristische Erklärung	33

1 Einführung

1.1 Motivation

Im Rahmen der praktischen Abschlussprüfung Winter 2012/13 soll ein Softwaresystem entwickelt werden, welches das sogenannte Nim-Spiel simuliert. Es ist ein 2 Spieler Spiel und eine Spieler Intelligenz (künstliche Intelligenz (KI)) wurde durch die Aufgabenstellung definiert. Aufgabe des Prüflings ist es, eine KI für den zweiten Spieler so zu entwickeln, dass der zweite Spieler in den meisten Fällen das Spiel gewinnt. Der zweite Spieler hat dabei den Vorteil, immer den ersten Zug machen zu dürfen.

1.2 Das Nim-Spiel

Das Nim-Spiel ist ein Zwei-Personen-Spiel, bei dem die Spieler abwechselnd Hölzer ziehen müssen. Diese Hölzer sind in mehreren Reihen und dürfen immer nur aus einer Reihe genommen werden. Es muss immer mindestens ein Holz gezogen werden und es dürfen maximal alle Hölzer aus einer Reihe aufgenommen werden. Gewonnen hat der Spieler, der das letzte Holz zieht.



Das Nim-Spiel gehört zu der Klasse von Zwei-Personen-Spielen, mit der sich die "Kombinatorische Spieltheorie", ein Zweig der Mathematik, befasst. Spiele dieser Klasse unterliegen nicht dem Zufallseinfluss, haben keine für den Spieler verborgenen Informationen, wie z.B. Kartenspiele und enden nach einer endlichen Zahl von Spielzügen. Zu dieser Klasse gehören mit gewissen Einschränkungen auch Go oder Schach.

2 Aufgabenanalyse

2.1 Problembeschreibung

Bei einer gegebenen Startverteilung von Hölzern, die den nachfolgend genannten Regeln genügt, soll Spieler1 mit seiner vom Prüfling entwickelten Strategie häufiger gewinnen, als Spieler2, dessen Strategie dem Zufall unterliegt. Eine gültige Startaufstellung hat 1 bis 9 Reihen mit jeweils 1 bis 9 Hölzern. Gewonnen hat, wer das letzte Holz aufnimmt. Wenn nur noch eine Reihe Hölzer hat, soll Spieler2 alle verbleibenden Hölzer aufnehmen. Gibt es mehr als eine Reihe, so soll Spieler2 aus einer zufälligen Reihe eine beliebige Menge entnehmen. Die Startaufstellung wird dem Programm mit einer Datei übergeben und es sollen 10 Spiele zwischen Spieler1 und Spieler2 simuliert werden. Die gewählten Züge müssen protokolliert werden. Eine Auswahl der Spiele soll zusammen mit einer Auswertung in eine Ausgabedatei geschrieben werden.

2.2 Lösungsansatz

Die Spieler entscheiden den nächsten Zug allein nach den Bedingungen die sie auf dem Spielfeld vorfinden. Sie brauchen sich also nicht auf die Strategie des Gegenspielers einstellen. Somit braucht Spieler1 nur versuchen in jedem Zug, eine für den Nachfolger möglichst ungünstige Situation zu schaffen. Existieren nur noch 2 Reihen, ist es eine ungünstige Situation, wenn beide Reihen gleich viel Hölzer enthalten. Egal wie viele Hölzer man dann nimmt, der Gegenspieler kann diese Situation wieder herstellen, was am Ende zu der Situation führt, dass nur noch ein Holz pro Reihe existiert. Nimmt der Spieler dann eines weg, gewinnt der Gegenspieler. Ziel des Spieler1 ist es also diese ungünstige Situation mit seinem Zug herzustellen. Eliminiert er alle anderen Reihen und kann dann ausgleichen, hat er gewonnen. Die Chance, ein ausgeglichenes Spielfeld mit 2 Reihen zu erzeugen, erhöht sich, wenn er immer die Felder mit der geringsten Anzahl an Hölzern vollständig aufnimmt. Diese Strategie scheitert jedoch recht schnell, wenn der Spieler gegen einen Spieler spielt, dessen Taktik nicht auf Zufall beruht. Wenn man im Internet nach dem Nim-Spiel sucht, findet man eine Strategie, die wesentlich früher Gewinnsituationen erreichen, als die von Spieler1 verfolgte. Auf Wikipedia wird dabei die Strategie nach Bouton genannt. Bouton stellt alle Reihen als Dualzahlen dar und errechnet die Summen der einzelnen Stellen über alle Reihen. Sind diese Summen geradzahlig, ist dies eine Verluststellung. Bouton versucht am Ende eines Zuges möglichst eine solche Verluststellung zu erzeugen. Dieser Ansatz arbeitet schon wesentlich Früher auf eine Gewinnstrategie hin, als die Strategie von Spieler1, der mit seinem Handeln versucht, erst gegen Spielende eine Strategie umzusetzen.

3 Entwicklerdokumentation

In der Entwicklerdokumentation wird der allgemeine Aufbau der Softwaresimulation erörtert. Es werden die verwendeten Komponenten vorgestellt und das Verfahren zur Problemlösung erklärt.

3.1 verbale Verfahrensbeschreibung

3.1.1 Datenstruktur

In der Realität benötigen wir für die Beschreibung des Problems verschiedene Körper und Personen. Es sind genau zwei Spieler, die ein Spiel spielen. In der Simulation sollen 10 Spiele durchgeführt werden und danach eine Auswertung passieren. Diese Körper lassen sich hervorragend in eine Objektstruktur übersetzen, weshalb ein objektorientierter Ansatz gewählt wurde.

Die Programmstruktur ist nach dem Model-View-Controller Konzept entwickelt worden.

Zum *Modell* gehören die Programmteile, die sich um die Datenhaltung und die Programmlogik kümmern. Das sind in diesem Fall: Turnier, Spiel und Spieler.

Zum *Controller* gehören die Programmteile, die zwischen der View und dem Modell vermitteln. Dies ist in diesem Fall der Controller selbst und der FeldBuilder.

Zur *View* gehören die Programmteile, die die Eingabe entgegennehmen und sich um die Ausgabe kümmern. In diesem Fall ist das der OutputBuilder.

Die Eingabe wird bereits vor dem Programmstart getätigt und mithilfe der Programmparameter an den Controller gereicht.

Modellschicht

Turnier

Die Turnierklasse speichert jeweils 10 Spiele und stellt diesen eine Kopie des Spielfeldes und die beiden Spieler zur Verfügung. Nach dem Start der Spiele wartet Turnier auf deren Ende, bevor der Mainthread wieder in den Controller zurückkehrt.

Spiel

Die Spielklasse fragt in ihrer run()-Methode die beiden Spieler abwechselnd nach ihrem nächsten Zug und prüft dann, ob dieser gültig ist. Falls dies wider Erwarten nicht der Fall sein sollte, fragt Spiel den Spieler erneut nach seinem Zug, bis die Gewinnbedingung erfüllt ist.

Die Spielklasse ist eine Ableitung von "Thread", weshalb zum Starten des Spieles die start()-Methode verwendet wird, um mehrere Spiele parallel laufen zu lassen. Dies ist ohne Threadlocks möglich, da sich Spiele untereinander keinerlei Daten teilen müssen.

Spielfeld

Das Spielfeld ist eine einfache Liste von Ganzzahlen. Ich habe Listen gewählt, da diese in Python sehr einfach zu händeln sind und zu den gebräuchlichen Mitteln gehören. Nativ gibt es in Python keine Arrays, diese sind erst in einem Zusatzmodul namens "numpy" implementiert.

Logik (Teil der Modellschicht)

Spieler

Nur die Spieler enthalten die Logik, die für das Spiel benötigt wird. Das Spiel selbst legt nur noch fest, ob es ein gültiger Zug ist und ob die Gewinnbedingung erfüllt wurde. Diese KI wurde für einen Spieler vorgegeben und musste für den anderen vom Prüfling entwickelt werden. Die 2 Spieler mit ihrer individuellen Strategie haben jedoch Gemeinsamkeiten, weshalb sie diese von einer gemeinsamen Basisklasse Spieler erben.

Controller

Controller

Der Controller wertet die Eingaben aus der Datei für das Programm aus und startet das Turnier, wenn die Eingabe gültig war. Im Falle einer ungültigen Eingabe gibt der Controller den Fehler an die Konsole zurück. Zusätzlich zum geforderten Umfang bietet der Controller dem Benutzer eine Hilfe an, wenn das Programm mit dem Parameter "help" gestartet wird.

FeldBuilder

Der FeldBuilder ist ein Teil der Controllerstruktur und übernimmt den Aufbau des Spielfeldes. Dafür benötigt er den Dateinamen von der Eingabedatei. Er sucht darin nach der Zeile, die den Aufbau des Spielfeldes beschreibt und baut das Spielfeld dann nach diesen Vorgaben.

View

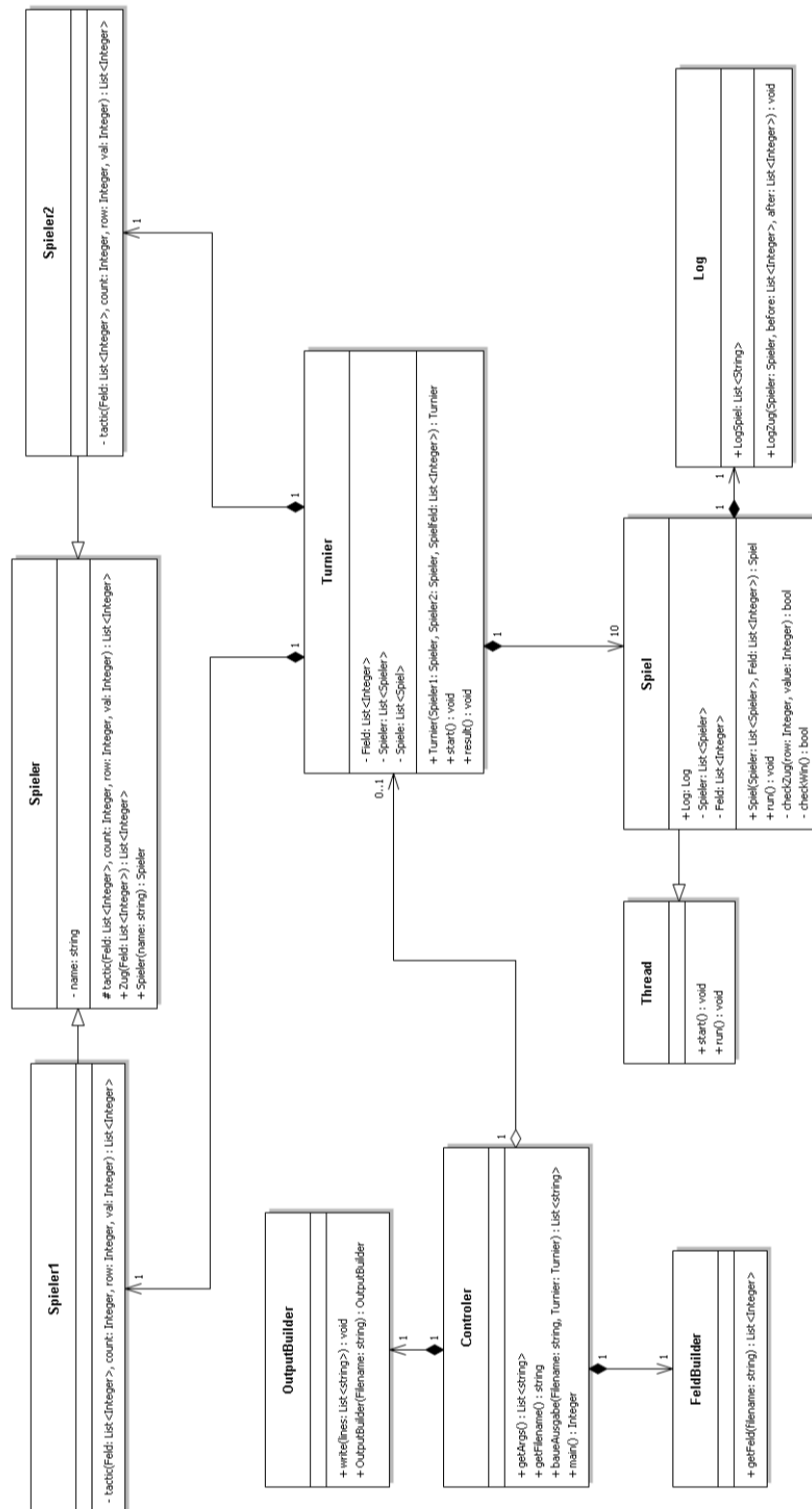
OutputBuilder

Der OutputBuilder schreibt in die ihm vorgegebene Datei den Inhalt, der ihm übergebenen Liste, gefüllt mit Strings. Dies ist die in der Aufgabenstellung geforderte Form der Ausgabe.

3.1.2 Lösungsalgorithmus

Wie im Analyseteil beschrieben, versucht Spieler1 den Zustand herzustellen, dass nur noch 2 Reihen mit gleicher Anzahl an Hölzern existieren. Dazu eliminiert er vorher in jedem Schritt die Reihe mit den wenigsten Hölzern und versucht, wenn nur noch 2 Reihen übrig sind, diese auszugleichen. Bleibt nur noch eine Reihe übrig, nimmt er dort alle Hölzer und gewinnt.

3.2 Konzeption UML-Klassendiagramm



3.3 Klassen

3.3.1 Controller

getArgs

“getArgs” Schreibt die vom System übergebenen Parameter in eine weitere Liste.

getFilename

“getFilename” prüft, ob in den Args genau ein Argument übergeben wurde und prüft, ob dieses einen Dateipfad enthält.

Falls als Argument ‘help’ übergeben wurde, erzeugt die Methode eine `UsrNeedHelp` Exception.

Falls kein oder mehr als ein Argument übergeben wurde erzeugt sie eine `UsrError` Exception.

Falls genau ein Argument übergeben wurde, prüft die Methode, ob dies auf ‘.in’ endet und der String einen Pfad zu einer existierenden Datei enthält. Ist dies nicht der Fall erzeugt sie ebenfalls eine `UsrError` Exception mit entsprechender Fehlermeldung.

baueAusgabe

“baueAusgabe” fügt den Inhalt der Eingabedatei mit dem Turnierergebnis zusammen und baut so eine Stringliste, die alle Informationen enthält, die ausgegeben werden sollen.

main

“main” ist die Einstiegsfunktion für das Programm. Sie lässt erst alle Eingaben prüfen, führt dann das Turnier aus und veranlasst dann die Ausgabe des Ergebnisses. Tritt währenddessen eine Exception auf, wird diese hier abgefangen und ausgewertet. 2 Arten von Exceptions werden erwartet und behandelt. Das sind die beiden selbst definierten Ausnahmefehler `UsrError` und `UsrNeedHelp`. Beide werden in der mitgelieferten Unit `UsrError` eingeführt. `UsrError` wird erzeugt, wenn der Benutzer eine falsche Eingabe gemacht hat. Es wird dann die mitgegebene Fehlermeldung in der Konsole ausgegeben und das Programm mit `sys.exit(1)` beendet.

`UsrNeedHelp` wird aufgerufen, wenn der Benutzer ‘help’ anstatt eines Dateipfades eingegeben hat. Daraufhin wird der Hilfetext auf der Konsole ausgegeben und das Programm mit `sys.exit(0)` beendet. Wird eine von den Typen abweichende Exception abgefangen, wird diese ausgegeben und das Programm mit `sys.exit(2)` beendet. Dies deutet auf einen unerwarteten Programmabbruch hin und soll so nicht passieren.

3.3.2 FeldBuilder

getFeld

“getFeld” baut aus der in der dem Programm übergebenen Datei das Spielfeld. Dabei werden alle Kommentarzeilen, die vor den Startbedingungen stehen, ignoriert. Kommentarzeilen beginnen mit einem “#”. Ist keine Zeile mit Startbedingungen vorhanden, erzeugt getFeld eine `UsrError Exception`. Als Zeile mit Startbedingungen wird die erste Zeile ohne “#” als erstes Zeichen angesehen. Diese soll bis zu 9, aber mindestens eine Ziffer enthalten. Ziffern sollen durch genau ein Leerzeichen voneinander getrennt werden. Ist dies nicht der Fall erzeugt die Methode ebenfalls eine `UsrError Exception` mit entsprechendem Statement.

3.3.3 OutputBuilder

__init__

“__init__” ist der Konstruktor des `OutputBuilder` Objektes. Dieser initialisiert das Feld “_Filename”, welches den Dateipfad zu der Ausgabedatei enthält.

write

“write” schreibt den Inhalt der als Parameter übergebenen Stringliste in die Ausgabedatei (_Filename). Ist diese Datei bereits vorhanden, wird der Inhalt überschrieben.

3.3.4 Spieler

__init__

“__init__” ist der Konstruktor des `Spieler` Objektes. Er initialisiert das Feld `name`.

_tactic

“_tactic” ist eine Methode die in `Spieler` eingeführt wird, aber von den Erben überschrieben werden soll.

Zug

“Zug” wird vom Spiel aufgerufen, wenn der Spieler einen Spielzug ausführen soll. Alle Spieler die diese Methode nicht überschreiben, gewinnen, falls nur noch Hölzer in einer Reihe existieren. Sonst wird die individuelle Strategie, die in `_tactic` implementiert ist, angewendet.

3.3.4.1 Spieler1

_tactic

“_tactic” überschreibt die `_tactic` Methode aus der Parent-Class `Spieler` und implementiert das Verhalten von `Spieler1`.

3.3.4.2 Spieler2

_tactic

“_tactic” überschreibt die `_tactic` Methode aus der Parent-Class `Spieler` und implementiert das Verhalten von `Spieler2`.

3.3.5 Turnier

`__init__`

“`__init__`” ist der Konstruktor des Spieler Objektes. Er initialisiert die Felder `_Spieler`, `_Spielfeld`, `_Spiele`.

`_laeuftSpiel`

“`_laeuftSpiel`” prüft ob noch mindestens ein Spielthread aktiv ist.

`start`

“`start`” legt im Feld `_Spiele` die in `count` übergebene Anzahl Spiele an und startet diese.

`result`

“`result`” ermittelt die Anzahl an gewonnenen Spielen, also die Anzahl derer, in denen der letzte Zug von Spieler1, das ist der erste an die Spiele übergebene Spieler, ausgeführt wurde. Es wird eine Stringliste mit den in der Ausgabe benötigten Zeilen erstellt. In der Methode wird sich jeweils das letzte gewonnene und verlorene Spiel gemerkt und als Beispiel in die Stringliste eingefügt.

3.3.6 Spiel

`__init__`

“`__init__`” ist der Konstruktor des Spieler Objektes. Er initialisiert die Felder `_Spieler`, `_Spielfeld`, `winner`, `Log` und ruft den Konstruktor der Superklasse “Thread” auf.

`_checkZug`

“`_checkZug`” prüft, ob der vom Spieler gewählte Zug gültig ist und gibt dann ein `True` zurück, andernfalls ein `False`.

`_checkWin`

“`_checkWin`” prüft, ob das Spiel gewonnen wurde, also alle Hölzer aufgenommen wurden und gibt dann ein `True` zurück, andernfalls ein `False`.

`run`

“`run`” wird durch die Thread-Methode “`start`” in einem eigenen Thread aufgerufen. Es werden solange abwechselnd Spieler nach dem nächsten Zug gefragt, bis die Gewinnbedingung erfüllt ist.

3.3.7 Log

`__init__`

“`__init__`” ist der Konstruktor des Spieler Objektes. Er initialisiert das Feld `LogSpiel`.

LogZug

“`LogZug`” baut aus dem Spieler, dem Spielfeld vor dem Zug und dem Spielfeld nach dem Zug einen Logeintrag zusammen und hängt diesen an `LogSpiel` an.

3.3.8 **UsrError**

`__init__`

“`__init__`” ist der Konstruktor des ExceptionObjektes und initialisiert das Feld `msg`.

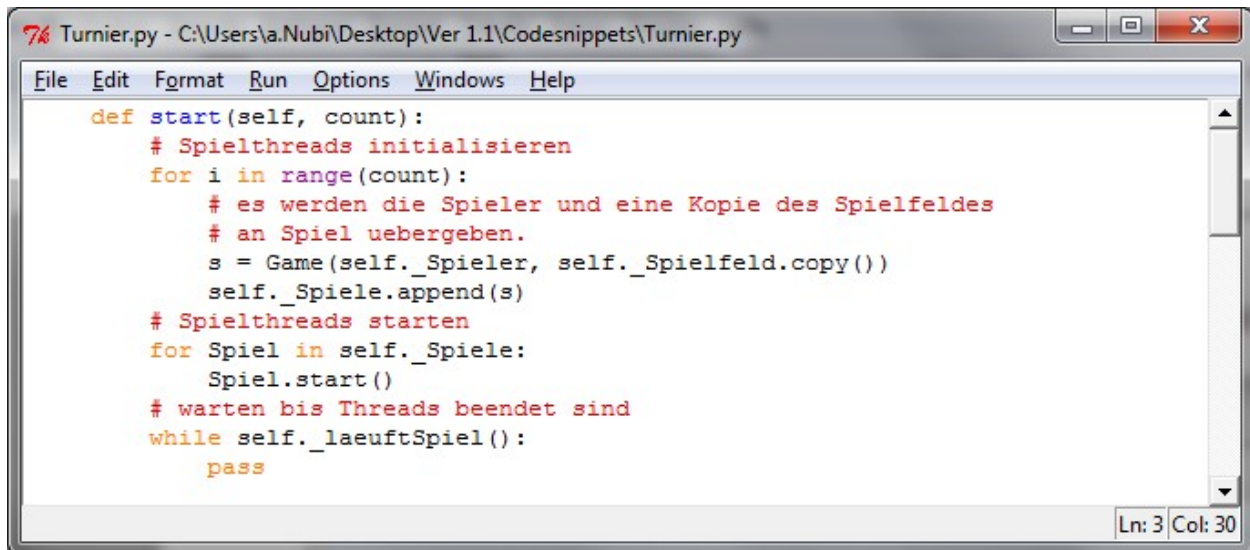
3.3.9 **UsrNeedHelp**

`__init__`

“`__init__`” ist der Konstruktor des ExceptionObjektes. Die Klasse hat keine von Exception abweichenden Eigenschaften. Sie wird nur wegen des Klassennamens angelegt.

3.4 Code Snippets

Damit der Code hier nicht zu viel Platz einnimmt, habe ich einige Kommentarzeilen entfernt. Die ausführliche Dokumentation kann beigefügten Dokumentation oder dem Quelltext entnommen werden.



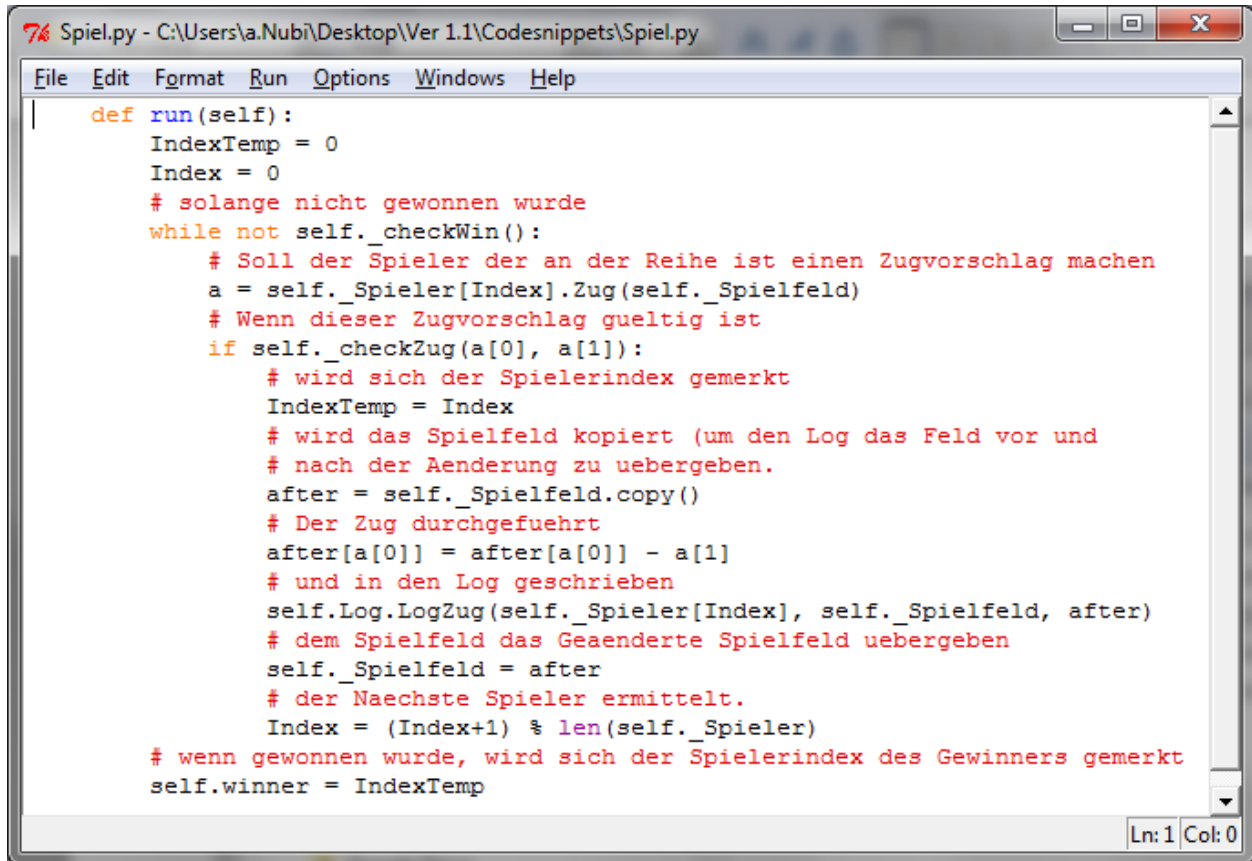
```
7% Turnier.py - C:\Users\A.Nubi\Desktop\Ver 1.1\Codesnippets\Turnier.py
File Edit Format Run Options Windows Help

def start(self, count):
    # Spielthreads initialisieren
    for i in range(count):
        # es werden die Spieler und eine Kopie des Spielfeldes
        # an Spiel uebergeben.
        s = Game(self._Spieler, self._Spielfeld.copy())
        self._Spiele.append(s)
    # Spielthreads starten
    for Spiel in self._Spiele:
        Spiel.start()
    # warten bis Threads beendet sind
    while self._laeuftSpiel():
        pass

Ln: 3 Col: 30
```

Turnier.start():

Die Anzahl benötigter Spiele für das Turnier wird der Methode vom Controller übergeben. Nun erstellt diese die geforderte Anzahl an Spielen und startet diese. Anschließend wird gewartet, bis alle Spiele beendet wurden.



```
74 Spiel.py - C:\Users\A.Nubi\Desktop\Ver 1.1\Codesnippets\Spiel.py
File Edit Format Run Options Windows Help

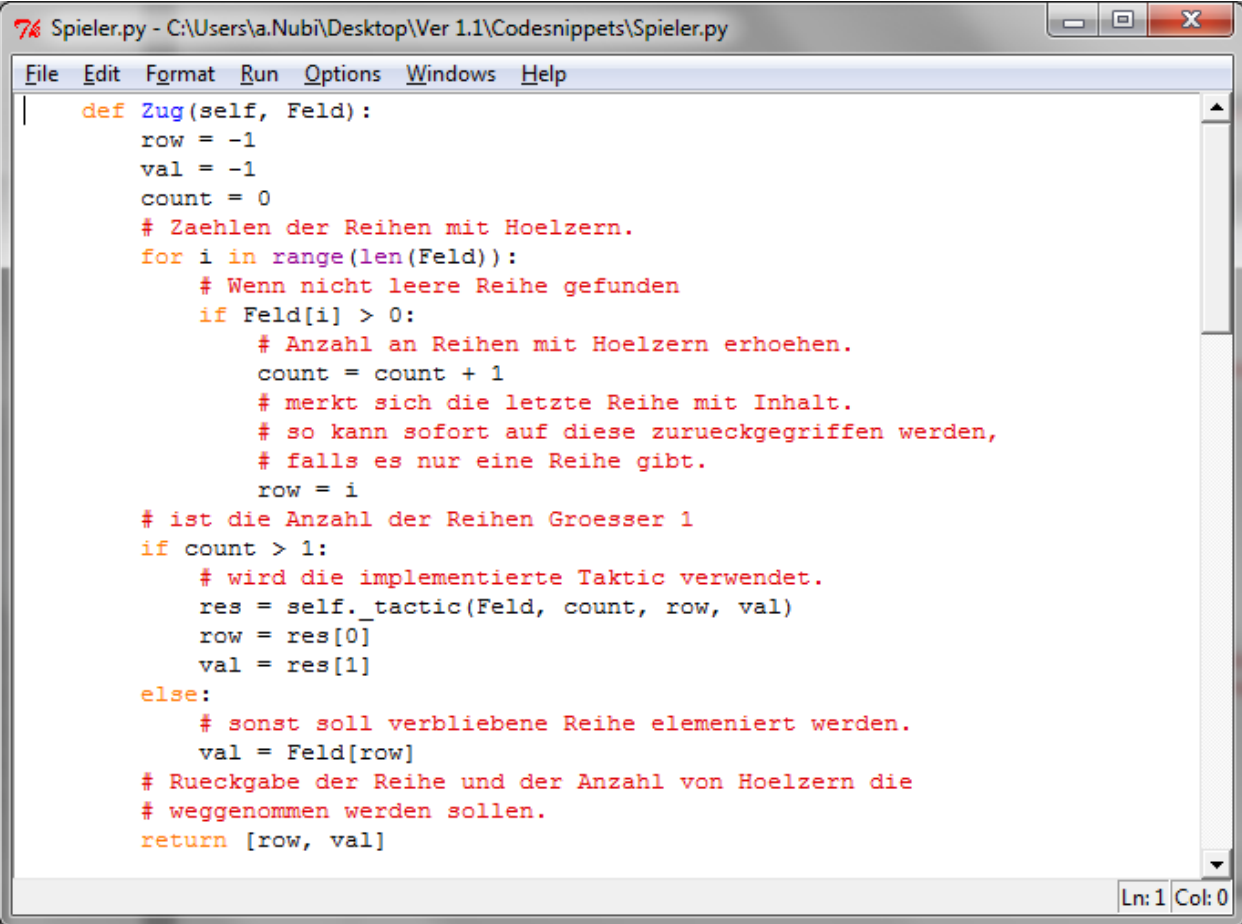
def run(self):
    IndexTemp = 0
    Index = 0
    # solange nicht gewonnen wurde
    while not self._checkWin():
        # Soll der Spieler der an der Reihe ist einen Zugvorschlag machen
        a = self._Spieler[Index].Zug(self._Spielfeld)
        # Wenn dieser Zugvorschlag gueltig ist
        if self._checkZug(a[0], a[1]):
            # wird sich der Spielerindex gemerkt
            IndexTemp = Index
            # wird das Spielfeld kopiert (um den Log das Feld vor und
            # nach der Aenderung zu uebergeben.
            after = self._Spielfeld.copy()
            # Der Zug durchgefuehrt
            after[a[0]] = after[a[0]] - a[1]
            # und in den Log geschrieben
            self.Log.LogZug(self._Spieler[Index], self._Spielfeld, after)
            # dem Spielfeld das Geaenderte Spielfeld uebergeben
            self._Spielfeld = after
            # der Naechste Spieler ermittelt.
            Index = (Index+1) % len(self._Spieler)
        # wenn gewonnen wurde, wird sich der Spielerindex des Gewinners gemerkt
    self.winner = IndexTemp

Ln: 1 Col: 0
```

Spiel.run():

Die vorher aufgerufene Methode Spiel.start() ruft die Methode run() als eigenen Thread auf. Diese simuliert nun das eigentliche Spiel und merkt sich den Gewinner.

Da die einzelnen Spieler eigene Strategien verfolgen, fragt run() den Spieler der an der Reihe ist nach seinem nächsten Zug.



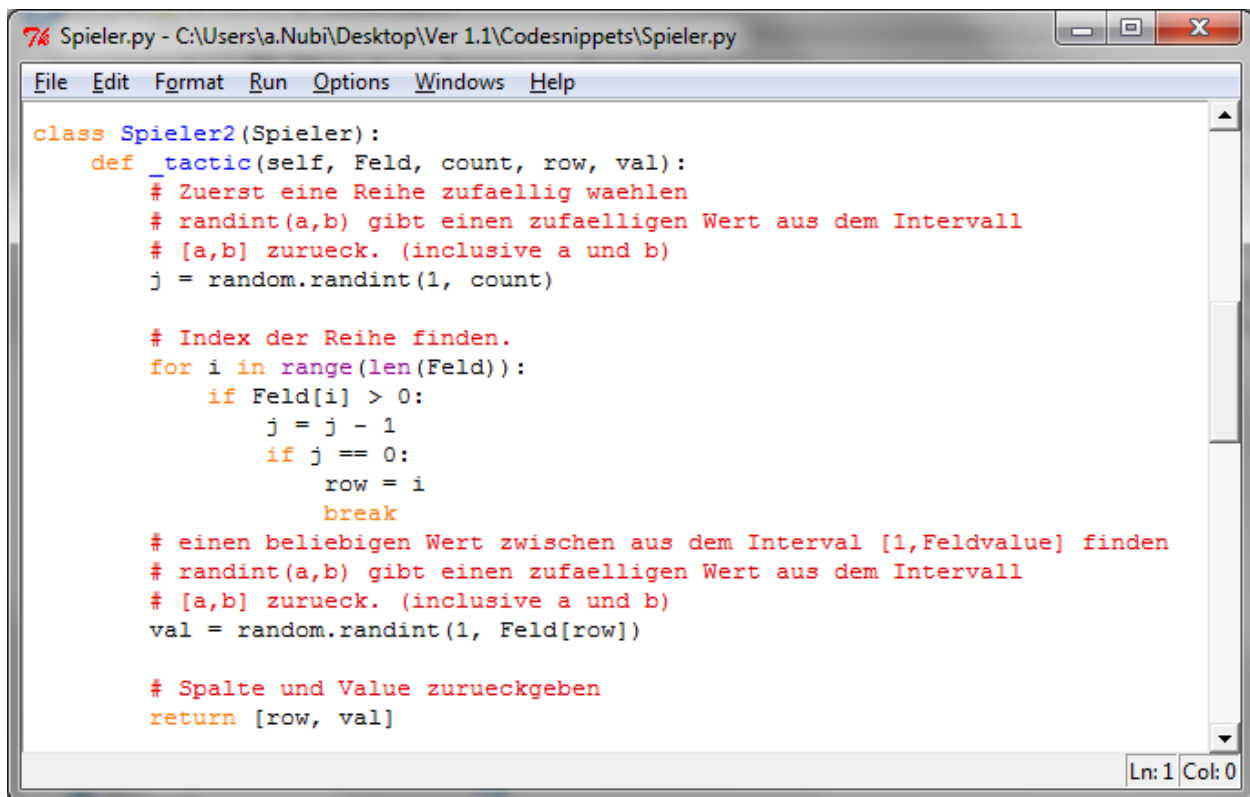
```
7% Spieler.py - C:\Users\A.Nubi\Desktop\Ver 1.1\Codesnippets\Spieler.py
File Edit Format Run Options Windows Help

def Zug(self, Feld):
    row = -1
    val = -1
    count = 0
    # Zaehlen der Reihen mit Hoelzern.
    for i in range(len(Feld)):
        # Wenn nicht leere Reihe gefunden
        if Feld[i] > 0:
            # Anzahl an Reihen mit Hoelzern erhoehen.
            count = count + 1
            # merkt sich die letzte Reihe mit Inhalt.
            # so kann sofort auf diese zurueckgegriffen werden,
            # falls es nur eine Reihe gibt.
            row = i
    # ist die Anzahl der Reihen Groesser 1
    if count > 1:
        # wird die implementierte Tactic verwendet.
        res = self._tactic(Feld, count, row, val)
        row = res[0]
        val = res[1]
    else:
        # sonst soll verbliebene Reihe elemeniert werden.
        val = Feld[row]
    # Rueckgabe der Reihe und der Anzahl von Hoelzern die
    # weggenommen werden sollen.
    return [row, val]

Ln: 1 Col: 0
```

Spieler.Zug()

Jeder Spieler, der die Zug Methode der Basisklasse nicht überschreibt, gewinnt das Spiel, sobald nur noch Hölzer in einer Reihe existieren. Ist die Anzahl der Reihen mit Hölzern jedoch größer 1 wird die Strategie des Spielers mittels `_tactic()` verwendet. Jeder Spieler, der von der Basisklasse ableitet soll `_tactic` überschreiben, um das sicher zu stellen, erzeugt `_tactic` in der Basisklasse eine Exception.



```
7% Spieler.py - C:\Users\A.Nubi\Desktop\Ver 1.1\Codesnippets\Spieler.py
File Edit Format Run Options Windows Help

class Spieler2(Spieler):
    def _tactic(self, Feld, count, row, val):
        # Zuerst eine Reihe zufaellig waehlen
        # randint(a,b) gibt einen zufaelligen Wert aus dem Intervall
        # [a,b] zurueck. (inclusive a und b)
        j = random.randint(1, count)

        # Index der Reihe finden.
        for i in range(len(Feld)):
            if Feld[i] > 0:
                j = j - 1
                if j == 0:
                    row = i
                    break

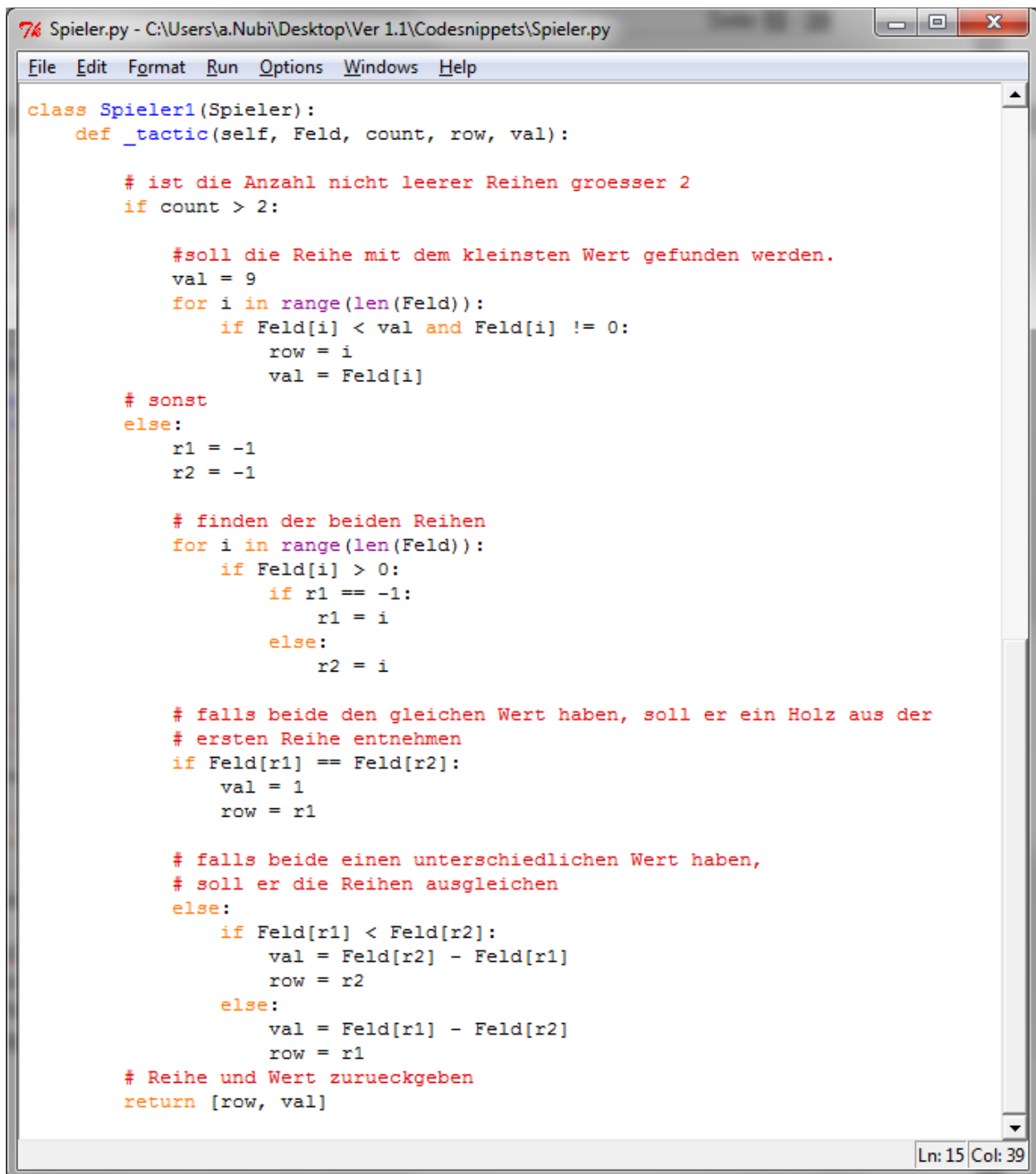
        # einen beliebigen Wert zwischen aus dem Interval [1,Feldvalue] finden
        # randint(a,b) gibt einen zufaelligen Wert aus dem Intervall
        # [a,b] zurueck. (inclusive a und b)
        val = random.randint(1, Feld[row])

        # Spalte und Value zurueckgeben
        return [row, val]
```

Ln: 1 Col: 0

Spieler2._tactic():

Nun muss für jeden Spieler nur noch eine eigene Strategie festgelegt werden, für den Fall, dass mehr als 1 Reihe noch Hölzer hat. Spieler2 setzt in diesem Fall auf den Zufall und wählt aus einer beliebigen Reihe eine beliebige Anzahl Hölzer.



```
7% Spieler.py - C:\Users\A.Nubi\Desktop\Ver 1.1\Codesnippets\Spieler.py
File Edit Format Run Options Windows Help

class Spieler1(Spieler):
    def _tactic(self, Feld, count, row, val):

        # ist die Anzahl nicht leerer Reihen groesser 2
        if count > 2:

            #soll die Reihe mit dem kleinsten Wert gefunden werden.
            val = 9
            for i in range(len(Feld)):
                if Feld[i] < val and Feld[i] != 0:
                    row = i
                    val = Feld[i]

        # sonst
        else:
            r1 = -1
            r2 = -1

            # finden der beiden Reihen
            for i in range(len(Feld)):
                if Feld[i] > 0:
                    if r1 == -1:
                        r1 = i
                    else:
                        r2 = i

            # falls beide den gleichen Wert haben, soll er ein Holz aus der
            # ersten Reihe entnehmen
            if Feld[r1] == Feld[r2]:
                val = 1
                row = r1

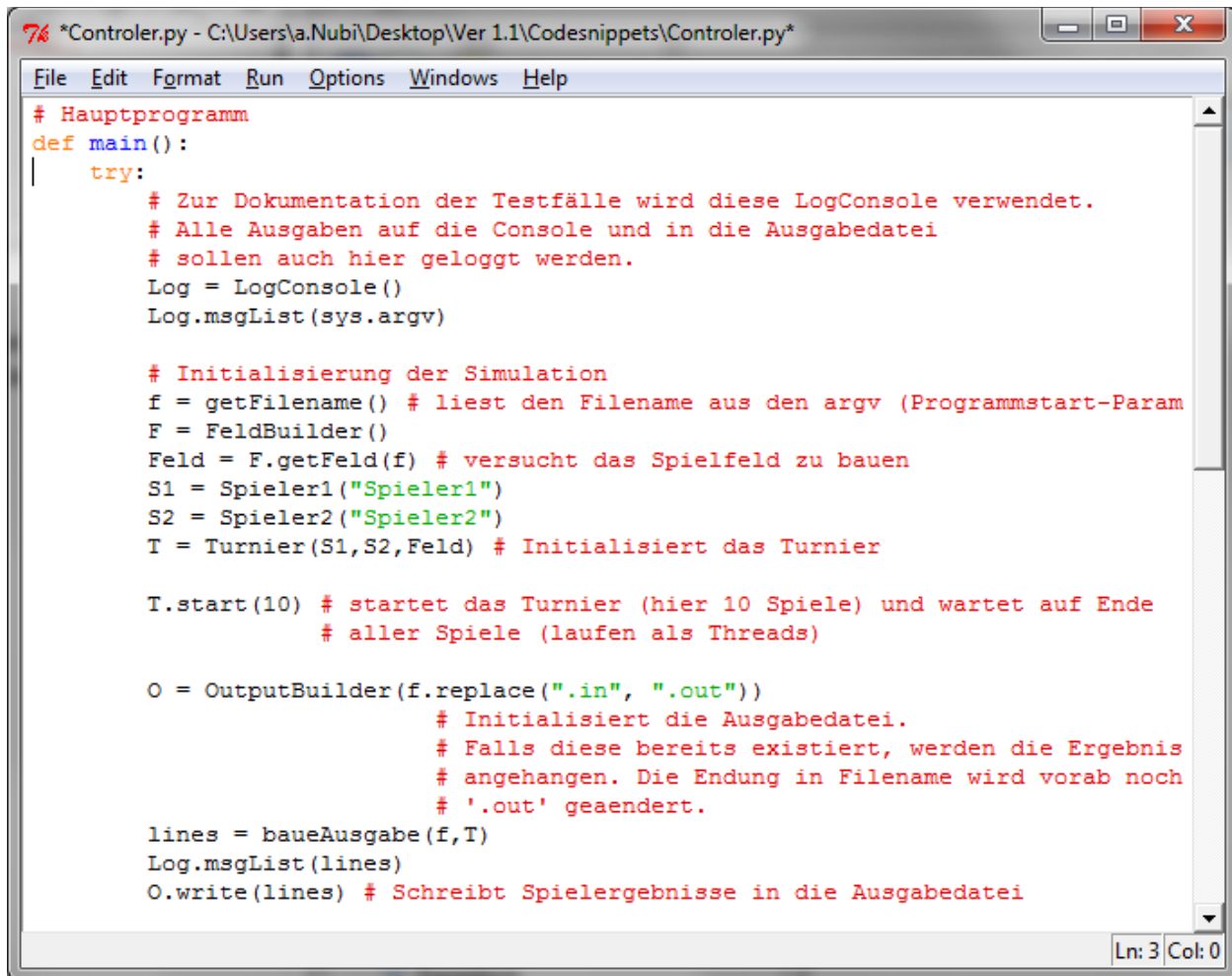
            # falls beide einen unterschiedlichen Wert haben,
            # soll er die Reihen ausgleichen
            else:
                if Feld[r1] < Feld[r2]:
                    val = Feld[r2] - Feld[r1]
                    row = r2
                else:
                    val = Feld[r1] - Feld[r2]
                    row = r1

            # Reihe und Wert zurueckgeben
            return [row, val]
```

Ln: 15 Col: 39

Spieler1._tactic():

Die Strategie von Spieler1 wurde am ersten Tag der Prüfung festgelegt. Er soll, solange mehr als 2 Reihen existieren, die jeweils kleinste vollständig aufnehmen und wenn nur noch 2 existieren, versuchen, die Reihen auszugleichen. Ist dies nicht möglich, soll er nur ein Holz aus einer der beiden Reihen nehmen.

The image shows a screenshot of a Python IDE window titled "7% *Controler.py - C:\Users\A.Nubi\Desktop\Ver 1.1\Codesnippets\Controler.py*". The window has a menu bar with "File", "Edit", "Format", "Run", "Options", "Windows", and "Help". The code is written in Python and is as follows:

```
# Hauptprogramm
def main():
    try:
        # Zur Dokumentation der Testfälle wird diese LogConsole verwendet.
        # Alle Ausgaben auf die Console und in die Ausgabedatei
        # sollen auch hier geloggt werden.
        Log = LogConsole()
        Log.msgList(sys.argv)

        # Initialisierung der Simulation
        f = getFilename() # liest den Filename aus den argv (Programmstart-Param
        F = FeldBuilder()
        Feld = F.getFeld(f) # versucht das Spielfeld zu bauen
        S1 = Spieler1("Spieler1")
        S2 = Spieler2("Spieler2")
        T = Turnier(S1,S2,Feld) # Initialisiert das Turnier

        T.start(10) # startet das Turnier (hier 10 Spiele) und wartet auf Ende
                   # aller Spiele (laufen als Threads)

        O = OutputBuilder(f.replace(".in", ".out"))
                   # Initialisiert die Ausgabedatei.
                   # Falls diese bereits existiert, werden die Ergebnis
                   # angehängen. Die Endung in Filename wird vorab noch
                   # '.out' geändert.

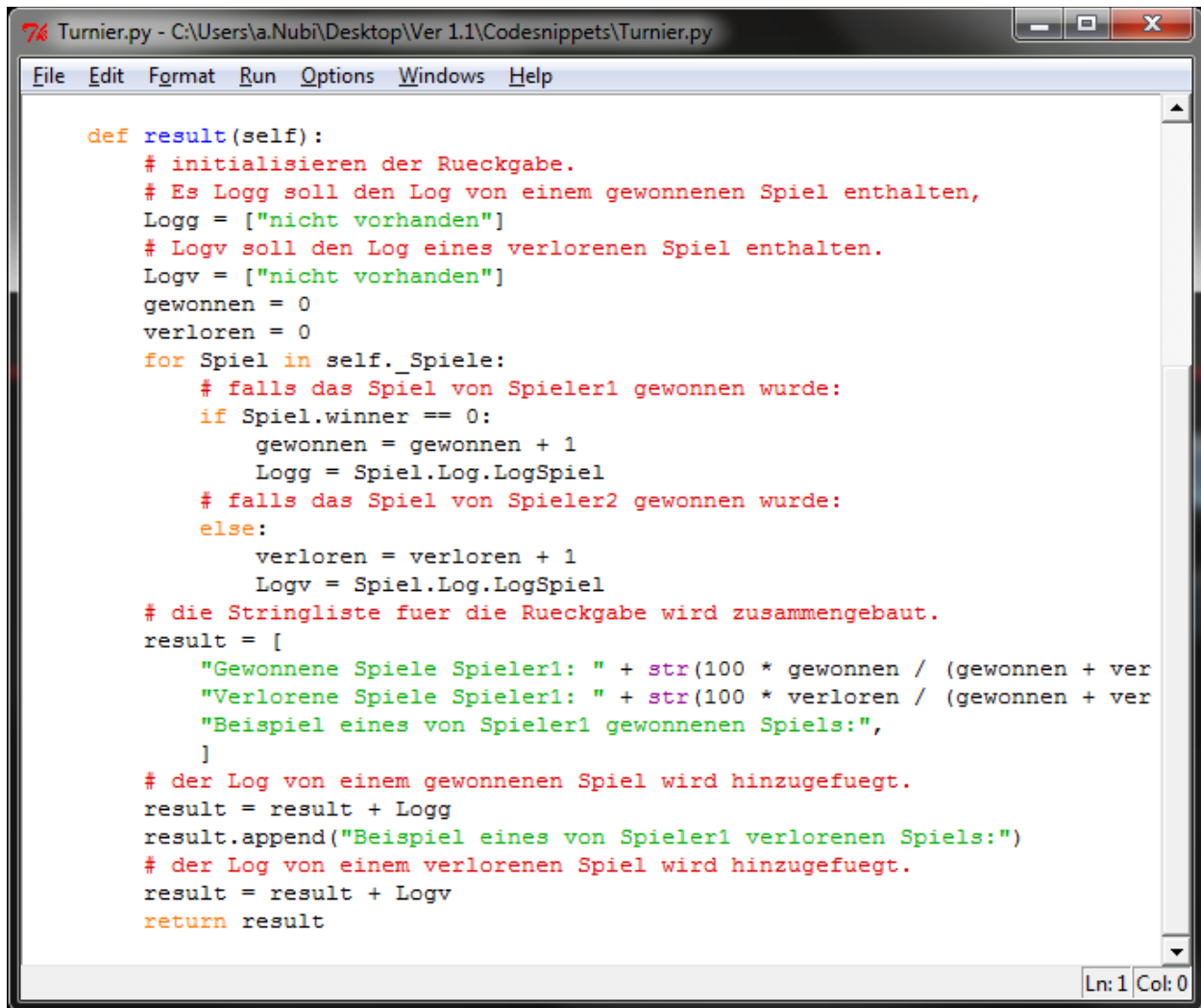
        lines = baueAusgabe(f,T)
        Log.msgList(lines)
        O.write(lines) # Schreibt Spielergebnisse in die Ausgabedatei
```

The status bar at the bottom right indicates "Ln: 3 Col: 0".

Controler.main():

Dies ist die Mainmethode des Programms. Abweichend von der Aufgabenstellung habe ich einen Log geschaffen, der alle Ausgaben des Programms in die "log.txt" schreibt. Somit können Testreihen ausgewertet werden und die Testfälle auch dokumentiert werden. Damit die Prüfungskommission dies nachvollziehen kann, wurde diese Funktionalität im Programm belassen.

Main übernimmt die Kommunikation zwischen dem User-Interface und dem Modell. Auf diese Weise wurde eine Art MVC-Pattern im Programm realisiert.



```
7% Turnier.py - C:\Users\A.Nubi\Desktop\Ver 1.1\Codesnippets\Turnier.py
File Edit Format Run Options Windows Help

def result(self):
    # initialisieren der Rueckgabe.
    # Es Logg soll den Log von einem gewonnenen Spiel enthalten,
    Logg = ["nicht vorhanden"]
    # Logv soll den Log eines verlorenen Spiel enthalten.
    Logv = ["nicht vorhanden"]
    gewonnen = 0
    verloren = 0
    for Spiel in self._Spiele:
        # falls das Spiel von Spieler1 gewonnen wurde:
        if Spiel.winner == 0:
            gewonnen = gewonnen + 1
            Logg = Spiel.Log.LogSpiel
        # falls das Spiel von Spieler2 gewonnen wurde:
        else:
            verloren = verloren + 1
            Logv = Spiel.Log.LogSpiel
    # die Stringliste fuer die Rueckgabe wird zusammengebaut.
    result = [
        "Gewonnene Spiele Spieler1: " + str(100 * gewonnen / (gewonnen + ver
        "Verlorene Spiele Spieler1: " + str(100 * verloren / (gewonnen + ver
        "Beispiel eines von Spieler1 gewonnenen Spiels:",
    ]
    # der Log von einem gewonnenen Spiel wird hinzugefuegt.
    result = result + Logg
    result.append("Beispiel eines von Spieler1 verlorenen Spiels:")
    # der Log von einem verlorenen Spiel wird hinzugefuegt.
    result = result + Logv
    return result

Ln: 1 Col: 0
```

Turnier.result():

Diese Methode baut eine Liste mit Strings zusammen, die anschließend der View übergeben wird. Es wird die Prozentzahl aller gewonnenen und verlorenen Spiele ermittelt und in die Auswertung eingebaut. Am Ende wird, falls vorhanden, jeweils ein Beispiel für ein gewonnenes und ein verlorenes Spiel angehängt.

3.5 Ausführliche Dokumentation

Eine ausführlichere Dokumentation aller Methoden und Attribute ist dieser Dokumentation in digitaler Form als HTML-Dokumentation beigelegt.

4 Benutzerhandbuch

4.1 Systemvoraussetzung

Das Programm benötigt auf dem System Python3.3. Es wurde nicht mit niedrigeren Versionen getestet. Python3.3 gibt es für Windows, Linux-Distributionen und iOS.

Auf dem der Prüfung beigelegten Datenträger finden Sie eine Installationsdatei für Windows und Linux.

4.2 Programmstart

Das Programm wird im Programmverzeichnis mit dem Befehl:
Controler.py [Dateiname.in]
aufgerufen.

Ein Ablauf einer Testsequenz ist unter Windows durch Ausführung von „start.bat“ im Programmverzeichnis möglich.

4.3 Bedienung

Nach dem Start der Simulation hat der Benutzer keine Möglichkeit mehr in das Programm einzugreifen. Die Ausgabe erfolgt in eine gleichnamige Datei wie die Eingabe, nur dass die Dateiendung sich unterscheidet.

Aus der Datei [Dateiname.in] wird gelesen und in die Datei [Dateiname.out] wird geschrieben.

5 Beispiele

Es werden in diesem Abschnitt einige Beispiele vorgestellt und deren Lösung präsentiert. Da der Spieler2 seine Züge zufällig wählt, sind diese Beispiele in der Regel nicht reproduzierbar. Die Beispiele wurden der auf dem Datenträger beigefügten Datei 'tests01.txt' im Ordner "/Tests" entnommen.

Beispiel 1

Dieses Beispiel sollte eine Fehlermeldung erzeugen, da die übergebene Datei nicht dem geforderten Typ entspricht.

Eingabe:

Controler.py test.out

Ausgabe:

Datei falschen Typs uebergeben

rufe 'Controler.py help' auf um mehr zu erfahren.

Beispiel 2

Dieses Beispiel sollte eine Fehlermeldung erzeugen, da die übergebene Datei nicht dem geforderten Typ entspricht.

Eingabe:

Controler.py test

Ausgabe:

Datei falschen Typs uebergeben

rufe 'Controler.py help' auf um mehr zu erfahren.

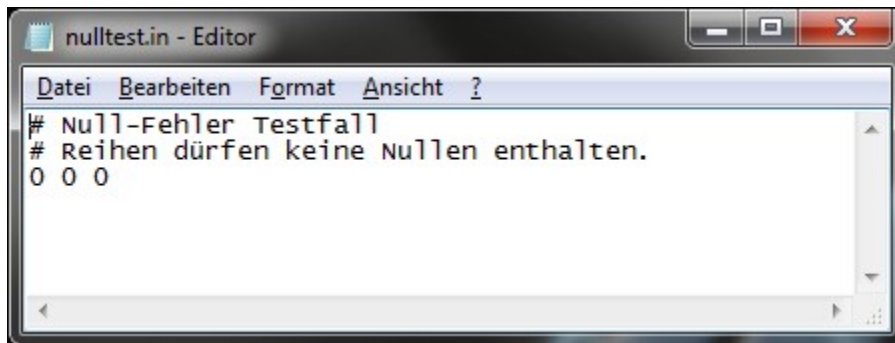
Beispiel 3

Dieses Beispiel sollte eine Fehlermeldung erzeugen, da mit der Eingabedatei Startreihen mit dem Wert "0" übergeben wurden.

Eingabe:

Controler.py nulltest.in

Eingabedatei:



```
# Null-Fehler Testfall
# Reihen dürfen keine Nullen enthalten.
0 0 0
```

Ausgabe:

Es dürfen keine Startreihen mit dem Wert '0' übergeben werden!
rufe 'Controler.py help' auf um mehr zu erfahren.

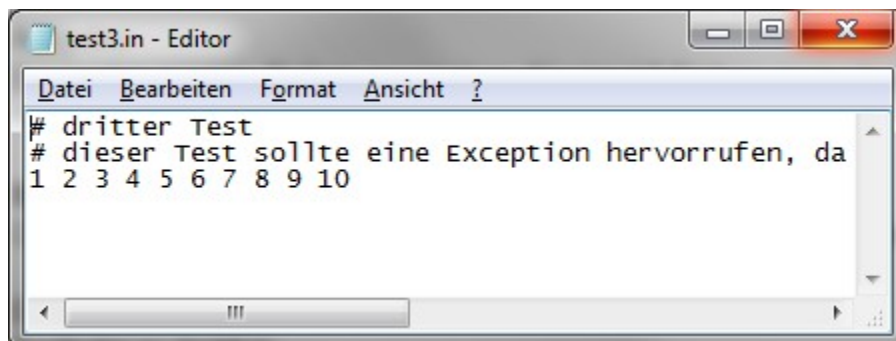
Beispiel 4

Dieses Beispiel sollte eine Fehlermeldung erzeugen, da mehr als 9 Reihen definiert wurden. Das ist laut Aufgabenstellung nicht erlaubt.

Eingabe:

Controler.py test3.in

Eingabedatei:



```
# dritter Test
# dieser Test sollte eine Exception hervorrufen, da
1 2 3 4 5 6 7 8 9 10
```

Ausgabe:

Es wurden zu viele Reihen angegeben. Maximal 9 Reihen sind erlaubt
rufe 'Controler.py help' auf um mehr zu erfahren.

Beispiel 5

Dieses Beispiel sollte eine Fehlermeldung erzeugen, da die Eingabedatei nicht existiert.

Eingabe:

Controler.py test4.in

Ausgabe:

Datei test4.in kann nicht gefunden werden
rufe 'Controler.py help' auf um mehr zu erfahren.

Beispiel 6

Dieses Beispiel sollte eine Fehlermeldung erzeugen, da keine Eingabedatei übergeben wurde.

Eingabe:

Controler.py

Ausgabe:

Keinen Dateinamen angegeben
rufe 'Controler.py help' auf um mehr zu erfahren.

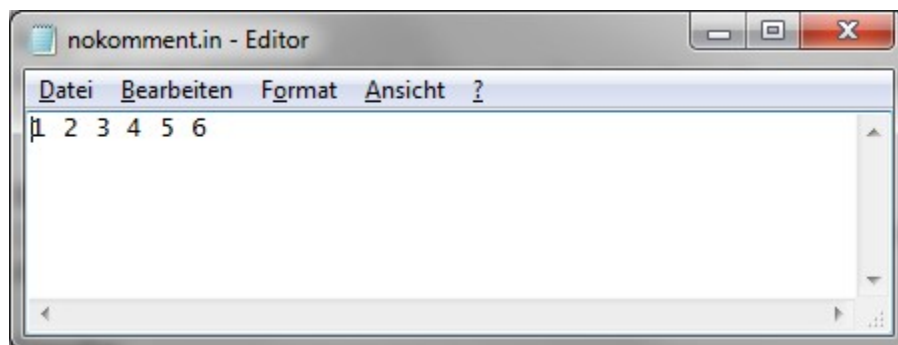
Beispiel 7

Dieses Beispiel sollte eine Fehlermeldung erzeugen, da in der Eingabedatei die Kommentarzeile fehlt.

Eingabe:

Controler.py nokomment.in

Eingabedatei:



Ausgabe:

In der Eingabedatei soll mindestens eine Kommentarzeile angegeben sein
rufe 'Controler.py help' auf um mehr zu erfahren.

Beispiel 8

Dieses Beispiel sollte die Hilfe des Programms ausgeben. Dies weicht von der Aufgabenstellung ab.

Eingabe:

Controler.py help

Ausgabe:

Controller zum starten der Simulation des Nim-Spiels.

Benutzung:

Controler.py [InputFilename]

InputFilename muss vom Typ ".in" sein, sonst kann das Programm die Datei nicht oeffnen. Ausgegeben wird das Ergebnis der Simulation in eine gleichnamige Datei mit der Endung ".out"

Die Inputfile kann beliebig viele Kommentarzeilen vor den Startbedingungen enthalten. Die Startbedingungen geben dann die Anzahl an Reihen und deren Groesse an. Es darf 1 bis 9 Reihen geben mit einem Startwert zwischen 1 und 9. Angegeben werden diese durch 1 bis 9 durch Leerzeichen voneinander getrennte Ganzzahlen zwischen 1 und 9 in der ersten Zeile ohne Kommentar.

Alle folgenden Zeilen werden ignoriert.

Im weiteren Verlauf wurde die Logik von Spieler1 getestet. Er sollte einen überwiegenden Teil aller Spiele gewinnen, sofern keine für ihn ungünstigen Startbedingungen vorlagen. Die Eingabedatei braucht hier nicht extra aufgeführt zu werden, da deren Inhalt in der Ausgabe erscheint.

Beispiel 9

Bei diesem Beispiel sollte überwiegend Spieler 1 gewinnen.

Eingabe:

Controler.py ihk1.in

Ausgabe:

IHK Beispiel 1

3 4 5

Gewonnene Spiele Spieler1: 100.0%

Verlorene Spiele Spieler1: 0.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (3, 4, 5) -> (0, 4, 5)

Zug2 , Spieler2 : (0, 4, 5) -> (0, 1, 5)

Zug3 , Spieler1 : (0, 1, 5) -> (0, 1, 1)

Zug4 , Spieler2 : (0, 1, 1) -> (0, 1, 0)

Zug5 , Spieler1 : (0, 1, 0) -> (0, 0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:

nicht vorhanden

Beispiel 10

Bei diesem Beispiel ist Spieler2 im Vorteil. Es sollte also Spiele geben, in denen Spieler2 gewinnt.

Eingabe:

Controler.py ihk2.in

Ausgabe:

IHK Beispiel 2

3 3

Gewonnene Spiele Spieler1: 90.0%

Verlorene Spiele Spieler1: 10.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (3, 3) -> (2, 3)

Zug2 , Spieler2 : (2, 3) -> (1, 3)

Zug3 , Spieler1 : (1, 3) -> (1, 1)

Zug4 , Spieler2 : (1, 1) -> (1, 0)

Zug5 , Spieler1 : (1, 0) -> (0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:

Zug1 , Spieler1 : (3, 3) -> (2, 3)

Zug2 , Spieler2 : (2, 3) -> (2, 2)

Zug3 , Spieler1 : (2, 2) -> (1, 2)

Zug4 , Spieler2 : (1, 2) -> (1, 1)

Zug5 , Spieler1 : (1, 1) -> (0, 1)

Zug6 , Spieler2 : (0, 1) -> (0, 0)

Beispiel 11

Bei diesem Beispiel ist Spieler1 im Vorteil und muss alle Spiele gewinnen.

Eingabe:

Controler.py ihk3.in

Ausgabe:

IHK Beispiel 3

hier gewinnt immer Spieler1

2 3

Gewonnene Spiele Spieler1: 100.0%

Verlorene Spiele Spieler1: 0.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (2, 3) -> (2, 2)

Zug2 , Spieler2 : (2, 2) -> (0, 2)

Zug3 , Spieler1 : (0, 2) -> (0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:

nicht vorhanden

Beispiel 12

Bei diesem Beispiel sollte Spieler1 die meisten Spiele gewinnen.

Eingabe:

Controler.py ihk4.in

Ausgabe:

IHK Beispiel 4

1 2 3 4 5 6 7 8 9

Gewonnene Spiele Spieler1: 100.0%

Verlorene Spiele Spieler1: 0.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (1, 2, 3, 4, 5, 6, 7, 8, 9) -> (0, 2, 3, 4, 5, 6, 7, 8, 9)

Zug2 , Spieler2 : (0, 2, 3, 4, 5, 6, 7, 8, 9) -> (0, 2, 3, 2, 5, 6, 7, 8, 9)
 Zug3 , Spieler1 : (0, 2, 3, 2, 5, 6, 7, 8, 9) -> (0, 0, 3, 2, 5, 6, 7, 8, 9)
 Zug4 , Spieler2 : (0, 0, 3, 2, 5, 6, 7, 8, 9) -> (0, 0, 3, 2, 4, 6, 7, 8, 9)
 Zug5 , Spieler1 : (0, 0, 3, 2, 4, 6, 7, 8, 9) -> (0, 0, 3, 0, 4, 6, 7, 8, 9)
 Zug6 , Spieler2 : (0, 0, 3, 0, 4, 6, 7, 8, 9) -> (0, 0, 3, 0, 4, 6, 7, 5, 9)
 Zug7 , Spieler1 : (0, 0, 3, 0, 4, 6, 7, 5, 9) -> (0, 0, 0, 0, 4, 6, 7, 5, 9)
 Zug8 , Spieler2 : (0, 0, 0, 0, 4, 6, 7, 5, 9) -> (0, 0, 0, 0, 4, 3, 7, 5, 9)
 Zug9 , Spieler1 : (0, 0, 0, 0, 4, 3, 7, 5, 9) -> (0, 0, 0, 0, 4, 0, 7, 5, 9)
 Zug10 , Spieler2 : (0, 0, 0, 0, 4, 0, 7, 5, 9) -> (0, 0, 0, 0, 4, 0, 3, 5, 9)
 Zug11 , Spieler1 : (0, 0, 0, 0, 4, 0, 3, 5, 9) -> (0, 0, 0, 0, 4, 0, 0, 5, 9)
 Zug12 , Spieler2 : (0, 0, 0, 0, 4, 0, 0, 5, 9) -> (0, 0, 0, 0, 4, 0, 0, 1, 9)
 Zug13 , Spieler1 : (0, 0, 0, 0, 4, 0, 0, 1, 9) -> (0, 0, 0, 0, 4, 0, 0, 0, 9)
 Zug14 , Spieler2 : (0, 0, 0, 0, 4, 0, 0, 0, 9) -> (0, 0, 0, 0, 4, 0, 0, 0, 3)
 Zug15 , Spieler1 : (0, 0, 0, 0, 4, 0, 0, 0, 3) -> (0, 0, 0, 0, 3, 0, 0, 0, 3)
 Zug16 , Spieler2 : (0, 0, 0, 0, 3, 0, 0, 0, 3) -> (0, 0, 0, 0, 0, 0, 0, 0, 3)
 Zug17 , Spieler1 : (0, 0, 0, 0, 0, 0, 0, 0, 3) -> (0, 0, 0, 0, 0, 0, 0, 0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:
 nicht vorhanden

Beispiel 13

Bei diesem Beispiel sollte Spieler1 die meisten Spiele gewinnen.

Eingabe:

Controler.py ihk5.in

Ausgabe:

IHK Beispiel 5

1 3 6 4

Gewonnene Spiele Spieler1: 100.0%

Verlorene Spiele Spieler1: 0.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (1, 3, 6, 4) -> (0, 3, 6, 4)

Zug2 , Spieler2 : (0, 3, 6, 4) -> (0, 3, 0, 4)

Zug3 , Spieler1 : (0, 3, 0, 4) -> (0, 3, 0, 3)

Zug4 , Spieler2 : (0, 3, 0, 3) -> (0, 1, 0, 3)

Zug5 , Spieler1 : (0, 1, 0, 3) -> (0, 1, 0, 1)

Zug6 , Spieler2 : (0, 1, 0, 1) -> (0, 0, 0, 1)

Zug7 , Spieler1 : (0, 0, 0, 1) -> (0, 0, 0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:
 nicht vorhanden

Beispiel 14

Bei diesem Beispiel sollte Spieler1 die meisten Spiele gewinnen.

Eingabe:

Controler.py test1.in

Ausgabe:

erster Test

kontrolle ob richtige Eingabe zum Erfolg führt.

1 2 3 4 5 6

Gewonnene Spiele Spieler1: 100.0%

Verlorene Spiele Spieler1: 0.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (1, 2, 3, 4, 5, 6) -> (0, 2, 3, 4, 5, 6)

Zug2 , Spieler2 : (0, 2, 3, 4, 5, 6) -> (0, 2, 2, 4, 5, 6)

Zug3 , Spieler1 : (0, 2, 2, 4, 5, 6) -> (0, 0, 2, 4, 5, 6)

Zug4 , Spieler2 : (0, 0, 2, 4, 5, 6) -> (0, 0, 2, 4, 5, 0)

Zug5 , Spieler1 : (0, 0, 2, 4, 5, 0) -> (0, 0, 0, 4, 5, 0)

Zug6 , Spieler2 : (0, 0, 0, 4, 5, 0) -> (0, 0, 0, 4, 4, 0)

Zug7 , Spieler1 : (0, 0, 0, 4, 4, 0) -> (0, 0, 0, 3, 4, 0)

Zug8 , Spieler2 : (0, 0, 0, 3, 4, 0) -> (0, 0, 0, 3, 2, 0)

Zug9 , Spieler1 : (0, 0, 0, 3, 2, 0) -> (0, 0, 0, 2, 2, 0)

Zug10 , Spieler2 : (0, 0, 0, 2, 2, 0) -> (0, 0, 0, 1, 2, 0)

Zug11 , Spieler1 : (0, 0, 0, 1, 2, 0) -> (0, 0, 0, 1, 1, 0)

Zug12 , Spieler2 : (0, 0, 0, 1, 1, 0) -> (0, 0, 0, 0, 1, 0)

Zug13 , Spieler1 : (0, 0, 0, 0, 1, 0) -> (0, 0, 0, 0, 0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:

nicht vorhanden

Beispiel 15

Bei diesem Beispiel sollte Spieler1 die meisten Spiele gewinnen.

Eingabe:

Controler.py test6.in

Ausgabe:

sechster Test

richtige Eingabe, gewinnt Spieler1?

9 9 9 9 9 9 9 9

Gewonnene Spiele Spieler1: 100.0%

Verlorene Spiele Spieler1: 0.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (9, 9, 9, 9, 9, 9, 9, 9, 9) -> (9, 9, 9, 9, 9, 9, 9, 9, 0)
 Zug2 , Spieler2 : (9, 9, 9, 9, 9, 9, 9, 9, 0) -> (9, 9, 9, 9, 9, 8, 9, 9, 0)
 Zug3 , Spieler1 : (9, 9, 9, 9, 9, 8, 9, 9, 0) -> (9, 9, 9, 9, 9, 0, 9, 9, 0)
 Zug4 , Spieler2 : (9, 9, 9, 9, 9, 0, 9, 9, 0) -> (9, 9, 9, 9, 6, 0, 9, 9, 0)
 Zug5 , Spieler1 : (9, 9, 9, 9, 6, 0, 9, 9, 0) -> (9, 9, 9, 9, 0, 0, 9, 9, 0)
 Zug6 , Spieler2 : (9, 9, 9, 9, 0, 0, 9, 9, 0) -> (9, 9, 9, 9, 0, 0, 9, 1, 0)
 Zug7 , Spieler1 : (9, 9, 9, 9, 0, 0, 9, 1, 0) -> (9, 9, 9, 9, 0, 0, 9, 0, 0)
 Zug8 , Spieler2 : (9, 9, 9, 9, 0, 0, 9, 0, 0) -> (9, 9, 9, 9, 0, 0, 6, 0, 0)
 Zug9 , Spieler1 : (9, 9, 9, 9, 0, 0, 6, 0, 0) -> (9, 9, 9, 9, 0, 0, 0, 0, 0)
 Zug10 , Spieler2 : (9, 9, 9, 9, 0, 0, 0, 0, 0) -> (9, 9, 9, 6, 0, 0, 0, 0, 0)
 Zug11 , Spieler1 : (9, 9, 9, 6, 0, 0, 0, 0, 0) -> (9, 9, 9, 0, 0, 0, 0, 0, 0)
 Zug12 , Spieler2 : (9, 9, 9, 0, 0, 0, 0, 0, 0) -> (2, 9, 9, 0, 0, 0, 0, 0, 0)
 Zug13 , Spieler1 : (2, 9, 9, 0, 0, 0, 0, 0, 0) -> (0, 9, 9, 0, 0, 0, 0, 0, 0)
 Zug14 , Spieler2 : (0, 9, 9, 0, 0, 0, 0, 0, 0) -> (0, 9, 5, 0, 0, 0, 0, 0, 0)
 Zug15 , Spieler1 : (0, 9, 5, 0, 0, 0, 0, 0, 0) -> (0, 5, 5, 0, 0, 0, 0, 0, 0)
 Zug16 , Spieler2 : (0, 5, 5, 0, 0, 0, 0, 0, 0) -> (0, 5, 3, 0, 0, 0, 0, 0, 0)
 Zug17 , Spieler1 : (0, 5, 3, 0, 0, 0, 0, 0, 0) -> (0, 3, 3, 0, 0, 0, 0, 0, 0)
 Zug18 , Spieler2 : (0, 3, 3, 0, 0, 0, 0, 0, 0) -> (0, 2, 3, 0, 0, 0, 0, 0, 0)
 Zug19 , Spieler1 : (0, 2, 3, 0, 0, 0, 0, 0, 0) -> (0, 2, 2, 0, 0, 0, 0, 0, 0)
 Zug20 , Spieler2 : (0, 2, 2, 0, 0, 0, 0, 0, 0) -> (0, 2, 0, 0, 0, 0, 0, 0, 0)
 Zug21 , Spieler1 : (0, 2, 0, 0, 0, 0, 0, 0, 0) -> (0, 0, 0, 0, 0, 0, 0, 0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:

nicht vorhanden

Beispiel 16

Bei diesem Beispiel kann Spieler1 kein Spiel gewinnen.

Eingabe:

Controler.py test7.in

Ausgabe:

erster Test

Spieler1 kann nicht gewinnen

1 1 1 1 1 1 1 1

Gewonnene Spiele Spieler1: 0.0%

Verlorene Spiele Spieler1: 100.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

nicht vorhanden

Beispiel eines von Spieler1 verlorenen Spiels:

Zug1 , Spieler1 : (1, 1, 1, 1, 1, 1, 1, 1) -> (0, 1, 1, 1, 1, 1, 1, 1)

Zug2 , Spieler2 : (0, 1, 1, 1, 1, 1, 1, 1) -> (0, 1, 1, 1, 1, 1, 1, 0)
 Zug3 , Spieler1 : (0, 1, 1, 1, 1, 1, 1, 0) -> (0, 0, 1, 1, 1, 1, 1, 0)
 Zug4 , Spieler2 : (0, 0, 1, 1, 1, 1, 1, 0) -> (0, 0, 1, 1, 0, 1, 1, 0)
 Zug5 , Spieler1 : (0, 0, 1, 1, 0, 1, 1, 0) -> (0, 0, 0, 1, 0, 1, 1, 0)
 Zug6 , Spieler2 : (0, 0, 0, 1, 0, 1, 1, 0) -> (0, 0, 0, 0, 0, 1, 1, 0)
 Zug7 , Spieler1 : (0, 0, 0, 0, 0, 1, 1, 0) -> (0, 0, 0, 0, 0, 0, 1, 0)
 Zug8 , Spieler2 : (0, 0, 0, 0, 0, 0, 1, 0) -> (0, 0, 0, 0, 0, 0, 0, 0)

Beispiel 17

Bei diesem Beispiel muss Spieler1 alle Spiele gewinnen.

Eingabe:

Controler.py test8.in

Ausgabe:

achter Test

Spieler1 muss gewinnen

1 1 1 1 1 1 1 1 1

Gewonnene Spiele Spieler1: 100.0%

Verlorene Spiele Spieler1: 0.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (1, 1, 1, 1, 1, 1, 1, 1) -> (0, 1, 1, 1, 1, 1, 1, 1)
 Zug2 , Spieler2 : (0, 1, 1, 1, 1, 1, 1, 1) -> (0, 1, 1, 1, 1, 1, 0, 1)
 Zug3 , Spieler1 : (0, 1, 1, 1, 1, 1, 0, 1) -> (0, 0, 1, 1, 1, 1, 0, 1)
 Zug4 , Spieler2 : (0, 0, 1, 1, 1, 1, 0, 1) -> (0, 0, 0, 1, 1, 1, 0, 1)
 Zug5 , Spieler1 : (0, 0, 0, 1, 1, 1, 0, 1) -> (0, 0, 0, 0, 1, 1, 0, 1)
 Zug6 , Spieler2 : (0, 0, 0, 0, 1, 1, 0, 1) -> (0, 0, 0, 0, 1, 0, 0, 1)
 Zug7 , Spieler1 : (0, 0, 0, 0, 1, 0, 0, 1) -> (0, 0, 0, 0, 0, 0, 0, 1)
 Zug8 , Spieler2 : (0, 0, 0, 0, 0, 0, 0, 1) -> (0, 0, 0, 0, 0, 0, 0, 1)
 Zug9 , Spieler1 : (0, 0, 0, 0, 0, 0, 0, 1) -> (0, 0, 0, 0, 0, 0, 0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:

nicht vorhanden

Beispiel 18

Bei diesem Beispiel sollte Spieler1 die meisten Spiele gewinnen.

Eingabe:

Controler.py test9.in

Ausgabe:

neunter Test

3 4 5 3 4 6 8 2 7

Gewonnene Spiele Spieler1: 100.0%

Verlorene Spiele Spieler1: 0.0%

Beispiel eines von Spieler1 gewonnenen Spiels:

Zug1 , Spieler1 : (3, 4, 5, 3, 4, 6, 8, 2, 7) -> (3, 4, 5, 3, 4, 6, 8, 0, 7)

Zug2 , Spieler2 : (3, 4, 5, 3, 4, 6, 8, 0, 7) -> (3, 4, 5, 3, 4, 6, 5, 0, 7)

Zug3 , Spieler1 : (3, 4, 5, 3, 4, 6, 5, 0, 7) -> (0, 4, 5, 3, 4, 6, 5, 0, 7)

Zug4 , Spieler2 : (0, 4, 5, 3, 4, 6, 5, 0, 7) -> (0, 4, 5, 3, 4, 4, 5, 0, 7)

Zug5 , Spieler1 : (0, 4, 5, 3, 4, 4, 5, 0, 7) -> (0, 4, 5, 0, 4, 4, 5, 0, 7)

Zug6 , Spieler2 : (0, 4, 5, 0, 4, 4, 5, 0, 7) -> (0, 4, 5, 0, 4, 4, 0, 0, 7)

Zug7 , Spieler1 : (0, 4, 5, 0, 4, 4, 0, 0, 7) -> (0, 0, 5, 0, 4, 4, 0, 0, 7)

Zug8 , Spieler2 : (0, 0, 5, 0, 4, 4, 0, 0, 7) -> (0, 0, 5, 0, 0, 4, 0, 0, 7)

Zug9 , Spieler1 : (0, 0, 5, 0, 0, 4, 0, 0, 7) -> (0, 0, 5, 0, 0, 0, 0, 0, 7)

Zug10 , Spieler2 : (0, 0, 5, 0, 0, 0, 0, 0, 7) -> (0, 0, 4, 0, 0, 0, 0, 0, 7)

Zug11 , Spieler1 : (0, 0, 4, 0, 0, 0, 0, 0, 7) -> (0, 0, 4, 0, 0, 0, 0, 0, 4)

Zug12 , Spieler2 : (0, 0, 4, 0, 0, 0, 0, 0, 4) -> (0, 0, 3, 0, 0, 0, 0, 0, 4)

Zug13 , Spieler1 : (0, 0, 3, 0, 0, 0, 0, 0, 4) -> (0, 0, 3, 0, 0, 0, 0, 0, 3)

Zug14 , Spieler2 : (0, 0, 3, 0, 0, 0, 0, 0, 3) -> (0, 0, 1, 0, 0, 0, 0, 0, 3)

Zug15 , Spieler1 : (0, 0, 1, 0, 0, 0, 0, 0, 3) -> (0, 0, 1, 0, 0, 0, 0, 0, 1)

Zug16 , Spieler2 : (0, 0, 1, 0, 0, 0, 0, 0, 1) -> (0, 0, 1, 0, 0, 0, 0, 0, 0)

Zug17 , Spieler1 : (0, 0, 1, 0, 0, 0, 0, 0, 0) -> (0, 0, 0, 0, 0, 0, 0, 0, 0)

Beispiel eines von Spieler1 verlorenen Spiels:

nicht vorhanden

6 Testprotokoll

<i>Beispiel</i>	<i>Datum</i>	<i>Erwartetes Ergebnis</i>
Beispiel 1	2012-12-06	ja
Beispiel 2	2012-12-06	ja
Beispiel 3	2012-12-06	ja
Beispiel 4	2012-12-06	ja
Beispiel 5	2012-12-06	ja
Beispiel 6	2012-12-06	ja
Beispiel 7	2012-12-06	ja
Beispiel 8	2012-12-06	ja
Beispiel 9	2012-12-06	ja
Beispiel 10	2012-12-06	ja
Beispiel 11	2012-12-06	ja
Beispiel 12	2012-12-06	ja
Beispiel 13	2012-12-06	ja
Beispiel 14	2012-12-06	ja
Beispiel 15	2012-12-06	ja
Beispiel 16	2012-12-06	ja
Beispiel 17	2012-12-06	ja
Beispiel 18	2012-12-06	ja

7 Schlusswort

Der Algorithmus des Spieler1 ist sehr erfolgreich gegen Spieler2. Recherchiert man aber ein wenig, kommt man schnell auf andere Lösungen, deren Erfolge gegen die gewählte Strategie sicher größer sein dürften, als die auf Zufall basierende Strategie von Spieler2.

So wäre es eine interessante Untersuchung, als wie gut sich die gewählte Strategie von Spieler1 gegen die Strategie von Bouton erweisen würde, die unter dem Thema "Nim-Spiel" auf Wikipedia vorgestellt wird.

Der gewählte Algorithmus hat einen wenig mathematischen Hintergrund. Er beruht nur auf Annahmen, die der Prüfling während des ersten schriftlichen Teils der Prüfung aufgestellt hat. Ob es immer einen Vorteil hat, die Reihe mit der kleinsten Anzahl an Hölzern zu entfernen, ist so nicht bewiesen. Gegen den Zufall halten die Annahmen jedoch stand.

7.2 Ausblick

Die Struktur des Programms erlaubt eine einfache Änderung der Strategien für die Spieler. So können leicht unterschiedliche Strategien gegeneinander getestet werden.

Dank dem MVC kann leicht eine Bedienoberfläche entwickelt werden, die es dem Anwender ermöglicht, gegen eine der künstlichen Intelligenzen anzutreten. Da das Spiel den Spieler fragt, welchen Zug er erwägt, ist an dieser Stelle eine Benutzerabfrage leicht zu implementieren.

Die Ausgabe ist derzeit nur in eine Datei vorgesehen. Dank MVC wäre hier auch leicht eine GUI möglich, die den jeweils aktuellen Spielfortschritt zeigt.

8 Anhang

8.1 Quellcode

Der Quellcode wird als extra Dokument eingereicht, da dieser über den Rahmen der Dokumentation hinaus gehen würde.

8.2 Juristische Erklärung

Ich erkläre verbindlich, dass das vorliegende Prüfprodukt von mir selbstständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. Ich versichere, dass der vorgelegte Ausdruck mit dem Inhalt des von mir erstellten Datenträgers identisch ist. Weder ganz noch in Teilen wurde die Arbeit bereits als Prüfungsleistung vorgelegt. Mir ist bewusst, dass jedes Zuwiderhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung des Prüfprodukts als Prüfungsleistung ausschließt.

Ort / Datum Unterschrift Andreas Maertens