Frankfurt University of Applied Sciences

Modul: Betriebssysteme und Rechnernetze
Betreuer: Prof Dr. Baun

Werkstück A – Alternativ 7
Schiffe-Versenken-Spiel

SS 2022

Abgabetermin: 27.06.2022

Projekt Mitglieder:

Mujtaba Atmar 1264447

Beniyamin Skachkov 1358324

Kevin Bauer 1349294

Inhalt

1. Abstract

Battleship oder auch Schiffe-Versenken-Spiel bezeichnet ein Strategie-Ratespiel für 1-2 Personen, welches besonders bei Jugendlichen besonders beliebt ist.
Im Rahmen unseres Projektes haben wir uns dazu entschieden die Alternative 7 vom Werkstück A zu entwickeln und zu implementieren. Das Programm bzw. das Spiel basiert auf Grundlage von Python (Python ist eine interpretierte Hochsprachen-Programmiersprache für allgemeine Zwecke) (Hatole, 2019).
Das Programm zu schreiben bzw. zu entwickeln stellte für uns eine wahre Herausforderung dar, offenbarte sich gleichzeitig jedoch als sehr interessant und fordernd, da die Methoden zur Lösungsfindung im Kontext der Aufgabe in diesem speziellen Fall besonders vielfältig waren.

2. Einleitung

Zu Beginn der Projektplanung stellte es sich als besonders anspruchsvoll heraus einen Leitfaden zu entwickeln, welcher durch das gesamte Projekt hindurch einen Orientierungspunkt liefert, damit dieses erfolgreich zum Abschluss gebracht werden kann. In Bezug auf die Entwicklung und interne Kommunikation des Programms zu entwickeln haben wir diverse Technogien herausarbeiten können. xxx
Diese Projektarbeit hat uns auch gezeigt, welchen Umfang ein solches Projekt annehmen kann und uns bereits einen Eindruck in Bezug auf unser zukünftiges Arbeitsleben geboten.

3. Das Problem

Das Problem oder besser gesagt „das Projekt“ war Entwickelung und Implementierung das sogenannte „Schiffe-Versenken-Spiel“ mit Interprozesskommunikation, mit dem zwei Spieler in der Shell gegeneinander spielen können.

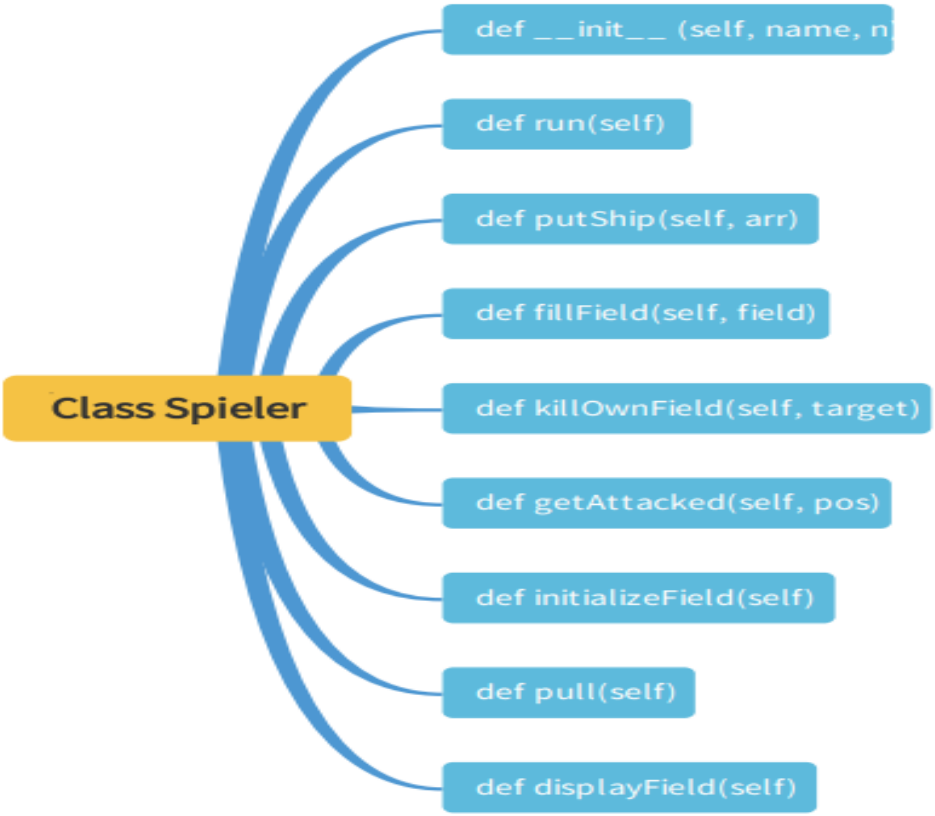
Beide Spieler sollten in der Lage sein ihr Schiffen nach eigenen Wünsch auf dem Spielfeld zu platzieren und der Spieler, der in der Reihe ist, soll entweder per Tastatur oder Maus definieren, wo soll es geschossen werden. Sobald ein Spieler alle schiffen seinen gegen Spieler versenken hat, hat der Spieler gewonnen.

Da unser Team noch nicht eine Berührung mit Python hatte, konnten wir uns kaum vorstellen wie Man so ein Programm entwickeln und Implementieren kann. Aus dem Grund haben wir uns ein Wenig Zeit genommen, damit wir uns mit der Programmieren Sprache Python beschäftigen können und uns die beibringen. Dank unserer Hochschule und vor allem unser Professor sowie Tutoren war dies ziemlich anspruchslos.

4. Unser Lösungsweg

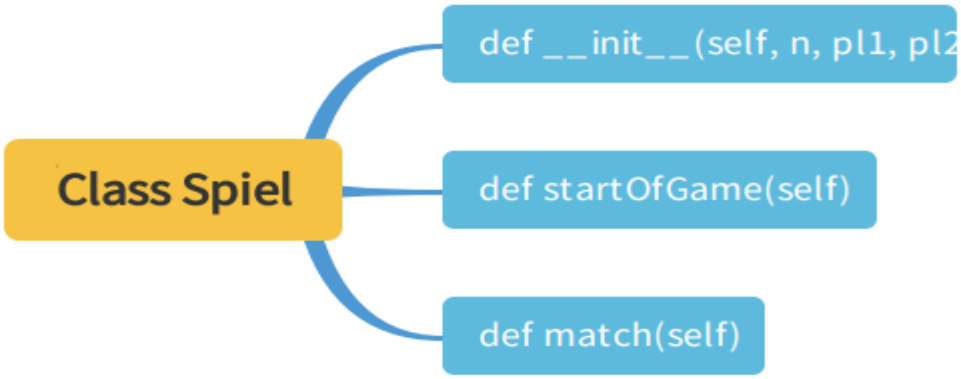
Wir haben uns verschiedenen Möglichkeiten überlegt und sind zum Entschluss gekommen, dass unser Programm mit einem implementierten Klassenprinzip genutzt werden soll. Unser Programm bzw. Code besteht aus zwei Klassen und zwei Funktionen bzw. Methoden außerhalb der Klassen. Unsere Klassen sind wie folgend aufgebaut:

- Class Spieler



Quelle: selbst erstellt

- Class Spiel



Quelle: selbst erstellt

5. Code

Wir haben unseren Code zunächst ohne Prozessparallelisierung oder Ähnliches implementiert. Erst als wir unser erstes funktionierendes Spiel programmiert haben, versuchten wir (mit Erfolg) die einzelnen Elemente auf ein mit Prozessparallelisierung arbeitendes System zu übertragen.

Wichtig:
Um das Programm einwandfrei zu starten ist die Installation folgender Python Packages notwendig:

- import time
- from cfonts import render, say
- from colorama import Fore, Back, Style
- import threading

a) Class Spieler

Am Anfang fiel uns sehr schwer eine Klasse als Spieler vorzustellen aber mit der Zeit hat sich das als eine nicht schwere Herausforderung herausgestellt. Die Klasse Spieler erbt von threading.Thread. Dies ermöglicht es, einen Spieler durch einen eigenen parallelen Prozess zu realisieren. Wie genau dies implementiert wurde, wird später erklärt.

In der “Spieler-Klasse” befinden sich unsere __init__() Methode sowie vier weitere Methoden.
In der __init__() Methode haben wir ein zweidimensionales Dictionary [key:value Pairs] initialisiert, welches das persönliche Spielfeld abbilden soll. Alle Werte werden zunächst auf 0 gesetzt und somit ist das Spielfeld leer. Dieses wird durch eine weitere Methode (Spieler.InitializeField()) befüllt. Darauf wird allerdings im Verlaufe der Dokumentation noch eingegangen. Ein zweites Dictionary (self.properties) dient zur Einstellung der Anzahl und Länge der Schiffe. In diesem Fall stellen die Keys die Länge der Schiffe und die Values die Anzahl der Schiffe dar, welche für die spezifische Länge verfügbar sind.

```
def __init__(self, name, n):
    threading.Thread.__init__(self)
    self.name = name
    self.n = n
    self.field = dict({})
    self.lifes = 0
    self.attacked = []
    self.otherPl = None
    self.game = None

    # These are bool-variables for signalizing and synchronising between the processes
    self.initialize = False
    self.turn = False

    # Asserts, that the field is not bigger than 10
    assert n <= 10
    assert n >= 5

    # Initalizing empty field
    for k in "ABCDEFGHIJ"[:n]:
        self.field[k] = dict({})
        for l in range(1, n+1):
            self.field[k][str(l)] = 0

    # This is a dictionary for the ships, where
    # the key is the length of the ship andthe value is the amount of ships available for this length.
    self.properties = {
        "5": 0,
        "4": 0,
        "3": 0,
        "2": 1
    }
```

Eine weitere Methode, die essenziell für unsere Klasse ist, ist die Methode Spieler.run(). Sie wird beim Starten des Thread-Objektes aufgerufen und kann gleichgesetzt werden mit einer allgemeinen Main()-Methode eines normalen Prozesses. Sie implementiert das, was der Subprozess parallel zum Hauptprozess machen soll. In unserem Programm haben wir mit While-Schleifen und Boolean-Variablen gearbeitet. Beide Spielerprozesse besitzen daher die Variablen self.initialize und self.turn. Beide sind zunächst auf False gesetzt. Erstere wird genau dann auf True gesetzt, wenn der Spieler

sein Feld initialisieren soll. In Folge dessen gelangt das Spieler-Objekt aus der ersten While-Schleife und führt die Funktion Spieler.initializeField() aus. Nach selbigem Prinzip verläuft das restliche Spiel durch die zweite, verschachtelte While-Schleife, wobei jedesmal, wenn der Spieler an der Reihe ist, die Variable self.turn auf True gesetzt wird, um daraufhin die Methode Spieler.pull() auszuführen.

Am Ende eines jeden Zuges wird die Variable self.turn des jeweiligen Spielers wieder auf False gesetzt, damit der Prozess erneut in die „Dauerschleife“ gerät.

```
def run(self):
    """Run Method of Thread Object

    This is the Main Process for one Player. He first waits, to initialize
    his own fields. After that he goes through a while loop, in which he
    makes his pull and waits until it's his turn again.

    :return:
    """
    while not self.initialize:
        time.sleep(1)

    self.initializeField()
    self.game.player_makes_pull = False


    while not self.game.finished:
        while not self.turn and not self.game.finished:
            time.sleep(1)
        if not self.game.finished:
            self.pull()
            self.game.player_makes_pull = False
            self.turn = False
```

Die Methode Spieler.putShip(arr) nimmt ein Array von Feldern entgegen (arr = [„A1“, „A2“, „A3“]). Die Methode führt einige Checks durch, um herauszufinden, ob die Platzierung so genehmigt werden kann. Diverse Gründe könnten für eine Nicht-Platzierung sorgen, darunter:

1. Mindestens eines der Felder ist bereits von einem anderen Schiff besetzt
2. Auswahl liegt außerhalb des Feldes
3. Ungültiges Eingabe-Muster („Aa1“, „1A“ etc. führen zu Fehlern)
4. Die Auswahl bildet keine Reihe (arr = [„A1“, „A2“, „A4“])

```
def putShip(self, arr):
    """This method puts one ship on the field First, it checks, whether everything is fine with the fields. After that,
       it puts the ship on the field.
    :param arr: Array of Strings of Fields (arr = ["A1", "A2", "D5"])
    :return: Nothing
    """
    size = len(arr)
    # Check, whether the size is in range.
    assert 1 < size <= 5

    # Check whether ship is available (especially for the given size).
    if not self.properties.get(str(size)) > 0:
        raise IndexError("Kein " + str(size) + "er Schiff verfuegbar!")

    # arr = ["A1", "A2", "D5"]
    for i in range(len(arr)):  # i = 1

        # Check, whether there is a ship already placed.
        if self.field[arr[i][0]][arr[i][1]] == 1:
            raise ValueError("Hier liegt bereits ein Schiff!")

        if i == 0:
            continue

        # Check, whether the current field and the field before (in "arr") are neighbours.
        elif not checkIfNeighbours(arr[i - 1], arr[i]):
            raise ValueError("Die Auswahl muss nebeneinander liegen!")
    # Decreases the availability of ships of the given size
    self.properties[str(size)] -= 1
    # If everything is fine, the ship gets placed.
    for i in arr:
        self.fillField(i)
```

Die Methode Spieler.getAttacked(pos) nimmt ein Feld entgegen und realisiert das „Angegriffen-Werden“. Geht man von folgender Situation aus: Spieler A greift Spieler B auf Feld „A1“ an. Dann muss B.getAttacked(„A1“) aufgerufen werden, unabhängig davon, ob sich auf dem Feld ein Schiff befindet oder nicht. Die Methode selber gibt zurück, ob getroffen (1) oder nicht getroffen (0) wurde. Zudem wird im Fall eines Treffers das Leben des Spielers um eins dekrementiert.

```
def getAttacked(self, pos):
    """This method gets called, whenever the player gets attacked.

    The method first checks, whether the attacked field is empty or not
    and after that it does change the values.

    :param pos: target position (pos = "A1")
    :return: 0: no hit; 1: hit; 2: already hitten;
    """

    pos = pos[0].capitalize() + pos[1]

    # Check, whether field is empty or not
    if self.field[pos[0]][pos[1]] == 1:

        # If field not empty, call method below and decrease the life
        self.killOwnField(pos)
        self.lifes -= 1
        return 1

    return self.field[pos[0]][pos[1]]
```

Die Methode Spieler.Pull() kommuniziert mit dem menschlichen Spieler über die Konsole und begleitet ihn jedesmal durch einen Zug.

```
    def pull(self):
        """This method assists the player through his pull/turn.

        :return:
        """
        pl = self
        print(100 * "\n")
        print(pl.name + " ist jetzt dran")
        input("Klicke Enter um das momentane Feld anzuzeigen [enter]")

        print(100 * "\n")
        self.displayField()
        print("\n\nDein Leben liegt bei: "  + str(self.lifes))
        time.sleep(1)

        while True:

            target = input(
                "\n\nWelches gegnerische Feld möchten Sie attackieren? ")

            try:
                if not target[0].capitalize() in "ABCDEFGHIJ"[:self.n] or not int(target[1]) in range(1, self.n + 1):
                    print("Eingabe entspricht nicht dem Format!")
                    continue
            except:
                print("Eingabe entspricht nicht dem Format!")
                continue

            target = target[0].capitalize() + target[1]

            if target in pl.attacked:
                print("Hier hattest du bereits hin geschossen...")
                print("Versuche es erneut!\n\n")
                time.sleep(3)
                continue

            # Call Method for Opponent so he gets attacked (no matter if hit or not)
            ret = self.otherPl.getAttacked(target)

            # Add target to List of Attacked Fields (no matter if hit or not)
            pl.attacked.append(target)

            print(100*"\n")

            if ret == 0:

                print(render("Kein Treffer",
                                        font='chrome', colors=['red', 'black'],
                                        align='center'))

                print("\n" + self.name + "s Leben: " + str(self.lifes))
                print(self.otherPl.name + "s Leben: " + str(self.otherPl.lifes))
            elif ret == 1:
                print(render("Getroffen",
                             font='chrome', colors=['green', 'yellow'],
                             align='center'))
                print("\n" + self.name + "s Leben: " + str(self.lifes))
                print(
                    self.otherPl.name + "s Leben: " + str(self.otherPl.lifes))

            time.sleep(3)
            break
```

b) Class Spiel

Die Klasse Spiel implementiert das Spiel selbst. Hierbei wird ebenfalls mit While-Schleifen und Boolean-Variablen gearbeitet. Man beachte die Variablen self.player_makes_pull und self.finished. Die erste Variable wird immer genau dann auf True gesetzt, wenn ein Spieler grade angestoßen wurde, um zu verhindern, dass der Main-Prozess parallel einfach weiter läuft. Hierbei geht das Spiel

in eine While-Schleife über, aus der es erst heraus kommt, wenn die jeweilige Boolean-Variable auf False gesetzt wird. Für nähere Erläuterung siehe Spiel.startOfGame() und Spiel.match().

Die __init__() Methode der Klasse Spiel nimmt folgende Parameter entgegen:

1. n: Länge/Breite des Feldes
2. pl1 / pl2: Beide Spieler

```
def __init__(self, n, pl1, pl2):
    """Init method of Spiel.

    :param n: length/width of field.
    :param pl1: Spieler-Object as a representation of Player 1
    :param pl2: Spieler-Object as a representation of Player 2
    """
    self.n = n
    self.pl1 = pl1
    self.pl2 = pl2
    self.pl1.otherPl = pl2
    self.pl1.game = self
    self.pl2.otherPl = pl1
    self.pl2.game = self

    self.pl1.start()
    self.pl2.start()

    # These are bool-variables for signalizing and synchronising between
    # the processes
    self.player_makes_pull = False
    self.finished = False
```

Die Methode Spiel.startOfGame() dient zur anfänglichen Initialisierung der Felder. Über das oben erwähnte Prinzip wird die Boolean Variable Spieler.initialize auf True gesetzt, um den jeweiligen Spieler „anzustoßen“.

```
def startOfGame(self):
    self.pl1.initialize = True
    self.player_makes_pull = True
    while self.player_makes_pull:
        time.sleep(1)

    self.pl2.initialize = True
    self.player_makes_pull = True
    while self.player_makes_pull:
        time.sleep(1)
```

Die Methode Spiel.match() implementiert das Hauptspiel. Hier befindet sich eine While-Schleife, bei der jeweils beide Spieler abwechselnd durch das oben beschriebene Prinzip „angestoßen“ werden. Damit ist gemeint, dass die Variable Spieler.turn auf True gesetzt wird, genau dann wenn er an der Reihe ist. Dieser Prozess wiederholt sich dann so lange, bis einer der Spieler kein Leben mehr besitzt. Anschließend wird der Gewinner herausgegeben.

```
def match(self):
    """This method represents the actual game.

    Through a while loop, the players will play the game until one of them
    has no life left.
    """
    player = True
    while self.pl1.lifes > 0 and self.pl2.lifes > 0:  # Until no life left
        if player:
            self.pl1.turn = True
            self.player_makes_pull = True
            while self.player_makes_pull:
                time.sleep(1)
        else:
            self.pl2.turn = True
            self.player_makes_pull = True
            while self.player_makes_pull:
                time.sleep(1)

        player = not player

    print(100*"\n")
    winner = self.pl1 if self.pl1.lifes > 0 else self.pl2
    print(winner.name + " hat das Spiel gewonnen.")
    time.sleep(3)
    self.finished = True
    self.pl1.join()
    self.pl2.join()

    winnerOfTheGame = render(winner.name +' ist der Gewinner ', font='block', colors=['red', 'red'], align='center')
    print(winnerOfTheGame)
```

7 Fazit

Dank dieses Projekts haben wie gelernt, wie man Projekte in einer Gruppe realisiert und haben dadurch auch einen Blick im Berufsleben bekommen. Wir haben das gemeinsam entwickelt und implementiert haben aber wir haben trodzdem versucht die Arbeit aufzuteilen zum Beispiel für die Klasse Spieler ich (Mujtaba Atmar) und Benjamin Skachkov. Für die Klasse Spiel war Kevin Bauer verantwortlich.
Wir haben das Problem der Prozessparallelisierung gemeinsam durch Threading realisiert. Die einzelnen Thread Objekte warten hierbei jeweils aufeinander, um Chaos zu vermeiden.