Frankfurt University of Applied Sciences

Engineering Business Information Systems - Wirtschaftsinformatik (B.Sc.)

Wirtschaftsinformatik mit Fokus auf IT

Name der Portfolioprüfung:

BSRN-Werkstück A Alternative 2 – Buzzword Bingo

Prüfer:

Prof. Dr. Christian Baun

Vorgelegt von:

Gracjan Sikora
Matrikelnummer: 1358780

Andreas Kranz
Matrikelnummer: 1360121

Delos Joel Herold
Matrikelnummer: 1352621

Fachsemester: Sommersemester 2021
Studiengang: EBIS - Wirtschaftsinformatik (B.Sc.)
Abgabe: 28.06.2021

Kapitel:

Aufgabenstellung:

Das Ziel ist ein interaktives Bingo-Spiel am Computer zu realisieren. Der Inhalt der Bingo-Karte sind Buzzwords aus Industrie und Wirtschaft. Es soll sich dabei um ein Skript handeln, welches über die Kommandozeile gesteuert wird.

Gedanken zu Beginn:

Wahl der Programmiersprache:

Wir entschieden uns für Python als Programmiersprache, da wir die ähnliche Objektorientierte Programmiersprache Java kannten und Python den Ruf einer für Neulinge einfachen Sprache innehat. C erscheint sehr mächtig gleichweise auch sehr komplex und somit nicht attraktiv.

In unseren Augen erfährt Python auch steigende Popularität was für zunehmende Relevanz in der Zukunft spricht, namentlich Sparten der Data-Science und Maschine-Learning. Auch sind reichlich aktive Entwickler mit der Weiterentwicklung des Python-Ökosystems, Bibliotheken werden aktualisiert und erweitert, zudem gibt es täglich neue Tutorials und Guides über Python-Entwicklung.

In verschiedenen Umfragen und Statistiken zählt Python zu den beliebtesten Programmiersprachen.

Brainstorming zur Struktur:

Bei dem Wort Bingo dachten wir direkt an eine klassische Bingo Karte, welche aus Feldern besteht. Danach haben wir uns Gedanken über den Spielablauf und Realisierung der Optischen Oberfläche gemacht. Wir haben eine Liste aufgestellt mit allen Anforderungen und sind diese immer wieder durchgegangen, um eine möglichst genaue Struktur zu erarbeiten. Erste Funktionalitäten haben wir Methoden zugeordnet die später realisiert werden mussten. Dabei haben wir uns schon erste Python Bibliotheken angeschaut, welche uns bei der Realisierung helfen könnten.

Wir wussten schnell das wir verschiedene Klassen implementieren würden, um Überblick und Abtrennung der Funktionalitäten gewährleisten zu können. Eine Klasse für die Felder

und eine für die Karte, die die Felder in einer dynamischen Datenstrukturen hält, erste Idee war eine zwei-dimensionale Liste/Array um Zeilen und Spalten abzubilden. Eine weitere Idee war es, mithilfe der Bibliothek „Numpy“ eine Matrix zu erstellen welche sich dynamisch bestimmen lässt. In Folge unserer ersten Gedanken haben wir einiges ausprobiert, Tutorials und Guides angeschaut, um langsam in das Thema einzusteigen bzw. eine Vorstellung zu bekommen.

Struktur und Funktionen

In Folgendem gehen wir auf die Grundstruktur unseres Programmes ein. Im Endeffekt besteht der Kern aus zwei Klassen, einmal Feld und Karte mit verschiedenen Methoden.

Feld:

```
class Feld():
```

Feld stellt auf klassischen Bingo Karten ein Feld mit einem Wort dar. So hat das Feld die Attribute buzzword (String) und durchgestrichen (Boolean), analog zu dem klassischen Bingo. Initialisiert werden diese Attribute in der Konstruktor Methode *__init__* für jedes einzelne Objekt vom Typ Feld. Die *IF-Anweisung* prüft, ob es sich um ein „Joker Feld“ handelt

```
def __init__(self, buzzword):
    self.buzzword = buzzword
    self.durchgestrichen = False
    if (buzzword == "JOKER"):
        self.durchgestrichen = True
```

Karte:

```
class Karte():
```

Die Klasse Karte verkörpert eine reale Bingo Karte. Sie enthält Methoden, welche für den Spielablauf notwendig sind.

Ein Objekt der Klasse Karte hat die Attribute:

```
def __init__(self):
    self.gewonnen = False
    self.felder = []
    self.dimension = 0
    self.anzahlFelder = 0
```

gewonnen (Boolean), dieses Attribut speichert, ob das Spiel gewonnen ist oder noch läuft

felder (Liste), diese Liste hält die Objekte der Klasse Felder auf der Karte, die Reihenfolge ist wichtig für weitere Funktionalitäten wie das Streichen eins Wortes und die Überprüfung der Siegbedingung – hingegen der ursprünglichen Ideen einer Matrix oder eines zwei-dimensionalen Arrays hat sich gezeigt das eine ein-dimensionale Liste einfacher zu initialisieren und Objekte direkter aufrufbar sind. Denn die variable Spielfeldgröße macht das Befüllen der Strukturen und aufrufen der Felder deutlich komplexer als es die verwendete Lösung zulässt. Da die Karte quadratisch und somit alle Spalten und Zeilen gleich lang sind, kann die Position leicht abgezählt werden. (Von oben-nach-unten und von links-nach-rechts) Passend dazu stellt die Position gleichzeitig auch den korrespondierenden Index in der Liste dar.

dimension(integer), speichert die Größe (Seitenlänge) der Bingo Karte ab

anzahlFelder(integer), speichert die Anzahl der Felder, das Quadrat der dimension

```
def woerter_Einlesen(self):
```

Die Methode *woerter_Einlesen* liest aus einer Text Datei(woerter_liste) alle Wörter ein, und speichert sie in der Liste alleWoerter. Dies sind alle möglichen Buzzwords.

```
def kartenParameter_eingeben(self, eingabe):
```

kartenParameter_eingeben nimmt die Nutzereingabe (eingabe) der Spielfeldgröße entgegen und speichert diese in der Objektvariablen dimension bzw. das Quadrat in anzahlFelder.

```
def erstelle_ZufallsListe(self):
```

In der nächsten Methode, *erstelleZufallsListe* wird mithilfe der Bibliothek RANDOM eine Liste erstellt (tempList). Diese wird mithilfe von *random.sample()* zufällig mit Buzzwords aus der Liste alleWoerter gefüllt. Somit wird garantiert, dass bei jeder neuen Runde Bingo eine neue zufällig generierten Karte zur Verfügung steht.
In der darauffolgenden *for-Schleife* füllen wir die Liste felder mit Objekten vom Typ Feld und als Parameter übergeben wir ein Buzzword (str) aus der tempList. Dabei erstellen wir nur so viele Objekte wie benötigt werden, z.B. bei einer Spielfeldgröße von 5 sind es 5*5= 25 Feld-Objekte. Die Liste felder symbolisiert also unser Spielfeld.

```
for i in range(0,self.anzahlFelder):
    str = tempList[i]
    bingokarte.felder.append(Feld(str))
```

In den zwei *IF-Anweisungen* wird geprüft, ob unser Spielfeld die Größe 5 oder 7 besitzt. In diesen Fällen bauen wir genau in der Mitte einen Joker ein. Bei einer Größe von 5 ist die Mitte des Spielfeldes an der Positon/Index 12.

```
if bingokarte.dimension == 5:
    bingokarte.felder[12] = Feld("JOKER")
```

```
def wortStreichen(self,index):
```

wortStreichen dient, wie der Name schon sagt zum bildlichen „durchstreichen“ eines Buzzwords. Ein Wort gilt als durchgestrichen, wenn die Variable durchgestrichen eines Objektes des Typen Feld True ist. Der index gibt dabei an, wo wir uns in der Liste felder (Spielfeld) befinden

```
def pruefeGewinn(self, index):
```

Die Methode *pruefeGewinn* wird nach jeder Streichung eines Wortes aufgerufen, um die Siegbedingung des Bingo Spiels zu testen. Der Parameter Index wird übergeben, um die Effizienz des Algorithmus zu steigern, denn durch den Index – der gleichzeitig index der Liste und Position auf der Karte darstellt – prüft die Methode nur die Spalten und Zeilen in denen das Wort also Feld, das gestrichen wurde, liegt.

Es gibt *for-Schleifen* zum Testen einer vollständigen Reihe in Spalte, Zeile, der Diagonale von oben-links nach unten-recht und der Diagonale von unten-links nach oben-rechts

Mathematik hinter dem Algorithmus:

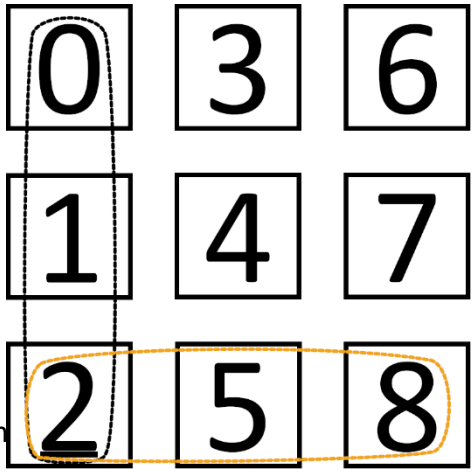
Da immer nur eine Spalte und Zeile das Wort enthalten, das gestrichen wurde, wollten wir vermeiden einen ineffizienten Algorithmus zu verwirklichen der das ganze Feld überprüft. Als Konsequenz der Entscheidung für eine ein-dimensionale Liste als dynamischen Speicher haben wir keine Spalten oder Zeile abgespeichert – somit benötigten wir eine allgemein gültige Formel, um aus dem Wissen über die Seitenlänge und die Position des gestrichenen Wortes die weiteren Felder in der korrespondierenden Zeile und Spalte zu bestimmen.

Nächster Schritt war das Aufzeichnen des Spielfelds (ähnlich dem Schaubild) auf Papier, um visuell Gesetzmäßigkeiten anhand der Zahlenfolgen der Felder zu finden.

Zuletzt haben wir diese erkannten Gesetzmäßigkeiten in Python-Code formuliert und stückweise auf Minimalität optimiert

Der komplexe Hintergrund wird anhand des Schaubilds beispielshaft erklärt. Ausgehend haben wir die Informationen über die Seitenlänge – hier 3 – und die Position des markierten Felds/Wortes – hier z.B. 2. Wir wollen nun mit diesen Informationen die Spalte und Zeile ermitteln in der das Wort/Feld liegt.

```
def pruefeGewinn(self, index):
    rest = index % self.dimension          #3
    quotient = int(index / self.dimension) #2
    counter = 0
```



Spalten: Die Spalte lässt sich mit dem Quotienten [#2] der sich aus der Division: [Position / Seitenlänge] ergibt, bestimmen. Hier im Beispiel (2 / 3 = 0.66667) – die erste Dezimalstelle (0), deswegen der Cast zum Integer, mit der Konstanten „1“ addiert ergibt die Spalte (0 + 1 = 1) also die 1.Spalte.
D.h. das Produkt aus Quotient [#2] und dimension ergibt das Minimum der berührten Spalte und das
Produkt aus Dimension und Quotient + 1 dementsprechend das exklusive Maximum.

```
for i in range(quotient * self.dimension, ((quotient + 1) * self.dimension)):
```

Zeilen: Die Zeile lässt sich mit dem Rest [#3] der sich aus der Ganzzahldivision von [Position / Seitenlänge] ergibt, bestimmen. Hier im Beispiel (2 % 3 = 2) – der Rest (2) addiert mit der Konstanten „1“ (2 + 1 = 3) ergibt die 3.Zeile.

D.h. wenn n=Nummer der Zeile (Rest #3) => Zeile ist die Reihe der Produkte von dimension*0+n; dimension*1+n;…; dimension*(dimension-1)+n

```
for z in range(self.dimension):
    stelle = z * self.dimension + rest
```

```
def export_BingoKarte(self):
```

Mithilfe dieser Methode ist es möglich eine Bingo Karte zu exportieren. Diese wird dann in der Text Datei „export.txt“ gespeichert. Die *for-Schleifen* bringen die Buzzwords in ein einheitliches Format und sorgen durch Spaltenumbrüche für das typische Bingo Karten Format.

```
bingokarte = Karte()
bingokarte.woerter_Einlesen()
input = bingokarte.felder
```

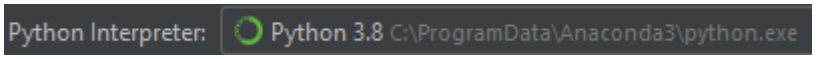
Nach den zwei Klassen (Feld, Karte) erstellen wir ein Objekt mit dem Namen bingokarte des Typen Karte. Mithilfe dieses Objektes rufen wir die Methode *woerter_Einlesen* auf. Somit wird unser Programm vorbereitet. Die Variable input brauchen wir später für die Ausgabe der Felder.

Command Line Interface (CLI)

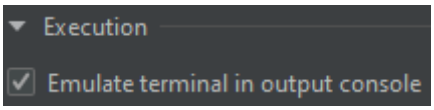
Vorwort:

Wir haben uns dazu entschieden, das CLI mit curses zu programmieren. Bei der Recherche fiel uns auf das curses häufig zur Verwendung kommt. Zudem sind reichlich Tutorials und viel Info-Material vorhanden. Zusätzlich ist ncurses in der Programmiersprache C sehr ähnlich wie curses in Python. Somit ist es sehr praktisch und wir konnten erste Erfahrung mit CLI´s sammeln.

Problematisch war die Implementierung in Windows. Wir haben uns dazu entschieden mit der Entwicklungsumgebung PyCharm zu arbeiten. Dabei gab es Probleme mit dem Erkennen der Bibliothek Windows-curses. Deswegen konnten wir nicht direkt Python als Interpreter nutzen. Um trotzdem PyCharm nutzen zu können haben wir Windows-curses über den Anaconda-Launcher installiert und diesen als Interpreter genutzt. Dazu ist zu sagen, dass man den Anaconda-Launcher und PyCharm Standard mäßig verknüpfen kann.



Des Weiteren mussten wir in der Entwicklungsumgebung das Terminal emulieren.



Abschließen ist zu sagen, dass wir dort einige Start Probleme hatten und es nicht ganz so einfach ist curses unter Windows zum Laufen zu bringen. Unter Linux ist es deutlich einfacher und unproblematischer.

Implementierung:

Unser CLI besteht im Kern aus 5 Methoden. Wir haben jeweils eine Methode, die das Menu/Spielfeld ausgibt und jeweils noch eine für die Steuerung. Die 5. Methode dient der Gewinnausgabe.

```
def print_menu(stdscr, selected_row_id):
```

In der Methode *print_menu* wird das Menu auf das Terminal ausgegeben. Mit *getmaxyx()* bekommen wir die maximale Größe des Terminals heraus. In der *for-Schleife* teilen wir einmal die maximalen x und y werte durch 2 sowie die Länge des Wortes (row) und des Menüs (menu)

```
x = w // 2 - len(row) // 2
y = h // 2 - len(menu) // 2
```

So bestimmen wir genau die Mitte unseres Bildschirmes und können dort mithilfe von *stdscr.addstr()* das Menü anzeigen.

```
if idx == selected_row_id:
```

Mithilfe der if-Anweisung prüfen wir die „aktuelle Position“ des Spielers und zeigen diese in einer andren Farbe an.

```
stdscr.attron(curses.color_pair(1))
```

Alle andren Felder werden in der „normalen“ Farbe ausgegeben.

```
def menu_main(stdscr):
```

In dieser Methode wird die Steuerung des Spielers geregelt sowie aller Ereignisse, wenn z.B. Play gedrückt wird. Um User-Input zu erfassen, bietet curses die Funktion *getch().*

```
key = stdscr.getch()
```

Dieser User-Input wird an eine Variable Übergeben und in der darauffolgenden *while-schleife* durch if-Anweisungen geprüft. Je nach gedrückter Taste gibt es ein anderes Ereignis. Des Weiteren haben wir mehrere Kontrollstrukturen eingebaut wie *try/except* um Fehler und falschen User Input zu vermeiden. Wenn z.B. ENTER gedrückt wird und sich der Spieler auf dem Feld PLAY befindet, werden mehrere Ereignisse ausgelöst. Zum einen wird der Spieler aufgefordert, die Spielfeldgröße festzulegen. Diese wird dann an die Klasse bingokarte übergeben. Folgend wird die Zufallsliste erstellt und das Programm beginnt alle wichtigen Funktionen vorzubereiten.

```
bingokarte.kartenParameter_eingeben(user_input)   #Übergeben der Spielfeldgröße
bingokarte.erstelle_ZufallsListe()                #Die Bingokarte "erstellen"
curses.wrapper(spielfeld_main)
```

curses.wrapper(spielfeld_main) ruft die „Spielfeldmethode“ auf und das Spiel beginnt. Die *wrapper* Funktion in curses erledigt das initialisieren und beenden des „Bildschirms“. Nach jedem Ereignis wird das Menu erneut ausgegeben.

```
def print_spielfeld(stdscr, selected_row_id, enter_list):
```

Diese Methode ist print_menu sehr ähnlich. Im Endeffekt geben wir das Spielfeld aus. Dabei gab es leider einige Probleme, welche zum Absturz unseres Programes geführt haben. Das Problem besteht darin, dass Terminals verschieden groß sein können. Daher kann es sein, dass unsere Bingokarte nicht „in das Terminal“ passt, was einen crash zur Folge hat. Wir konnten es nur bedingt lösen bzw. unser Programm so optimieren, dass die Wörter nicht „zu viel Platz“ brauchen.

```
max = ""
for a in input:
    if len(a.buzzword) > len(max):
        max = a.buzzword
```

Mithilfe dieser *for-Schleife* nehmen wir das längste Wort, welches ausgegeben wird und passen alle Spalten daran an. Somit haben wir eine dynamische Ausgabe, damit möglichst viele Wörter angezeigt werden können. Mit dieser *for-Schleife* realisieren wir das klassische

Aussehen des Bingo Felds. Bei einer Spielfeldgröße von 5*5 kommt es nach 5 Wörtern in der Vertikalen zu einem Spaltenumbruch.

```
for b in range(1,9):
    if idx == spielfeld * b:
        x += len(max)
        y = 0
```

Eine Besonderheit der Methode ist die enter_list. In ihr werden alle Wörter gespeichert, welche vom Spieler „durchgestrichen“ werden. Diese sind dann permanent grün gefärbt.

```
elif idx in enter_list:
    stdscr.addstr(y, x, row.buzzword, curses.color_pair(2))
```

Alle anderen Buzzwords werden in der normalen Farbe ausgegeben bzw. die aktuelle Position wird farblich hinterlegt.

```
def spielfeld_main(stdscr):
```

In dieser Methode regeln wir die Steuerung im Spielfeld sowie das „Durchstreichen“ der Wörter. Im Gegensatz zum Menü kann der Spieler sich hier auch nach links und rechts bewegen. Sobald ENTER gedrückt wird, erscheint eine Nachricht mit dem ausgewählten Wort. Dieses wird der enter_list hinzugefügt, dadurch ist es grün markiert.

```
bingokarte.wortStreichen(current_row_id)
bingokarte.pruefeGewinn(current_row_id)
```

Des Weiteren wird das Wort im Programm auf „True“ gesetzt (*bingokarte.wortStreichen(current_row_id)*) und der Gewinnalgorithmus prüft, ob der Spieler gewonnen hat. (*bingokarte.pruefeGewinn(current_row_id)*).

Am Ende der Methode wird das gesamte Spielfeld erneut ausgegeben inklusiver aller Veränderungen.

```
print_spielfeld(stdscr, current_row_id, enter_list)
```

```
def gewonnen():
```

Diese Methode wird aufgerufen, wenn der Spieler gewonnen hat. Mithilfe einer *for-Schleife* und der Bibliothek time konnten wir eine vertikale Laufschrift von oben nach unten realisieren. Diese Laufschrift erscheint mittig im Terminal. Dies haben wir wieder mit der erwähnten Methode *getmaxyx()* gemacht.

```
h, w = stdscr.getmaxyx() #maximale Größe der Konsole
x = w // 2               # mitte der x-Achse
```

Time.sleep() verzögert die Ausgabe um 0.05 Sekunden. Insgesamt wiederholt sich die Laufschrift dreimal. Danach schließt sich unser Programm

Schlusswort

Die Dokumentation hat das Ziel, einen groben Überblick über unser Programm zu geben. Wir erklären die Kernfunktionen sowie unsere Herausforderungen diese umzusetzen. Die Kombination aus Dokumentation und kommentierter Code bietet ein tieferes Verständnis für unser Programm. Wenn wir es in einem Satz beschreiben müssten, wäre es wie folgt: Im Endeffekt haben wir eine Liste mit zufälligen Wörtern, mit der über ein Terminal interagiert wird und nach gewissen Mustern auf gewinn geprüft wird.

Abschließend bleibt zu sagen, dass wir sehr viel Spaß hatten beim Programmieren. Es war zwar teilweise sehr schwierig und komplex aber zusammen im Team konnten wir jedes Problem lösen.