
# Buzzword-Bingo Dokumentation

Gruppenmitglieder:

Ioana-Carmen Moldovan (1514892)
Vanessa Nguyen (1513741)
Pantea Dolatabadian (1414653)
Aliia Nurbekova (1526039)
Thi Song Thu Pham (1413382)

Betriebssysteme und Rechnernetze
Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences

Abgabedatum: 30.06.2024

# 1 Einleitung

Die Gruppe, bestehend aus Ioana Moldovan, Vanessa Nguyen, Pantea Dolatabadian, Aliia Nurbekova und Thi Song Thu Pham hat sich für die Alternative 3 - Buzzword-Bingo-Spiel entschieden, da es visuell einfach zu veranschaulichen ist und bereits schon Kenntnisse mit dem Bingo-Spiel vorliegen. Es werden dafür verschiedene Bingokarten (vom Benutzer definierbar) und verschiedene Spieler erstellt, die gegeneinander spielen, bis eine Person gewinnt oder das Spiel beendet.

# 2 Vorgehensweise

Als Programmiersprache eignet sich Python aufgrund der Einfachheit am sinnvollsten, als Bibliothek für die grafische Darstelllung, pyTermTk. Das Projekt wird während der gesamten Laufzeit auf GitHub regelmäßig hochgeladen, um deutlich erkennbar zu zeigen, welche Teammitglieder sich um welche Aufgaben kümmern.

## 2.1 Aufgabenverteilung

Zunächst recherchiert jedes Mitglied die wichtigsten Aspekte des Spiels beziehungsweise wesentliche Python-Funktionen, da diese Programmiersprache zum Zeitpunkt des Projekts noch unbekannt ist.
Die Installation von Windows Subsystem für Linux und damit verbunden auch Ubuntu ist ein wichtiger Schritt, der zum Anfang des Projekts vorgenommen wurde, um später die Interprozesskommunikation gestalten zu können.
Die einzelnen Teile der Aufgabenstellung werden auf die Teammitgliedern aufgeteilt. Dies erscheint allerdings nach kurzer Zeit nicht mehr sinnvoll, da Aufgaben wie die Bingokarte und Textdatei zu erstellen und die Wörter aus der Liste in die Felder der Karte zu verteilen schnell erledigt wurden. Im Gegenteil ist die Interprozesskommunikation ein komplexes, bisher unbekanntes Thema, das die Beteiligung aller Mitglieder erfordert.

## 2.2 Herausforderungen

Beim Programmieren stellt sich heraus, dass es einige Herausforderungen gibt, für welche es verschiedene Lösungen gibt.
Als erstes ist das Erlernen einer neuen Programmiersprache wie Python schwierig, allerdings eignen sich dafür zahlreiche Videos und Anleitungen auf YouTube. Dies war der erste Schritt für das Projekt, indem alle Mitglieder sich genügend Wissen über Python, dessen Module und Funktionen, aneignen.
Nachdem dieser Schritt erfolgreich abgeschlossen wird, muss jedes Teammitglied die gesamte Aufgabenstellung gründlich lesen und jeden Teil verstehen. Dies wird auch in der Gruppe kommuniziert, sodass es keine Unklarheiten gibt. Allerdings stellen sich die ersten Fragen darüber, wie die Kommandozeilenanwendung aussehen sollte. Die Gruppe hatte als erste Idee einige Funktionen,

die über *input* den Benutzer auffordern, etwas in der Konsole einzugeben. Die zweite Option ist das Modul *argparse*, jedoch wird hierfür mehr Wissen benötigt. Nach gründlicher Recherche entscheidet sich das Team jedoch für argparse, da dies tatsächlich eine Kommandozeilenanwendung implementiert und zusätzlich eine sehr gute Übung mit der Shell darstellt, die sicherlich in der Zukunft gut zu gebrauchen ist.

```
1  # Kommandozeilenargumente mit dem argparse-Modul hinzufuegen
2  parser = argparse.ArgumentParser(description="Erstellen␣einer␣Bingo-
       Karte.")
3  # Definiert das Kommandozeilenargument 'filename' (Dateiname)
4  parser.add_argument("wordfile", nargs='?', help='Der␣Name␣der␣zu␣
       oeffnenden␣Datei')
5  # Definiert das Kommandozeilenargument 'felder_Anzahl' (Anzahl der
       Felder auf der Bingo-Karte)
6  parser.add_argument("felder_Anzahl", type=int, help="Die␣Anzahl␣der␣
       Felder␣der␣Bingo-Karte.")
7  # Server wird gestartet, um die Kommunikation zwischen den Prozessen
        zu gewaehrleisten
8  parser.add_argument("--server", action='store_true', help="Starte␣
       als␣Server")
9  # Hiermit wird jeder Spieler (Client) - Prozess gestartet
10 parser.add_argument("--name", type=str, help="Name␣des␣Spielers")
```

Eine weitere Herausforderung ist die Umsetzung der Ideen aller Mitglieder, insbesondere die Vorstellung, wie das Spiel letztendlich aussehen sollte. Dafür ist eine gute und kontinuierliche Kommunikation sehr wichtig. Zunächst präsentiert jeder in der Gruppe seine Vorstellungen und seine Ideen zur Umsetzung und die Gruppe entscheidet sich, die Buttons von *pyTermTk* zu verwenden. Dazu schaut sich jeder die Dokumentation von *pyTermTk* an, um die wichtigsten Kenntnisse darüber zu bekommen.

Da jeder sich mit Teilaufgaben beschäftigt, entsteht eine Herausforderung im Bezug auf das Kombinieren der Ergebnisse. Für die verschiedenen Klassen werden Funktionen mit Variablen erstellt, die später Konflikte mit den anderen Funktionen haben, da die Variablennamen erst angepasst werden müssen. Dazu findet sich schnell eine Lösung, indem man sich mit einem anderen Teammitglied austauscht und gemeinsam die Funktionen und Variablen so anpasst, dass sie in einem Programm funktionieren. Schritt für Schritt werden diese Teile des Projekts weiterentwickelt und ständig auf GitHub hochgeladen. Verbesserungen werden für die gesamte Laufzeit des Projekts von allen Teammitgliedern vorgenommen.

# 3 Probleme und Lösungen

## 3.1 Interprozesskommunikation

Die schwierige Anforderung des Projekts ist, den Datenaustausch zwischen verschiedenen Prozessen durchzuführen. Dies muss auf eine Art und Weise funk-

tionieren, die es ermöglicht, dass mehrere Spieler gleichzeitig gegeneinander spielen können und sobald einer gewinnt, die anderen auch die Nachricht bekommen, dass sie verloren haben und nicht weiterspielen können.

Um dieses Problem zu lösen, benutzt die Gruppe benannte Pipes als Tool für die Interprozesskommunikation wie folgt: Ein Server-Prozess erstellt eine benannte Pipe als First In First Out mit der Unix-Funktion mkfifo:

```
1    if not os.path.exists(pipe_name):
2       os.mkfifo(pipe_name)
```

Die Client-Prozesse öffnen diese Pipe im Schreibmodus, um eine Nachricht an den Server zu senden, dass sie dem Spiel beigetreten sind:

```
1    with open(pipe_name, 'w') as pipe:
2          pipe.write(f"{name} ist beigetreten.\n")
3          print(name + " ist beigetreten.")
4          pipe.flush()
```

Im Server-Prozess wird außerdem eine Liste der Clients erstellt, die befüllt wird, sobald ein Client die Beitrittsnachricht gesendet hat. Über diese Liste kann der Server im Falle eines Gewinns oder einer freiwilligen Spielbeendigung die entsprechende Nachricht an die jeweiligen Clients senden. Dadurch wird eine Kommunikation zwischen allen Client-Prozessen (Spieler) über den Server ermöglicht.

Ein Client-Prozess sendet die Nachricht *NAME* HAT GEWONNEN! an den Server. Dieser überprüft die Liste und schickt die Nachricht *CLIENT*, DU HAST VERLOREN :( an alle Clients, die nicht den jeweiligen Namen besitzen, der in der ersten Nachricht zu finden ist. Damit haben alle Spieler Zugriff auf den Spielstatus und zu diesem Zeitpunkt wird die Bingo-Karte deaktiviert, sodass keiner weiterspielt. Hiermit ist die Runde beendet.

## 3.2 Installation von WSL, Ubuntu und Nutzung von Pipes in Linux

Als nächstes ergibt sich das Problem mit dem Ausführen des Programms, da die *mkfifo*-Funktion nicht für Windows gedacht ist. Dafür hat sich jeder, mit Schwierigkeiten, Ubuntu und WSL installiert, und somit funktioniert das Programm. Jedoch ist hier zu beachten, dass pyTermTk auch in der Ubuntu-Umgebung installiert sein muss und dazu muss das Team entsprechend weitere Kommandos ausführen, um dies zu ermöglichen. Nach einer langen Recherche erlangt dies jedem Mitglied und somit ist das Problem gelöst.

## 3.3 Logdatei

Die Implementierung einer detaillierten Logdatei ist eine besondere Herausforderung. Die Logdatei muss nicht nur jede Aktion eines Spielers aufzeichnen,

sondern auch sicherstellen, dass die Daten in einem lesbaren und verständlichen Format vorliegen. Ein zusätzliches Problem war die Synchronisation der Logdatei mit dem tatsächlichen Spielverlauf, um sicherzustellen, dass keine wichtigen Ereignisse verloren gehen.

Zunächst entscheidet sich das Team, für jede generierte Bingokarte eine individuelle Logdatei anzulegen. Für die Implementierung der Logdatei verwendet das Team die logging-Bibliothek von Python, die es ermöglicht, zeitgestempelte Einträge zu erstellen und diese strukturiert in einer Datei zu speichern.

```
log_filename = f"{timestamp}-bingo-{self.name}.txt"
```

Der Dateiname wird erstellt, wobei der aktuelle Zeitstempel und der Spielername (self.name) verwendet werden. Dies stellt sicher, dass jeder Dateiname eindeutig ist.

```
self.logger = logging.getLogger(name)
```

Ein Logger-Objekt wird erstellt oder abgerufen, das dem angegebenen Namen (name) zugeordnet ist. Dadurch können Logeinträge gruppiert und organisiert werden.

```
handler = logging.FileHandler(log_filename)
```

Ein FileHandler wird erstellt, um Logeinträge in die Datei mit dem erstellten Dateinamen zu schreiben.

```
formatter = logging.Formatter('%(asctime)s - %(message)s')
```

Ein Formatter wird definiert, der den Zeitstempel (asctime) und die eigentliche Lognachricht (message) im gewünschten Format enthält.

```
handler.setFormatter(formatter)
```

Der Formatter wird dem FileHandler hinzugefügt, um sicherzustellen, dass alle Logeinträge entsprechend formatiert werden.

```
self.logger.addHandler(handler)
```

Der Handler wird dem Logger hinzugefügt, sodass alle Lognachrichten über diesen Handler in die Datei geschrieben werden.

```
self.logger.setLevel(logging.DEBUG)
```

Das Logging-Level des Loggers wird auf DEBUG gesetzt, was bedeutet, dass alle Nachrichten ab diesem Level erfasst werden.
Die Logdatei beginnt mit einem Eintrag für den Spielstart und endet mit der Spielbeendigung. Zusätzlich werden alle relevanten Aktionen, wie das Markieren von Wörtern und das Drücken des Bingo-Buttons, durch spezielle Log-Nachrichten dokumentiert. Damit die Logdatei stets synchron mit dem Spielverlauf blieb, werden alle Log-Einträge unmittelbar nach der entsprechenden Aktion im Spiel geschrieben.
Diese Implementierung gewährleistet, dass alle relevanten Aktionen und Informationen während des Bingo-Spiels in die Logdatei geschrieben werden, wodurch eine nachvollziehbare Aufzeichnung des Spielverlaufs ermöglicht wird.

## 3.4 Erstellen der Bingokarte

Mithilfe von klickbaren Buttons (aus der pyTermTk-Bibliothek) in einem separaten Fenster, wird eine Bingokarte generiert, die die Wörter aus einer Datei enthält. Die Anzahl der Felder in x- und y-Richtung wird beim Erstellen des Servers festgelegt. Zudem sollten die Felder in der Bingokarte korrekt positioniert werden, sodass die Anzahl der Felder auf der x- und y-Achse übereinstimmen. Bei einer ungeraden Eingabe für die Anzahl der Felder soll ein Joker-Feld in der Mitte der Karte gesetzt werden.

Durch verschachtelte Schleifen platzieren sich die Felder an den richtigen Positionen. Eine zusätzliche Bedingung, dass wenn die Anzahl der Felder ungerade und das Feld in der Mitte der Karte positioniert ist, ruft die Methode

`"JOKER_ausfüllen"`

auf. Diese Methode setzt den Button auf angeklickt und ersetzt den Text mit "Joker". Um die Bingoüberprüfung in einer separaten Methode zu gewährleisten, werden die Felder in eine 2D ArrayList gespeichert.

## 3.5 Bingo-Button

Der Bingo-Button ist eine zentrale Komponente des Spiels, die es den Spielern ermöglicht, einen möglichen Sieg zu melden. Das Hauptproblem besteht darin, sicherzustellen, dass der Button nur dann gedrückt werden kann, wenn der Spieler tatsächlich eine vollständige Bingo-Reihe hat. Ein weiteres Problem ist die Handhabung von Fehlermeldungen, wenn ein Spieler den Button drückt, ohne tatsächlich eine Bingo-Reihe zu haben.

Um dieses Problem zu lösen, wird ein spezieller Bingo-Button in die Benutzeroberfläche integriert. Dieser Button kann nur angeklickt werden, wenn der Spieler tatsächlich eine vollständige Reihe, Spalte oder Diagonale markiert hat.

```
1    if all(self.felder_matrix[i][i].isChecked() for i in range(len(self.
         felder_matrix))):
2        bingo = True
3
4    if all(self.felder_matrix[i][len(self.felder_matrix) - 1 - i].
         isChecked() for i in range(len(self.felder_matrix))):
5        bingo = True
6
7    for row in self.felder_matrix:
8        if all(feld.isChecked() for feld in row):
9            bingo = True
10           break
11
12   for col in range(len(self.felder_matrix[0])):
13       if all(row[col].isChecked() for row in self.felder_matrix):
14           bingo = True
15           break
```

Dazu wird eine Funktion zur Überprüfung des Spielstatus implementiert, die beim Drücken des Buttons aufgerufen wird. Diese Funktion überprüft alle möglichen Bingo-Konstellationen und meldet einen Sieg nur dann, wenn die Bedingungen erfüllt sind. Zudem wird ein Mechanismus eingeführt, der sicherstellt, dass nach dem Drücken des Bingo-Buttons keine weiteren Änderungen am Spielfeld vorgenommen werden können. Dadurch wird verhindert, dass Spieler nach einer Siegmeldung weiterhin am Spiel teilnehmen können.

```
1   def bingo_check(self):
2       if self.pruefe_Ob_Bingo():
3           self.has_won = True
4           self.zeige_gewonnen_nachricht()
5           message = f"{self.name} hat gewonnen!!!"
6           logging.info(message)
7           self.logger.info("Ende des Spiels")
8           with open(self.pipe_name, 'w') as pipe:
9               pipe.write(message + "\n")
10              pipe.flush()
```

Um die Benutzerfreundlichkeit zu erhöhen, wird der Spieler durch eine Nachricht informiert, ob er tatsächlich gewonnen hat oder nicht. Die Nachricht wird sowohl auf der Benutzeroberfläche des Spielers als auch im Server-Log angezeigt.

## 3.6 Neue Runde-Methode

Der Button “neue Runde” startet keine neue Runde. Es werden jedoch alle Eingaben blockiert und das Bild friert ein.
Um das Einfrieren zu lösen, wird die Methode clear_layout implementiert. Dies führt jedoch zu einem neuen Problem, da kein neues Spielfeld mehr erstellt wird. Nach dem Löschen des alten Bingo Felds, wird kein neues Feld erstellt obwohl die Methode create_bingo_card aufgerufen wird.
Die Lösung dieses Problems besteht darin, die Klasse *Spieler* um die Liste *words[]* zu erweitern. Zudem wird in der Methode "neue Runde" anstelle von *self.words*, *self.words.copy()* verwendet.
Nachdem diese Probleme behoben sind, ist der Button voll funktionsfähig.
Ein Problem besteht jedoch weiterhin: Die Nachrichten gehen aufgrund von unbekannten IPC-Problemen verloren. Andere Spieler werden ab der zweiten Runde deshalb nicht mehr über ein verlorenes Spiel informiert.
Deshalb ist die Funktion zwar fertig programmiert, hat es jedoch nicht in die finale Version des Programms geschafft.

## 3.7 Synchrone Spielbeendigung

Ein weiteres Problem ist die synchrone Beendigung des Spiels auf allen verbundenen Terminals. Nachdem ein Spieler "Bingo“ gerufen hat und das Spiel beendet ist, muss sichergestellt werden, dass alle anderen Spieler ebenfalls informiert werden und das Spiel für sie beendet wird. Dies stellte eine besondere Herausforderung dar, da die Spielinstanzen unabhängig voneinander liefen und kommunizieren mussten.

Die Lösung dieses Problems besteht in der Implementierung einer Interprozesskommunikation über benannte Pipes. Sobald ein Spieler "Bingo" ruft und die Überprüfung erfolgreich ist, sendet dieser Spieler eine Nachricht an den Server, der den Sieg des Spielers bestätigt. Der Server sendet daraufhin eine Nachricht an alle anderen Spieler, dass das Spiel beendet ist und sie verloren haben. Diese Nachricht wurde zuerst nur auf dem Terminal des Spielers, welcher gewonnen hat, angezeigt. Es stellt sich heraus, dass die Nutzung von pyTermTk in eine Endlosschleife – mainloop – passiert, um die Ereignisse auf der Bingo-Karte zu verzeichnen. Die Client-Prozesse müssen allerdings aus dieser Schleife herausspringen, um eine Nachricht über die Pipe zum Gewinn senden zu können. Die Lösung ist die Nutzung von Threading, um nebenbei weitere Prozesse laufen lassen zu können. Im Client-Prozess wird ein separater Thread

```
threading.Thread(target=lese_pipe, daemon=True).start()
```

gestartet, um das Lesen aus der Pipe asynchron auszuführen. Der Thread ruft die Funktion lese_pipe auf, um auf eingehende Nachrichten zu reagieren.

Erklärung einzelner Bestandteile:

```
1. threading.Thread:
      Dies ist eine Klasse aus dem threading-Modul in Python, die es
          ermöglicht, einen neuen Thread zu erstellen.
2. target=lese_pipe:
      Das Argument target gibt die Funktion an, die der Thread ausfü
          hren soll. In diesem Fall ist lese_pipe die Funktion, die im
          neuen Thread ausgeführt wird.
3. daemon=True:
      Dieses Argument gibt an, dass der Thread ein Daemon-Thread ist.
          Daemon-Threads laufen im Hintergrund und werden automatisch
          beendet, wenn der Hauptprogramm-Thread beendet wird. Das
          bedeutet, dass der Daemon-Thread keine vollständige Ausfü
          hrung garantieren kann, da er abrupt beendet wird, wenn das
          Hauptprogramm endet.
4. .start():
      Diese Methode startet den Thread und führt die angegebene target-
          Funktion (hier lese_pipe) im neuen Thread aus.
```

## 3.8 Anzeige der Beitritte von allen Spielern im Server

Der Server soll alle Spieler auflisten, die dem Spiel beitreten. Jedoch zeigt er nur den Beitritt des ersten Spielers an.

Ein erster Lösungsversuch war die Verwendung des *asyncio*-Moduls, um asynchrone Operationen zu ermöglichen, insbesondere für die Kommunikation zwischen Server- und Client-Prozessen über eine benannte Pipe.

Der Server-Prozess verwendet *async def* um eine asynchrone Funktion zu definieren.

```
await asyncio.sleep(0.1)
```

in der read_pipe()-Funktion stellt sicher, dass die Schleife nicht die CPU überlastet und anderen Tasks (falls vorhanden) die Ausführung ermöglicht.

`broadcast_message`

Eine Hilfsfunktion, die eine Nachricht an alle verbundenen Clients sendet. Sie iteriert über die Liste der Clients und schreibt die Nachricht in die benannte Pipe für jeden Client.

`await read_pipe(pipe_name)`

Der Server wartet asynchron auf Nachrichten von Clients über die read_pipe()-Funktion. Dadurch bleibt der Serverprozess reaktionsfähig und blockiert nicht das Lesen von Nachrichten. Mit dieser Implementierung wird der Beitritt von allen beliebigen Spielern im Server angezeigt. Jedoch löst es nicht das Problem im folgenden Absatz.

## 3.9 Anzeige des Spielergebnisses bei allen Spielern

Bei mehr als zwei Spielern bekommen nur zwei Spieler ein Fenster, in dem das Spielergebnis angezeigt wird. Durch das systematische Testen des Codes kam die Erkenntnis, dass nur ein Client-Prozess, bzw. Spieler die Nachricht vom Server empfängt. Eine Vermutung dafür ist, dass durch die Nutzung eines einzigen Pipes für die Kommunikation zwischen dem Server und allen Clients, eine Race Condition auftritt. Genauere Recherchen ergaben, dass dies auftritt, wenn mehrere Prozesse auf dieselbe Pipe zugreifen. Daraus resultiert, dass nur ein Spieler die Nachricht, die der Server in die Pipe schreibt, aus der Pipe liest.

Als Lösung soll der Server an jeden einzelnen Client-Prozess eine Nachricht senden, indem für jeden Client eine eigene Pipe erstellt wird. Der Server verwendet eine eigene Pipe, über die er Nachrichten von den Clients und Methoden, die die Funktionen der Bingokarte implementieren, liest. Beim Empfangen von Nachrichten erstellt der Server in der Methode

```
1   def broadcast_message(clients, message):
2       for client in clients:
3           client_pipe_name = f"/tmp/{client}_pipe"
4           if not os.path.exists(client_pipe_name):
5               os.mkfifo(client_pipe_name)
6           with open(client_pipe_name, 'w') as pipe:
7               pipe.write(f"{message}\n")
8               pipe.flush()
```

für jeden Client eine eigene Pipe und schreibt jeweils die Nachricht in die Pipe.

Jeder Client kann dann in der Methode

```
1   def lese_pipe():
2           client_pipe_name = f"/tmp/{name}_pipe"
3           if not os.path.exists(client_pipe_name):
4               os.mkfifo(client_pipe_name)
```

```
5             with open(client_pipe_name, 'r') as pipe:
6             ...
```

die Nachricht aus der Pipe lesen und dementsprechend verarbeiten. Dies ermöglicht, dass alle Clients die Nachricht vom Server erhalten und über das Spielergebnis benachrichtigt werden.

## 3.10 Fehlermeldungen

Diese Dokumentation beschreibt die möglichen Fehlermeldungen und deren Ursachen, die während der Ausführung des Programms auftreten können. Ein besonderer Fokus liegt dabei auf der Erkennung und Behandlung fehlerhafter Kommandozeilenargumente. Das Programm soll in der Lage sein, fehlerhafte Eingaben zu erkennen und entsprechende Fehlermeldungen auszugeben oder das Programm sinnvoll abzubrechen.

1. **ImportError**
   - `ImportError: No module named 'TermTk'`
     - *Ursache*: Das Modul `TermTk` ist nicht installiert oder nicht korrekt importiert.
2. **Argument Parsing Errors**
   - `argparse.ArgumentError: argument wordfile is required`
     - *Ursache*: Das Kommandozeilenargument `wordfile` fehlt.
   - `argparse.ArgumentTypeError: invalid int value: 'abc'`
     - *Ursache*: Das Argument `felder_Anzahl` ist keine gültige Ganzzahl.
3. **File Handling Errors**
   - `FileNotFoundError: No such file or directory: 'worldfile.txt'`
     - *Ursache*: Die angegebene Datei existiert nicht.
   - `PermissionError: Permission denied: '/tmp/bingo_pipe'`
     - *Ursache*: Es fehlen die Berechtigungen zum Erstellen oder Lesen der Pipe.
4. **Logging Errors**
   - `FileNotFoundError: No such file or directory: 'LogDatei.txt'`
     - *Ursache*: Fehler beim Erstellen oder Schreiben der Logdatei, möglicherweise aufgrund eines ungültigen Pfads oder fehlender Berechtigungen.
5. **IndexError**
   - `IndexError: list index out of range`

- *Ursache*: Zugriff auf eine Indexposition, die nicht existiert, z. B. in der Methode `pruefe_Ob_Bingo`.

6. **TypeError**
   - `TypeError: TTkButton() missing 1 required positional argument: 'parent'`
     - *Ursache*: Der Konstruktor von `TTkButton` wurde nicht mit den erforderlichen Argumenten aufgerufen.

7. **Custom Errors (durch `parser.error` ausgelöst)**
   - `SystemExit:`
     - *Ursache*: Benutzerspezifische Fehler bei der Argumentüberprüfung, z. B. wenn die Anzahl der Felder $\leq 0$ ist oder kein Dateiname angegeben wurde.

8. **Allgemeine Fehlermeldungen:**
   - `except Exception as e:`
     Konstrukt fängt alle anderen Arten von Ausnahmen ab, die während der Ausführung des Codes auftreten können, die nicht spezifisch abgefangen wurden. Das kann eine Vielzahl von Problemen beinhalten, wie z.B. Laufzeitfehler oder andere unerwartete Bedingungen.

# 4 Fazit

Dieses Projekt implementiert ein vollständiges Bingo-Spiel mit einer grafischen Benutzeroberfläche und ermöglicht das Spielen gegeneinander. Die Benutzer können entweder als Server fungieren, der das Spiel hostet, oder als Clients, die dem Spiel beitreten. Die Kommunikation erfolgt über benannte Pipes, und das Spiel wird mithilfe einer Textdatei mit Wörtern konfiguriert.

Als Erweiterung würde sich die Implementierung von Sockets eignen, um eine höhere Skalierbarkeit zu erreichen und das gemeinsame Spielen über unterschiedlichen Computer zu ermöglichen.

# 5 Anhang

Code-Übersicht:

```
1 Spieler klasse: Repräsentiert einen einzelnen Spieler im Spiel
2
3 Create-bingo-card Methode: Erstellt die Bingo-Karte als 2D-Liste und fü
      llt sie mit Wörtern.
4
5 JOKER-ausfüllen Methode: Füllt das mittlere Feld der Karte mit einem
      Joker und markiert es automatisch.
```

```
6
7  zeige-gewonnen/verloren-nachricht Methode: Zeigt eine Nachricht, wenn
       jemand das Spiel gewinnt/verliert.
8
9  on_button_clicked(self,button): Wird aufgerufen, wenn ein Feld
       angeklickt wird.
10
11 spiel_beenden(self): beendet das Spiel und schickt eine Nachricht an
       die Pipe.
12
13 bingo_check(self): Überprüft, ob es ein Bingo gibt und zeigt
       demendsprechend die Nachricht dafür.
14
15 pruefe_ob_bingo(self): Überprüft, ob es eine vollständige Spalte, Reihe
        oder Diagonale (Bingo) gibt.
16
17 start(self): Startet das Spiel.
```

Hilfsfunktionen:

```
1  open_file(filename): Öffnet eine Datei und liest die Zeilen in eine
       Liste ein.
2
3  Server- und Client-Prozesse:
4
5  server_process(...): Der Server-Prozess verwaltet das Spiel und die
       Kommunikation zwischen den Spielern.
6  client_process(...): Der Client-Prozess erstellt einen Spieler und
       seine Bingo-Karte und kommuniziert mit dem Server.
```

Hauptfunktion main():

```
1     def main(): Verarbeitet Kommandozeilenargumente und erstellt die
          Prozesse.
```

```
1     Argumente:
2  wordfile: Datei mit der Liste der Wörter.
3  felder_Anzahl: Größe der Bingo-Karte
4  -server: Option, um den Server zu starten
5  -name: Name des Spielers, startet den Client-Prozess
6
7  Um das Spiel zu starten, sind folgende Befehle im Terminal einzugeben:
8  python3 Bingo.py wordfile.txt 3 --server : hiermit startet der Server-
       Prozess und die Spielrunde
9  python3 Bingo.py wordfile.txt 3 --name "Alex" : hiermit startet der
       Client-Prozess und der Spieler fängt an zu spielen
10
11 Die Zahl 3 und der Name "Alex" sind hier nur Beispiele und können
       beliebig gewählt werden.
12
13 Wihtig ist, dass durch die mkfifo-Funktion das Programm nur in einer
       Unix-Shell funktioniert.
```

# 6 Quellenverzeichnis


- PyTermTk: https://pypi.org/project/pyTermTk/
- pyTermTk Dokumentation: https://ceccopierangiolieugenio.github.io/pyTermTk/
- GitHub: https://github.com/vanessaduyennguyen/Projekt
- argparse: https://docs.python.org/3/library/argparse.html
- asyncio: https://docs.python.org/3/library/asyncio.html