Inhaltsverzeichni

[Inhaltsverzeichnis1](#_Toc133963023)

[**1.** **V Modell:**3](#_Toc133963024)

[1.1 Architektur:3](#_Toc133963025)

[**1.2 Komponenten Spezifikation:**3](#_Toc133963026)

[**1.3 Implementierung:**3](#_Toc133963027)

[**2.** **Dokumente:**4](#_Toc133963028)

[2.1 Klassendiagramm:4](#_Toc133963029)

[2.2 Aktivitätsdiagramme:5](#_Toc133963030)

[**3.** **Software**5](#_Toc133963031)

[3.1 Verwendete Bibliotheken:5](#_Toc133963032)

[**4.** **Spiel**6](#_Toc133963033)

[4.1 Beschreibung Dateiformat zur Speicherung6](#_Toc133963034)

[4.2 Beschreibung des Nutzerinterfaces6](#_Toc133963035)

[4.3 Beschreibung der CPU7](#_Toc133963036)

[4.4 Dokumentation Spielablauf:7](#_Toc133963037)

[**5.** **Unittests:**10](#_Toc133963038)

[**6.** **Fehlersicherheit**10](#_Toc133963039)

[**7.** **Fehlersicherheit**10](#_Toc133963040)

# **V Modell:**

## 1.1 Architektur:

* Vorüberlegungen über Vorraussetzungen des Projektes
* Planung der Dateistruktur mithilfe eines Klassendiagrammes gemeinsam
* Besprechen der Übergabe und Rückgabeparameter
* Richtlinien festlegen für Kommentare, Variablen-/ Funktionsbenennung
* Zuordnung der einzelnen Komponenten zu den Personen

# **1.2 Komponenten Spezifikation:**

* Einzeln Spezifizieren und ausarbeiten anhand der ausgearbeiteten Richtlinien
* Bei Unklarheiten Rücksprache halten
* Überprüfen ob gestellte Anforderungen so umsetzbar
* Ständige Absprache über geänderte Parameter
* (Einbindung verschiedener Libraries)

# **1.3 Implementierung:**

* Erneutes planen, wenn beim Programmieren auffällt, das etwas nicht so funktioniert, wie gedacht
* Räumliches Zusammentreffen, um Austausch zu verbessern
* Testen der Funktionen nach Fertigstellung im Einzelnen
* Testen des Programmes nach Fertigstellung als Gesamtes

Entwicklungsbeispiel:

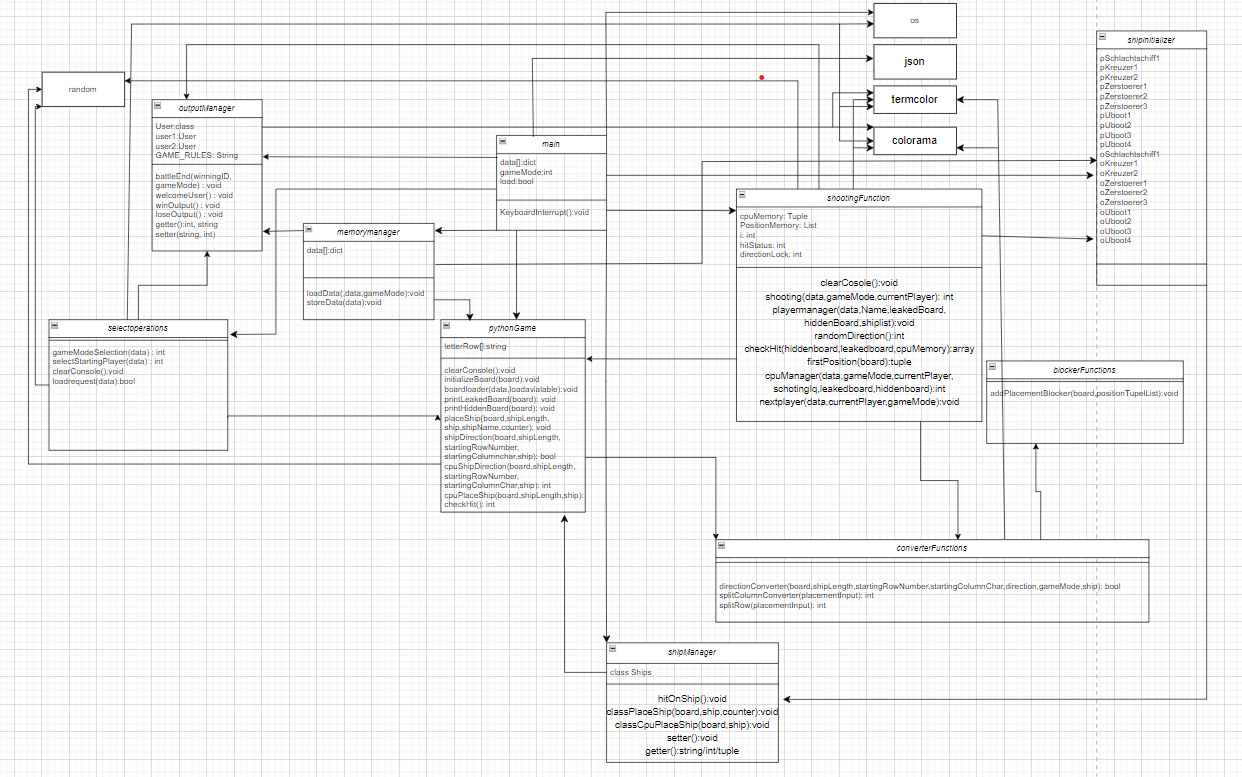
Vor Entwicklungsbeginn haben wir uns zusammen gesetzt und eine Mindmap erstellt um den Umfang des Programms zu erfassen und sinnvoll an zu ordnen. Wir haben versucht ab zu schätzen wie viel Arbeit („workload“) sich hinter welchem Teil des Programms verbirgt. Dabei haben wir auch grobe Zuständigkeiten festgelegt und versucht diese mengenmäßig ungefähr gleich zu gestalten. Anschließend haben wir festgelegt, welche Parameter und externe Libraries wir vorraussichtlich in welcher Funktion implementiert werden. Anschließend folgte eine intensive Entwicklungszeit mit viel Austausch (gewährleistet durch zusammen programmieren am Wochenende), um das Programm laufend an zu passen und funktionstüchtig zu halten. Wir haben hierbei versucht uns erst auf die Grundaufgaben der Programmteile zu fokussieren um so ein lauffähiges und testfähiges Programm zu erhalten, in dem wir dann mit der Zeit immer mehr Funktionen, sowie verbesserte CPU oder die Speicherung eines Spielstandes implementiert haben. Wir haben uns auch einmal zusammen gesetzt um zu überprüfen, ob wir die Anforderungen noch einhalten, oder ob wir unsere Arbeitsweise ändern müssen und haben unsere Fortschritte miteinander abgeglichen, um zu bestimmen, ob noch jemand Hilfe von den anderen braucht.

Gegen Ende haben wir dann nochmal kurzfristig zusammen die Variablen- und Dateinamen von camelCase auf snake\_case geändert, was nur mit einer gemeinsamen Aktion möglich war.

(Entwicklung der Tests noch einfügen und Testabdeckung)

# **Dokumente:**

## 2.1 Klassendiagramm:

(auch in draw.io einsehbar, name: classDiagram.drawio)

## 2.2 Aktivitätsdiagramme:

Wir haben für jede Funktion mit mehr als 5 Zeilen ein dazu gehöriges Aktivitätsdiagramm mit draw.io erstellt, dabei haben wir immer die Funktionen eines Teilprogramms auch in einem Diagramm festgehalten. Aus Gründen der Übersichtlichkeit haben wir darauf verzichtet die Diagramme alle einzeln in die Dokumentation ein zu binden. Die Diagramme liegen in dem Ordner „diagramme“, um die Dateien leichter finden zu können. Die Diagramme finden sich in dem Order mit dem Programmcode, gleichnamig wie die dazu gehörigen Teilprogramme. Öffnen lassen sich die Diagramme in der Desktopversion von draw.io oder aus dem Browser mit der Option „Vorhandenes Diagramm öffnen“. Die Funktionen, welche wir nicht als Aktivitätsdiagramm gezeichnet haben, weisen nicht die nötige Komplexität auf um mit einem Aktivitätsdiagramm etwas zu veranschaulichen. Hierunter fällt auch shipinitializer, da dort lediglich die Schiffe angelegt werden und es ansonsten keine Aktivität in dieser Datei gibt. Getter und Setter Funktionen wurden abgekürzt, um die Übersichtlichkeit des Klassendiagramms zu erhöhen. Die Softwareconvention, alle Namen in Snake Case zu machen sind auf Grund von kurzfristigen Änderungen von Camel Case auf Snake Case nicht umgesetzt, in den Diagrammen werden die Namen im Camel Case Format geschrieben.

# **Software**

## 3.1 Verwendete Bibliotheken:

Random: Wir haben die Random Bibliothek eingebunden, um es dem Computer zu ermöglichen auf ein beliebiges Feld zu schießen und seine Schiffe beliebig zu platzieren. Darüber hinaus wählt der Computer auch die Schussrichtung nach einem Treffer zufällig, über die eine „Random“ - Funktion. Also wird die Bibliotek verwendet um zufällige Zahlen zu ermitteln, welche von der CPU verwendet werden.

Unittests: Die Unittest-Bibliothek haben wir eingebunden um Unittests ausführen zu können.

termcolor/colorama: Mit diesen Bibliotheken haben wir die Ausgaben und Boards farblich gestaltet, um das Programm anschaulicher zu machen, die Aufmerksamkeit des Nutzers zu lenken und das Programm allgemein weniger eintönig zu gestalten. So haben wir zum Beispiel Warnungen für den Nutzer rot hinterlegt.

json: Json beutzen wir, um einen aktuellen Spielstand zwischenspeichern zu können, sollte das Programm mitten im Spiel geschlossen werden und eine Speicherung zur späteren Weiterführung des Spieles gewünscht sein. Zudem kann man einen gespeicherten Spielstand weiter spielen, indem man dieses zu Beginn des Programms auswählt und ein gespeicherter Spielstand in der Datei vorhanden ist.

os: Die OS Bibliothek verwenden wir um den Bildschirm zu leeren, damit man nicht sieht, wo der andere Spieler seine Schiffe platziert hat. Dazu benutzen wir die OS Bibliothek zur Ermittlung des aktuellen Pfades, auf dem das Programm ausgeführt wird.

# **Spiel**

## 4.1 Beschreibung Dateiformat zur Speicherung

Zur Speicherung der Daten verwenden wir das Dateiformat json. Zur Speicherung wandeln wir die Spieldaten in ein dictionary um. Dieses kann man über eine write-Funktion aus dem laufenden Python-Programm in die .json Datei speichern.

Für json haben wir uns entschieden, weil es uns schon bekannt war und relativ einfach ein zu binden ist. Darüber hinaus kann man eine Json-Datei gut lesen und in der Speicherdatei eventuelle Fehler leicht überprüfen und beheben. Des weiteren ist das Json-Dateiformat Speicherplatz effizient und man braucht keine große Rechenleistung, um die Datentypen umzuwandeln

## 4.2 Beschreibung des Nutzerinterfaces

Als Nutzerinterface benutzen wir die normale Konsolenausgabe.

Zu Beginn des Spiels gibt es eine große Ausgabe des Spieltitels “FleetBattle“ mittels einer print Funktion, am Ende des Spiels gibt es eine große „Sieg“ Ausgabe in Verbindung mit dem jeweiligen Spielernamen, beziehungsweise eine „Niederlage“ Ausgabe, sollte gegen die CPU verloren worden sein. Unterstützt wird die Ausgabe noch durch farbliche Elemente. So wird zum Beispiel ein Schiff versenkt, wird die dazugehörige Ausgabe in grün ausgegeben, oder eine Warnung, dass der andere Spieler die Schiffsplatzierung nicht sehen sollte wird in weißer Schrift ausgegeben und mit rotem Hintergrund unterlegt.

## 4.3 Beschreibung der CPU

Der Computer verteilt, sofern der ein Spieler Modus gewählt wurde, die Schiffe zufällig im Feld. Wenn der Computer an der Reihe ist, schießt er auf ein zufälliges Feld. Sollte dieses schon beschossen worden sein, dann schießt der Computer erneut auf ein zufälliges Feld.

Wenn der Computer einen Treffer landet, dann wird das Feld gespeichert und in eine lige Himmelsrichtung weiter geschossen, sollte ein Wassertreffer folgen wird in die entgegengesetzte Richtung weiter geschossen. Falls in dieser Richtung auch ein direkt ein Wassertreffer fällt, wird die Dimension gewechselt und die in die zwei noch verbleibenden Richtungen geschossen. Wenn alle Richtungen ausprobiert wurden, wird das Schiff vom Spieler versenkt sein. Die noch auszuprobierenden Richtungen werden im Programm von der shootingIq vorgegeben. Diese wird nach einem Wassertreffer um eins erhöht und gespeichert, sodass wenn die cpu wieder dran ist eine andere Richtung wählt, in die geschossen wird. Durch eine while Schleife wird solange in eine Himmelsrichtung geschossen, bis ein Wassertreffer fällt. Die erste Schießposition wird mithilfe der random-Bibliothek zufällig bestimmt und auch gespeichert, da man beim wechseln der Himmelsrichtung von der Ursprungskoordinate aus weiterschießen muss. Die erste Himmersrichtung in die geschossen wird wird auch zufällig in der Funktion firstDirection festgelegt und anhand dieser werden die nachfolgenden Richtungen auch angepasst.

Beim ändern der Richtung gibt es Fälle bei denen die cpu aus dem Feld hinaus schießt und somit ans andere Ende des Feldes schießt. Dies wird nicht abgefangen. Auch nachdem das Schiff versenkt wurde, schießt die cpu noch ein weiteres mal in die vorherige Himmelsrichtung, was auch nicht optimal ist, allerdings fehlte uns die nötige Zeit um dies auch noch umzusetzten. Die stärke der cpu reicht allerdings aus, um auch mal gegen den menschlichen Gegener zu gewinnen.

## 4.4 Dokumentation Spielablauf:

Bevor das Spiel wird der Nutzer gefragt, ob er einen vorherigen Spielstand hat, welchen er laden möchte. Sollte dass der Fall sein springt der Nutzer sofort in sein gespeichertes Spiel zurück, welches über eine json-Datei gespeichert und geladen wird, hierfür wird die Funktion loadrequest aus selectOperations verwendet.

Sollte es keinen gespeicherten Spielstand geben, oder will der Nutzer den alten Spielstand nicht mehr fortsetzten, so wird ein neues Spiel gestartet.

Hierfür wird der Nutzer gefragt, ob er im Einspieler Modus oder im Zweispieler Modus spielen möchte. Dies passiert mit Hilfe der selectStartingPlayer Funktion aus selectoperations. Ist eine Wahl getroffen, so startet die jeweilige Sequenz.

Einzelspieler Modus:

Im Einspieler Modus wird erst der Name des Computergegners auf „Der Computer“ gesetzt und der Nutzer nach seinem Namen gefragt. Anschließend wird der Nutzer aufgefordert seine Schiffe zu platzieren, was in der Funktion classPlaceShip aus shipManager passiert. Der Ablauf des Schiffe platzierens sieht vor, dass man ein Feld für die Spitze des Schiffes mit Hilfe der Tastatur auswählt und anschließend die Richtung in welche das Schiff laufen soll angeben muss(mit w,a,s,d). Hierbei gilt es zu beachten, dass die platzierten Schiffe sich nicht berühren dürfen und auch nicht über das Eck zusammenstoßen dürfen, sollte dass der Fall sein, so muss man das Schiff erneut auf einen freien Platz des Boards platzieren. Der Computer platziert anschließend heimlich seine Schiffe in classCpuPlaceShip ebenfalls aus der Datei shipManager. Er sucht sich immer zufällige Felder und eine zufällige Richtung aus und überprüft im Anschluss, ob es Regelkonform ist, dass Schiff dort zu platzieren. Ist die CPU fertig damit ihre Schiffe zu platzieren, dann wird zufällig bestimmt, welcher Nutzer das Spiel beginnen darf, dies geschiet in der Funktion selectStartingPlayer.

Anschließend beginnt das tatsächliche Spiel, in shooting aus shootingfunction darf der ausgeloste Startspieler beginnen ein Feld zu beschießen. Wir nehmen an, dass die CPU beginnen darf. Die CPU beginnt, indem sie zu beginn auf ein zufällig ausgewähltes Feld schießt. Dieses Feld darf von der CPU zuvor noch nicht beschossen worden sein. Sollte die CPU einen Treffer landen, so setzt sie Logik ein und versucht dass ganze Schiff zu versenken(siehe Beschreibung CPU). Bei einem Treffer darf erneut geschossen werden. Nach jedem Schuss wird das aktualisierte Board auf der Benutzeroberfläche ausgegeben.Der CPU ist nicht möglich ein schon einmal beschossenes Feld erneut zu beschießen. Trifft die CPU jedoch Wasser, so ist der Nutzer an der Reihe, der Wechsel funktioniert über die Funktion nextPlayer aus shootingfunction. Der Nutzer bekommt das board ausgegeben und darf über Tastatureingabe wählen, wohin er schießen möchte, ein schon einmal beschossenes Feld darf nicht erneut Ziel eines Angriffes werden. Das aktualisierte Board wird ausgegeben. Bei einem Treffer darf der Nutzer erneut schießen, bei einem Wasser Treffer wird erneut die Funktion nextPlayer aus shootingfunction aufgerufen und die CPU ist wieder an der Reihe. Das ganze wiederholt sich so lange, bis einer der beiden alle Schiffe des anderen zerstört hat. Dies wird im playerManager aus der Datei shootingfunction ermittelt. Sollte einer der Beiden alle Schiffe des anderen zerstört haben, so wird die Nummer dieses Nutzers an das Hauptprogramm zurück gegeben, wo dann die Funktion battleEnd des outputmanagers eine finale Ausgabe tätigt, wer das Spiel gewonnen hat. Die beiden Boards werden geleert und die Json-Datei mit leeren Boards überschrieben, der gespeicherte Spielstand wird gelöscht. Danach beendet sich das Programm und man kann es zum wiederholten spielen erneut starten.

Zweispieler Modus:

Beim Zweispieler Modus wird erst der erste Nutzer aufgefordert seinen Namen ein zu geben. Bevor dieser seine Schiffe über Tastatureingabe eingeben soll, wird der zweite Nutzer dazu aufgefordert den Bildschirm nicht an zu schauen, damit er die Position der gegnerischen Schiffe nicht schon kennt. Das Platzieren funktioniert nach dem gleichen Prinzip wie beim Nutzer im Singleplayer. Eingabe des ersten Feldes über die Tastatur und Richtungswahl mit w,a,s,d. Auch hier gilt zu beachten, dass die platzierten Schiffe sich nicht berühren dürfen und auch nicht über das Eck zusammenstoßen dürfen, sollte dass der Fall sein, so muss man das Schiff erneut auf einen freien Platz des Boards platzieren.

Nachdem der erste Nutzer fertig mit seinen Eingaben ist wird der Bildschirm geleert und das Endgerät wird an den zweiten Nutzer weiter gereicht. Dieser darf, unter Geheimhaltung vor dem ersten Nutzer nun auch seinen Namen eingeben und seine Schiffe nach dem selben Prinzip platzieren, all dies geschieht in der Funktion gameModeSelction, die Schiffe werden mit Hilfe der Funktion classPlaceShip aus der Datei shipManager platziert. Anschließend wird der Startspieler über die Funktion selectStartingPlayer aus selectoperations zufällig festgelegt und übergeben. In der Funktion shooting aus der Datei shootingfunction darf der bestimmte Nutzer beginnen. Der Nutzer, welcher an der Reihe ist bekommt nun das Board, auf welches er schießen soll ausgegeben, im ersten Zug sind alles Wasserfelder. Der Nutzer, welcher an der Reihe ist, darf nun auf ein Feld seiner Wahl schießen, er tut dies über eine Tastatureingabe, welche in playermanager in der Datei shootingfunction ausgeführt wird. Trifft er, so erscheint an der Stelle ein x, verfehlt er so wird ein o gezeichnet. Bei einem Treffer darf der Nutzer, der an der Reihe war erneut schießen, verfehlt er, so ist der andere Nutzer an der Reihe und die Konsole wird geleert. Der andere Nutzer darf ebenfalls über eine Tastatureingabe schießen. Auch bei ihm gilt, wenn ein Treffer gelandet wurde, dann darf er erneut schießen, sollte er Wasser treffen, ist der andere Nutzer wieder an der Reihe. Die Boards werden nach jedem Schuss aktualisiert und erneut auf der Konsole ausgegeben. Es ist nicht möglich ein schon einmal beschossenes Feld erneut zu beschießen. Sobald es einem der beiden Nutzer gelingt alle Schiffe des Gegners zu versenken wird seine Nummer an das Hauptprogramm zurück gegeben. Die Boards der Spieler werden geleert und in die Json-Datei geschrieben um den eventuell gespeicherten Spielstand zu löschen. Im Anschluss wird für den Nutzer, welcher gewonnen hat eine Funktion winOutput aus der Datei outputmanager aufgerufen, die zeigt, welcher Nutzer gewonnen hat.

Abschließend beendet sich das Programm und man kann es erneut starten um erneut eine Runde Schiffe versenken Spielen zu können.

# **Unittests:**

(Logs Tests/Coverage)

# **Fehlersicherheit**

(Wie wurde mit Fehlern umgegangen, wie abgefangen)

# **Fehlersicherheit**