

**Dokumentation –**

**Snakes and Ladders**

**Softwareprojekt**

im Studienfach „Programmieren II“

vorgelegt von

**Team „Kurzer Prozess“**

**Lukas Becker: 2160710**

**Steven Geiger: 2180131**

**Sebastian Linn: 2180749**

**Nick Ringelmann: 2180749**

am 14.06.2019

an der Hochschule Hamm-Lippstadt

**Inhaltsverzeichnis**

1 Projektbeschreibung 1

1.1 Anforderungen 1

1.2 Zielsetzung 1

2 Planung und begleitende Dokumentation 1

2.1 Projektplan 2

2.2 Arbeitspakete und Verantwortlichkeiten 2

2.3 Aufwands-/ Ressourcenschätzung 3

2.4 Ablaufplanung 3

3 Durchführungsdokumentation 4

*4* Ergebnisse 10

# Projektbeschreibung

In diesem Projekt wird das Spiel „Snakes and Ladders“ mit einer Java-Lösung umgesetzt. Dabei wenden wird die weiterführenden Begriffe, Methoden und Konzepte des Programmierens an und analysieren Problemstellungen. Durch dieses Vorgehen können komplexe, praktische Probleme eigenständig in Java gelöst werden, indem die Grundlagen der objektorientierten Programmierung angewendet werden.

## Anforderungen

Das Projekt in Form einer Umsetzung eines Spieles zielt vor allem auf den Einbezug der grafischen Programmierung ab. Hierbei soll vorwiegend auf Swing zur Erstellung einer ansprechenden GUI zurückgegriffen werden. Das Spiel kann an einem, sowie an zwei miteinander verbundenen Geräten gespielt werden. Hierbei versuchen wir die Bordmittel von Java bestmöglich auszunutzen.

## Zielsetzung

Das Ziel des Projekts ist ein möglichst flüssiges und grafisch ansprechendes Spielerlebnis in Java umzusetzen. Die Software soll dabei leicht verständlich sein und sich einfach handhaben lassen. Ein besonderes Augenmerk wird auf die Ausgestaltung des Spielfelds gesetzt. Die Spieler sollen sich möglichst flüssig auf dem 100 Feld großem Spielfeld bewegen. Der Spieler, welcher das Feld 100 zuerst erreicht, hat das Spiel gewonnen. Die Spieler würfeln per Ausführung des Würfels, wodurch eine zufällige Zahl zwischen 1 und 6 generiert wird. Das Spielfeld ist mit Ufos und Wurmlöchern bestückt. Dabei verhelfen die Ufos dem Spieler Felder zu überspringen. Wurmlöcher dagegen schicken Spieler auf eine tiefere Ebene des Spieles zurück. Die Spieler selbst sind mit verschiedenen Farben gekennzeichnet. Ein weiteres Feature ist die verteilte Programmierung, wodurch Personen an verschiedenen Geräten miteinander interagieren können.

# Planung und begleitende Dokumentation

In diesem Kapitel finden sie eine Darstellung des Projektplans, die Arbeitspakete und deren Verantwortlichkeiten und die Ablaufplanung.

## Projektplan

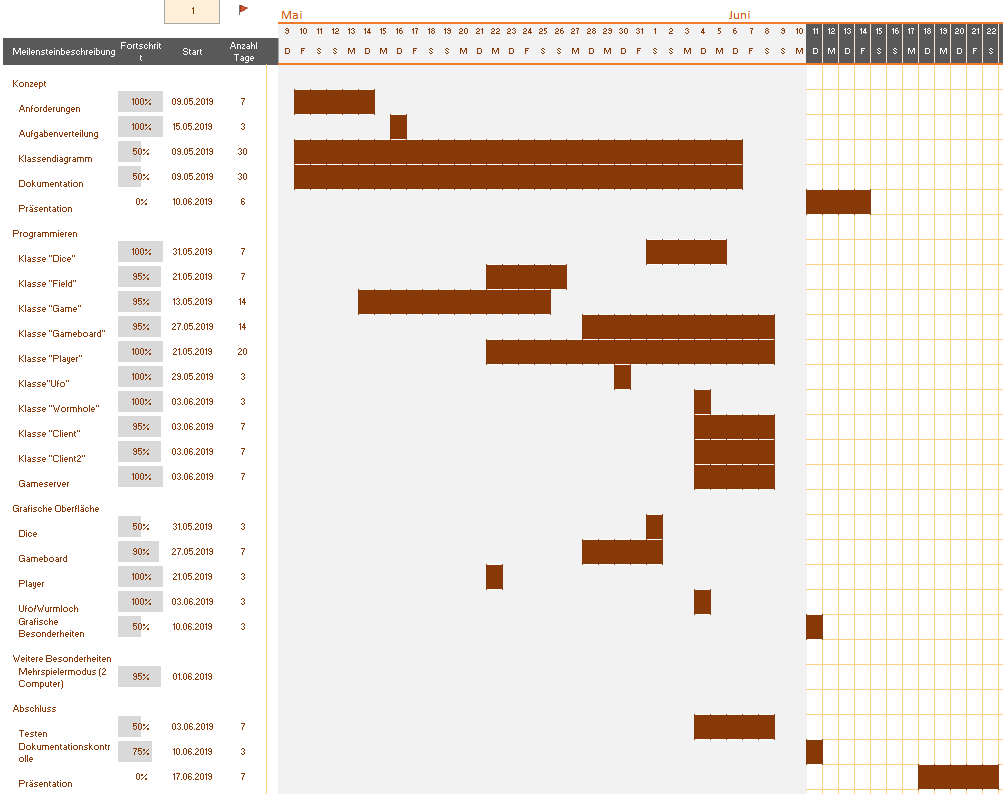


Abbildung 1: Projektplan

## Arbeitspakete und Verantwortlichkeiten

Lukas Becker: Klassendiagramm, Dokumentation, Präsentation

Basti: Grafische Umsetzung, Klasse Ufo und Wurmloch

Nick Ringelmann: Grafische Umsetzung, Klasse Field

Steven Geiger: Spiellogik (Game), verteilte Programmierung

## Aufwands-/ Ressourcenschätzung

Abbildung 2: Aufwands-/ Ressourcenschätzung

## Ablaufplanung

Abbildung 3: Ablaufplan

# Durchführungsdokumentation

Zu Anfang der Bearbeitungsphase unseres Projekts erstellten wir nach einiger Überlegung ein erstes Klassediagramm, um abzustecken, welche Klassen und Funktionen wir benötigen. Nach der Aufteilung in verschiedene Klassen erstand folgendes Klassendiagramm, welches im Laufe der Programmierung angepasst und weiter bearbeitet wurde.



Abbildung 4: Klassendiagramm

Wie in Abbildung 4 zu erkennen ist haben wir die Problemstellung in 11 Klassen und ein Interface aufgespalten, sodass wir nun auf eine überschaubare Struktur zurückgreifen konnten, um das weitere Programmiergeschehen zu koordinieren. Im folgenden Kapitel wird zuerst auf das Spielgeschehen auf einem Gerät eingegangen, bevor die verteilte Programmierung betrachtet wird.

Die Klasse *Game* ist dabei der Ausgangspunkt des Spieles. Sie initialisiert das Spiel, welches über die *Main-Klasse* gestartet wurde. Dabei wird auf die Klasse *Gameboard* zugegriffen, um die Spielfeldgröße abzufragen. Im gleichen Zug werden die Spieler, der Würfel und das Spielfeld erzeugt. Des Weiteren überprüfen die Methoden *checkUfo* und *checkWormhole*, ob sich der aktive Spieler auf einem Feld befindet, dass mit einem Ufo oder Wurmloch ausgestattet ist. Ist dieses der Fall, so wird der Spieler, wie in Abbildung 5 beschrieben, 20 Felder hinauf oder hinab gesetzt.

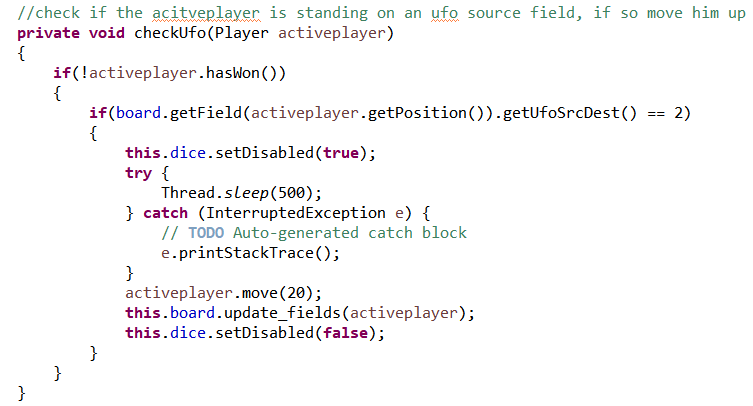


Abbildung 5: Abfrage auf Ufo-Feld

Eine weitere Funktion, welche per Methode *movePlayer* umgesetzt wurde, ist das setzen der Spielfiguren mit gleichzeitiger Aktualisierung des *Gameboards* und nachfolgender Abfrage, welcher Spieler nun am Zug ist. Eine der wichtigsten Methoden ist die *run-Methode*, welche veranlasst die zuvor erwähnten Methoden solange auszuführen sind bis die beiden Bedingungen erfüllt sind. Sobald die Bedingungen erfüllt sind, springen wir zu der Siegerausgabe und verlassen das Spiel. Im Folgenden ist die Beschaffung der Würfelziffer, das Setzen des jeweiligen Spielers, die Abfrage der Position und der Spielerwechsel zu sehen. Ebenfalls lässt sich hier die Gewinnerlogik gut darstellen.

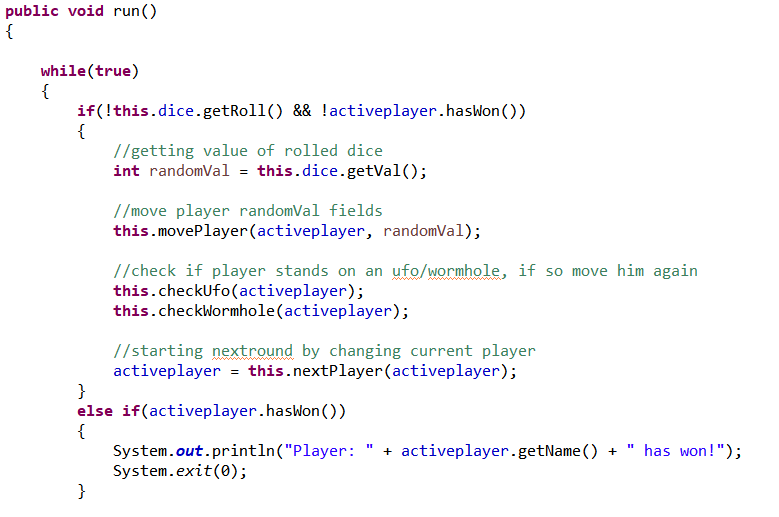


Abbildung 6: Spiellogik

Nachdem wir nun die Hauptfunktionen betrachtet haben, werden nachfolgend die Klassen *Gameboard* und *Field* genauer betrachtet. Erstere legt den Fokus auf das *Graphical User Interface (GUI).* Hierbei wird ein JFrame erstellt und Einstellungen für die Höhe, Breite und die Anzahl der Felder festgehalten. Außerdem werden Standardfunktionen, wie die Ausrichtung des Spielfelds und die Aktion beim Klicken des Schließen-Buttons eingefügt. Außerdem wird die Gittergrundlage des Spielfelds über die Klasse *Grindbagconstraints* eingestellt. Die Abmessungen des Spielfelds und der Einzelfelder wird implementiert, bevor die Ufos und Wurmlöcher initialisiert werden und ein letztes Update durchgeführt wird, um alle Änderungen zu aktualisieren. In Abbildung 7 kann man die Initialisierung der Ufos und Wurmlöcher betrachten, wobei gut die Verwendung des zuvor eingefügten Hintergrundgitters zu erkennen ist, da die Startpunkte und dessen Länge angegeben sind.

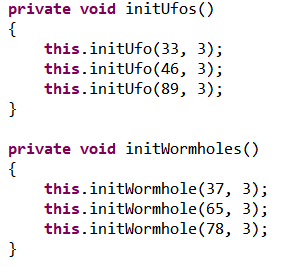


Abbildung 7: Initialisierung der Ufos und Wurmlöcher

Über die Initialisierung der Funktion der einzelnen Ufos und Wurmlöcher wird per for-Schleife das Versetzen der Spielfiguren veranlasst. Die Methode *initWormhole* und *initUfo* beschreibt das Versetzen genauer.Durch die folgenden Abfragen in der for-Schleife wird geregelt, dass ein Spieler bei dem Betreten eines Ufo-Felds hinauf und nicht hinab gesetzt wird. Gleiches gilt in umgekehrter Funktion für die Wurmloch-Felder. Um die Position nun nach jedem Spielzug zu aktualisieren, wird durch die Methode *update\_fields* geprüft, ob der Spieler bereits gewonnen hat, wenn dies nicht der Fall ist, werden die Felder nach jeder Runde aktualisiert.

Die Klasse *Field* beschreibt zum großen Teil das Aussehen jedes einzelnen Felds. Hierbei wird jedem Feld eine Id, ein Link und die Höhe und Breite zugewiesen. Der Link ist hierbei die Verbindung von Feldern durch Ufos und Wurmlöcher. Außerdem wird die Id -100 gesetzt, sodass der Spielstart auf das unterste linke Feld fällt. Wie in Abbildung 8 zu erkennen wird der Hintergrund für die einzelnen Bilder durch die Methode *init\_bg\_picture* festgelegt. Im gleichen Zuge wurde ein Overlay-Layout implementiert, welches die Anzeige der Spielfiguren über dem Spielfeldhintergrund sicherstellt. Außerdem werden die Spielfeldnummern hinzugefügt, welche in der Methode *initFieldNumber* implementiert wurden. Dabei wurden Einstellungen zur Schriftfarbe und Schriftgröße getätigt.

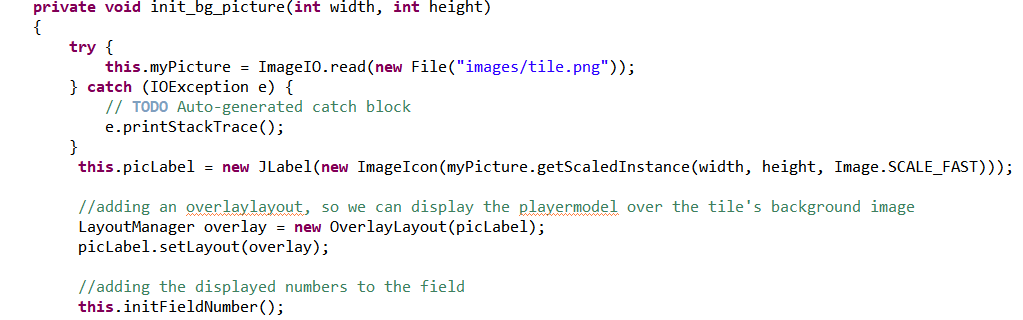


Abbildung 8: Feldhintergrund und Overlay

Um die richtige Ausrichtung der Spielfiguren zu gewährleisten wurde in der Methode *setplayer* die zentrierte Anzeige per *setAligmentX* und *setAligmentY* festgelegt. In den Methoden *setUfo* und *setWormhole* sind die grafischen Eigenschaften der Ufos und Wurmlöcher auf dem Spielfeld implementiert. Dabei wird geprüft, ob das Feld keine *WormModel* hat und das Feld kein Startpunkt oder Endpunkt eines Ufos oder Wurmlochs ist (s. Abbildung 9). Treffen diese beiden Aussagen zu, so werden, wie in Abbildung 7 gezeigt, Ufos oder Wurmlöcher, wie im Gameboard implementiert hinzugefügt. In den Klassen *Ufo* und *Wormhole* wurden die *getter-* und *setter-Methoden* für die Größe implementiert.



Abbildung 9: setWormhole

Nachdem nun der Gesichtspunkt auf das Spielfeld und die einzelnen Felder gesetzt wurde, werden anchließend die Klassen *Dice* und *Player* genauer betrachtet. Zu Anfang wird die grafische Oberfläche des Würfels implementiert. Über *JFrame* wird ein extra Fenster angelegt, welches mit einem Button mit der Beschriftung *Dice* per *JButton* umgesetzt wurde. Bei der Hintergrundstruktur wurde wieder auf das *gridbagconstraints* zurückgegriffen. Wie in Abbildung 10 zu erkennen ist wird über die Methode *setVal* eine Zufallszahl zwischen 1 und 6 erzeugt. Der *ActionListener* betrachtet die Änderungen des Buttons.

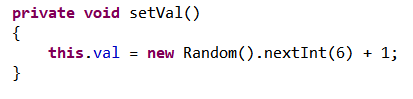


Abbildung 10: Zufallszahl Würfel

Die Klasse *Player* beschreibt das Aussehen, den Namen, die Position des Spielers und die Gewinnerabfrage. Durch die Methode *setModel* wird die ID des Spielers per „if-else“ abgefragt und den Spielern verschiedene Spielfiguren zugewiesen. Über die Setter-Methode *setPosition* wird die neue Position des Spielers festgesetzt. In der Methode *move* wird die Bewegung des Spielers in Relation seiner aktuellen Position festgehalten. Wenn der Spieler seine Felder nun setzt, wird überprüft, ob seine Endposition kleiner 100 ist. Ist dieses der Fall, so wird ganz normal gesetzt. In dem Falle, dass seine Endposition Feldposition 99 ist, so hat der Spieler gewonnen (s. Abbildung 11).

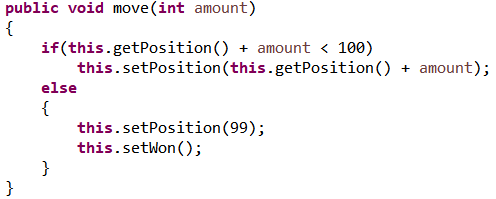


Abbildung 11: Move-Methode

Dem Spieler wird nachfolgend der Won-Flag gesetzt.

Der zweite große Fokus lag neben der grafischen Oberfläche auf der verteilten Programmierung. Für das Spielen auf 2 Geräten implementierten wir die Klassen *Client, Client2* und *Gameserver.* Des Weiteren mussten wir eine Schnittstelle erstellen, das *Gameinterface.* Auf der Server-Schnittstelle wurden alle Methoden implementiert, die wir serverseitig für einen flüssiges Spielerlebnis benötigten. In Abbildung 12 sehen wir die Methoden *moveOnlinePlayer*, *turn, roundFinished und getOtherPlayer.* All diese Methoden können (Fehler-) Meldungen ausgeben. Die Implementierung der Ausgaben, sowie das Setzen der Spieler über die *turn-Methode* ist in der Klasse *Game* beschrieben.

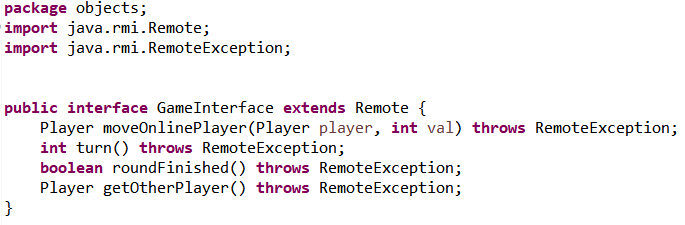


Abbildung 12: GameInterface

In der Klasse *GameServer* wird die Instanziierung der Implementierungsklassen durchgeführt. Außerdem werden die Objekte der Implementierungsklasse und die das Remote-Objekt exportiert. Anschließend wird das Remote-Objekt in der Registry eingebunden. Im Folgenden sehen wir den genauen Programmcode:

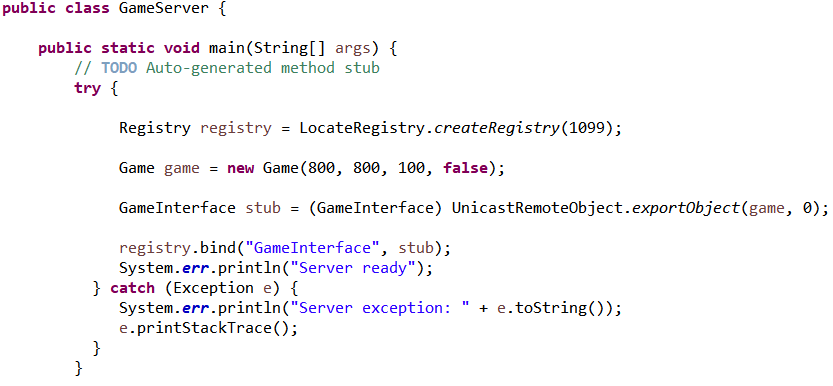


Abbildung 13: GameServer

# Ergebnisse

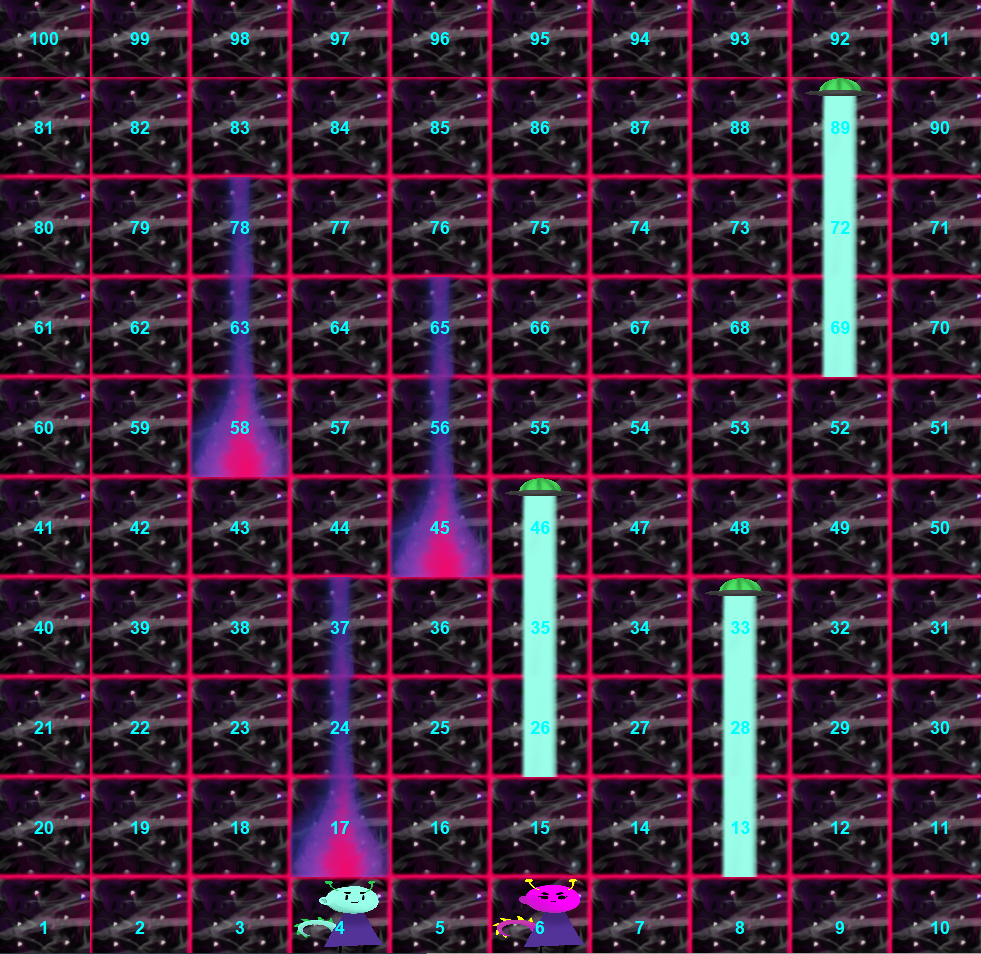


Abbildung 14: Gameboard

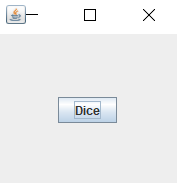


Abbildung 15: Dice