JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von:
**Johannes Hacker**
**Moritz Neuwirth**
**Julian Lumetsberger**

Gruppe:
Gruppe 4

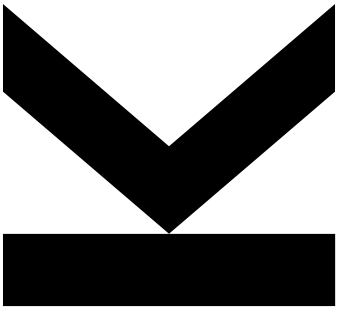
Angefertigt am:
**Institut für Wirtschaftsinformatik – Software Engineering**

Beurteiler / Beurteilerin:
**Dr. Johannes Sametinger**

Mitbetreuung
**-**

Datum
18.07.2023

# GO-SPIEL



Systemdokumentation

PR Software Engineering
259.035
SS 2023

**JOHANNES KEPLER**
**UNIVERSITÄT LINZ**
Altenberger Straße 69
4040 Linz, Österreich
jku.at
DVR 0093696

## Inhaltsverzeichnis

# 1. Zweck und Zielgruppe

Das vorliegende Kapitel dient dazu, die Zwecke und Ziele dieser Systemdokumentation zu erläutern. Hier werden die grundlegenden Absichten der Dokumentation erläutert, um eine klare Orientierung und Verständnis über die Software zu vermitteln. Die Zielgruppe, für die diese Dokumentation erstellt wurde, wird ebenfalls beschrieben.

## 1.1. Zweck

Der Hauptzweck dieser Systemdokumentation besteht darin, eine umfassende Erklärung und Übersicht über unser Go-Spiel zu bieten, die während des Entwicklungsprozesses entstanden ist. Die Dokumentation soll eine detaillierte Darstellung der Systemarchitektur, der Komponenten und der internen Struktur liefern, um den Lesern ein fundiertes Verständnis der Software zu ermöglichen. Dabei werden insbesondere die folgenden Aspekte abgedeckt:

## 1.2. Aufbau und Struktur

Die Systemdokumentation wird die Architektur der Software beschreiben und die wesentlichen Komponenten erläutern, aus denen sie besteht. Hierbei wird auf die logischen Zusammenhänge, Abhängigkeiten und Interaktionen zwischen den einzelnen Modulen und Paketen eingegangen. Dadurch erhalten die Lesenden einen umfassenden Einblick in den Aufbau der Software und können die Verbindungen zwischen den verschiedenen Teilen nachvollziehen.

## 1.3. Muster und Conventions

Ein weiterer wichtiger Aspekt dieser Dokumentation ist die Darstellung der verwendeten Conventions und Muster innerhalb des Quellcodes. Dabei werden spezifische Namenskonventionen, Design Patterns und bewährte Entwicklungspraktiken erläutert, die während der Entwicklung angewendet wurden. Dadurch soll eine einheitliche und gut strukturierte Codebasis gewährleistet werden.

# 1. Zielgruppe

## 1.1. Beurteilende Professoren

Die Professorinnen und Professoren, die das Softwareprojekt begleiten und bewerten, sind eine zentrale Zielgruppe dieser Dokumentation. Sie benötigen eine umfassende Übersicht über die Struktur und den Aufbau der Software, um das Projekt angemessen beurteilen zu können. Die Dokumentation ermöglicht es ihnen, die Designentscheidungen und die technische Umsetzung nachzuvollziehen.

## 1.2. Mitstudierende

Auch die Mitstudierenden und Entwickler*innen, die am Softwareprojekt beteiligt sind, bilden einen wichtigen Leserkreis. Diese Zielgruppe möchte die Software aus technischer Sicht verstehen und sich schnell im Quellcode zurechtfinden können. Die Systemdokumentation bietet ihnen eine Orientierungshilfe, um sich in der Architektur und den Paketstrukturen zurechtzufinden.

## 1.3. Weitere Interessenten

Darüber hinaus können auch andere Personen, die ein allgemeines Interesse an der Software und ihrer Funktionsweise haben, von dieser Dokumentation profitieren. Die Informationen sollen

es diesen Leserinnen und Lesern ermöglichen, einen Einblick in das Softwareprojekt zu erhalten, ohne dabei in technische Details einzutauchen.

Die vorliegende Systemdokumentation wurde speziell für diese Zielgruppen erstellt, um ein klares Verständnis der Softwarearchitektur zu vermitteln und eine effektive Bewertung und Nutzung der Software zu ermöglichen.

## 2. Aufbau und Struktur

Unser GO-Spiel wurde mithilfe der JavaFX-Bibliothek implementiert und folgt dem MVC (Model-View-Controller) Pattern, um eine klare Trennung der Verantwortlichkeiten zu gewährleisten. Die Ordnerstruktur des Projekts ist übersichtlich gestaltet und folgt bewährten Praktiken für die Organisation des Quellcodes und der Ressourcen.

### 2.1. Technologien und Pattern

Die Implementierung unseres GO-Spiels basiert auf der Java-Programmiersprache in Kombination mit dem JavaFX Framework für grafische Anwendungen. JavaFX ermöglicht es uns, die visuelle Darstellung des Spiels ansprechend und intuitiv zu gestalten. Als Programmierumgebung entschieden wir uns für IntelliJ IDEA Ultimate, da dies für aus unserer Sicht für die Erstellung von Java-Anwendungen die beste Wahl ist. Für die Gestaltung der GUI kam zusätzlich der JavaFX-SceneBuilder zum Einsatz, welcher sich durch einfache Drag & Drop Operationen bedienen lässt. Um die Struktur und Organisation des Codes zu optimieren, setzen wir das Model-View-Controller (MVC) Pattern ein. Dieses Pattern ermöglicht eine klare Trennung der Datenhaltung (Model), der Benutzeroberfläche (View) und der Anwendungslogik (Controller). Dadurch wird der Code besser wartbar, erweiterbar und leichter zu verstehen. Ordner / Package – Struktur
Die Ordnerstruktur (siehe Abb. 1) unseres GO-Spiels ist übersichtlich und sinnvoll organisiert, um eine klare Trennung der Komponenten und eine einfache Wartung und Erweiterbarkeit der Anwendung zu ermöglichen.

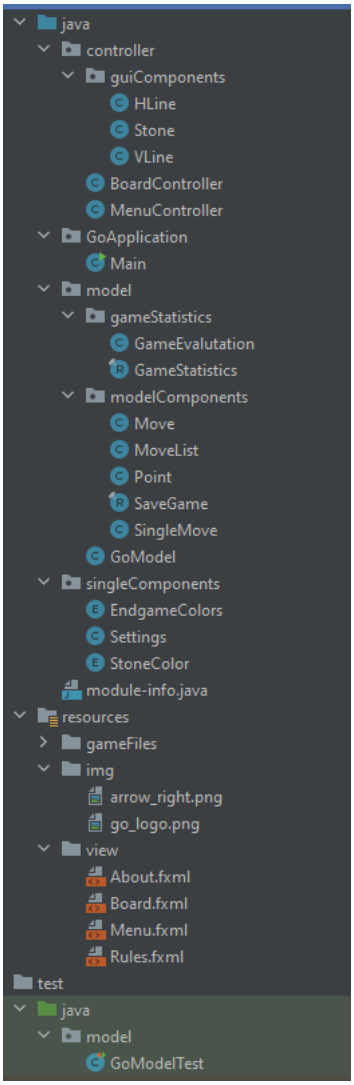


Abb. 1: Struktur

### 2.2. Hauptordner - src/main

Im Hauptordner befinden sich die Quellcodedateien für die Anwendung. Der Unterordner "java" enthält alle Komponenten für den Controller und das Model der Anwendung.

Der "controller"-Ordner bzw. das Package enthält die Klassen, die die Steuerung und Logik der Benutzeroberfläche verwalten.
Die Klasse "GoApplication" fungiert als Einstiegspunkt der Anwendung und startet das JavaFX-Programm.
Der "model"-Ordner bzw. das Package enthält die Klassen, die für die Datenhaltung und die Geschäftslogik des GO-Spiels verantwortlich sind.

Der "SingleComponents"-Ordner bzw. das Package enthält eigenständige Komponenten wie "Settings" oder "Colors", die in der Anwendung mehrfach verwendet werden. Diese Komponenten sind als separate Klassen implementiert, um eine Wiederverwendbarkeit und bessere Strukturierung des Codes zu ermöglichen.
Der Ordner "resources" enthält Ressourcen wie Bilder und FXML-Dateien, die für die grafische Benutzeroberfläche und die Gestaltung der Anwendung verwendet werden.

### 2.3. Testordner - src/test

Im Testordner befinden sich die Unit-Tests für das Model der Anwendung. Der Unterordner "model" enthält die Testklassen, die die Funktionalität und Korrektheit des Model-Teils überprüfen. Dieser Aufbau ermöglicht eine klare Trennung der Tests von den Produktionsklassen und erleichtert die Durchführung der Tests.

### 2.4. Vorteile der Struktur

Die gewählte Ordnerstruktur bietet mehrere Vorteile:

- Klare Trennung der Komponenten: Die strikte Trennung von Controller, Model und wiederverwendbaren SingleComponents erleichtert das Verständnis und die Wartung des Codes.

- Einfache Erweiterbarkeit: Durch die klare Struktur können neue Funktionen oder Komponenten leicht hinzugefügt werden, ohne die bestehende Funktionalität zu beeinträchtigen.

- Wiederverwendbarkeit: Die Einbindung von eigenständigen SingleComponents fördert die Wiederverwendung von Code und die Vermeidung von Redundanzen.

- Gute Testbarkeit: Die klare Trennung der Testklassen von den Produktionsklassen ermöglicht eine effektive Durchführung der Unit-Tests und gewährleistet eine hohe Codequalität.

Insgesamt bietet die strukturierte Ordnerstruktur eine solide Grundlage für eine gut organisierte, leicht wartbare und erweiterbare GO-Spielanwendung. Sie trägt dazu bei, dass Entwickler sich schnell in der Software zurechtfinden und die Softwarearchitektur verstehen können, was besonders für Beurteilende Professoren und Studienkolleg*innen von großem Nutzen ist.

# 3. Klassendiagramm

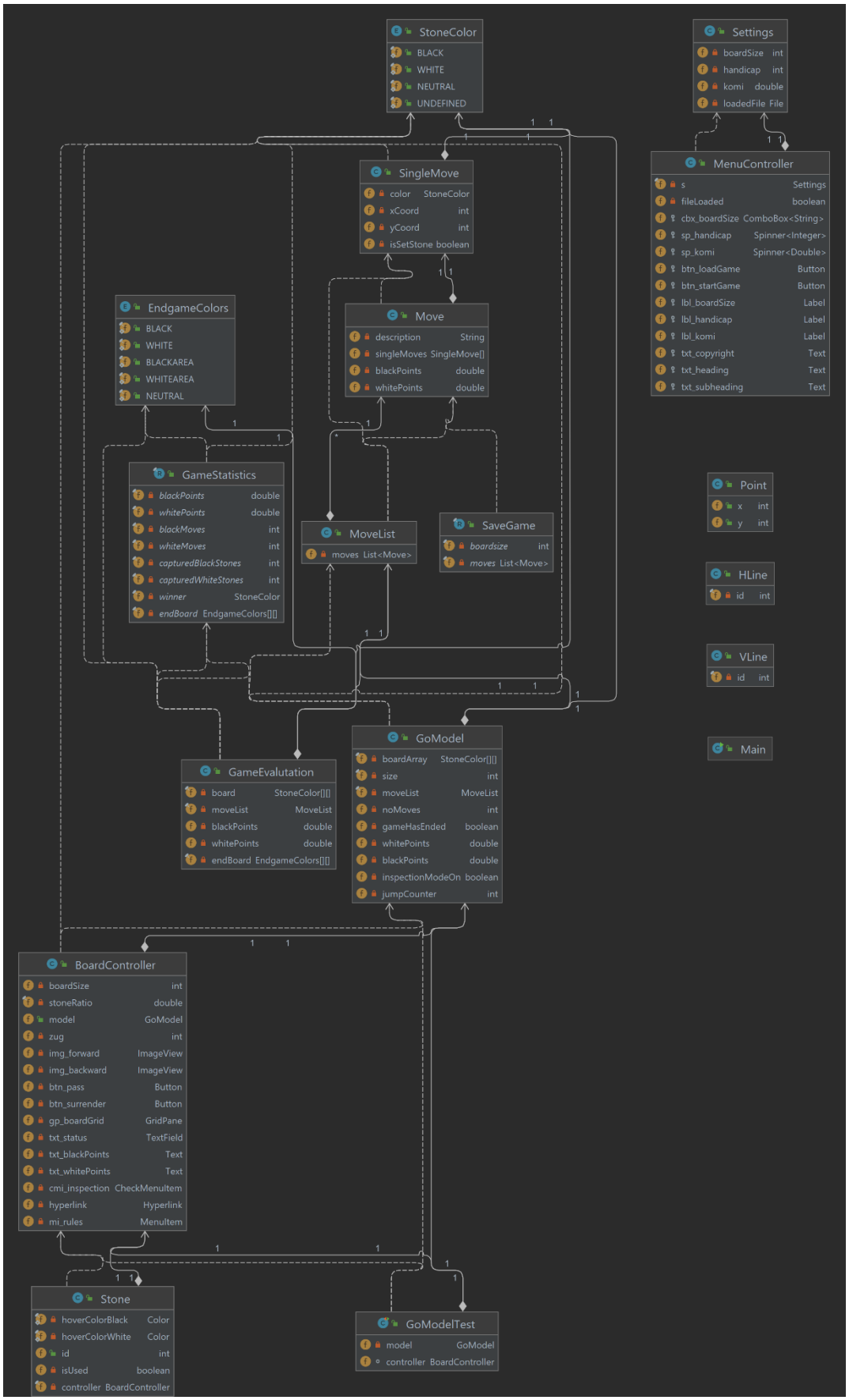



Abb. 2: UML-Klassendiagramm

# 4. Der Einstiegspunkt: Main

Der Einstiegspunkt unserer GO-Spielanwendung befindet sich im Package GoApplication in der Klasse "Main", die die JavaFX-Klasse "Application" erweitert. Diese Klasse dient als Startpunkt für die Ausführung der JavaFX-Anwendung und enthält die Implementierung der "start" und "main" Methoden.

## 4.1. Die start() Methode

Die "start()" Methode wird aufgerufen, wenn die JavaFX-Anwendung gestartet wird. Sie nimmt ein "Stage"-Objekt als Parameter, das das Hauptfenster der Anwendung darstellt. In der "start()" Methode werden die wichtigsten Schritte für den Start der Anwendung durchgeführt:

Zunächst wird das FXML-Layout definiert, das die Benutzeroberfläche der Anwendung beschreibt. Dies geschieht mithilfe eines "FXMLLoader"-Objekts, das die entsprechende FXML-Datei aus dem Ressourcenordner ("view/Menu.fxml") lädt.

Anschließend wird eine neue "Scene" erstellt und das zuvor geladene FXML-Layout als Inhalt dieser Szene festgelegt.

Das BootstrapFX-Stylesheet wird der Szene hinzugefügt, um das Layout der Anwendung mit Bootstrap-Stilen zu gestalten.

Die erstellte Szene wird dann auf der Hauptbühne (primary stage) des JavaFX-Fensters gesetzt.

Ein Icon für das Anwendungsfenster ("go_logo.png") wird aus den Ressourcen geladen und als Symbol für die Anwendung festgelegt.

Schließlich wird die Hauptbühne angezeigt, und die Benutzeroberfläche der GO-Spielanwendung wird dem Benutzer präsentiert.

## 4.2. Die main() Methode

Die "main()" Methode ist die klassische Einstiegsmethode für eine Java-Anwendung. Hier wird lediglich die "launch()" Methode von JavaFX aufgerufen, die den JavaFX Application-Lifecycle initialisiert und die "start()" Methode aufruft, um die Anwendung zu starten.

Die Kombination dieser beiden Methoden ermöglicht den Start der GO-Spielanwendung und stellt sicher, dass die definierte Benutzeroberfläche (FXML-Layout) in einem JavaFX-Fenster angezeigt wird. Durch den Einsatz von BootstrapFX wird zudem ein ansprechendes Design für die Anwendung bereitgestellt.

# 5. View – Komponenten

## 5.1. Menu

Die Menu-View ist eine FXML-Datei, die die Benutzeroberfläche für die Spieleinstellungen und den Start des GO-Spiels definiert.

Die View ist ein GridPane-Container mit zwei Spalten und drei Zeilen, der die verschiedenen UI-Elemente anordnet. Es gibt zwei VBox-Container, die in den Zellen des GridPane platziert sind, um die Einstellungen und den "Spiel starten"-Button zu gruppieren. Die View enthält Labels, ComboBoxen und Spinner für die Einstellungen wie Spielfeldgröße, Handicap und Komi.
Zudem gibt es einen "Spiel starten"-Button und einen "Spiel laden..."-Button. Die View enthält auch zwei Text-Elemente für den Anwendungstitel und den Copyright-Hinweis. Der Controller ist durch die "fx:controller" Attribut in der FXML-Datei ("fx:controller="controller.MenuController") definiert. Der Controller ist dafür verantwortlich, die Logik hinter den Interaktionen der Benutzeroberfläche zu verwalten. Er enthält Methoden, die auf die Aktionen der Buttons reagieren, z.B. "startGame()" und "onLoad()". Der Controller ermöglicht die Verbindung zwischen der View (FXML-Datei) und dem Model (Java-Code) und steuert, was bei Benutzerinteraktionen passieren soll. Eine genaue Beschreibung des Controllers wird allerdings auch in den nachfolgenden Kapiteln vorhanden sein.

Zusammenfassend bietet die Menu-View die Möglichkeit, Spieleinstellungen festzulegen (Spielfeldgröße, Handicap, Komi) und das GO-Spiel entweder zu starten oder ein gespeichertes Spiel zu laden. Der zugehörige Controller "MenuController" verarbeitet diese Interaktionen und leitet die entsprechenden Aktionen weiter, um das GO-Spiel entsprechend zu initialisieren und zu laden.

### 5.2. Board

Die Board-View ist ebenfalls eine FXML-Datei, die die Benutzeroberfläche für das GO-Spielbrett und die Steuerungselemente enthält. Auch hier wird das MVC-Muster (Model-View-Controller) verwendet, wobei die View durch diese FXML-Datei repräsentiert wird.

Die FXML-Struktur der Board-View besteht aus einem GridPane-Container mit drei Spalten und vier Zeilen. Es gibt eine VBox, die die Anzeige der Punkte für Schwarz und Weiß gruppiert, und ein weiteres GridPane, das das GO-Spielbrett darstellt. Die View enthält auch einen Menübalken (MenuBar) mit verschiedenen Optionen wie "Neues Spiel", "Speichern", "Inspektionsmodus", "Beenden" sowie "Regeln" und "Über". Zwei ImageView-Elemente ermöglichen das Vor- und Zurückspringen im Spielverlauf.

Der zugehörige Controller ist durch das "fx:controller" Attribut in der FXML-Datei ("fx:controller="controller.BoardController") definiert. Der Controller verarbeitet die Logik hinter den Interaktionen der Benutzeroberfläche und enthält Methoden wie "newGame()", "onSave()", "onExit()", "openRules()", etc., die auf die Aktionen der Buttons auf dem Spielbrett reagieren. Er fungiert als Bindeglied zwischen der View (FXML-Datei) und dem Model (Java-Code) und steuert die entsprechenden Aktionen, um das GO-Spiel zu aktualisieren und zu steuern.

### 5.3. Rules

Die FXML-Struktur der Rules-View enthält ein GridPane-Element mit einer Spalte und zwei Zeilen. In der ersten Zeile befindet sich ein Text-Element mit dem Inhalt "Alle Regeln im Überblick:", das die Überschrift für den Hyperlink darstellt. In der zweiten Zeile befindet sich ein Hyperlink-Element mit dem Text "https://de.wikipedia.org/wiki/Go_(Spiel)", der auf die Wikipedia-Seite zum Spiel "Go" verweist.

Der zugehörige Controller für diese View ist durch das "fx:controller" Attribut in der FXML-Datei ("fx:controller="controller.BoardController") definiert. Der Controller enthält die Logik, um auf die Aktionen des Hyperlinks zu reagieren. Wenn der Benutzer auf den Hyperlink klickt, kann der Controller entsprechende Aktionen ausführen, z. B. die Wikipedia-Seite in einem Webbrowser öffnen, um weitere Informationen über das Spiel "Go" anzuzeigen.

### 5.4. About

Die dargestellte View mit dem GridPane und der VBox dient der Darstellung von Informationen über die Entwickler des Spiels "JKU GO". Die View zeigt in der Mitte eine Überschrift "JKU GO" in fetter Schrift an, um den Fokus auf den Namen des Spiels zu legen. Darunter befinden sich die Namen der drei Entwickler "Johannes Hacker", "Moritz Neuwirth" und "Julian Lumetsberger". Die View hat eine insgesamt fixe Größe von 400x600 Pixeln und ist darauf ausgelegt, die Informationen in der Mitte zentriert anzuzeigen. Es handelt sich um eine einfache statische Informationsansicht, die den Benutzern Aufschluss über die Entwickler des Spiels gibt und keine interaktiven Elemente enthält.

## 6. Controller – Komponenten

### 6.1. Menu-Controller

Der Controller "MenuController", ist für die Verwaltung der Benutzeroberfläche (View) des Menüs verantwortlich. Er implementiert das Interface "Initializable", was bedeutet, dass die Methode "initialize" aufgerufen wird, sobald die View geladen wird.

Der Controller enthält FXML-Variablen, die mit den entsprechenden GUI-Elementen der View verknüpft sind, wie z.B. ComboBox, Spinner, Label, Text und Buttons. Diese Variablen ermöglichen es dem Controller, auf die Benutzereingaben und Ereignisse in der View zuzugreifen und darauf zu reagieren.

Die Methode "initialize" wird beim Start der Anwendung aufgerufen und setzt die Parameter für die Spinner-Elemente (Handicap und Komi) und füllt die Optionen für das BoardSize-Dropdown (Spielfeldgröße) mit den Werten "19x19", "13x13" und "9x9". Standardmäßig wird "13x13" als ausgewählte Option gesetzt.

Der Controller verfügt auch über eine Methode "onLoad", die aufgerufen wird, wenn der Benutzer auf den "Spiel laden..."-Button klickt. Diese Methode öffnet einen FileChooser, der es dem Benutzer ermöglicht, eine .json-Datei auszuwählen, die ein gespeichertes Spiel enthält. Die ausgewählte Datei wird analysiert, um die Größe des Spielfelds zu ermitteln, und die entsprechenden Einstellungen werden im "Settings"-Objekt gespeichert. Anschließend wird die Methode "startGame" aufgerufen, um das Spiel zu laden.

Die Methode "startGame" wird auch aufgerufen, wenn der Benutzer auf den "Spiel starten"-Button klickt. Wenn zuvor kein Spiel geladen wurde, werden die ausgewählten Einstellungen für Spielfeldgröße, Handicap und Komi im "Settings"-Objekt gespeichert. Dann wird die "Board.fxml"-Datei geladen, die die Spielfeldansicht darstellt. Der Controller der Spielfeldansicht wird initialisiert

und das "Settings"-Objekt wird übergeben, um die Spielfeldeinstellungen zu übernehmen. Schließlich wird die Spielfeldansicht in einem neuen Fenster angezeigt.

Der "MenuController" dient somit als Schnittstelle zwischen der Benutzeroberfläche des Menüs und dem "Settings"-Objekt, das die Einstellungen für das Spielfeld speichert, und der Spielfeldansicht, um das Spiel zu starten und die entsprechenden Einstellungen zu übergeben.

### 6.2. Board-Controller

Der Controller, "BoardController", ist für die Verwaltung der eigentlichen Spiel-Benutzeroberfläche (View) des Go-Spiels verantwortlich. Dieser Controller ermöglicht es dem Benutzer, das Go-Spiel zu spielen und die Interaktionen mit dem Spielbrett zu steuern.

Der Controller enthält FXML-Variablen, die mit den entsprechenden GUI-Elementen der View verknüpft sind, wie z.B. ImageView, Button, GridPane, TextField, Text und MenuItem. Diese Variablen ermöglichen es dem Controller, auf Benutzereingaben und Ereignisse in der View zuzugreifen und darauf zu reagieren.

Folgende Funktionen und Aufgaben werden im BoardController erfüllt:

Spielsteine setzen: Der Controller verfügt über eine Methode "setStone", die aufgerufen wird, wenn der Benutzer auf ein Feld im Gitter klickt, um einen Spielstein zu setzen. Die Methode platziert einen Spielstein in der Farbe des aktuellen Spielers an der ausgewählten Position und aktualisiert das Spielfeld entsprechend.

Zugübersicht und Passen: Der Controller enthält Funktionen "jumpForward" und "jumpBackward", die es dem Benutzer ermöglichen, durch die Zugübersicht zu blättern. Der Benutzer kann auch auf den "Passen"-Button klicken, um seinen Zug zu beenden und zum nächsten Spieler zu wechseln.

Speichern und Laden: Der Controller bietet Funktionen "onSave" und "onSurrender" zum Speichern des aktuellen Spiels in einer JSON-Datei und zum Aufgeben des Spiels.

Menü-Funktionen: Der Controller enthält Methoden für die Menüpunkte "Inspektionsmodus", "Regeln anzeigen" und "Über anzeigen", um das Spiel zu konfigurieren und zusätzliche Informationen anzuzeigen.

Spielende und Auswertung: Wenn das Spiel endet, bewertet der Controller die Punkte und zeigt den Gewinner oder ein Unentschieden an.

Board- und GUI-Verwaltung: Der Controller erstellt und konfiguriert das Spielbrett basierend auf der im Menü ausgewählten Größe. Es verwaltet auch die Größenänderungen der GUI-Elemente und passt die Darstellung der Steine und Linien entsprechend an.

Datenübertragung: Der Controller interagiert mit dem GoModel, das die Spiellogik und die Daten des Spiels verwaltet. Er ruft Methoden im GoModel auf, um die Spielsteine zu setzen, das Spiel zu bewerten und die Inspektionsmodus-Optionen zu steuern.

Insgesamt dient der "BoardController" als Bindeglied zwischen der Benutzeroberfläche des Go-Spiels und dem GoModel, das die Spiellogik und -daten enthält. Es ermöglicht dem Benutzer, das Go-Spiel zu spielen, Einstellungen vorzunehmen und das Spielbrett zu verwalten.

### 6.3. Weitere Komponenten für die Controller

#### 6.3.1. HLine

Die "HLine" ist eine spezielle Klasse in der Go-View, die eine horizontale Linie darstellt. Sie wird zusammen mit einem Spielstein in einem JavaFX-Group-Element platziert. Jede "HLine" hat eine eindeutige ID, die für Eventlistener der GUI-Größe verwendet wird, um die Position der Linie anzupassen. Die Klasse erbt von der JavaFX-Klasse "Line" und hat einen Konstruktor, der die ID und die Start- und Endposition der Linie festlegt. Eine Getter-Methode "getGoLineId()" gibt die ID der horizontalen Linie zurück. Insgesamt ermöglicht die "HLine"-Komponente das Zeichnen und Verwalten horizontaler Linien auf dem Go-Spielbrett in der Benutzeroberfläche.

#### 6.3.2. VLine

Die "VLine" ist eine spezielle Klasse in der Go-View, die eine vertikale Linie darstellt. Ähnlich wie die "HLine" wird auch die "VLine" zusammen mit einem Spielstein in einem JavaFX-Group-Element platziert. Jede "VLine" hat eine eindeutige ID, die für Eventlistener der GUI-Größe verwendet wird, um die Position der Linie entsprechend anzupassen. Die Klasse erbt ebenfalls von der JavaFX-Klasse "Line" und verfügt über einen Konstruktor, der die ID sowie die Start- und Endposition der vertikalen Linie festlegt. Die Getter-Methode "getGoLineId()" ermöglicht den Zugriff auf die ID der vertikalen Linie. Insgesamt ermöglicht die "VLine"-Komponente das Zeichnen und Verwalten vertikaler Linien auf dem Go-Spielbrett innerhalb der Benutzeroberfläche.

#### 6.3.3. Stone

Die "Stone" Klasse repräsentiert einen Spielstein in der Go-View. Sie ist eine spezielle JavaFX-Klasse, die von "Circle" erbt, um einen kreisförmigen Spielstein zu erstellen. Jeder Spielstein hat eine eindeutige ID, die verwendet wird, um seine Position auf dem Go-Spielbrett zu bestimmen. Die Klasse verfügt über eine Variable "isUsed", die angibt, ob der Spielstein bereits auf dem Spielbrett platziert wurde oder nicht.

Der Konstruktor der "Stone" Klasse initialisiert den Spielstein mit einem gegebenen Radius und einer transparenten Farbe. Er fügt auch einen "MouseClicked"-Listener hinzu, der aufgerufen wird, wenn der Spielstein angeklickt wird. Wenn der Spielstein noch nicht verwendet wurde und das Spiel noch nicht beendet ist, ruft der Listener die "setStone()" Methode des "BoardController" auf, um den Spielstein auf dem Spielbrett zu platzieren.

Darüber hinaus fügt die "Stone" Klasse einen "MouseEntered"-Listener hinzu, der aufgerufen wird, wenn der Mauszeiger über den Spielstein bewegt wird. Wenn der Spielstein noch nicht verwendet wurde und das Spiel noch nicht beendet ist, ändert der Listener die Farbe des Spielsteins je nachdem, welcher Spieler am Zug ist (schwarz oder weiß). Ein "MouseExited"-Listener wird hinzugefügt, der die Farbe des Spielsteins auf transparent zurücksetzt, wenn der Mauszeiger den Spielstein verlässt.

Die Klasse enthält auch eine Methode "setUsed()", mit der der Zustand des Spielsteins auf "verwendet" oder "nicht verwendet" gesetzt werden kann.

Die Stone-Klasse ermöglicht das Zeichnen und Verwalten von Spielsteinen auf dem Go-Spielbrett und interagiert mit dem "BoardController", um die Spielsteine gemäß den Spielregeln zu platzieren und deren Farbe entsprechend dem aktuellen Spieler zu ändern.

# 7. Model – Komponenten

## 7.1. Allgemein

Das "GoModel" enthält die Spiellogik, den Spielzustand und Informationen über den aktuellen Spielverlauf.

Die Klasse enthält eine Variable namens "boardArray", ein 2D-Array des Typen "StoneColor", dass das Go-Spielbrett repräsentiert. Es gibt auch eine Variable "size", die die Größe des Go-Spielbretts angibt (z. B. 9x9, 13x13 oder 19x19).

Das "GoModel" verfügt über eine "MoveList", die alle bisherigen Züge im aktuellen Spiel speichert. Die Variable "noMoves" zählt die Anzahl der bisherigen Züge im Spiel, und "gameHasEnded" gibt an, ob das Spiel beendet ist oder nicht.

Die Methoden in der Klasse umfassen Funktionen, um das Spiel zu steuern und den Spielzustand zu verwalten. Dazu gehören Methoden, um einen Stein auf dem Spielbrett zu platzieren, das Spiel zu speichern und zu laden, den aktuellen Spielstatus anzuzeigen, das Endergebnis des Spiels zu ermitteln und den Inspektionsmodus zu aktivieren oder zu deaktivieren.

Das "GoModel" enthält auch Methoden, um Handicap-Steine zu setzen, um das Spiel zu beginnen, sowie Methoden, um die Punkte und Farben von Steinen basierend auf ihrer ID abzurufen. Es gibt auch Funktionen, um Steine von bestimmten Positionen auf dem Brett zu entfernen.

Das "GoModel" ist der zentrale Bestandteil der Go-Anwendung, da es die Spiellogik und -regeln implementiert und den aktuellen Spielzustand verwaltet.

## 7.2. Move

Die Klasse "Move" repräsentiert einen Zug im Go-Spiel und enthält verschiedene Informationen über den Zug sowie die damit verbundenen Punktzahlen. Ein Zug im Go-Spiel besteht aus einer optionalen Beschreibung des Zugs und einer Abfolge von einzelnen Zügen, die als "SingleMove" bezeichnet werden. Die einzelnen Züge werden in einem Array von "SingleMove"-Objekten gespeichert. Zusätzlich enthält jeder Zug die Punktzahlen für die schwarze und weiße Seite nach dem Zug. Die Klasse verfügt über mehrere Konstruktoren. Der Standardkonstruktor "Move()" wird verwendet, um ein leeres "Move"-Objekt zu erstellen. Ein weiterer Konstruktor "Move(SingleMove[] singleMoves, double blackPoints, double whitePoints)" erlaubt es, ein "Move"-Objekt mit einer Liste von "SingleMove"-Objekten sowie den Punktzahlen für Schwarz und Weiß zu erstellen. Es gibt auch einen dritten Konstruktor "Move(SingleMove[] singleMoves, String description, double blackPoints, double whitePoints)", der zusätzlich zur Liste der Züge eine Beschreibung für den Zug entgegennimmt. Um auf die Eigenschaften des "Move"-Objekts zuzugreifen, gibt es mehrere Getter-Methoden. "getDescription()" gibt die optionale Beschreibung

des Zugs zurück, "getSingleMoves()" liefert das Array der "SingleMove"-Objekte zurück und "getBlackPoints()" und "getWhitePoints()" geben die Punktzahlen für Schwarz und Weiß nach dem Zug zurück. Die Klasse verfügt auch über Setter-Methoden, die jedoch hauptsächlich vom Jackson Object Mapper verwendet werden und daher intern genutzt werden.
Zusätzlich gibt es die Methode "toString()", die eine Zeichenkette zurückgibt, die die Farben der einzelnen "SingleMove"-Objekte im Zug repräsentiert. Diese Methode dient möglicherweise der Debugging-Unterstützung oder der Anzeige von Informationen über den Zug.

### 7.3. Move-List

Die Klasse "MoveList" dient dazu, alle Züge des Spielverlaufs im Go-Spiel zu speichern und zu verwalten. Die Klasse enthält eine Liste von "Move"-Objekten, die jeden Zug im Spiel repräsentieren. Die Liste wird als LinkedList implementiert, um die Effizienz beim Hinzufügen neuer Züge zu erhöhen.

Es gibt verschiedene Methoden, um Züge der Liste hinzuzufügen:

- "addMove(SingleMove[] singleMoves, double blackPoints, double whitePoints)": Fügt einen neuen Zug ohne Beschreibung hinzu und speichert die Punktzahlen für Schwarz und Weiß nach dem Zug.
- "addMoveWithDescription(SingleMove[] singleMoves, String description, double blackPoints, double whitePoints)": Fügt einen neuen Zug mit einer Beschreibung hinzu und speichert die Punktzahlen für Schwarz und Weiß nach dem Zug.

Zum Exportieren der Züge in eine Datei wird die Methode "exportMoves(File saveFile, int size)" verwendet. Dabei wird ein Objekt "SaveGame" erstellt, das die Boardgröße und die Liste der Züge enthält. Die Daten werden dann mithilfe der Jackson-Bibliothek in JSON-Format umgewandelt und in die angegebene Datei geschrieben.

Die Methode "importMoves(File loadedFile)" ermöglicht das Importieren von Zügen aus einer zuvor gespeicherten Datei. Die Methode liest die Daten aus der JSON-Datei und erstellt ein entsprechendes "SaveGame"-Objekt, aus dem die Liste der Züge extrahiert wird.

### 7.4. Point

Die Klasse "Point" in dem Paket "model.modelComponents" repräsentiert die Koordinaten eines Steins auf dem Go-Spielbrett. Sie wird verwendet, um die Positionen von Steinen zu speichern und wird insbesondere in der Berechnung von Stein-Gruppen verwendet.

Die Klasse hat zwei öffentliche Integer-Variablen "x" und "y", die die x- und y-Koordinaten des Steins darstellen.

Der Konstruktor "Point(int x, int y)" erstellt ein "Point"-Objekt mit den übergebenen x- und y-Koordinaten.

Die Methode "toString()" gibt eine Zeichenkette zurück, die die x- und y-Koordinaten des Punkts im Format "x:y" enthält, gefolgt von einem Leerzeichen.

### 7.5. SaveGame

Die Klasse "SaveGame" im Paket "model.modelComponents" repräsentiert ein Spiel, das in einem JSON-Format gespeichert werden kann. Sie ist ein sogenanntes "record", das in Java 14 eingeführt wurde und als Datenspeicherungsklasse dient.

Das "SaveGame"-Record enthält zwei Felder:

1. "boardsize" (Typ: int): Dieses Feld speichert die Größe des Go-Spielbretts.

2. "moves" (Typ: List<Move>): Dieses Feld ist eine Liste von "Move"-Objekten, die die gesamte Abfolge der Züge im Spiel speichern.

Ein "record" ist in Java eine spezielle Klasse, die nur zum Speichern von Daten verwendet wird. Es wird von Java automatisch mit einem Konstruktor, Gettern für die Felder und anderen nützlichen Methoden generiert, um die Daten einfach zu verwalten. In diesem Fall werden die Gettermethoden für "boardsize" und "moves" automatisch generiert.

### 7.6. SingleMove

Die Klasse "SingleMove" im Paket "model.modelComponents" speichert die Informationen über das Setzen oder Entfernen eines einzelnen Steins in einem Zug.

Die Klasse enthält vier Felder:

1. "color" (Typ: StoneColor): Dieses Feld speichert die Farbe des Steins, der in diesem Zug gesetzt oder entfernt wurde.

2. "xCoord" (Typ: int): Dieses Feld speichert die x-Koordinate (Spalte) des Spielbretts, auf der der Stein platziert oder entfernt wurde.

3. "yCoord" (Typ: int): Dieses Feld speichert die y-Koordinate (Zeile) des Spielbretts, auf der der Stein platziert oder entfernt wurde.

4. "isSetStone" (Typ: boolean): Dieses Feld gibt an, ob der Stein in diesem Zug gesetzt oder entfernt wurde. Wenn "isSetStone" true ist, wurde der Stein gesetzt, andernfalls wurde er entfernt.

Die Klasse enthält auch einen Parameterlosen Konstruktor und einen Konstruktor mit Parametern, um die Werte der Felder zu setzen. Außerdem gibt es Getter-Methoden für jedes Feld, um auf die gespeicherten Werte zuzugreifen.

Es gibt auch Setter-Methoden, die für die Jackson Object Mapper (eine Bibliothek zum Lesen und Schreiben von JSON-Daten) erforderlich sind, um die Daten in ein JSON-Format zu konvertieren und daraus zu lesen.

Die Methode "toString()" gibt die Farbe des Steins als String zurück und wird wahrscheinlich für Debugging- oder Anzeigezwecke verwendet.

# 8. SingleComponents – Komponenten

SingleComponents stellen Komponenten dar, welche immer wieder im Model verwendet werden.

## 8.1. EndgameColors

Die Enumeration "EndgameColors" im Paket "singleComponents" repräsentiert die verschiedenen möglichen Farben oder Zustände, die im Endspiel eines Go-Spiels auftreten können.

Die Enumeration enthält fünf Konstanten:

1. "BLACK": Diese Konstante repräsentiert schwarze Steine.

2. "WHITE": Diese Konstante repräsentiert weiße Steine.

3. "BLACKAREA": Diese Konstante repräsentiert Gebiete auf dem Spielbrett, die von schwarzen Steinen umschlossen sind.

4. "WHITEAREA": Diese Konstante repräsentiert Gebiete auf dem Spielbrett, die von weißen Steinen umschlossen sind.

5. "NEUTRAL": Diese Konstante repräsentiert neutrale Bereiche auf dem Spielbrett, die weder von schwarzen noch von weißen Steinen umschlossen sind.

Die Enumeration "EndgameColors" ermöglicht es, verschiedene Endspielzustände zu identifizieren und zu unterscheiden, was in einer Go-Anwendung nützlich sein kann, um beispielsweise das Punktesystem oder den Endspielgewinner zu berechnen. Die Verwendung von Enumerationen erleichtert den Umgang mit diesen verschiedenen Zuständen, da sie als Konstanten definiert und einfach verglichen werden können.

## 8.2. Settings

Die Klasse "Settings" im Paket "singleComponents" dient dazu, die Einstellungen (Settings) aus dem Menü in das Spielbrett zu speichern und zu übertragen.

Die Klasse hat folgende private Instanzvariablen:

1. "boardSize": Eine Ganzzahl, die die Größe des Spielbretts repräsentiert.

2. "handicap": Eine Ganzzahl, die die Anzahl der Handicap-Steine repräsentiert.

3. "komi": Eine Gleitkommazahl, die den Komi-Wert repräsentiert. Der Komi-Wert ist ein Punktevorteil, den Weiß zu Spielbeginn erhält.

4. "loadedFile": Ein Objekt vom Typ "File", das die ausgewählte Datei repräsentiert, die geladen werden soll.

Die Klasse "Settings" enthält entsprechende Getter- und Setter-Methoden für jede Instanzvariable, um den Zugriff auf die Werte zu ermöglichen. Die Getter-Methoden geben den Wert der entsprechenden Einstellung zurück, während die Setter-Methoden verwendet werden, um die Einstellungen zu setzen bzw. zu aktualisieren.

Die Klasse dient als Container, um die Einstellungen zu speichern und zwischen dem Menü und dem Spielbrett zu übertragen, um sicherzustellen, dass die richtigen Einstellungen auf das Spielbrett angewendet werden, wenn ein neues Spiel gestartet oder ein gespeichertes Spiel geladen wird.

### 8.3. StoneColor

Das Enum "StoneColor" im Paket "singleComponents" repräsentiert die verschiedenen Farben der Spielsteine im Spiel.

Es enthält die folgenden vier Konstanten:

1. "BLACK": Diese Konstante repräsentiert die Farbe Schwarz für einen Spielstein.

2. "WHITE": Diese Konstante repräsentiert die Farbe Weiß für einen Spielstein.

3. "NEUTRAL": Diese Konstante repräsentiert die neutrale Farbe, die verwendet wird, wenn kein Spielstein auf einem bestimmten Gitterpunkt platziert ist.

4. "UNDEFINED": Diese Konstante repräsentiert eine undefinierte Farbe, die in bestimmten Situationen verwendet wird, um einen Fehlerzustand oder eine nicht spezifizierte Farbe anzugeben.

Das Enum "StoneColor" dient dazu, die Farben der Spielsteine im Spiel zu definieren und zu verwalten. Es ermöglicht eine einfache Identifizierung und Unterscheidung der verschiedenen Farben während des Spiels und erleichtert die Implementierung von Spielregeln und Spielmechaniken, die auf den Farben der Spielsteine basieren.

# 9. Muster und Conventions

## 9.1. MVC-Pattern (Model-View-Controller)

Das MVC-Pattern (Model-View-Controller) ist ein Architekturmuster, das in der Softwareentwicklung häufig verwendet wird, um die Struktur und das Verhalten einer Anwendung zu organisieren. Es hilft, den Code übersichtlich, wartbar und skalierbar zu gestalten, indem es die Verantwortlichkeiten klar aufteilt. Das Pattern besteht aus drei Hauptkomponenten:

Model: Das Model repräsentiert die Daten und die Geschäftslogik der Anwendung. Es ist unabhängig von der Benutzeroberfläche und bietet Schnittstellen zum Zugriff und zur Manipulation der Daten.

View: Die View ist für die Darstellung der Daten und die Benutzeroberfläche verantwortlich. Sie empfängt Informationen vom Model und zeigt sie dem Benutzer an. Die View sollte keine direkte Interaktion mit dem Model haben, sondern nur Daten darstellen, die ihr vom Controller bereitgestellt werden.

Controller: Der Controller handhabt die Benutzereingaben und reagiert darauf. Er verarbeitet die Interaktionen des Benutzers mit der View und aktualisiert entsprechend das Model. Der Controller ist die Vermittlungsstelle zwischen Model und View.

Das MVC-Pattern ermöglicht eine klare Trennung von Daten, Darstellung und Benutzerinteraktionen, was die Wartung und Erweiterung der Anwendung erleichtert. Änderungen an einem Bereich haben keine Auswirkungen auf die anderen, solange die Schnittstellen zwischen den Komponenten gleichbleiben.

## 9.2. Umsetzung des MVC-Patterns in unserem Go-Spiel mit JavaFX

In unserem Go-Spiel haben wir das MVC-Pattern verwendet, um die Struktur und den Ablauf der Anwendung zu organisieren.

### 9.2.1. Model

Die Model-Komponente ist durch die Klasse "GoModel" repräsentiert. Diese Klasse enthält die Datenstrukturen und die Logik des Spiels.
"GoModel" speichert den Zustand des Spielbretts, den Zugverlauf, die Anzahl der Züge, die Punkte der Spieler und andere relevante Informationen.
Es bietet Methoden zum Setzen von Steinen, Überprüfen von Zügen auf Gültigkeit, Bewertung des Spielendes und zur Handhabung von Inspektionsmodus und Spielzurückspulung.

### 9.2.2. View

Die View-Komponente wird in unserem Go-Spiel mit JavaFX umgesetzt. Wir haben verschiedene FXML-Dateien erstellt, um die Benutzeroberfläche zu definieren.
Jede FXML-Datei repräsentiert eine bestimmte Ansicht, wie das Spielfeld, das Hauptmenü, die Spielinformationen und die Spielsteine.
Die Views verwenden CSS für das Styling und Bindings, um sich automatisch zu aktualisieren, wenn sich der Zustand des Models ändert.

### 9.2.3. Controller

Der Controller ist für die Verwaltung der Benutzerinteraktionen und die Aktualisierung des Models und der Views verantwortlich.
Wir haben verschiedene Controller-Klassen für jede Ansicht erstellt, um die Interaktionen zu behandeln und die entsprechenden Aktionen auf dem Model auszuführen.
Die Controller verwenden das Model, um die Spiellogik zu steuern, und aktualisieren die Views, um Änderungen anzuzeigen.

## 10. Ablauf des Spiels

Wenn der Benutzer eine Aktion ausführt, wie das Setzen eines Steins oder das Laden eines gespeicherten Spiels, interagiert der Controller mit dem Model, um die entsprechenden Operationen durchzuführen. Das Model überprüft die Gültigkeit des Zuges, aktualisiert den Zustand des Spielbretts und berechnet die Punkte der Spieler. Das Model benachrichtigt den Controller über Änderungen, und der Controller aktualisiert die Views, um die neuesten Informationen anzuzeigen. Die Views verwenden Bindings, um sich automatisch zu aktualisieren, wenn sich der Zustand des Models ändert, wodurch eine reibungslose Aktualisierung der Benutzeroberfläche ermöglicht wird.

Die Verwendung des MVC-Patterns hat uns geholfen, unser Go-Spiel strukturiert und gut organisiert zu halten. Es hat die Zusammenarbeit im Team erleichtert, da wir klare Verantwortlichkeiten hatten und unabhängig an verschiedenen Komponenten arbeiten konnten. Durch diese klare Trennung von Model, View und Controller war es auch einfacher, Änderungen und Erweiterungen an der Anwendung vorzunehmen, ohne dass andere Teile beeinträchtigt wurden. Insgesamt hat die Verwendung des MVC-Patterns dazu beigetragen, dass unser Go-Spiel eine konsistente und benutzerfreundliche Benutzeroberfläche und ein reibungsloses Spielerlebnis bietet.

# 11. Abbildungsverzeichnis