



# Programmierpraktikum

## "Mops Royal"

Im Wintersemester 2025/26



Eingereicht von:

Lars Gaethke

Fachrichtung: Wirtschaftsinformatik

Matrikelnummer: 106045

Fachsemester: 7

Referent:

Dipl.-Ing. (FH) Gerit Kaleck

Fachhochschule Wedel

Feldstraße 143, 22880 Wedel

E-Mail: gerit.kaleck@fh-wedel.de

# Inhaltsverzeichnis

<b>1 BENUTZERHANDBUCH .....</b>	<b>1</b>
1.1 ABLAUFBEDINGUNGEN .....	1
1.2 PROGRAMMINSTALLATION/PROGRAMMSTART.....	1
1.3 BEDIENUNGSANLEITUNG.....	2
1.3.1 <i>Idee und Ziel des Spiels</i> .....	2
1.3.2 <i>Spielvorbereitung</i> .....	2
1.3.3 <i>Spielablauf</i> .....	3
1.3.4 <i>Platzierungsregeln</i> .....	4
1.3.5 <i>Kombinationen und Wertungen</i> .....	5
1.3.5.1     Gültige Kombinationen.....	5
1.3.5.2     Wertung durchführen .....	5
1.3.5.3     Punkteberechnung .....	8
1.3.6 <i>Benutzeroberfläche/Bedienung</i> .....	9
1.3.7 <i>Spielende und Gewinnermittlung</i> .....	11
1.3.8 <i>Speichern und Laden</i> .....	12
1.3.8.1     Spielstand speichern.....	12
1.3.8.2     Beispieldatei eines Spielstandes .....	13
1.3.8.3     Aufbau der Plättchen-Codes .....	14
1.3.8.4     Spielstand laden .....	14
1.4 FEHLERMELDUNGEN .....	15
1.4.1 <i>Programmstart</i> .....	15
1.4.2 <i>Spieleinrichtung</i> .....	15
1.4.3 <i>Spielablauf</i> .....	16
1.4.3.1     Spiel-Events .....	16
1.4.4 <i>Spielstand speichern</i> .....	16
1.4.5 <i>Spielstand laden</i> .....	17
1.4.6 <i>Ressourcen und Dateien</i> .....	18
<b>2 PROGRAMMIERHANDBUCH .....</b>	<b>19</b>
2.1 ENTWICKLUNGSKONFIGURATION .....	19
2.2 PROBLEMANALYSE UND REALISATION .....	20
2.2.1 <i>Kombinationserkennung</i> .....	20
2.2.2 <i>Board-Verwaltung mit HashMap</i> .....	26
2.2.3 <i>Board-Rendering mit Origin-Offset</i> .....	30
2.2.4 <i>GUI-Logik-Trennung</i> .....	33
2.2.5 <i>Spielablauf-Steuerung</i> .....	36
2.2.6 <i>Speichern/Laden mit JSON</i> .....	39
2.3 VERWENDETE FORTGESCHRITTENE JAVA-KONZEpte.....	43
2.3.1 <i>Lambda-Ausdrücke und funktionale Interfaces</i> .....	43
2.3.2 <i>Streams API</i> .....	43
2.3.3 <i>PauseTransition für verzögerte Aktionen</i> .....	44
2.4 ALGORITHMEN.....	45
2.5 PROGRAMMORGANISATIONSPLAN.....	46
2.6 DATEIEN .....	48
2.6.1 <i>Spielstandsdateien (JSON)</i> .....	48
2.7 PROGRAMMTESTS.....	49
2.7.1 <i>Spielstart</i> .....	49
<b>ANHANG I: KOMBINATIONSÜBERSICHT .....</b>	<b>53</b>

# 1 Benutzerhandbuch

## 1.1 Ablaufbedingungen

Mindestangaben zur Nutzung der Software:

Komponente	Version
Bildschirm	Mindestens 1450 × 710 Pixel
Java Runtime Environment	Java 21 oder höher
Java JDK	JDK 21 (LTS)
Windows	Windows 10 oder höher (64-Bit)
JavaFX	21.0.3 (ist in der Anwendung enthalten)
Arbeitsspeicher (RAM)	Mindestens 4 GB empfohlen
Maus	Erforderlich für Spielsteuerung

Tabelle 1.1: Technische Voraussetzungen

## 1.2 Programminstallation/Programmstart

Um das Programm ausführen zu können, müssen die in Kapitel 1.1 beschriebenen Systemanforderungen erfüllt sein.

Das Programm kann unter Windows und macOS durch einen Doppelklick auf die Datei PP\_MopsRoyal\_Gaethke.jar gestartet werden, sofern eine geeignete Java-Laufzeitumgebung installiert ist.

Unter Windows besteht zusätzlich die Möglichkeit, das Programm über einen Doppelklick auf die Startdatei startMopsRoyal.bat auszuführen. Diese führt beim Start automatische Prüfungen durch und gibt bei auftretenden Fehlern entsprechende Hinweise im Kommandozeilenfenster aus (vgl. Kapitel 1.4.1).

Alternativ kann das Programm unter Windows, macOS und Linux über die Konsole gestartet werden. Dazu ist zunächst in das Verzeichnis ppws\_21 zu wechseln. Anschließend kann das Programm mit folgendem Befehl ausgeführt werden:

```
java -jar PP_MopsRoyal_Gaethke.jar
```

Falls es beim Starten des Programms zu Fehlern kommen sollte, kann der Abschnitt Fehlermeldungen zur Identifikation und Behebung des Fehlers genutzt werden (siehe Kapitel 1.4).

## 1.3 Bedienungsanleitung

### 1.3.1 Idee und Ziel des Spiels

Bei dem Spiel "Mops Royal" handelt es sich um ein strategisches Legespiel für 2-4 Spieler. Jeder Spieler verfügt über ein eigenes 5×5 Spielbrett und greift auf einen gemeinsamen Vorrat an Plättchen zu.

Ziel des Spiels ist es, durch geschicktes Platzieren von Plättchen auf dem eigenen Spielbrett und strategisches Bilden von Kombinationen möglichst viele Punkte zu erzielen. Kombinationen aus gleichfarbigen oder gleichartigen Plättchen können gewertet werden, wobei größere Kombinationen mehr Punkte bringen.

Der Spieler mit den meisten Punkten gewinnt, sobald alle Spieler ihr 5×5 Spielbrett vollständig befüllt haben.

### 1.3.2 Spielvorbereitung

Zu Beginn eines neuen Spiels legt der Benutzer über einen Konfigurationsdialog die Rahmenbedingungen fest:

#### **Spieleranzahl und Namen**

- Auswahl der Spieler (2-4 Spieler)
- Eingabe der Spielernamen (alle Namen müssen unterschiedlich und dürfen nicht leer sein)

#### **Automatische Spielinitialisierung**

Nach Bestätigung der Konfiguration richtet das System das Spiel automatisch ein. Jeder Spieler erhält:

- Ein eindeutiges Startplättchen, das zufällig zugelost und automatisch platziert wird
- Einen Zugriff auf den gemeinsamen Vorrat von 36 Plättchen
- Ein leeres 5×5 Spielbrett

Die Spielerreihenfolge richtet sich nach der Vergabe der Spielernamen und wird klar im Konfigurationsdialog gekennzeichnet.

### 1.3.3 Spielablauf

Die Spieler sind abwechselnd am Zug. Zu Beginn jeder Runde wird ein Plättchen aus dem gemeinsamen Vorrat gezogen. Dieses Plättchen ist allgemeingültig und wird für jeden Spieler in der Anzeige "Nächstes Plättchen" dargestellt.

#### Ablauf eines Spielzugs

##### 1. **Plättchen platzieren** [Spieleraktion]

Der Spieler wählt durch Mausklick eine gültige Position auf seinem 5×5 Spielbrett aus. Gültige Positionen werden farblich hervorgehoben (siehe Abb. 1.2). Das Plättchen wird nach dem Klick sofort platziert

##### 2. **Kombinationen prüfen** [automatisch]

Das System analysiert automatisch, ob nach der Platzierung wertbare Kombinationen vorhanden sind

##### 3. **Wertung einer Kombination (optional)** [Spieleraktion]

Existieren gültige Kombinationen, öffnet sich automatisch ein Wertungsfenster (siehe Abb. 1.3). Der Spieler kann nun:

- Eine Kombination auswählen und werten
- Die Wertung ablehnen

Bei einer Wertung wird ein Plättchen der gewerteten Kombination automatisch umgedreht.

##### 4. **Zugende** [automatisch]

Nach Abschluss oder Ablehnung der Wertung ist automatisch der nächste Spieler an der Reihe

### 1.3.4 Platzierungsregeln

Plättchen müssen nach folgenden Regeln platziert werden:

#### Angrenzung

- Jedes neue Plättchen muss orthogonal (horizontal oder vertikal) an mindestens ein bereits platziertes Plättchen angrenzen
- Alleiniges diagonales Angrenzen ist nicht möglich
- Es dürfen insgesamt nicht mehr als 5 Spielplättchen horizontal oder vertikal in einer Reihe liegen

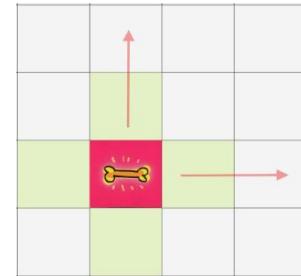


Abbildung 1.1: Valide Zugrichtungen

#### Spielfeldgröße

- Das Spielbrett ist auf eine maximale Größe von  $5 \times 5$  Feldern begrenzt
- Insgesamt können pro Spieler genau 25 Plättchen platziert werden (inklusive Startplättchen)
- Das Endergebnis ist ein vollständig ausgefülltes  $5 \times 5$  Quadrat

#### Gültigkeitsprüfung

- Das System markiert alle gültigen Platzierungspositionen farblich
- Nur auf hervorgehobenen Positionen kann das Plättchen platziert werden
- Ungültige Positionen sind nicht auswählbar

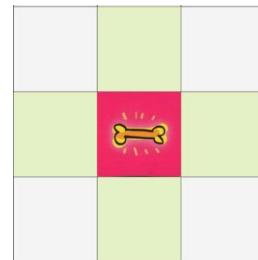


Abbildung 1.2: Gültige Platzierungspositionen

Da jeder Spieler mit einem Startplättchen beginnt und insgesamt 25 Felder zur Verfügung stehen, werden von den ursprünglich 36 Plättchen im Vorrat genau 24 weitere Plättchen im Spielverlauf platziert.

## 1.3.5 Kombinationen und Wertungen

### 1.3.5.1 Gültige Kombinationen

Kriterium	Anforderung
<b>Größe</b>	Eine Kombination besteht aus genau 3, 4 oder 5 Plättchen. Anordnungen mit weniger als 3 oder mehr als 5 Plättchen können nicht gewertet werden
<b>Merkmal</b>	Alle Plättchen der Kombination müssen entweder dieselbe Farbe oder denselben Gegenstand aufweisen. Mischformen sind ungültig
<b>Anordnung</b>	Die Plättchen müssen orthogonal oder Diagonal zusammenhängend sein (siehe Kapitel 1.3.4)
<b>Geometrische Form</b>	Die Kombination kann in beliebiger geometrischer Form vorliegen (gerade Linie, L-Form, T-Form, Kreuz, etc.). Gedrehte und gespiegelte Varianten derselben Form sind gültig (siehe Anhang I.1 für Kombinationsübersicht)
<b>Status der Plättchen</b>	Bereits umgedrehte Plättchen können nicht mehr Bestandteil neuer Kombinationen sein. Sie behalten ihre Position und blockieren Platzierungen

Tabelle 1.2: Gültigkeitskriterien für Kombinationen

### 1.3.5.2 Wertung durchführen

Nach der Platzierung eines Plättchens öffnet sich automatisch das Wertungsfenster, falls gültige Kombinationen erkannt wurden. Eine Kombination ist eine Anordnung mehrerer Plättchen, die bestimmte Kriterien (siehe 1.3.5.1) erfüllt.

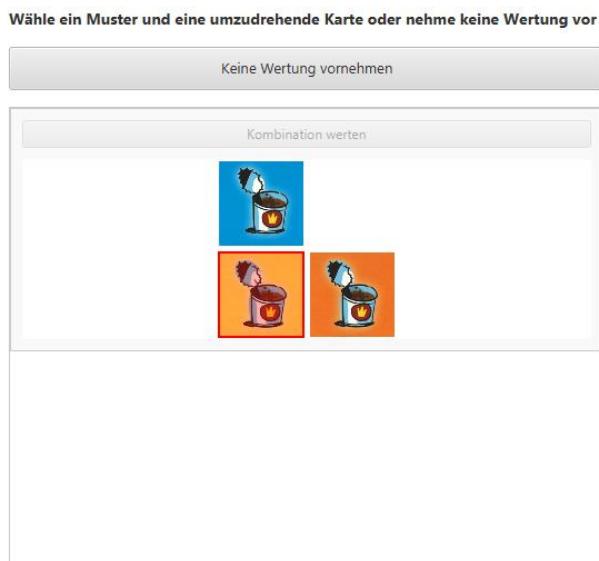


Abbildung 1.3 Wertungsfenster

## Kombinationsbereiche

- Jede erkannte Kombination wird in einem eigenen Bereich dargestellt
- Jeder Bereich enthält den Button "Kombination werten"
- Die Plättchen der Kombination werden visuell in ihrer Originalanordnung dargestellt
- Das Wertungsfenster markiert alle zur Drehung verfügbaren Plättchen rot (vgl. Abb. 1.4)

## Auswahlmechanismus

Im Zuge jeder durchgeführten Wertung muss der Spieler zwingend ein in der gewerteten Kombination mittig liegendes Plättchen umdrehen. Bei 4er-Kombinationen gibt es dafür zwei zulässige Plättchen (vgl. Abb. 1.4), aus denen der Spieler eins auswählt. Dieses Plättchen steht für zukünftige Wertungen nicht mehr zur Verfügung.



Abbildung 1.4: Zwei zur Drehung verfügbaren Plättchen

- In jedem Fall wählt der Spieler durch Mausklick das gewünschte Plättchen aus und bestätigt die Wertung
- Nach der Auswahl wird das Plättchen umgedreht und zeigt seine Rückseite (vgl. Abb. 1.5)

## Auswirkungen umgedrehter Plättchen

- Umgedrehte Plättchen bleiben physisch auf dem Spielbrett an ihrer Position
- Sie werden bei der automatischen Kombinationserkennung nicht mehr berücksichtigt
- Sie blockieren Platzierungen neuer Plättchen



Abbildung 1.5:  
Plättchendarstellung auf  
Spielfeld nach Wertung

### Beispiele für gültige Kombinationsmuster



Abbildung 1.6:  
Farbkombination aus 3  
Plättchen



Abbildung 1.7: Symbolkombination aus 4  
Plättchen



Abbildung 1.8: Symbolkombination  
aus 5 Plättchen

Eine Auflistung aller möglichen Kombinationen findet sich im Anhang I.1.

### 1.3.5.3 Punkteberechnung

Die Punktzahl einer Kombination setzt sich aus Basispunkten und optionalen Bonuspunkten zusammen.

#### Basispunkte nach Kombinationsgröße

Kombinationsgröße	Basispunkte
<b>3 Plättchen</b>	2 Punkte (vgl. Abb. 1.6)
<b>4 Plättchen</b>	4 Punkte (vgl. Abb. 1.7)
<b>5 Plättchen</b>	7 Punkte (vgl. Abb. 1.8)

Tabelle 1.3: Basispunkte

#### Bonuspunkte für Kronensymbole

- Zeigt ein Plättchen der gewerteten Kombination das Kronensymbol, erhält der Spieler einen zusätzlichen Bonuspunkt
- Hierbei kann pro Kombination nur ein Kronensymbol integriert werden
- Bonuspunkte werden zu den Basispunkten hinzugerechnet



Abbildung 1.9:  
Plättchen mit  
Kronensymbol

### 1.3.6 Benutzeroberfläche/Bedienung

Die Bedienung von "Mops Royal" erfolgt über die Menüleiste und Mausinteraktionen. Das Spiel führt den Benutzer durch einen automatisierten Spielfluss, bei dem nach jeder Aktion die nächsten verfügbaren Optionen angeboten werden.

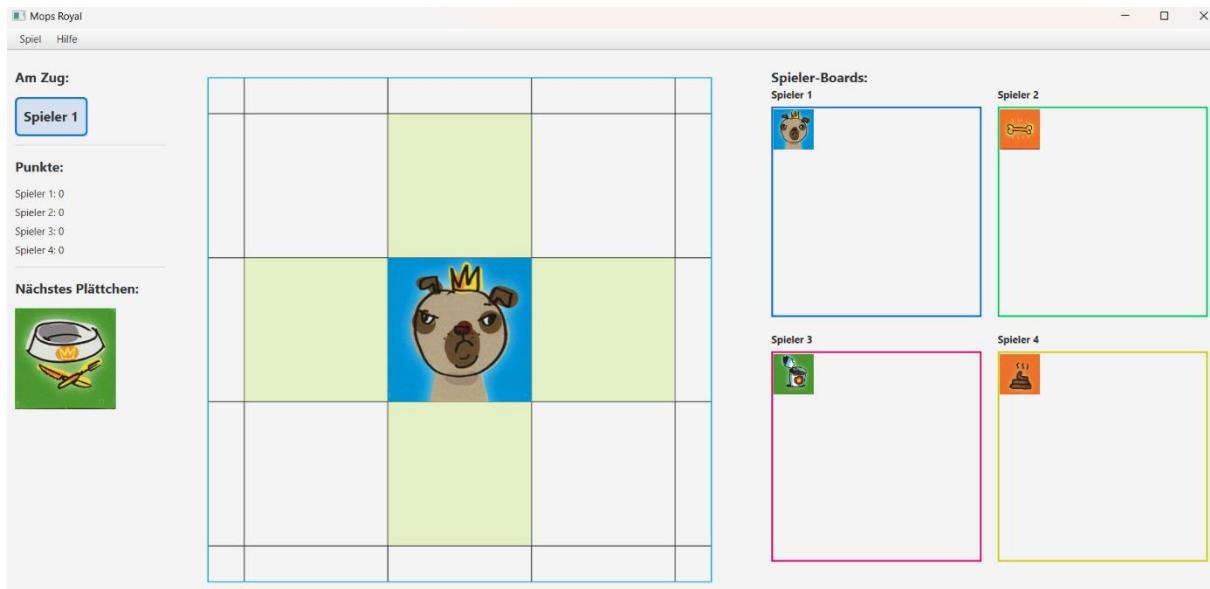


Abbildung 1.10: Benutzeroberfläche im initialen Spielzustand

#### Menüleiste

Die Menüleiste am Fensterrand links oben bietet folgende Funktionen:

Menüpunkt	Unterreiter	Funktion
Spiel	Neues Spiel starten	Öffnet einen Dialog zur Konfiguration einer neuen Partie (Spieleranzahl und Namen)
	Spielstand laden	Lädt einen zuvor gespeicherten Spielstand aus einer JSON-Datei
	Spielstand speichern	Speichert den aktuellen Spielstand in einer JSON-Datei
	Spiel beenden	Beendet die Anwendung
Hilfe	Spieldaten	Zeigt eine Übersicht der Spielregeln und Bedienung
	Kombinationen	Zeigt alle gültigen Kombinationsmuster mit Punktwerten

Tabelle 1.4: Funktionsübersicht Menüleiste

## Maussteuerung

Die Steuerung des Spiels erfolgt ausschließlich über die Maus:

Aktion	Funktion
<b>Mausrad</b>	Vergrößert oder verkleinert die Ansicht des aktiven Spielbretts (Zoom)
<b>Linke Maustaste halten + Ziehen</b>	Verschiebt den sichtbaren Ausschnitt des Spielbretts
<b>Klick auf hervorgehobene Position</b>	Platziert das aktuelle Plättchen auf dem eigenen Spielbrett. Bei vorhandenen Kombinationen öffnet sich anschließend automatisch das Wertungsfenster
<b>Klick auf Rot markiertes Plättchen</b>	Wählt im Wertungsfenster das Plättchen aus, das nach der Wertung umgedreht werden soll
<b>Klick auf Button "Kombination werten"</b>	Wertet die ausgewählte Kombination und dreht das gewählte Plättchen um
<b>Klick auf Button "keine Wertung vornehmen"</b>	Überspringt die Wertung und beendet den aktuellen Zug

Tabelle 1.5: Funktionsübersicht Maussteuerung

### 1.3.7 Spielende und Gewinnermittlung

Das Spiel endet automatisch, wenn alle Spieler ihr  $5 \times 5$  Spielbrett vollständig befüllt haben (jeweils 25 Plättchen platziert).

#### Gewinnermittlung

Die Platzierung der Spieler wird nach folgenden Kriterien bestimmt:

1. **Höchste Gesamtpunktzahl:** Der Spieler mit den meisten Punkten gewinnt das Spiel
2. **Tie-Breaker bei Punktgleichstand:** Haben mehrere Spieler die gleiche Punktzahl, entscheidet die Anzahl umgedrehter Plättchen
  - Der Spieler mit weniger umgedrehten Plättchen erhält den besseren Platz
  - Dies belohnt effizientere Wertungsstrategien
3. **Geteilter Sieg:** Besteht auch nach Anwendung des Tie-Breakers Gleichstand, wird dies als geteilter Sieg gewertet

#### Ergebnisanzeige

Nach Spielende öffnet sich automatisch ein Ergebnisdialog, der alle Spieler in einer Rangliste nach Platzierung sortiert anzeigt. Für jeden Spieler werden dargestellt:

- Platzierung
- Spielername
- Spielerfarbe
- Endpunktzahl
- Anzahl umgedrehter Plättchen

Der Gewinner wird visuell hervorgehoben (vgl. Abb. 1.11).



Abbildung 1.11: Automatisch erzeugter Gewinnerdialog nach Spielende

## 1.3.8 Speichern und Laden

### 1.3.8.1 Spielstand speichern

Über "Datei → Spielstand speichern" wird ein Dateidialog geöffnet. Der vorgeschlagene Dateiname folgt dem Format MopsRoyal\_YYYYMMDD\_HHMMSS.json. Nach Bestätigung wird der komplette Spielzustand im JSON-Format gespeichert.

Beim Speichern werden folgende Angaben im Spielstand abgelegt:

#### **Spieler**

Für jeden Spieler werden gespeichert:

- Reihenfolge im Spiel
- Name
- Punktestand
- Startplättchen
- Bereits gelegte Plättchen (5×5 - Matrix)

#### **Aktueller Spieler**

Welcher Spieler gerade am Zug ist.

#### **Nächstes Plättchen**

Welches Plättchen als nächstes gelegt wird.

### 1.3.8.2 Beispieldatei eines Spielstandes

```
{  
    "players": [  
        {  
            "nr": 0,  
            "name": "Alice",  
            "initial": 110,  
            "points": 12,  
            "cards": [  
                [110,120,130,210,220],  
                [230,310,320,330,410],  
                [420,430,510,520,530],  
                [610,620,630,111,221],  
                [331,441,551,990,990]  
            ]  
        },  
        {  
            "nr": 1,  
            "name": "Bob",  
            "initial": 220,  
            "points": 18,  
            "cards": [  
                [220,230,240,250,260],  
                [310,320,330,340,350],  
                [410,420,430,440,450],  
                [510,520,530,540,550],  
                [610,620,630,640,650]  
            ]  
        }  
    "turn": 0,  
    "nextCard": 33  
}
```

Abbildung 1.12: Beispiel Spielstands-Datei mit 2 Spielern  
Der Inhalt wird automatisch erzeugt

### 1.3.8.3 Aufbau der Plättchen-Codes

Alle Plättchenangaben im Spielstand werden als dreistellige Zahl nach folgendem Muster kodiert:

1. Stelle: Farbe (1=Blau, 2=Grün, 3=Orange, 4=Pink, 5=Lila, 6=Gelb)
2. Stelle: Objekt (1=Kissen, 2=Knochen, 3=Napf, 4=Dose, 5= Kot, 6=Mops)
3. Stelle: Umgedreht (0=nicht umgedreht, 1=umgedreht)

#### Beispiele:

- 610 = Gelbes Kissen, nicht umgedreht
- 331 = Orangener Napf, umgedreht
- 990 = Leere Position

#### Sonderfall nextCard:

Hier entfällt die 3. Stelle, da das nächste Plättchen niemals umgedreht sein kann (z.B. 33 = Orangener Napf).

### 1.3.8.4 Spielstand laden

Über "Datei → Spielstand laden" kann ein gespeicherter Spielstand wiederhergestellt werden. Nach Auswahl einer gültigen JSON-Datei wird der Spielzustand vollständig wiederhergestellt.

Bei Ladefehlern (ungültige Datei, beschädigte Daten) wird eine Fehlermeldung angezeigt und der vorherige Zustand bleibt erhalten (siehe Kapitel 1.4.5).

## 1.4 Fehlermeldungen

Dieses Kapitel listet alle Fehlermeldungen auf, die während der Nutzung von Mops Royal auftreten können. Jede Fehlermeldung wird mit ihrer Ursache und der entsprechenden Lösungsmöglichkeit beschrieben.

### 1.4.1 Programmstart

Fehlermeldungen, die beim Programmstart auftreten können.

Fehlermeldung	Ursache	Lösung
"java" is not recognized as an internal or external command	Java 21 JDK nicht installiert	→ <a href="#">Java 21 installieren</a>
"BUILD FAILURE"	Maven-Build fehlgeschlagen (z.B. Kompilierungsfehler)	→ Maven Lifecycle ausführen
"FileNotFoundException: PP_MopsRoyal_Gaethke.jar"	JAR-Datei nicht in target/ vorhanden	→ Projekt neu bauen
javafx.fxml.LoadException / IllegalStateException: Location is not set	Ordnerstruktur ist falsch oder Dateien an falscher Stelle	→ Datei verschieben

Tabelle 1.6: Fehlermeldungen beim Programmstart

### 1.4.2 Spieleinrichtung

Fehlermeldungen, die beim Einrichten eines neuen Spiels auftreten können.

Fehlermeldung	Ursache	Lösung
"Spieler [Spielernummer]: Name darf nicht leer sein."	Ein teilnehmender Spieler hat keinen Namen eingegeben	Für jeden aktiven Spieler einen Namen eingeben
"Namen müssen eindeutig sein: '[Spielername]'"	Zwei Spieler haben denselben Namen (Groß-/Kleinschreibung wird ignoriert)	Jedem Spieler einen unterschiedlichen Namen geben
"Dialog konnte nicht geladen werden: [IOException Message]"	Die FXML-Datei konnte nicht geladen werden	Programm neu installieren

Tabelle 1.7: Fehlermeldungen beim Einrichten eines neuen Spiels

### 1.4.3 Spielablauf

Fehlermeldungen, die während des laufenden Spiels auftreten.

Fehlermeldung	Ursache	Lösung
"Ungültige Position! Bitte wähle eine markierte Position."	Der Spieler hat auf eine nicht markierte Position geklickt	Nur auf die grün markierten Positionen klicken
"Fehler beim Laden der Wertungsansicht: [Fehlermeldung]"	Die Wertungsansicht (Scoring Overlay) konnte nicht geladen werden	Programm neu starten. Falls Problem bestehen bleibt, Programm neu installieren

Tabelle 1.8: Fehlermeldungen während des laufenden Spiels

#### 1.4.3.1 Spiel-Events

Meldungen über besondere Spielereignisse.

Info-Meldung	Bedeutung
"[Spielername] muss aussetzen! (Eigene Startkarte gezogen)"	Der Spieler hat seine eigene Startkarte gezogen und muss eine Runde aussetzen
"Nachholrunde: [Spielername] ist dran!"	Eine Nachholrunde wurde gestartet, weil ein Spieler ausgesetzt hat
"[Spielername] muss aussetzen! (Spielfeld ist voll - 25 Plättchen gelegt)"	Das Spielfeld des Spielers ist vollständig gefüllt, er kann keine weiteren Plättchen legen

Tabelle 1.9: Meldungen über besondere Spielereignisse

### 1.4.4 Spielstand speichern

Fehlermeldungen beim Speichern des Spielstands.

Fehlermeldung	Ursache	Lösung
"Speichern fehlgeschlagen: [Fehlermeldung]"	Allgemeiner Schreibfehler (z.B. Festplatte voll, Datei schreibgeschützt)	Speicherplatz prüfen, Dateirechte prüfen, anderen Speicherort wählen

Tabelle 1.10: Fehlermeldungen beim Speichern des Spielstands

## 1.4.5 Spielstand laden

Fehlermeldungen beim Laden einer Spielstandsdatei.

Fehlermeldung	Ursache	Lösung
"Ungültige Spielstandsdatei: Ungültiges JSON-Format: [Details]"	Die JSON-Datei ist nicht korrekt formatiert oder beschädigt	Eine andere, unbeschädigte Spielstandsdatei verwenden
"Ungültige Spielstandsdatei: JSON-Datei ist leer oder ungültig"	Die Datei enthält keine Daten oder nur leere/null-Werte	Eine gültige Spielstandsdatei auswählen
"Ungültige Spielstandsdatei: Keine Spieler in Spielstandsdatei"	Das players-Array fehlt oder ist leer	Spielstand ist korrupt. Eine andere Datei verwenden
"Ungültige Spielstandsdatei: Ungültige Spieleranzahl: [Anzahl] (erwartet: 2-4)"	Die Datei enthält weniger als 2 oder mehr als 4 Spieler	Nur Spielstände mit 2-4 Spielern sind gültig. Neue Spielstandsdatei verwenden
"Ungültige Spielstandsdatei: Ungültiger turn: [Wert] (erwartet: 0-[Spieleranzahl])"	Der turn-Wert liegt außerhalb des gültigen Bereichs	Spielstand ist fehlerhaft. Eine korrekte Datei verwenden
"Ungültige Spielstandsdatei: Ungültiger nextCard Code: [Code]"	Der Code für das nächste Plättchen ist ungültig (Format: 11-66 oder 99)	Spielstand ist fehlerhaft. Korrekten Spielstand verwenden
"Ungültige Spielstandsdatei: Spieler [Nummer] ist null"	Ein Spieler-Objekt im Array ist null	Spielstandsdatei ist beschädigt. Eine andere Datei verwenden
"Ungültige Spielstandsdatei: Spieler [Nummer] hat falsche number: [Wert]"	Die Spielernummer stimmt nicht mit der Array-Position überein	Spielstand ist inkonsistent. Neue Datei verwenden
"Ungültige Spielstandsdatei: Spieler [Nummer] hat leeren Namen"	Ein Spieler hat keinen Namen oder nur Leerzeichen	In der JSON-Datei muss jeder Spieler einen gültigen Namen haben
"Ungültige Spielstandsdatei: Spieler [Nummer] hat negativen Score: [Wert]"	Die Punktzahl eines Spielers ist negativ	Spielstand ist fehlerhaft. Korrekten Spielstand verwenden
"Ungültige Spielstandsdatei: Ungültiger initial Code: [Code]"	Der Code der Startkarte ist ungültig	Startkarten-Code muss Format 110-660 haben. Neue Datei verwenden
"Ungültige Spielstandsdatei: Spieler [Nummer] hat keine cards"	Das cards-Array eines Spielers fehlt	Spielstandsdatei ist unvollständig. Andere Datei verwenden

"Ungültige Spielstandsdatei: Spieler [Nummer] hat [Anzahl] Zeilen (erwartet: 5)"	Das Board eines Spielers hat nicht genau 5 Zeilen	Board muss 5x5 sein. Korrekte Spielstand verwenden
"Ungültige Spielstandsdatei: Spieler [Nummer] Zeile [Zeile] ist null"	Eine Zeile im Board-Array ist null	Spielstandsdatei ist beschädigt. Neue Datei verwenden
"Ungültige Spielstandsdatei: Spieler [Nummer] Zeile [Zeile] hat [Anzahl] Spalten (erwartet: 5)"	Eine Zeile im Board hat nicht genau 5 Spalten	Board muss 5x5 sein. Korrekte Spielstand verwenden
"Ungültige Spielstandsdatei: Spieler [Nummer] Position ([Zeile],[Spalte]): Ungültiger Code [Code]"	Ein Plättchen-Code an einer Position ist ungültig	Gültige Codes: 110-661 (3-stellig) oder 990 (leer). Neue Datei verwenden
"Unerwarteter Fehler: [Fehlermeldung]"	Ein unvorhergesehener Fehler ist beim Laden aufgetreten	Fehlermeldung notieren, Programm neu starten. Bei wiederholtem Auftreten Entwickler kontaktieren

Tabelle 1.11: Fehlermeldungen beim Laden einer Spielstandsdatei

#### 1.4.6 Ressourcen und Dateien

Diese Fehler werden teilweise nur in der Konsole ausgegeben und führen zu fehlenden Bildern in der Oberfläche.

Fehlermeldung (Konsole/Log)	Ursache	Lösung
"Fehler: Bilddatei nicht gefunden: [Pfad]"	Eine Plättchen-Bilddatei fehlt im resources-Ordner	Programm neu installieren, um alle Ressourcen-Dateien wiederherzustellen
"Fehler beim Laden des Tiles: [Fehlermeldung]"	Ein Plättchen-Bild konnte nicht geladen werden	Prüfen, ob alle Bilddateien im Verzeichnis resources/gui/images/ vorhanden sind
"Fehler beim Laden: [Pfad]"	Ein Bild in der Wertungsansicht konnte nicht geladen werden	Programm neu installieren

Tabelle 1.12: Fehlermeldungen bei fehlenden oder beschädigten Programmdateien

## 2 Programmierhandbuch

### 2.1 Entwicklungskonfiguration

Die folgende Tabelle enthält die Konfigurationen mit den das Programm entwickelt wurde:

Komponente	Version
Betriebssystem	Windows 10/11
Java Runtime Environment	21
Java Development Kit	21
IDE	IntelliJ IDEA 2023.3.4 (Ultimate Edition)
Build-Tool	Maven 3.8.x / 3.9.x

Tabelle 2.1: Tabellarische Darstellung der Entwicklungskonfiguration

Dabei wurden folgende externe Bibliotheken eingebunden und verwendet:

Bibliothek	Version	Verwendungszweck
JSON	2.10.1	JSON-Parser
JavaFX	21.0.3	Grafische Benutzeroberfläche
JUnit Jupiter	5.10.0	Testing

Tabelle 2.2: Tabellarische Darstellung der externen Bibliotheken

## 2.2 Problemanalyse und Realisation

Im Folgenden werden die Kernaspekte des Spiels „Mops Royal“ aufgearbeitet und mehrere Realisationsmöglichkeiten für diese dargestellt und diskutiert. Daraufhin wird die im Projekt gewählte Realisationsmöglichkeit einmal genauer beschrieben.

### 2.2.1 Kombinationserkennung

#### Problemanalyse

Die Kombinationserkennung ist das Herzstück von Mops Royal und gleichzeitig die komplexeste Aufgabe des Projekts. Laut Aufgabenstellung muss eine gültige Kombination folgende Bedingungen erfüllen: Sie besteht aus 3 bis 5 zusammenhängenden Plättchen mit gleicher Farbe oder gleichem Objekt. Als zusammenhängend gelten Plättchen, die orthogonal (waagerecht oder senkrecht) oder diagonal benachbart sind. Umgedrehte Plättchen dürfen nicht Teil einer Kombination sein. Die offizielle Spielanleitung definiert 17 verschiedene zulässige Kombinationsformen.

Diese Anforderungen bringen mehrere besondere Herausforderungen mit sich:

1. Es gelten komplexe Validierungsregeln. Nicht jede beliebige Gruppe von 3-5 zusammenhängenden Plättchen bildet automatisch eine gültige Kombination. Die 17 spezifischen Formen aus der Spielanleitung müssen exakt eingehalten werden. Während der Entwicklung stellte sich heraus, dass sich diese Formen durch mathematische Eigenschaften beschreiben lassen, was einen algorithmischen Ansatz ermöglicht.
2. Die Definition von Nachbarschaft unterscheidet sich je nach Kontext. Für die Platzierung neuer Plättchen zählen nur orthogonale Nachbarn (4 Richtungen), während für Kombinationen auch diagonale Nachbarn (8 Richtungen insgesamt) berücksichtigt werden müssen. Das erfordert verschiedene Nachbarschaftsprüfungen im Code.
3. Eine größere zusammenhängende Gruppe kann mehrere überlappende Kombinationen enthalten. Liegen beispielsweise 5 blaue Kissen in einer Reihe, hat der Spieler folgende Optionen: die vollständige 5er-Kombination (7 Punkte), mehrere überlappende 4er-Kombinationen (je 4 Punkte) oder mehrere 3er-Kombinationen (je 2 Punkte). Das System muss daher alle möglichen gültigen Teilkombinationen finden und dem Spieler zur Auswahl anbieten.
4. Nach der Wertung müssen die umklappbaren "inneren" Plättchen korrekt identifiziert werden. Die Regeln dafür unterscheiden sich je nach Größe (3/4/5 Plättchen) und Art (orthogonal/diagonal/gemischt) der Kombination. Bei jeder Wertung wird genau ein Plättchen umgedreht. Bei 4er-Kombinationen gibt es dafür zwei zulässige Plättchen, aus denen der Spieler eins auswählt.

## Realisationsanalyse

Für die Implementierung der Kombinationserkennung wurden mehrere grundsätzlich verschiedene Ansätze in Betracht gezogen und gegeneinander abgewogen.

**Ansatz 1: Pattern-Matching mit festen Templates** wurde zunächst evaluiert. Die Grundidee: Alle 17 erlaubten Formen als  $5 \times 5$ -Bitmuster speichern und jede gefundene Plättchen-Gruppe gegen diese Templates prüfen, inklusive aller Rotationen und Spiegelungen.

Dieser Ansatz hätte den Vorteil, exakt der Darstellung in der Spielanleitung zu entsprechen und wäre intuitiv nachvollziehbar. Allerdings ergaben sich erhebliche Nachteile: Mit 17 Mustern, jeweils 4 möglichen Rotationen und 2 Spiegelungen wären bis zu 136 Vergleichsoperationen pro Gruppe notwendig. Das führt bei vielen Plättchen auf dem Board zu ineffizienter Performance. Außerdem wäre der Ansatz schwer erweiterbar – neue Formen hätten sich nur umständlich hinzufügen lassen. Die Offset-Berechnung (Templates müssen an verschiedenen Board-Positionen geprüft werden) hätte zusätzliche Komplexität bedeutet. Aufgrund dieser Skalierbarkeits- und Wartbarkeitsprobleme habe ich diesen Ansatz verworfen.

**Ansatz 2: Regelbasierte Validierung mit Kompaktheit** wurde als finale Lösung gewählt. Statt feste Muster zu vergleichen, werden mathematische Eigenschaften der Kombinationen analysiert. Dazu gehören die Kompaktheit (Verhältnis von Plättchen-Anzahl zur Fläche des umschließenden Rechtecks), der Zusammenhang (Erreichbarkeit aller Plättchen via Tiefensuche), das Vorhandensein linearer diagonaler Ketten (konsistente Bewegungsrichtung), die Erkennung von L-Formen (Eck-Plättchen mit zwei orthogonalen Nachbarn) sowie zusätzliche geometrische Einschränkungen für bestimmte Größen.

**Geometrische Einschränkungen:** Für 4er-Kombinationen muss mindestens eine Dimension des umschließenden Rechtecks  $\geq 3$  sein, um zu kompakte Formen wie  $2 \times 2$  Quadrate auszuschließen. Bei 5er-Kombinationen werden gerade Linien ( $1 \times 5$  oder  $5 \times 1$ ) explizit erlaubt, während für andere 5er-Formen strengere Regeln gelten: Das umschließende Rechteck darf in keiner Dimension größer als 4 sein und bei gemischten Formen (orthogonal + diagonal verbunden) muss jedes Plättchen mindestens einen orthogonalen Nachbarn innerhalb der Kombination haben. Diese Regel verhindert verstreute Formen mit nur schwach (diagonal) verbundenen Ausreißer-Plättchen.

Dieser Ansatz bietet mehrere Vorteile: Er ist generisch und funktioniert für beliebige Formen, arbeitet effizient durch Berechnung von Gruppeneigenschaften statt umfangreicher Mustervergleiche, ist gut testbar (jede Regel kann isoliert getestet werden) und flexibel erweiterbar durch Anpassung der Regeln. Der Nachteil ist, dass die Validierungsregeln durch empirische Analyse der 17 Muster aus der Spielanleitung abgeleitet werden mussten und der resultierende Code komplexe Fallunterscheidungen erfordert. Diese Nachteile werden jedoch durch die deutlich höhere Effizienz und bessere Wartbarkeit aufgewogen.

Für das spezifische Problem, alle möglichen 3-5er Kombinationen aus einer größeren zusammenhängenden Gruppe zu finden, wurden verschiedene algorithmische Ansätze evaluiert.

---

Konkret standen hierzu zwei Ansätze zur Auswahl:

**1. Greedy-Ansatz:**

Die naheliegende Idee wäre ein Greedy-Verfahren, das immer die größtmögliche Kombination aus einer Gruppe nimmt. Der Algorithmus würde bei 5 Plättchen starten, prüfen ob die Teilmenge gültig ist, diese bei Erfolg zur Ergebnisliste hinzufügen und mit den verbleibenden Plättchen fortfahren. Dieser Ansatz hat jedoch ein fundamentales Problem: Er übersieht systematisch alle kleineren Teilkombinationen, die in einer größeren enthalten sind. Wenn beispielsweise 5 Plättchen in einer Reihe liegen, würde der Greedy-Algorithmus nur die vollständige 5er-Kombination finden und die darin enthaltenen zwei 4er-Kombinationen sowie drei 3er-Kombinationen ignorieren. Der Spieler verliert dadurch taktische Optionen, da er möglicherweise strategisch nur eine kleinere Kombination werten möchte, um die übrigen Plättchen für spätere Züge zu behalten.

**2. Backtracking mit vollständiger Enumeration:**

Stattdessen kommt ein Backtracking-Verfahren zum Einsatz, das garantiert alle möglichen Kombinationen findet. Der Algorithmus generiert systematisch für jede Zielgröße  $k \in \{3, 4, 5\}$  alle mathematisch möglichen  $k$ -elementigen Teilmengen der Gruppe. Jede dieser Teilmengen wird dann auf zwei Kriterien geprüft: Erstens muss sie zusammenhängend sein (verifiziert durch Breitensuche über alle acht Richtungen). Zweitens muss sie die Validierungsregeln erfüllen, also Kompaktheit, Linearität oder L-Form-Struktur aufweisen. Nur wenn beide Bedingungen erfüllt sind, kommt die Kombination in die Ergebnisliste.

**Optimierung durch Pruning:**

Gruppen mit mehr als fünf Plättchen werden nicht als Ganzes validiert, da sie zu groß für eine Einzelkombination sind. Stattdessen werden direkt die enthaltenen 3er-, 4er- und 5er-Subsets betrachtet. Selbst im Worst-Case von acht zusammenhängenden Plättchen lassen sich 161 Validierungsprüfungen auf modernen Rechnern in wenigen Millisekunden durchführen. Der entscheidende Vorteil dieses Ansatzes liegt in seiner mathematischen Vollständigkeit: Da jede mögliche Teilmenge explizit geprüft wird, ist garantiert, dass keine gültige Kombination übersehen wird. Das ist für das Spielgeschehen wichtig, damit Spieler keine taktischen Optionen verlieren.

## Realisationsbeschreibung

Die Implementierung erfolgt in der Klasse `Board.java` und gliedert sich in mehrere miteinander verzahnte Komponenten, die im Folgenden detailliert beschrieben werden.

### 1. Hauptmethode: `findCombinations(boolean byColor)`

Diese zentrale Methode steuert den gesamten Erkennungsprozess. Sie iteriert über alle nicht umgedrehten Plättchen auf dem Board und ermittelt für jedes noch nicht besuchte Plättchen eine zusammenhängende Gruppe mittels Tiefensuche. Gruppen mit weniger als 3 Plättchen werden übersprungen. Für alle anderen Gruppen – auch solche mit mehr als 5 Plättchen – werden systematisch alle gültigen 3er-, 4er- und 5er-Subkombinationen identifiziert. So werden auch in großen zusammenhängenden Bereichen alle Wertungsmöglichkeiten gefunden. Die Methode wird in zwei Varianten aufgerufen: `findCombinationsByColor()` und `findCombinationsByObject()`.

### 2. Gruppen finden: `findConnectedGroup(Position start, boolean byColor)`

Ein Tiefensuche-Algorithmus (DFS) identifiziert alle zusammenhängenden Plättchen mit gleicher Farbe oder gleichem Objekt. Der Algorithmus startet bei einer Position und prüft rekursiv alle 8 Nachbarn (orthogonal und diagonal). Wichtig: Für Kombinationen zählt auch diagonale Nachbarschaft – im Gegensatz zur Plättchen-Platzierung, wo nur orthogonale Nachbarn relevant sind.

### 3. Subkombinationen finden: `findConnectedSubsetsOfSize(Set<Position> group, int targetSize)`

Diese Methode generiert mittels Backtracking alle k-elementigen Teilmengen einer Gruppe für  $k \in \{3, 4, 5\}$ . Jede Teilmenge wird auf zwei Kriterien geprüft: Zusammenhang (Breitensuche über 8 Richtungen) und Gültigkeit (Validierungsregeln). Beispiel: Aus 5 Plättchen in einer Reihe {A, B, C, D, E} entstehen eine 5er-Kombination, vier 4er-Kombinationen und sechs 3er-Kombinationen.

### 4. Validierung: `isValidCombination(Set<Position> combo)`

Diese Methode implementiert die Kernlogik der Kombinationserkennung und beschreibt die 17 erlaubten Formen durch mathematische Regeln:

Regel 1: Rein diagonale Ketten müssen linear sein

Kombinationen mit ausschließlich diagonalen Verbindungen (keine orthogonalen Nachbarn) müssen eine konsistente Bewegungsrichtung aufweisen. Die Prüfung erfolgt durch Analyse der Richtungsvektoren zwischen aufeinanderfolgenden Plättchen.

Regel 2: Geometrische Einschränkungen nach Größe

- 4er-Kombinationen: Mindestens eine Dimension des umschließenden Rechtecks muss  $\geq 3$  sein. Das verhindert ungültige Formen wie  $2 \times 2$  Quadrate.
- 5er-Kombinationen: Gerade Linien ( $1 \times 5$  oder  $5 \times 1$ ) werden explizit erlaubt. Für andere 5er-Formen gilt: Das umschließende Rechteck darf in keiner Dimension größer als 4 sein.

#### Regel 3: Orthogonale Nachbarschaft bei gemischten Formen

Kombinationen mit gemischten Verbindungen (orthogonal + diagonal) müssen für jedes Plättchen mindestens einen orthogonalen Nachbarn innerhalb der Kombination aufweisen. Diese Regel verhindert verstreute Formen mit nur diagonal verbundenen Ausreißer- Plättchen.

#### Regel 4: Kompaktheit $\geq 0.5$

Für gemischte Formen wird die Kompaktheit als Quotient aus Plättchen-Anzahl und Fläche des umschließenden Rechtecks berechnet und muss mindestens 0,5 betragen. Beispiele: Eine L-Form (3 Plättchen, 2×2 Rechteck) erreicht  $3/4 = 0,75$  (gültig). Eine zu lockere Zickzack-Form (3 Plättchen, 3×3 Rechteck) erreicht nur  $3/9 \approx 0,33$  (ungültig).

#### Regel 5: Filter für schwach verbundene Plättchen#

Plättchen ohne orthogonale Nachbarn und nur einem diagonalen Nachbarn gelten als "schwach verbunden" und werden iterativ entfernt. Gehen dabei Plättchen verloren, war die Form ungültig.

#### Regel 6: L-Form-Erkennung für 3er-Kombinationen

3er-Kombinationen mit Kompaktheit  $< 1,0$  müssen eine L-Form bilden: Genau ein Plättchen (das "Eck") muss zwei orthogonale Nachbarn haben.

Diese regelbasierte Validierung ermöglicht die präzise Erkennung aller 17 gültigen Kombinationsformen ohne aufwändiges Pattern-Matching. Eine bekannte Limitation besteht bei bestimmten Spiralformen, die fälschlicherweise akzeptiert werden – ein bewusster Kompromiss zugunsten der Wartbarkeit, da solche Formen im Spielverlauf äußerst selten auftreten.

GÜLTIG: L-Form

X	X
X	

Bounding Box: 2x2

Plättchen: 3

Kompaktheit  $3/4 = 0,75$

- ✓ Alle Plättchen haben orthogonale Nachbarn
- ✓ Kompaktheit  $\geq 0,5$

GÜLTIG: S-Form

X	
X	X
	X

Bounding Box: 3x2

Plättchen: 4

Kompaktheit  $4/6 = 0,66$

- ✓ Alle Plättchen haben orthogonale Nachbarn
- ✓ Kompaktheit  $\geq 0,5$

UNGÜLTIG:  
Zu geringe Kompaktheit

X	X		
		X	X

Bounding Box: 2x5

Plättchen: 4

Kompaktheit  $4/10 = 0,4$

X Kompaktheit  $< 0,5$  (zu verstreut)

Abbildung 2.1: Beispiele für gültige und ungültige Kombinationsformen mit umschließendem Rechteck und Kompaktheitsberechnung

## 2.2.2 Board-Verwaltung mit HashMap

### Problemanalyse

Das Spielfeld in Mops Royal bringt widersprüchliche Anforderungen mit sich: Einerseits muss es während des Spiels dynamisch in alle Richtungen wachsen können. Andererseits gilt eine strikte Bedingung: Am Spielende müssen alle Plättchen in ein  $5 \times 5$ -Raster passen. Diese Bedingung muss vor jeder Platzierung geprüft werden.

Zusätzlich ist zu beachten, dass typischerweise nur etwa die Hälfte der möglichen Positionen belegt ist (Sparse Storage). Eine Datenstruktur, die für jedes potenzielle Feld Speicher reserviert, wäre ineffizient. Die Transformation zwischen der internen flexiblen Repräsentation und dem von GUI sowie JSON-Format erwarteten  $5 \times 5$ -Array stellt eine weitere Herausforderung dar.

### Realisationsanalyse

#### Ansatz 1: Festes 2D-Array (verworfen)

Ein `int[5][5]`-Array wäre die naheliegendste Lösung mit direkter Indizierung in  $O(1)$ . Allerdings kann sich das Spielfeld damit nicht über den Bereich (0,0) bis (4,4) hinausbewegen. Das Startplättchen müsste zwingend in der Mitte platziert werden und bei jedem Zugriff wäre eine Koordinaten-Transformation erforderlich. Da etwa 50% der Felder leer sind, würde zudem unnötig Speicher verschwendet. Dieser Ansatz ist für die dynamischen Anforderungen zu unflexibel.

#### Ansatz 2: Dynamisches Array mit Offset-Tracking (verworfen)

Eine Weiterentwicklung wäre ein Array variabler Größe mit separaten `minRow/minCol`-Variablen für den Offset. Das Array würde bei Bedarf wachsen, müsste dafür aber neu zugewiesen und kopiert werden ( $O(n^2)$  bei Expansion). Die Koordinaten-Transformation wäre bei jeder Operation nötig und die Offset-Verwaltung entsprechend fehleranfällig. Zudem entsteht Speicher-Overhead durch die rechteckige Bounding-Box, selbst wenn das Board L-förmig ist. Die Komplexität rechtfertigt den Aufwand nicht.

#### Ansatz 3: `HashMap<Position, Tile>` (gewählt)

Die finale Lösung verwendet eine `HashMap` mit `Position` als Key. Beliebige Koordinaten (auch negativ) sind problemlos möglich und durch Sparse Storage verbrauchen nur belegte Felder Speicher. Zugriff, Einfügen und Löschen erfolgen durchschnittlich in  $O(1)$ . Die `Position` wird als Record implementiert, was unveränderliche `HashMap`-Keys und automatische `equals()`/`hashCode()`-Methoden garantiert.

Die Nachteile sind überschaubar: Etwas höherer Speicher-Overhead pro Entry (~32 Bytes) und keine direkte Iteration über Zeilen/Spalten. Die Transformation zu  $5 \times 5$ -Arrays für GUI und JSON erfordert zusätzlichen Code. Diese Nachteile werden jedoch durch die Flexibilität aufgewogen – spätere Erweiterungen (z.B. größere Spielfelder) bleiben unkompliziert.

## Realisationsbeschreibung

### Position als Record und HashMap-Deklaration

Die Position wird als Rekord `Position(int row, int col)` implementiert. Records sind unveränderlich und damit sichere HashMap-Keys. Der Compiler generiert automatisch korrekte `equals()`- und `hashCode()`-Methoden. Die HashMap selbst ist `final`, sodass die Referenz konstant bleibt und nur der Inhalt sich ändern kann.

### Plättchen-Platzierung mit Copy-Mechanismus

Ein kritisches Problem: Wenn mehrere Spieler das gleiche Plättchen-Objekt vom Deck erhalten, würde `flip()` bei Spieler 1 auch das Plättchen bei Spieler 2 umdrehen. Die Lösung ist simpel aber wichtig: `placeTile()` erstellt mit `tile.copy()` eine Kopie vor dem Einfügen in die HashMap. Jeder Spieler hat damit sein eigenes unabhängiges Plättchen-Objekt.

### Bounding-Box und 5×5-Validierung

Für GUI und JSON muss die HashMap in ein 5×5-Array transformiert werden. Dazu wird die Bounding-Box berechnet (das kleinste Rechteck, das alle platzierten Plättchen umschließt). Die Methode `getBoundingBox()` iteriert einmal über alle Positionen und ermittelt die minimale/maximale Zeilen- und Spaltenindizes.

Diese Bounding-Box dient auch zur Validierung: Vor jeder Platzierung prüft `fitsIn5x5WithNewTile()`, ob nach Hinzufügen des neuen Plättchens die Dimensionen noch  $\leq 5$  sind. Falls ja, ist die Position gültig.

### Anlegepositionen berechnen

Die Methode `computeValidPositions()` ermittelt alle Positionen, an denen das nächste Plättchen legal platziert werden kann. Sie iteriert über alle belegten Felder und prüft deren vier orthogonale Nachbarn (oben, unten, links, rechts). Leere Nachbarn sind Kandidaten und werden zusätzlich per 5×5-Check validiert.

Wichtig: Für Kombinationserkennung zählen acht Richtungen (inklusive diagonal), für die Anlegung jedoch nur vier orthogonale Richtungen. Diese Unterscheidung ist spielregelkonform.

### Transformation für GUI und JSON

Die GUI benötigt ein `Tile[][][]`-Array zur Darstellung. Die Methode `snapshotTiles()` erstellt dieses Array, indem Plättchen relativ zur Bounding-Box-Origin positioniert werden. Dabei ist `Array[0][0]` nicht die Board-Position (0,0), sondern (`minRow, minCol`). Die Methode `getSnapshotOrigin()` gibt diese Origin explizit zurück.

---

Für JSON konvertiert `toSaveFormat()` das Board in ein `int[][]`-Array mit dem Encoding `Farbe*100 + Objekt*10 + (umgedreht ? 1 : 0)`. Leere Zellen werden mit 990 kodiert. Beim Laden durch `loadFromSaveFormat()` wird dieser Prozess umgekehrt. Da JSON immer (0,0) bis (4,4) verwendet, werden geladene Boards unabhängig von ursprünglichen Koordinaten normalisiert.

Diese beiden Informationen – das Array und die Origin – werden an die GUI übergeben (siehe Kapitel 2.2.3 für die weitere Verarbeitung in der GUI).

### **Kritische Bewertung**

Die Berechnung gültiger Anlegepositionen iteriert über alle Plättchen und prüft jeden Kandidaten. Bei 24 Plättchen und ~40 Kandidaten ergeben sich etwa 960 Bounding-Box-Berechnungen pro Zug. Eine Optimierung wäre das Caching der Bounding-Box mit inkrementeller Aktualisierung. In der Praxis ist die Performance jedoch ausreichend (<1ms), sodass diese Optimierung in diesem Szenario verzichtbar ist.

HashMap (flexible Koordinaten):

	-1	0	1	2
-1		A		
0		B	C	
1		D		

(-1,0): Tile A  
 (0,0): Tile B (Start)  
 (0,1): Tile C  
 (1,0): Tile D

Bounding-Box: minRow = -1, maxRow = 1, minCol = 0, maxCol = 1  
 → Größe: 3x2 passt in 5x5

↓ Transformation (mit Offset)

5x5 Array für GUI/JSON:

	0	1	2	3	4
0	A				
1	B	C			
2	D				
3					
4					

← Offset: arrayRow = row - minRow  
 0 = (-1) - (-1)

Abbildung 2.2: Bounding-Box-Transformation – von negativen Koordinaten zu Array-Indizes

## 2.2.3 Board-Rendering mit Origin-Offset

### Problemanalyse

Das Board liefert nach der Platzierung eines Plättchens zwei Informationen an die GUI: Ein  $5 \times 5$ -Array via `snapshotTiles()` und die Origin-Position via `getSnapshotOrigin()` (siehe Kapitel 2.2.2). Die GUI muss diese Daten im `UserInterfaceController` verarbeiten und anzeigen. Dabei treten zwei Herausforderungen auf.

Erstens verwendet die GUI ein  $9 \times 9$ -GridPane, um Zoom und Scroll zu ermöglichen. Das kleinere  $5 \times 5$ -Board soll zentriert dargestellt werden. Die Board-Position  $(0,0)$  sollte optisch in der Mitte des Grids liegen, nicht in der oberen linken Ecke. Die GUI muss also für jedes Plättchen berechnen, an welcher Stelle im  $9 \times 9$ -Grid es dargestellt werden soll.

Zweitens muss die Transformation bidirektional funktionieren. Beim Rendering werden Board-Daten in UI-Koordinaten umgewandelt, bei Mausklicks läuft der Prozess umgekehrt: Die Pixel-Position des Klicks muss zurück in Board-Koordinaten transformiert werden, damit die Spiellogik entscheiden kann, ob die Position gültig ist. Die zweistufige Transformation verkompliziert die Sache zusätzlich: Array-Index  $\rightarrow$  Board-Koordinaten  $\rightarrow$  UI-Koordinaten beim Rendering, umgekehrt bei Klicks.

### Realisationsanalyse

#### Ansatz 1: Festes Mapping ohne Zentrierung (verworfen)

Die naheliegende Lösung wäre, Array $[0][0]$  direkt auf GridPane $[0][0]$  zu mappen. Das Board würde in der oberen linken Ecke erscheinen, der Rest des Grids bliebe leer. Das funktioniert technisch, sieht aber unästhetisch aus. Zudem erschwert es das Zoomen, weil der Spieler nach jedem Zoom manuell zur oberen linken Ecke scrollen müsste. Die fehlende Zentrierung macht die Bedienung umständlich.

#### Ansatz 2: Dynamische Zentrierung pro Rendering (verworfen)

Die GUI könnte bei jedem Rendering neu berechnen, wo das Board optimal positioniert werden sollte. Das hätte den Vorteil maximaler Flexibilität. Das Board würde somit immer perfekt zentriert sein, auch wenn sich die Grid-Größe ändert. Der Nachteil: Bei jedem Frame müsste die Bounding-Box erneut berechnet werden und das Board würde bei Größenänderungen des Fensters "springen". Das irritiert den Spieler und kostet Performance.

#### Ansatz 3: Statischer UI-Offset beim Start (gewählt)

Die finale Lösung berechnet den Offset einmalig beim Programmstart. Das UI-Grid-Zentrum wird ermittelt (bei 9 Spalten: Spalte 4) und der Offset wird so gesetzt, dass Board-Position  $(0,0)$  auf dieses Zentrum gemappt wird. Der Offset bleibt während des gesamten Spiels konstant. Das ermöglicht eine vorhersagbare, stabile Darstellung ohne Rechenaufwand pro Frame. Der einzige

Nachteil: das Board ist nur bei fester Grid-Größe perfekt zentriert. Das wird in Kauf genommen, da die Grid-Größe während des Spiels konstant bleibt.

## Realisationsanalyse

### Ausgangslage: Array und Origin vom Board

Das Board liefert (siehe Kapitel 2.2.2) ein  $5 \times 5$ -Array und eine Origin-Position.  $\text{Array}[0][0]$  entspricht dabei der Position `origin` im Board-Koordinatensystem. Diese beiden Informationen werden an `renderBoard(Tile[][] tiles, Position origin)` übergeben.

### UI-Offset-Berechnung

Die Methode `computeUiCenterAndOffsetsSafely()` berechnet beim Programmstart den statischen Offset. Das  $9 \times 9$ -GridPane hat sein Zentrum bei Index 4 (Ganzzahl-Division:  $9/2 = 4$ ). Der UI-Offset ergibt sich als Differenz zwischen UI-Zentrum und Board-Zentrum:

```
uiColOffset = (gridPane.getColumnCount() / 2) - 0; // = 4
uiRowOffset = (gridPane.getRowCount() / 2) - 0; // = 4
```

Damit wird Board-Position (0,0) auf UI-Position (4,4) gemappt und das Board erscheint zentriert im Grid.

### Klick-Transformation: Umgekehrter Weg

Bei einem Mausklick liefert JavaFX Pixel-Koordinaten, die zunächst in UI-Grid-Koordinaten umgerechnet werden:

```
int uiCol = (int) (mouseEvent.getX() / cellWidth);
int uiRow = (int) (mouseEvent.getY() / cellHeight);
```

Dann wird der UI-Offset subtrahiert, um Board-Koordinaten zu erhalten:

```
int boardCol = uiCol - uiColOffset;
int boardRow = uiRow - uiRowOffset;
```

Die resultierende Board-Position wird an `game.onBoardClick(boardRow, boardCol)` übergeben. Die Spiellogik entscheidet dann über die Gültigkeit der Position und platziert gegebenenfalls ein Plättchen (siehe Kapitel 2.2.2). Das neue Board-Array wird dann wieder via `renderBoard()` angezeigt.

Diese Architektur trennt sauber die Verantwortlichkeiten: Das Board (Kapitel 2.2.2) kümmert sich um die Transformation von `HashMap` zu `Array` und die GUI um die Transformation von `Array` zu UI-Koordinaten. Die Origin-Information ist die Brücke zwischen beiden Welten.

INPUT: 5x5 Array + Origin

	0	1	2	3	4
0	A				
1	B	C			
2	D				
3					
4					

↓ Zweistufige Transformation mit uiOffset

OUTPUT: 9x9 UI-GridPane (zentriert)

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3					A				
4					B	C			
5					D				
6									
7									
8									

uiOffset: Row=4, Col=4  
 Board(0,0) wird auf UI(4,4) gemappt

Beispiel Tile B (Startplättchen):  
 $\text{Array}[1][0] \rightarrow \text{Board}: (-1,0)+(1,0) = (0,0) \rightarrow \text{UI}: (0,0)+(4,4) = (4,4)$

Klick auf UI(5,4):  
 $\text{UI}(5,4) \rightarrow \text{Board}: (5,4)-(4,4) = (1,0) \rightarrow \text{Tile D}$

Abbildung 2.3: Transformation des Board-Arrays ins zentrierte UI-Grid mittels uiOffset.

## 2.2.4 GUI-Logik-Trennung

### Problemanalyse

Eine saubere Trennung zwischen Benutzeroberfläche und Spiellogik ist in der Aufgabenstellung explizit vorgeschrieben und hat mehrere praktische Gründe. JUnit-Tests müssen die Spiellogik ohne JavaFX testen können, was problematisch ist, da JavaFX einen eigenen Application Thread benötigt und die Initialisierung Zeit kostet. Die GUI soll theoretisch austauschbar sein – andere Projektteilnehmer könnten ein Redesign erstellen oder die Oberfläche durch eine andere Technologie ersetzen. Zudem gehören Ausgabetexte zur Präsentation, nicht zur Logik. Fehlermeldungen und Spielereignisse werden als typsichere Enums übergeben, während die GUI diese in die gewünschte Sprache übersetzt.

Die konkreten Vorgaben sind eindeutig: Das GUI-Paket darf Logik-Klassen importieren, aber das Logik-Paket darf keinerlei GUI-Imports enthalten – kein `javafx.*`, kein `java.awt.*`. Das `GUIConnector`-Interface liegt im Logik-Paket, die Implementierung `JavaFXGUI` im GUI-Paket und für Tests existiert eine `FakeGUI` ohne echte UI-Komponenten.

### Realisationsanalyse

Obwohl die Verwendung eines `GUIConnector`-Interfaces vorgegeben war, mussten konkrete Designentscheidungen getroffen werden. Die naheliegende Alternative wären direkte GUI-Aufrufe aus der Logik gewesen. Die Game Klasse würde einfach Methoden des `UserInterfaceController` aufrufen. Das macht die Logik aber vollständig von einer spezifischen GUI-Implementierung abhängig, Tests bräuchten echte UI-Komponenten und ein Austausch der Oberfläche würde Änderungen in der Logik erfordern.

Ein fortgeschrittenerer Ansatz wäre das Observer-Pattern: Die Game Klasse feuert Events und die GUI registriert sich als Listener. Das ist ein Standard-Pattern in Java und würde multiple Listener ermöglichen. Allerdings ist das für die 1:1-Beziehung zwischen Game und GUI überdimensioniert, denn Listener-Management und Event-Definitionen würden unnötige Komplexität schaffen.

Die gewählte Interface-Lösung ist der Mittelweg: Saubere Entkopplung durch Abstraktion, aber ohne den Overhead eines vollständigen Observer-Systems. Wichtige Designentscheidungen wurden dabei getroffen:

**Enums statt Strings:** Statt `showError(String message)` existiert `showUserError(ErrorType errorType)`. Die Logik über gibt nur den Fehlertyp (`INVALID_POSITION` oder `INVALID_SELECTION`), die GUI kümmert sich dann um die konkrete Formulierung. Das gleiche Prinzip gilt für Spielereignisse wie `PLAYER_SKIP` oder `CATCH_UP_ROUND_START`.

**Callbacks für asynchrone Operationen:** Die Wertungsansicht ist modal – das Spiel muss warten, bis der Spieler seine Auswahl getroffen hat. Die Methode `showScoringView()` erhält daher einen Callback (`Runnable onScoringComplete`), den die GUI nach Abschluss aufruft.

**Void-Methoden:** Alle Interface-Methoden geben `void` zurück. Die GUI aktualisiert nur, fragt aber nicht ab. Das ist eine bewusste One-Way-Communication von Logik zu GUI.

## Realisationsbeschreibung

### GUIConnector-Interface

Das Interface im `logic`-Paket definiert 17 Methoden, die sich in mehrere Kategorien einteilen lassen: Spielfeld-Darstellung (`renderBoard`, `highlightPositions`), Spieler-Information (`showActivePlayer`, `updatePlayerBoards`), Plättchen-Anzeige (`showNextTile`), Meldungen (`showUserError`, `showGameEvent`) und Spielfluss-Steuerung (`showScoringView`, `scheduleNextPlayer`, `showWinner`).

Die Verwendung von Enums ist konsequent umgesetzt. `ErrorType` definiert zwei Fehlerarten: `INVALID_POSITION` (ungültige Position beim Platzieren) und `INVALID_SELECTION` (ungültiges Plättchen zum Umdrehen). `GameEvent` umfasst drei Spielereignisse: `PLAYER_SKIP` (eigene Startkarte gezogen), `CATCH_UP_ROUND_START` (Nachholrunde beginnt) und `PLAYER_MUST_SKIP_FULL_BOARD` (Board voll mit 25 Plättchen).

Die Spiellogik übergibt ein Lambda, das die GUI nach Abschluss der Wertung ausführt. Das vermeidet Polling-Schleifen oder blockierende Waits.

Callbacks lösen ein typisches Problem bei modalen Dialogen: Das Spiel muss auf eine Benutzereingabe warten, darf aber weder den UI-Thread blockieren noch aktiv prüfen, ob der Benutzer fertig ist. Eine mögliche Alternative wäre Polling: Eine Schleife die ständig prüft, ob die Wertung abgeschlossen ist. Das ist ineffizient und verschwendet CPU-Leistung. Eine weitere Alternative wäre blockierendes Warten (`Thread.sleep`), was jedoch die gesamte Benutzeroberfläche einfrieren würde. Callbacks umgehen beide Probleme: Die GUI ruft die übergebene Funktion genau einmal auf, sobald der Benutzer seine Auswahl bestätigt hat.

### JavaFXGUI-Implementierung

Die Klasse `JavaFXGUI` im `gui`-Paket implementiert das Interface und delegiert alle Aufrufe an den `UserInterfaceController`. Dabei kommt `Platform.runLater()` zum Einsatz. Dabei handelt es sich um eine Methode, die UI-Updates in den JavaFX Application Thread verschiebt. In diesem Projekt läuft die Spiellogik im selben Thread wie die GUI, sodass `Platform.runLater()` technisch nicht zwingend erforderlich wäre. Die Verwendung stellt jedoch sicher, dass der Code auch bei zukünftigen Erweiterungen (etwa asynchrone Dateioperationen oder Netzwerk-Features) robust bleibt.

## UserInterfaceController und Enum-Übersetzung

Der UserInterfaceController im gui-Paket führt die eigentlichen JavaFX-Operationen aus. Hier erfolgt auch die Übersetzung von Enums in Ausgabetexte. Bei showUserError(ErrorType errorType) wird per Switch-Case der entsprechende deutsche Text ermittelt:

```
String message = switch (errorType) {  
    case INVALID_POSITION ->  
        "Ungültige Position!\nBitte wähle eine markierte Position.";  
    case INVALID_SELECTION ->  
        "Ungültige Auswahl!\nDieses Plättchen kann nicht umgedreht werden.";  
};
```

Für eine englische Version müsste nur diese eine Methode angepasst werden. Das gleiche Prinzip gilt für showGameEvent(), das Spielereignisse wie "Nachholrunde: [Spieler] ist dran!" übersetzt.

## FakeGUI für Tests

Die FakeGUI im Test-Paket implementiert GUIConnector mit leeren Methoden, sodass Tests die Spiellogik ohne JavaFX prüfen können. Als zusätzliches Feature protokolliert sie alle Methodenaufrufe in einer Liste:

```
public List<String> calls = new ArrayList<>();  
  
@Override  
public void showUserError(ErrorType errorType) {  
    calls.add("showUserError:" + errorType);  
}
```

Die Tests können damit verifizieren, dass die Logik die GUI korrekt ansteuert.

## Initialisierung

In ApplicationMain wird die Kette aufgebaut: Der UserInterfaceController wird vom FXMLLoader erstellt, daraus entsteht eine JavaFXGUI-Instanz, die an Game übergeben wird. Die Game-Klasse kennt nur das Interface und ruft bei Zustandsänderungen die entsprechenden Methoden auf. Umgekehrt setzt der UserInterfaceController eine Referenz auf Game, um auf Benutzerinteraktionen reagieren zu können. Diese bidirektionale Verbindung ist unkritisch, da die Logik nur das Interface kennt, während die GUI direkt auf die Logik zugreifen darf.

## 2.2.5 Spielablauf-Steuerung

### Problemanalyse

Anders als bei vielen Brettspielen verwendet die digitale Version von Mops Royal ein gemeinsames Plättchen pro Runde. Alle Spieler müssen also dasselbe Plättchen legen, bevor ein neues gezogen wird. Diese Regel bringt spezifische Anforderungen mit sich.

Erstens muss der Spielerwechsel präzise getaktet sein. Ein neues zentrales Plättchen wird erst gezogen, wenn alle Spieler einmal dran waren. Bei drei Spielern bedeutet das: Spieler A → B → C, dann neues Plättchen, dann wieder A → B → C. Die Logik muss tracken, wann eine volle Runde abgeschlossen ist.

Zweitens erfolgt die GUI-Interaktion asynchron. Nach der Platzierung eines Plättchens kann eine Wertungsansicht erscheinen, die modal auf Benutzereingabe wartet. Der Spielablauf muss pausieren, ohne den UI-Thread zu blockieren oder aktiv zu pollen. Callbacks lösen dieses Problem (siehe Kapitel 2.2.4 für Details).

Drittens ist das Spielende an mehrere Bedingungen geknüpft. Das reguläre Spiel endet, wenn alle Spieler 24 Plättchen gelegt haben (plus Startplättchen = 25 gesamt). Allerdings können Spieler während des Spiels ihre eigene Startkarte ziehen und müssen dann aussetzen. Diese Spieler erhalten nach dem regulären Ende weitere Nachholrunden. Die Spielende-Prüfung muss zwischen "alle fertig" und "warten auf Nachholrunden" unterscheiden.

### Realisationsanalyse

#### Ansatz 1: Synchroner Game Loop mit Blockierung (verworfen)

Eine klassische while-Schleife würde nacheinander jeden Spieler aktivieren und auf dessen Zug warten. Das ist das Standard-Pattern bei Konsolenspielen oder rundenbasierten Servern. Allerdings würde bei modalen Dialogen (Wertungsansicht) der gesamte Loop blockieren oder ineffizientes Polling erfordern (siehe Kapitel 2.2.4 für Alternativen-Diskussion). Dieser Ansatz ist für GUI-Anwendungen ungeeignet.

#### Ansatz 2: Event-basiert mit State Machine (verworfen)

Der Spielzustand könnte als explizite State Machine modelliert werden (WAITING\_FOR\_CLICK, SHOWING\_SCORING, SWITCHING\_PLAYER). Jedes UI-Event triggert einen Zustandsübergang. Das ist ein sauberes theoretisches Konzept und würde asynchrone Abläufe elegant handhaben. Der Nachteil: Die Komplexität steigt erheblich. Zustandsübergänge müssen explizit definiert werden, Edge Cases erfordern zusätzliche States und das Debugging wird schwieriger, da der Kontrollfluss über viele Methoden verteilt ist. Für ein Spiel dieser Größe ist eine vollständige State Machine überdimensioniert.

---

### Ansatz 3: Callback-basierte Steuerung (gewählt)

Die finale Lösung kombiniert anweisungsbasierten Code mit asynchronen Callbacks. Der Hauptablauf ist in `onBoardClick()` linear lesbar: Validierung → Platzierung → Wertung → Callback mit Spielerwechsel. Die Callback-Mechanik und deren Vorteile gegenüber Polling/Blockierung werden in Kapitel 2.2.4 detailliert beschrieben. Das `waitingForNextPlayer`-Flag verhindert mehrfache Klicks während der Callback-Ausführung. Dieser Ansatz ist pragmatisch: Der Code bleibt nachvollziehbar und die Komplexität bleibt überschaubar, da maximal eine Callback-Ebene verschachtelt wird.

## Realisationsbeschreibung

### Spielstart und Initialisierung

Die Methode `startNewGame(List<String> names, boolean[] takesPart)` initialisiert das Spiel. Zunächst werden alle Zustände zurückgesetzt: `gameEnded = false`, `currentIndex = 0` und die Nachholrunden-Queue wird geleert. Dann wird das Spieler-Array mit exakt so vielen Elementen erstellt, wie Spieler teilnehmen. Ein zentrales Deck wird erstellt, jeder Spieler erhält ein zufälliges Startplättchen via `drawStartTile()` und das erste gemeinsame Plättchen wird gezogen. Der Speichern-Button wird aktiviert und die GUI aktualisiert.

### Zug-Ablauf mit asynchronen Callbacks

Der Haupteinstieg ist `onBoardClick(int row, int col)`. Diese Methode koordiniert den kompletten Spielablauf nach einem Klick. Sie führt zunächst zwei Prüfungen durch: Hat der Spieler noch Platz auf seinem Board? Ist das aktuelle Plättchen die eigene Startkarte? Falls eine Bedingung greift, wird der Spieler automatisch übersprungen.

Bei einem normalen Spielzug wird die Position validiert, das Plättchen platziert und das Board auf Kombinationen geprüft. Falls Kombinationen existieren, öffnet sich die Wertungsansicht via `gui.showScoringView(player, combinations, callback)`. Der übergebene Callback enthält den weiteren Spielablauf: Entfernung aus der Nachholrunden-Queue (falls zutreffend), Spielerwechsel via `nextPlayer()`, eventuelles Ziehen eines neuen Plättchens, GUI-Update und Spielende-Prüfung. Das `waitingForNextPlayer`-Flag wird gesetzt, um Mehrfachklicks zu verhindern. Bei fehlenden Kombinationen folgt ein kürzerer Callback via `scheduleNextPlayer()` mit identischem Ablauf.

### Spielerwechsel und Tile-Ziehen

Die Methode `nextPlayer()` erhöht den `currentIndex` zyklisch: `currentIndex = (currentIndex + 1) % players.length`. Bei drei Spielern ergibt sich die Sequenz  $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ . Ein neues zentrales Plättchen wird nur gezogen, wenn der `previousIndex` auf den letzten Spieler zeigt. Die Prüfung `if (previousIndex == players.length - 1)` stellt sicher, dass erst nach einer vollständigen Runde ein neues Plättchen kommt. Bei vier Spielern bedeutet `previousIndex == 3`: Spieler D war dran, also wird jetzt gezogen.

## Spielende-Prüfung und Gewinner-Ermittlung

Die Methode `checkGameEnd()` zählt via Stream alle Spieler mit mindestens 24 gelegten Plättchen (plus Startplättchen = 25). Drei Fälle werden unterschieden:

- Erstens, alle Spieler haben 25 Plättchen und die Nachholrunden-Queue ist leer – das Spiel endet via `determineWinner()`.
- Zweitens, die Queue ist nicht leer und genügend Spieler haben 24+ Plättchen – `handleCatchUpRounds()` wird aufgerufen.
- Drittens, weder noch – das Spiel läuft weiter.

Die Gewinner-Ermittlung in `determineWinner()` findet zunächst den höchsten Score. Bei Gleichstand greift der Tie-Breaker: Der Spieler mit weniger umgedrehten Plättchen gewinnt. Das Board zählt via `countFlippedTiles()` die umgedrehten Plättchen. Bei identischem Score und identischer Anzahl umgedrehter Plättchen sind mehrere Spieler gemeinsam Sieger. Die Liste aller Gewinner wird an die GUI übergeben.

## Nachholrunden-System

Spieler, die ihre eigene Startkarte ziehen, kommen in eine Queue: `playersWhoNeedCatchUp.add(current)`. Diese Queue wird nach dem regulären Spielende abgearbeitet. Die Methode `handleCatchUpRounds()` verwendet `peek()` statt `poll()`, um den nächsten Spieler zu identifizieren ohne ihn zu entfernen. Vor der Aktivierung erfolgt eine kritische Prüfung: Hat der Spieler inzwischen 24 Plättchen gelegt, wird er via `poll()` aus der Queue entfernt und eine rekursive Aktivierung des nächsten Spielers erfolgt. Das verhindert Endlosschleifen.

Kann der Spieler noch spielen, wird er als `currentIndex` aktiviert und ein neues Plättchen gezogen. Nach erfolgreicher Platzierung wird der Spieler in den Callbacks von `onBoardClick()` via `remove(current)` aus der Queue entfernt. Der nächste `checkGameEnd()`-Aufruf prüft, ob weitere Nachholspieler warten und aktiviert gegebenenfalls den nächsten via erneutem `handleCatchUpRounds()`-Aufruf. Ist die Queue leer, endet das Spiel endgültig.

Die Architektur kombiniert imperativen Code für den Hauptfluss mit Callbacks für asynchrone GUI-Interaktion. Das `waitingForNextPlayer`-Flag und die Nachholrunden-Queue fungieren als zentrale Koordinationsmechanismen. Die Trennung zwischen regulärem Ende und Nachholrunden erfolgt durch die Bedingung in `checkGameEnd()`, die beide Phasen sauber unterscheidet.

## 2.2.6 Speichern/Laden mit JSON

### Problemanalyse

Die Aufgabenstellung verlangt das Speichern und Laden kompletter Spielstände im JSON-Format mit präzisen Vorgaben: Ein players-Array mit Namen, Punkten, Startplättchen und 5×5-Boards, ein turn-Feld für den aktuellen Spieler (0-basiert) und ein nextCard-Feld für das nächste zu legende Plättchen (siehe Kapitel 1.3.8.2 Abb. 1.12). Das Format muss mit dem Aufgabenformat kompatibel sein und alle denkbaren Eingabefehler abfangen.

Die Herausforderung liegt in der bidirekionalen Transformation: Beim Speichern muss der komplette Spielzustand (Boards, Spieler, Punkte, aktueller Index) in JSON übersetzt werden. Beim Laden muss aus JSON ein funktionsfähiger Game-Zustand rekonstruiert werden, inklusive Validierung aller Eingaben. Zusätzlich gibt es unterschiedliche Code-Formate: Board-Plättchen werden 3-stellig kodiert (code = Farbe × 100 + Objekt × 10 + Flipped), während nextCard 2-stellig ist (Farbe\*10 + Objekt).

Ein kritisches Problem trat während der Entwicklung auf: Nach dem Laden eines Spielstands waren bereits gelegte Plättchen nicht aus dem zentralen TileDeck entfernt worden. Dies führte zu Duplikaten. Plättchen existierten sowohl auf den Boards der Spieler als auch im Vorrat und konnten erneut gezogen werden. Das Problem wurde durch systematisches Testen mit konkreten Spielständen aufgedeckt: Ein Plättchen mit Code 620 (Gelber Knochen) lag beispielsweise sowohl im Board als auch im Deck vor, da das Deck beim Laden mit allen 36 Tiles initialisiert wurde, ohne die bereits platzierten zu berücksichtigen. Das besondere an dieser Anforderung ist, dass nextCard in einem anderen Format kodiert ist (2-stellig) als Board-Tiles (3-stellig), was zu zusätzlicher Komplexität beim Laden führt.

### Realisationsanalyse

#### Ansatz 1: Vollständig manuelles JSON (verworfen)

Sowohl Schreiben als auch Lesen würde händisch mit StringBuilder und String-Parsing erfolgen. Das gibt maximale Kontrolle über das Format und vermeidet Bibliotheks-Abhängigkeiten. Allerdings ist manuelles JSON-Parsing fehleranfällig. Fehlende Kommata, falsche Klammern und Escape-Sequenzen führen schnell zu Bugs. Die Validierungslogik müsste komplett selbst implementiert werden, was den Code deutlich aufblätzt. Zudem ist die Wartbarkeit schlecht, da Änderungen am Format an vielen Stellen nachgezogen werden müssen.

#### Ansatz 2: Java Serializable (verworfen)

Die Game-Klasse könnte Serializable implementieren und via ObjectOutputStream gespeichert werden. Das wäre technisch am einfachsten und würde automatisch alle Felder serialisieren. Der entscheidende Nachteil: Das Aufgabenformat verlangt explizit JSON mit menschenlesbarem Text. Serializable erzeugt binäre Dateien, die nicht mit dem geforderten Format kompatibel sind. Zudem wäre die Versionierung problematisch, denn Änderungen an Klassen-Strukturen würden bestehende Spielstände brechen.

---

### **Ansatz 3: Hybrid mit Gson (gewählt)**

Die finale Lösung kombiniert manuelles Schreiben mit Gson-basiertem Lesen. SaveGameWriter baut das JSON via StringBuilder, was volle Kontrolle über das exakte Format gibt und die Kompatibilität mit Aufgabenvorgaben sichert. SaveGameReader nutzt Gson für das Parsing, was vor fehlerhaftem JSON schützt und automatisch in Datenklassen deserialisiert. Eine umfangreiche Validierung prüft Spieleranzahl (2-4), turn-Index, nextCard-Code und Board-Dimensionen. Dieser Ansatz verbindet die Vorteile beider Welten: Kontrolliertes Schreiben und robustes Lesen.

## **Zusätzliche Realisationsansätze für die TileDeck-Konsistenz**

### **Ansatz A: Tiles im SaveGameReader entfernen (verworfen)**

Man könnte die bereits gelegten Tiles bereits beim Einlesen identifizieren und direkt vom Deck entfernen. Der Nachteil: Der SaveGameReader hätte dann Verantwortung für Game-Logik, was die klare Separation of Concerns verletzt. Zudem hätte SaveGameReader eine Abhängigkeit auf TileDeck. Dies würde die Architektur unnötig koppeln.

### **Ansatz B: Beim Laden im Game-Konstruktor entfernen (gewählt)**

Der spezialisierte Game-Konstruktor (Konstruktor 3) rekonstruiert nicht nur Spieler und Boards, sondern auch den korrekten Zustand des TileDeck. Dies folgt dem Symmetrie-Prinzip: Im normalen Spielfluss entfernt die Methode `draw()` einzelne Tiles aus `remaining`. Beim Laden entfernen wir alle bereits gespielten Tiles auf einmal mit einer neuen Methode `markTilesAsPlayed()`. Dadurch erhält TileDeck keine Game-Logik-Abhängigkeiten und die vollständige Zustandsrekonstruktion bleibt in Game lokalisiert. Diese Trennung der Verantwortlichkeiten ermöglicht es jede Klasse einfach zu testen und zu verstehen.

## **Realisationsbeschreibung**

### **Speichern mit SaveGameWriter**

Die Klasse SaveGameWriter baut das JSON manuell mit StringBuilder. Die Methode `buildJson()` iteriert über alle Spieler und serialisiert deren Namen, Punkte und Boards. Für jedes Board ruft sie `board.toSaveFormat()` auf, dass die HashMap in ein `int[][]`-Array mit dem 3-stelligen Encoding konvertiert (siehe Kapitel 2.2.2). Besondere Sorgfalt erfordert die Escape-Sequenz-Behandlung: Spielernamen mit Anführungszeichen oder Backslashes müssen korrekt escaped werden. Das fertige JSON wird via `Files.writeString()` geschrieben.

### **Laden mit SaveGameReader**

Die Klasse SaveGameReader liest die Datei und nutzt Gson zum Parsing in eine GameSaveData-Datenklasse. Diese enthält nur public fields, deren Namen exakt den JSON-Keys entsprechen.

Gson mappt automatisch via Reflection. Nach dem Parsing erfolgt umfangreiche Validierung: Spieleranzahl 2-4, turn-Index im gültigen Bereich, nextCard-Code zwischen 11-66 oder 99, alle Boards exakt 5×5. Bei Fehlern wird IllegalArgumentException mit aussagekräftiger Meldung geworfen. Die Validierung verhindert Crashes und erfüllt die Aufgabenanforderung "alle denkbaren Fehler abfangen".

## Rekonstruktion in Game-Konstruktor

Ein spezieller Konstruktor (Konstruktor 3) rekonstruiert den Spielzustand aus den geladenen Daten. Für jeden Spieler wird ein Player-Objekt erstellt, der Score gesetzt und das Board via `loadFromSaveFormat()` geladen (siehe Kapitel 2.2.2). Das Startplättchen wird aus Position (0,0) des Boards rekonstruiert, da es dort per Definition liegt. Die Anzahl gelegter Plättchen wird aus der Board-Größe abgeleitet: `tilesPlaced = getTileCount() - 1`. Das nächste Plättchen wird aus dem 2-stelligen nextCard-Code rekonstruiert, wobei 99 für "kein Plättchen mehr" steht. Nach vollständiger Rekonstruktion wird die GUI aktualisiert.

## Sicherung der TileDeck-Konsistenz beim Laden

Um das Duplikat-Problem zu beheben, wurde folgendes Konzept umgesetzt: Nach dem Laden aller Boards wird TileDeck durch eine neue Methode `markTilesAsPlayed()` auf den korrekten Zustand gesetzt. Diese Methode entfernt Tiles aus dem remaining-Array, indem sie nach Farbe und Objekt-Typ vergleicht. Der Flipped-Status wird ignoriert, da ein Tile eindeutig durch diese zwei Attribute definiert ist und der Flipped-Status nur die Darstellung betrifft.

Die Rekonstruktion läuft in zwei logisch getrennten Phasen ab:

### Phase 1: Board-Tiles und Startplättchen sammeln

Zunächst werden alle Positionen aller geladenen Boards iteriert. Für jede nicht-leere Position wird der 3-stellige Tile-Code dekodiert: Die Farbe ergibt sich aus `code / 100`, das Objekt aus `(code % 100) / 10`. Zusätzlich werden die Startplättchen aus dem initial-Feld aller Spieler dekodiert und zur Liste hinzugefügt. Nach dem Sammeln aller dieser Tiles wird `centralDeck.markTilesAsPlayed(playedTiles)` aufgerufen, um alle Board-Tiles in einer Operation aus dem Deck zu entfernen.

### Phase 2: nextCard aus Deck entfernen

Das nächste Plättchen wird separat behandelt. Der 2-stellige nextCard-Code wird dekodiert - Farbe aus `nextTileCode / 10`, Objekt aus `nextTileCode % 10`. Das Tile wird rekonstruiert und als `currentTile` gespeichert. Da auch `nextCard` bereits vom Deck gezogen wurde (steht in der Speicherdatei), muss es ebenfalls aus dem Deck entfernt werden. Hierzu wird erneut `markTilesAsPlayed()` aufgerufen, diesmal mit dem einzelnen nextCard-Tile.

---

### Begründung der Zwei-Phasen-Aufteilung

Die Separation in zwei Phasen macht die Architektur klar verständlich: Board-Tiles werden als eine konzeptionelle Gruppe entfernt. nextCard separat, da es konzeptionell unterschiedlich ist: nicht auf einem Board befindlich, sondern gerade gezogen und in der nächsten Runde zu spielen. Diese Aufteilung spiegelt auch wider, wie die Daten in der JSON-Datei strukturiert sind.

### Besonderheit: Code-Format-Unterschied

Eine Besonderheit ist die Unterscheidung zwischen 3-stelligen Board-Codes (beispielsweise 620 = Farbe 6, Objekt 2, Flipped 0) und 2-stelligen nextCard-Codes (beispielsweise 62 = Farbe 6, Objekt 2). Diese Unterscheidung ist notwendig, da nextCard beim Draw niemals umgedreht sein kann und daher das Flipped-Feld redundant ist. Der ursprünglich vergessene Schritt, nextCard aus dem Deck zu entfernen, führte genau zu solchen Duplikaten: Die 620 vom Board und die 62 als nextCard waren identisch und konnten beide als Duplikat vorhanden sein. Das Problem wurde aufgedeckt, indem ein konkreter Spielstand mit Board-Tile 620 und nextCard 62 geladen wurde und dann beide Tiles im Deck vorhanden waren.

### Architektur und Separation of Concerns

Die Architektur trennt sauber zwischen Schreiben (manuell, kontrolliert) und Lesen (Gson, robust). Die Board-Methoden `toSaveFormat()` und `loadFromSaveFormat()` kapseln die Koordinaten-Transformationen und werden sowohl beim Speichern als auch beim Laden wiederverwendet. Die neue Methode `markTilesAsPlayed()` in TileDeck ermöglicht es, den Deck-Zustand nachträglich zu korrigieren, ohne dass SaveGameReader oder Board davon wissen müssen. Jede Klasse behält so ihre Verantwortung. SaveGameReader kümmert sich nur um das Einlesen und die Validierung, Game-Konstruktor 3 um die Zustandsrekonstruktion und TileDeck um die Tile-Verwaltung. Dadurch bleibt der Code wartbar und ermöglicht zukünftige Änderungen ohne Kaskadeneffekte.

## 2.3 Verwendete fortgeschrittene Java-Konzepte

In den vorangegangenen Kapiteln wurden bereits einige fortgeschrittene Konzepte im Kontext ihrer Anwendung erwähnt. Hierbei wurden insbesondere Records als HashMap-Keys (Kapitel 2.2.2), Callbacks mit Runnable (Kapitel 2.2.4) sowie Platform.runLater() für Thread-sichere GUI-Updates (Kapitel 2.2.4) thematisiert. Das folgende Kapitel ergänzt diese Erläuterungen um weitere im Projekt verwendete Java-Konzepte, die über den Umfang von PS2 hinausgehen.

### 2.3.1 Lambda-Ausdrücke und funktionale Interfaces

Lambda-Ausdrücke ermöglichen es Funktionen als Parameter zu übergeben, ohne explizit eine Klasse oder anonyme innere Klasse definieren zu müssen. Die allgemeine Syntax lautet (Parameter) -> Ausdruck bzw. (Parameter) -> { Anweisungen }. Im Projekt werden Lambda-Ausdrücke an zahlreichen Stellen eingesetzt, beispielsweise bei der Übergabe von Callbacks an die GUI:

```
gui.showScoringView(current, combinations, () -> {
    advanceToNextPlayer();
    updateGUI();
});
```

Der Ausdruck () -> { ... } definiert eine parameterlose Funktion, die später von der GUI aufgerufen wird. Das funktionale Interface Runnable mit seiner einzigen abstrakten Methode run() dient dabei als Typ für den Parameter. Lambda-Ausdrücke reduzieren den Code erheblich, da die Alternative eine anonyme innere Klasse mit deutlich mehr Boilerplate-Code wäre.

### 2.3.2 Streams API

Die Streams API ermöglicht eine funktionale Verarbeitung von Collections und Arrays. Statt imperativer Schleifen mit expliziter Zustandsverwaltung werden Daten durch eine Pipeline von Operationen transformiert. Im Projekt wird die Streams API insbesondere bei der Siegerermittlung in der Methode determineWinner() eingesetzt:

```
int maxScore = Arrays.stream(players)
    .mapToInt(Player::getScore)
    .max()
    .orElse(0);

List<Player> winners = Arrays.stream(players)
    .filter(p -> p.getScore() == maxScore)
    .toList();
```

Die erste Pipeline erzeugt aus dem Spieler-Array einen Stream, transformiert jeden Spieler zu seinem Punktestand mittels `mapToInt()`, ermittelt das Maximum und liefert bei leerem Stream den Standardwert 0. Die zweite Pipeline filtert alle Spieler mit dem Höchstpunktestand und sammelt sie in einer Liste. Diese deklarative Schreibweise macht die Absicht des Codes unmittelbar ersichtlich, ohne dass Schleifenvariablen oder Zwischenergebnisse manuell verwaltet werden müssen.

### 2.3.3 PauseTransition für verzögerte Aktionen

In JavaFX-Anwendungen darf `Thread.sleep()` nicht verwendet werden, da dies den GUI-Thread blockieren und die gesamte Benutzeroberfläche einfrieren würde. Für zeitverzögerte Aktionen stellt JavaFX die Klasse `PauseTransition` bereit, die eine nicht-blockierende Pause ermöglicht:

```
PauseTransition pause = new PauseTransition(Duration.seconds(0.5));
pause.setOnFinished(event -> onComplete.run());
pause.play();
```

Diese Technik wird im Projekt beim Spielerwechsel eingesetzt, um dem Benutzer eine kurze visuelle Pause zwischen den Zügen zu geben. Die Animation läuft im Hintergrund, während die GUI reaktionsfähig bleibt. Nach Ablauf der 0,5 Sekunden wird der übergebene Callback ausgeführt, der den nächsten Spieler aktiviert. `PauseTransition` ist damit das JavaFX-Äquivalent zu einem Timer, integriert sich aber nahtlos in das Animation-Framework und garantiert Thread-Sicherheit.

## 2.4 Algorithmen

Zu komplexeren Algorithmen wir in dieser Aufgabe die Kombinationserkennung gezählt. Da diese Umsetzung für mich keine gerade Linie bedeutete, sondern viele Aspekte zur Abwägung herangezogen und bewertet werden mussten, empfand ich es für sinnvoller, diesen Abschnitt in Kapitel 2.2.1 „Kombinationserkennung“ im Kontext von Problemanalyse und Realisation anzusiedeln.

## 2.5 Programmorganisationsplan

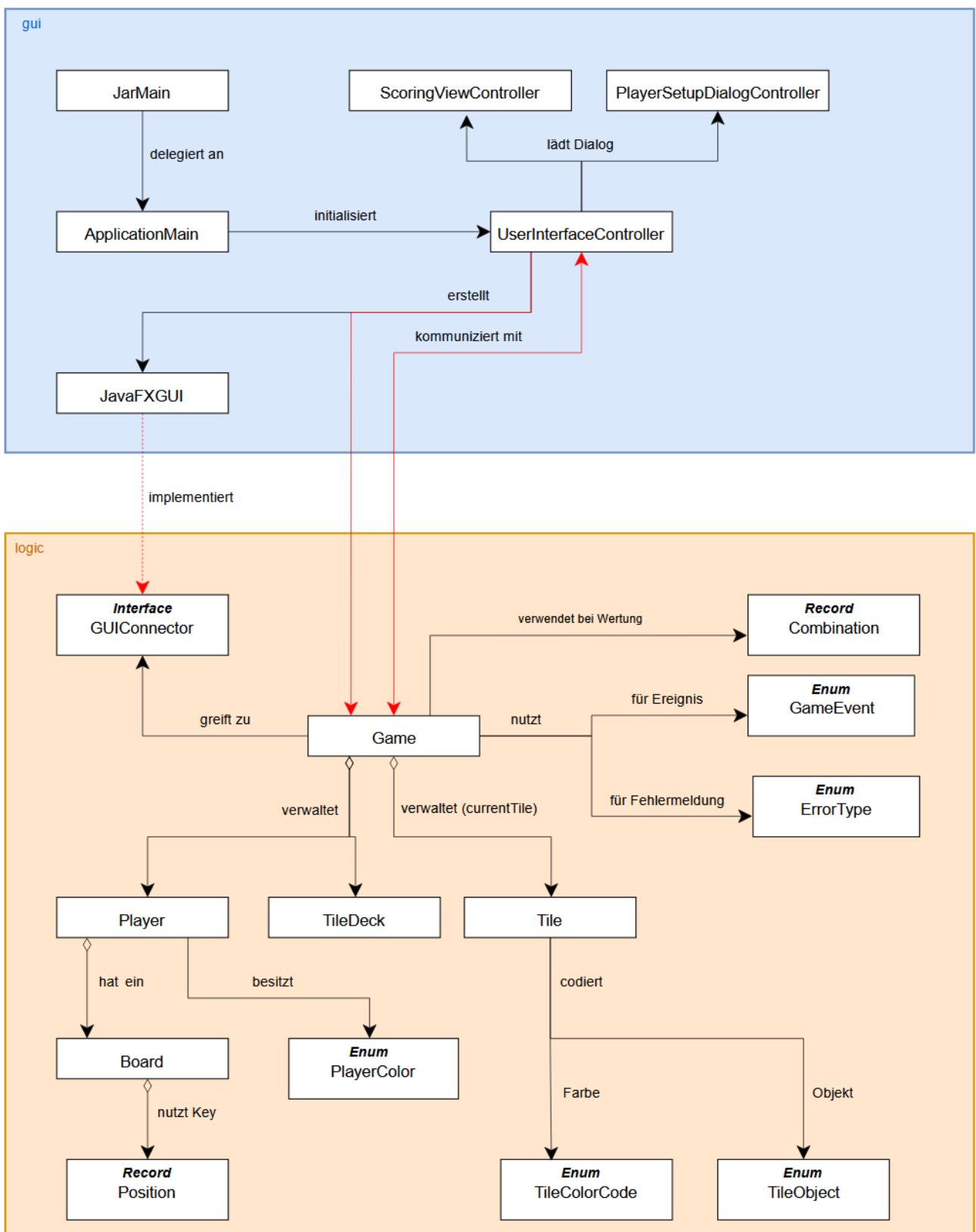


Abbildung 2.4: Programmorganisationsplan erstellt mit draw.io

Die Abbildung 2.4 zeigt eine Auflistung aller wichtigen Klassen des Projekts Mops Royal, wobei diese grundlegend in GUI und Logik aufgeteilt sind. Die GUI kommuniziert dabei nur über eine Game-Instanz mit der Logik. Auf der anderen Seite besitzt die Game-Klasse keinen direkten Zugriff auf Klassen der GUI, sondern kommuniziert mit dieser nur über das GUIConnector-Interface. In der GUI befinden sich neben dem UserInterfaceController noch die Klassen JarMain und ApplicationMain, welche für das Ausführen der Anwendung verantwortlich sind. Über die Klasse JavaFXGUI, die das GUIConnector-Interface implementiert, können Änderungen an der Benutzeroberfläche vorgenommen werden. Zusätzlich nutzt die GUI noch spezialisierte Dialog-Controller wie PlayerSetupDialogController und ScoringViewController, welche für die Spieler-Konfiguration beziehungsweise die Wertung von Kombinationen zuständig sind.

Die Logik enthält alle Klassen, die für das Spielgeschehen notwendig sind. Die Game-Klasse bildet dabei das Herzstück und verwaltet den Spielzustand. Sie speichert das aktuelle Plättchen, einen zentralen Kartenstapel (TileDeck) sowie ein Array der teilnehmenden Spieler. Jeder Spieler besitzt sein eigenes Spielfeld (Board), auf welchem die gelegten Plättchen mittels einer HashMap gespeichert werden. Als HashMap-Keys werden Position-Records verwendet, die Zeilen- und Spaltenkoordinaten definieren. Dadurch ermöglicht das System flexible Platzierung ohne feste Grenzen. Jedes Plättchen (Tile) wird durch eine Farbe und ein Objekt definiert, die durch die Enums TileColorCode beziehungsweise TileObject kodiert werden. Für die Spielmechanik werden zusätzlich die Klasse Combination sowie die Enums GameEvent und ErrorType benötigt. Combination repräsentiert eine gefundene Kombination von Plättchen mit ihren umzudrehenden Positionen. GameEvent wird genutzt, um wichtige Spielereignisse wie Spieler-Skips oder Nachholrunden zu signalisieren, während ErrorType für nutzerfreundliche Fehlermeldungen verwendet wird. Jeder Spieler hat zusätzlich eine PlayerColor, die für die visuelle Unterscheidung in der Benutzeroberfläche dient. Die strikte Trennung von GUI und Logik wird durch das GUIConnector-Interface gewährleistet: Game kennt nur dieses Interface und weiß nicht, dass JavaFXGUI es implementiert. Dies ermöglicht eine vollständige Entkopplung und macht die Logik austauschbar gegenüber verschiedenen UI-Implementierungen.

Neben den in Abbildung 2.4 dargestellten Klassen enthält das Projekt weitere Hilfsklassen, die aus Gründen der Übersichtlichkeit nicht im Diagramm aufgeführt sind. Die Klasse CombinationFinder kapselt die gesamte Logik zur Erkennung gültiger Kombinationen und wird von Game sowie Board verwendet. Für die Persistenz sind SaveGameReader und SaveGameWriter zuständig, welche das Laden und Speichern von Spielständen im JSON-Format übernehmen. Die Klasse Direction stellt Konstanten für Richtungsberechnungen bereit (orthogonal, diagonal), während InfoType analog zu ErrorType für Erfolgsmeldungen verwendet wird. Auf GUI-Seite existieren zusätzlich TileImageHelper für die Generierung von Bilddateinamen, PlayerSetupValidator für die Validierung der Spielerkonfiguration sowie HelpWindow für die Anzeige der Hilfe-Dialoge.

## 2.6 Dateien

Das Programm nutzt ausschließlich JSON-Dateien zur Verwaltung von Spielständen. Diese werden vom Benutzer über das Menü „Datei → Spielstand speichern/laden“ erstellt und geladen.

### 2.6.1 Spielstandsdateien (JSON)

Die genaue Struktur und Verwendung von Spielstandsdateien ist bereits im Benutzerhandbuch (siehe Kapitel 1.3.8) dokumentiert. Aus Entwicklersicht relevant sind folgende zwei Klassen:

**SaveGameWriter:** Erzeugt das JSON manuell mit StringBuilder. Das hat den Vorteil, dass man volle Kontrolle über das Format hat und sichergehen kann, dass es den Anforderungen der Aufgabenstellung entspricht.

**SaveGameReader:** Parsed die JSON-Datei mit Gson und validiert alle Eingaben gründlich. Die Validierungslogik ist in Kapitel 2.2.6 beschrieben.

Die Dateien werden vom Benutzer selbst verwaltet und können von jedem beliebigen Ort im Dateisystem geladen werden.

## 2.7 Programmtests

Im Folgenden werden Tests der Benutzeroberfläche aufgelistet. Dabei wird in jedem Test eine definierte Situation beschrieben, die ein genau definiertes Ergebnis erwartet. Ist das tatsächliche Ergebnis abweichend von dem erwarteten Ergebnis, so wird dies in der dritten Spalte „Abweichung“ angemerkt. Ist dies nicht der Fall, so bleibt die dritte Spalte leer.

Die teils benötigten Testdateien sind unter ppws\_21 / files hinterlegt.

### 2.7.1 Spielstart

Nr.	Testfall	Erwartetes Ergebnis
1	<b>Programmstart:</b> Das Programm wird ohne Parameter gestartet.	Ein Fenster öffnet sich mit ca. 1450×710 Pixel. Der Dialog "Neues Spiel" wird sofort angezeigt mit 4 Checkboxen für Spieler (alle aktiviert). Das Spielfeld im Hintergrund ist leer und zentriert, Zoom zeigt ca. 3-4 Reihen/Spalten.
2	<b>Im Dialog "Neues Spiel":</b> Nur Spieler 1 aktivieren (Checkbox von Spieler 2 deaktivieren).	Nicht möglich; Checkbox ist ausgegraut und nicht benutzbar
3	<b>Im Dialog "Neues Spiel":</b> 2 Spieler aktivieren. Spieler 1: "Alice", Spieler 2: "Alice" (identisch). OK-Button klicken.	Fehlerdialog: "Namen müssen eindeutig sein". Dialog bleibt offen.
4	<b>Im Dialog "Neues Spiel":</b> 2 Spieler aktivieren. Spieler 1: "Alice", Spieler 2: "Bob". OK-Button klicken.	Dialog schließt sich. Spielfeld wird angezeigt: <ol style="list-style-type: none"><li>Label "Am Zug" oben links: "Alice" (blau hervorgehoben)</li><li>Punkte-Anzeige: "Alice: 0", "Bob: 0"</li><li>Nächstes Tile wird angezeigt (zufälliges Plättchen)</li><li>Mini-Boards rechts für Alice und Bob sichtbar (inkl. Startplättchen)</li><li>Vier valide Anlegepositionen werden orthogonal um das Startplättchen herum in hellgrün dargestellt</li><li>Main Board ist in Spielerfarbe umrandet (Blau)</li><li>Speicher-Button ist aktiv (nicht ausgegraut)</li></ol>
5	<b>Im Dialog "Neues Spiel":</b> 2 Spieler aktivieren. Spieler 1: "Alice", Spieler 2: (leer). OK-Button klicken.	Eingaben sind valide; Spieler 2 wird der default Name "Spieler 2" zugewiesen
6	<b>Vorbereitung:</b> Spiel läuft (nach Testfall 5), Alice ist aktiv.	Das Tile wird platziert. Highlights verschwinden. Nach Pause wechselt zu Bob (Label wird grün hervorgehoben). Nächstes Tile wird aktualisiert.

	<p><b>Aktion:</b>          Auf das grüne Highlight-Feld neben dem Startplättchen klicken. 0,5 Sekunden warten.</p>	Alice's Mini-Board zeigt 2 Tiles. Punkte ändern sich nicht.
7	<p><b>Vorbereitung:</b> Spiel läuft, Highlights sind sichtbar.   <b>Aktion:</b> Auf ein Feld klicken, das nicht grün hervorgehoben ist.</p>	Fehlerdialog: "Ungültige Position! Bitte wähle eine markierte Position." Kein Tile wird platziert. Spieler wechselt nicht.
8	<p><b>Aktion:</b>          Menü "Datei" → "Laden". Datei test_3er_combo.json auswählen. Button "Öffnen" klicken.</p>	Info-Dialog: "Spiel geladen: test_3er_combo.json". Spielfeld wird mit geladenen Daten angezeigt. Alice's Board zeigt 2 Tiles. Aktiver Spieler ist Alice.
9	<p><b>Vorbereitung:</b>          Datei test_3er_combo.json laden (Testfall 8).   <b>Aktion:</b>          Auf markiertes Feld klicken um nächstes Tile zu platzieren.</p>	Wertungs-Overlay öffnet sich auf der rechten Seite. Das Overlay zeigt eine Wertungsübersicht. Das mittig liegende Tile steht zur potenziellen Wertung zu Verfügung und wird rot hervorgehoben. Darüber hinaus werden zwei Buttons abgebildet: "Kombination werten" (ausgegraut, bis Auswahl getroffen wurde) und "Keine Wertung vornehmen".
10	<p><b>Vorbereitung:</b>          Testfall 8 durchgeführt, Wertungsansicht offen.   <b>Aktion:</b>          Box des umdrehbaren Tiles wird aktiviert.          Button "Kombination werten" klicken.</p>	Das ausgewählte Tile wird im Wertungsfenster zunächst grün markiert und im Falle der Wertung auf dem Main Board umgedreht (Rückseite sichtbar). Punkte werden vergeben: "Alice: 2" wird angezeigt. Wertungsansicht schließt sich. Nach 0,5 Sekunden wird Bob aktiver Spieler. Nächstes Tile wird aktualisiert.
11	<p><b>Vorbereitung:</b>          Neues Spiel mit 3er-Kombi starten (wie Testfall 8), Wertungsansicht offen.   <b>Aktion:</b>          Button "Überspringen" klicken.</p>	Keine Punkte werden vergeben (Label bleibt "Alice: 0"). Keine Tiles werden umgedreht. Wertungsansicht schließt sich. Nach Pause wird Bob aktiver Spieler.
12	<p><b>Vorbereitung:</b>          Datei test_start_card_drawn.json laden (Alice's Startkarte = nächstes Tile = Grüner Napf). Alice ist aktiv.   <b>Aktion:</b>          Auf ein Highlight-Feld klicken.</p>	Dialog: "Alice muss aussetzen und erhält eine Nachholrunde! (Eigene Startkarte gezogen)". Tile wird nicht platziert. Spieler wechselt zu Bob. Alice wird intern in Nachholqueue eingefügt.
13	<p><b>Vorbereitung:</b></p>	Der FileChooser öffnet sich. ES ist nun möglich einen Speicherort und ein Dateiname im

	<p>Spiel läuft normal (Alice hat gerade Zug beendet, keine Wertung aktiv).</p> <p><b>Aktion:</b>          Menü "Datei" → "Speichern". FileChooser öffnet sich. Dateiname eingeben: "mein_spiel.json". Button "Speichern" klicken.</p>	<p>Format "MopsRoyal_YYYYMMDD_HHMMSS" abzuspeichern.          Bei erfolgreicher Speicherung öffnet sich ein Info-Dialog: "Spiel gespeichert: mein_spiel.json". Datei wird im Verzeichnis erstellt. Spiel läuft ohne Unterbrechung weiter.</p>
14	<p><b>Vorbereitung:</b>          Spiel mit offener Wertungsansicht (Testfall 8).</p> <p><b>Aktion:</b>          Menü "Datei" → "Speichern" klicken.</p>	<p>Fehler-Dialog: "Speichern während der Wertung ist nicht möglich!          Bitte schließe erst die Wertungsansicht ab."</p>
15	<p><b>Aktion:</b>          Menü "Datei" → "Laden". Datei test_only_1_player.json auswählen (nur 1 Spieler). Button "Öffnen" klicken.</p>	<p>Fehler-Dialog: "Ungültige Spielstandsdatei: Ungültige Spieleranzahl: 1 (erwartet: 2-4)"</p>
16	<p><b>Aktion:</b>          Menü "Datei" → "Laden". Datei test_invalid_tile_code.json auswählen (Code außerhalb 1-6). Button "Öffnen" klicken.</p>	<p>Fehler-Dialog: "Ungültige Spielstandsdatei: Spieler 0 Position (0,0): Ungültiger Code 0"</p>
17	<p><b>Vorbereitung:</b>          Lade die Datei "test_determine_winner.json"          2er Spiel Alice vs. Bob, beide haben 25 Tiles gelegt (Alice: 18 Pkt, Bob: 16 Pkt).</p> <p><b>Beobachtung:</b>          Spielende wird automatisch erkannt.</p>	<p>Gewinner-Dialog öffnet sich:          "🏆 Spiel beendet! 🏆".          Zeigt Platzierung:          1. Alice - 18 Pkt.          2. Bob - 16 Pkt.</p>
18	<p><b>Vorbereitung:</b>          Lade die Datei "test_tie_breaker.json".          2er-Spiel mit Alice: 18 Pkt./7 Tiles umgedreht, Bob: 18 Pkt./0 Tile umgedreht. Beide 25 Tiles gelegt.</p>	<p>Gewinner-Dialog öffnet sich:          "🏆 Spiel beendet! 🏆".          Zeigt Platzierung:          1. Bob - 18 Pkt. – 0 umgedreht (Beispiel Szenario)          2. Alice - 18 Pkt. – 7 umgedreht</p>
19	<p><b>Vorbereitung:</b>          Spiel läuft (beide haben min. 1 Zug gemacht).</p> <p><b>Aktion:</b>          Menü "Datei" → "Neues Spiel" klicken. Bei Dialog "Aktuellen Spielstand speichern?" Button "Nicht speichern" klicken.</p>	<p>Dialog: "Neues Spiel" öffnet sich (Spieler-Setup). Können 2-4 neue Spieler mit neuen Namen eingeben. Altes Spiel wird verworfen.</p>

20	<b>Vorbereitung:</b> Spiel läuft.  <b>Aktion:</b> X-Button des Fensters klicken. Button "Nicht speichern" wählen.	Dialog: "Spiel beenden - Möchten Sie das Spiel vor dem Beenden speichern?" mit 3 Buttons ("Speichern", "Nicht speichern", "Abbrechen"). Bei "Nicht speichern": Programm wird sofort beendet.
21	<b>Vorbereitung:</b> Spiel läuft. Cursor über Spielfeld.  <b>Aktion 1:</b> Mausrad nach OBEN drehen (Zoom In), mehrmals.  <b>Aktion 2:</b> Mausrad nach UNTEN drehen (Zoom Out), mehrmals.	Spielfeld wird größer. Weniger Reihen/Spalten sichtbar. Bei mehrfachem Drehen stoppt beim Maximum (ca. 3.0x).  Spielfeld wird kleiner. Mehr Reihen/Spalten sichtbar. Bei mehrfachem Drehen stoppt beim Minimum (ca. 0.38x).
22	<b>Vorbereitung:</b> Spiel läuft mit Zoom $\geq 1.0$ . Cursor über Spielfeld.  <b>Aktion:</b> Klicken und Dragging nach links (mehrere Zentimeter).	Sichtbarer Bereich verschiebt sich. Neue Felder werden sichtbar je nach Drag-Richtung. Spielfeld bleibt in ScrollPane-Grenzen.
23	<b>Aktion:</b> Menü "Hilfe" → "Bedienung" klicken.	Dialog öffnet sich mit Bedienungsanleitung (Zoom, Klick, Regeln, Punkte, etc.).
24	<b>Aktion:</b> Menü "Hilfe" → "Kombinationen" klicken.	Fenster öffnet sich mit Bild der gültigen Kombinationen (alle Formen mit 3-5 Plättchen).

## Anhang I: Kombinationsübersicht



**Abbildung I.1.** Alle möglichen Kombinationen, die sich legal (siehe Vorschriften: Kapitel 1.3.4) bilden lassen