

Fachhochschule Wedel
University of applied sciences

Dokumentation Programmierpraktikum SS22 „Crosswise“

Vorgelegt von:

Name: Jonathan El Jusup

Matrikelnummer: cgt104707

Kurs: Programmierpraktikum SS22

Fachsemester: 5

Studiengang: Computer Games Technology

Abgabe: 12.08.22

Dozent: Gerit Kaleck

Inhaltsverzeichnis

1.	Benutzerhandbuch	1
1.1.	Ablaufbedingungen	1
1.2.	Programminstallation/ Programmstart.....	1
1.3.	Bedienungsanleitung.....	1
1.3.1.	Spieloberfläche.....	1
1.3.2.	Das Spiel Crosswise	4
1.3.3.	Spielablauf	4
1.3.4.	Spielende und Punktebewertung.....	6
1.4.	Fehlermeldungen	7
1.4.1.	Beim Erstellen eines neuen Spiels	7
1.4.2.	Im Spielverlauf	8
1.4.3.	Beim Laden/ Speichern.....	8
2.	Programmiererhandbuch	9
2.1.	Entwicklungskonfiguration	9
2.2.	Problemanalyse & Realisation.....	9
2.2.1.	GUI-Erstellung der Benutzeroberfläche	9
2.2.2.	GUI-Darstellung der Benutzeroberfläche	11
2.2.3.	GUI-Darstellung des Spielfelds	13
2.2.4.	GUI-Darstellung der Spielerhände.....	15
2.2.5.	GUI-Darstellung der Menüleiste.....	18
2.2.6.	GUI-Darstellung der optionalen Punkteleisten	19
2.2.7.	GUI-Darstellung der Team-Informationsbereiche.....	22
2.2.8.	GUI-Darstellung der genutzten Wildcards	23
2.2.9.	GUI-Darstellung der Punktetabelle	25
2.2.10.	GUI-Darstellung des „New Game“ Menüs	27
2.2.11.	GUI-Darstellung des „Game Over“ Menüs	30
2.2.12.	Implementierung der Spielstein Tasche	33
2.2.13.	Implementierung der Spielerzüge	34
2.2.14.	Implementierung der KI	36
2.2.15.	Implementierung des Spielstandes	37
2.2.16.	Implementierung des Logs	39
2.3.	Algorithmen.....	40
2.3.1.	Spielfeld Evaluation	40
2.3.2.	KI Spielzug Entscheidung	41
2.3.3.	KI Zug Animation	44

2.4.	Programmorganisationsplan	46
2.4.1.	Erläuterungen des Programmorganisationsplans	47
2.4.2.	Erläuterungen zum Paket „gui“	47
2.4.3.	Erläuterungen zum Paket „logic“	47
2.5.	Dateien	48
2.5.1.	Spielstand Datei.....	48
2.5.2.	Log Datei.....	48
2.6.	Programmtests	49
2.6.1.	Menü Tests	49
2.6.2.	„New Game“ Menü Tests	50
2.6.3.	Spielverlauf Tests.....	50
2.6.4.	„Game Over“ Menü Tests.....	51

1. Benutzerhandbuch

1.1. Ablaufbedingungen

Um die kompilierte Version des Programms ausführen zu können, sollte der Benutzer mindestens das Betriebssystem „Windows 10“ auf seinem Computer installiert haben. Die installierte Java Version muss mindestens der „Version 8 Update 331“ entsprechen. Wenn diese Vorbedingungen erfüllt sind, sollte das kompilierte Programm problemlos starten können.

1.2. Programminstallation/ Programmstart

Das Programm liegt bereits als .jar Datei vor und muss demnach nicht installiert werden. Durch Öffnen der „pp_Crosswise_Jusup-1.0-SNAPSHOT.jar“ lässt sich das Programm starten. Alternativ lässt sich das Programm auch in der Konsole durch folgenden Befehl öffnen: „`java -jar pp_Crosswise_Jusup-1.0-SNAPSHOT.jar`“. Hierbei gilt die Voraussetzung, dass sich der Nutzer bereits im Verzeichnis des Programms befinden muss.

1.3. Bedienungsanleitung

1.3.1. Spielfläche

Wenn der Nutzer die Anwendung startet, wird ihm folgende Nutzeroberfläche angezeigt. Sowohl das Spielfeld als auch die Spielerhände werden mit zufälligen Spielsteinen belegt, um die Oberfläche nicht leer erscheinen zu lassen. Dies dient allein visuellen Zwecken. Der Spieler kann währenddessen nicht mit dem Spielfeld oder den Spielerhänden interagieren. Folgend werden alle Elemente der Nutzeroberfläche näher erläutert.

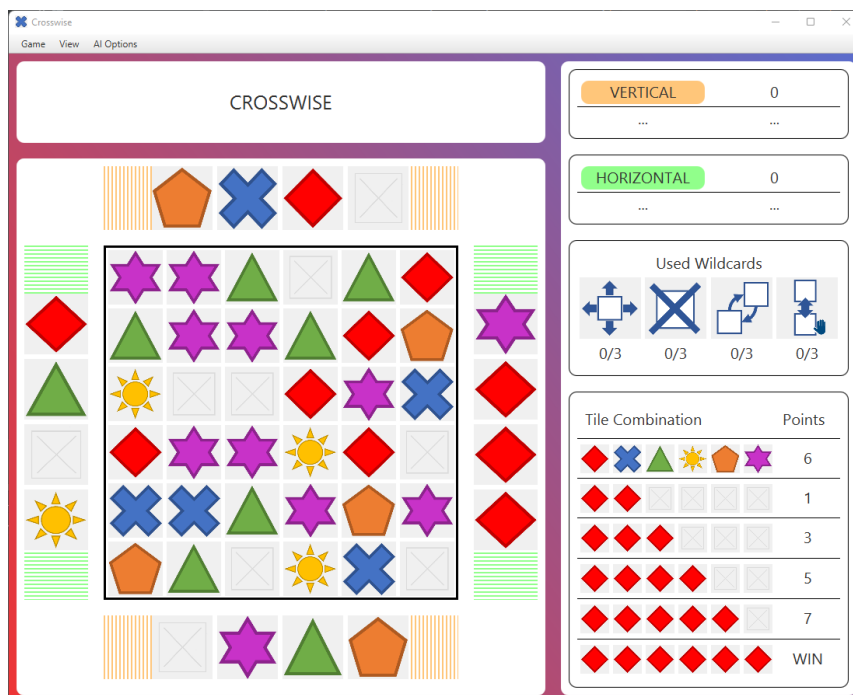


Abbildung 1: Benutzeroberfläche

1.3.1.1. Spielbereich

Auf der linken unteren Seite befinden sich das Spielfeld und die Spielerhände. In diesem Bereich findet die hauptsächliche Interaktion statt. Von dort aus werden Spielsteine auf und vom Spielfeld gezogen. Zu jeder Zeit wird nur die Hand des Spielers angezeigt, welcher gerade am Zug ist; alle anderen Spielerhände werden ausgeblendet. Zu sehen ist, dass die obere und untere Spielerhand mit orangen vertikalen Balken gefärbt ist. Dies symbolisiert, dass diese im orangen vertikalen Team sind. Analog dazu sind die linke und rechte Spielerhand mit grünen horizontalen Balken gefärbt. Dies symbolisiert, dass diese Spielerhände zum grünen horizontalen Team gehören.

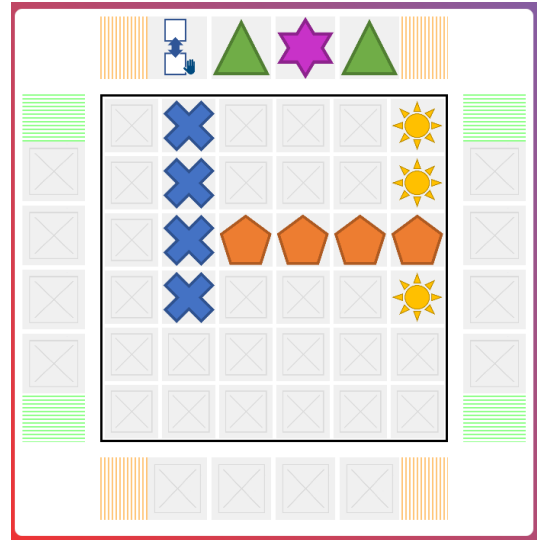


Abbildung 2: Spielbereich

1.3.1.2. Informationsanzeige

Darüber befindet sich eine textuelle Informationsanzeige, welche die Spieler mit Informationen zum aktuellen Spielverlauf anzeigt. Angezeigt wird unter anderem, welcher Spieler gerade am Zug ist, wie welche Wildcard gespielt werden kann, ob ein Spielerzug vollzogen werden kann und welches Team das Spiel gewonnen hat.

1.3.1.3. Seiteninformationen (Team Informations-Bereiche, genutzte Wildcards, Punktetabelle)

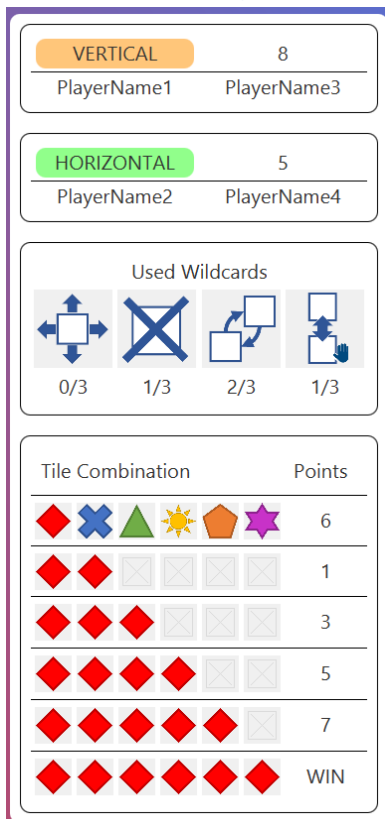


Abbildung 3: Seitlicher Informations-Container

Auf der rechten Seite befinden sich alle Informationen, die zum Spiel benötigt werden.

Angefangen mit den **Team-Informationsbereichen** beider Teams. Sobald das Spiel startet, werden hier die Spielernamen der beiden Teams angezeigt, passend dazu der Name des Teams und die korrespondierende Farbe. Optional lässt sich die Anzeige für die Teampunkte ein- oder ausblenden. Standardmäßig ist diese eingeschaltet.

Darunter befindet sich die Anzeige der **genutzten Wildcards**. Diese zeigt an, welche Wildcards und wie viele davon im Spielverlauf genutzt worden sind und welche noch genutzt werden können.

Das letzte Element ist die **optionale Punktetabelle**, welche zu jeder Kombination von Standard Spielsteinen die korrespondierende Punktzahl auflistet. Diese Tabelle kann optional ein- oder ausgeschaltet werden. Hierbei sind die Kombinationen, die nur rote Quadrate enthalten so zu verstehen, dass nur gleiche Spielsteine in einer Reihe gelegt werden müssen. Es müssen nicht notwendigerweise rote Quadrate sein. Alle Standard Spielsteine können in solchen Kombinationen die gleichen Punkte erzielen. Außerdem ist es möglich, dass mehrere Kombinationen pro Spalte bzw. Zeile gelegt werden können.

1.3.1.4. Menüleiste

Ganz oben des Anwendungsfensters befindet sich eine Menüleiste mit 3 Untermenüs: „Game“, „View“ und „AI Options“. Das erste Menü „Game“ ist zum Starten, Speichern, Laden und Beenden des Spiels zuständig. Zum Start der Anwendung kann kein Spiel gespeichert werden, da noch keines gestartet worden ist. Geladen kann prinzipiell jedes mögliche Spiel, ob mit 2 oder 4 Spielern, mit oder ohne KI Spielern, ob es ein laufendes oder schon bereits beendetes Spiel ist. Jede Konstellation soll hier möglich sein. Die zu ladende Spieldatei muss dabei eine gültige .json Datei sein. Gespeichert wird ebenfalls mit .json Dateien. Beim Speichern und Laden öffnet sich jeweils ein neues Auswahl-Fenster, in welchem der Nutzer den Pfad der zu Ladenden bzw. speichernden Datei auswählen kann.

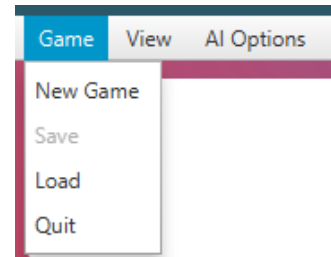


Abbildung 4: "Game" Menü

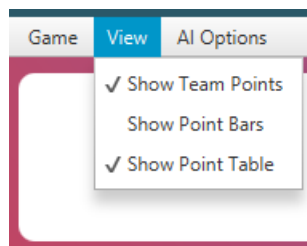


Abbildung 5: "View" Menü

Im Menü „View“ kann der Nutzer optionale Anzeigen ein- oder ausblenden. Optionale Anzeigen sind die Teampunkte im Team Informations-Bereich, die Punkteleisten am Spielfeld und die Punktetabelle an der Seite. Dabei werden die Teampunkte im Team Informations-Bereich und die Punktetabelle standardmäßig eingeblendet. Die Punkteleiste auf dem Spielbrett wird standardmäßig ausgeblendet. Vor oder nach Spielstart können diese optionalen Anzeigen ein- oder ausgeblendet werden.

Team Punkte		
VERTICAL	9	
PlayerName1	PlayerName3	
HORIZONTAL	11	
PlayerName2	PlayerName4	

Abbildung 6: Optionale Teampunkte

Punkte Leisten									
1	2	3	0	3	0				

Abbildung 7: Optionale Punkteleisten

Punktetabelle						
Tile Combination						Points
♦	✕	▲	☀	⬢	★	6
♦	♦	☐	☐	☐	☐	1
♦	♦	♦	☐	☐	☐	3
♦	♦	♦	♦	☐	☐	5
♦	♦	♦	♦	♦	☐	7
♦	♦	♦	♦	♦	♦	WIN

Abbildung 8: Optionale Punktetabelle

Das letzte Menü „AI Options“ bezieht sich auf jegliche KI Optionen. Darunter kann der Nutzer die KI Spielzug Animation überspringen, die KI Spielerhand anzeigen lassen, wenn diese am Zug ist und die KI Spielzug Animation einstellen. Standardmäßig wird die KI Spielerhand ausgeblendet. Der Nutzer kann sich zwischen 3 Animationsgeschwindigkeiten entscheiden; von kurz, mittel bis lang. Standardmäßig ist die kurze Animationsdauer ausgewählt.

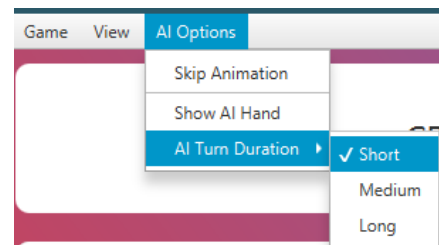


Abbildung 9: "AI Options" Menü

1.3.2. Das Spiel Crosswise

Crosswise ist ein Brettspiel, welches durch strategischen Platzieren von Spielsteinen auf einem Spielbrett gespielt wird. Ziel ist es, durch Kombinationen von Spielsteinen in Zeilen bzw. Spalten die meisten Punkte zu erzielen. Dabei müssen die Spieler in alle Richtungen schauen, denn jeder Spielzug wirkt sich auf die Punktzahl des eigenen Teams, aber auch auf die des gegnerischen Teams aus. Gespielt wird mit 2 oder 4 Spielern, wobei die Spieler eines Teams immer gegenüber voneinander sitzen und Punkte in ihren Spalten sammeln müssen.

1.3.3. Spielablauf

Gespielt wird mit 2 oder 4 Spielern in 2 Teams auf einem 6 x 6 Spielfeld. Die Spieler eines Teams sitzen immer gegenüber. Bei 2 Spielern sitzen diese 90° gedreht voneinander. Die obere (und untere) Spielerhand gehören zum orangen vertikalen Team; die linke (und rechte) Hand gehören zum grünen horizontalen Team. Das vertikale Team muss Spielsteinkombinationen in Spalten des Spielfeldes legen, das horizontale Team muss Spielsteinkombinationen in Zeilen des Spielfeldes legen. Der oberste Spieler macht immer den ersten Zug. Die nächsten Spieler kommen nacheinander im Uhrzeigersinn zum Zug. Nur die Hand des aktuellen Spielers wird angezeigt, alle anderen Hände werden derweil mit leeren Spielsteinen gefüllt angezeigt.

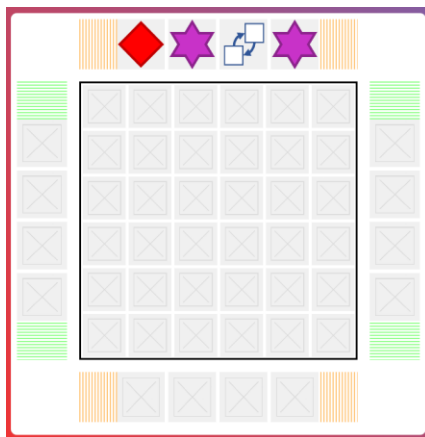


Abbildung 10: 4 Spieler Layout

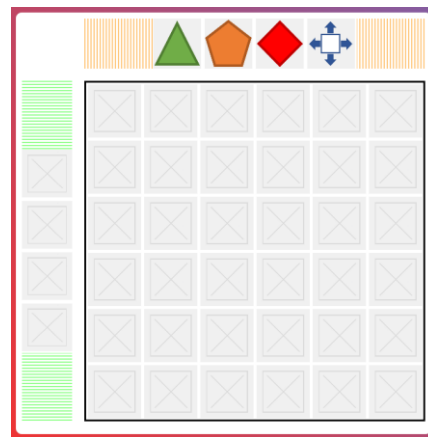


Abbildung 11: 2 Spieler Layout

Jeder Spieler erhält zu Anfang 4 zufällige Spielsteine aus der Spielsteintasche, welche nur sichtbar für ihn selbst sind. Der Reihe nach im Uhrzeigersinn zieht jeder Spieler einen beliebigen Spielstein von seiner Hand auf das Spielfeld und erhält danach einen neuen zufälligen Stein auf der Hand, sodass die Spielerhand immer gefüllt ist, solange die Spielsteintasche Spielsteine enthält. Dabei ist es Ziel des Spielers, der gerade am Zug ist, die eigenen Spielsteine so zu platzieren, dass dabei die bestmöglichen Kombinationen zustande kommen, um möglichst viele Punkte erzielen. Kombinationen werden für das vertikale Team spaltenweise und für das horizontale Team zeilenweise erzielt. Die Spalten des vertikalen Teams sind die Zeilen des horizontalen Teams und andersherum. Zu Spielende werden die Kombinationen aller Spalten des vertikalen Teams und die Kombinationen aller Zeilen des horizontalen Teams zusammengezählt. Das Team mit den meisten erzielten Punkten gewinnt das Spiel.

Es gibt 2 verschiedene Arten von Spielsteinen. Die sogenannten „**Standard Spielsteine**“ und die „**Wildcards**“. Insgesamt gibt es 6 verschiedene Standard Spielsteine mit jeweils 7 Exemplaren und 4 verschiedene Wildcards mit jeweils 3 Exemplaren, die einmalig verwendet werden können. Die Gesamtanzahl an Spielsteinen beträgt also 54.

Die Standard Spielsteine. Diese sehen wie folgt aus:

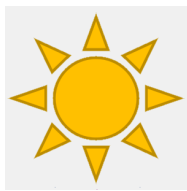


Abbildung 12:
Sun



Abbildung 13:
Cross

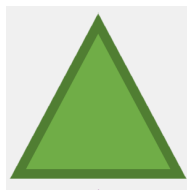


Abbildung 14:
Triangle

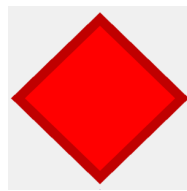


Abbildung 15:
Square

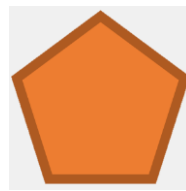


Abbildung 16:
Pentagon

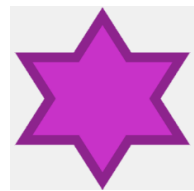


Abbildung 17:
Star

Mit diesen Standard Spielsteinen werden die Kombinationen gelegt. Um einen Standard Spielstein zu legen, zieht der Spieler einfach einen solchen Spielstein von seiner Hand per Drag & Drop auf eine leere Fläche des Spielfeldes und legt diesen dort ab. Spielfeldflächen, auf die ein Standard Spielstein gelegt werden kann, werden während des Ziehens des Spielsteins auf die betreffende Fläche zusätzlich hervorgehoben.

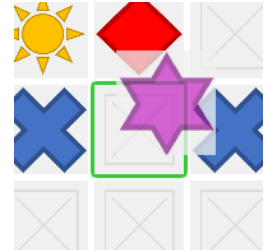


Abbildung 18: Spielfeld
Hervorhebung

Es gibt 4 verschiedene Wildcards. Von jedem Wildcard-Typ existieren jeweils 3 Exemplare, welche einmalig pro Spiel verwendet werden können. Die Anzahl der bereits verwendeten Wildcards wird im Seiteninformationsbereich angezeigt. Diese Wildcards lassen sich in 2 Typen unterscheiden: „**1-Phase-Wildcards**“, zu den der Remover gehört und „**2-Phase-Wildcards**“, zu der die restlichen Wildcards gehören. „1-Phase-Wildcards“ können wie Standard Spielsteine in einem durchgängigen Zug gespielt werden.



Abbildung 19: Komplette
Spielfeld Hervorhebung

„2-Phase-Wildcards“ hingegen müssen zuerst aktiviert werden. Dazu zieht der Spieler eine solche Wildcard zunächst auf ein beliebiges Feld auf dem Spielfeld. Dafür wird das gesamte Spielfeld farblich hervorgehoben, um dem Spieler mitzuteilen, dass die jeweilige Wildcard überall auf dem Spielfeld gelegt werden kann. Sobald dies geschehen ist, gilt diese Wildcard als aktiviert und kann genutzt werden. Dies wird dem Spieler durch die Informationsanzeige mitgeteilt. Wenn eine Wildcard aktiviert wurde, muss diese gespielt werden. Der Spieler kann sich nicht mehr anders entscheiden und einen anderen Spielstein zum Spielen auswählen. Für alle Wildcards gilt folgende Spezialregel.

Keine Wildcard kann gespielt werden, wenn das Spielfeld komplett leer ist. Dies wird dem Spieler ggf. auf der Informationsanzeige mitgeteilt.

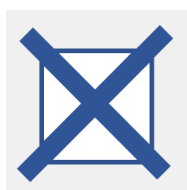


Abbildung 20:
Remover

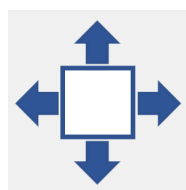


Abbildung 21:
Mover

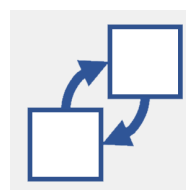


Abbildung 22:
Swap On Board

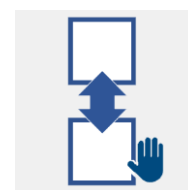


Abbildung 23:
Swap With Hand

Der „**Remover**“ ist ein Sonderstein, den man auf ein nicht-leeres Feld auf dem Spielfeld legen kann. Dadurch wird der jeweilige Spielstein vom Feld auf die Hand genommen und das Spielfeld an der Stelle ist nun leer. Der Remover ist eine „1-Phase-Wildcard“. Den Remover zieht der Spieler durch Drag & Drop auf ein nicht leeres Spielfeld. Wo der Remover platziert werden kann und wohin nicht, wird ebenfalls farblich hervorgehoben.

Der „**Mover**“ ist ein Sonderstein, den man per Drag & Drop initial überall auf dem Feld hinziehen kann. Sobald diese abgelegt worden ist, gilt diese als aktiviert. Von jetzt an kann der Spieler einen Spielstein

auf dem Spielfeld auf ein leeres Spielfeld ziehen, um diesen zu verschieben. Die alte Position ist nun leer. Der Mover ist eine „2-Phase-Wildcard“. Kein leeres-Feld kann verschoben werden.

Die „**Swap On Board**“ ist ein Sonderstein, die dem Spieler erlaubt 2 Spielsteine auf dem Spielfeld zu tauschen. Diese Wildcard ist ebenfalls eine „2-Phase-Wildcard“, muss also zuerst aktiviert werden. Danach kann der Spieler einen Spielstein auf dem Spielfeld nehmen und auf einen anderen ziehen, um diese miteinander auszutauschen. Diese Wildcard kann nicht gespielt werden, wenn nur 1 Spielstein auf dem Spielfeld liegt. Dies wird dem Spieler ggf. auf der Informationsanzeige mitgeteilt. Es können nur Standard Spielsteine miteinander getauscht werden, keine leeren Spielfelder.

Die „**Swap With Hand**“ ist ein Sonderstein, welche ähnlich wie der Remover funktioniert. Diese Wildcard erlaubt es dem Spieler, einen beliebigen Standard-Spielstein auf dem Spielfeld durch einen Standard Spielstein auf seiner Hand zu ersetzen. Dafür zieht der Spieler einen beliebigen Standard Spielstein von seiner Hand auf einen Spielstein auf dem Spielfeld, um diese miteinander zu tauschen. Der Spieler kann nicht eine Wildcard mit einem Spielstein auf dem Spielfeld tauschen. Diese Wildcard ist ebenfalls eine „2-Phase-Wildcard“ und muss zunächst aktiviert werden. Sie kann nicht gespielt werden, wenn der Spieler keinen Standard Spielstein auf der Hand besitzt, mit dem er tauschen kann. Beim Aktivieren der Wildcard erhält der Spieler einen neuen zufälligen Spielstein, noch bevor er diese fertig genutzt hat, kann also ggf. den neuen Stein, den er gerade bekommen hat, mit einem Spielstein auf dem Feld tauschen.

1.3.4. Spielende und Punktebewertung

Das Spiel endet, wenn das Spielfeld vollkommen gefüllt ist, also keine leeren Spielfelder mehr übrig sind oder wenn ein Team einen sogenannten „**Win of Sixes**“ frühzeitig erzielt hat. Um einen „Win of Sixes“ zu erzielen, muss das Team in einer seiner Spalten bzw. Zeilen eine Kombination von 6 gleichen Spielsteinen haben. Wenn ein „Win of Sixes“ erzielt wurde, hat das Team sofort gewonnen. Falls es KI Spieler gibt, kann das Spiel frühzeitig enden, wenn die KI keinen Zug ausführen kann.

Für die Punktebewertung werden Zeilen und Spalten der jeweiligen Teams folglich als Segmente bezeichnet. Die Punktebewertung kommt folgendermaßen zustande:

- 6 Punkte, wenn alle Spielsteine im Segment unterschiedlich sind
- 1 Punkt, wenn 2 gleiche Spielsteine im Segment vorliegen (**Zwilling**)
- 3 Punkte, wenn 3 gleiche Spielsteine im Segment vorliegen (**Drilling**)
- 5 Punkte, wenn 4 gleiche Spielsteine im Segment vorliegen (**Vierling**)
- 7 Punkte, wenn 5 gleiche Spielsteine im Segment vorliegen (**Fünfling**)
- „Win of Sixes“, wenn 6 gleiche Spielsteine im Segment vorliegen (**Sechsling**)





































Tile Combination	Points
     	6
     	1
     	3
     	5
     	7
     	WIN

Abbildung 24: Punktetabelle

Es ist möglich, dass mehrere Kombinationen in einem Segment erzielt werden können. So kann es beispielsweise sein, dass in einem Segment 3 Zwillinge, 2 Drillinge oder ein Zwilling und ein Vierling zustande kommen.

Zu Spielende werden alle Punkte für die jeweiligen Segmente beider Teams zusammengezählt. Das Team, welches die meisten Punkte erzielt hat, gewinnt das Spiel. Es kann auch sein, dass ein

Unentschieden erzielt wird, wenn gleich viele Punkte verdient wurden. Dies kann unter anderem auch der Fall sein, wenn beide Teams gleichzeitig ein „Win of Sixes“ erzielt haben.

Wenn das Spiel beendet wurde, öffnet sich ein neues Fenster, welches das Spielergebnis und die Punkteübersicht anzeigt. Ganz oben wird angezeigt, welches Team gewonnen hat oder ob ein Gleichstand erzielt worden ist. Darunter folgt die Übersicht beider Teams. Diese beinhaltet die Spielernamen der Teams und die erreichten Punkte. Wenn ein „Win of Sixes“ erzielt wurde, wird dies anstelle der erreichten Punkte angezeigt.

Ebenfalls wird passend zum Gewinner-Team ein gestrichelter Hintergrund in der Farbe und Richtung des Teams angezeigt. Gewinnt das vertikale Team, wird ein oranger Hintergrund mit vertikalen Balken angezeigt. Gewinnt das horizontale Team, wird ein grüner Hintergrund mit horizontalen Balken angezeigt. Bei Gleichstand werden diagonale graue Balken angezeigt.

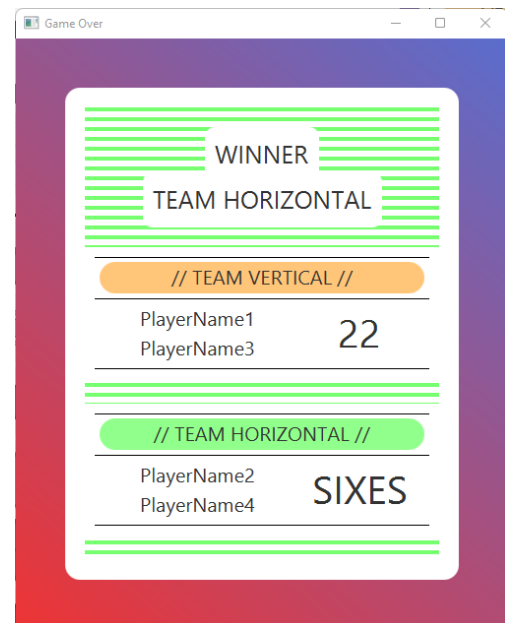


Abbildung 25: "Game Over" Menü

1.4. Fehlermeldungen

Während des Spielverlaufs, beim Laden und Speichern von Dateien und beim Starten eines neuen Spiels können Fehler auftreten. Die meisten Fehler sind auf ein Fehlverhalten des Benutzers zurückzuführen und lassen sich leicht beheben. Folgend lassen sich die auftretenden Fehlermeldungen mit den entsprechenden Ursachen und wie sie zu beheben sind finden. Diese Fehlermeldungen werden unter anderem als Alert Popups angezeigt, aber auch in Form von Informationstextfeldern.

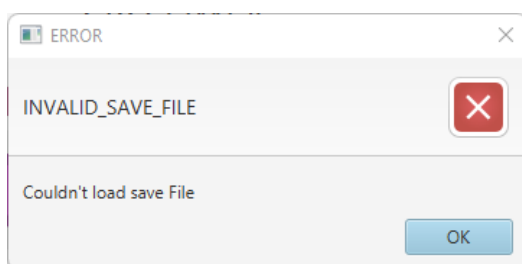


Abbildung 26: Beispiel Alert Fehlermeldung

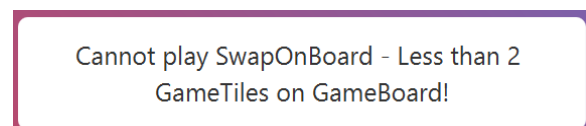


Abbildung 27: Beispiel Informationstext Fehlermeldung

1.4.1. Beim Erstellen eines neuen Spiels

Fehlermeldung	Ursache	Behebungsmaßnahme
Name(s) invalid!	Spielernamen entsprechen nicht dem Muster "[a-zA-Z\d]*" oder ist größer als 15 Zeichenlang.	Spielernamen dürfen nur Groß-/Kleinbuchstaben oder Zahlen enthalten und sind höchstens 15 Zeichen lang.
No duplicate Names allowed!	Mindestens 2 Spielernamen sind identisch.	Doppelte Spielernamen ändern, sodass alle Namen einzigartig sind.

1.4.2. Im Spielverlauf

Fehlermeldung	Ursache	Behebungsmaßnahme
Game Over - First player only contains Wildcards! Please start a new game	Zu Spielstart hat der erste Spieler zufällig nur Wildcards in seine Hand zugewiesen bekommen und kann sie nicht auf einem komplett leeren Spielfeld spielen.	Ein neues Spiel muss gestartet werden
Cannot play wildcard - Empty GameBoard!	Es wurde versucht, eine Wildcard zu spielen, aber das ganze Spielfeld war leer.	Spieler einen anderen Spielstein. Falls dies nicht möglich ist, kann das Spiel nicht weitergespielt werden und ein neues Spiel muss gestartet werden.
Cannot play SwapOnBoard - Less than 2 GameTiles on GameBoard!	Es wurde versucht, die Wildcard „SwapOnBoard“ zu spielen, aber auf dem Spielfeld liegt nur ein Spielstein.	Spieler einen anderen Spielstein. Falls dies nicht möglich ist, kann das Spiel nicht weitergespielt werden und ein neues Spiel muss gestartet werden.
Cannot play SwapWithHand - At least 1 standard GameTile in PlayerHand required!	Es wurde versucht, die Wildcard „SwapWithHand“ zu spielen, aber der Spieler hat kein Standard Spielstein auf der Hand, mit dem er tauschen kann.	Spieler einen anderen Spielstein. Falls dies nicht möglich ist, kann das Spiel nicht weitergespielt werden und ein neues Spiel muss gestartet werden.

1.4.3. Beim Laden/ Speichern

Fehlermeldung	Ursache	Behebungsmaßnahme
Couldn't load save File!	Es wurde versucht ein Spiel zu laden, aber die zu ladende Datei ist fehlerhaft oder entspricht nicht dem richtigen Format.	Lade ein neues Spiel mit einer validen Spieldatei.

2. Programmiererhandbuch

Dies ist das Programmierhandbuch zum Spiel Crosswise. Es beinhaltet die tabellarische Entwicklungskonfiguration, sämtliche Problemanalysen und Realisationen, welche den Hauptteil des Programmiererhandbuches ausmachen. Darauf folgt die Erläuterung der Algorithmen, welche besondere Stellung einnehmen oder sich als äußerst komplex erweisen. Anschließend folgen der Programmorganisationsplan, die Dateien und Programmtests der Benutzeroberfläche.

2.1. Entwicklungskonfiguration

Software Komponenten	Version
Betriebssystem: Windows	11
Tortoise SVN	Version 1.14
Java	Version 8 Update 331
IntelliJ IDEA Ultimate	2022.1.3
JDK: Eclipse Temurin	Version 17.0.2

2.2. Problemanalyse & Realisation

Folgende Analysen und Beschreibungen beziehen sich auf das Spiel Crosswise, welche als JavaFX-Anwendung umgesetzt wurde. Alle folgenden Problemstellungen lassen sich in 2 Teile aufteilen. Einerseits die Realisierung der grafischen Benutzeroberfläche und dessen Elemente, andererseits die Realisierung der Logik. Hier werden Lösungsansätze und Datenstrukturen diskutiert und erörtert.

2.2.1. GUI-Erstellung der Benutzeroberfläche

Problemanalyse

Für das Spiel Crosswise soll eine Benutzeroberfläche erstellt werden. Diese soll alle wichtigen Elemente wie das Spielfeld, Spielerhände und weitere Informationen beinhalten. Hierauf findet jede mögliche Interaktion des Spielers statt. Die Oberfläche soll visuell ansprechend sein und sich bei Größenveränderung des Fensters entsprechend anpassen können. Auf die Elemente der Benutzeroberfläche und die Darstellung wird separat eingegangen. Folgend soll der Prozess zur Erstellung der allgemeinen Benutzeroberfläche diskutiert und erörtert werden.

Realisationsanalyse

Die gesamte GUI, darunter auch Benutzeroberfläche wird in JavaFX erstellt bzw. gestaltet. Hierfür bieten sich zunächst 2 Optionen an. Die erste Option wäre, den Scene Builder für die Erstellung der Benutzeroberfläche zu nutzen. Die Alternative dazu wäre, alles via Code im User-Interface Controller zu schreiben.

Der Scene Builder bietet ein übersichtliches User-Interface mit allen nutzbaren UI-Elementen an, die genutzt werden können. Diese Elemente und dessen Eigenschaften können ebenfalls individuell eingestellt werden. Mit dem Scene Builder kann man somit die Benutzeroberfläche mit möglichst wenig selbstgeschriebenen Code erstellen. Außerdem arbeitet der Scene Builder mit einer Vorschau, mit der sich die Benutzeroberfläche schon vor Programmstart anzeigen lässt.

Jedoch kann der Scene Builder nicht alles. Es mangelt hauptsächlich an diversen feingranularen und speziellen Einstellungen, insbesondere bezüglich der Größenverhältnisse von Nodes zueinander und logisch gesetzter Parameter der Eigenschaften von Nodes. Mit dem Scene Builder ist es zum Beispiel nicht oder nur schwer über Umwege möglich, bestimmte Node Layouts von anderen Nodes abhängig zu machen. Dies ist aber notwendig für die Responsivität der Größen- und Layout-Anpassung der

Benutzeroberfläche, wenn sich die Fenstergröße ändert. Ebenso muss die Benutzeroberfläche teilweise durch Logik verändert werden können. Demnach müsste man die jeweiligen Elemente im Scene Builder erstellen, im Code referenzieren und diese dann im Controller modifizieren. Dies ist möglich und auch sinnvoll, jedoch teilt sich dadurch die Erstellung der Benutzeroberfläche an willkürlichen Stellen in zwei Bereiche auf. Einerseits wäre diese im Scene Builder in der .fxml Datei definiert und andererseits würde diese durch den Controller nochmal verändert werden. Dies kann zur Unübersichtlichkeit führen und das soll bei einer Benutzeroberfläche mit vielen Elementen so gut wie möglich vermieden werden.

Ein weiterer Nachteil wäre, dass die Vorschau des Scene Builders nicht zwingend die korrekte Benutzeroberfläche anzeigen würde, da bestimmte Element darin durch den Controller beim Start der Anwendung verändert werden könnten. Ebenfalls ist bekannt, dass der Scene Builder in seltenen Fällen die platzierten bzw. modifizierten Elemente nicht immer sofort in die .fxml Datei übernimmt. Dies kann während der Entwicklungsphase für ungewollte und unnötige Verwirrung sorgen.

Alternativ biete sich an, die Benutzeroberfläche komplett per Code im User Interface Controller zu erstellen und anzupassen. Ein Vorteil, der sich sofort zu erkennen gibt, ist der Lokalität des geschriebenen Codes. Die komplette Benutzeroberfläche kann an einem Ort geschrieben und mit Logik dynamisch verändert werden. Jedoch bildet sich schnell auch ein offensichtlicher Nachteil: Der Umfang im Vergleich zum Scene Builder wäre um Weiten größer. Viel mehr Code müsse geschrieben werden, um allein ein UI-Element zu erstellen und einem Node hinzuzufügen. Daraus ergibt sich aber wieder der Vorteil, dass man vollkommene Kontrolle über alle Nodes und dessen Eigenschaften hat. Insbesondere kann auf bestimmte Nodes und dessen Eigenschaften zurückgegriffen werden, welche man dann nutzen kann. In Code gibt es allgemein viel mehr Möglichkeiten, Eigenschaften von Nodes zu beeinflussen, welche im Scene Builder nicht mal als Option angezeigt werden. Der Mangel des Scene Builders, keine feingranulare Einstellungen und Veränderungen an Nodes durchführen zu können, kann mit selbstgeschriebenen also Code aufgelöst werden. Dies jedoch auf Kosten von Code Umfang.

Der Nachteil des Scene Builders, dass die Benutzeroberfläche in der Vorschau nicht korrekt angezeigt werden kann, wird hierbei aber auch nicht gelöst; im Gegenteil. Die Benutzeroberfläche im Controller zu schreiben hat zur Folge, dass es überhaupt keine Vorschau der Benutzeroberfläche gibt. Demnach müsste das Programm immer wieder aufs Neue kompiliert und ausgeführt werden, um Änderungen in der Benutzeroberfläche sehen zu können. Dies würde in der Entwicklungsphase zeitlichen Anspruch erfordern. Ebenso erfordert die Möglichkeit, alle Eigenschaften von Nodes ändern zu können, dass der Entwickler die Dokumentation ausgiebig gelesen haben muss, um zu verstehen, welche Eigenschaften verändert werden können, was für einen Einfluss diese haben und wie diese mit anderen Eigenschaften zusammenhängen und miteinander interagieren.

Zusammenfassend kann zur Variante mit dem Scene Builder genannt werden, dass dessen Benutzung intuitiver und einfacher erfolgen würde, als die ganze Benutzeroberfläche in Code zu schreiben. Dies ginge jedoch auf Kosten auf mangelnder Feingranularität bezüglich der Veränderung der Eigenschaften der Nodes und den Zusammenhang von Nodes zueinander. Diese Feingranularität und absolute Kontrolle über alle Eigenschaften bietet dafür die Variante, alles in Code zu schreiben. Dies kommt jedoch mit erheblich mehr Code Umfang als im Scene Builder.

Realisationsbeschreibung

Die Benutzeroberfläche wird mittels Code im User Interface Controller erstellt. Die vollständige Kontrolle aller Eigenschaften überwiegt den Nachteil des Code-Umfangs. Um trotz des erhöhten Umfangs für Übersichtlichkeit zu sorgen, werden generalisierte Hilfsmethoden definiert, welche überall genutzt werden können und durch übergebene Parameter speziellere Ergebnisse liefern können. So oft wie möglich wird modularisiert und Code in Methoden ausgelagert. Dennoch gibt es

Methoden, die vom Umfang sehr lang sind. Aus diesem Grund werden neben erklärenden Kommentaren auch strukturelle Kommentare verwendet, um für Übersichtlichkeit und Ordnung zu sorgen.

2.2.2. GUI-Darstellung der Benutzeroberfläche

Problemanalyse

Für das Spiel Crosswise soll eine Benutzeroberfläche erstellt werden. Diese soll alle für das Gameplay notwendigen Elemente enthalten. Darunter zählen das Spielfeld, die Spielerhände und unterschiedliche Informationsanzeigen. Auf der Benutzeroberfläche soll jegliche Interaktion der Spieler mit der Anwendung stattfinden. Die Oberfläche soll visuell konsistent und ansprechend sein und sich bei Größenveränderung des Fensters entsprechend anpassen können. Auf einige Elemente der Benutzeroberfläche wird später näher eingegangen. Folgend soll die Struktur und Anordnung der Elemente und die Verwendung bestimmter Node Typen diskutiert werden.

Realisationsanalyse

Die Benutzeroberfläche beinhaltet viele Elemente. Diese müssen individuell angeordnet werden können. Für die horizontale und vertikale Anordnung von Elementen eignen sich idealerweise Nodes vom Typ HBox und VBox an. Für die Oberfläche müssen Elemente jedoch im 2-dimensionalen Raum angeordnet werden. Elemente nur horizontal oder vertikal aneinandergereiht darzustellen, scheint nicht optimal zu sein. Dafür eignet sich ein GridPane hervorragend. Das Layout von Zeilen und Spalten des GridPanes können individuell angepasst werden und weitere JavaFX-Komponenten können per Spalten- und Zeilen-Angabe wie im Koordinatensystem dem Grid hinzugefügt werden. Dies hat u.a. den Vorteil, dass Elemente schnell an anderen Stellen im GridPane eingefügt oder neu angeordnet werden können.

Die Realisierung der Benutzeroberfläche wird auf jeden Fall eine Kombination mehrerer Typen von Nodes erfordern. Die Verwendung folgender JavaFX Komponenten sind sinnvoll:

- **HBox** und **VBox** können hervorragend verwendet werden, um Elemente zu zentrieren und vertikal bzw. horizontal anzuordnen.
- **GridPanes** können für komplexere, zusammenhängende Konstrukte verwendet werden. Dessen Layouts können exakt angepasst werden und auf die Kinder kann einfach mit Spalten- und Zeilenangaben zugegriffen werden
- **StackPanes** eignen sich ideal für zusätzliche Ebenen auf der gleichen Benutzeroberfläche. Die Elemente des StackPanes können dann einfach in den Vordergrund oder Hintergrund gerückt werden.
- **Panes** eignen sich für einfache Elemente, die keine großartige Struktur darstellen.

Die Nutzung weiterer Komponenten ist möglich. Wichtig wird es aber sein, das Layout aller Elemente an Größen anderer zu binden, sodass diese sich abhängig voneinander anpassen, sich dementsprechend vergrößern oder verkleinern, sich strecken oder zusammenquetschen. Hauptsache das Spielfeld nimmt den größten Platz ein, denn dort findet die hauptsächliche Interaktion statt.

Realisationsbeschreibung

Die Benutzeroberfläche wird durch mehrere GridPanes untergliedert. Diese sind wiederum in VBox-HBox-Kombinationen untergeordnet, um diese richtig ausrichten zu können. Demnach ist die Wurzel der Benutzeroberfläche eine VBox, welche als erstes Element eine Menüleiste mit ihren Elementen beinhaltet. Das Haupt-Grid ist das zweite Element der VBox und befindet sich zusätzlich in einer HBox und steht horizontal immer mittig, egal wie breit das Fenster gezogen ist. Die Breite der ersten Spalte des Haupt-Grids, in der sich das Spielfeld befindet, ist an der Höhe der HBox gebunden und wächst bei extrem breiten Fenstern gezielt nicht weiter, sondern nur, wenn das Fenster gleichzeitig breit und hoch

genug ist. Der ungenutzte horizontale Platz wird durch einen weiß-grau gestreiften Hintergrund gefüllt, welcher nicht in den Fokus gerät. Der Hintergrund des Haupt-Grids ist ein Gradient, welcher von unten links mit der Farbe Rot nach oben rechts zur Farbe Blau verschmilzt. Dies lenkt den Fokus des Spielers darauf.

Das Haupt-Grid wird in 3 wesentliche Bereiche aufgeteilt. In einem Spielbereich und zwei Informationsbereiche. Jeder Bereich besitzt einen weißen, abgerundeten Hintergrund und hebt sich vom Gradienten-Hintergrund ab. Das Haupt-Grid enthält 2 Spalten und 2 Zeilen. Oben links befindet sich das allgemeine Informations-Label, welches im Spielverlauf u. a. den aktuellen Spieler am Zug anzeigt. Unten links befindet sich der Bereich für das Spielfeld und die Spielhände. Das Spielfeld muss unter allen Umständen quadratisch bleiben. Deshalb ist der Container, welcher das Spielfeld und die Spielerhände enthält, in einer HBox-VBox Kombination unterordnet, die ein quadratisches Layout erzwingt. Die komplette linke Spalte des Haupt-Grids nimmt den größten Platz ein, da auf dem Spielfeld die meiste Interaktion stattfindet.

Die komplette rechte Spalte ist mit ihren zwei Zeilen verbunden und dient als Seiten Container für weitere Informationsanzeigen. Darunter zählen die Team-Übersichts-Bereiche vom vertikalen und horizontalen Team, die Anzeige der bereits genutzten Wildcards und die optionale Punktetabelle. Diese sind von oben nach unten angeordnet. Der gesamte Seiten-Container ist eine VBox, dessen Elemente vertikal angeordnet sind und deren Reihenfolge ggf. angepasst werden kann. Jedes der Elemente besitzt einen abgerundeten schwarzen Rahmen, damit sie von den anderen abgegrenzt werden können.

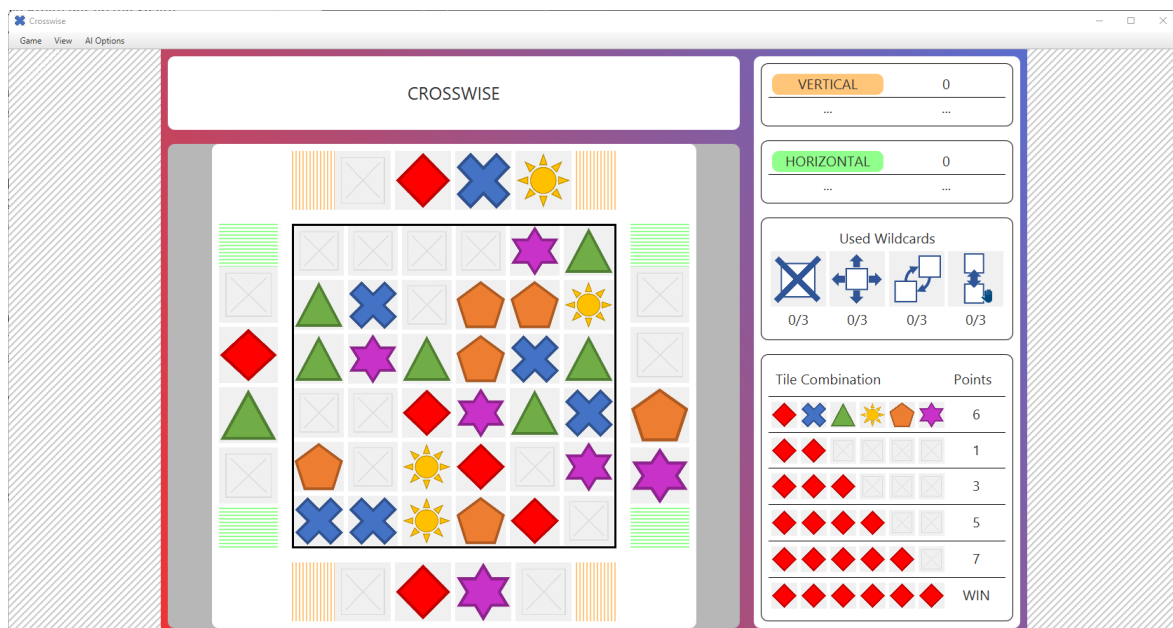


Abbildung 28: Allgemeine Benutzeroberfläche

2.2.3. GUI-Darstellung des Spielfelds

Problemanalyse

Das Spielfeld bei Crosswise besteht aus einer 6×6 Tabelle. Diese soll die größte Fläche des Fensters einnehmen und sich bei verändernden Fenstergrößen entsprechen anpassen können. Jede Zelle beinhaltet Bilder von Spielsteinen, die per Drag & Drop hineingezogen werden können sollen. Jede Zelle muss stets quadratisch bleiben, unabhängig der Fenstergröße.

Realisationsanalyse

Das Spielfeld muss auf jeden Fall mit Hilfe von ImageViews erstellt werden. In der Aufgabe wurde zum einen gefordert, dass Spielsteine per Drag & Drop gespielt werden sollen. Zum anderen wurde ein Bilderverzeichnis mitgegeben, welches benutzt werden soll. Die Spielsteine also ohne ImageViews darzustellen, wäre sinnlos. Demnach steht der Gebrauch von ImageViews und Images schon mal fest. Folgend gilt es zu diskutieren, wie mit den ImageViews ein Spielfeld dargestellt werden soll.

1. Darstellung mit GridPane

Hierfür eignet sich ein 6×6 **GridPane** hervorragend. Zum einen kann man ein komplett quadratisches GridPane erstellen, dessen Spalten- und Zeilen-Layouts den gleichen Platz einnehmen. So kann einfach sichergestellt werden, dass alle ImageViews ein quadratisches Layout annehmen. Zum anderen kann man ein GridPane benutzen, um den Zugriff über Spalten- und Zeilenangabe zu realisieren. Dieser Zugriff lässt sich also Koordinatenweise gestalten. So kann schnell, effizient und einfach auf einzelne ImageViews in den Zellen des Spielfeld Grids zugegriffen werden.

2. Design

Das Spielfeld lässt sich weiter visuell aufbereiten. Zum Beispiel könnte man die horizontale und vertikale Lücke des GridPanes mit einer konstanten Lücke setzen, damit die ImageViews nicht direkt aneinandergrenzen. Außerdem kann man mit Hilfe von CSS das Spielfeld weiter anpassen. Um das Spielfeld extra hervorzuheben, kann man einen breiten Rahmen um das Spielfeld ziehen lassen, um das Spielfeld von den anderen Elementen der Benutzeroberfläche abzugrenzen.

3. Experimentelle Größen

Man könnte den extra Aufwand betreiben und die Spielfeldgröße variabel halten. So könnte man mit verschiedenen Spielfeldgrößen arbeiten, die kleiner oder größer als 6×6 sind. Bei größeren Spielfeldern würde es sich aber anbieten, die Lücke zwischen den ImageViews auf 0 zu reduzieren.

4. Effiziente Nutzung von Images

Wenn man auf dem Spielfeld Bilder setzt, muss normalerweise eine Image Instanz erstellt werden. Bei einem 6×6 Spielfeld ist dies noch machbar. Dann gibt es insgesamt 36 Image Instanzen für das Spielfeld. Wenn man dies jedoch ausweitet und eventuell ein größeres Spielfeld benutzt, steigt die Anzahl der Image Instanzen quadratisch an. Deshalb wäre es sinnvoll, für jeden Spielsteintyp eine Image Instanz zu verwenden. Dazu kann man einfach eine Datenstruktur verwenden, welche zum jeweiligen Spielstein Enum eine Image Instanz speichert. Denn es ist möglich mehreren ImageViews die gleiche Image Instanz zu geben. Wenn dies richtig implementiert wird, können Images viel effizienter verwendet werden, denn es würde nur so viele Image Instanzen geben, wie viele Typen von Spielsteinen es gibt.

Realisationsbeschreibung

Das Spielfeld wird mithilfe eines 6×6 **GridPanels** dargestellt. Dieses enthält in jeder Zelle ImageViews, welche jeweils mit Images belegt werden können. Damit die ImageViews nicht direkt aneinander grenzen, wird die HGap und VGap des GridPanels auf eine Lücken-Konstante gesetzt. Das Spielfeld passt sich an der Fenstergröße an, wächst aber nur quadratisch mit der gleichen Höhe und Breite, um das Seitenverhältnis zu wahren. Bei Vergrößerung des Spielfeldes wachsen auch die ImageViews entsprechend und die Images werden größer.

Für experimentelle Zwecke kann das Spielfeld mit verschiedenen Spielfeld-Größen arbeiten. Der Programmierer kann hierfür die Konstante für die Spielfeldgröße ändern. Dies ist rein experimentell und soll als „Proof of concept“ dienen. Im Endprodukt merkt der Spieler nichts davon. Dies bietet eine gute Basis, wenn darauf aufgebaut werden soll.

Damit mit den ImageViews interagiert werden kann, bekommen alle ImageViews entsprechende Drag & Drop Event Handler, die auf das Ziehen und Hineinlegen von Spielstein ImageViews reagieren. Wenn ein passender Standard Spielstein auf ein passendes leeres ImageView (gefüllt mit leerem Spielstein Image) gezogen wird, leuchtet dieses ImageView grün auf und zeigt dem Spieler an, dass der gezogene Spielstein dort hingelegt werden kann. Wenn der Spielstein vom ImageView weggezogen wird, verschwindet der Effekt. Wenn der Spielstein dort abgelegt wird, wird das entsprechende Spielstein Image auf das jeweilige ImageView auf dem Spielfeld gesetzt. Wenn eine „2-Phase-Wildcard“ gespielt wird, leuchtet das gesamte Spielfeld auf, um den Spieler zu zeigen, dass die Wildcard überall auf dem Spielfeld gespielt werden kann. Weitere Effekt Konstellationen sind hier möglich. Für die Effekte und Reaktionen auf bestimmte Fälle kann im Benutzerhandbuch unter dem Punkt Spielablauf geschaut werden (siehe Abschnitt 1.3.3.).

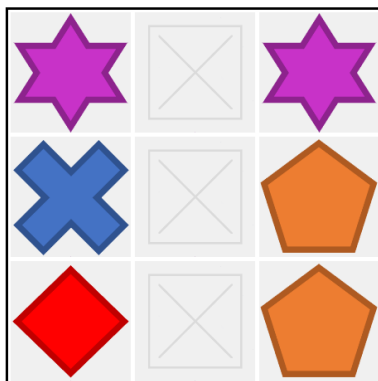


Abbildung 29: 3x3 Spielfeld

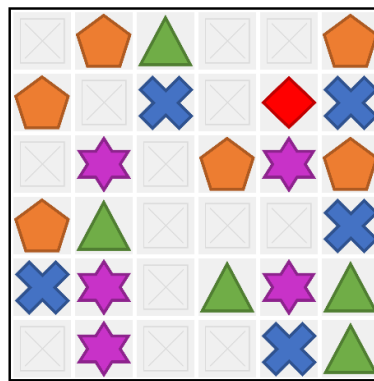


Abbildung 30: 6x6 Spielfeld

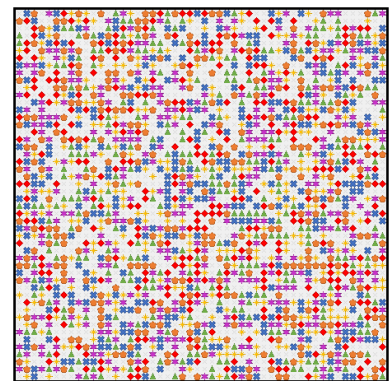


Abbildung 31: 50x50 Spielfeld

2.2.4. GUI-Darstellung der Spielerhände

Problemanalyse

Für das Spiel Crosswise müssen die Spielerhände gestaltet werden. Jeder Spieler kann in seiner Spielerhand bis zu 4 Spielsteine halten. Mit Drag & Drop soll ein Spielstein von der Spielerhand gezogen werden können. Die Spielerhand soll sich automatisch der Fenstergröße anpassen und sich ggf. vergrößern oder verkleinern.

Realisationsanalyse

1. Mehrere Spielerhände

Ein Punkt, bei dem man sich Gedanken machen sollte, ist die Anzahl der angezeigten Spielerhände. Hierbei bieten sich 2 Konstellationen an. Entweder hat man eine Spielerhand, die dann die Hand des aktuellen Spielers anzeigt oder man hat genau so viele Hände auf der Benutzeroberfläche, wie es aktive Spieler gibt.

Ein Grund für eine Hand wäre die Simplizität. Es müsste einfach nur jede Spielerhand nacheinander auf der gleichen Hand angezeigt werden. Diese eine Hand könnte man dann unter dem Spielfeld anbringen. Dies bringt aber den subtilen Nachteil mit sich, dass die Spielerhand an einem Ort fixiert ist. Gerade in dem Spiel Crosswise ist es von Relevanz, dass Spieler ihre Hand so zum Spielfeld ausgerichtet haben, sodass die Handposition dem Team entspricht. Sprich das vertikale Team hat ihre Hände über und unter dem Spielfeld und das horizontale Team hat ihre Hände links und rechts vom Spielfeld. Diese Anordnung macht von der Benutzerfreundlichkeit und Intuitivität auch sofort Sinn, weil die Spieler sofort wissen, wie sie ihre Spielsteine zu platzieren haben.

Demnach ist es sinnvoll, sich auch über mehrere Spielerhände um das Spielfeld Gedanken zu machen. Bei 2 bzw. 4 Spielerhänden (je nach aktiver Spieleranzahl) würde das vertikale Team ihre Hände vertikal am Spielfeld ausrichten und das horizontale Team ihre Hände horizontal vom Spielfeld. Dies macht das Spielen intuitiver. Wenn es nur eine Hand gäbe, müsste jedes Team erstmal schauen zu welchem Team sie gehören und dann ihre Spielsteine entsprechend in Zeilen bzw. Spalten legen. Bei mehreren Händen sind diese schon passend ausgerichtet und die Spieler müssen sich keine Gedanken darüber machen, wie sie ihre Spielsteine zu legen haben.

Ein weiterer Vorteil wäre, dass sich die Spielerhände abwechselnd anzeigen und so der Spielverlauf, und welcher Spieler am Zug ist, viel leichter abzulesen wäre. 4 Spielerhände würden dann so im Uhrzeigersinn nacheinander die aktuelle Spielerhand anzeigen. Dies ahmt ein gewisses Brettspiel Gefühl nach. Jeder kann sofort nachvollziehen, welcher Spieler am Zug ist, vorher am Zug war und als Nächstes am Zug sein wird. Dies wäre direkt an der Position der Hand sichtbar. Dies würde für die Benutzerfreundlichkeit und Intuitivität sehr viel beitragen.

Die Variante mit mehreren Spielerhänden bringt aber auch Komplexität und ungenutzten Platz mit sich. Zurzeit kann nur eine Spielerhand angezeigt werden – die des aktuellen Spielers. Die anderen Hände stünden dann nur da und zeigen leere Spielsteine an, um nicht zu verraten, was die anderen Spieler in der Hand haben.

Dennoch kann man sagen, dass die Variante, in der die Handanzahl der Anzahl der aktiven Spieler gleicht, als viel nutzerfreundlicher und intuitiver empfunden werden würde, da durch die Handpositionen viel mehr herausgelesen werden kann. Dies sorgt für Struktur und Ordnung und lässt das Spiel mehr wie ein Brettspiel wirken.

2. Handdarstellung

Wie für die Spielfeld-Darstellung eignet sich für die Darstellung der Spielerhand ebenfalls ein GridPane. Je nachdem, ob eine Hand vertikal oder horizontal ausgerichtet ist, kann es ein GridPane geben, welches 4 Spalten mit einer Zeile oder 4 Zeilen mit einer Spalte enthält. Da die gesamte Benutzeroberfläche via Code erstellt wird, ist es außerdem möglich, die Spalten- / Zeilenanzahl variabel zu halten.

Für die Zellen des GridPanes bietet sich demnach nur eine sinnvolle JavaFX-Komponente an: **ImageViews**. Da das Spiel hauptsächlich via Drag & Drop bedient werden soll, macht es am meisten Sinn ImageView mit den Bildern der jeweiligen Spielsteine zu verwenden. Optisch ließe sich mit dieser Variante das beste Resultat erzielen.

Natürlich ist es auch möglich, Labels zu verwenden und diese mit den jeweiligen Spielsteinnamen zu versehen. Dies sieht visuell aber sehr ungeeignet aus. Labels würden nichts für den Spielfluss beitragen, sondern diesen eher behindern. Zudem würde man dann die mitgelieferten Bilder aus der Aufgabenstellung nicht benutzen, was sinnlos wäre. Diesen Nebengedanken kann man also sofort verwerfen.

Demnach sind ImageView die optimale Komponente für die Zellen der GridPanes für die Spielerhand. Diese können dann quadratisch gehalten werden, sodass die Bilder, die in den ImageView gesetzt werden, auch quadratisch sind. Hinzu kommt, dass die Bilder leicht austauschbar wären. So kann man für den gleichen Spielstein verschiedene Bilder benutzen. Dies erfordert jedoch, dass die Bilder in das Projekt mit eingebunden werden müssen. Eine Optimierung wäre aber, dass man zur Laufzeit des Programms eine Liste von Bildern instanziiert, sodass für jeden Spielstein ein Bild bereits existiert. So können gleiche Bildinstanzen für verschiedene ImageView benutzt werden. Dies bringt ebenfalls den Vorteil mit sich, dass Bilderinstanzen direkt miteinander verglichen werden können. Sonst wäre dies nur mit URL Vergleichen machbar. Dieser Ansatz würde auf jeden Fall viel Rechenleistung sparen.

Ein interessanter Aspekt, der genannt werden kann, ist die Erweiterbarkeit und Gestaltung der Spielerhände. Da die ganze Spielerhand via Code gestaltet wird, bieten sich erweiterte designtechnische und funktionale Möglichkeiten an. Die Spielerhände können beispielsweise in HBox bzw. VBox Containern untergebracht werden. Das GridPane für die Spielerhand könnte man so mittig zentrieren. Auch kann man dann den Hintergrund der Container passend zum Team gestalten, was für mehr Nutzerfreundlichkeit und visuellen Anspruch sorgen würde.

3. Experimentelle Ansätze

Ein interessanter experimenteller Aspekt wäre die Möglichkeit, die Spielerhand vergrößern zu können. Man könnte annehmen, dass der Spieler eventuell eine größere Spielerhand bekommen könnte. Es wäre mit zunehmender Größe unmöglich, bei dem begrenzten Platz die Zellen der Spielerhand quadratisch zu halten. Die Images würden demnach zusammengequetscht werden. Dafür bieten sich Nodes vom Typ **ScrollPane** hervorragend an. Diese kann man sogar in einer Scroll-Richtung (horizontal oder vertikal) beschränken. So könnte eine Spielerhand bei dem begrenzten Platz viel mehr Spielsteine tragen, ohne Verlust an Funktionalität.

Realisationsbeschreibung

Es werden genauso viele Spielerhände dargestellt, wie aktive Spieler am Spiel teilnehmen. Die Spielerhände werden mit Hilfe von GridPanes dargestellt. Je nachdem, ob die Spielerhand vertikal oder horizontal (je nach Team) ausgerichtet wird, werden GridPanes mit n-Spalten und einer Zeile bzw. einer Spalte und n-Zeilen erstellt. Diese befinden sich jeweils in einem HBox- bzw. VBox-Container. Eine vertikal ausgerichtete Hand befindet sich in einer VBox, eine horizontal ausgerichtete Hand befindet sich in einer HBox. Die GridPanes werden in ihren jeweiligen Containern mittig ausgerichtet.

Die Container selbst bekommen einen gestreiften Hintergrund, ausgerichtet nach der jeweiligen Richtung des Teams. Die Hintergrundfarbe entspricht ebenfalls der Team-Farbe.

Jedes GridPane ist an gewisse Größen gebunden. Die Breite eines horizontal ausgerichteten GridPanes ist an seiner Höhe gebunden, die mit der Spielerhandgröße multipliziert wird. Darauf wird eine Konstante addiert, um für die Lücke zwischen den ImageViews aufzukommen. Umgekehrt gilt dies für vertikal ausgerichtete GridPanes. Dessen Höhe wird an seine Breite gebunden.

Jede Zelle der GridPanes beinhaltet ein ImageView, die mit einem Image gesetzt werden kann. Die Bilder richten sich in ihrer Größe an das ImageView aus. Die ImageViews werden an der Zellengröße des GridPanes gebunden. Diese sind stets quadratisch, damit auch gesetzten Images quadratisch sind.



Abbildung 32: Vertikale Spielerhand

Für experimentelle Zwecke wird die Hand bei Handgrößen über 6 in einem **ScrollPane** eingebunden, welches je nach Ausrichtung der Hand vertikale oder horizontal „scrollbar“ ist. Somit werden potenziell größere Spielerhände unterstützt und die ImageViews in den Zellen des GridPanes bleiben quadratisch. Dies soll lediglich als „Proof of concept“ verstanden werden und ist keineswegs ausgereift oder verfeinert, bietet aber einen guten Startpunkt, falls man dieses Feature ausbauen möchte.

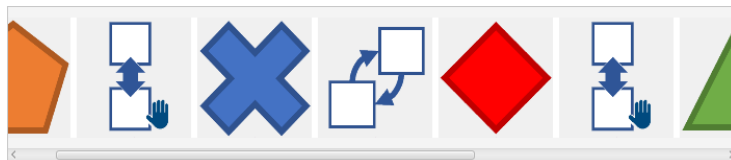


Abbildung 33: Experimentelle Spielerhand mit ScrollPane

Damit mit der Spielerhand interagiert werden kann, werden jedem nicht leeren ImageView Drag & Drop Event Handler vergeben, sodass die Spielsteine von der Spielerhand auf das Spielfeld gezogen werden können. Für weitere Information zum Spielablauf kann im Benutzerhandbuch nachgeschlagen werden (siehe Abschnitt 1.3.3.).

2.2.5. GUI-Darstellung der Menüleiste

Problemanalyse

Aufgabe ist es, ein Optionsmenü zu erstellen, welche folgende Optionen anbietet: Das Ein- oder Ausschalten einer Anzeige der Punktwerte für jede Zeile und Spalte neben den Zeilen und Spalten, einer Anzeige für die aktuellen Punkte für das horizontale und vertikale Team, die Anzeige der Hand des KI-Spielers, wenn dieser am Zug ist und die Dauer der Hervorhebung bzw. Animation der vom Computer abgelegten Steine. Darunter soll es möglich sein, zwischen 3 verschiedenen Geschwindigkeiten (kurz, mittel, lang) auszuwählen. Ziel ist es, dies übersichtlich darzustellen, sodass der Benutzer dieses Menü intuitiv bedienen kann.

Realisationsanalyse

Neben den Optionen, die dem Spieler zur Verfügung gestellt werden müssen, sollen die Spieler auch in der Lage sein ein Spiel starten, speichern, laden oder beenden zu können. Unabhängig vom Zustand der Anwendung müssen der Spieler diese Optionen erreichen und mit denen interagieren können. All diese Interaktionsmöglichkeiten und Einstellungen lassen sich hervorragend in einem **Menü** unterbringen. Hierfür kann eine Menüleiste mit bestimmten Sub-Menüs gestaltet werden, von der aus der Benutzer alles steuern kann. Unabhängig von Zustand der Anwendung, ob ein Spiel gerade läuft oder nicht, sollen die Optionen dem Spieler zugänglich gemacht werden. Eine Ausnahme bildet das Speichern eines Spiels, wenn kein Spiel läuft. Diese Option kann beim Start der Anwendung initial deaktiviert werden.

1. Nutzung von Menü Komponenten

Für die Menü-Leiste ist eine **MenuBar** ideal. Diese kann **Menu** Elemente beinhalten. Diese müssen zunächst sinnvoll untergliedert werden. Es lassen sich 3 wesentliche Menüs bilden. Ein Menü zum allgemeinen Steuern des Spiels, also das Starten, Speichern, Laden und Beenden des Spiels. Dazu die Optionen für das Ein-/ Ausschalten der optionalen Anzeigen. Zuletzt bietet sich ein Menü für die KI Optionen an, mit der alles, was mit dem KI-Spieler zusammenhängt, bündelt. Darunter zählt das Abbrechen bzw. Überspringen der KI Animation/ Hervorhebung, die Dauer davon und das Anzeigen der KI-Hand.

Zum Menü, mit dem das Spiel verwaltet werden kann, passen offensichtlich **MenuItems**, die nur angeklickt werden müssen, um mit denen zu interagieren und eine Aktion auszuführen. Anders ist es beim Menü, in der optionale Anzeigen ein- oder ausgeschaltet werden können. Dazu passen **CheckMenuItems** sehr gut. Diese lassen sich auswählen oder abwählen. Dessen Zustand kann dann abgelesen und intern verarbeitet werden. Diese **CheckMenuItems** können hervorragend dort benutzt werden, wo etwas ein- oder ausgeschaltet werden können soll. Zum KI-Menü passen alle darüber genannten Menü-Elemente. Zum Abbrechen bzw. Überspringen kann ein einfaches **MenuItem** verwendet und zur Anzeige der KI Hand kann ein **CheckMenuItem** verwendet werden.

2. CheckMenuItem vs. RadioMenuItem

Für die Einstellung der KI Animationsdauer soll zwischen drei verschiedenen Zeiten gewählt werden können. Nur eine Option kann gleichzeitig ausgewählt werden. Hierfür würde man zunächst darauf kommen, dass **RadioMenuItems** sich dafür anbieten würden. Diese erlauben nur eine Auswahl aus. In einem Sub-Menü funktioniert diese Auswahl aber leider nicht zuverlässig. Demnach kann man die nächst bessere Lösung nehmen: **CheckMenuItems**. Diese erlauben eine Mehrfachauswahl. Dies kann man aber in Code auf eine Einfachauswahl beschränken. So kann man sicherstellen, dass nur einer der 3 Optionen gleichzeitig ausgewählt werden kann. Dies kann ohne großen Aufwand realisiert werden. Somit lässt sich die Funktionalität von **RadioMenuItems** leicht mit **CheckMenuItems** realisieren. Außerdem sehen **CheckMenuItems** mit den Haken visuell besser aus als Punkte. Ebenfalls würde die

Nutzung von CheckMenuItems im Menü mit den anderen Sub-Menüs einheitlicher wirken. So werden nur Haken und keine Punkte bei der Auswahl angezeigt.

Realisationsbeschreibung

Die Menüleiste wird mit einer **MenuBar** erstellt. Die Menüleiste sitzt oben am Fenster. Diese enthält drei **Menu** Elemente. Diese werden folgend näher erläutert.

Das erste Menü ist für das Spiel zuständig und wird „Game“ genannt. Dieses Menü enthält **MenuItems**, zum Starten, Speichern, Laden und Beenden eines Spiels. Das zweite Menü trägt den Namen „View“ und dient für die optionalen Anzeigen der Team-Punkte, der Punkte-Leisten und der Punktetabelle. Diese können mit **CheckMenuItems** angezeigt oder versteckt werden. Das letzte Menü ist für die KI-Optionen zuständig und trägt den Namen „AI Options“. In diesem Menü kann die KI Animation mit einem **MenuItem** übersprungen werden. Die KI Hand kann durch ein CheckMenuItem angezeigt oder versteckt werden. Und zuletzt kann über ein weiteres Sub-Menü mit CheckMenuItems bestimmt werden, wie lange die KI Animation dauert. Folgende Animationsdauer kann gewählt werden:

- Short = 500ms, Medium = 1000ms, Long = 2000ms

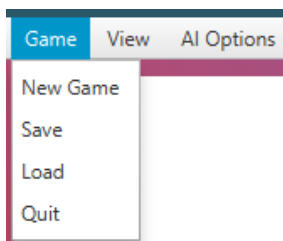


Abbildung 34: "Game" Menu

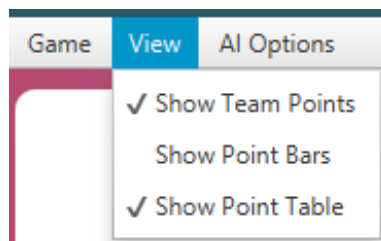


Abbildung 35: "View Menu"

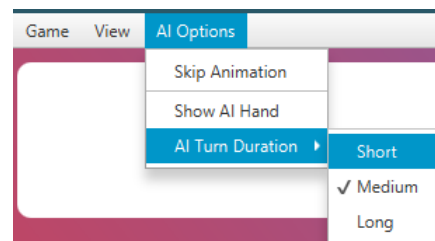


Abbildung 36: "AI Options" Menu

2.2.6. GUI-Darstellung der optionalen Punkteleisten

Problemanalyse

Neben dem Spielfeld soll es eine optionale Punkteanzeige geben, die zu jeder Zeile und Spalte die Punkte anzeigt. Diese Anzeige soll sich nach jedem Zug aktualisieren und bei sich ändernden Fenstergrößen automatisch anpassen.

Realisationsanalyse

Zuerst muss man sich Gedanken darüber machen, wie die einzelnen Spalten und Zeilen jeweils mit Punkten beschriftet werden können. Dies gilt es einheitlich und geordnet zu machen.

1. Spielfeld GridPane erweitern

Da sich das Spielfeld schon in einem GridPane befindet, ist es naheliegend einfach eine weitere Spalte und Zeile hinzuzufügen und darin Labels für die Punkteanzeigen hinzuzufügen. Daraus ergibt sich sofort der Vorteil, dass die neuen Zellen mit den Labels perfekt ausgerichtet zu jeder Spalte und Zeile liegen würden. Was jedoch auch beachtet werden muss ist die Logik, die mit dem Spielfeld arbeitet. Die Erweiterung des Spielfelds wäre insofern problematisch, da die Logik die zusätzliche Spalte und Zeile ebenfalls als Spielfeld wahrnehmen würde. Man könnte demnach auch auf den zusätzlichen Spalten und Zeilen spielen. Dies kann man natürlich in der Logik einschränken lassen. Dennoch würde diese Lösung für unnötige Komplexität sorgen.

2. Separate GridPanes

Eine weitere mögliche Lösung wären 2 neue GridPanes; ein horizontales GridPane für die Spaltenpunktzahlen und ein vertikales GridPane. Beide würden Labels für die Punkteanzeige enthalten. Diese GridPanes müssen aber noch an das Spielfeld geheftet werden. Ebenso ergibt sich die zusätzliche Herausforderung, dass die Labels richtig an den jeweiligen Zeilen und Spalten angeordnet werden müssen, damit alles zusammenpasst. Dafür müssten die Punkte GridPanes das gleiche Spalten- bzw. Zeilenlayout wie das Spielfeld haben, damit sie aneinander passen.

Die Herausforderung, die Punkte GridPanes an das Spielfeld zu heften, kann sich relativ einfach mit einem äußeren GridPane realisieren. Man könnte das Layout des großen Container GridPanes so anpassen, sodass das Spielfeld immer einen quadratischen Platz bekommen, das horizontale Punkte GridPane exakt so breit wie das Spielfeld GridPane und das vertikale Punkte Grid genauso hoch sein würde.

Dieser Ansatz bringt einige Vorteile mit sich. Zum einen bleibt das Spielfeld in sich geschlossen und muss nicht erweitert werden. Die Logik kann dann einfach das Spielfeld nutzen, ohne den Zugriff auf bestimmte Spalten und Zeilen einzuschränken. Außerdem passt das Layout der Punkte GridPanes dann genau zum Spalten- bzw. Zeilen Layout des Spielfeldes. So liegen die Punkte Grid Zellen exakt über bzw. neben den Spalten und Zeilen des Spielfeld Grids.

3. Realisation Ein- & Ausschalten

Die Punkte GridPanes sind optional, müssen also ein- & ausgeschaltet werden können. Es muss also realisiert werden, dass die Punkte Grids versteckt und angezeigt werden können, ohne dass sie den Spielfluss hemmen. Zudem wäre es optimal, wenn die Punkte Grids im versteckten Zustand keinen unnötigen Platz auf der Benutzeroberfläche einnehmen würden. So würde das Spielfeld den maximalen Platz in seinem Container GridPane einnehmen.

Eine naheliegende Möglichkeit, die Punkteleisten zu verstecken, wäre diese **einfach unsichtbar zu machen**. Sie würden im Container GridPane weiterhin Platz einnehmen, jedoch nicht mehr sichtbar für den Spieler sein. Dies wäre eine einfache und schnelle Lösung, wäre aber nicht optimal, da der eingenommene Platz dann unnötig verschwendet wird.

Die weitere Möglichkeit wäre, die Punkte Grids im Controller zu speichern und einfach **vom Container Grid zu entfernen**, wenn diese ausgeschaltet werden. Diese würden weiterhin aktualisiert werden, nehmen aber keinen Platz mehr im Container GridPane ein. So kann das Spielfeld Grid den größtmöglichen Platz einnehmen. Dafür könnte man das GridPane einfach das gesamte Container GridPane überspannen lassen. So würde eine maximale Spielfeld-Größe erzielt werden. Wenn die Punkteleisten wieder angezeigt werden sollen, dann kann man diese einfach in das Container Grid wieder hinzufügen und das Spielfeld wieder einem verkleinerten Bereich im Grid zuweisen.

4. „Win of Sixes“ Punkteanzeige

Da die Spielfeld-Evaluation für einen „Win of Sixes“ den maximalen Wert einer Integer liefert, sollte diese nicht in der Punkteanzeige so angezeigt werden. Der Grund bei einem „Win of Sixes“ die maximale Integer zu vergeben ist die Irrelevanz der Punkte bei einem solchen Gewinnfall. Wenn ein „Win of Sixes“ erzielt worden ist, auf den Punkteleisten nicht die Punktzahl der betroffenen Spalte bzw. Spalte angezeigt werden, sondern dem Spieler mitgeteilt werden, dass dies für einen sofortigen Gewinn sorgt. Demnach könnte man anstelle einer Punktzahl ein Symbol nehmen, welches den sofortigen Gewinn verdeutlicht.

5. Design

Die Punkte Grids können zwar passend an das Spielfeld Grid angeordnet werden, sodass abgelesen werden kann, zu welcher Spalte bzw. Zeile sie Punkte anzeigen. Dennoch kann man ein wenig Mehraufwand betreiben und dem Spieler die Denkaufgabe etwas abnehmen. Via CSS können die Punkteleisten der Teamfarbe entsprechend gefärbt werden. Die Punkteleiste für die Zeilen des Spielfeldes sind für das horizontale Team relevant und die Punkteleiste für die Spalten des Spielfeldes für das vertikale Team. Demnach könnte man den Punkte GridPanes eine Hintergrundfarbe geben, die der entsprechenden Teamfarbe entspricht. Dies zudem visuell ansprechender wirken.

Realisationsbeschreibung

Die optionalen Punkteleisten werden mit Hilfe von zwei separaten GridPanes erstellt. Das horizontale Punkte Grid besteht aus genauso vielen Spalten wie das Spielfeld Grid und einer Zeile; das vertikale Punkte Grid besteht aus einer Spalte und genauso viele Zeilen wie das Spielfeld Grid. Jede Zelle der GridPanes enthält jeweils ein Label, das zentriert angeordnet ist. Da das Spielfeld und die Punkte Grids eigenständige GridPanes sind, kann die Logik auch problemlos mit dem Spielfeld arbeiten.

Jedes Label des horizontalen Punkte Grids ist für die Punktzahl jeweils einer Spalte zuständig. Umgekehrt ist jedes Label des vertikalen Punkte Grids für die Punktzahl jeweils einer Zeile zuständig. Wenn ein „Win of Sixes“ erreicht wurde, werden in der jeweiligen betroffenen Spalte bzw. Zeile nicht die Punkte dazu ausgegeben. Stattdessen wird ein Trophäen-Symbol angezeigt, um dem Spieler zu zeigen, dass ein sofortiger Gewinn erzielt worden ist. Die Punkte des Gewinnerteams sind bei einem „Win of Sixes“ irrelevant.

Sowohl das Spielfeld Grid als auch die beiden Punkteleisten Grids sind in einem 2×2 Container GridPane untergebracht. Das Spielfeld nimmt die obere linke Position ein, das vertikale Punkte Grid die oben rechte und das horizontale Punkte Grid die unten linke Position. Da die Grids das passende Layout zum Spielfeld Grid haben, werden die Labels richtig zur passenden Spalte bzw. Zeile angeordnet. Wenn die Punkte Grid ausgeblendet werden sollen, werden diese einfach vom Container Grid entfernt, bleiben jedoch im User Interface Controller gespeichert, um diese ggf. wieder dort hinzuzufügen. Wenn die Punkte Grids ausgeblendet bzw. entfernt werden, wird das Spielfeld im Container Grid auch die nun freien Zellen überspannen. So wird das Spielfeld Grid größer, wenn die Punkteleisten nicht angezeigt werden und kleiner, wenn diese angezeigt werden.

Für die designtechnische Darstellung wird u. a. CSS verwendet. Das vertikale Punkte Grid bekommt einen gestreiften Hintergrund in der Farbe des korrespondierenden horizontalen Teams. Das horizontale Punkte Grid bekommt die Hintergrundfarbe des vertikalen Teams. Außerdem werden die Labels in ihren GridPane Zellen zentriert.

						3
						7
						5
						1
						7
						1
0	6	6	0	0	0	

Abbildung 37: Optionale Punkteleisten

2.2.7. GUI-Darstellung der Team-Informationsbereiche

Problemanalyse

Alle für die Bedienung des Spiels Crosswise notwendigen Komponenten müssen in der Benutzeroberfläche angezeigt werden. Dies schließt die Team-Informationen mit ein. Die Spieler müssen wissen, welchem Team sie zugeordnet sind und ggf. wie viele Punkte sie als Team aktuell haben. Man kann von den Spielern nicht erwarten, dass sie sich die Startkonfiguration ihres Spiels merken können, vor allem da das Menü zum Erstellen eines neuen Spiels nach Start des Spiels verschwindet. Dies soll zur Nutzerfreundlichkeit und Bedienbarkeit dienen.

Realisationsanalyse

Die Team Informationsbereiche sind eine gebündelte Struktur, welche ideal mit GridPanes strukturiert dargestellt werden können. Mit GridPanes ließen sich auch alle Elemente leicht adressieren. Außerdem kann man Zellen sehr einfach mehrere Spalten und / oder Zeilen umspannen lassen, was für deutliche Flexibilität der Elemente sorgt. Elemente können demnach einfach hinzugefügt, entfernt oder verschoben werden. Dies bietet viel Freiraum für die Erstellung der Team-Informationsbereiche.

1. JavaFX-Komponenten

Als sinnvolle JavaFX-Komponenten kommen also weiterhin **GridPanes**, **HBox** und **Labels** zur Nutzung. Auf diese muss aber nicht weiter eingegangen werden, da dies schon an mehreren Stellen in den Problemanalysen. Zusammenfassend können hier HBox-Label Kombinationen gewählt werden, damit die Labels innerhalb der Zellen im GridPane mittig ausgerichtet werden.

2. Design

Wie die anderen Elemente in der Benutzeroberfläche kann CSS verwendet werden, um die Team-Informationsbereiche entsprechend visuell aufzubessern. So können zum Beispiel die Team-Namen die entsprechenden Farben als Hintergrund bekommen oder die GridPanes können einen abgerundeten schwarzen Rahmen bekommen. Weiteres Styling z. B. mit Trennlinien ist möglich.

Realisationsbeschreibung

Die Darstellung der Team-Informationsbereiche lässt sich als einfaches 2×2 GridPane realisieren. Beide Spalten nehmen jeweils 50% der Gesamtbreite ein. Die letzte Zeile des GridPanee enthält in beiden Zellen jeweils ein in einer HBox zentriertes Label. Diese Label zeigen die Spielernamen des jeweiligen Teams an. Wenn in einem Team nur ein aktiver Spieler vorkommt, wird auch nur ein Label angezeigt. Das Label für den zweiten Spielernamen wird nicht angezeigt. Die erste Zeile beinhaltet den

Team-Namen. Dieser Team-Name wird durch ein Label repräsentiert, welches sich wiederum in einer HBox befindet, die das Label zentriert. Diese HBox als Team-Namen-Container wird entsprechend des passenden Teams gefärbt. Der Container erhält beim vertikalen Team einen orangen, abgerundeten Hintergrund. Beim horizontalen Team erhält der HBox Container einen grünen abgerundeten Hintergrund.

Optional soll der Spieler die Team-Punkte anzeigen lassen können. Wenn die Punkte angezeigt werden sollen, wird eine extra HBox-Label Kombination in der oberen rechten Ecke des GridPanes hinzugefügt. Wenn die Team-Punkte nicht angezeigt werden sollen, wird die HBox-Label Kombination wieder aus dem GridPane entfernt, bleibt jedoch intern gespeichert. Da jetzt wieder Platz in der ersten Zeile im GridPane ist, kann der Team-Name beide Spalten umspannen. Wenn die Punkte angezeigt werden, bleibt der Team-Name in der oberen linken Ecke.

Beide Team Information-GridPanes bekommen durch CSS ebenfalls einen abgerundeten schwarzen Rahmen, damit diese voneinander und von den anderen Elementen in der Benutzeroberfläche unterschieden werden können. Wenn sich das Fenster vergrößert, wachsen die GridPanes bedingt in der Breite. Die Höhe wird auf ein gewisses Maximum begrenzt, damit die Team-Informationsübersicht nicht so viel Platz im Seiten-Container einnimmt.

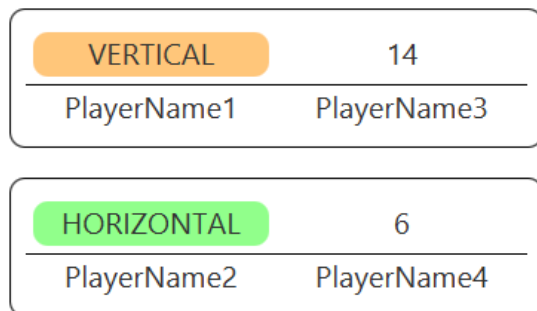


Abbildung 38: Team Information mit Punktzahl

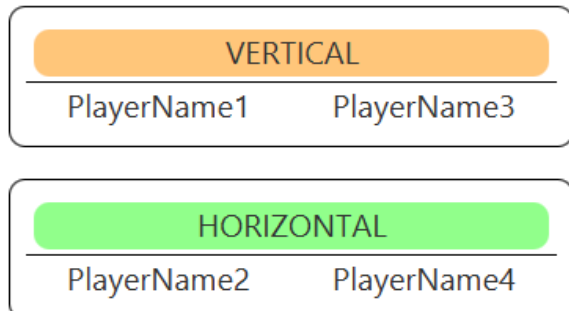


Abbildung 39: Team Information ohne Punktzahl

2.2.8. GUI-Darstellung der genutzten Wildcards

Problemanalyse

Es muss eine Anzeige geben, die alle 4 Wildcards darstellt und wie oft diese schon verwendet worden sind. Über den Spielverlauf soll sich diese Anzeige aktualisieren und die aktuelle Anzahl der bereits verwendeten Wildcards anzeigen. Wenn das sich die Fenstergröße verändert, soll sich die Anzeige der genutzten Wildcards entsprechend anpassen. Wie die Anzeige gestaltet werden soll, ist dem Programmierer überlassen, Hauptsache, die Anzeige sieht visuell ansprechend und zusammen mit den anderen Elementen der Benutzeroberfläche konsistent aus. Der Spieler muss sofort erkennen, wofür die Anzeige dient und was diese darstellt.

Realisationsanalyse

Für die Anzeige der genutzten Wildcards können die gleichen JavaFX-Komponenten wie für die anderen Elemente der Benutzeroberfläche verwendet werden. Darunter die klassischen Elemente wie das **GridPane**, **HBox**, **VBox**, **Labels**, usw. Ideal wäre es, wenn für die genutzten Wildcards auch die Bilder angezeigt werden. Hierfür können **ImageViews** und **Images** verwendet werden. Diese Komponenten haben sich schon bei ausgiebiger Nutzung als passend erwiesen. Vom Design her sind wie bei den anderen Elementen der Benutzeroberfläche zahlreiche Styling Optionen möglich wie zum Beispiel das Zentrieren der Labels mittels einer HBox oder das Setzen eines Rahmens via CSS.

Realisationsbeschreibung

Die Darstellung der bereits genutzten Wildcards wird mit einem GridPane realisiert, welches eine Spalte und 3 Zeilen hat. Die erste Zeile beinhaltet das in einer HBox zentrierte Label, damit der Spieler weiß, wofür diese Anzeige dient. Das Label wird mit dem Text „Used Wildcards“ versehen.

Die zweite Zeile beinhaltet ein weiteres Grid für die Anzeige der Wildcard Images. Das Grid besteht aus 4 Spalten und einer Zeile. Die Breite des Grids ist gebunden an der Breite des äußeren Grids und nimmt immer die volle Breite ein, die möglich ist. Dieses Grid erhält in jeder seiner 4 Zellen ImageViews, welche die 4 unterschiedlichen Wildcards repräsentieren sollen. Die Breite und Höhe der ImageViews ist an der Breite des GridPanes gebunden, welches diese beinhaltet. Diese Breite wird dann nochmal durch die Spaltenanzahl geteilt und davon wird ebenfalls die Konstante für die Lücke zwischen den ImageViews abgezogen. Somit sind alle ImageViews quadratisch aufgebaut. Die ImageViews werden von links nach rechts mit folgenden Wildcard-Images belegt: REMOVE, MOVER, SWAPONBOARD, SWAPWITHHAND. Die Bilder werden aus der Utilities Klasse aus dem GUI-Package geholt, in der es eine Liste mit allen bereits instanziierten Spielstein Images gibt. Da die ImageViews schon quadratisch sind, sind die gesetzten Images ebenfalls quadratisch.

Die dritte und somit letzte Zeile des GridPanes enthält wie die darüber liegende Zeile ein GridPane mit 4 Spalten und einer Zeile. Ebenso sind alle Spalten gleichmäßig aufgeteilt und nehmen jeweils die gleiche Breite ein. Dieses Grid beinhaltet jeweils einen in einer HBox eingeschlossenes zentriertes Label, welches die Anzahl der jeweils benutzten Wildcard anzeigen soll. Die Labels werden mit einem Text versehen, welches die Anzahl der benutzten Wildcards in Bezug auf dessen Maximum stellt. Da die GridPanes der zweiten und dritten Zeile die gleichen Spaltenaufteilungen haben, in der jede Spalte 25% des Platzes einnimmt, wird garantiert, dass die Labels exakt mittig unter den jeweiligen ImageViews der Wildcards liegen.

Die zweite Zeile mit den ImageViews der Wildcards soll den meisten Platz einnehmen. Demnach sind die Höhen der ersten und dritten Zeile an einem Bruchteil der Höhe der zweiten Zeile gebunden. Das gesamte GridPane wird wie jedes Element im seitlichen Informationscontainer via CSS Styling in einem abgerundeten schwarzen Rahmen umschlossen.

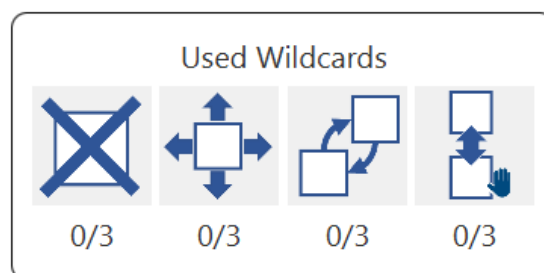


Abbildung 40: Anzeige genutzter Wildcards

2.2.9. GUI-Darstellung der Punktetabelle

Problemanalyse

Nicht jeder ist mit den Regeln von Crosswise vertraut und kennt ggf. nicht die grundlegenden Punktekonfigurationen für gelegte Kombinationen von Spielsteinen. Deshalb muss es eine optionale Anzeige geben, welche die möglichen Kombinationen von Spielsteinen und dessen korrespondierende Punktzahl anzeigt. Wie die Anzeige gestaltet werden soll, ist weiterhin dem Programmierer überlassen. Voraussetzung sind wieder Konsistenz mit den anderen Elementen der Benutzeroberfläche und die Intuitivität, damit der Spieler sofort weiß, wofür diese Anzeige dient. Ebenfalls soll die Anzeige sich an der Größe des Fensters dynamisch anpassen.

Realisationsanalyse

Für die Punktetabelle gibt es verschiedene Möglichkeiten, diese zu gestalten. Der einfache Weg, wäre die Kombination mit den korrespondierenden Punkten textuell anzuzeigen. Dies ließe sich natürlich einfach mit einer Hand voll Labels realisieren. Es würde jedoch visuell nicht so ansprechend wirken. Der Anspruch, den die Benutzeroberfläche weckt, würde die Punktetabelle nicht mehr gerecht werden. Der Spieler erwartet eine Punktetabelle von qualitativ hochwertigem Umfang. Da das Spiel hauptsächlich mit ImageViews arbeitet, sollten dementsprechend auch die Punktekombination visuell aufbereitet, mit Bildern von Spielsteinen aufgebaut werden.

Dass Bilder von Spielsteinen für die Kombinationen verwendet werden sollen, steht also außer Frage. Dies impliziert die Notwendigkeit des Gebrauchs von ImageViews und Images. Auf diese muss folgend nicht weiter eingegangen werden. Die Bilderkombinationen müssen in Gruppen von 6 gebündelt werden. Hierfür würden sich generell 2 Typen von Nodes anbieten. Zum einen die HBox und zum anderen das GridPane.

1. HBox als Kombinations-Container

Die Gruppe von ImageViews nebeneinander angeordnet werden. Hierfür kommt eine HBox infrage. Diese erlaubt eine horizontale Anordnung seiner Kindelemente. Ohne weitere Einstellungen könnte man einfach die 6 ImageViews nacheinander einfügen. Hierauf folgen aber Probleme. Eine HBox passt sich immer automatisch an seine Kinder an und wächst gegebenenfalls in der Breite. Die ImageViews müssen aber quadratisch bleiben und brauchen einen Referenzwert für ihre Breite. Die Breite der HBox könnte nicht genommen werden, da diese an die Summe der Breite aller seiner Kinder gebunden ist. Dies würde zu einer ringförmigen Abhängigkeit führen. Man könnte alternativ die HBox in einer VBox untergliedern, dessen Breite als Referenzwert genommen werden kann, denn nur seine Höhe wächst mit der Höhe seiner Kinder. Dies würde aber bei insgesamt 6 Kombinationen für unnötige Komplexität sorgen.

2. GridPane als Kombinations-Container

Eine GridPane würde es auch erlauben, dass Elemente nebeneinander angeordnet werden könnten, sowohl vertikal als auch horizontal. Ein GridPane enthält alle Vorteile einer HBox, bietet aber noch zusätzliche Vorteile. Ein GridPane kann in seiner Größe und Zellengröße exakt beschränkt werden. Dessen Layout kann ebenfalls als Referenzwert für die ImageViews genommen werden. Mit einem GridPane ließe sich einfacher garantieren, dass die ImageViews in den Zellen quadratisch aussehen würden. Ein weiterer Vorteil, den GridPanee mit sich bringen, wäre die Koordinatenweise Erreichbarkeit der einzelnen ImageViews durch Spalten- und Zeilenangabe. So kann jede ImageView Zelle exakt angesprochen werden. Diese sind die Hauptgründe, warum die Spielerhände und das Spielfeld jeweils GridPanee als Container für ihre ImageViews verwenden.

3. Kombinationsanordnung

Es gibt jeweils 6 verschiedene Punktekombinationen, wobei nur 2 davon die komplette Segmentlänge von 6 ausnutzen. Die restlichen Kombinationen erlauben willkürliche Spielsteine dazwischen oder weitere Kombinationen. Damit aber alle Kombinationen von der Länge her konsistent aussehen, empfiehlt es sich, die Kombinationen, die nicht die volle Segmentlänge ausnutzen, mit leeren Spielsteinen aufzufüllen. Dies bringt den Vorteil mit sich, dass die Punktetabelle sauberer aussieht, aber auch, dass alle ImageViews aller Kombinationen gleich groß aussehen. So bleiben alle Kombinations-Container GridPanes gleich groß.

4. Design

Für das weitere Design lässt sich CSS hervorragend verwenden. So können Elemente in einer HBox zentriert werden und Kombinationen ließen sich durch Trennlinien abgrenzen. Wie bei den anderen Elementen im Seiten-Container ließe sich wieder ein schwarzer abgerundeter Rahmen einbauen, damit die Punktetabelle als Element von den anderen getrennt wahrgenommen werden kann. Dies sorgt weiter für Ordnung und Konsistenz.

Realisationsbeschreibung

Die optionale Punktetabelle wird mithilfe eines GridPanes realisiert. Das GridPane besteht aus 2 Spalten und 7 Zeilen, wobei die erste Zeile die Tabellenbeschreibung enthält. Die linke Spalte nimmt mit ihren Kombinationen den meisten Platz mit 75% ein. Die zweite Spalte beinhaltet mit den restlichen 25% die Punkte Labels zu den passenden Kombinationen. Die erste Zeile enthält die Labels zur Beschreibung der jeweiligen Spalte. Die erste Spaltenbeschreibung trägt den Text „Tile Combination“ und die zweite Spaltenbeschreibung trägt den Text „Points“.

Die restlichen Zeilen enthalten die Kombinationen und die korrespondierenden Punkte. Die erste Spalte trägt die Kombinationen, welche jeweils durch GridPanes dargestellt werden. Diese sind wiederum in einer HBox-VBox-Kombination umschlossen, um sie zu zentrieren. Außerdem kann auf das Layout der umschließenden Boxen später zurückgegriffen werden, um das GridPane Layout anzupassen. Jedes GridPane besitzt 6 Spalten und eine Zeile. Jede Zelle enthält jeweils ein ImageView für ein Image eines Spielsteins, welcher in der Kombination vorkommt. Die Breite GridPanes ist an der Breite seiner äußeren VBox gebunden und seine Höhe an seiner eigenen Breite geteilt durch die Anzahl seiner Spalten. Dadurch wird garantiert, dass jede Zelle quadratisch ist und somit die ImageViews auch. Die ImageViews sind wieder an das Layout ihres umgebenen GridPanes gebunden. Die Kombinationen werden mittels einer Hilfsfunktion gesetzt, in der das 6×1 GridPane und eine Liste von Spielsteinen übergeben wird. Dieses GridPane wird entsprechend der Kombination gefüllt. Kombinationen, die nicht die volle Arraylänge ausnutzen, werden mit leeren Spielsteinen gefüllt. Alle ImageViews haben weiterhin einen horizontalen, konstanten Abstand voneinander.

Die zweite Spalte trägt die Punkte Labels, jeweils eingeschlossen von einer zentrierenden HBox, welche die korrespondierenden Punkte als Text enthält. Die Sechsling Kombination enthält den Text „WIN“ als Zeichen für einen sofortigen „Win of Sixes“. Mit Hilfe von CSS wird das GridPane wie alle anderen Elemente des seitlichen Informations-Containers durch einen schwarzen abgerundeten Rahmen umgeben. Zusätzlich wird jede Zeile durch einen unteren schwarzen Rahmen abgegrenzt, um die Punktekombinationen visuell anzugrenzen.





































Tile Combination	Points
     	6
     	1
     	3
     	5
     	7
     	WIN

Abbildung 41: Optionale Punktetabelle

2.2.10. GUI-Darstellung des „New Game“ Menüs

Problemanalyse

Für das Spiel Crosswise sollte es möglich sein, ein neues Spiel zu starten. Natürlich bräuchte man hierfür ein Menü. Dieses muss alle notwendigen Elemente enthalten, die zum Starten eines neuen Spiels notwendig sind. Darunter zählt u.a. die Spieleranzahl. Es sollen entweder 2 oder 4 Spieler ein Spiel spielen können. Dazu müssen die Spielernamen für die jeweiligen Teams angegeben werden können und ob diese durch eine KI gespielt werden sollen. Letztlich muss es möglich sein, diese eingetragenen Informationen zu übermitteln und somit ein neues Spiel zu starten. Das Menü soll für den Spieler so intuitiv wie möglich bedienbar sein. Der Spieler sollte sehen können, was möglich ist, was nicht und was möglich sein kann.

Realisationsanalyse

1. Overlay vs. Fenster + Controller

Eine Möglichkeit, das „New Game“ Menü zu erstellen, wäre ein neues Fenster zu öffnen und darin das Menü zu erstellen. Jedoch bringt ein neues Fenster den Nachteil mit sich, dass diese wie alle Fenster die obere Leiste mit den Optionen zum Verschieben, Minimieren, Maximieren und Schließen enthält. Dies füllt das Sichtfenster unnötig. Der Spieler wird keinen Nutzen darin sehen, das Menü zum Beispiel zu minimieren. Vielleicht möchte er das Menü größer machen oder schließen. Das kann man dem Spieler aber auch durch andere Wege abnehmen. Ein neues Fenster zu erstellen bietet dem Spieler somit Optionen, die er höchstwahrscheinlich nicht nutzen wird.

Dennoch ist es valide anzunehmen, dass der Spieler das Menü eventuell schließen oder vergrößern will. Dies wäre mit einem neuen Fenster sehr einfach. Das neue Fenster könnte man dann separat in einer neuen .fxml Datei erstellen und einbinden. Dies erfordert aber ggf. einen neuen Controller, der separat erstellt werden muss.

Es bietet sich aber auch eine andere Option an, und zwar das Menü auch im User Interface Controller zu generieren. Dieser wäre nicht allzu komplex aufgebaut und kann leicht per Code erstellt werden. Hierfür kann eine weitere Ebene im bereits bestehenden des StackPane des Haupt-Grids verwendet werden. Das Menü rückt hierbei beim Anzeigen in die vordere Ebene und beim Verschwinden wieder nach hinten. Für das Menü können wieder wahlweise Nodes verschiedener Typen verwendet werden.

Der Vorteil, das Menü in der bestehenden Benutzeroberfläche zu integrieren, wäre dass sich das Menü an der Fenstergröße automatisch anpassen würde, ohne dass der Nutzer dies wie bei einem neuen

Fenster manuell tun müsste. Dies nimmt dem Nutzer die Mühe und sorgt für eine bessere User Experience (UX). Das Menü könnte auch immer zentriert mittig im Fenster liegen, sodass der Nutzer das Fenster nicht aus den Augen verlieren kann.

Dennoch besteht der Punkt des schließen des Menüs. Einen extra Button dafür erstellen, um das Menü zu schließen wäre anstrengend und kann leicht durch ein neues Fenster gelöst werden. Hierbei muss man sich aber die Frage stellen, ob der Spieler überhaupt jemals das Menü schließen will. Er hat sich doch an allerersten Stelle dafür entschieden, ein neues Spiel starten zu wollen. Dennoch soll die Möglichkeit, das Menü zu schließen, dem Nutzer nicht genommen werden. Der Nutzer soll so viel Freiheit wie möglich gegeben werden, damit er denkt, er habe die Kontrolle.

2. JavaFX-Komponenten

Nun geht es um die Elemente, die im Menü benötigt werden, um ein neues Spiel zu starten. Folgende Komponenten sind vorstellbar:

- Auf **HBox**, **VBox**, **GridPane**, **Label** muss nicht mehr eingegangen werden, da diese bereits in den vorherigen Punkten ausgiebig beschrieben wurden. Diese gelten weiterhin als allgemeine Struktur Elemente.
- **CheckBoxen** eignen sich hervorragend für das aus- & abwählen von Optionen, insbesondere dafür, ob eine Spieler eine KI sein oder nicht. Gleiches ist vorstellbar für die Auswahl der Spieleranzahl, in der der Spieler zwischen 2 und 4 Spielern wählen muss.
- **RadioButtons** können sich auch für die Auswahl der Spieleranzahl eignen. Diese würden ausschließen, dass mehrere RadioButtons gleichzeitig ausgewählt werden können. Diese machen aber nur Sinn, wenn sie an einem gebündelten Ort z.B. untereinander aufgelistet dargestellt werden. Die gleiche Funktionalität kann aber auch mit CheckBoxen erzielt werden.
- **TextFields** können zur Eingabe der Spielernamen verwendet werden. Diese können außerdem mit einem Initialtext versehen werden, um dem Spieler anzudeuten, dass dort hineingeschrieben werden kann.
- Ein **Button** eignet sich ideal für die abschließende Übermittlung der Spielerkonfiguration und das Starten des Spiels.

3. CheckBox vs. RadioButton

Hier lohnt es sich, ein wenig über die CheckBoxen und RadioButtons zu diskutieren. Beide bieten grundsätzlich das Gleiche an, jedoch unterstützen Checkboxes Mehrfachauswahlen und RadioButtons nicht. Die Auswahl der Spieleranzahl suggeriert eher die Nutzung von RadioButtons, da nur eine Option zurzeit ausgewählt werden kann. Das gleiche zwar auch mit CheckBoxen erzielt werden, wenn man dies im Code etwas anpasst, doch RadioButtons bieten alles schon nativ an. Dennoch kommt mit der Nutzung von CheckBoxen sinnvoller aus, da RadioButtons nur Sinn, wenn mehrere Auswahlmöglichkeiten bestehen und wir haben nur zwei zur Auswahl. Außerdem machen RadioButtons mehr Sinn, wenn diese vertikal untereinander aufgelistet angeordnet sind. Dies ist nicht der Fall. Da ich mich aber designtechnisch mit den Elementen auseinandergesetzt habe, muss ich sagen, dass CheckBoxen visuell ansprechender aussehen. Deshalb bevorzuge ich eher CheckBoxen über RadioButtons wegen des Design-Aspektes.

4. Design

Da die Erstellung via Code mehr Kontrolle bietet, gibt dies mehr Spielraum für visuelle Feinheiten. Man kann z.B. wenn das Menü im Vordergrund ist, alles andere verdunkeln lassen. Dies ist sehr einfach möglich, indem der Haupt-Container einen schwarzen Hintergrund mit einer gewissen Transparenz erhält. Dies trägt indirekt den Vorteil mit sich, dass man mit nichts dahinter mehr interagieren kann, da das Menü und die Verdunkelung im Vordergrund sind und alle Spielelemente auf dem Spielfeld überdecken.

Viele weitere designtechnischen Anpassungen sind insbesondere durch Ränder und Padding möglich. Diese weiter zu beschreiben, würde aber den Rahmen der Dokumentation sprengen. Hauptziel ist es, für Konsistenz zwischen allen Elementen der Benutzeroberfläche zu sorgen. Dies kann u.a. mit CSS hervorragend gelöst werden.

Realisationsbeschreibung

Das „New Game“ Menü wird als Overlay auf dem StackPane der Benutzeroberfläche erstellt und wird bei der Erstellung eines neuen Spiels ggf. in den Vordergrund oder Hintergrund gerückt. Das Overlay liegt in einer HBox-VBox Kombination, welche schwarz und leicht transparent gefärbt werden. Alles Dahinterliegende wird also abgedunkelt und der Spieler kann damit nicht mehr interagieren. Der Haupt-Container ist eine BorderPane, welche den gleichen Gradienten Hintergrund der Benutzeroberfläche besitzt. Sein äußerer Rahmen ist extra breit, weiß und abgerundet, damit Konsistenz zwischen den Elementen der Benutzeroberfläche herrscht. Der Haupt-Container wird zentriert in der Mitte des Fensters angeordnet.

Der obere Bereich des Haupt-Containers wird mit einem Header belegt, welches ein in einer HBox zentriertes Label ist und den Text „NEW GAME“ trägt. Der untere Bereich des Haupt-Containers enthält einen Button, zum Übermitteln der Spielerkonfiguration und zum Starten eines neuen Spiels. Der Button wird via CSS gestylt.

Es folgen nun die restlichen Elemente des Menüs. Diese befinden sich in dem Hauptbereich der **BorderPane**. Darin befindet sich wieder eine HBox-VBox-Kombination, welche die Elemente zentriert und vertikal anordnet. Zuerst werden zwei **CheckBox** Elemente mit **Labels** horizontal nebeneinander angeordnet. Diese stehen für die Auswahl zwischen 2 und 4 Spielern. Diese sind dafür verantwortlich, wie sich die folgenden Spielerformulare ändern.

Es folgen die zwei untereinanderliegenden Team Konfigurations-Formulare, in denen der Spieler angeben können soll, wie die Spieler heißen und ob diese von einer KI gespielt werden sollen. Dies wird einfach über **TextFields** und **Checkboxes** geregelt. Jedes Formular ist ein 1 × 3 GridPane, welches in der ersten Zeile den Teamnamen anzeigt, in der zweiten Zeile das Formular für den ersten Spieler und in der letzten Zeile das Formular des zweiten Spielers enthält.

Je nachdem, ob es nur 2 oder 4 Spieler gibt, wird der jeweils zweite Spieler jedes Teams aktiviert oder deaktiviert. Dies soll zu Folge haben, dass der Spieler damit ggf. interagieren kann oder nicht. Wenn das Formular deaktiviert ist, kann der Spieler trotzdem sehen, dass es einen weiteren Spieler geben könnte. Dies soll für mehr Intuitivität sorgen. So können die Spieler sofort sehen, dass die Auswahl der Spieleranzahl unmittelbar mit der Anzeige der Formulare zusammenhängt.

Designtechnisch werden an vielen Stellen via CSS kleine und größere Anpassungen vorgenommen. Diese hier zu nennen, würde jedoch den Rahme der Dokumentation sprengen und kann daher vernachlässigt werden.

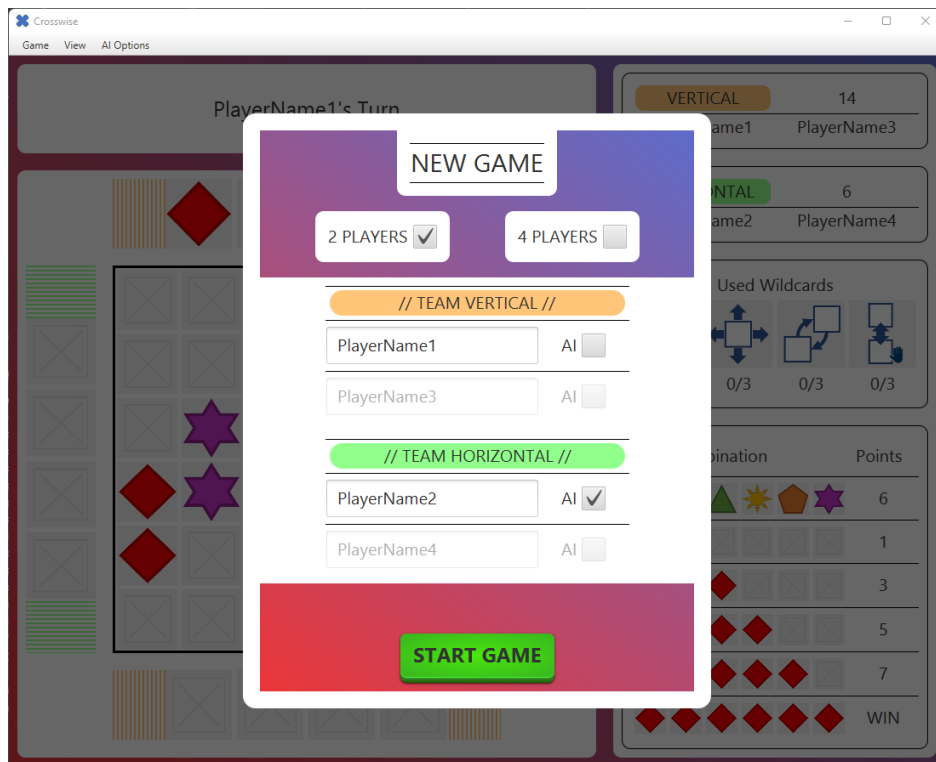


Abbildung 42: „New Game“ Menu

2.2.11. GUI-Darstellung des „Game Over“ Menüs

Problemanalyse

Zu Spielende muss angezeigt werden, welches Team gewonnen hat. Dazu zählen die erreichten Punkte beider Teams und/ oder ob durch ein „Win of Sixes“ gewonnen wurde. Eine nähere Beschreibung zum „Win of Sixes“ lässt sich im Benutzerhandbuch (1.3.4. Spielende und Punktebewertung) finden.

Realisationsanalyse

1. Overlay vs. Separates Fenster

Es lohnt sich wie beim „New Game“ Menü (siehe Abschnitt 2.2.10) zu diskutieren, ob es sich eher anbieten würde, ein neues Fenster für das „Game Over“ Menü zu erstellen und damit verbunden einen zugehörigen Controller oder doch lieber ein Overlay im User Interface Controller zu gestalten, welches dann zu Spielende in den Vordergrund gerückt werden könnte.

Hierbei möchte der Spieler aber sicher die Möglichkeit haben, das Spielfeld zu betrachten und dafür das „Game Over“ Menü zur Seite verschieben oder schließen. Dies wäre bei einem Overlay nicht ohne weiteres möglich. Dieser würde das Spielfeld nur verdecken. Es wäre also vorteilhaft, das „Game Over“ Menü verschieben und ggf. schließen zu können. Demnach würde es sich deshalb anbieten, für dieses Menü ein separates Fenster zu gestalten. Ein Fenster würde nativ solche Funktionalitäten bereits zur Verfügung stellen.

2. „Win of Sixes“ Punkteanzeige

Es ergibt sich aber das Entscheidungsproblem der Anzeige der Punkte. Einerseits kann man bei jedem Spielende die erreichten Punkte beider Teams anzeigen lassen. Auf der anderen Seite könnte man im Falle eines „Win of Sixes“ dies auch anstelle der Punkte anzeigen lassen. Denn ein „Win of Sixes“ ist immer ein sofortiger Gewinn des Teams. Die erreichten Punkte des Gewinnerteams wären dann irrelevant. Der Spieler würde sich dafür nicht weiter interessieren. Wenn dem Spieler die Punktzahl des „Win of Sixes“ Gewinnerteams doch interessiert, kann dieser ja jederzeit die optionalen

Punkteleisten anzeigen lassen und die Punkte dann selbst ausrechnen. Dennoch sehe ich es als überflüssig an, bei einem „Win of Sixes“ zusätzlich die Punkte des Gewinnerteams anzuzeigen. Insbesondere wenn das Gewinnerteam trotz eines „Win of Sixes“ eine niedrigere Punktzahl als das Verliererteam hätte, wäre es für das Gewinnerteam nur verwirrender. Deshalb könnte man anstelle der Punkte einfach ein aussagekräftiges Symbol nehmen, welches den „Win of Sixes“ verdeutlicht.

3. Node Elemente

Wie bei den anderen Elementen der Benutzeroberfläche lässt sich wieder aus einem breiten Arsenal aus verschiedenen JavaFX-Komponenten wählen. Die Komponenten, welche infrage kommen würden, werden folgend kurz dargestellt:

- **HBox** und **VBox** haben die hervorragenden Fähigkeiten, Elemente zu zentrieren und vertikal bzw. horizontal anzuordnen. Außerdem können ihre Layout-Grenzen als Referenzwerte für dessen Kinder benutzt werden. Außerdem können sie als sog. „SpacerNodes“ benutzt werden, um für einen gleichmäßigen Abstand zwischen Menü-Elementen zu sorgen.
- **GridPanes**, welche die Teamübersicht strukturiert und dynamisch halten kann. Zudem kann ohne große Mühe das Layout beliebig angepasst werden.
- **Labels** können alle wichtigen textuellen Informationen wie das Gewinnerteam, Teamnamen, Punktzahl, usw. enthalten.

Weitere JavaFX-Komponenten zu benutzen ist vorstellbar.

4. Design Aspekte

Es würde sehr gut passen, wenn das neue Fenster den gleichen Stil wie die Benutzeroberfläche hätte. Demnach würde es visuell sehr ansprechend aussehen, wenn das neue Fenster den gleichen Gradienten Hintergrund hätte wie das Main-Grid hätte. Der Container, der die Elemente beinhaltet, könnte dann ebenfalls einen weißen abgerundeten Hintergrund haben. Der Stil des neuen Fensters wäre somit mit dem Haupt-Fenster konsistent und sähe eher wie eine Erweiterung der Benutzeroberfläche aus, als ein separater Teil, obwohl es dennoch ein neues Fenster ist. Das gesamte Styling ließe sich relativ einfach mithilfe einer externen CSS Datei gestalten. Dafür müssen nur die jeweiligen Nodes bei ihrem Klassennamen oder ihrer manuell vergebenen ID adressiert werden.

Außerdem wäre es den Mehraufwand wert, wenn ein Teil des „Game Over“ Menüs einen Hintergrund hätte, welcher zum Gewinnerteam passt. So kann der Spieler sofort erkennen, welches Team gewonnen hat. Einerseits durch das Gewinner Label und andererseits durch den Hintergrund. Diese Aspekte würden sich komplementär unterstützen.

Ebenfalls würde die Team Übersicht mit den erreichten Punkten durch extra Design-Anpassungen profitieren. Zum Beispiel könnten die Team-Namen entsprechend gefärbt werden. So wurde es ebenfalls in der Team-Infos-Übersicht in dem Seiten-Container gemacht. Das Menü würde dadurch polierter wirken und sorgt wieder für Konsistenz. Da die beiden Team-Übersichtsbereiche das gleiche Design haben, würden Teamnamen Farben für die Unterscheidung der Teams ebenfalls beitragen.

Realisationsbeschreibung

Das „Game Over“ Menü wird als neues Fenster mit seinem eigenen Controller implementiert. Die komplette Erstellung findet via Code im Controller statt. Das Wurzelement ist ein `AchorPane`, welches eine `HBox-VBox` Container-Kombination enthält. Der Container ist für die zentrierte, vertikale und horizontale Ausrichtung der Elemente zuständig. Hinzu kommt eine `VBox`, welche folgend als Haupt-Container bezeichnet wird. Dieser Container enthält alle weiteren Elemente des „Game Over“

Menüs. Die Elemente werden vertikal von oben nach unten angeordnet. Folgende Elemente werden in absteigender Reihenfolge dem Haupt-Container hinzugefügt:

- SpacerNode, Header, (Gewinner Team), SpacerNode, Team 1 Übersicht, SpacerNode, Team 2 Übersicht, SpacerNode

Folgend werden die einzelnen Elemente näher erläutert.

Ein **SpacerNode** ist eine einfache VBox, dessen Priorität des vertikalen Wachstums auf ALWAYS eingestellt wurde. Dieser SpacerNode nimmt so viel Platz ein möglich, ohne andere Element zu behindern. Die Hauptaufgabe der SpacerNodes ist die gleichmäßige Aufteilung der dazwischenliegenden Elemente, sodass sie alle den gleichen Abstand haben.

Der **Header** ist ein einfaches Label in einer zentrierenden HBox und beinhaltet entweder den Text „WINNER“ oder „DRAW“, je nachdem, ob ein Team gewonnen hat, oder ob ein Gleichstand erzielt wurde.

Das **Gewinner-Team** ist ebenfalls ein Label in einer zentrierenden HBox und beinhaltet den Namen des Gewinner-Teams „TEAM VERTICAL“ oder „TEAM HORIZONTAL“. Wenn kein Team gewonnen hat, wird dieses Label nicht im Menü mit aufgenommen und nur der Header und die Team-Übersichten werden angezeigt.

Die **Team-Übersicht** beider Teams ist jeweils ein in einer zentrierenden HBox liegendes 1×2 GridPane. Die erste Zeile enthält eine HBox mit einem zentrierten Label, welche den Team-Namen mit Akzentschrägstrichen „// TEAM VERTICAL //“ oder „// TEAM HORIZONTAL //“ anzeigt. Dieses Label wird durch CSS in der korrespondierenden Team Farbe eingefärbt. Der Hintergrund wird an den Ecken abgerundet.

Die zweite Zeile des GridPanes enthält ein weiteres 2×2 GridPane, welches links die Teamnamen und rechts die erreichte Punktzahl enthält. Wenn es zwei aktive Spieler pro Team gab, bekommt jede Zeile in der ersten Spalte ein in einer HBox zentriertes Label, welches den Spielernamen des Teams anzeigt. Wenn es nur einen aktiven Spieler pro Team gab, wird nur ein Label erstellt, welches beide Zeilen umspannt. In der rechten Spalte befindet sich ebenfalls ein zentriertes Label, welches immer beide Zeilen umspannt und die Punktzahl anzeigt. Dieses Label wird mit extra groß gestaltet. Falls ein Team einen „Win of Sixes“ erzielt hatte, wird dies durch den Text „SIXES“ dargestellt. Warum hier keine Punkte angezeigt werden, kann aus der Realisationsanalyse unter dem Punkt 2 („Win of Sixes“ Punkteanzeige) entnommen werden.

Der Haupt-Container wird durch CSS umfangreich gestylt. Der Hintergrund passt sich dem Gewinnerteam an. Wenn das vertikale Team gewonnen hat, wird der Hintergrund mit orangen vertikalen Balken gesetzt. Wenn das horizontale Team gewonnen hat, wird der Hintergrund mit grünen horizontalen Balken gesetzt. Falls ein Gleichstand zustande gekommen ist, wird ein grau-weißes diagonal gestreifter Hintergrund gewählt.

Damit der Haupt-Container wie die Elemente aus der Benutzeroberfläche aussieht, sollte dieser auch einen weißen abgerundeten Hintergrund haben. Der Hintergrund ist aber schon für das Gewinnerteam vergeben. Um dies zu umgehen wurde ein extra breiter weißer Rahmen um den Haupt-Container erstellt. Dies erzeugt die Illusion, dass es zwei Container mit verschiedenen Hintergründen gäbe. Dies sieht visuell nochmals weiter aufbereitet aus und sieht mit den anderen Benutzeroberflächen Elementen konsistent aus.

Damit die Labels des Headers und des Gewinner-Teams lesbar sind, wird dessen Hintergrund ebenfalls weiß abgerundet dargestellt. Beide Hintergründe verschmelzen miteinander, sodass beide Labels als

eines wahrgenommen werden. Ebenso haben die Team Übersicht-GridPanes einen weißen Hintergrund, damit der Text darin lesbar ist. Die Elemente werden zusätzlich von horizontalen Streifen strukturiert und akzentuiert.

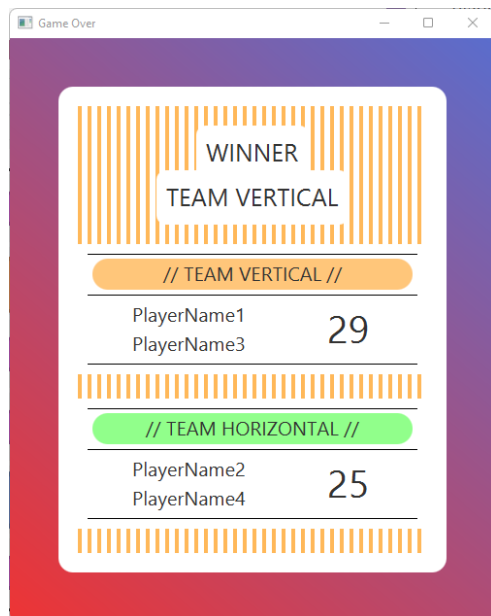


Abbildung 43: Game Over (Vertikales Gewinnerteam)

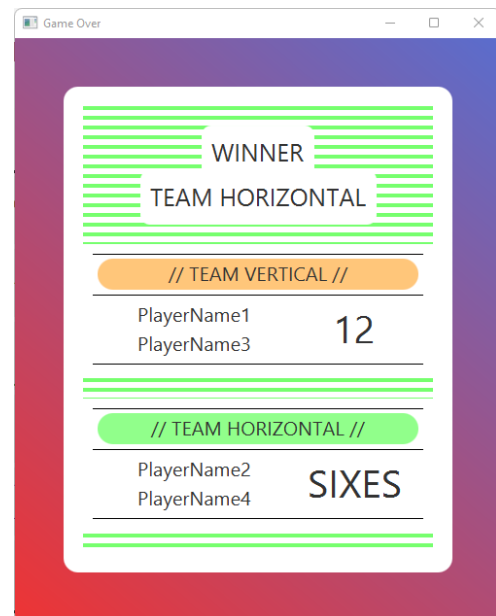


Abbildung 44: Game Over (Horizontales Gewinnerteam)

2.2.12. Implementierung der Spielstein Tasche

Problemanalyse

Für das Spiel Crosswise muss eine Spielsteintasche implementiert werden. Von dieser sollen zufällige Spielsteine gezogen werden. Die Spielsteine enthält exakt 6×7 Standard Spielsteine und 4×3 Wildcards. Von jedem Standard Spielstein also 7 Exemplare und von jedem Wildcard-Typ 3 Exemplare. Diese muss initialisiert und ggf. gespeichert werden können.

Realisationsanalyse

Für die Spielsteintasche lassen sich Datenstrukturen des Java Collection Frameworks anwenden. Da immer zufällige Spielsteine von der Spielsteintasche gezogen werden sollen, kann es Sinn machen, dass die Spielsteintasche intern zufällig generiert wird und dann als Queue gespeichert wird. Denn es muss nur vorne gezogen werden. Dies kann effizient mit einer **LinkedList Queue** geschehen, mit der am Anfang gepollt werden kann. Der gezogene Spielstein wird gleichzeitig auch direkt von der Spielsteintasche entfernt.

So steht nun die interne Initialisierung der Spielsteintasche infrage. Dafür kann man einfach eine **ArrayList** verwenden, diese mit der richtigen Anzahl an Spielsteinen füllen und mithilfe der **Random** Klasse zufällige Spielsteine davon nehmen, entfernen und der **Queue** hinzufügen. Das indizierte Zugreifen auf der **ArrayList** wäre effizient, das indizierte Löschen hingegen aber nicht. Da dies aber nur initial zum Spielstart passiert, sollte dies kein großes Problem darstellen.

Realisationsbeschreibung

Die Spielsteintasche wird als LinkedList Queue implementiert, welche intern mit einer ArrayList gefüllt wird. Diese ArrayList enthält alle vorkommenden Spielsteine, geordnet. Von dieser wird per **Random** Klasse zufällig ein Spielstein genommen und der Queue Spielsteintasche hinzugefügt. Die Queue muss dem Spiel nichts Weiteres anbieten, als die **poll** Möglichkeit. So kann das Spiel einen zufälligen Spielstein ziehen und die Spielsteintasche löscht ihn gleichzeitig effizient.

Zusätzlich gibt es die Option, die Spielsteintasche anhand eines existierenden Spiels zu rekonstruieren. Dies ist möglich, indem alle verwendeten Wildcards und alle Spielsteine auf den Spielerhänden und auf dem Spielfeld gezählt werden. Dann wird die Differenz mit deren maximalen Vorkommen gebildet und eine geordnete Liste von Spielsteinen kann der Spielsteintasche übergeben werden. Der Konstruktor arbeitet mit dieser initialen ArrayList und holt sich die einzelnen Spielsteine dann zufällig und packt diese in seine Queue. So kann die Spielsteintasche dennoch zufällig rekonstruiert werden.

2.2.13. Implementierung der Spielerzüge

Problemanalyse

Der menschliche Spieler muss in der Lage sein, einen Spielzug zu tätigen. Hierbei können Standard-Spielsteine oder Wildcards gespielt werden. Hierfür müssen separate Verhalten implementiert werden. Der Spieler macht einen Zug, indem er u.a. Spielsteine von seiner Hand auf das Spielfeld zieht. Bei Wildcards sind speziellere Konstellationen vorgesehen. Das Ziehen von Spielsteinen soll über Drag & Drop geschehen. Hierfür müssen die ImageViews der Spielerhände mit den entsprechenden Drag & Drop Event Handlern ausgestattet werden. Das Gleiche gilt für die ImageViews auf dem Spielfeld, welche die gezogenen Spielsteine u.a. akzeptieren sollen.

Realisationsanalyse

1. Drag & Drop Event Handler

Zuerst gilt es, die jeweiligen Image Views mit passenden Drag & Drop Event Handlern auszustatten. Es wird wahrscheinlich vonnöten sein, mehrere verschiedene Drag & Drop Event-Handler zu erstellen. Diese könnten in einer extra Klasse ausgelagert werden. Aufgabe ist es, die verschiedenen Handler für die jeweilige passende Zug Konstellation zu identifizieren und zu implementieren.

2. Multi-Phase-Spielzug

Bestimmte Wildcards erfordern eine Aktivierung, denn sie können nicht in einem Zug gespielt werden. Beispielsweise muss die MOVE Wildcard erst platziert werden. Dann erst soll ein Spielstein auf dem Spielfeld auf ein leeres Feld gezogen werden können. Demnach muss es in der Game-Instanz möglich sein, einen Multi-Phase-Spielzug als einen einzigen Spielerzug zu werten und zu unterstützen. Dafür darf die erste Phase einer 2-Phasen Wildcard keinen nächsten Spielerzug auslösen. Die Game-Instanz muss diesen Zug als Teil-Zug ansehen und dem Spieler erlauben, die 2. Hälfte seines Zuges noch auszuführen.

Realisationsbeschreibung

Um Spielerzüge tätigen zu können, müssen entsprechende Drag & Drop Even Handler gesetzt werden. Dies gilt sowohl für die ImageViews der Spielerhände, als auch für die ImageViews auf dem Spielfeld. Hierbei kommt ein breites Arsenal an Drag & Drop Handler zum Einsatz. Diese werden extra in einer statischen **DragHandlers**-Klasse definiert und sind in 3 grobe Gruppen aufgeteilt.

Die erste Gruppe beinhaltet die allgemeinen Drag & Drop Handler, welche die ImageViews der Spielerhände und Spielfelder bekommen. Die ImageViews der Spielerhände sollen gezogen werden können. Auf dem Spielfeld sollen diese ImageViews gegebenenfalls akzeptiert werden können. Wenn ein Standard Spielstein von der Spielerhand gezogen und auf ein freies Spielfeld gezogen wird, reagiert das ImageViews durch den setOnDragOver und setOnDragEntered Handler darauf. Zum einen akzeptiert das ImageView die Transfermethode, zum anderen zeigt er dies, indem ein grün leuchtender Effekt gesetzt wird. Via setOnDragExited Handler wird der Effekt wieder zurückgesetzt. Mit dem setOnDragDropped werden die Informationen des Dragboards vom gezogenen ImageView entnommen. Beim Platzieren eines ImageViews wird der Effekt zurückgesetzt, die Position berechnet und der Spielstein identifiziert, welcher platziert wurde. Die Identifikation ist nur möglich, da das

gezogene ImageView mithilfe des `onDragDetected` ein Dragboard mit sich zieht, in dem die Bildinstanz und dessen URL abgelegt werden können. So kann also erkannt werden, welcher Spielstein an welchem Ort platziert worden ist. Die `onDragDropped` Methode benötigt die Game-Instanz, damit beim Platzieren des Spielsteins ein **playerTurn** ausgeführt werden kann. Dies ist einer der wenigen Ausnahmen, bei der die Game-Instanz von der GUI benötigt wird.

Damit nur bestimmte Spielsteine nur auf bestimmte Spielfelder gezogen werden können, wird pro Handler individuell mit speziellen Bedingungen reagiert. Beispielsweise kann ein Standard Spielstein nur auf leere Spielfelder gelegt werden oder eine REMOVER Wildcard kann nur auf ein nicht-leeres Spielfeld gelegt werden.

Werden 2-Phase Wildcards gespielt, werden die anderen beiden Gruppen von Drag & Drop Handler benötigt. Das gesamte Spielfeld bekommt die sogenannten Board Drag Handler, damit das gesamte Spielfeld aufleuchtet und somit 2-Phase Wildcards überall auf dem Feld akzeptieren werden können. Die letzte Gruppe von Handlern ist notwendig, wenn eine 2-Phase-Wildcard aktiviert wurde und ImageViews vom Spielfeld auf dem Spielfeld gezogen werden sollen. Dann müssen die ImageViews auf dem Spielfeld „dragbar“ sein. Dies darf aber nicht immer der Fall sein, sondern nur, wenn eine solche Wildcard aktiviert wurde. Wenn also eine 2-Phase-Wildcard aktiviert wurde, werden den Spielfeldern neue Handler vergeben, um bereit für die zweite Phase zu sein.

Die Game-Instanz reagiert entsprechend auf das Ablegen von Spielsteinen auf Spielfelder, indem er einen **playerTurn** auslöst. Dort wird der abgelegte Spielstein erstmal logisch gesetzt, dann auch grafisch auf der Benutzeroberfläche angezeigt. Wenn Wildcards gespielt werden, muss das Spiel entsprechend auf diese besonderen Spielzüge reagieren.

Es gibt sogenannte 1-Phase-Wildcard-Züge und 2-Phase-Wildcard-Züge. Diese wurden bereits näher im Benutzerhandbuch beschrieben (siehe Abschnitt 1.3.3). 1-Phase Wildcard Züge können wie Standard Spielzüge in einem durchgängigen Zug ausgeführt werden. Der Spieler zieht einen Spielstein von seiner Hand auf ein Spielfeld und der Zug ist abgeschlossen. Bei 2-Phase-Wildcard-Zügen können diese nicht in einem durchgängigen Zug ausgeführt werden. Dazu zählen die MOVER, SWAPONBOARD und SWAPWITHHAND Wildcards. Diese müssen zunächst aktiviert werden, indem sie irgendwo auf das Spielfeld gelegt werden. Danach gelten diese als aktiviert. Der Spielerzug ist noch nicht vollendet und der gleiche Spieler ist noch am Zug. Danach kann der Spieler die 2. Phase vollziehen und seinen Zug beenden. Wie die Standard- und Wildcard-Züge genau vollzogen werden, wurde ebenfalls bereits im Benutzerhandbuch beschrieben (siehe Abschnitt 1.3.3).

Nach einem vollendeten Spielerzug bereitet sich die Game-Instanz auf den nächsten Spielzug vor. Dafür wird zunächst das Spielfeld evaluiert. Wenn das Spiel weitergehen soll, wird der nächste Spieler gesetzt, alle anderen Hände außer seiner versteckt und seine Hand wird aktiviert. Der nächste Spieler kann seinen Zug durchführen.

2.2.14. Implementierung der KI

Problemanalyse

Es soll einen KI-Spieler geben können, gegen den gespielt werden kann. Dieser folgt gewissen Regeln und bildet nach diesen Regeln einen KI-Zug. Wie die KI ihre Zug-Entscheidung trifft, wird im Punkt KI Spielzug Entscheidung näher erläutert (siehe Abschnitt 2.3.2.). Dort werden ebenfalls die Regeln erläutert, nach denen sich die KI orientiert. Aufgabe ist es, diese KI in das Spiel mit einzubinden, sodass diese neben menschlichen Spielern oder sogar komplett gegen andere KIs selbstständig spielen kann.

Realisationsanalyse

Der KI Zug ist im Grunde ein automatisierter normaler Spielerzug. Hierfür kann die gleiche **playerTurn** Methode in der Game-Instanz verwendet werden. Das Spiel müsste nur das Ergebnis der KI Zug Evaluation nehmen und ausgehend davon einen **playerTurn** aufrufen.

Wichtig zu beachten ist, dass die KI nicht selbstständig einen Zug macht. Das Spiel muss das für die KI erledigen. Das Spiel muss aber dafür wissen, ob der aktuelle Spieler eine KI ist oder nicht, dementsprechend könnte das Spiel einen KI-Zug triggern. Aufgabe ist es also, Einstiegspunkte für KI Spielzüge und eine Art Kreislauf zu realisieren, sodass die Game-Instanz automatisch entsprechende KI Spielzüge ausführt und für die nächsten Züge bereit ist. Demnach müssen alle Orte identifiziert werden, an denen die Spiel-Instanz einen KI-Zug initialisieren muss, wie z.B. am Anfang des Spiels, oder wenn ein Spiel geladen wird und die KI am Zug wäre.

Realisationsbeschreibung

Der KI ist ebenfalls ein Player und teilt die gleichen Eigenschaften wie dieser. Deshalb erbt der KI_Player von Player und erhält alle seine Member. Die KI implementiert ihrerseits weitere KI-spezifische Methoden. Es ist nun die Aufgabe der Game-Instanz, die KI im Spielfluss zu integrieren, denn im Gegensatz zum menschlichen Spieler kann die KI nicht eigenständig Züge ausführen, nur die Parameter für einen Zug liefern. Dies nutzt die Game-Instanz aus. Immer wenn eine KI spielen soll, nimmt die Game-Instanz die Zug-Parameter, welche die KI liefert und führt damit ggf. einen Zug aus. Hierbei wird die gleiche **playerTurn** Methode verwendet.

Damit die Game-Instanz weiß, wann diese einen KI-Zug ausführen soll, schaut diese auf den aktuellen Spieler und kann erkennen, ob dieser Spieler eine KI ist oder nicht. Wenn der Spieler eine KI ist, fordert die Game-Instanz die KI auf, einen besten Zug zu berechnen, welche die KI spielen würde. Diesen Zug entnimmt die Game-Instanz dann die notwendigen Parameter und führt im Namen der KI einen **playerTurn** aus.

Es ist insbesondere notwendig zu schauen, ob ein neues Spiel geladen oder erstellt wurde. Wenn der erste Spieler eine KI ist, soll das Spiel das erkennen und auch da einen Zug für sie ausführen. Problematisch wird es aber, wenn die KI keinen Zug liefern kann. Demnach kann die Game-Instanz dann keinen Zug ausführen. Dieses Problem wird gelöst, indem das Spiel einen KI-Zug anfordert und diesen dann prüft, ob der null ist oder nicht. Wenn der Zug null ist, dann kann kein Zug ausgeführt werden und das Spiel muss beendet werden. Dies passiert nur in den seltensten Fällen, z.B. wenn die KI am Anfang des Spiels nur Wildcards besitzt und diese nicht auf ein leeres Spielfeld spielen kann oder wenn der einzige Zug für die KI einen „Win of Sixes“ für das Gegnerteam herbeiführen würde. Die KI würde so einen Zug niemals als valide ansehen. So kann das Spiel überprüfen, ob ein KI Spielzug vorhanden ist, bevor er einen echten **playerTurn** für die KI ausführt.

Das Spiel führt einen KI-Zug in folgenden drei Fällen aus: Die ersten beiden Fälle decken den Spielstart ab, also wenn ein neues Spiel gestartet oder geladen wurde. Wenn der aktuelle Spieler eine KI ist, wird ein KI-Zug berechnet und ausgeführt. Der letzte Fall ist der wichtigste, welcher den Kreislauf realisiert. Immer, wenn der nächste Spielerzug vorbereitet wird, wird überprüft, ob der nächste Spieler eine KI

ist. Hierfür muss aber der vorherige Spieler keine KI gewesen sein. Dies ist relevant, da schon der nächste Spieler gesetzt wurde und hier die Zukunft betrachtet wird. Ohne diese Bedingungen können keine KI Züge ausgeführt werden, wenn sowohl menschliche als auch KI Spieler ein Spiel spielen.

2.2.15. Implementierung des Spielstandes

Problemanalyse

Da das Spiel Crosswise eine Option zum Speichern und Laden von Spielständen anbieten soll, erfordert dies natürlich die Implementation eines Spielstandes. Diese soll im .json Format gespeichert werden können. Aus dieser Datei soll der Spieler ein bestehendes Spiel speichern oder ein bereits laufendes Spiel laden können. Dem JSON Spielstand dürfen weitere Informationen hinzugefügt werden. Dennoch muss ein Spiel auch im Original-Format geladen werden können. Das genaue JSON Format wird näher unter (2.5 Dateien) beschrieben.

Realisationsanalyse

Die Realisierung des Spielstandes lässt sich mit **Gson** lösen. Die **Gson** Klasse bietet Methoden zum Erstellen einer .json Datei aus einer Klasseninstanz und das Lesen von Spielständen im JSON Format zu einer Klassen-Instanz. Es müssen nur passende Datenhaltungsklassen definiert werden, damit die Gson Klasse daraus eine .json Datei generieren kann. Es liegt nahe, direkt die **Game** Instanz zu nehmen und zur .json Datei zu wandeln. Problematisch wird es bei der Repräsentation der Spielsteine, denn intern wird mit Enum Werten gearbeitet. Das vorgegebene JSON Format sieht es aber vor, dass jegliche Spielsteine mit ihren Ordinalitäten repräsentiert werden sollen. So kann sichergestellt werden, dass Tests mit Zahlenwerten arbeiten können, ohne die Enumeration der Spielsteine zu kennen.

1. Datenhaltungsklassen

Deshalb scheint es notwendig, eine eigene Spieldaten-Klasse zu definieren, denn die Game-Instanz enthält ein Spielfeld, das aber mit Enum Werten repräsentiert wird. Das gleiche gilt für die **Player** Instanzen, welche ihre Spielerhand intern als Enum Liste speichern. Deshalb bietet es sich an, also für die Game- und die Player-Instanz(en) eigene Datenhaltungsklassen zu definieren, welche keine Logik, sondern nur Daten halten und dessen Zugriff via Getter ermöglicht.

2. Aktueller Spieler Repräsentation : Integer vs. int

Für die Repräsentation des aktuellen Spielers ist es naheliegend, diese Nummer als einfachen int-Wert abzuspeichern. Wenn dies aber gemacht wird, ergibt sich ein fatales Problem. Wenn die Spielstands-Datei den aktuellen Spieler nicht enthält, dann ist dieser nicht vergeben. Das heißt aber nicht, dass ein geladenes Spiel den Wert dann nicht setzt. Das Problem ist, dass bei nicht gesetzten int-Werten immer ein Standardwert von 0 zugewiesen wird. Dies würde Probleme für den korrekten Spielfluss sorgen. Deshalb macht es Sinn, den aktuellen Spieler als Integer zu repräsentieren, dessen Wert null sein und somit abgefangen werden kann. Eine 0 kann man nicht abfangen, da es einen 0. Spieler gibt.

3. Rekonstruktion der Spielsteintasche

Das JSON Format ist ohne Speicherung der Spielsteintasche vorgegeben. Dennoch ist es erlaubt, zusätzliche Daten zu speichern. Voraussetzung ist aber, dass ein Spiel auch immer im ursprünglichen JSON Format geladen werden können soll. Dies macht es notwendig, dass die Spielsteintasche intern abgeleitet werden müsste. Dies kann realisiert werden, indem man die bereits benutzten Wildcards, die Spielerhände und das Spielfeld in Betracht zieht und die fehlenden Spielsteine, die also noch gezogen werden können sollen, berechnet. Dies ist jedoch mit erhöhtem Aufwand verbunden.

Realisationsbeschreibung

Der Spielstand wird mithilfe einer **GameFileManager** Klasse implementiert. Diese Klasse erstellt Spielstände und kann diese laden. Das Erstellen von Spielständen wird mit der **Gson** Klasse realisiert, welche Klassen ins JSON Format konvertieren kann. Dafür werden für die Game-Instanz und Player-Instanzen nur separate Datenhaltungsklassen benötigt. Folgend werden die GameData und die PlayerData Datenhaltungs-Klassen beschrieben:

1. GameData

Die **GameData** Klasse beinhaltet alle wichtigen Elemente des Spiels. Darunter zählen das Spielfeld, ein Array von Spielerdaten (PlayerData), der aktuelle Spieler und die bereits genutzten Wildcards als Array. Zuletzt kann die Spielsteintasche auch abgespeichert werden.

Das Spielfeld wird als 2D Array von Spielstein-Ordinalitäten gespeichert. Da das Spielfeld intern mit einem 2D Array von Enum Werten repräsentiert wird, muss diese erst in Ordinalitäten umgewandelt werden. Dies ist einfach mit verschachtelten Stream möglich.

Die Spieler werden durch ein Array von PlayerData Klassen beschrieben. Der aktuelle Spieler ist eine einfache Integer, welche nicht null sein darf. Darauf folgt ein einfaches int-Array, welches zu den verschiedenen Wildcards die Anzahl der Nutzungen speichert. Die Indices stimmen in aufsteigender Reihenfolge mit folgenden Wildcards überein:

- REMOVER (0), MOVER (1), SWAPONBOARD (2), SWAPWITHHAND (3)

Zuletzt wird die Spielsteintasche in GameData gespeichert, welche ein int-Array von Spielstein Ordinalitäten beinhaltet. Da die Spielsteintasche intern eine Queue von Enum Werten ist, muss diese ebenfalls zu einer Liste von Spielstein Ordinalitäten umgewandelt werden. Dies ist ebenfalls einfach mit Java-Streams möglich.

2. PlayerData

Die **PlayerData** Klasse beinhaltet alle wichtigen Informationen zum jeweiligen Spieler. Darunter zählen der Spielername als String, boolean Flags, ob der Spieler aktiv ist und ob er eine KI ist und zuletzt seine Spielerhand.

Die Spielerhand muss wie das Spielfeld und die Spielsteintasche als int-Array gespeichert werden. Intern wird die Hand jedoch als Liste von Enum Werten gespeichert. Auch diese kann einfach in eine Liste bzw. Array von Spielstein Ordinalitäten umgewandelt werden.

3. Spielstand Erstellung & Laden

Wenn das Spiel gespeichert werden soll, liefert die Spielinstanz ein GameData Objekt, woraus mit **Gson** einfach in eine JSON Datei erstellt werden kann. Diese kann dann gespeichert werden. Aus diesem Spielstand kann dann Spiel geladen werden, indem **Gson** den JSON Spielstand wieder in eine GameData Klasse umwandelt. Mit dem erhaltenen GameData Objekt kann der Konstruktor des Spiels ein Spiel laden.

4. Spielstand ohne Spielsteintasche

Es sollen auch Spielstände geladen werden können, die keine Spielsteintaschen beinhalten. Dies wird mit einer Fallunterscheidung im Konstruktor des Spiels geregelt. Falls keine Spielsteintasche in GameData existiert, wird die Spielsteintasche durch GameData rekonstruiert. Darauf wurde bereits in der Implementierung der Spielsteintasche eingegangen (siehe Abschnitt 2.2.12.)

2.2.16. Implementierung des Logs

Problemanalyse

Für den Programmierer soll der Spielverlauf geloggt werden. Dies soll in eine Logdatei geschrieben werden und für den Programmierer sichtbar sein. Bei jedem Spielstart soll diese Datei überschrieben bzw. neu erstellt werden. Zum einen soll zu Spielbeginn die Initialisierung dokumentiert werden, also die Spielerkonfiguration mit ihren Händen. Zum anderen soll jeder Zug für den Programmierer nachvollziehbar ausgegeben werden. Dazu soll nach jedem Zug das neue Spielfeld ausgegeben werden. Dem Programmierer ist es frei überlassen, die Ausgaben ausführlicher zu gestalten. Der Text soll in der Logik erstellt werden dürfen und dient nicht der Kommunikation mit dem Benutzer. Nach jedem Schreibvorgang soll die Datei geschlossen werden, damit im Falle eines Fehlers keine Informationen verloren gehen.

Realisationsanalyse

Das Logging kann sehr einfach mit der `java.util.logging.Logger` Klasse implementiert werden. Diese bietet einfache Methoden zum Schreiben eines Logs in eine spezielle Log-Datei. Hierbei muss man sich keine Sorgen machen, die Log-Datei bei jedem Schreiben schließen zu müssen. Lediglich die Log-Datei muss beim Spielstart neu erstellt bzw. überschrieben werden. Jedoch bringt die Logger-Klasse den Nachteil mit sich, dass sie zu viele Informationen loggt. Für das folgende Problem sind beispielsweise Zeitstempel nicht von Relevanz. Um den Logger individuell zu gestalten, müsste man einen benutzerdefinierten Formatter schreiben. Dies kann jedoch in weitere Komplexitäten ausarten.

Eine weitere einfache Möglichkeit wäre, das Logging mit `FileWriter` zu implementieren. Hierbei kann man eine einfache `.txt` Datei erstellen und in diese dann hinein schreiben. Der Vorteil hierbei wäre, dass Log Einträge exakt vorgeben kann. Hierbei muss jedoch beachtet werden, dass mögliche I/O Exceptions behandelt werden müssen. Dies kann mit einem einfachen try-catch-Block gelöst werden. Das Schließen der Log-Datei kann dann auch ohne großen Aufwand erreicht werden.

Zusammenfassend lohnt es sich mehr, einen `FileWriter` für das Logging zu verwenden, als die `Logger` Klasse zu verwenden. Somit kann man genau definierte Log Einträge vornehmen und der zusätzliche Aufwand zum Schließen der Log-Datei nach jedem Schreiben hält sich auch in Grenzen und ist ohne großen Aufwand zu realisieren.

Realisationsbeschreibung

Das Logging wird mit der `FileWriter` Klasse realisiert. Hierfür wird zum folgenden Pfad `„./src/log/Log.txt“` eine Log-Datei gesucht oder neu erstellt. Je nach übergebenen Fehlertyp findet ein abgeänderter Log-Eintrag statt. Mit der `write`-Methode wird die übergebene Nachricht und ggf. der Fehlertyp in die Log-Datei geschrieben. Dies geschieht anhängend. Jeder Eintrag geschieht in einer neuen Zeile. Nach jedem Log-Eintrag wird der `FileWriter` geschlossen. Falls währenddessen eine I/O Exception auftreten sollte, was nie der Fall sein sollte, wird dies entsprechend auf dem Error Output der Konsole ausgegeben. Die Log-Datei liegt in einem separaten Log-Verzeichnis, welche nur für den Programmierer von Interesse ist und dem Benutzer verborgen bleibt.

Das Logging findet normalerweise einmal zum Spielstart statt. Dort wird der Initialzustand des Spiels geloggt. Dazu zählen alle Spielerkonfigurationen mit ihren Händen und die Spielsteintasche, bevor von dieser Spielsteine gezogen werden. Nach jedem Spielerzug wird der jeweilige Spielerzug passend zur Aktion geloggt. Dazu wird das neue Spielfeld nach dem Spielerzug geloggt und der Spielzustand. Wie die Log-Einträge exakt aufgebaut sind, wird separat im Dateien Abschnitt des Programmierer-Handbuchs beschrieben (siehe Abschnitt 2.5.).

2.3. Algorithmen

2.3.1. Spielfeld Evaluation

Der Algorithmus für die Spielfeld-Evaluation ist nicht allzu komplex aufgebaut. Dennoch sollte hier darauf eingegangen werden, da die Evaluation für das Spielfeld essenziell ist, u.a. von der KI benötigt wird. Sich mit dem Algorithmus ausführlich auseinanderzusetzen ist Voraussetzung zum Verständnis der KI und wie diese ihre Spielzug-Entscheidungen trifft. Die Spielfeld-Evaluation wird in drei Methoden aufgeteilt. Diese Aufteilung ermöglicht die effiziente Berechnung der Punkte speziell für bestimmte Teile des Spielfeldes. Im Folgenden werden die drei Methoden näher erläutert.

1. **getTilesPerSegment**

Zuerst sollen die Spielsteine für ein angegebenes Segment gezählt werden. Hierzu wird das Segment benötigt und ob die Spalte oder Zeile gezählt werden soll. Ausgehend davon wird in einer Schleife eine Laufvariable hochgezählt bis zur Spielfeldlänge der äußersten Dimension. Das Spielfeld ist ohnehin quadratisch; deshalb muss man sich wegen der Zeilen- bzw. Spaltenlänge keine Sorgen machen. Wenn eine Spalte gezählt werden soll, wird im 2D Game Board Array der erste Index auf das Segment beschränkt und der zweite Index wird dann die Laufvariable der Schleife. Wenn eine Zeile betrachtet werden soll, dann umgekehrt.

Diese Methode zählt alle Vorkommen der Spielsteine für das jeweilige Segment und liefert ein Array zurück, wobei dessen Indices die Spielstein-Typen entsprechen und die Werte deren Vorkommen. Diese Methode ist nur eine Hilfsmethode für die eigentliche Punkteberechnung für das jeweilige Segment.

2. **calculatePoints**

Diese Methode nutzt das zurückgegebene Array der oben genannten getTilesPerSegment Methode und liefert Punkte für das die vorkommenden Spielsteine im Array. Hierbei wird nicht darauf geachtet, ob gleiche Spielsteine in Kombinationen direkt aufeinander folgen. Es müssen nur gleiche Spielsteine vorkommen, damit diese als Kombinationen gelten. Es gibt zwei verschiedene Kombination-Konstellationen, für die Punkte vergeben wird. Die erste Konstellation tritt auf, wenn alle Spielsteine exakt einmal im Array vorkommen. Wenn dies der Fall ist, werden 6 Punkte vergeben. Wenn dies nicht der Fall ist, werden alle Spielsteine gezählt. Folgende Kombinationen werden durch ein switch-Statement unterschieden. Es sind folgende weitere Kombinationen möglich:

- Zwillinge -> jeweils 1 Punkt
- Drillinge -> jeweils 3 Punkte
- Vierling -> 5 Punkte
- Fünfling -> 7 Punkte
- Sechsling -> Integer.MAX_VALUE

Zwillinge und Drillinge können ggf. mehrfach im Segment mit einer Größe von 6 vorkommen. Die Punkte werden dann addiert. Der Grund, warum bei einem Sechsling der maximale Integer-Wert vergeben wird ist der, dass bei einem „Win of Sixes“ die erreichten Punkte des Teams irrelevant werden, denn ein „Win of Sixes“ führt zu einem sofortigen Gewinn des Teams. Außerdem macht es die Erkennung eines „Win of Sixes“ relativ einfach, denn es müssen keine internen Flags verwendet werden.

Diese Methode ist besonders wichtig für die KI, wenn diese ihren Zug evaluieren will. Je nachdem, wo die KI einen Spielstein platzieren würde bzw. von wo die KI einen Spielstein wegzieht, können dort die

neuen Punkte beider Teams berechnet werden, ohne das ganze Spielfeld evaluieren zu müssen. Dies spart viel Rechenleistung, besonders bei potenziell großen Spielfeldern.

3. **getTeamPoints**

Dies ist die letzte Methode, welche die oben genannten Methoden vereint und alle Punkte aus allen Spalten bzw. Zeilen zusammenrechnet. So ergeben sich die Punkte für das jeweilige Team, das angegeben wurde. Diese Methode wird benötigt, um das ganze Spiel zu evaluieren und den Gewinner zu bestimmen.

2.3.2. KI Spielzug Entscheidung

Die KI benutzt einen komplexen Algorithmus zur Spielzugentscheidung. Diese findet den besten Zug nach folgenden absteigend priorisierten Kriterien:

- Wenn möglich, soll ein „Win of Sixes“ erzielt werden
- Wenn notwendig, soll ein „Win of Sixes“ des Gegnerteams verhindert werden
- Es soll die bestmögliche Bewertung erzielt werden

Falls es mehrere mögliche bestmögliche Züge gibt, werden Standard Spielsteine gegenüber Wildcards bevorzugt werden. Verbleiben immer noch mehrere mögliche Züge, wird der Zug mit dem Spielstein gewählt, welcher am meisten auf der Hand vorkommt. Wenn dies nicht genug war, wird der Zug mit dem Spielstein gewählt, welcher am wenigsten auf dem Spielfeld vorhanden ist. Verbleiben immer noch mehr Züge, wird der Zug mit dem Spielstein mit der niedrigsten Ordinalität gewählt. Wenn dies immer noch nicht genug ist, wird der Zug mit der niedrigsten Position ausgewählt und gespielt. Folgend wird der Algorithmus zum besten Spielzug Entscheidung dargestellt.

Zuerst läuft die KI ihre eigene Hand in einer Schleife durch. Für Standard Spielsteine werden alle möglichen Spielzüge in einer Hilfsmethode „**getAllTurns_basic**“ berechnet. Dies geschieht, indem der jeweilige Spielstein auf der Hand hypothetisch auf das aktuelle Spielfeld an jeder möglichen Stelle gelegt wird. Dazu wird das komplette Spielfeld durchlaufen. Nur auf nicht leere Positionen kann der Spielstein gelegt werden. Zu jedem möglichen Zug werden die resultierenden Punkte des jeweiligen Segments berechnet, wo der Spielstein abgelegt wurde. Sowohl die Punkte des eigenen Teams, als auch des Gegnerteams. Das Segment ist hierbei eine Spalte bzw. Zeile des jeweiligen Teams, das betrachtet wird. Zusätzlich wird ein Punktegewinn des eigenen Teams aus der Differenz der alten und der neuen Segmentpunktzahl berechnet. Wie die Punkte berechnet werden, wurde im darüber liegenden Algorithmus erläutert (siehe 2.3.1.).

Für jeden möglichen-Standard Zug werden also die alten und neuen Team-Punkte berechnet, die neuen Punkte des Gegnerteams, und der eigene Punktegewinn. Alles zusammen wird in eine für die KI speziell erstellte **PossibleTurn** Klasseninstanz gespeichert. Jeder mögliche Zug ist eine PossibleTurn-Instanz. Das Ergebnis der Hilfsmethode ist eine Liste von **PossibleTurns**.

Ein **PossibleTurn** ist ein wie der Name schon verrät möglicher Zug der KI. Dessen Konstruktor nimmt alle möglichen Parameter entgegen, um so jede mögliche Zug-Konstellation abbilden zu können. Sowohl Standard Spielzüge als auch alle vier verschiedenen Wildcard Züge können mit einer PossibleTurn Instanz dargestellt werden.

Wenn eine Wildcard in der KI Hand betrachtet wird, findet ebenfalls eine Hilfsmethode „**getTurns_wildcard**“ die meisten möglichen Züge für die jeweilige Wildcard. In dieser Methode werden erstmal alle möglichen Wildcard Züge berechnet und in einer Liste der möglichen Züge gespeichert, bevor es weiter geht. Für jeden Wildcard-Typ findet eine andere Berechnung aller

möglichen Züge statt, denn die Wildcards funktionieren alle unterschiedlich. Dies wird ebenfalls in separaten Hilfsmethoden ausgelagert. Folgend werden diese Hilfsmethoden erläutert:

1. allWildcardTurns_REMOVER

Diese Methode findet für die REMOVER Wildcard alle möglichen Züge. Dafür wird das komplette Spielfeld durchlaufen. An jeder Position, die ein nicht-leeres Spielfeld enthält, darf der REMOVER gespielt werden. Dazu werden wie beim jedem möglichen Zug alle resultierenden Punkte berechnet und dazu ein neuer PossibleTurn erstellt und der Ergebnisliste hinzugefügt.

2. allWildcardTurns_MOVER

Diese Methode findet für die MOVER Wildcard alle möglichen Züge. Dafür wird das komplette Spielfeld durchlaufen. Jedes Feld, das ein nicht-leeres Feld enthält, ist ein verschiebbarer Spielstein. Dieser wird Spielstein und dessen Position wird gespeichert und das Spielfeld wird erneut durchgelaufen. An jeder nun leeren Position darf der gespeicherte Spielstein verschoben werden. Dazu wird eine zweite Position gespeichert. Wieder werden alle resultierenden Punkte berechnet und ein neuer PossibleTurn erstellt und der Ergebnisliste hinzugefügt.

3. allWildcardTurns_SWAPONBOARD

Diese Methode findet für die SWAPONBOARD Wildcard alle möglichen Züge. Diese Methode funktioniert ähnlich zur MOVER Methode, doch diesmal darf der zu verschiebende Spielstein nur auf nicht-leere Positionen, also andere Spielsteine auf dem Spielfeld gelegt werden, um mit diesen zu tauschen. Natürlich darf nicht mit der gleichen Position getauscht werden. Folgend werden beide Tauschpositionen und beide Spielsteine gespeichert. Für die neue Position des ersten Spielsteines werden die relevanten Punkte berechnet und wieder jeweils ein neuer PossibleTurn erstellt. Dieser wird der Ergebnisliste hinzugefügt.

4. allWildcardTurns_SWAPWITHHAND

Diese Methode findet für die SWAPWITHHAND Wildcard alle möglichen Züge. Diesmal für jedes nicht-leere Feld geprüft, ob ein Standard Spielstein von der KI Hand mit diesen tauschen kann. Dafür muss also das Spielfeld komplett durchlaufen werden, aber auch die KI Hand. Für jeden möglichen Zug werden wieder alle relevanten Punkte berechnet und ein neuer PossibleTurn für die Ergebnisliste erstellt.

Sobald alle möglichen Wildcard Züge als Liste berechnet worden sind, kann die „getTurns_wildcard“ Hilfsmethode diese reduzieren. Folgende Reduktion ist nur oberflächlich und soll die wichtigsten Wildcard-Züge beibehalten, bevor die eigentliche Zug-Reduzierung stattfindet. Hierfür werden alle möglichen Wildcard-Spielzüge in einer Schleife durchlaufen. Für jeden Zug wird überprüft, ob dieser ein „Win of Sixes“ erzielen kann. Hierfür werden die neuen Team-Punkte einfach betrachtet. Kommt ein Wert raus, welcher den Wert der maximalen Integer entspricht, wurde ein „Win of Sixes“ erreicht. Dieser Zug wird dann als sogenannter **SixesTurn** gespeichert. Wenn ein SixesTurn existiert, dann wird nur dieser Zug der Hauptmethode zurückgeliefert.

Falls der Zug keinen „Win of Sixes“ erbrachte, muss überprüft werden, ob ein gegnerischer „Win of Sixes“ verhindert wurde. Hierfür wird an der relevanten Position überprüft, ob die gegnerische Punktzahl 7 betrug. Dies wäre die Punktzahl für einen Fünfling. Es bestünde dann eine Gefahr für einen gegnerischen „Win of Sixes“. Dann werden die gegnerischen Punkte nach dem potenziellen Zug betrachtet. Wenn sich die Punkte nicht verändert haben oder sogar gesunken sind, wurde ein gegnerischer „Win of Sixes“ verhindert. Bevor der Zug als SixesTurn gesetzt wird, muss aber noch überprüft werden, ob dieser Turn nicht zu einem gegnerischen „Win of Sixes“ beigetragen haben könnte. Es könnte ja sein, dass durch einer SWAPONBOARD Wildcard ein CROSS „Win of Sixes“ unverändert bleibt, weil ein CROSS mit einem CROSS getauscht wurde. Demnach darf der platzierte Spielstein in dem Risiko Segment nicht vorkommen, sonst wäre dieser Teil der Kombination, die

verhindert werden muss. Wenn alles in Ordnung ist, dann wird der jeweilige Zug als ein SixesTurn gespeichert, wenn dieser noch null ist.

Ein SixesTurn kann sowohl ein Zug sein, mit dem ein „Win of Sixes“ erreicht, als auch ein Zug, der einen gegnerischen „Win of Sixes“ verhindert. Wenn beim kompletten Zugdurchlauf kein SixesTurn gesetzt wurde, wird die komplette Liste an PossibleTurns für die jeweilige Wildcard zurückgeliefert.

Nun sind wir wieder in der Hauptmethode „**evaluateToBestTurn**“. Diese hat nun für jeden Standard Spielstein und Wildcard der KI Hand einer Liste von PossibleTurns. Alle Ergebnisse kommen in einer PossibleTurn ArrayList zusammen. Diese Liste gilt es nun zu einem Zug zu reduzieren.

Folglich werden alle möglichen Züge in einer Schleife durchlaufen. Zuerst wird wieder geprüft, ob ein „Win of Sixes“ erzielt werden kann. Dies findet auf gleiche Art und Weise statt, wie bei der initialen Reduktion der Wildcard Züge. Falls ein „Win of Sixes“ erzielt wurde, wird wieder der entsprechende SixesTurn gesetzt. Wenn nicht, wird geprüft, ob ein gegnerischer „Win of Sixes“ verhindert werden konnte. Hierbei gibt es aber 2 Fallunterscheidungen. Wurde eine MOVER Wildcard gespielt, müssen die Gegnerpunkte an der Stelle betrachtet werden, von wo der Spielstein ursprünglich weggezogen worden war, nicht wohin der gezogen wurde. Dann findet wieder die gleiche Prüfung wie sonst statt. Die Gefahr muss erkannt werden. Die Gegnerpunkte müssen gleich geblieben sein oder sich verschlechtert haben. Natürlich darf nicht ein Spielstein platziert werden, aus der die gegnerische Kombination bereits besteht. Dies würde zu nichts beitragen. Sonst, wenn keine MOVER Wildcard gespielt wurde, wird die Position betrachtet, an der ein Spielstein platziert wurde. Hier wird potenziell der SixesTurn gesetzt.

Falls der SixesTurn noch null ist, versucht die KI nun die bestmögliche Punktzahl zu erzielen. Hierfür wird das Maximum der Punktegewinne der Züge berechnet. Dies geschieht einfach über Streams. Dann werden alle Züge in einer neuen Liste hinzugefügt, die diesen maximalen Punktegewinn erzielten.

Danach findet die eigentliche Reduktion der Züge statt. Hierfür wird die Methode „**reduceToFinalTurns**“ benutzt. Diese wendet die oben genannten Regeln an, welche gelten, wenn mehrere bestmögliche Züge vorhanden sind. Diese Methode wird folglich erklärt:

Vor jedem Reduktionsschritt wird erstmal geprüft, ob die Liste mehr als einen Turn enthält. Wenn nicht, dann muss keine weitere Reduktion stattfinden und die Methode ist fertig. Der erste Reduktionsschritt bevorzugt Standard Spielsteine gegenüber Wildcards. Hierfür werden alle Standard Züge in eine Liste aufgelistet und alle Wildcard Züge in eine Liste. Wenn die Liste der Standard Züge leer ist, dann werden nur die Wildcard Züge genommen, sonst werden nur die Standard-Züge genommen.

Für die nächste Reduktion wird geschaut, welcher Spielstein für die Züge am meisten auf der Hand vorhanden ist. Hierfür werden alle gespielten Spielsteine in einem Set gespeichert. Ein Set hat den Vorteil, dass Spielsteine nur einmalig gespeichert werden. Für diese relevanten Spielsteine berechnet eine Hilfsmethode, welche von denen am meisten in der KI Hand vorkommen. Es können mehrere Spielsteine am häufigsten vorkommen. Dies wird in Form eines Sets zurückgeliefert. Nun wird über alle restlichen möglichen Züge iteriert. Wenn dessen gespielter Spielstein im Set vorkommt, werden sie in die neue Liste aufgenommen. Es bildet sich somit eine neue, reduzierte Liste von möglichen Zügen.

Für die nächste Reduktion wird geschaut, welcher Spielstein der Züge am wenigsten auf dem Spielfeld vorkommt. Dafür wird wieder ein Set von relevanten Spielsteinen gebildet, welche die Züge haben. Zu diesem Set berechnet eine Hilfsmethode die Spielsteine, die am seltensten auf dem Spielfeld vorkommen. Dafür wird wieder ein Set zurückgeliefert. Es wird über die neue Liste verbleibender Züge

iteriert. Jeder Zug, dessen Spielstein am wenigsten auf dem Spielfeld vorkommt, wird in eine neue reduzierte Liste aufgenommen.

Im nächsten Reduktionsschritt werden nur die Züge mit den Spielsteinen weiterkommen, die die niedrigste Ordinalität haben. Hierbei kann es nur einen niedrigsten Spielstein geben. Hierfür wird wieder ein Set relevanter Spielsteine gebildet. Dazu wird der Spielstein mit der niedrigsten Ordinalität berechnet. Jeder übriggebliebene Zug, der diesen Spielstein benutzt, kommt weiter.

Im letzten Reduktionsschritt wird die niedrigste Position berechnet. Nur die Züge, welche diese niedrigste Position beinhalten, werden finale Züge genannt. Idealerweise liefert diese Hilfsmethode eine Liste mit nur einem optimalen Zug. Nun können wir zur Hauptmethode zurückgehen.

Indessen wurden alle Züge zu idealerweise einem finalen Zug reduziert. Falls die Reduktion dennoch mehr Züge durchließ, wird als letzter Schritt einfach nur das erste Element der Liste genommen. Wenn es bereits einen SixesTurn gab, dann fand ohnehin keine Reduktion statt. Auf jeden Fall haben wir jetzt einen finalen Zug.

Hier gilt es zu unterscheiden, ob der finale Zug ein 2-Phase-Wildcard Zug ist oder nicht. Wenn der finale Zug ein 1-Phase Zug ist, also ein Standard-Zug oder ein REMOVER Zug, dann wird der aufrufenden Game-Instanz einfach der resultierende Zug zurückgeliefert. Wenn aber ein 2-Phase Wildcard Zug rauskam, muss die 2. Phase zwischen gespeichert werden. Diese wird beim erneuten Aufruf der Zug Entscheidung dann zurückgeliefert, um den 2-Phase-Wildcard Zug zu vollenden. Die 2. Phase wird also zwischengespeichert. Dann wird die 1. Phase des Wildcard-Zuges gebildet und zurückgeliefert.

2.3.3. KI Zug Animation

Die KI Zug Animation ist von der Komplexität nicht sehr anspruchsvoll. Dennoch ist diese **createAndPlayAnimation** Methode von großer Bedeutung, da diese für jede mögliche Animation aller Spielerzüge zum Einsatz kommt. Sowohl Spielsteine von Spielerhänden auf das Spielfeld oder umgekehrt, als auch Spielsteine vom Spielfeld auf das Spielfeld können so animiert werden. Diese Methode wird im Folgenden näher erklärt.

Basis für die Animation ist ein unsichtbares Animation **Pane**, welches unmittelbar auf dem StackPane des Haupt-Containers der Benutzeroberfläche, jedoch hinter dem „New Game“ Menü liegt, wenn dies angezeigt wird. Dieses Pane ist unsichtbar und der Spieler kann durchklicken. Dieses Pane ist wichtig, damit animierte ImageView Instanzen dort platziert und bewegt werden können. Die Animation findet stets vor dem Spielfeld, aber hinter dem „New Game“ Menü statt, wenn dieses angezeigt wird.

Für die Animation muss zunächst eine **PathTransition** erstellt werden. Um eine PathTransition zu erstellen braucht ein ImageView mit einem Image von dem Spielstein, welches animiert werden soll. Dazu braucht man eine Position, wo die Transition startet und eine Position, wo die Animation endet. Die Start-Position kann eine einerseits auf dem Spielfeld liegen, andererseits aber auch auf einer Hand Slot Position des Spielers.

Die Methode nimmt für den animierte Spielstein deshalb einen Spielstein entgegen, für die Position den aktuellen Spieler mit einem Hand-Slot. Wenn die Animation aber vom Spielfeld beginnt, wird eine Spielfeld-Position gebraucht. Das Ziel der Transition kann auch eine Position auf dem Spielfeld sein, aber auch eine Handposition. Man kann aber den Fakt ausnutzen, dass kein Spielstein von der Hand auf die gleiche Hand gespielt werden kann. Deshalb kann die Handposition sowohl für Startposition als auch für die Zielposition stehen. Außerdem wird ausnahmsweise die Game-Instanz benötigt, um den nächsten Spielerzug auszulösen. Falls eine 2-Phase-Wildcard gespielt wurde, ist der Spielerzug noch nicht vorbei. Deshalb wird ein Flag benötigt, um den nächsten Spielerzug ggf. nicht auszulösen.

Nun müssen die Start- und Ziel-Positionen für die **PathTransition** berechnet werden. Dafür wird einfach die Start-Position genommen, wenn die Animation auf dem Spielfeld startet oder die Hand-Position, wenn die Animation auf der Spielerhand startet. Das Gleiche kann für die Ziel-Position getan werden.

Für die jeweiligen Positionen werden dann separat die LocalBounds (Breite & Höhe) und die Position in der Scene berechnet. Um die tatsächliche Position nun zu berechnen, muss man die Position des Nodes in der Scene nehmen und dazu seine LocalBounds addieren. Die LocalBounds müssen aber zuvor halbiert werden, da die PathTransition ein Node immer ausgehend von seiner Mitte aus auf eine Position platziert, nicht von seinem Ausgangspunkt oben links aus. Zuletzt muss vom Ergebnis einen gewissen Offset abziehen, der aus der Differenz der Stage Höhe und der Scene Höhe entsteht. Jetzt hat man sowohl die Start-, als auch die End-Position berechnet.

Nun kann ein ImageView erstellt werden, dessen Breite und Höhe die des Ziel-Nodes entspricht. Entsprechend wird das Image des übergebenen Spielsteins gesetzt. Die initiale Position des neu erstellten ImageViews muss jedoch außerhalb des sichtbaren Bereichs für den Spieler erfolgen. Auch wenn das ImageView im Zuge der Animation sofort auf die Start-Position verschoben wird, gibt es jedoch einen Augenblick, bei der das ImageView an der Initialposition zu sehen ist. Dies kann vermieden werden, indem das ImageView an einer hohen negativen Position initialisiert wird.

Das erstellte ImageView wird nun dem Animation Pane hinzugefügt und die PathTransition kann erstellt werden. Für die PathTransition wird das ImageView zunächst initial per **MoveTo** auf die Start-Position verschoben. Dann findet eine Transition per **LineTo** zur Ziel-Position statt. Die Dauer der PathTransition wird durch eine globale duration Variable festgelegt und überprüft diese bei jeder neuen Animation, falls sich die duration geändert hat. Sobald die PathTransition fertig ist, wird das animierte ImageView vom Animation Pane wieder entfernt. Falls der nächste Zug ausgelöst werden soll, darf dies erst nach einer **PauseTransition** passieren, welche genauso lange dauert wie die Animation selbst.

The diagram illustrates the architecture of a game application, organized into two main layers: **gui** (orange background) and **logic** (green background).

GUI Layer (gui):

- ApplicationMain** starts **JarMain** and creates **UserInterfaceController**.
- UserInterfaceController** is associated with **static Utilities** and **static DragHandlers**. It creates **JavaFXGUI** and **PopupController**.
- JavaFXGUI** creates **PopupController**.
- PopupController** is associated with **Enum GameStates** and **Game**.

Logic Layer (logic):

- Game** is the central class, containing **TileBag** (Spielstein Tasche) and **GameBoard** (Spielfeld Referenz). It creates **GameData** and **PlayerData**.
- GameData** contains **Playerdaten**.
- GameBoard** contains **Game** and **AI_Player**.
- AI_Player** creates **Record PossibleTurn** and **Player**.
- Record PossibleTurn** is associated with **Game** and **Player**.
- Player** is associated with **Game** and **Record PossibleTurn**.
- Game** is associated with **Interface GUIConnector**.
- Interface GUIConnector** is associated with **Game** and **Record PossibleTurn**.

Relationships and Data Flow:

- Game** contains **TileBag** and **GameBoard**.
- GameBoard** contains **Game** and **AI_Player**.
- AI_Player** contains **Record PossibleTurn** and **Player**.
- Record PossibleTurn** contains **Game** and **Player**.
- Player** contains **Game** and **Record PossibleTurn**.
- Game** contains **GameData** and **PlayerData**.
- GameData** contains **Playerdaten**.
- Game** is associated with **Interface GUIConnector**.
- Interface GUIConnector** is associated with **Game** and **Record PossibleTurn**.

Static Elements:

- static Utilities** and **static DragHandlers** are associated with **UserInterfaceController**.
- static GameFileManager** and **static Utilities** are associated with **Game**.
- Record Position**, **Enum ErrorType**, and **Enum GameTiles** are associated with **Game**.

2.4.1. Erläuterungen des Programmorganisationsplans

Der Programmorganisationsplan ist in 2 Bereiche bzw. Pakete aufgeteilt. Einerseits in das orange gui-Paket und andererseits das grüne logic-Paket. Diese Pakete spiegeln 1:1 die Hierarchien der Quelldateien wider und dienen zur Aufteilung der Logik des Spiels und der grafischen Benutzeroberfläche. Demnach kann die Logik auch ohne das gui-Package selbstständig funktionieren, aber nicht die GUI ohne die Logik.

Hierbei werden Klassen mit rechteckigen Kästen dargestellt. Wenn diese Besonderheiten aufweisen – seien sie Enumerationen, statisch oder Records, wird dies zusätzlich textuell über dem Bezeichner genannt.

Es werden hauptsächlich 2 Arten von Beziehungen dargestellt. Einerseits die **Aggregation** oder auch **Assoziation**, welche am Pfeil mit der Raute näher beschrieben werden. Hier wird bewusst wage die Aussage gemacht, dass solche verbundene Klassen grob im Zusammenhang stehen. Deshalb wird sowohl für die Aggregation als auch für die Assoziation derselbe Pfeil verwendet. Wenn eine solche Beziehung einer Aggregation gleichen soll, wird dies an der Beschriftung des Pfeils angedeutet. Auf der anderen Seite stehen die Erzeugt-Beziehungen, welche ausgegraut die Beziehungen vom Erzeuger und Erzeugnis veranschaulichen soll. Diese Beziehung soll den Ablauf der Erzeugung von Klassen, nicht jedoch den logischen Zusammenhang der Klassen verdeutlichen.

Zuletzt gibt es 2 spezielle Beziehungen. Zuerst die **Vererbungsbeziehung**, welche von der Klasse Player ausgeht, welche seine Member an die AI_Player Klasse vererbt. Dies wird durch den blauen Pfeil dargestellt. Und zuletzt die **Implementiert-Beziehung**, welche von der JavaFXGUI Klasse ausgeht in Richtung des GUIConnector Interfaces. Dies wird mit dem gestrichelten grünen Pfeil dargestellt. Dieser Programmorganisationsplan soll somit einen guten Überblick über alle Klassen geben und wie diese miteinander zusammenhängen.

2.4.2. Erläuterungen zum Paket „gui“

Im Programmorganisationsplan unter dem Bereich „gui“ nehmen 2 Klassen eine Sonderrolle ein und wurden deshalb am Rande dargestellt. Die erste davon ist die statische **Utilities** Klasse, welche im gesamten gui-Package verwendet werden und nicht speziell eine Klasse zugeordnet werden kann. Diese enthält nützliche Hilfsmethoden, die keiner Klasse komplett eigen sind, sondern klassenübergreifend verwendet werden können. Eine ähnliche Rolle spielt die statische **DragHandlers** Klasse, welche ebenfalls im gesamten gui-Package verwendet werden kann. Diese beinhaltet alle notwendigen Drag & Drop Handlers und bündelt diese. Beide Sonderklassen werden hauptsächlich von der UserInterfaceController- und JavaFXGUI-Klasse verwendet.

2.4.3. Erläuterungen zum Paket „logic“

Unter dem Bereich „logic“ nehmen 5 Klassen eine Sonderrolle ein. Wie im gui-Package besitzt das logic-Package ebenfalls eine statische **Utilities** Klasse, dessen Methoden im gesamten gui-Package verwendet werden können. Eine ähnliche Rolle spielt der statische **GameFileManager**, welcher Methoden zum Laden und Speichern von Spielständen bereitstellt. Diese wird hingegen sowohl in logic-Tests, als auch im gui-Package verwendet, ist demnach also öffentlich zugänglich.

Speziell die Datenhaltungsklasse **Position** wird gesondert betrachtet, weil jede Klasse im logic-Package eine Position erstellen und verarbeiten können soll. Demnach würde der Programmorganisationsplan zu unübersichtlich werden. Das Gleiche gilt für die Enumeration **GameTiles**, welche von nahezu jeder Klasse im logic-Package verwendet werden kann und für die logisch-interne Betrachtung der Spielsteine zuständig ist, damit die Spielsteine nicht als Ganzzahlen betrachtet werden. Auf gleiche Weise wird die Enumeration **ErrorType** betrachtet, welche für die logisch-interne einheitliche Betrachtung von Fehlertypen zuständig ist. All diese genannten gesonderten Klassen und

Enumerationen gesondert zu betrachten ermöglicht es, dass der Programmorganisationsplan übersichtlicher gestaltet werden konnte und sorgt für bessere Lesbarkeit.

2.5. Dateien

In diesem Abschnitt werden alle im Programm verwendeten Dateien aufgeführt und wie diese aufgebaut werden. Insgesamt gibt es zwei Typen von Dateien, welche im Programm verwendet werden: Die Spielstand Datei und die Log-Datei. Folgend wird deren Aufbau dargelegt.

2.5.1. Spielstand Datei

Die Spielstand-Datei besteht aus einem Game Objekt-Eintrag. Dieser enthält unter anderen jeweils 4 PlayerData Objekt-Einträge. Jedes **PlayerData** Objekt ist folgendermaßen aufgebaut:

- Ein String Wert für den **Spielernamen**.
- Jeweils ein boolescher Wert, ob der Spieler **aktiv** ist und ob der Spieler eine **KI** ist.
- Ein int-Array mit **Spielsteinen** mit ihren Ordinalitäten als Werte.

Die weiteren **GameData** Einträge sehen wie folgt aus:

- Ein Integer-Wert für den aktuellen Spieler.
- Ein 2D int-Array mit Spielsteinen in ihren Ordinalitäten.
- Ein int-Array mit der Anzahl der genutzten Wildcards. Die Indices des Arrays stehen in aufsteigender Reihenfolge für folgende Wildcards: REMOVE, MOVER, SWAPONBOARD, SWAPWITHHAND.
- Ein int-Array für die Spielsteintasche mit Spielsteinen mit ihren Ordinalitäten als Werte.

2.5.2. Log Datei

Die Log-Datei besteht aus mehreren Komponenten. Bei jedem Spielanfang wird ein initialer Log-Eintrag geschrieben. Diese beinhaltet folgende Elemente:

- Spielsteintasche als Liste von Enum-Werten.
- **4 Spielereinträge** jeweils in einer Zeile. Diese beinhalten die **logische Spielerbezeichnung**, den **Spielernehmen**, ob dieser **Mensch oder KI** ist und zuletzt seine **Hand** als Liste von Spielsteinen mit ihren Ordinalitäten als Werte.

Während des Spielverlaufs werden pro Zug folgende Log-Einträge geschrieben: Der Spielzug des aktuellen Spielers, das neue Spielfeld und der neue Spielzustand. Es gibt fünf verschiedene Zug Konstellationen. Entweder vollzieht der Spieler einen Standardzug oder er spielt vier verschiedene Wildcards. Für jede Konstellation gibt es einen separaten Log-Eintrag. Folgend werden alle fünf Konstellationen beschrieben:

- Standard Spielzug: Spielernamen, Spielstein, Position, Handslot, neue Hand
 - z.B. „**PlayerName2** | [**1**] placed on (**5**|**1**) | handSlot [**3**] | new Hand : [**10, 3, 7, 2**]“
- REMOVE Wildcard: Spielernamen, Spielstein, Position, Handslot, neue Hand
 - z.B. „**PlayerName2** | [**1**] placed on (**5**|**1**) | handSlot [**3**] | new Hand : [**10, 3, 7, 2**]“
- MOVER Wildcard: Spielernamen, Spielstein, Position, Position, neue Hand
 - z.B. „**PlayerName2** | [**4**] moved from (**1**|**0**) to (**2**|**1**) | new Hand : [**8, 2, 3, 10**]“
- SWAPONBOARD Wildcard: Spielernamen, Spielstein, Position, Position, neue Hand
 - z.B. „**PlayerName1** | [**4**] swapped (**2**|**1**) with (**3**|**2**) | new Hand : [**5, 4, 5, 1**]“
- SWAPWITHHAND Wildcard: Spielernamen, Spielstein, Handslot, Position, neue Hand
 - z.B. „**PlayerName2** | [**2**] swapped from hand at [**1**] to (**3**|**2**) | new Hand : [**8, 4, 3, 9**]“

Nach dem Spielerzug wird das Spielfeld als 2D int-Array mit Spielstein Ordinalitäten als Werte ausgegeben. Danach folgt der Evaluationszustand des Spiels. Es gibt vier verschiedene Zustände:

- laufendes Spiel: ONGOING_GAME
- Gleichstand: DRAW
- vertikales Team hat gewonnen: TEAM_VERTICAL
- horizontales Team hat gewonnen: TEAM_HORIZONTAL

Am Ende des Spiels wird wieder das initiale Log mit den aktualisierten Werten ausgegeben. Dazu das Spielfeld und der letzte Spielzustand.

2.6. Programmtests

Im folgenden Abschnitt werden grafische Programmtests systematisch genannt und dessen Durchführung erklärt. Da die Logik bereits im Projekt ausgiebig getestet wurde, soll hier nur das Testen der grafischen Benutzeroberfläche stattfinden. Es werden hier keine willkürlichen Tests dargestellt, sondern nur diese, die Besonderheiten aufweisen, für den generellen Spielverlauf relevant sind oder von interessanter Natur sind.

2.6.1. Menü Tests

Testbeschreibung	Erwartetes Ergebnis
Der Spieler sollte nicht in der Lage sein, beim Start der Anwendung ein Spiel zu speichern, welches noch gar nicht gestartet wurde.	Unter dem Menü „Game“ muss das „Save“ Menuelement deaktiviert sein
Der Spieler sollte in der Lage sein, die gleiche Animationsdauer nochmals auszuwählen, ohne dass sich was ändert.	Wenn unter dem Menü „AI Options“ und unter „AI Turn Duration“ für die Animationsdauer „Short“ mehrfach angeklickt werden, soll der gesetzte Haken dort bleiben. Das Gleiche gilt für die anderen 2 Optionen.
Das Spielfeld soll beim Einschalten der optionalen Punkteleisten weniger Platz einnehmen.	Beim Einschalten der optionalen Punkteleisten im Menü „View“ schrumpft das Spielfeld und berührt recht und unten nicht mehr den schwarzen Rahmen.
Die optionalen Punkteleisten sollen beim Ausschalten keinen unnötigen Platz mehr einnehmen.	Beim Ausschalten der optionalen Punkteleisten im Menü „View“ soll das Spielfeld den gesamten verfügbaren Platz einnehmen.
Beim Ein- & Ausschalten der optionalen Punktetabelle soll visuell sichtbar sein.	Beim Ein- & Ausschalten der optionalen Punktetabelle im Menü „View“ soll diese im Seiten-Container rechts unten erscheinen oder verschwinden
Das Ein- & Ausschalten der optionalen Team-Punkte soll visuell sichtbar sein.	Beim Ausschalten der optionalen Team-Punkte soll im Team-Information-Grid der Team Name die komplette Zeilenbreite einnehmen. Beim Einschalten nimmt der Team Name nur die Hälfte der Zeilenbreite in Anspruch und die Punkte werden rechts in der 2. Spalte angezeigt.
Wenn bei der KI dessen Animation abgebrochen bzw. übersprungen wird, muss dies visuell sichtbar sein	Beim Überspringen der KI-Animation muss das animierte ImageView sofort verschwinden und keine Bewegung darf während der Animationsdauer stattfinden.

Wenn die KI Hand angezeigt werden soll, muss dies beim KI Zug sichtbar sein.	Wenn die KI Hand angezeigt werden soll, muss die KI Hand sichtbar sein und nicht mehr ImageViews mit leeren Spielsteinen anzeigen
Die KI Animationsdauer muss mit den unterschiedlichen Animationsdauern visuell unterscheidbar sein.	Keine der 3 Animationsdauern darf gleich lang dauern. Die Animationsdauer muss sich von langsam zu mittel zu lang jeweils verdoppeln. Dies muss mit dem bloßen Auge beurteilt werden.

2.6.2. „New Game“ Menü Tests

Testbeschreibung	Erwartetes Ergebnis
Bei 4 Spielern soll mit allen Spieler-Formularen interagiert werden können.	Wenn das „4 PLAYERS“ CheckMenuItem ausgewählt ist, müssen alle Spieler-Formulare im aktiven Zustand sein und interaktiv sein.
Bei 2 Spielern soll mit nur 2 Spieler-Formularen interagiert werden können.	Wenn das „2 PLAYERS“ CheckMenuItem ausgewählt ist, müssen die jeweils zweiten Spieler-Formulare für das jeweilige Team im deaktivierten Zustand sein. Mit diesen darf nicht interagiert werden können.
Die CheckMenuItems sollen jeweils mehrfach angeklickt werden können, ohne dass sie abgewählt werden.	Beim doppelten Anklicken der „2 PLAYERS“ und „4 PLAYERS“ CheckMenuItems müssen diese jeweils ausgewählt bleiben.
Bei invaliden Spielernamen muss eine Fehlermeldung angezeigt werden.	Über dem „START GAME“ Button muss der Text „Name(s) invalid!“ erscheinen.
Bei doppelten Spielernamen muss eine Fehlermeldung angezeigt werden.	Über dem „START GAME“ Button muss der Text „No duplicate Names allowed!“ erscheinen.

2.6.3. Spielverlauf Tests

Testbeschreibung	Erwartetes Ergebnis
Bei einem Standard Spielzug muss zu erkennen sein, wo ein gezogener Spielstein hingelegt werden kann.	Wenn ein Standard Spielstein über ein leeres Feld auf dem Spielfeld gezogen wird, soll das jeweilige Feld grün umrandet aufleuchten.
Bei einem beliebigen Spielzug darf kein Spielstein auf eine andere Hand abgelegt werden.	Wenn ein beliebiger Spielstein über jede Spielerhand, einschließlich der eigenen gezogen wird, darf kein ImageView aufleuchten. Dies muss für jeden Spielsteintyp getestet werden, inklusive Wildcards.
Wenn eine 2-Phase Wildcard gespielt wird, soll diese überall auf dem Feld abgelegt werden können.	Wenn eine MOVER-, SWAPONBOARD- oder SWAPWITHHAND Wildcard auf irgendein Feld auf dem Spielfeld gezogen wird, muss das komplette Spielfeld grün umrandet aufleuchten
Wenn eine 2-Phase Wildcard aktiviert wurde, muss dem Spieler erklärt werden, wie er diese Wildcard zu nutzen hat.	Wenn eine MOVER-, SWAPONBOARD- oder SWAPWITHHAND Wildcard aktiviert wurde, muss das Label über dem Spielfeld einen Instruktionstext anzeigen.
Kann keine Wildcard gespielt werden, muss dies dem Spieler angezeigt werden.	Wenn eine REMOVER-, MOVER-, SWAPONBOARD- oder SWAPWITHHAND Wildcard aktiviert wird, aber das Spielfeld ist komplett leer, dann muss der Spieler die Wildcard in der Hand behalten und das Label

	über dem Spielfeld muss dem Spieler eine Warnung anzeigen.
Wird eine REMOVER Wildcard gespielt, darf nur auf nicht-leere Felder gespielt werden.	Nur nicht-leere Felder grün umrandet auf, wenn auf diese der REMOVER gezogen wird
Wird eine MOVER Wildcard gespielt, dürfen nur noch Spielsteine auf dem Spielfeld verschoben werden.	Nichts mehr von der eigenen Hand kann gezogen werden. Wenn ein Spielstein vom Spielfeld gezogen wird, leuchten leere Felder grün umrandet auf, wenn auf diese gezogen wird.
Wird eine SWAPONBOARD Wildcard gespielt, dürfen nur noch nicht leere Spielsteine auf nicht leere Spielsteine auf dem Spielfeld gelegt werden.	Nur noch nicht-leere Spielsteine auf dem Spielfeld können gezogen werden. Nur nicht-leere Spielsteine leuchten grün umrandet auf, wenn auf diese ein Spielstein gezogen wird.
Wird eine SWAPWITHHAND Wildcard gespielt, dürfen nur Standard Spielsteine von der eigenen Hand auf nicht leere Spielfelder gezogen werden.	Wenn Standard Spielsteine auf das Spielfeld gezogen werden, leuchten nur nicht-leere Felder grün umrandet auf, bei Wildcards gibt es keinen Effekt.

2.6.4. „Game Over“ Menü Tests

Testbeschreibung	Erwartetes Ergebnis
Wenn ein Team gewonnen hat, soll dies im Menü durch einen passend gewählten Hintergrund sichtbar gemacht werden.	Gewinnt das vertikale Team, muss das Menü einen grünen, vertikal gestreiften Hintergrund haben. Gewinnt das horizontale Team, muss das Menü einen orangen, horizontal gestreiften Hintergrund haben.
Gewinnt kein Team, muss der Hintergrund des Menüs entsprechend neutral aussehen.	Gewinnt kein Team, wird ein grau-weißer diagonal gestreifter Hintergrund angezeigt.