

## **Programmierpraktikum Pentago**

Autor: Solodownik, Dimitri

Fachrichtung: Informatik

Matrikelnummer: inf102736

Fachsemester & Verwaltungssemester: 5

# Programmierhandbuch

## Inhaltsverzeichnis

1 Entwicklungskonfigurationen	3
2 Problemanalyse und Realisation	3
2.1 GUI-Darstellung des Spielfelds	3
2.2 GUI-Rotation eines Quadranten	4
2.3 Spieltheorie Alpha-Beta-Suche	5
2.4 Rotation eines Quadranten im Baum	7
2.5 Bewertung einer Spielsituation	9
2.6 Gewinn eines Spielers erkennen	10
2.7 Gewährleistung des korrekten Ablaufs	12
2.8 Spielstand speichern / laden	13
2.9 Protokoll - Spielzüge zurücknehmen / wiederholen	14
3 Programmorganisationsplan	15
4 Beschreibung grundlegender Klassen	16
5 Programmtests	20

## 1 Entwicklungskonfiguration

Genutzte Softwarekomponenten zur Entwicklung des Spiels:

Betriebssystem	Windows 10 Home 64-Bit
Entwicklungsumgebung	NetBeans IDE 8.2 mit Java SE Development Kit (JDK) Version 8

## 2 Problemanalyse und Realisation

*Die folgende Analyse bezieht sich auf das Spiel Pentago, welches als JavaFX-Anwendung umgesetzt werden soll. In den Unterpunkten dieses Kapitels werden die wesentlichen Problemstellungen der Aufgabe analysiert und deren Lösung diskutiert.*

### 2.1 GUI-Darstellung des Spielfelds

#### Problemanalyse

Pentago besteht aus einem Spielfeld mit 36 Positionen, die aus einer 6x6 Tabelle gebildet werden.

Ein Spieler kann in jeder freien Position des Spielfelds seinen Spielstein platzieren. Wurde ein Spielstein platziert, gilt diese Zelle für den weiteren Verlauf des Spiels als besetzt und kann nicht von dem Gegner genutzt werden. Des Weiteren ist das Spielfeld in 4 rechteckige Quadranten aufgeteilt, die nach jeder Spielsteinplatzierung um 90 Grad nach links oder nach rechts gedreht werden und dadurch die Spielfeldbelegung bei jeder Drehung verändern. Ist ein Spieler am Zug, so kennzeichnet er seine Zelle, indem er sie anklickt. Jeder Spieler erhält Spielsteine in seiner Farbe. Bei einer Größenveränderung des Fensters soll sich das Spielfeld automatisch anpassen.

#### Realisationsanalyse

Auf den ersten Blick eignet sich für das Spielfeld ein GridPane hervorragend. Die Zuordnung der Zeilen- und Spaltenaufteilung kann optimal für das Zuordnen von Zellen zu Spielfeldkoordinaten genutzt werden. Durch die Berücksichtigung der Drehungen der einzelnen Quadranten stellt man fest, dass dieses GridPane diese Aufgabe nicht allein bewerkstelligen kann. Somit ist es naheliegend, das Spielfeld auf der Oberfläche durch weitere Unterteilungen, mit insgesamt 4 einzelnen GridPanes zu realisieren. Diese fungieren jeweils als ein Quadrant des Spielfelds mit 3x3 Zellen. Hier werden weitere JavaFX-Komponenten angeordnet, welche die Zellen repräsentieren.

Da bei Pentago Spielsteine als Spielelemente für die Spieler verwendet werden, wäre hierbei der Gebrauch von Labels oder Buttons zwar vorstellbar, doch die Verwendung von ImageViews ist natürlich am besten dafür geeignet, vorgefertigte Bilder darzustellen. Nachteil hierbei ist jedoch, dass für jedes Symbol ein eigenes Bild kreiert werden und als Datei dem Projekt beigelegt werden muss.

### Realisationsbeschreibung

Das Spielfeld wird durch ein äußeres Haupt-GridPane mit je 2 Zeilen/Spalten repräsentiert. Jede Zelle wird weiter durch jeweils 1 GridPane mit je 3 Zeilen/Spalten dargestellt. In jeder Zelle wird ein ImageView eingebunden. Bei der Erstellung eines ImageView werden durch das Hinzufügen zum GridPane, die Koordinaten der entsprechenden Zelle gesetzt, um im späteren Spielverlauf eine Zuordnung zu ermöglichen. Um die korrekte Anpassung bei einer Größenveränderung zu gewährleisten, wird die Größe eines ImageView an die der Zelle des GridPanes gebunden. Parallel dazu werden die erstellten Komponenten in einem ImageView-Array abgelegt, welches der GUI im Anschluss übergeben wird.

## 2.2 GUI-Rotation eines Quadranten

### Problemanalyse

Nach jeder Platzierung eines Spielsteins, dreht der jeweilige Spieler einen Quadranten.

Jeder Quadrant kann um 90 Grad nach links oder nach rechts gedreht werden. Bei der Drehung werden auch die Spielsteine rotiert, sodass das Spielfeld eine andere Belegung aufweist.

### Realisationsanalyse

Jeder Quadrant benötigt eine Möglichkeiten um nach links, und eine Möglichkeit um nach rechts gedreht zu werden. Dazu eignen sich pro Quadrant 2 Buttons. Einer für die Rotation gegen- und einer für die Rotation im Uhrzeigersinn.

Da nach der Rotation eines GridPanes die Koordinaten der Zellen nicht mehr den augenscheinlichen entsprechen, empfiehlt es sich, die Drehung nach der Animation zurückzusetzen und die angezeigte Zellenbelegung anzupassen.

Für die Durchführung der Animation eignet sich eine RotateTransition.

### Realisationsbeschreibung

Es werden jeweils ein Buttons für die Drehung nach links und für die Drehung nach rechts am Rande jedes Quadranten angebracht. Durch den Klick auf einen dieser Rotationsbuttons reagiert der FXMLController auf das ActionEvent und teilt der Logik mit, welcher Quadrant und in welche Richtung gedreht werden soll. Nach erfolgreicher Prüfung des korrekten Ablaufes in der Methode handleRotation(), wird die GUI angewiesen die Rotation darzustellen. In der GUI wird eine RotateTransition mit übergebener Dauer gestartet und das ausgewählte GridPane rotiert. Nach Ablauf dieser Animation wird die Rotation des GridPanes durch setRotate(0) wieder zurückgesetzt. Anschließend wird mit der Methode rotateImageViews() für jedes ImageView des gedrehten Quadranten die Constraints gesetzt und somit die Zellenbelegung wieder angepasst.

## 2.3 Spieltheorie Alpha-Beta-Suche

### Problemanalyse

Die Berechnung für einen Computer-Spielzug erfolgt durch eine Auswertung mit dem Alpha-Beta Algorithmus. Da perfekte Information besteht, kann eine optimale Strategie entwickelt werden. Alle Zugmöglichkeiten werden betrachtet.

### Realisationsanalyse

Der Spielbaum stellt durch eine baumartige Struktur, alle möglichen Abläufe des Spielverlaufs für eine bestimmte Länge dar.

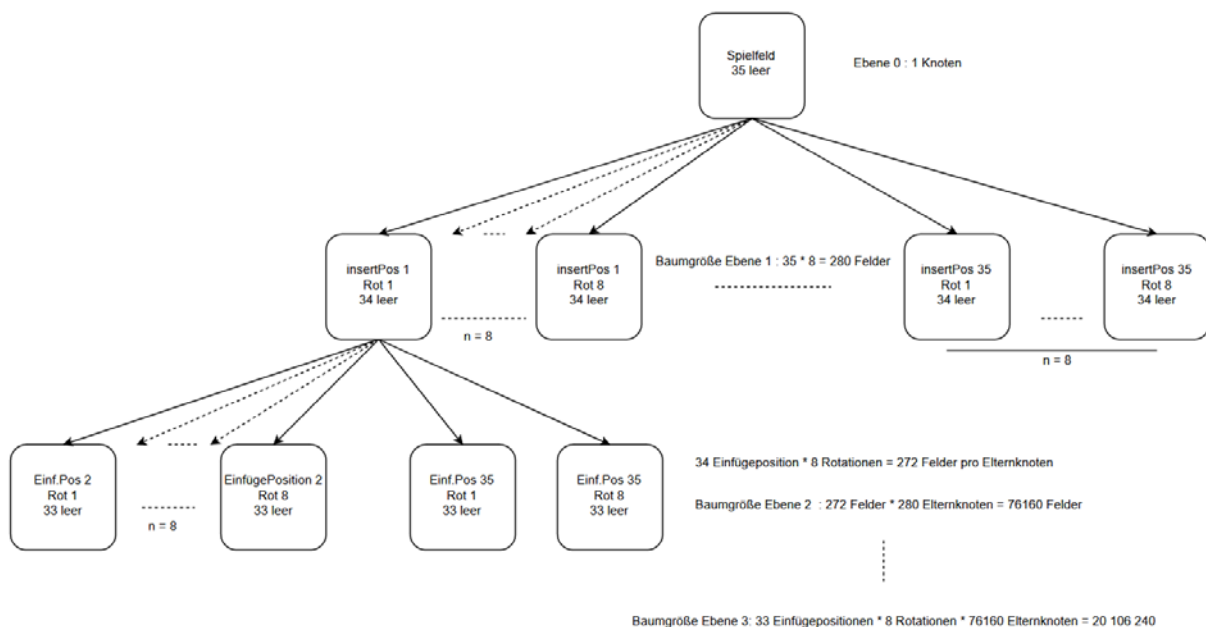


Abbildung 2a. Ein Baum der Länge 2 nach dem ersten Spielzug des Spielers.

Nach dem ausgeführten Spielzug des Spielers bestehen für den darauffolgenden Computerzug mehrere Legemöglichkeiten. Die Knoten des Baumes repräsentieren alle Zugmöglichkeiten, die für den zukünftigen Spielverlauf denkbar sind. Da auf das Platzieren eines Spielsteins immer das anschließende Drehen eines Quadranten folgt, resultieren für jede Legepositionen je nach Rotation, 8 mögliche Spielsituationen. Infolgedessen wird der Baum schnell sehr breit und beinhaltet bereits in der ersten Ebene bis zu 280 Knoten.

Bei einer Baumtiefe von 3, weist der Baum insgesamt 20 160 240 Elemente auf.

Als Datenstruktur für Felder im Baum bieten sich verschiedene Möglichkeiten an:

### 1. 2D-Array des Aufzählungstyps

Für die Aufzählung der möglichen Belegungen (Spieler, Computer, leer) durch Enums ergeben sich 36 Worte. Für das 2D-Array kommen weitere 7 Worte hinzu.

Somit ergeben sich 43 Worte für die Darstellung einer Spielsituation.

32 Bit \* 43 Worte = 1376 Bit = 172 Byte

64 Bit \* 43 Worte = 2752 Bit = 344 Byte

### 2. 1D Array des Aufzählungstyps

Für die Aufzählung der möglichen Belegungen (Spieler, Computer, leer) durch Enums ergeben sich 36 Worte. Für das 1D-Array kommt ein weiteres Wort hinzu.

Es ergeben sich 38 Worte für die Darstellung einer Spielsituation.

32 Bit \* 38 Worte = 1216 Bit = 152 Byte

64 Bit \* 38 Worte = 2432 Bit = 304 Byte

### 3. BigInteger

Für jede mögliche Belegung (Spieler, Computer, Leer) werden 2 Bits benötigt.

Dies erfordert bei 36 Positionen mindestens einen Speicher von 72 Bit.

Für die Darstellung eines Spielfeldes benötigt man dementsprechend nur 72Bit (9 Byte).

## Realisationsbeschreibung

Die Berechnung des Computerzuges gestaltet sich als langwierig und speicheraufwändig.

Trotz der Effizienz des Alpha-Beta-Suche kann der Baum bei ungünstiger Auswertungsreihenfolge sehr breit werden. Für die Kindknoten des Baums werden BigInteger-Werte verwendet, die durch ihre Bits die Belegung des Spielfeldes abbilden.

Die Klasse Tree implementiert die Methode alphabeta(). Diese wertet anhand des Alpha-beta-Algorithmus den weiteren Spielverlauf aus, indem für jede freie Position (2 Bits im BigInteger auf 0) einen Spielzug nachgebildet wird. Durch rekursiven Ablauf kommt man bei der maximal gesetzten Tiefe des Baums an und wertet das Spielfeld anhand der Bewertungsmethode der Klasse Board aus. Zwischenzeitlich mögliche Gewinne werden durch Bit-Masken-Operationen der Klasse WinValidator erkannt.

Ein Spielende wird bewertet mit:

Sieg von Computer = Integer.MAX\_VALUE, Sieg von Player = Integer.MIN\_VALUE

Es kann passieren, dass der Computer keine Möglichkeit mehr hat zu gewinnen, da der Spieler durch mehrere Konstellationen gewinnen kann. Hierbei muss trotzdem ein Zug berechnet werden. Dazu wird bei einer Auswertung von einem Gewinn des Spielers, der Baum erneut mit einer kürzen Tiefe durchlaufen.

Folgendermaßen zeigt sich der Arbeitsspeicheraufwand durch Messungen mit dem Windows-Task-Manager:

Start der jar. Datei: 125 MB

	Rekursionstiefe 2	Rekursionstiefe 3
1. Computerzug	178 MB	802 MB
2. Computerzug	211 MB	750 MB
5. Computerzug	190 MB	720 MB

## 2.4 Rotation eines Quadranten im Baum

### Problemanalyse

Die Rotation eines Quadranten wird sowohl nach einem Spielzug des Spielers, als auch nach einem Spielzug des Computers durchgeführt. Dabei wird der Quadrant um 90 Grad gegen- oder im Uhrzeigersinn gedreht.

### Realisationsanalyse

Rotationen eines Quadranten finden in 2 verschiedenen Bereichen statt. Zum einen, für die Erstellung eines Kindknoten im Baum und zum anderen für die Vertauschung der Elemente der Spielfelddatenstruktur in der Logik.

Die Rotation wird bei der Auswertung des Computerzuges in manchen Spielsituationen über tausende Male durchgeführt und muss daher sehr effizient, mit geringer Laufzeit und geringem Speicheraufwand erfolgen. Leere Quadranten, sowie Quadranten mit nur einem belegten Feld in der Mitte sind neutral und brauchen nicht verändert werden.

Das Spielfeld hat in der BigInteger Repräsentation 72 Bit (Jede Zelle 2 Bit). Die einzelnen Bits können durch die Methode toString(radix=2) entnommen werden.

Darstellung des Spielfeldes in folgender Reihenfolge:

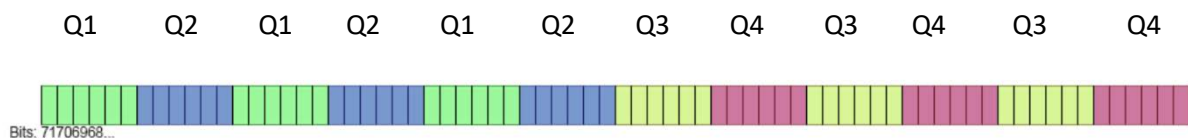


Abbildung 2b. Bitbelegung eines Spielfelds in einem BigInteger-Wert.

Für die Vertauschung einzelner Bits kommen folgende Implementierungen infrage:

#### 1. Bit-Operationen

Die entscheidenden Bits des BigInteger-Wertes werden durch Bit-Operationen ausgelesen und einzeln vertauscht. Mit der Methode testBit(index n) wird an der Stelle n ein Bit ausgelesen werden. Mit der Methode setBit(index n) wird ein Bit an der Stelle index gesetzt. Die Darstellung für einen Quadranten besteht aus 18 Bits. Da das mittlere Feld nicht vertauscht wird, müssen für jede Rotation 16 Bits umgeschrieben werden. Da jedoch setBit(), immer ein Bit setzt, muss zuvor noch durch eine if-Abfrage überprüft werden, ob die zu schreibenden Bits gesetzt sind. Somit erhöht sich der Aufwand.

## 2. Arrays

Eine Datenstruktur mit indizierten Zugriffen bietet durch effektives Vertauschen der Elemente einen Vorteil für eine Rotation. Für die Rotation anhand von `String[s]?` eignen sich die Methoden der Klasse `String` nur bedingt. Stattdessen lässt sich aber die `String`-Repräsentation leicht in ein `char`-Array umwandeln, an welchem dann, durch indizierte Zugriffe, die Rotation des Quadranten sehr effizient durchgeführt werden kann. Bei der Erstellung eines `BigInteger`-Wertes für ein Spielfeld, wird dieses von links nach rechts und von oben nach unten durchlaufen. Dadurch ergibt sich eine Streuung des Quadranten in der Bitdarstellung. Zudem muss sichergestellt werden, dass das Array trotz der von-links weggelassenen 0Bits des `BigInteger`, die komplette Länge erhält. Um die Bits für einen Quadranten zu vertauschen, werden 16 Zuweisungen durchgeführt wonach anschließend ein neues `BigInteger` durch das `char[]` erstellt wird.

## 3. Rotation nach Umwandlung in die Spielfeld-Klasse

Die Klasse `Board` implementiert bereits eine geeignete Methode zur Rotation eines Quadranten, sowie ebenfalls auch einen Konstruktor mit einem `BigInteger`-Parameter. Sie bietet den Vorteil, dass kein neuer Programmcode mehr geschrieben werden muss und die Rotation einheitlich durch eine Methode ausgeführt wird. Eine Vertauschung erfolgt durch 8 Zuweisungen. Anschließend muss das Spielfeld wieder in ein `BigInteger` geschrieben werden. Durch die Umwandlung eines `BigInteger`-Wertes in `Board` erhöht sich auch hier der Aufwand. Um die Quadranten des Boards nach der Erstellung eines Kindes nicht zurückzudrehen, muss für jeden Kindknoten im Baum ein neues `Board`-Spielfeld erstellt werden. Das bedeutet, dass für jeden Knoten Laufzeit für die Umwandlung, sowie Speicherplatz für die `Board`-Instanz verbraucht wird.

## Realisationsbeschreibung

Durch die Analyse aus Kapitel 2.4 (Datenstrukturen für Felder in einem Baum), ergibt sich bereits in der zweiten Ebene eine Baumbreite von 76160 Knoten. Um eine effiziente Performance zu gewährleisten, werden die `BigInteger`-Werte nicht in die `Board`-Klasse umgewandelt, sondern anhand von `char`-Arrays durchgeführt.

Dazu implementiert die Klasse `Rotator` die Methode `performRotation()`. Diese testet zuerst durch eine Prüfung mit Bit-Masken, ob die zu vertauschenden Bits eventuell alle nicht gesetzt sind und somit die Rotation übersprungen werden kann.

Anschließend wird mithilfe der `toCharArray()`-Methode der `BigInteger`-Wert in ein `char`-Array mit korrekter Länge konvertiert und die Elemente des Arrays vertauscht.



## 2.5 Bewertung einer Spielsituation

### Problemanalyse

Ein Spielfeld muss in jeder Konstellation bewertbar sein. Für die Bewertung eines Blattes im Baum, muss die Belegung eines Spielfeldes aussagekräftig bewertet werden, um anhand von Vergleichen die Entscheidung für den nächsten Zug zu treffen. Ein möglicher Gewinn eines Spielers muss rechtzeitig erkannt werden. Des Weiteren soll es möglich sein, die Strategie des Computers zwischen offensiv und defensiv zu verändern. Ist die Strategie des Computers defensiv eingestellt, so soll zunächst primär verhindert werden, dass der Spieler auf einem Quadranten 3 Steine in einer Linie erhält.

### Realisationsanalyse

Um eine informative Aussage über den Zustand einer Spielsituation zu machen, ist es nötig auf eine geschickte Art und Weise die Belegungen der Spielsteine konstruktiv auszuwerten. Es braucht ein Verfahren, dass ein Spielfeld für beide Spieler gleichzeitig auswertet.

Hierbei wären verschiedene Ansätze denkbar:

#### 1. Komplette Auswertung

Das Spielfeld wird für beide Spieler gleichzeitig ausgewertet. Es werden alle potenziellen Gewinnreihen überprüft. Sowohl vor der Drehung als nach einer Drehung. Somit ist es möglich eine optimale Aussage über das Spielfeld zu treffen.

#### 2. Einfache Auswertung

Das Spielfeld wird ebenfalls für beide Spieler gleichzeitig ausgewertet.

Statt jedoch die anschließende Rotation zu berücksichtigen, wird das Spielfeld nur in vorliegender Konstellation betrachten und ausgewertet.

Der Vorteil ist, dass wesentlich weniger Reihen analysiert werden müssen.

Der Nachteil hierbei ist, das potentielle Reihen, die nach einer Drehung entstehen würden nicht betrachtet werden und somit nicht in die Bewertung mit einfließen.

### Realisationsbeschreibung

Durch die Annahme, dass das Spiel für einen Spieler lösbar sein muss, braucht die Auswertung eines Spielfeldes nicht an das Höchstmaß durchgetestet zu werden. Da beim Erstellen von Spielfeldern im Baum, alle Platzierungen und Drehungen im vollen Umfang durchgeführt werden, reicht es aus nur einfache potentielle Gewinnreihen zu erkennen. Dazu implementiert die Board-Klasse die Methode `evaluation()`. Es werden alle Zeilen, Spalten sowie die beiden Hauptdiagonalen und jeweils 2 anliegende kürzere Diagonalen betrachtet. Dazu wird in der `evaluateSix()`-Methode die Reihe durchlaufen und je nach Spielerstein Punkte aufsummiert. Handelt es sich um einen Spielerstein, wird die Spielervariable inkrementiert. Handelt es sich um einen Computerstein, wird die Computervariable inkrementiert. Andernfalls die Variable die die freien Positionen repräsentiert. Zusätzlich werden für beide Spieler boolesche Flags gesetzt, falls auf den inneren Positionen der Reihe (1-4) ein Spielerstein vorliegt und somit für den Gegner eine Gewinnreihe verhindert wird. Danach wird überprüft ob es eine Spielervariable gibt die inklusive der freien-Positionen Variable  $\geq 5$  ist und gleichzeitig das Flag für des Gegners nicht gesetzt ist. Sollte es der Fall sein gibt es für die Konstellation  $\text{Spielervariable(Häufigkeit)} * 10 + \text{freie Positionen Punkte}$ .

Ebenfalls wird im Anschluss geprüft ob die Strategie des Computers defensiv ist. In diesem Fall gibt es nach erfolgreicher Prüfung durch die beiden Methoden `twoOfThreeArePlayer()` und `blocked()` 45 Punkte. Ansonsten ist diese Reihe unbrauchbar und weißt eine Bewertung von 0 auf.

Sollte eine Reihe eine Bewertung von mindestens 50/-50 Punkten aufweisen, wird erkannt, dass ein Spieler gewinnen kann und die Auswertung des Spielfelds endet. In diesem Fall ist der Rückgabewert für einen Gewinn des Computer `Integer.MAX_VALUE` und für einen Gewinn des Spieler `Integer.MIN_VALUE`.

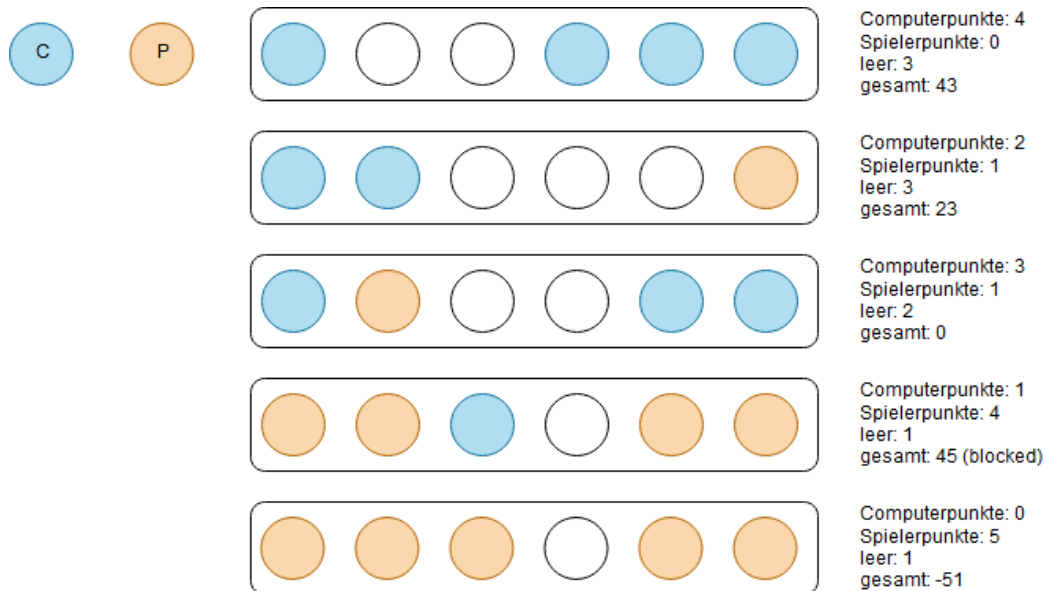


Abbildung 2c. Bewertung von 6 Spielsteinen.

## 2.6 Gewinn eines Spielers erkennen

### Problemanalyse

Ein Spiel kann entweder direkt nach der Platzierung eines Spielsteins oder auch nach der darauffolgenden Rotation gewonnen werden. Somit muss der Gewinn eines Spielers zu verschiedenen Zeitpunkten getestet werden. Des Weiteren muss es ebenfalls bei der Erstellung der Kindknoten im Baum möglich sein, ein Spielende zu erkennen, um rechtzeitig die Rekursion zu verlassen.

Wenn Gewinnreihen für beide Spieler entstehen endet das Spiel unentschieden.

Alle Spielsteine, die zu einer Gewinnreihe führen, müssen auf der GUI markiert werden.

### Realisationsanalyse

Die Regeln des Spiels sehen vor, dass ein Spieler gewinnt sobald er 5 eigene Steine in eine Reihe bildet. Somit ist erst ab dem zehnten Spielzug (fünften Spielzug des Spielers) möglich eine Gewinnreihe zu bilden. Horizontal und vertikal gibt es somit für jede Zeile/Spalte 2 Möglichkeiten eine Gewinnreihe zu bilden. Für die beiden Hauptdiagonalen kommen jeweils 2 weitere hinzu. Die 4 kurzen Diagonalen bilden jeweils eine weitere Option eine Gewinnreihe aufzubauen. Somit ergeben sich bei einem Spielfeld mit 36 Position insgesamt 32 mögliche Gewinnkonstellationen.

Um zu testen ob ein Spieler gewonnen hat wären folgende Ansätze denkbar:

### 1. Spielfeld durch Systematische Auswertung prüfen, ob ein Gewinn vorliegt

Dieser Ansatz bietet den Vorteil, dass die Implementierung für die Bewertung des Spielfeldes übernommen werden kann, da diese das gewünschte Ergebnis liefert. Bei der Erstellung des Baums jedoch, müsste man für jeden Kindknoten, den vorliegenden BigInteger-Wert in die Datenstruktur der Logik umwandeln und anschließend für jeden Knoten 32 Möglichkeiten testen. Für über tausende von Knoten ist dieses Vorgehen weniger performant.

### 2. Binären String des BigInteger durch contains() oder Pattern-Matching validieren

Ein Gewinn wird durch einen Vergleich mit einem regex-Ausdruck validiert.

Für horizontal, vertikal und diagonal wird nur noch jeweils ein Ausdruck benötigt, da die Verteilung der Bits jeweils gleich ist. Dieser Ansatz ist effizient doch bietet leider einen Nachteil, der das Ergebnis verfälscht. Bei der Repräsentation des Feldes im BigInteger, werden die Zeilen hintereinandergeschrieben. Dadurch ist die Struktur des Feldes nicht mehr gegeben, sodass folglich Gewinne über Zeilen/Spalten vermischt werden.

### 3. Bit-Masken

In einem Spielbaum wird das Spielfeld als BigInteger-Wert gespeichert. Diese Klasse stellt Bit-Operationen zur Verfügung, wodurch es möglich ist, mehrere BigInteger-Instanzen miteinander zu vergleichen. Durch die AND Verknüpfung und anschließendem Auslesen der gesetzten Bits erhält man die Anzahl der übereinstimmenden Bits, und erkennt ab einer Anzahl von mindestens 5 einen Gewinn.

## Realisationsbeschreibung

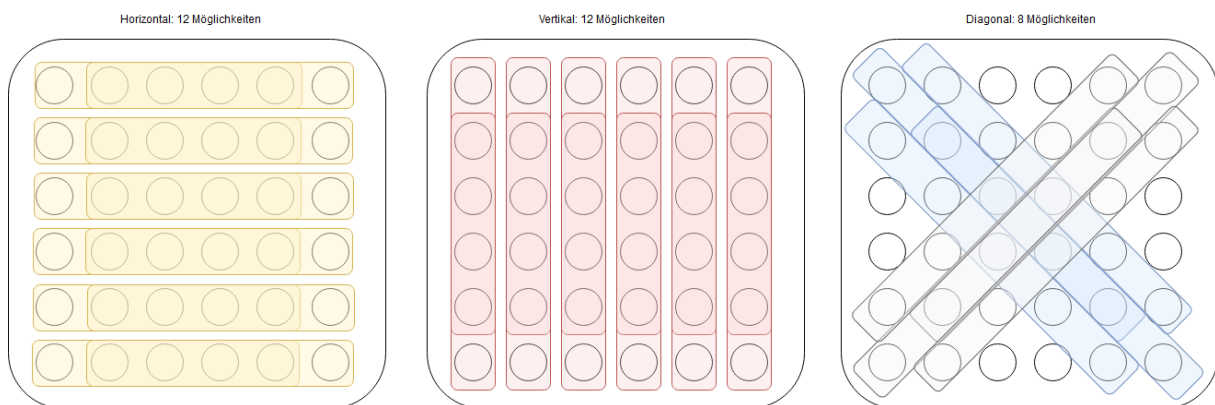


Abbildung 2d. Die 32 Gewinnmöglichkeiten bei Pentago.

Bit-Operationen haben einen sehr niedrigen CPU-Aufwand und sind dadurch sehr performant. Durch die Häufigkeit der Gewinnüberprüfung eignet sich es sich, ein Spielfeld als BigInteger-Wert durch 32 Bit-Masken durch eine AND-Bit-Operation festzustellen, ob die relevanten Bits gesetzt sind. Um dies zu ermöglichen wird die Klasse WinValidator erstellt. Diese enthält ein BigInteger-Array mit 32 Bitmasken, von denen jede eine mögliche Gewinnbelegung darstellt. Die Klasse besitzt als einzige Methode getSetWinBits(), welche ein Set der übereinstimmenden Bit-Masken liefert. Somit ist es in der Logik möglich, für alle Bit-Masken des Sets, die gesetzten Bits auszulesen und die Spielsteine die zu einer Gewinnreihe führen auf der GUI zu markieren.

## 2.7 Gewährleistung des korrekten Ablaufs

### Problemanalyse

Ein Spielerzug besteht immer aus dem Platzieren eines Spielsteins auf ein leeres Feld und der darauffolgenden Rotation eines Quadranten. Es muss gewährleistet werden, dass das Platzieren eines Spielsteins sowie die Rotation in korrekter Reihenfolge ausgeführt wird und erst stattfinden kann, wenn die vorherige Animation beendet ist.

Des Weiteren ist sicherzustellen, dass das Speichern eines Spiels sowie das Zurücknehmen von Spielzügen nur nach einem durchgeführten Computerzug ausgeführt werden kann.

Das Spiel endet sobald eine Gewinnreihe von mindestens 5 gleichen Spielsteinen entsteht.

Dies ist sowohl nach der Platzierung eines Spielsteins, sowie auch nach der Drehung eines Quadranten möglich. Im Anschluss wird der Benutzer über Sieg oder Niederlage informiert.

### Realisationsanalyse

Für den korrekten Ablauf des Spiels muss mithilfe von Zuständen sichergestellt werden, dass der Spieler durch seine Handlung keinen falschen Zustand des Spiels herbeiführen kann.

Der FXML-Controller ruft bei jeder Handlung des Benutzers die entsprechenden Methoden der Logik auf. Diese kann durch boolesche Zustände die Gültigkeit des gewünschten Handelns validieren und durchführen.

Da die Kontrollelemente des Spiels durch Buttons realisiert werden, bietet es sich an zusätzlich die Buttons entsprechend der Ablauflogik zu Aktivieren und Deaktivieren.

### Realisationsbeschreibung

Für die Gewährleistung der Ablauflogik dient die Klasse GameLogic. Um Aktionen des Spielers auf ihrer Korrektheit zu prüfen werden boolesche Zustände eingeführt.

Dazu besitzt die GameLogic die Zustände rotationRemain, actualPlayer und bothPlayerMoved.

So kann vor dem Platzieren eines Spielsteins mit !rotationRemain überprüft werden, dass nicht auf eine ausstehende Rotation gewartet wird.

Nachdem die GUI eine Animation durchgeführt hat, ruft sie die Logic durch die Methode proceedAfterGuiRotation() wieder auf.

Durch die Auswertung von actualPlayer kann der aktuelle Spieler ermittelt werden. So wird im Falle vom Spieler, auf den Computerspieler gewechselt und der Zug des Computers durchgeführt.

Nachdem der Spielzug des Computers durchgeführt wurde wird der Zustand von bothPlayerMoved auf true gesetzt und so das Zurücknehmen und Wiederholen von Zügen ermöglicht.

Parallel dazu wird im Laufe des Spiels durch die beiden Methoden der GUI disableTmSaveAndHistory() und disableRotButtons() die Kontrollelemente zu entsprechenden Momenten die Kontrollelemente der Oberfläche aktiviert und deaktiviert.

## 2.8 Spielstand speichern / laden

### Problemanalyse

Um ein Spiel zu einem späteren Zeitpunkt wieder aufnehmen zu können, soll der Spielstand nach jedem Computerzug gespeichert werden können. Der Benutzer soll hierfür den Speicherort und einen Dateinamen auswählen können. Als Dateiendung soll per Default '.pen' verwendet werden. Der Spielstand ist definiert durch das Spielfeld, welches in einer Textdatei abzulegen ist, in der jede Spielfeldzeile durch eine Textzeile mit sechs Zeichen dargestellt wird.

### Realisationsanalyse

Ein freier Platz wird durch '-', ein Computerstein durch 'C' und ein Spielerstein durch 'P' dargestellt.

Ein gespeicherter Spielstand kann jederzeit geladen werden. Passiert dies während eines Spiels, so wird das aktuelle Spiel bei erfolgreichem Laden ohne Nachfrage verworfen, andernfalls nach einer aussagekräftigen Fehlermeldung wieder aufgenommen.

### Realisationsbeschreibung

Das Speichern eines Spielfeldes wird dem Spieler über das Menü, unter dem Eintrag Spiel speichern angeboten. Hierbei wird durch die Methode `saveGame()` des `FXMLController` ein `FileChooser` erstellt, der dem Nutzer ein Dialogfenster mit einem Datei-Explorer zur Verfügung stellt. Für die voreingestellte Dateiendung '.pen' wird dem `FileChooser` ein `ExtensionFilter` hinzugefügt.

Nach erfolgreicher Auswahl für das Speicherziel, bekommt die Logik die Datei übergeben, in die gespeichert wird.

Die Klasse `Board` implementiert die Methode `toString()`, welche für die aktuelle Spielfeldbelegung einen String, mit Zeilenumbruch nach jeder Zeile erstellt.

Für das Schreiben und Lesen in/aus eine/einer Datei dient die Klasse `FileUtil`. Dessen Methode `writeGamefield()` schreibt den erstellten String des Spielfeldes in die ausgewählte Datei.

Das Laden eines Spielstandes, wird im `FXMLController` durch die Methode `loadGame()` realisiert. Wie auch beim Speichern, wird auch hier dem Spieler per `FileChooser` die Option geboten eine Datei auszuwählen, welche ebenfalls im Anschluss der Logik übergeben wird.

Mit der Methode `loadFile()` wird in der Logik, durch die `readGamefield()`-Methode der `FileUtil`, versucht ein Spielfeld aus der Datei zu lesen. Der resultierende String wird daraufhin durch den Konstruktor der `Board`-Klasse validiert und schmeißt bei ungültigem Format eine Exception, die die Logik abfängt und durch eine Alert (`alertError()` der GUI) den Nutzer informiert.

## 2.9 Protokoll - Spielzüge zurücknehmen / wiederholen

### Problemanalyse

Neben dem Spielfeld ist ein Protokoll anzuzeigen, welches jeden ausgeführten Schritt textuell darstellt.

Ein ausgeführter Spielzug und der darauf gefolgte Computerzug sollen zurücknehmbar und auch wiederholbar sein, was auch im Protokoll sichtbar werden soll. Wurde nach einem zurückgenommenen Zug ein neuer Zug ausgeführt oder die Strategie des Computers geändert, so können die bis dahin zurückgenommen Züge nicht mehr wiederholt, sondern müssen dann aus dem Protokoll entfernt werden.

### Realisationsanalyse

Nach jedem Zug und der darauffolgenden Rotation muss eine Abbildung des Spielfeldes, sowie der durchgeführte Zug abgespeichert werden. Das Zurücknehmen besteht nicht daraus die Drehung rückgängig zu machen und den Spielstein zu entfernen, sondern einfach die vorherige Spielsituation wieder aufzunehmen.

### Realisationsbeschreibung

Um einen Spielerzug im Protokoll darzustellen, wird die Methode `writeLog()` der GUI mit dem entsprechenden String als Parameter aufgerufen. Diese wiederum fügt dann dem TextArea Element, den übergebenen String per `appendText()` bei.

Für die Option einen Spielzug zurückzunehmen und zu wiederholen wird jeweils ein Button im rechten Teil des Fensters, über dem Protokoll angebracht.

Die Spielhistorie wird in einer Liste vom Typ `LogEntry` abgespeichert, welches nach jedem durchgeführten Zug der Liste hinzugefügt wird. Ein `LogEntry` beinhaltet die Spielfeldbelegung als String-Repräsentation, den Spielernamen als String, die Position des Einfügens und die durchgeführte Rotation.

Sobald ein Computerzug durchgeführt wurde, ermöglicht das gesetzte boolesche Flag `bothPlayerMoved`, durch die Spielhistorie vor- und zurückzugehen.

Zusätzlich wird ein Indexzähler für die Liste eingefügt, der sich die momentane Position der Liste merkt und somit einen indizierten Zugriff der Liste ermöglicht.

Bei zurücknehmen eines Zuges, wird das Spielfeld der Logik, durch den vorletzten Eintrag ab dem gesetzten Index ersetzt. Beim Wiederholen eines Zuges, wird in der selber Art und Weise der übernächste Eintrag der Liste verwendet. Nach jedem ausgeführten Zug des Spielers, sowie nach jeder Änderung der Strategie, werden die Einträge Liste ab dem gesetzten Index gelöscht.

Während durch die Spielhistorie gesprungen wird, werden die Einträge aus dem Protokoll zeilenweise gelöscht, beziehungsweise durch die gesetzten Attribute des `LogEntry` wieder in das Protokoll geschrieben.

### 3 Programmorganisationsplan

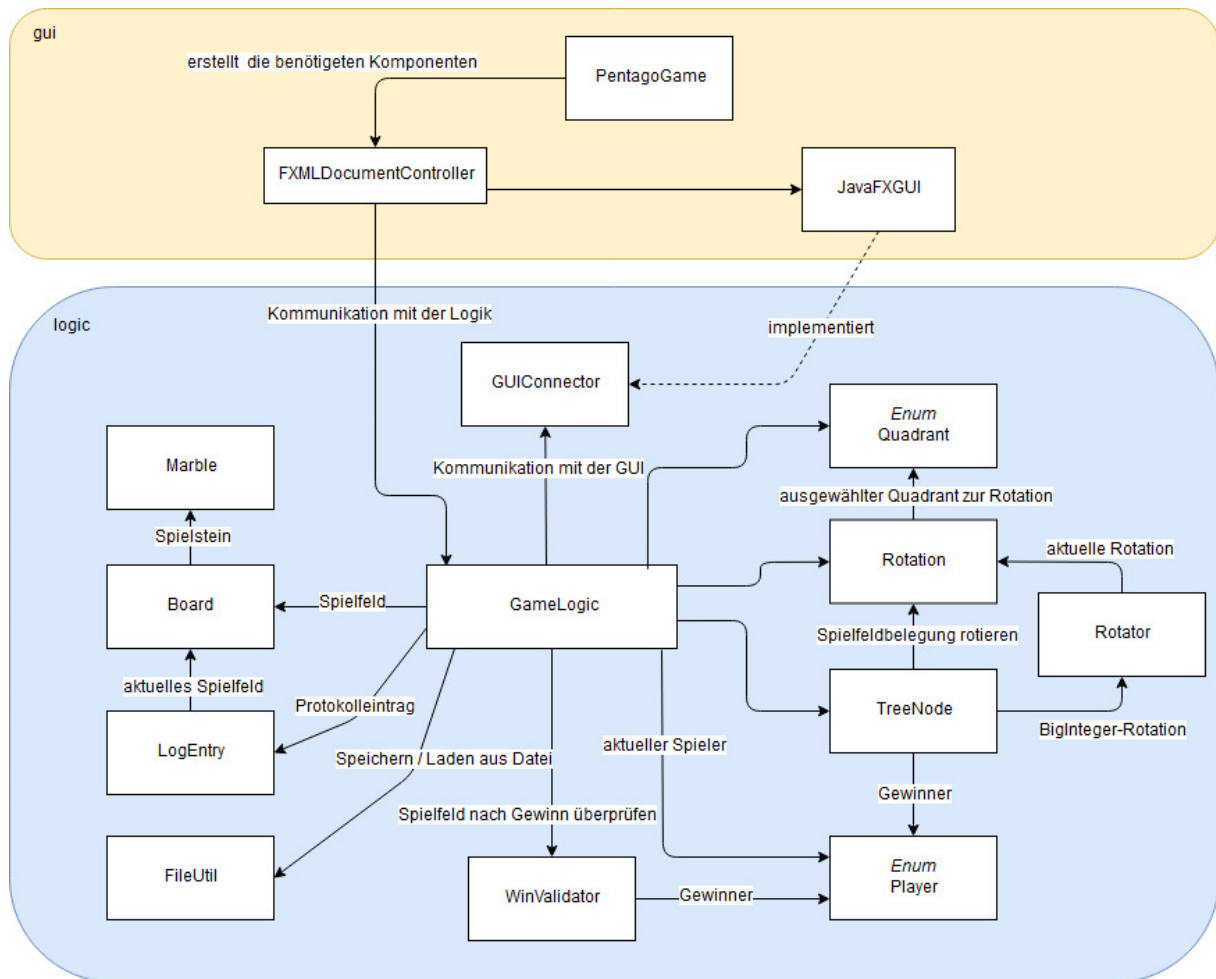


Abbildung 3a: Händisch erstellter POP

#### Anmerkungen

Die farbliche Kennzeichnung ermöglicht hier eine einfache Zuordnung der Klassen zu den Paketen. Die Beschriftung der Pfeile geben an, in welchem Zusammenhang die Klassen miteinander in Verbindung stehen. Ein ausgehender Pfeil repräsentiert das Instanzieren der Klasse und Aufrufen der Methoden, in die der Pfeil gerichtet ist.

## 4 Beschreibung grundlegender Klassen

Board
- field : Marble[] - LINESIZE_DEF: Integer - linesize: int - size: int - emptyPositionsNum: - pointsTarget:
+ Board() + Board(String) - blocked(Marble...): - setCell(Marble, int, int): void + evaluate(boolean): + performRotation(Rotation): + toBigInt(): BigInteger + insert(Marble, int, int): + toString(): String

### Board

Die Klasse Board Repräsentiert das Spielfeld als zweidimensionales Array. Sie besitzt Zustände über die Größe und Anzahl leere Positionen des Feldes.

Board implementiert ein Konstruktor der für den Start eines Spiels ohne Parameter aufgerufen wird.

Ein weiterer Konstruktor ermöglicht das Initialisieren eines Feldes durch einen String. Dieser wird beim Laden von gespeicherten Zuständen sowie für das Zurücknehmen und wiederholen von Spielzügen verwendet.

FileUtil
+ readGamefield(File): String + writeGamefield(File, String)

### FileUtil

Hier werden die 2 Methoden für das Speichern und Laden von Dateien zur Verfügung gestellt.

GUIConnector <<interface>>
+ setImage(Player, int, int): void + rotateQuadrant(Quadrant, boolean, long, GameLogic): void + writeLog(String): void + alertWin(Player, GameLogic): void + alertError(String): void + markWin(int, int): void

### GUIConnector

GUIConnector ist das Interface für die JavaFXGUI sowie für die FakeGUI.



## Fortsetzung Beschreibung grundlegender Klassen

<b>GameLogic</b>
<ul style="list-style-type: none"> <li>- gui: GUIConnector</li> <li>- gamefield: Board</li> <li>- tree: Tree</li> <li>- logEntry: LogEntry</li> <li>- recordList: List&lt;LogEntry&gt;</li> </ul>
<ul style="list-style-type: none"> <li>+ GameLogic(GUIConnector, Board)</li> <li>+ handlePlayerMove(int, int, int, int): void</li> <li>+ handleAiMove(): void</li> <li>- checkWin(): void</li> <li>+ handleRotation(Quadrant, boolean): int</li> <li>+ undoMove(): void</li> <li>+ redoMove(): void</li> <li>+ loadFile(File): void</li> <li>+ saveGame(File): void</li> </ul>

### GameLogic

GameLogic dient zur Realisation der Logik des Spiels. Hier werden die korrekte Reihenfolge und Ausführung der Spielzüge gewährleistet.

Das Spielfeld wird mittels einer Board-Instanz verwaltet.

Durch die Kommunikation mit der GUI wird auf das Handeln des Spielers reagiert und bei korrektem Ablauf das Spielfeld modifiziert. Nach jeder Platzierung und jeder Rotation wird überprüft, ob ein Spieler gewonnen hat.

Für die Berechnung des nächsten Computerzuges wird die Klasse Tree gestartet.

Für das Zurücknehmen und Wiederholen von Spielzügen wird die Spielhistorie in einer Liste mit LogEntry als Elementen verwaltet.

<b>LogEntry</b>
<ul style="list-style-type: none"> <li>- field: String</li> <li>- player: String</li> <li>- xCoord: int</li> <li>- yCoord: int</li> <li>- rot: Rotation</li> </ul>
<ul style="list-style-type: none"> <li>+ LogEntry(String)</li> <li>+ getField(): String</li> <li>+ getPlayer(): String</li> <li>+ getX(): int</li> <li>+ getY(): int</li> <li>+ getRotation(): Rotation</li> </ul>

### LogEntry


Für die Erstellung einer Spielhistorie werden die Spielzüge der Spieler und die derzeitige Belegung des Spielfeldes in einer Liste gespeichert. Diese Information wird in einem LogEntry abgelegt. Eine Instanz enthält das Spielfeld als String, sowie die Angaben über den Spieler und dessen getätigten Spielzug.

<b>Marble</b>
<ul style="list-style-type: none"> <li>- player: Player</li> </ul>
<ul style="list-style-type: none"> <li>+ Marble(String)</li> <li>+ toBinary(): String</li> <li>+ toString(): String</li> <li>+ getPlayer(): Player</li> </ul>

### Marble


Marble bildet einen Spielstein ab. Dieser enthält als einzige Instanzvariable seinen Spielertyp. Durch die Methode toBinary() wird Marble als Binärzahl an die Erstellung eines BigIntegers übergeben. Die Methode toString() wird für die Erstellung der Spielfeldes in String-Darstellung verwendet.

## Fortsetzung Beschreibung grundlegender Klassen

 <b>Rotation</b>
- qdr: Quadrant - toLeft: boolean
+ Rotation(Quadrant, boolean) + Rotation() + next(): void + copy(): Rotation


### Rotation

Stellt eine Instanz für eine Rotation dar. Beinhaltet Attribute über den Quadranten und die Richtung, in die die Rotation erfolgt. Die Methode next() implementiert den Durchlauf der 8 Möglichkeiten einer Rotation (4 Quadranten, jeweils 2 Richtungen).

 <b>Rotator</b>
+ Rotator() - toCharArray(BigInteger): char[] - canSkip(BigInteger, Rotation): boolean + performRotation(BigInteger, Rotation): BigInteger

### Rotator


Rotiert einen BigInteger-Wert durch vorherige Umwandlung in ein char-Array. Prüft zuvor durch Bit-Masken, ob die Rotation eventuell keine Auswirkung hat und somit ausgelassen werden kann.

 <b>Tree</b>
- NUMOFROTATIONS: int - rotation: Rotation - runRotation: Rotation - depth: int
+ startAI(BigInteger, int, boolean) - alphabeta(BigInteger, int, int, boolean, depth): void + getRotation(): Rotation + getInsertPos(): int

### Tree

Die Klasse Tree dient der Alpha-Beta-Suche. Diese ermöglicht die Berechnung des nächsten Computerzuges.

Es wird nach dem Alpha-Beta-Algorithmus ein Baum aufgebaut, der als Wurzelknoten das aktuelle Spielfeld als BigInteger-Wert besitzt. Ausgehend von diesem, werden die Kindknoten erstellt, von denen jeder einen möglichen Spielverlauf darstellt und das entsprechende Feld als BigInteger-Wert speichert. Angekommen bei der maximalen Tiefe im Baum, wird das bis dahin konfigurierte Feld bewertet und reicht diese Information an sein Elternknoten weiter.

 <b>WinValidator</b>
- winMasks: BigInteger[]
+ getSetOfWinBits(BigInteger, Player): Set<BigInteger>

### WinValidator

WinValidator enthält ein BigInteger-Array mit 32 BigInteger-Werten, die als Bit-Masken fungieren. Jeder einzelne Wert bildet eine der 32 Gewinnmöglichkeiten ab. Die Methode getSetOfWinBits() liefert alle BigInteger-Werte die durch die

AND-Bit-Operation eine Übereinstimmung mit dem Spielfeld von 5 Bits haben.

## Verwendete Dateien

Zur Darstellung der Spielelemente auf der Oberfläche werden Bilder verwendet, die in das Projekt mit eingebunden werden.

Folgende Dateien sind unter dem Pfad "src/gui/images" abgelegt:

Datei	Beschreibung
background.jpg	Hintergrund einer Spielfeldzelle
marble_0.jpg	Spielstein in der Farbe Blau
marble_1.jpg	Spielstein in der Farbe Orange
left0.png	Pfeil mit Drehrichtung links nach unten gerichtet
left1.png	Pfeil mit Drehrichtung links nach links gerichtet
left2.png	Pfeil mit Drehrichtung links nach rechts gerichtet
left3.png	Pfeil mit Drehrichtung links nach oben gerichtet
right0.png	Pfeil mit Drehrichtung rechts nach rechts gerichtet
right1.png	Pfeil mit Drehrichtung rechts nach unten gerichtet
right2.png	Pfeil mit Drehrichtung rechts nach oben gerichtet
right3.png	Pfeil mit Drehrichtung rechts nach links gerichtet

## 5 Programmtests

Testfall	Ergebnis
<i>Das Fenster wird vergrößert.</i>	Das Spielfeld und die liegenden Steine vergrößern sich nur, wenn das Fenster quadratisch vergrößert wird. Bei der Vergrößerung bleibt das Spielfeld immer quadratisch.
<i>Der Spieler ist am Zug. Er klickt auf eine besetzte Zelle. Anschließend klickt er auf eine leere Zelle des Spielfeldes. Danach erfolgt ein erneutes Klicken auf eine leere Zelle.</i>	Beim Klick auf eine besetzte Zelle bleibt das Feld unverändert. Der Zug wird nicht im Protokoll aufgeführt. Durch die Auswahl einer leeren Zelle, wird auf die gewünschte Position der Spielstein des Spielers platziert. Der Zug wird im Protokoll textuell dargestellt.
<i>Ein Spieler legt Stein. Eine gewinnbringende Reihe entsteht.</i>	Der Gewinn des Spielers wird erkannt und die gewinnbringenden Steine werden markiert. Der Sieger wird auf der Oberfläche ausgegeben. Nach Bestätigung des Informationsfensters, startet ein neues Spiel mit einem leeren Spielfeld. Es gibt keine Spielhistorie, sodass Spielzüge weder zurückgenommen noch wiederholt werden können.
<i>Der Spieler rotiert einen Quadranten und platziert anschließend einen Spielstein.</i>	Die Rotation des Quadranten wird nicht ausgeführt. Das Spielfeld und das Protokoll bleiben zunächst unverändert. Anschließend wird der gesetzte Spielstein hinzugefügt und im Protokoll notiert.
<i>Der Spieler platziert einen Stein im oberen linken Quadranten und dreht diesen anschließend nach rechts. Während der Animation der Drehung erfolgt ein Klick auf eine andere Zelle des Spielfeldes.</i>	Der durchgeführte Zug wird korrekt durchgeführt. Während der Animation, sind die Button ‚Zurücknehmen‘ sowie ‚Wiederholen‘ deaktiviert. Klicks zeigen keine Wirkung auf das Spielfeld.

## Fortsetzung Programmtests

### Testfall

### Ergebnis

*Der Spieler platziert seinen Spielstein auf einem leeren Feld. Anschließend möchte er diesen Zug unverzüglich zurücknehmen und klickt auf ‚Rückgängig‘.*

Nach der Platzierung des Spielsteins auf dem Spielfeld, ist der Button "Zurücknehmen" deaktiviert. Somit ist es dem Spieler nicht möglich diesen Zug zurückzunehmen, bis der Computerzug durchgeführt wurde.

*Nachdem ein Zug des Computers durchgeführt wurde, wird auf "Rückgängig" geklickt. Anschließend wird auf "Wiederholen" geklickt.*

Nachdem die letzten beiden Spielzüge zurückgenommen werden, erscheint das Spielfeld im selben Zustand wie vor der letzten Platzierung des Spielers. Der jeweils letzte Zug beider Spieler wurde korrekt zurückgenommen und aus dem Protokoll vorläufig entfernt. Nach der Auswahl die Züge zu wiederholen, erscheint das Spielfeld wieder in demselben Zustand, bevor die Züge zurückgenommen wurden. Die wiederholten Spielzüge werden im Protokoll wieder angezeigt.

*Nach einem ausgeführten Computerzug klickt der Spieler auf ‚Rückgängig‘. Nachdem die Strategie von offensiv auf defensiv umgestellt wird, klickt der Spieler auf Wiederholen.*

Die letzten beiden Züge werden korrekt zurückgenommen. Nachdem die Strategie des Computers verändert wird, können die zurückgenommenen Züge nicht wiederholt werden und werden aus dem Protokoll entfernt.

### *Spielstand speichern*

*Der Spieler platziert einen Stein und wählt im Menü die Option ‚Spielstand speichern‘.*

Das Menü-Item ist deaktiviert und kann nicht ausgewählt werden.

*Der Spieler platziert einen Stein und dreht einen Quadranten. Bevor der Computerzug zu Ende durchgeführt wurde, wird versucht den Spielstand zu speichern.*

Das Menü-Item ist deaktiviert und kann nicht ausgewählt werden.

## Fortsetzung Programmtests

Testfall	Ergebnis
<i>Nachdem der Computerzug durchgeführt wurde, wird versucht den Spielstand zu speichern.</i>	Das Menü-Item "Spielstand speichern" ist aktiviert. Nach dem Klick erscheint ein Datei-Explorer. Der Benutzer kann den Speicherort und einen Dateinamen auswählen. Als Dateiendung ist '.pen' voreingestellt.
<i>Spielstand laden</i>	
<i>Der Spieler ist am Zug und wählt im Menü "Spielstand laden".</i>	Es erscheint ein Datei-Explorer und der Benutzer kann eine Datei auswählen.
<i>Der Spieler lädt eine gültige Spielstanddatei.</i>	Das aktuelle Spielfeld wird ohne Nachfrage verworfen.  Die Spielstanddatei konnte erfolgreich geladen werden und wird auf dem Spielfeld abgebildet. Das Protokoll ist leer.
<i>Der Spieler lädt eine ungültige Spielstanddatei.</i>	Eine erscheinende Fehlermeldung informiert den Benutzer über ein falsches Format der Datei.  Nach der Bestätigung dieser Information wird das bisherige Spiel wieder unverändert aufgenommen.
<i>Defensive Strategie</i>	
<i>Der Spieler wählt im Menü die defensive Strategie aus. Nachdem er seinen ersten Spielstein platziert hat und der Computer ebenfalls seinen Zug durchgeführt hat, platziert der Spieler seinen nächsten Stein horizontal neben seinen ersten Stein. Anschließend dreht er einen Quadranten.</i>	Der Computer platziert seinen Spielstein horizontal neben den beiden Spielersteinen und blockiert somit eine Dreierreihe.