

„Mensch ärgere Dich nicht“

## PROGRAMMENTWURF

in der Vorlesung „Advanced Software Engineering“

im fünften und sechsten Semester

des Studienganges Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Nadine Weiß und Manuel Berg**

31.05.2022

Matrikelnummern

3196898 (Weiß), 5931590 (Berg)

Kurs

TINF19B5

Dozent

Maurice Müller

## **Eidesstattliche Erklärung**

(gemäß §5(3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29.09.2017)

Wir versichern hiermit, dass wir unsere Programmentwurf vom 31.05.2022 mit dem Thema „Mensch ärgere Dich nicht“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

---

Ort, Datum

Unterschrift

# Inhaltsverzeichnis

Abbildungsverzeichnis	VI
Tabellenverzeichnis	VII
<b>1 Einführung (4P)</b>	<b>1</b>
1.1 Übersicht über die Applikation (1P) . . . . .	1
1.2 Wie startet man die Applikation? (1P) . . . . .	1
1.3 Technischer Überblick (2P) . . . . .	2
<b>2 Clean Architecture (8P)</b>	<b>3</b>
2.1 Was ist Clean Architecture? (1P) . . . . .	3
2.2 Analyse der Dependency Rule (2P) . . . . .	3
2.3 Analyse der Schichten (5P) . . . . .	6
<b>3 SOLID (8P)</b>	<b>8</b>
3.1 Analyse SRP (3P) . . . . .	8
3.2 Analyse OCP (3P) . . . . .	10
3.3 Analyse LSP/ISP/DIP (2P) . . . . .	13
<b>4 Weitere Prinzipien (8P)</b>	<b>15</b>
4.1 Analyse GRASP: Geringe Kopplung (4P) . . . . .	15
4.2 Analyse GRASP: Hohe Kohäsion (2P) . . . . .	18
4.3 DRY (2P) . . . . .	20
<b>5 Unit Tests (8P)</b>	<b>22</b>
5.1 10 Unit Tests (2P) . . . . .	22
5.2 ATRIP: Automatic (1P) . . . . .	22
5.3 ATRIP: Thorough (1P) . . . . .	23

5.4	ATRIP: Professional (1P)	24
5.5	Fakes und Mocks (1P)	25
<b>6</b>	<b>Domain Driven Design (8P)</b>	<b>27</b>
6.1	Ubiquitous Language (2P)	27
6.2	Repositories (1,5P)	28
6.3	Aggregates (1,5P)	29
6.4	Entities (1,5P)	30
6.5	Value Objects (1,5P)	31
<b>7</b>	<b>Refactoring (8P)</b>	<b>32</b>
7.1	Code Smells (2P)	32
7.2	2 Refactorings (6P)	37
<b>8</b>	<b>Entwurfsmuster (8P)</b>	<b>39</b>
<b>9</b>	<b>Anhang: Bedienungsanleitung</b>	<b>41</b>

# Abbildungsverzeichnis

2.1	Positiv-Beispiel Dependency Rule [Eigene Darstellung aus <i>IntelliJ</i> ] . . . .	4
2.2	Negativ-Beispiel Dependency Rule [Eigene Darstellung aus <i>IntelliJ</i> ] . . .	5
2.3	Klasse der Domain Code-Schicht [Eigene Darstellung aus <i>IntelliJ</i> ] . . . .	6
2.4	Klasse der Application Code-Schicht [Eigene Darstellung aus <i>IntelliJ</i> ] . .	7
3.1	Positiv-Beispiel SRP [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	8
3.2	Negativ-Beispiel SRP [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	9
3.3	Positiv-Beispiel OCP [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	10
3.4	Negativ-Beispiel OCP [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	12
3.5	Positiv-Beispiel DIP [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	13
3.6	Negativ-Beispiel DIP [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	14
4.1	Positiv-Beispiel GRASP [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	16
4.2	Negativ-Beispiel GRASP [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	17
4.3	GRASP: Hohe Kohäsion [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	18
4.4	DRY [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	20
5.1	Positiv-Beispiel Professional [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	24
5.2	Negativ-Beispiel Professional [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	25
5.3	BoardFourMock [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	25
5.4	GameFrameMock [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	26
6.1	Repository [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	28
6.2	Aggregate [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	29
6.3	Entitiy Player [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	30
6.4	Enum als Value Object [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	31
7.1	Eliminierung der Long Method [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	32
7.2	Long Method [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	33

7.3	Switch-Statements [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	34
7.4	Erweiterung um fünf Instanzvariablen [Eigene Darstellung aus <i>IntelliJ</i> ] .	35
7.5	Hinzufügen der vier Funktionalitäten [Eigene Darstellung aus <i>IntelliJ</i> ] . .	36
7.6	Extract Method (vorher) [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	37
7.7	Extract Method (nacher) [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	37
7.8	Rename Method [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	38
8.1	Graph als Singleton-Enum [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	39
8.2	Observer-Pattern [Eigene Darstellung aus <i>IntelliJ</i> ] . . . . .	40
9.1	Auswahl der Spieleranzahl . . . . .	41
9.2	Auswahl der Anzahl an simulierten Spielern . . . . .	41
9.3	Benennung eines Spielers und Auswahl eines existierenden Spielers . . . .	42
9.4	Hauptfenster der Anwendung mit Spielbrett . . . . .	44

# Tabellenverzeichnis

5.1	10 Unit Tests . . . . .	22
6.1	4 Beispiele für die Ubiquitous Language . . . . .	27

# 1. Einführung (4P)

## 1.1 Übersicht über die Applikation (1P)

*[Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?]*

Die Applikation dient rein der Unterhaltung und simuliert das bekannte deutsche Gesellschaftsspiel *Mensch ärgere Dich nicht* mit dem klassischen Spielbrett für vier Spieler. Hierbei kann – entsprechend dem originalen Brettspiel – zwischen zwei, drei oder vier Teilnehmenden gewählt werden. Eine Besonderheit der Applikation besteht darin, dass nicht alle Teilnehmenden „menschlich“ sein müssen, sondern nach Belieben auch ein Algorithmus das Würfeln und die Ausführung der Züge übernehmen kann. Weitere Details bezüglich der Funktionsweise der Applikation sind im Kapitel 9 nachzulesen.

## 1.2 Wie startet man die Applikation? (1P)

*[Wie startet man die Applikation? Welche Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]*

Eine Anleitung befindet sich im Kapitel 9. Auf der Maschine, die zum Ausführen der Anwendung genutzt werden soll, muss lediglich Java 11 installiert sein. Die **.jar**-Datei, die im entsprechenden GitHub-Repository unter [https://github.com/NadineWeiss/project\\_swe/blob/master/sweProject.jar](https://github.com/NadineWeiss/project_swe/blob/master/sweProject.jar) zu finden ist, kann dann ausgeführt werden.



## 1.3 Technischer Überblick (2P)

*[Nennung und Erläuterung der Technologien (z.B. Java, MySQL, ...), jeweils Begründung für den Einsatz der Technologien]*

Als Programmiersprache wurde Java verwendet, da dies in den Anforderungen bereits definiert wurde und wir mit dieser Sprache bisher die meisten Erfahrungen gesammelt haben. Auch in der Vorlesung „Programmieren“ im ersten und zweiten Semester des Studienganges wurden die Grundlagen objektorientierter Programmiersprachen anhand von Java erläutert, weshalb sich dies auch für den Programmentwurf angeboten hat. Bezüglich der GUI ist die Entscheidung auf das Framework Swing gefallen. Auch hier haben wir in der genannten Vorlesung und auch privat bereits Erfahrungen sammeln können. Da der Fokus des Programmentwurfs nicht auf der GUI liegen sollte, haben wir hier die für uns einfachste Variante ausgewählt. Die selben Argumente haben uns auch zu der Entscheidung für das Abhängigkeitsmanagement mit Maven geführt.

## 2. Clean Architecture (8P)

### 2.1 Was ist Clean Architecture? (1P)

*[Allgemeine Beschreibung der Clean Architecture in eigenen Worten]*

Unter der *Clean Architecture* versteht man eine Architekturrichtlinie. Die Grundidee kann man sich als eine Zwiebel vorstellen, welche aus fünf Schichten besteht. Die fünf Schichten Plugins, Adapters, Application Code, Domain Code sowie Abstraction Code stellen hierbei die Bereiche der Software dar. So bildet der Abstraction Code den innersten Kern, und die Plugins bilden die äußerste Hülle. Je weiter ein Ring vom Kern entfernt ist, desto „sichtbarer“ ist er in der tatsächlichen Anwendung und desto mehr Änderungen erfährt er.

Ein zentraler Aspekt der *Clean Architecture* ist hierbei die sogenannte *Dependency Rule*. Diese besagt, dass Abhängigkeiten in dem zuvor erwähnten Modell nur nach Innen – also in Richtung des Kerns (Abstraction Code) – zeigen dürfen. So darf die Adapters-Schicht beispielsweise von der Schicht des Application Codes abhängig sein, jedoch nicht von den Plugins.

### 2.2 Analyse der Dependency Rule (2P)

*[1 Klasse, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML der Klasse und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]*

#### **Positiv-Beispiel: Dependency Rule**

Die Klasse *Graph* befindet sich in der Domain Code-Schicht und realisiert die Knoten und Kanten des Spielfeldes. Da dies nur die Struktur des Spielfeldes enthält und in keinsten Weise mit Spielständen zu tun hat, sind keine Abhängigkeiten in Richtung äußerer Schichten enthalten.

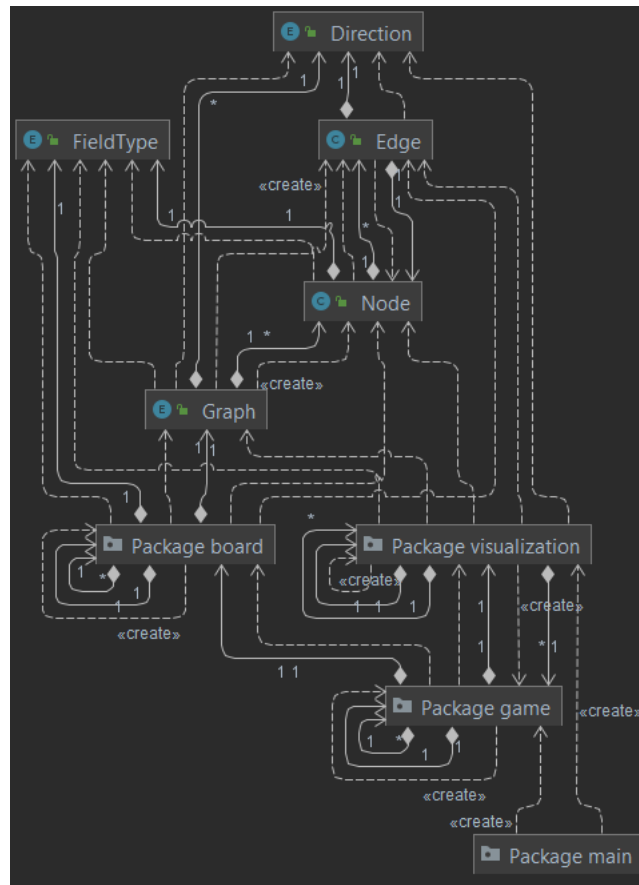


Abbildung 2.1: Positiv-Beispiel Dependency Rule [Eigene Darstellung aus *IntelliJ*]

In der Abbildung 2.1 sind alle sichtbaren Klassen in derselben Schicht wie die *Graph*-Klasse. Die in den Packages enthaltenen Klassen wiederum befinden sich in anderen Schichten. Da es von der *Graph*-Klasse keine Abhängigkeit auf eines der anderen Packages (und somit auch nicht auf andere Schichten) gibt, ist dies ein Positivbeispiel für die Einhaltung der Dependency Rule.

## Negativ-Beispiel: Dependency Rule

Bei der Klasse *GameService* handelt es sich um Application Code, denn die Klasse kümmert sich um einen reibungslosen Spielablauf. Deshalb sollten hier auch grundsätzlich keine Abhängigkeiten auf die GUI vorhanden sein.

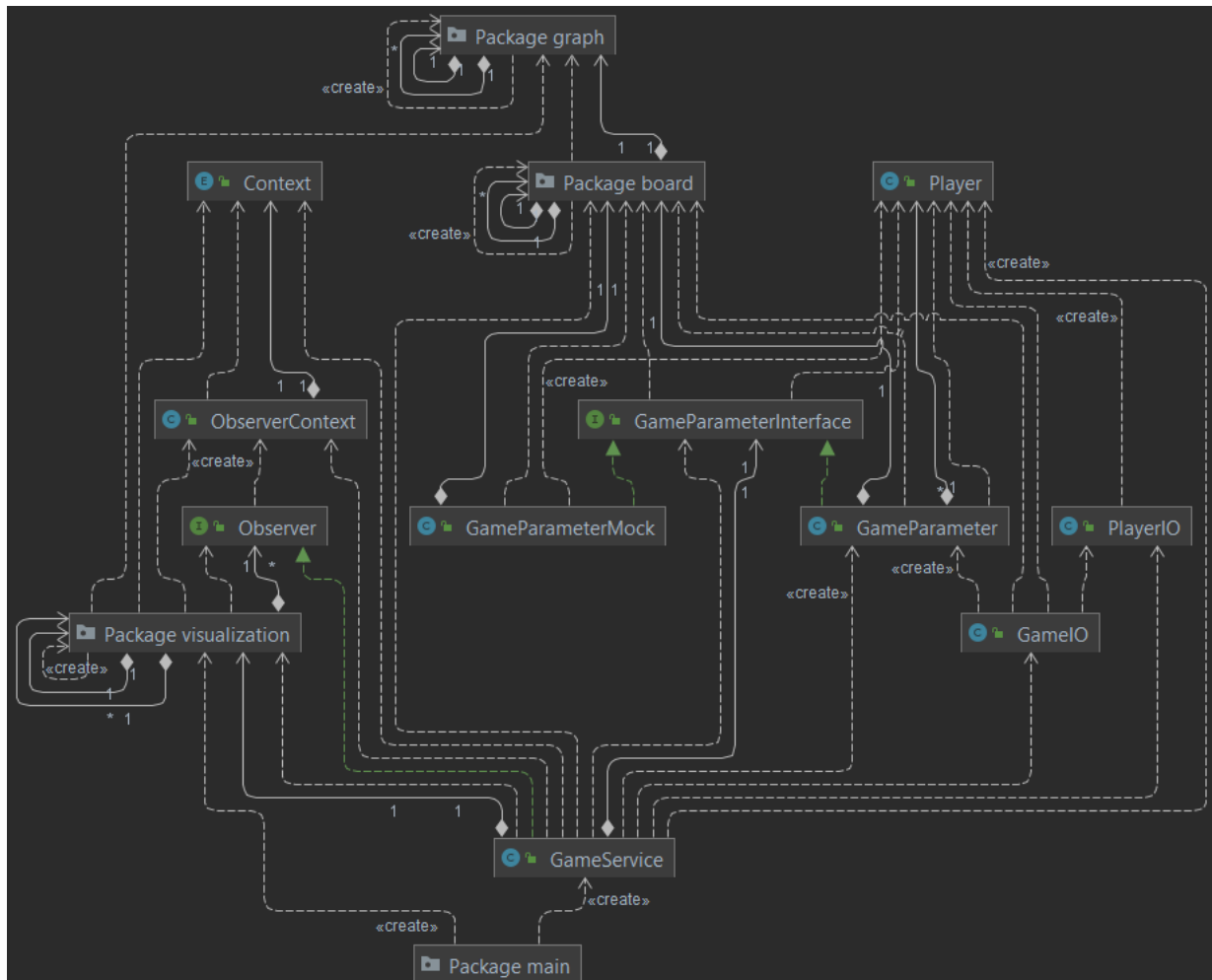


Abbildung 2.2: Negativ-Beispiel Dependency Rule [Eigene Darstellung aus *IntelliJ*]

In Abbildung 2.2 ist zu erkennen, dass die *GameService*-Klasse Abhängigkeiten in Richtung der Klassen des Packages *visualization* aufweist. Die Klassen dieses Packages sind verantwortlich für die GUI und gehören somit zur äußersten Schicht. Aus diesem Grund liegt hier ein Negativ-Beispiel vor.

## 2.3 Analyse der Schichten (5P)

[jeweils 1 Klasse zu 2 unterschiedlichen Schichten der Clean-Architecture: jeweils UML der Klasse (ggf. auch zusammenspielenden Klassen), Beschreibung der Aufgabe, Einordnung mit Begründung in die Clean-Architecture]

### Schicht: Domain Code

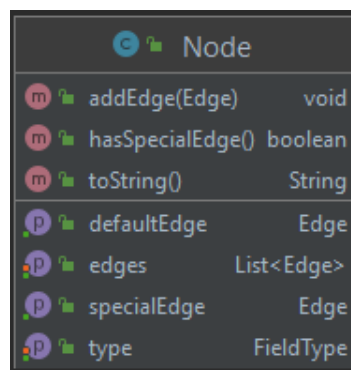


Abbildung 2.3: Klasse der Domain Code-Schicht [Eigene Darstellung aus *IntelliJ*]

Die Klasse *Node* entspricht einem Spielfeld bei *Mensch ärgere Dich nicht*. Jedes Feld, welches von einer Spielfigur besetzt werden kann, wird durch diese Klasse einem Feldtypen zugeordnet. Dies kann beispielsweise ein rotes Zielfeld oder auch ein grünes Startfeld sein. Außerdem enthält die Klasse eine Liste von Kanten, die ausgehend von diesem Knoten eine Verbindung zu einem anderen Knoten darstellen. In obiger Abbildung ist auch zu sehen, dass es Standardkanten und Spezialkanten gibt. Erstere können durch jede Spielfigur unabhängig von der Farbe befahren werden. Die Spezialkanten wiederum sind nur durch eine spezielle Farbe nutzbar. Das sind die vier Kanten, die zu den Zielfeldern zeigen.

Die Klasse *Node* gehört zum inneren Kern der Anwendung und ist unabhängig vom Spielbetrieb. Die grundlegende Logik hinter dem Spielbrett mit den genannten Knoten und Kanten ändert sich nie. Gemäß den Vorlesungsinhalten handelt es sich deshalb um ein Teil der Domain Code-Schicht.

## Schicht: Application Code

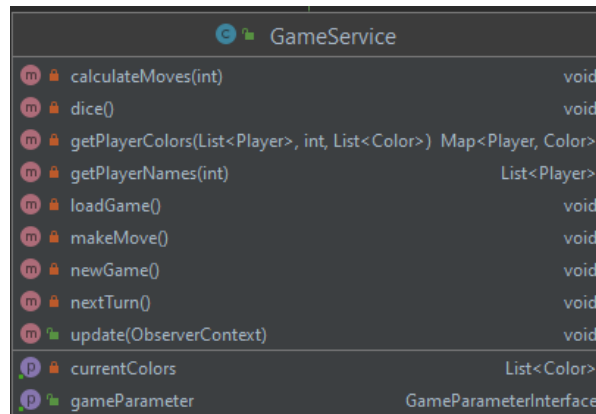


Abbildung 2.4: Klasse der Application Code-Schicht [Eigene Darstellung aus *IntelliJ*]

Die Klasse *GameService* kümmert sich um das Spielgeschehen. Dabei weiß sie um die aktuell am Spiel teilnehmenden Farben. Die Klasse kann ausgeben, wo sich eine Spielfigur nach einer gegebenen Augenzahl durch Würfeln befindet. Außerdem führt sie den Prozess des Würfeln durch Generierung von Zufallszahlen aus. Auch kann sie einen Zug durchführen, ein neues Spiel starten und bestimmen, welcher Spieler gerade an der Reihe ist. Die Klasse ist an der Implementierung des Observer-Patterns beteiligt. Sie reagiert dadurch auf Änderungen in der GUI.

Die im vorherigen Abschnitt genannten Funktionen entsprechen anwendungsspezifischer Geschäftslogik. Deswegen wird dieser Klasse der Application Code-Schicht zugeordnet.

## 3. SOLID (8P)

### 3.1 Analyse SRP (3P)

*[Jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML der Klasse und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]*

#### Positiv-Beispiel

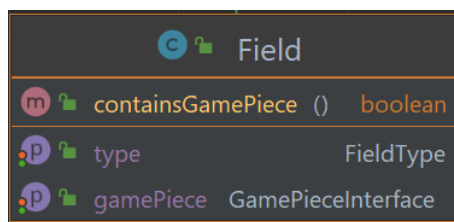


Abbildung 3.1: Positiv-Beispiel SRP [Eigene Darstellung aus *IntelliJ*]

Die Klasse *Field* stellt als einzige Aufgabe ein Spielfeld dar. Dieses kann einem Typen zugeordnet sein, und optional eine Spielfigur enthalten. Durch einen Methodenaufruf wird zurückgegeben, ob sich eine Spielfigur auf dem entsprechenden Feld befindet. Das Single Responsibility Principle wird durch die hier dargestellte Klasse nicht verletzt.

## Negativ-Beispiel

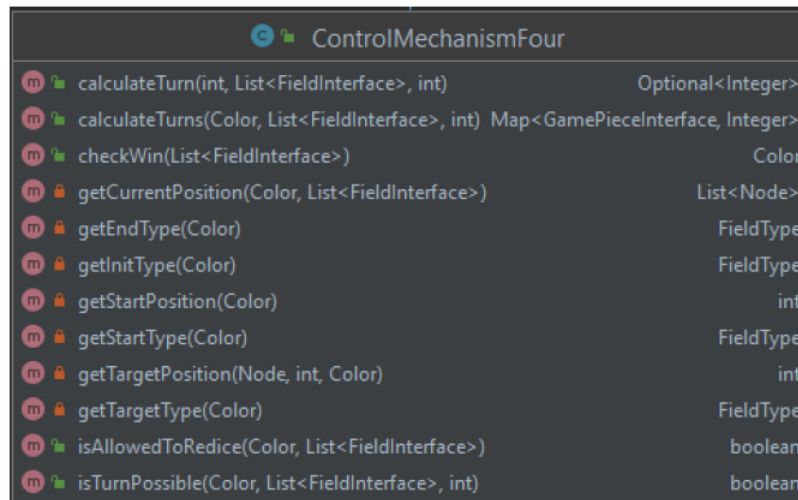


Abbildung 3.2: Negativ-Beispiel SRP [Eigene Darstellung aus *IntelliJ*]

Die Klasse *ControlMechanismFour* ist dafür verantwortlich,

- mögliche Züge zu berechnen,
- zu prüfen, ob ein Spieler bereits gewonnen hat,
- zu prüfen, ob ein Spieler nochmals würfeln darf und
- zu prüfen, ob ein Spieler überhaupt einen Zug durchführen kann.

Es wird deutlich, dass die Klasse für mehrere Tätigkeiten verantwortlich ist und somit das Single Responsibility Principle verletzt wird. Um das Prinzip einhalten zu können, sollte der Umfang der Klasse auf die Erfüllung einer einzigen Aufgabe reduziert werden. Hierzu könnte *ControlMechanismFour* in zwei neue Klassen aufgeteilt werden. Dabei wäre eine Klasse für die Berechnung der Züge verantwortlich; die andere prüft das weitere mögliche Vorgehen eines Spielers. Außerdem enthält *ControlMechanismFour* sieben private Methoden, welche in eine Hilfsklasse ausgelagert werden sollten. Diese privaten Methoden sind in Abbildung 3.2 mit roten Schloss-Symbolen gekennzeichnet. Eine Auslagerung ist sinnvoll, weil die Methoden nicht nur in *ControlMechanismFour* Verwendung finden, sondern auch in anderen Klassen (wie beispielsweise in der *Graph*-Klasse) aufgerufen werden. Hierdurch wird redundanter Quellcode eingespart. Der Code war bis zum Commit **7ac8c518c06d50472cb3a9a94ea5a60faea3f09e** vom 26.05.2022 wie hier dargestellt im Repository vorhanden. Danach wurde er refactored.



## 3.2 Analyse OCP (3P)

[Jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML der Klasse und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

### Positiv-Beispiel

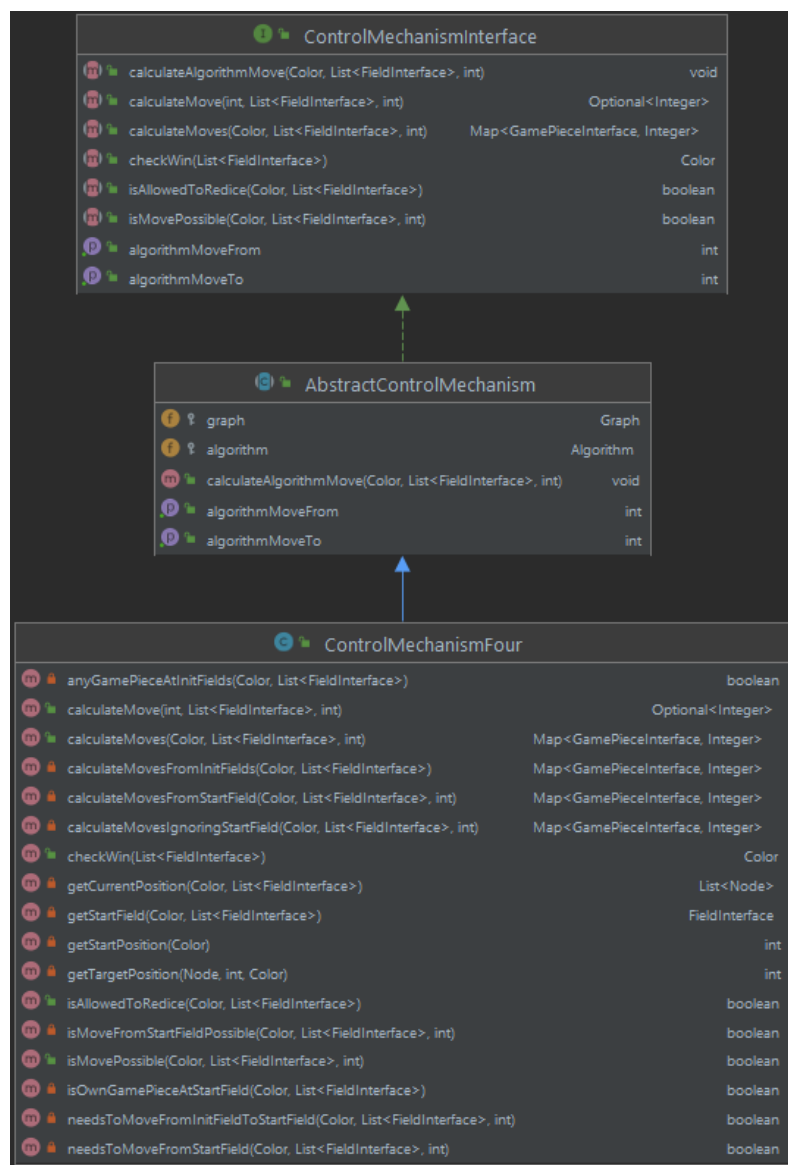
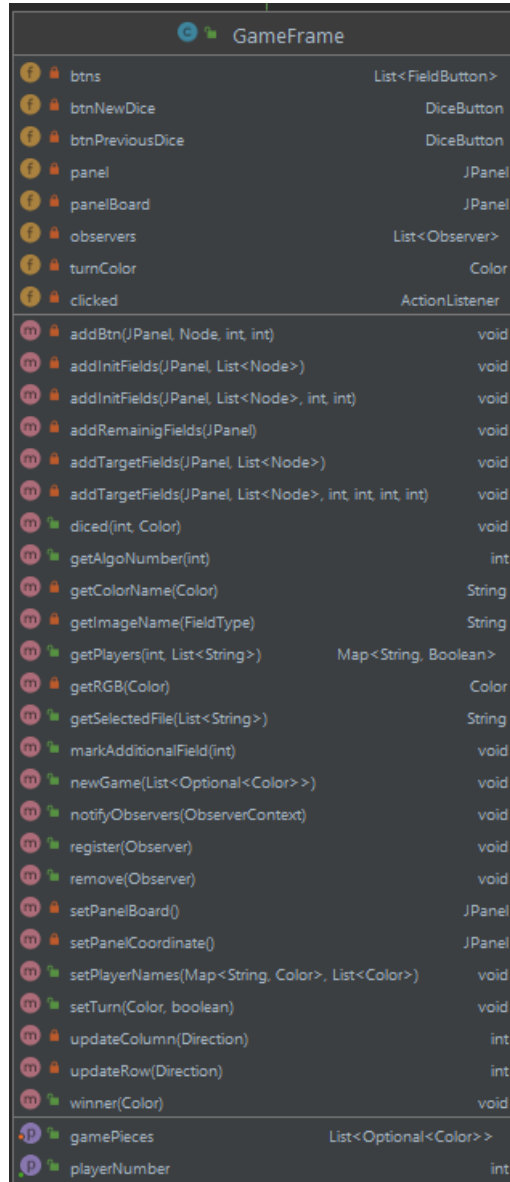


Abbildung 3.3: Positiv-Beispiel OCP [Eigene Darstellung aus *IntelliJ*]

Der *ControlMechanism* ist offen für Erweiterungen bezüglich der Spieleranzahl. Dies wurde so entwickelt, da eine Abweichung von der Spielerzahl vier (z.B. Erhöhung auf sechs Spieler) eine optionale Anforderung des Projektes ist. Um beispielsweise einen *ControlMechanism* für sechs Spieler umzusetzen, muss analog zur Klasse *ControlMechanismFour* eine Klasse *ControlMechanismSix* implementiert werden. Diese würde dann auch von *AbstractControlMechanism* erben.

Die Methoden werden immer über das *ControlMechanismInterface* aufgerufen. Das bedeutet, ohne Änderungen an diesem kann das Verhalten wie oben dargestellt erweitert werden. Die OCP-Regel der verschlossenen Module gegenüber Modifikationen ist hierdurch erfüllt.

## Negativ-Beispiel



GameFrame		
🔒	btns	List<FieldButton>
🔒	btnNewDice	DiceButton
🔒	btnPreviousDice	DiceButton
🔒	panel	JPanel
🔒	panelBoard	JPanel
🔒	observers	List<Observer>
🔒	turnColor	Color
🔒	clicked	ActionListener
🔒	addBtn(JPanel, Node, int, int)	void
🔒	addInitFields(JPanel, List<Node>)	void
🔒	addInitFields(JPanel, List<Node>, int, int)	void
🔒	addRemainingFields(JPanel)	void
🔒	addTargetFields(JPanel, List<Node>)	void
🔒	addTargetFields(JPanel, List<Node>, int, int, int, int)	void
🔒	diced(int, Color)	void
🔒	getAlgoNumber(int)	int
🔒	getColorName(Color)	String
🔒	getImageName(FieldType)	String
🔒	getPlayers(int, List<String>)	Map<String, Boolean>
🔒	getRGB(Color)	Color
🔒	getSelectedFile(List<String>)	String
🔒	markAdditionalField(int)	void
🔒	newGame(List<Optional<Color>>)	void
🔒	notifyObservers(ObserverContext)	void
🔒	register(Observer)	void
🔒	remove(Observer)	void
🔒	setPanelBoard()	JPanel
🔒	setPanelCoordinate()	JPanel
🔒	setPlayerNames(Map<String, Color>, List<Color>)	void
🔒	setTurn(Color, boolean)	void
🔒	updateColumn(Direction)	int
🔒	updateRow(Direction)	int
🔒	winner(Color)	void
🔒	gamePieces	List<Optional<Color>>
🔒	playerNumber	int

Abbildung 3.4: Negativ-Beispiel OCP [Eigene Darstellung aus *IntelliJ*]

Dies ist ein Negativ-Beispiel für OCP, weil hier gegensätzlich zum *ControlMechanism* keine Erweiterung der Spieleranzahl möglich ist.

### 3.3 Analyse LSP/ISP/DIP (2P)

[Jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP); jeweils UML der Klasse und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird. Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

#### Positiv-Beispiel DIP

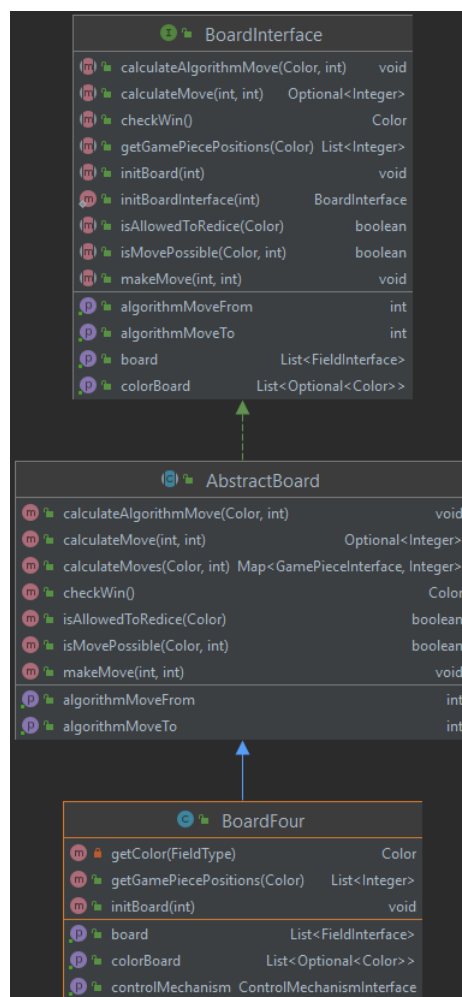
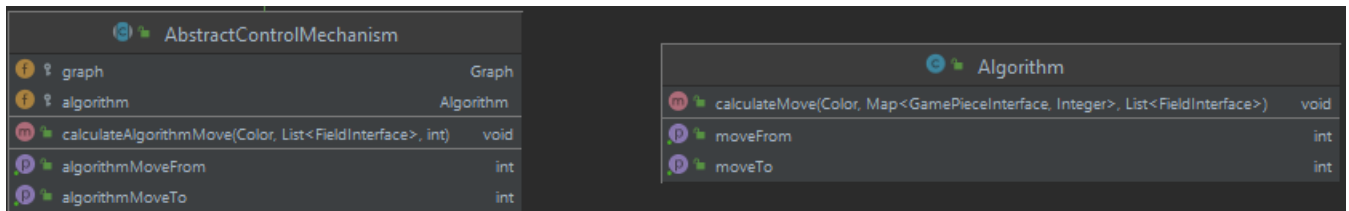


Abbildung 3.5: Positiv-Beispiel DIP [Eigene Darstellung aus *IntelliJ*]

Durch die Abstraktion des Spielbretts wird das Dependency Inversion Principle erfüllt, denn hier hängen die Details von Abstraktionen ab. Es ist zu erkennen, dass die Details – also die konkreten Implementierungen – in der Klasse *BoardFour* zu finden sind.

Der Methodenaufruf beim Spielbrett ändert sich demnach nicht, wenn die konkrete Implementierung verändert wird. Beispielsweise könnte auch ein weiteres Spielbrett umgesetzt werden, ohne dass sich am restlichen Programm etwas ändert.

### Negativ-Beispiel DIP



AbstractControlMechanism	
graph	Graph
algorithm	Algorithm
calculateAlgorithmMove(Color, List<FieldInterface>, int)	void
algorithmMoveFrom	int
algorithmMoveTo	int

Algorithm	
calculateMove(Color, Map<GamePieceInterface, Integer>, List<FieldInterface>)	void
moveFrom	int
moveTo	int

Abbildung 3.6: Negativ-Beispiel DIP [Eigene Darstellung aus *IntelliJ*]

Der *ControlMechanism* steht in direkter Abhängigkeit der Klasse *Algorithm*. Falls sich die Implementierung letzterer ändert, so muss auch *AbstractControlMechanism* entsprechend angepasst werden. In dieser Zusammenstellung kann nur der vorhandene Algorithmus angewandt werden. Um dies zu umgehen und weitere Algorithmen verwenden zu können, wäre ein Interface sinnvoll.

## 4. Weitere Prinzipien (8P)

### 4.1 Analyse GRASP: Geringe Kopplung (4P)

*[jeweils eine bis jetzt noch nicht behandelte Klasse als positives und negatives Beispiel geringer Kopplung; jeweils UML Diagramm mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung warum hier eine geringe Kopplung vorliegt bzw. Beschreibung, wie die Kopplung aufgelöst werden kann]*

#### **Positiv-Beispiel**

Eine geringe Kopplung wurde durch den Einsatz des Observer-Patterns zwischen der für die Visualisierung zuständigen „GameFrame“-Klasse und der für den Spielablauf zuständigen „GameService“-Klasse erreicht. Das heißt, wenn in der UI durch den Benutzer eine Aktion ausgeführt wird, wird die „GameService“-Klasse benachrichtigt. Bei der Benachrichtigung weiß das benachrichtigende Objekt nichts Näheres über das zu benachrichtigende Objekt. Anders ausgedrückt, das Observable kennt nur die Observer-Schnittstelle. Dadurch wurde hier eine lose Kopplung geschaffen.

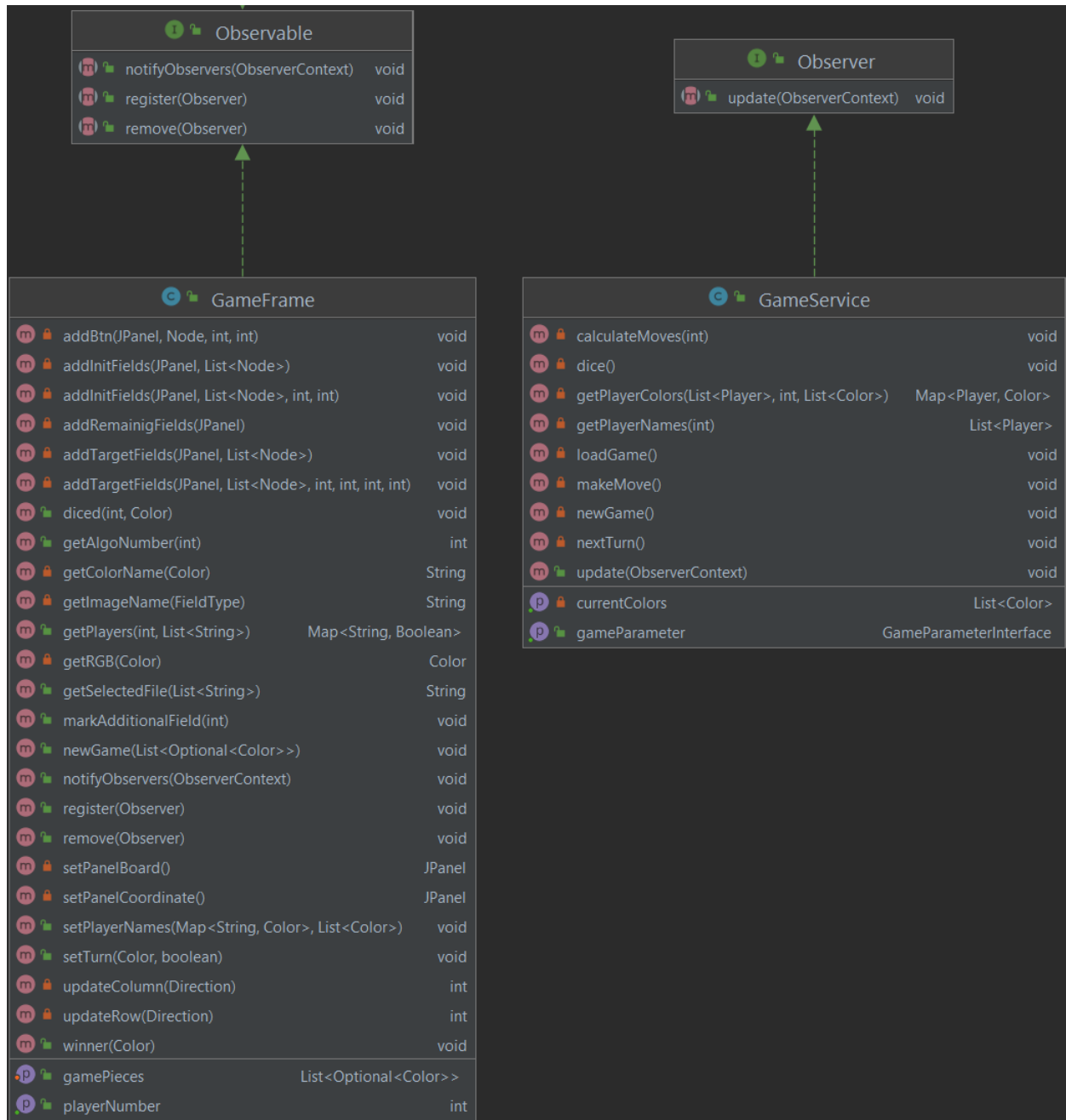


Abbildung 4.1: Positiv-Beispiel GRASP [Eigene Darstellung aus *IntelliJ*]

## Negativ-Beispiel

Eine starke Kopplung liegt zwischen der „Algorithm“- Klasse und der „AbstractControl-Mechanism“-Klasse vor. Erstere berechnet einen Zug, der nicht durch einen Benutzer ausgeführt wird – wenn also ein Benutzer gegen den Algorithmus spielt. Die „AbstractControl-Mechanism“-Klasse ruft die Zugberechnung auf. Die Klasse ist abstrakt, da der Aufruf für ein 4-Personen- oder ein 6-Personen-Spielfeld gleichermaßen gilt.

Eine starke Kopplung liegt vor, da hier auf die konkrete „Algorithm“- Klasse zugegriffen wird. Die Kopplung könnte man durch die Implementierung einer Algorithm-Schnittstelle lockern. Dadurch könnten auch verschiedene Algorithmen implementiert werden.

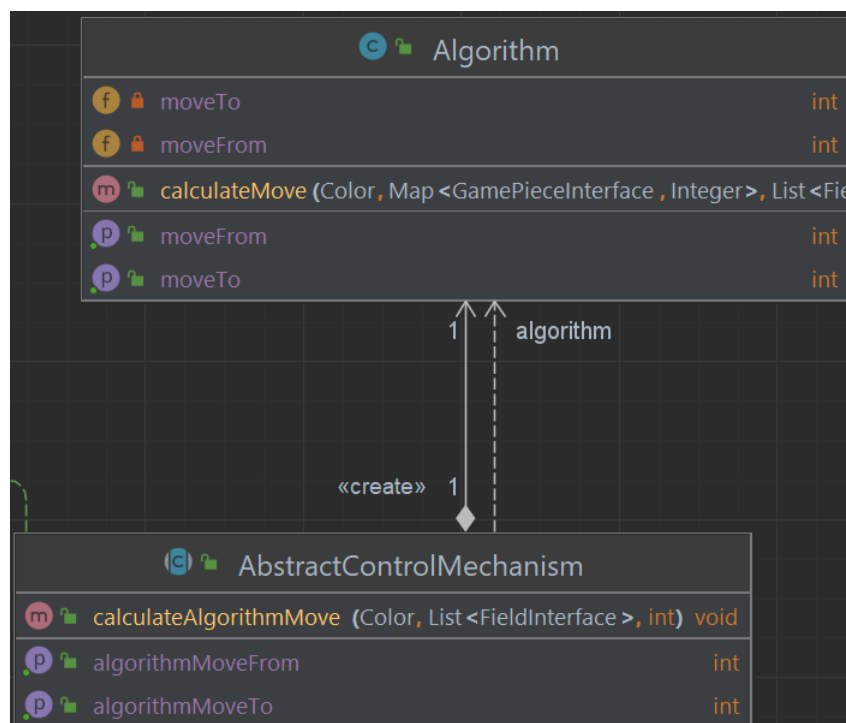


Abbildung 4.2: Negativ-Beispiel GRASP [Eigene Darstellung aus *IntelliJ*]



## 4.2 Analyse GRASP: Hohe Kohäsion (2P)

*[eine Klasse als positives Beispiel hoher Kohäsion; UML Diagramm und Begründung, warum die Kohäsion hoch ist]*

Die Kohäsion erhöht sich, je mehr Verantwortlichkeiten und Teilaufgaben in andere Klassen ausgelagert sind. Ein Beispiel für hohe Kohäsion wäre hier der dem Spielfeld unterliegende Graph.

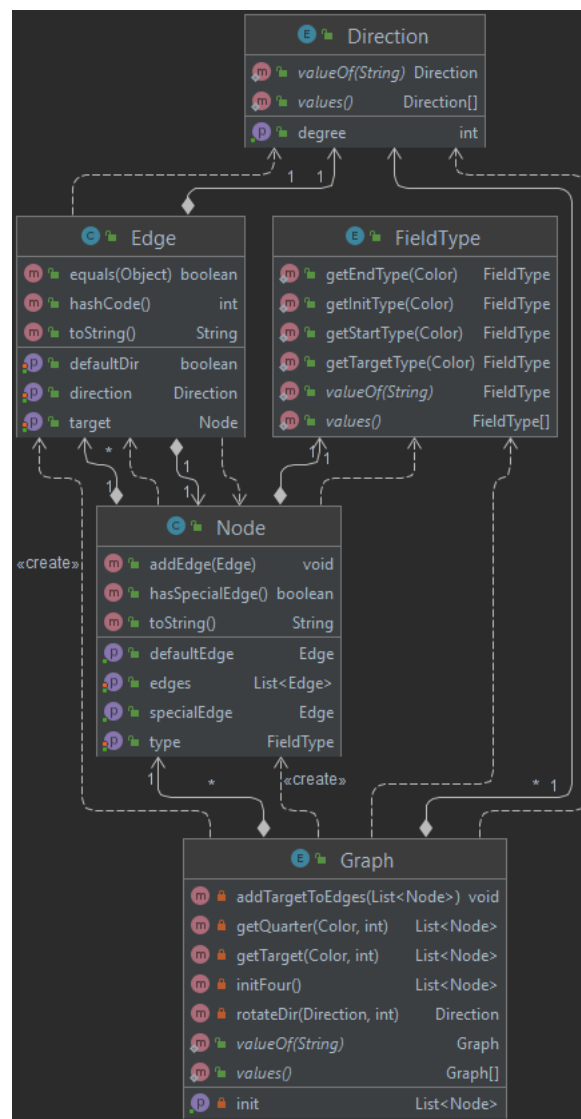


Abbildung 4.3: GRASP: Hohe Kohäsion [Eigene Darstellung aus *IntelliJ*]

In der „Graph“-Klasse wird der Graph aufgebaut. Hierzu wurden aber nicht alle Verantwortlichkeiten in dieser Klasse belassen, sondern in vier weitere Klassen ausgelagert.

- Die „Node“-Klasse stellt einen Knoten dar und enthält den Feldtyp (Siehe „FieldType“) dieses Knotens und die ausgehenden Kanten (Siehe „Edge“)
- Das „FieldType“-Enum gibt den Feldtyp an. Das kann ein neutrales Feld sein oder zum Beispiel ein rotes Startfeld oder ein gelbes Zielfeld.
- Die „Edge“-Klasse enthält den Zielknoten, die Richtung (Siehe „Direction“) und die Information, ob es ein Default-Kante ist. Letzteres bedeutet, dass die Kante von allen Spielfiguren befahren werden kann. Dies ist zum Beispiel bei der Kante zu den Zielfeldern der jeweiligen Farben nicht der Fall.
- Das „Direction“-Enum gibt an, in welche Richtung die Kante geht.

## 4.3 DRY (2P)

*[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher); begründen und Auswirkung beschreiben]*

Die Klasse „GraphUtilities“ wurde hinzugefügt, da die jetzt enthaltenen vier Methoden früher von den drei Klassen „Graph“, „AbstractBoard“ und „ControlMechanismFour“ jeweils extra implementiert wurden und sich der Code dadurch dupliziert hat. Das Ergebnis:

```
public class GraphUtilities {

    public static FieldType getInitType(final Color color) {
        if (color.equals(Color.RED)) {
            return FieldType.REDINIT;
        }
        if (color.equals(Color.BLUE)) {
            return FieldType.BLUEINIT;
        }
        if (color.equals(Color.GREEN)) {
            return FieldType.GREENINIT;
        }
        return FieldType.YELLOWINIT;
    }

    public static FieldType getStartType(final Color color) {
        if (color.equals(Color.RED)) {
            return FieldType.REDSTART;
        }
        if (color.equals(Color.BLUE)) {
            return FieldType.BLUESTART;
        }
        if (color.equals(Color.GREEN)) {
            return FieldType.GREENSTART;
        }
        return FieldType.YELLOWSTART;
    }

    public static FieldType getEndType(final Color color) {
        if (color.equals(Color.RED)) {
            return FieldType.REDEND;
        }
        if (color.equals(Color.BLUE)) {
            return FieldType.BLUEEND;
        }
        if (color.equals(Color.GREEN)) {
            return FieldType.GREENEND;
        }
        return FieldType.YELLOWEND;
    }

    public static FieldType getTargetType(final Color color) {
        if (color.equals(Color.RED)) {
            return FieldType.REDTARGET;
        }
        if (color.equals(Color.BLUE)) {
            return FieldType.BLUETARGET;
        }
        if (color.equals(Color.GREEN)) {
            return FieldType.GREENTARGET;
        }
        return FieldType.YELLOWTARGET;
    }
}
```

Abbildung 4.4: DRY [Eigene Darstellung aus *IntelliJ*]

In der „Graph“-Klasse waren früher alle vier Methoden, in der „AbstractBoard“-Klasse war nur die „getInitType“-Methode und in der „ControlMechanismFour“-Klasse waren ebenfalls alle vier Methoden vorhanden. Der Unterschied zum jetzigen Stand besteht darin, dass die Methoden mittlerweile statisch gemacht worden sind. Da sich innerhalb der Methode nichts verändert hat, sei hier der frühere Stand nicht als Codebeispiel aufgezeigt.

Positiv ist, dass sich der Code durch das Eliminieren von Duplikaten reduziert hat. Außerdem ist aus den einzelnen Klassen Code, der nicht zu deren Verantwortlichkeiten gezählt hat, herausgenommen worden. Die Klasse „GraphUtilities“ existierte bis zu dem Commit **75cf8293f43de67a5e087053f9a0e3cc762132a5** vom 26.05.2022.

## 5. Unit Tests (8P)

### 5.1 10 Unit Tests (2P)

*[Nennung von 10 Unit-Tests und Beschreibung, was getestet wird]*

Unit Test	Beschreibung
checkWinTest	Überprüft, ob ein Spieler mit allen Spielfiguren in den Zielfeldern als Gewinner erkannt wird.
calculateMoveTest1	Überprüft, ob bei gegebenem Würfelergebnis eine Spielfigur an die richtige Stelle positioniert wird.
calculateMoveTest2	Überprüft, ob ein Zug ausgehend von einem unbesetzten Spielfeld als nicht durchführbar erkannt wird.
isAllowedToRediceTest	Überprüft, ob bei einem Spieler, bei dem alle Spielfiguren auf den Initialfeldern stehen, erkannt wird, dass er nochmals würfeln darf.
calculateMovesTest	Überprüft, ob bei gegebenem Würfelergebnis für einen Spieler alle möglichen validen Züge zurückgegeben werden.
startNodeTest	Überprüft, ob alle Startfelder im Graphen vorhanden sind.
endNodeTest1	Überprüft, ob im Graphen die Kanten zu den Zielfeldern vorhanden sind.
endNodeTest2	Überprüft, ob die Anzahl an Kanten im Graphen richtig ist.
calculateMovesTest2	Überprüft, ob ein von der Visualisierung kommender valider Zug richtig weitergeleitet wird.
diceTest	Überprüft, ob beim Würfeln eine Zahl zwischen 1 und 6 herauskommt.

Tabelle 5.1: 10 Unit Tests

### 5.2 ATRIP: Automatic (1P)

*[Begründung/Erläuterung, wie 'Automatic' realisiert wurde]*

Die Tests sind einfach auszuführen, denn es reicht auf dem Ordner, in dem alle Tests enthalten sind, das Starten aller Tests auszuwählen. Ein anderen Weg wäre das Ausführen des Befehls **mvn clean test**. Außerdem müssen bei keinem Test Daten manuell eingegeben werden. Somit laufen alle Tests automatisch ab. Des Weiteren überprüfen sich alle Tests selbst und geben als Ergebnis entweder bestanden oder fehlgeschlagen zurück.

## 5.3 ATRIP: Thorough (1P)

*[Code Coverage im Projekt analysieren und begründen]*

Die Code Coverage im Projekt ist leider eher dürftig. Es wurden zwar die Kernfunktionen mit Tests abgedeckt, aber dennoch fehlen auch einige Aspekte, die wir als notwendig bezeichnen würden. Zum Beispiel gibt es keinerlei Tests zur GUI. Diese wurde nur händisch getestet.

Bei den Tests wurden einmal die möglichen Aktionen auf dem Spielbrett betrachtet. Hierzu zählen fünf ersten Tests aus der vorherigen Tabelle. Außerdem wurde die dem Spielbrett zugrundeliegende Struktur, also der darunterliegende Graph, getestet. Hierbei sind aber auch nur circa 10% des fertigen Graphen überprüft worden. Da dieser Graph zur Validierung von Zügen beisteuert ist dies eine kritische Stelle, die nicht mit Tests abgedeckt ist. Zum Schluss wurde dann noch der „GameService“ mithilfe dreier Mocks getestet. Aber auch hier fehlt zum Beispiel die Überprüfung, ob ein neues Spiel richtig aufgebaut wird.

Bei der Erstellung der Tests wurde der Fokus auf die Kernfunktionalitäten gelegt. Es wurde aber beim Auftritt eines Fehlers dieser im Code verbessert und nicht direkt ein Test geschrieben. Zukünftig sollte das anders gehandhabt und auch direkt das Umfeld des Fehlers betrachtet werden.

## 5.4 ATRIP: Professional (1P)

*[jeweils 1 positives und negatives Beispiele zu „Professional“; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell an den Beispielen ist]*

```
public class ControlMechanismFourTest {

    private BoardInterface board;
    private ControlMechanismFour controlMechanismFour;

    @BeforeEach
    public void beforeEachTest() {

        controlMechanismFour = new ControlMechanismFour();
        board = new BoardFourMock(controlMechanismFour);
        board.initBoard( playerNumber: 4);

    }

    @Test
    public void checkWinTest(){

        Color winColor = controlMechanismFour.checkWin(board.getBoard());
        assertEquals(Color.BLUE, winColor);

    }

    @Test
    public void isAllowedToRediceTest1(){

        boolean isAllowedToRedice = controlMechanismFour.isAllowedToRedice(Color.RED, board.getBoard());

        assertTrue(isAllowedToRedice);

    }

}
```

Abbildung 5.1: Positiv-Beispiel Professional [Eigene Darstellung aus *IntelliJ*]

Hier ist ein positives Beispiel zu „Professional“ zu sehen. Es wird ein Mock für das Spielbrett eingesetzt. In der Testklasse werden verschiedene Aktionen beziehungsweise Funktionalitäten auf dem Spielbrett getestet. Hierbei wird der Code in der „beforeEachTest“-Methode wiederverwendet, was für „Professional“ spricht. Hierdurch werden auch Fehlerquellen verkleinert, denn beispielsweise bei Änderungen muss hier nur eine Stelle im Code bedacht werden. Dies spricht auch für eine gute Code-Qualität.

```

@Test
public void graphInstanceTest(){

    assertEquals(graph, Graph.INSTANCE);

}

```

Abbildung 5.2: Negativ-Beispiel Professional [Eigene Darstellung aus *IntelliJ*]

Dieser Test überprüft, ob es tatsächlich nur eine Graph-Instanz gibt. Das ist aber sinnlos und ein Negativ-Beispiel zu „Professional“, da der Graph ein Singleton ist und somit nur eine Instanz existiert.

## 5.5 Fakes und Mocks (1P)

*[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten; zusätzlich jeweils UML Diagramm der Klasse]*

### BoardFourMock

Dieses Mock ist stellvertretend für ein Spielbrett. Dabei wurden nur die notwendigen Funktionalitäten implementiert. Das heißt, es gibt einen festen Spielstand der sozusagen „initialisiert“ wird. Bei diesem Spielstand wurden absichtlich die Spielfiguren so gestellt, dass die möglichen Aktionen gut getestet werden können. Zum Beispiel ist die Farbe Blau im Ziel und somit kann die Funktion, ob jemand gewonnen hat, überprüft werden. Ein anderen Beispiel ist, dass Rot noch auf der Startposition steht und somit überprüft werden kann, ob Rot auch dreimal würfeln darf.

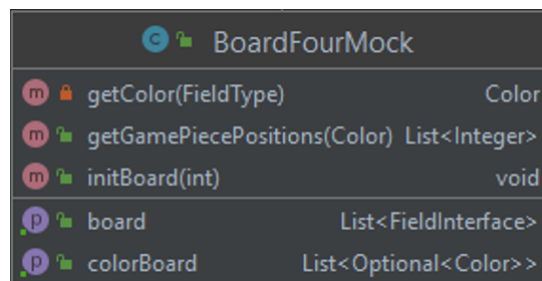
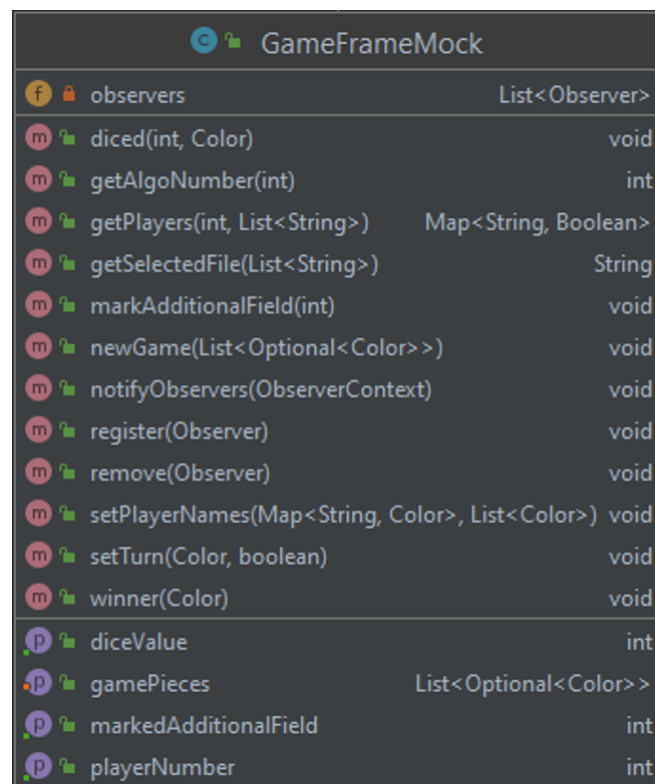


Abbildung 5.3: BoardFourMock [Eigene Darstellung aus *IntelliJ*]



## GameFrameMock

Das Mock „GameFrameMock“ ist stellvertretend für die GUI da. Um den „GameService“ ohne Abhängigkeiten testen zu können, wurden durch dieses Mock die minimal notwendigen Funktionalitäten geboten. Zum Beispiel konnte ohne die große GUI-Komponente mit einem akzeptablen Aufwand getestet werden, ob ein von der Visualisierung kommender valider Zug richtig weitergeleitet wird.



GameFrameMock		
f	observers	List<Observer>
m	diced(int, Color)	void
m	getAlgoNumber(int)	int
m	getPlayers(int, List<String>)	Map<String, Boolean>
m	getSelectedFile(List<String>)	String
m	markAdditionalField(int)	void
m	newGame(List<Optional<Color>>)	void
m	notifyObservers(ObserverContext)	void
m	register(Observer)	void
m	remove(Observer)	void
m	setPlayerNames(Map<String, Color>, List<Color>)	void
m	setTurn(Color, boolean)	void
m	winner(Color)	void
p	diceValue	int
p	gamePieces	List<Optional<Color>>
p	markedAdditionalField	int
p	playerNumber	int

Abbildung 5.4: GameFrameMock [Eigene Darstellung aus *IntelliJ*]

## 6. Domain Driven Design (8P)

### 6.1 Ubiquitous Language (2P)

*[4 Beispiele für die Ubiquitous Language; jeweils Bezeichnung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]*

Bezeichnung	Bedeutung	Begründung
Field	Spielfeld, auf dem eine Figur stehen kann	Spielfeld / -brett kann für zwei in diesem Projekt genutzte Konstrukte stehen. Deshalb muss hier unterschieden werden, um zwischen Entwicklern und Benutzern eine gemeinsame Sprache zu finden.
Board	Spielbrett, das die Spielfiguren und die „Fields“ enthält	
Move	Zug eines Spielers	Hier könnte „turn“ für den Entwickler ein Zug eines Spielers heißen, für den Benutzer wäre dies aber ein „move“. Deswegen müssen diese beiden Begriffe in die Ubiquitous Language aufgenommen werden.
Turn	Sagt aus, wer am Zug ist.	

Tabelle 6.1: 4 Beispiele für die Ubiquitous Language

## 6.2 Repositories (1,5P)

*[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]*

Das Repository wird durch die Klasse „GameIO“ repräsentiert und bietet Zugriff auf persistenten Speicher. Außerdem wird die konkret verwendete Speichertechnologie vor dem Domain Code verborgen. In unserem Beispiel ist es das Abspeichern und Laden von Spielständen, was mithilfe von JSON umgesetzt wird. Dadurch kann ein Spiel unterbrochen und später fortgesetzt werden.

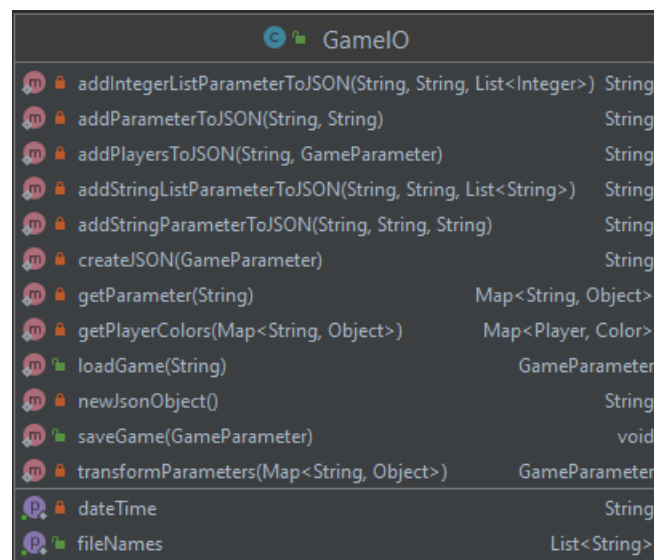


Abbildung 6.1: Repository [Eigene Darstellung aus *IntelliJ*]

## 6.3 Aggregates (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

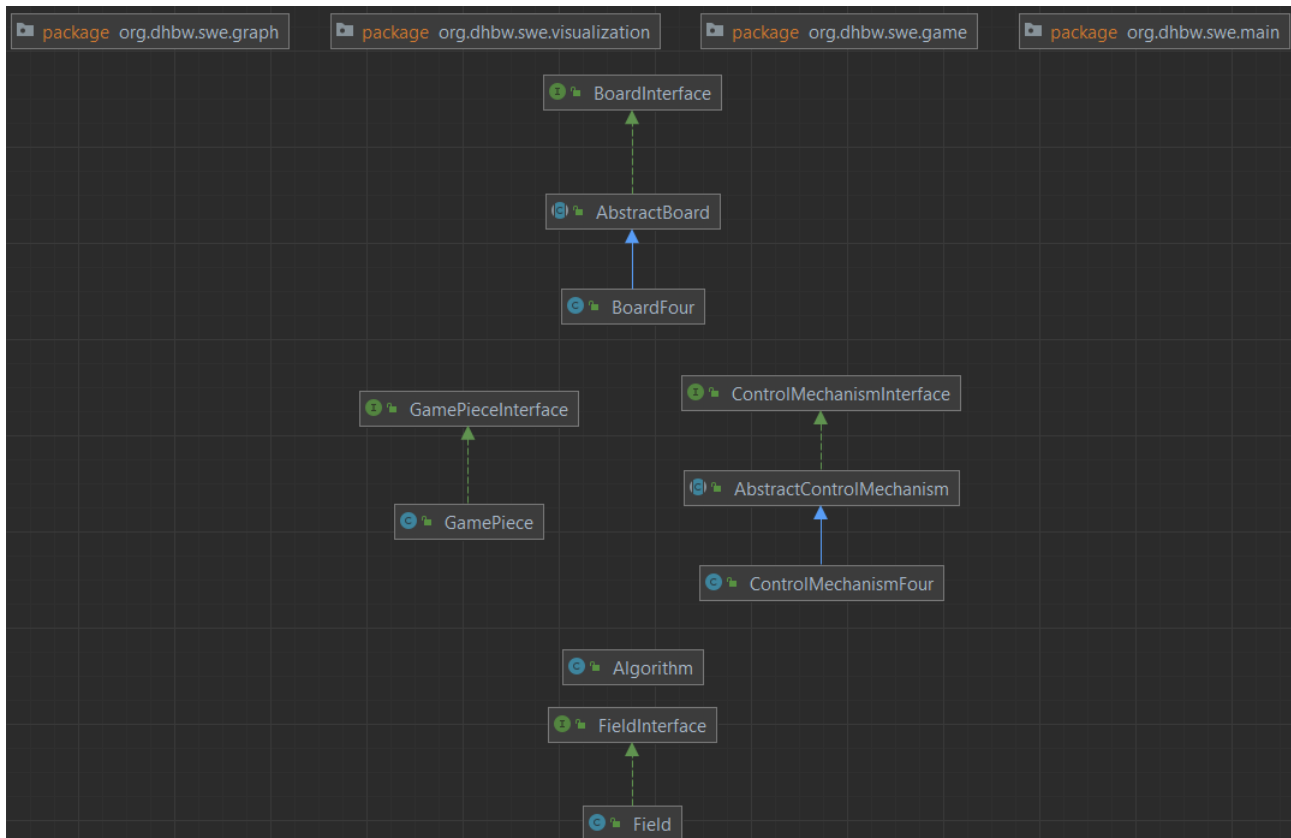


Abbildung 6.2: Aggregate [Eigene Darstellung aus *IntelliJ*]

Die in Abbildung 6.2 dargestellten Klassen gehören zu einem Aggregate. Dies ist eine gemeinsam verwaltete Einheit und ist innerhalb seiner Grenzen konsistent. Durch diese Konsistenzgrenze war es sinnvoll, das Aggregate für das Spielbrett einzusetzen, da beispielsweise auch die Züge stets konsistent sein müssen. Als Aggregatswurzel fungiert das „BoardInterface“. Dieses kontrolliert alle Zugriffe auf das Aggregate. Wenn eines der vier oben dargestellten Packages Änderungen am Spielbrett vornehmen möchte, geht dies nur über die Aggregatswurzel.

## 6.4 Entities (1,5P)

*[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]*

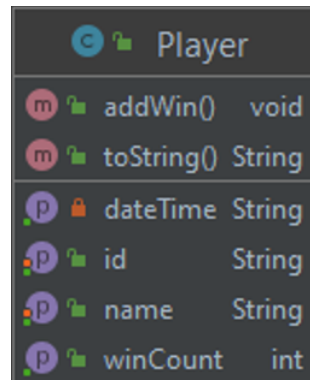


Abbildung 6.3: Entitiy Player [Eigene Darstellung aus *IntelliJ*]

Jeder Player hat eine eindeutige Identität. Alle Spieler, die jemals teilgenommen haben, werden in einer JSON-Datei gespeichert und durch die „id“ identifiziert. Dadurch kann auch die insgesamt Anzahl an Gewinnen abgespeichert werden. Anhand dieser sich über die Zeit verändernden Zahl „winCount“ ist zu erkennen, dass der „Player“ veränderliche Eigenschaften und einen Lebenszyklus hat.

Aufgrund der Anforderungen waren hier eine Identität und veränderliche Eigenschaften notwendig. Da dies Merkmale einer Entity sind war es hier sinnvoll den „Player“ als eine solche zu implementieren.

## 6.5 Value Objects (1,5P)

*[UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]*

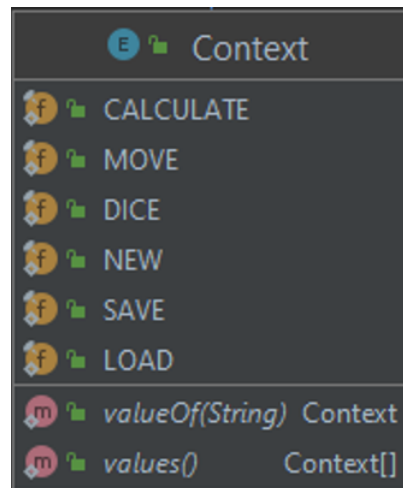


Abbildung 6.4: Enum als Value Object [Eigene Darstellung aus *IntelliJ*]

Ein Enum hat keine Identität, was auch auf Value Objects zutrifft. Außerdem sind Value Objects gleich, wenn sie denselben Wert haben. Dies lässt sich einfach durch **Context.DICE.equals(Context.DICE)**. Da dies wahr ist, trifft auch dieser Aspekt des ValueObject auf das „Context“-Enum zu. Des Weiteren haben ValueObjects keinen Lebenszyklus, was auch hier passt. Da das „Context“-Enum keine Instanzvariablen besitzt ist es auch unveränderlich. Das bedeutet nicht alle Enums sind Value Objects, aber das hier gezeigte Enum ist eines.

# 7. Refactoring (8P)

## 7.1 Code Smells (2P)

*[jeweils 1 Code-Beispiel zu 2 unterschiedlichen Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]*

### Long Method

Dieser Code Smell war in der „calculateTurn“-Methode in der „ControlMechanismFour“-Klasse enthalten. Wie in Abbildung 7.1 ersichtlich, ist die Methode viel zu lang. Zusätzlich mussten noch Kommentare eingefügt werden, dass die Methode überhaupt einigermaßen verständlich ist.

Nach der Eliminierung des Code Smells sind nur noch die drei logischen Möglichkeiten nach den Spielregeln in der Methode enthalten. Das eigentliche Berechnen der Züge ist ausgelagert. Durch die Methodennamen sind auch keine Kommentare mehr notwendig.

```
public Map<GamePieceInterface, Integer> calculateMoves(Color color, List<FieldInterface> board, int dice) {  
  
    if (needsToMoveFromInitFieldToStartField(color, board, dice)) {  
        return calculateMovesFromInitFields(color, board);  
    }  
  
    if (needsToMoveFromStartField(color, board, dice)) {  
        return calculateMovesFromStartField(color, board, dice);  
    }  
  
    return calculateMovesIgnoringStartField(color, board, dice);  
}
```

Abbildung 7.1: Eliminierung der Long Method [Eigene Darstellung aus *IntelliJ*]

```

public Map<GamePieceInterface, Integer> calculateTurns(final Color color, final List<FieldInterface> field, final int dice) {

    final Map<GamePieceInterface, Integer> result = new HashMap<>();

    //Falls eine 6 gewürfelt wurde und eine Figur der Farbe im Haus ist, muss aus dem Haus rausgesprungen werden
    if (dice == 6 && field.stream()
        .filter(x -> x.getType().equals(GraphUtilities.getInitType(color)))
        .anyMatch(x -> x.getGamePiece() != null) && !field.stream()
            .filter(x -> x.getType().equals(GraphUtilities.getStartType(color)))
            .anyMatch(x -> x.getGamePiece() != null && x.getGamePiece().color().equals(color))) {

        //Besetzten Felder im Haus herausfinden
        final List<FieldInterface> initFields = field.stream()
            .filter(x -> x.getType().equals(GraphUtilities.getInitType(color)) && x.getGamePiece() != null)
            .collect(Collectors.toList());

        //Die möglichen Züge aus dem Haus zurückgeben
        for (final FieldInterface initField : initFields) {
            result.put(initField.getGamePiece(), this.getStartPosition(color));
        }

        return result;
    }

    final FieldInterface startField = field.stream()
        .filter(x -> x.getType().equals(GraphUtilities.getStartType(color)))
        .findFirst().get();

    //Das Feld vor dem Haus muss freigemacht werden, wenn da eine eigene Figur steht
    //Ausführlich: Falls Figuren im Haus & das Feld vor dem Haus durch eigene Figur besetzt ist &
    // das Feld, auf das die Figur vor dem eigenen Haus springen könnte, nicht durch eigene Figur besetzt oder leer ist
    if (field.stream()
        .filter(x -> x.getType().equals(GraphUtilities.getInitType(color)))
        .anyMatch(x -> x.getGamePiece() != null)
        && startField.getGamePiece() != null
        && startField.getGamePiece().color().equals(color)
        && ((field.get(this.getTargetPosition(Graph.INSTANCE.four.get(field.indexOf(startField))), dice, color)).getGamePiece() != null
        && !field.get(this.getTargetPosition(Graph.INSTANCE.four.get(field.indexOf(startField))), dice, color)).getGamePiece().color().equals(color))
        || field.get(this.getTargetPosition(Graph.INSTANCE.four.get(field.indexOf(startField))), dice, color)).getGamePiece() == null)) {

        return Map.of(startField.getGamePiece(), this.getTargetPosition(Graph.INSTANCE.four.get(field.indexOf(startField))), dice, color));
    }

    //Alle Möglichkeiten auf dem Spielfeld zurückgeben
    final List<Node> currentPositions = this.getCurrentPosition(color, field).stream()
        .filter(x -> !x.getType().equals(GraphUtilities.getInitType(color))).collect(Collectors.toList());

    for (final Node node : currentPositions) {

        final int target = this.getTargetPosition(node, dice, color);
        if (!currentPositions.stream().map(x -> Graph.INSTANCE.four.indexOf(x)).anyMatch(x -> x == target) && target != -1) {
            result.put(field.get(this.graph.four.indexOf(node)).getGamePiece(), target);
        }
    }

    return result;
}

```

Abbildung 7.2: Long Method [Eigene Darstellung aus *IntelliJ*]



## Switch-Statements

```
public class GraphUtilities {

    public static FieldType getInitType(final Color color) {
        if (color.equals(Color.RED)) {
            return FieldType.REDINIT;
        }
        if (color.equals(Color.BLUE)) {
            return FieldType.BLUEINIT;
        }
        if (color.equals(Color.GREEN)) {
            return FieldType.GREENINIT;
        }
        return FieldType.YELLOWINIT;
    }

    public static FieldType getStartType(final Color color) {
        if (color.equals(Color.RED)) {
            return FieldType.REDSTART;
        }
        if (color.equals(Color.BLUE)) {
            return FieldType.BLUESTART;
        }
        if (color.equals(Color.GREEN)) {
            return FieldType.GREENSTART;
        }
        return FieldType.YELLOWSTART;
    }

    public static FieldType getEndType(final Color color) {
        if (color.equals(Color.RED)) {
            return FieldType.REDEND;
        }
        if (color.equals(Color.BLUE)) {
            return FieldType.BLUEEND;
        }
        if (color.equals(Color.GREEN)) {
            return FieldType.GREENEND;
        }
        return FieldType.YELLOWEND;
    }

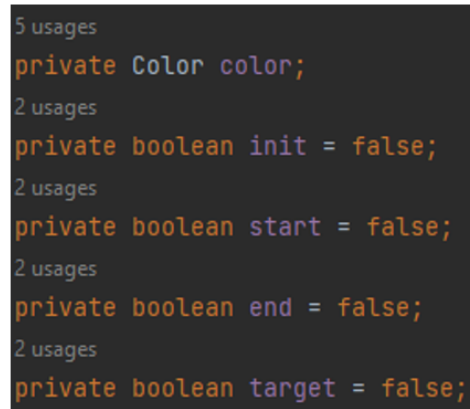
    public static FieldType getTargetType(final Color color) {
        if (color.equals(Color.RED)) {
            return FieldType.REDTARGET;
        }
        if (color.equals(Color.BLUE)) {
            return FieldType.BLUETARGET;
        }
        if (color.equals(Color.GREEN)) {
            return FieldType.GREENTARGET;
        }
        return FieldType.YELLOWTARGET;
    }

}
```

Abbildung 7.3: Switch-Statements [Eigene Darstellung aus *IntelliJ*]

Der Code-Smell bezieht sich auf die vier Methoden der „GraphUtilities“-Klasse (siehe Abbildung 7.3). Der Switch-Statements Code Smell gilt auch hier, da die if-else-Statements einfach durch switch-Statements ersetzt werden können. Da sich diese Methoden alle um das „FieldType“-Enum drehen, wäre an dieser Stelle eine Implementierung der Methoden im Enum sinnvoller und würden den Code-Smell entfernen.

Das Enum wurde um fünf Instanzvariablen erweitert:



```
5 usages
private Color color;
2 usages
private boolean init = false;
2 usages
private boolean start = false;
2 usages
private boolean end = false;
2 usages
private boolean target = false;
```

Abbildung 7.4: Erweiterung um fünf Instanzvariablen [Eigene Darstellung aus *IntelliJ*]

Außerdem wurden die vier Funktionalitäten aus den Graph-Utilities hinzugefügt, wie auf der nächsten Seite zu sehen ist.

```

6 usages new *
public static FieldType getInitType(Color color){

    FieldType result = null;

    for (FieldType fieldType : values()) {
        if (color == fieldType.color && fieldType.init == true) {
            result = fieldType;
        }
    }

    return result;
}

3 usages new *
public static FieldType getStartType(Color color) {

    FieldType result = null;

    for (FieldType fieldType : values()) {
        if (color == fieldType.color && fieldType.start == true) {
            result = fieldType;
        }
    }

    return result;
}

2 usages new *
public static FieldType getEndType(Color color) {

    FieldType result = null;

    for (FieldType fieldType : values()) {
        if (color == fieldType.color && fieldType.end == true) {
            result = fieldType;
        }
    }

    return result;
}

6 usages new *
public static FieldType getTargetType(Color color) {

    FieldType result = null;

    for (FieldType fieldType : values()) {
        if (color == fieldType.color && fieldType.target == true) {
            result = fieldType;
        }
    }

    return result;
}

```

Abbildung 7.5: Hinzufügen der vier Funktionalitäten [Eigene Darstellung aus *IntelliJ*]

## 7.2 2 Refactorings (6P)

[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]

### Extract Method

Die Methode „saveGame“ in der Klasse „GameIO“ hat zuerst einen JSON-String erstellt und dann diesen mit dem aktuellen Zeitpunkt im Dateinamen abgespeichert. Beim Refactoring wurde hier einmal das Erstellen des JSON-Strings und das Generieren des aktuellen Zeitpunkts ausgelagert. Jetzt befindet sich nur noch der tatsächliche Abspeicherungsprozess in der Methode. Sichtbar war der alte Stand bis zum Commit **aee0fe5f52b545c68111ffcfbef3e99b8f85caf8** vom 27.05.2022.

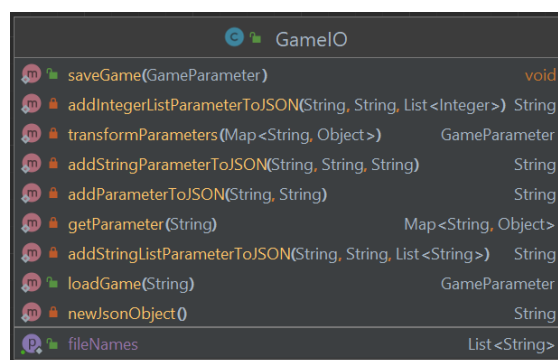


Abbildung 7.6: Extract Method (vorher) [Eigene Darstellung aus *IntelliJ*]

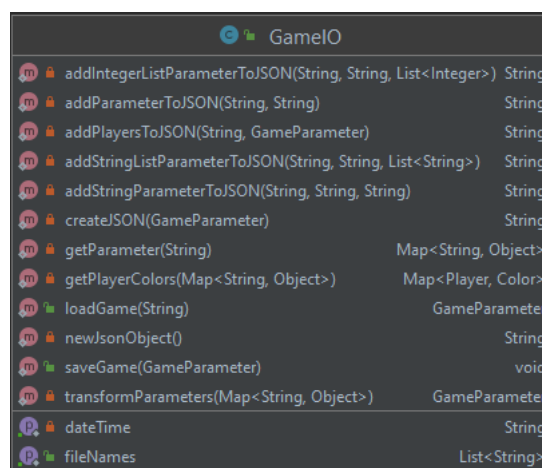
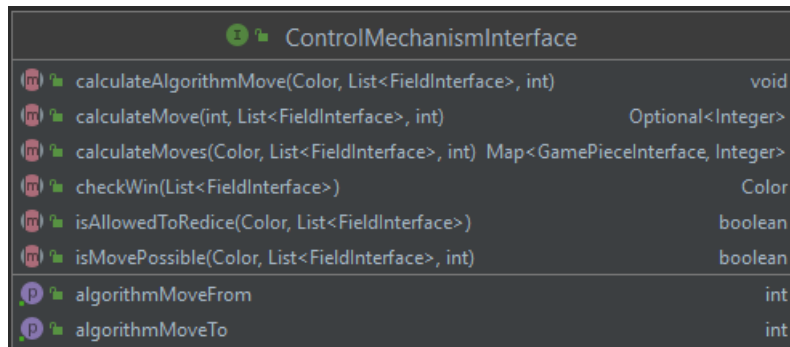


Abbildung 7.7: Extract Method (nacher) [Eigene Darstellung aus *IntelliJ*]

## Rename Method

Bei diesem Refactoring wurde auf die einheitliche Ubiquitous Language umgestellt. Dies ist zum Beispiel bei der *CalculateMove*-Methode im *ControlMechanismInterface* der Fall. Hier stand in den Argumenten statt *board* der Begriff *field*. Sichtbar war der alte Stand bis zum Commit **aee0fe5f52b545c68111ffcfbef3e99b8f85caf8** vom 27.05.2022.



ControlMechanismInterface		
calculateAlgorithmMove	(Color, List<FieldInterface>, int)	void
calculateMove	(int, List<FieldInterface>, int)	Optional<Integer>
calculateMoves	(Color, List<FieldInterface>, int)	Map<GamePieceInterface, Integer>
checkWin	(List<FieldInterface>)	Color
isAllowedToRedice	(Color, List<FieldInterface>)	boolean
isMovePossible	(Color, List<FieldInterface>, int)	boolean
algorithmMoveFrom		int
algorithmMoveTo		int

Abbildung 7.8: Rename Method [Eigene Darstellung aus *IntelliJ*]

## 8. Entwurfsmuster (8P)

[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils sinnvoll einsetzen, begründen und UML-Diagramm]

### Entwurfsmuster: Singleton (4P)

Der Graph ist ein Singleton-Enum und liegt sozusagen unter dem Spielbrett. Er besteht aus Knoten und gerichteten Kanten und gibt somit die Struktur wieder. Da der Graph von mehreren Teilen des Systems verwendet wird und sich nie ändert, ist es hier nicht notwendig mehrere Instanzen des Graphen zu haben. Deswegen haben wir uns an dieser Stelle für das Singleton-Pattern entschieden.

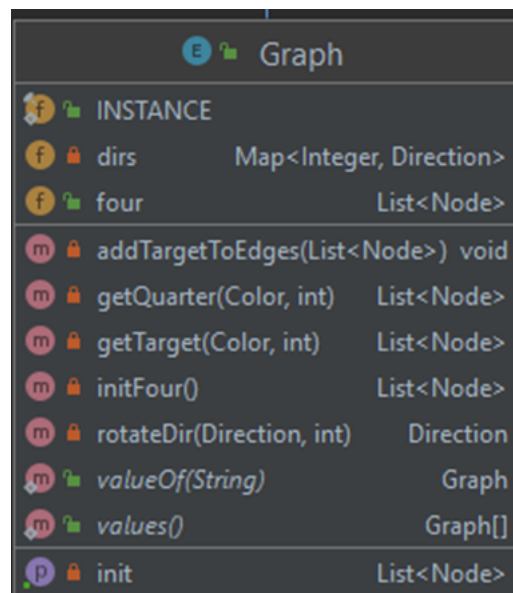


Abbildung 8.1: Graph als Singleton-Enum [Eigene Darstellung aus *IntelliJ*]

## Entwurfsmuster: Observer-Pattern (4P)

Das Observer-Pattern wurde eingesetzt, um eine lose Kopplung zwischen der Verwaltung des Spiels (*GameService*-Klasse) und der Visualisierung (*GameFrame*-Klasse) zu schaffen. Da die *GameFrame*-Klasse mit dem Benutzer interagiert, muss dies an die *GameService*-Klasse weitergeleitet werden. Ein direkter Methodenaufruf würde hier eine starke Kopplung schaffen und deshalb haben wir uns für das Observer-Pattern entschieden.

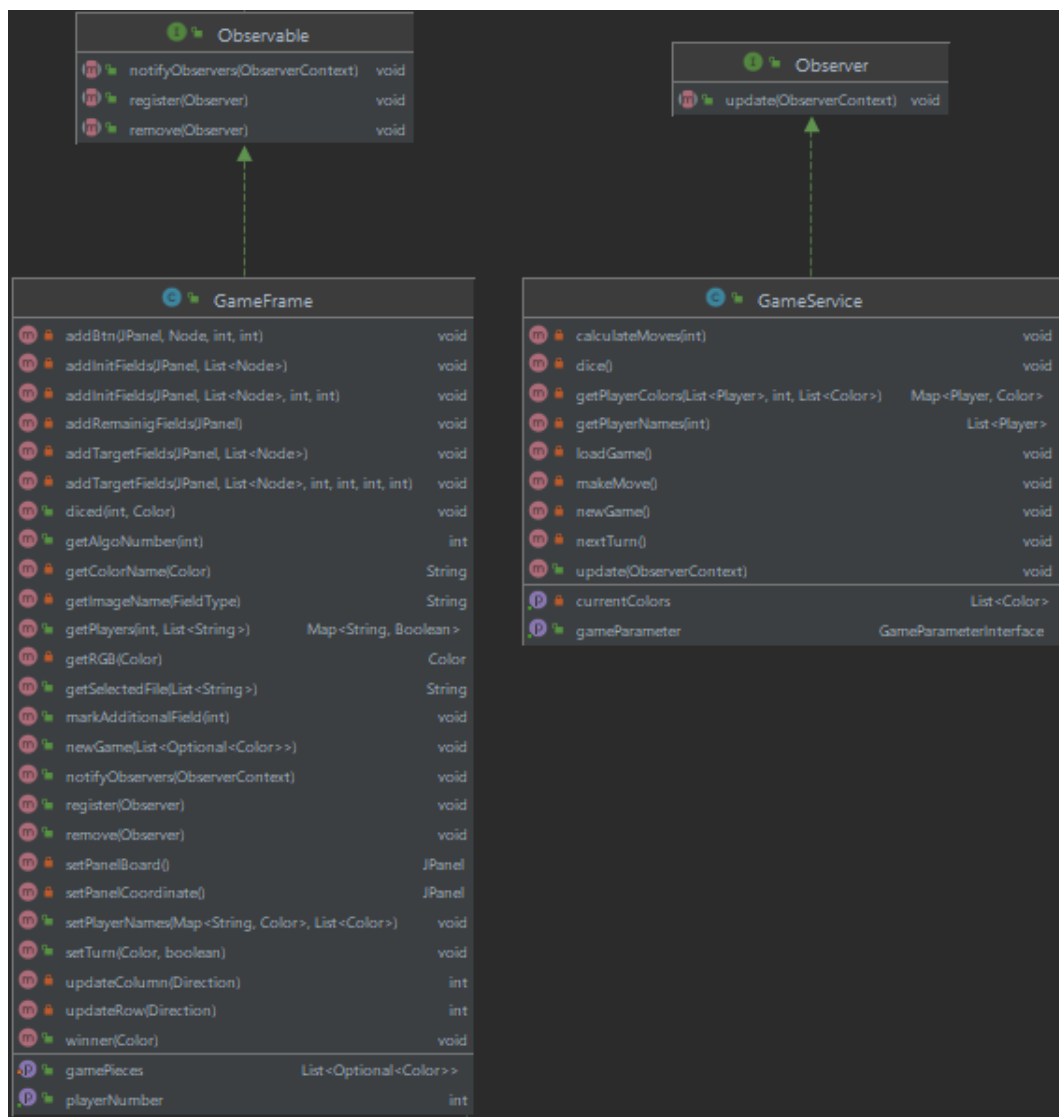


Abbildung 8.2: Observer-Pattern [Eigene Darstellung aus *IntelliJ*]

## 9. Anhang: Bedienungsanleitung

Wie bereits erläutert kann die Applikation durch Ausführen der Datei **sweProject.jar** gestartet werden. Zu Beginn eines Spiels wird zunächst die Anzahl an Spielern angegeben. Hierbei kann zwischen 2, 3 oder 4 ausgewählt werden.

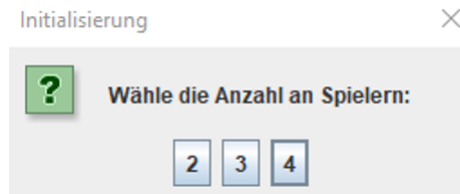


Abbildung 9.1: Auswahl der Spieleranzahl

Nachdem die Spieleranzahl ausgewählt wurde, wird als nächstes die Anzahl der durch den Algorithmus gespielten Spielern ausgewählt. Es können bis zu 3 Spieler als Algorithmus initialisiert werden.

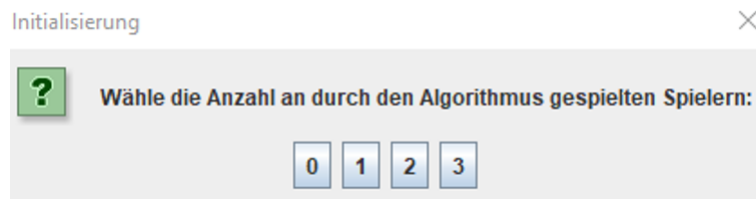


Abbildung 9.2: Auswahl der Anzahl an simulierten Spielern

Im nächsten Schritt werden die nicht-algorithmischen Spieler festgelegt. Hierbei können Spielern deren Namen zugewiesen werden. Alternativ können auch bereits existierende Spieler ausgewählt werden. Im Folgenden wurde Spieler 1 als neuer Spieler unter dem Namen Nadine angelegt. Als Spieler 2 wurde ein bereits existierender Spieler ausgewählt.



Wähle die Spieler

**Spieler 1**

☒ Neuer Spieler ☐ Existierender Spieler

Name:

**Spieler 2**

☐ Neuer Spieler ☒ Existierender Spieler

Name:

Ok

Abbildung 9.3: Benennung eines Spielers und Auswahl eines existierenden Spielers

Nachdem abschließend OK ausgewählt wurde beginnt das Spiel. Jeder Spieler erhält 4 Spielfiguren einer Farbe. Alle 4 Figuren stehen zu Beginn eines Spiels auf dem Feld A seiner Farbe. Die jeweiligen Spieler werden unterhalb des Spielfeldes mit Namen und ihrer jeweiligen Farbe angezeigt.

Die weißen Felder des Spielbretts stellen die Laufbahn C dar, die alle Spielfiguren zurücklegen müssen. Auf den jeweils farbigen Feldern auf der Laufbahn C beginnen die Spielfiguren ihren Weg über die weißen Felder. Auf der A-Feldern warten die Spielfiguren auf ihren Einsatz. Wer seine 4 Spielfiguren als erster in sein jeweiliges Ziel (Punkt D) gebracht hat, gewinnt das Spiel.

Der Spieler, der an der Reihe ist, würfelt durch Klicken auf das Symbol „Fragezeichen in der Box“ bei Punkt B. Wer keine Spielfigur auf der Laufbahn hat, weil alle Figuren geschlagen wurden oder zu Beginn des Spiels, darf dreimal würfeln. Die Spielfiguren, die auf den A-Feldern stehen, können nur mit einer „6“ in das Spiel gebracht werden. Wird eine „6“ gewürfelt, kann der Spieler selbst entscheiden mit welcher Figur er vom Startfeld A gehen möchte. Durch das Anklicken einer Figur wird diese ausgewählt und die Startposition auf der Laufbahn wird in Orange angezeigt. Wer eine „6“ würfelt, hat nach seinem Zug einen weiteren Wurf frei. Erzielt er dabei wieder eine „6“, darf er nach dem Ziehen erneut würfeln. Bei einer „6“ muss man eine neue Figur ins Spiel bringen, solange noch Spielfiguren auf dem eigenen A-Feldern stehen. Die neue Figur wird dann auf die Startposition der Laufbahn gestellt. Ist dieses Feld noch von einer anderen eigenen Spielfigur besetzt, muss diese Figur erst mit der „6“ weitergezogen werden. Steht dagegen eine fremde Figur auf dem Feld, wird sie geschlagen. Wer eine „6“ würfelt und keine Spielfigur mehr auf den B-Feldern hat, darf mit einer seiner Figuren auf der Laufbahn sechs Felder weiterziehen und dann noch einmal würfeln.

Eigene und fremde Figuren können übersprungen werden. Die besetzten Felder werden aber mitgezählt. Wer mehrere Spielfiguren auf der Laufbahn hat, kann sich aussuchen, mit welcher Figur er weiterzieht. Wer mit dem letzten Punkt seiner Augenzahl auf ein Feld tritt, das von einer fremden Spielfigur besetzt ist, schlägt diese Figur, welche wiederum automatisch zurück in den Startplatz gesetzt wird.

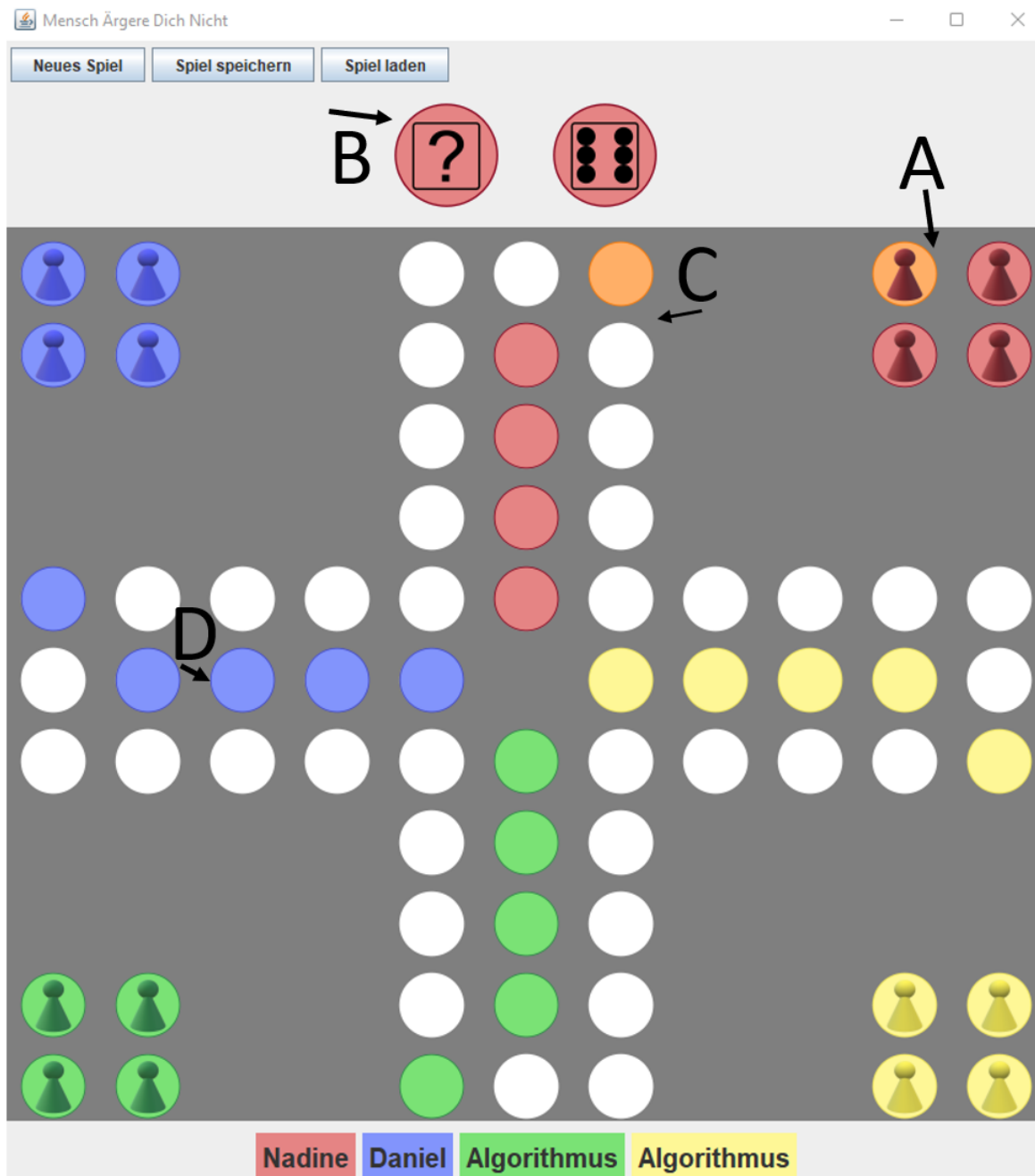


Abbildung 9.4: Hauptfenster der Anwendung mit Spielbrett