# ElferRaus

Gruppe P23
Lara Sievers, Adrian Schmidt, Fabian Schneider

Projektdokumentation

24.06.2015

# Inhalt

# 1. Pflichtenheft

ElferRaus
P23
08.05.2015

## 1.1. Management- und Dokumentationsattribute

| Dokumentationsattribute | |
|---|---|
| **Autor** | Lara Sievers, Adrian Schmidt, Fabian Schneider |
| **Eindeutige Teamnummer** | P23 |
| **Quelle** | |
| **Version** | 1.0 |
| **Bearbeitungsstatus** | Final |

## 1.2. Visionen und Ziele

/PV10/ Digitalisierung des Spiels als taktisches Kartenspiel

/PZ10/ Der Gegner soll eine K.I. sein.
/PZ20/ Das Spiel soll in der Kommandozeile laufen.

## 1.3. Rahmenbedingungen

/PR10/ Das Spiel soll auf einem Computer unabhängig vom Betriebssystem laufen.
/PR20/ Die Entwicklungsumgebung soll Eclipse sein.
/PR30/ Als Programmiersprache soll Java verwendet werden.
/PR40/ Das digitale Kartenspiel soll von Laien bedienbar sein.
/PR50/ Das Spiel soll für 1-2 Runden geeignet sein.
/PR60/ Das Programm soll als jar-Datei gepackt und ausführbar sein.
/PR70/ Das Spiel soll für Leute der Altersgruppe von 8 – 60 geeignet sein.
/PR80/ Höfliche Anredeform.
/PR90/ In dem Spiel dürfen keine gewalttätigen, rassistischen oder pornografischen Inhalte dargestellt werden.

## 1.4. Kontext und Überblick

/PK10/ Das Programm soll von allen zugänglich und benutzbar sein.
/PK20/ Zum Spielen wird eine herkömmliche Computermaus oder ein vergleichbares Zeigegerät und eine Tastatur oder ein vergeichbares Eingabegerät benötigt.
/PK30/ Das Spiel soll auf einem Computer unabhängig vom Betriebssystem laufen.
/PK40/ Das Spiel wird in einer IDE (Eclipse) entwickelt.

## 1.5. Funktionale Anforderungen

/PF10/ Muss eine K.I. enthalten.
/PF20/ Kann einstellbare Schwierigkeitsgrade haben (Leicht, Schwer).
/PF30/ Kann einen Multiplayermodus haben für insgesamt maximal 4 Spieler.
/PF40/ Kann mehrere K.I.-Gegner gleichzeitig haben (maximal 3).
/PF50/ Muss mit der Maus und Tastatur bedienbar sein.
/PF60/ Muss den Gewinner ausgeben.
/PF70/ Muss die Karten des aktiven Spielers (ausgenommen K.I.) anzeigen.
/PF80/ Muss die Karten der K.I. immer verbergen.
/PF90/ Muss die Anzahl der Karten auf der Hand darstellen.
/PF100/ Muss die Karten auf dem Spielfeld darstellen.
/PF110/ Soll den Stapel mit der Anzahl der verbleibenden Karten darstellen.
/PF120/ Muss zwischen vier Farben unterscheiden können (Rot, Orange, Grün, Blau).
/PF130/ Muss pro Farbe je eine Karte der Zahlen 1 bis 20 beinhalten.
/PF140/ Muss zwischen den Spielphasen unterscheiden, sodass der Gegner nicht in den Zug des aktiven Spielers eingreifen kann.
/PF150/ Das Spiel kann ein Hilfedokument zur Verfügung stellen.
/PF160/ Kann eine GUI haben.

## 1.6. Qualitätsanforderungen

| Systemqualität | Sehr gut | Gut | Normal | Nicht relevant |
|---|---|---|---|---|
| **Funktionalität** | | X | | |
| **Zuverlässigkeit** | | X | | |
| **Benutzbarkeit** | | X | | |
| **Effizienz** | | | X | |
| **Wartbarkeit** | | | X | |
| **Portabilität** | | | | X |

**Tabelle 1: Qualitätsanforderungen**

## 1.7. Abnahmekriterien

Das Programm startet und erstellt ein Set aus 20 Karten pro Farbe, bei vier Farben.
Die Karten werden gemischt.
Jeder Spieler erhält 11 Karten.
Eine Karte kann nur auf einen Stapel der gleichen Farbe gelegt werden.
Karten gleicher Farbe können nur zu Karten auf einen Stapel gelegt werden, wo vorher eine Karte um 1 kleiner der zu legenden Karte lag, bei Zahlen größer 11.
Karten gleicher Farbe können nur zu Karten auf einen Stapel gelegt werden, wo vorher eine Karte um 1 größer der zu legenden Karte lag, bei Zahlen kleiner 11.
Bei gelegten finalen Karten (1 oder 20) können keine weiteren Karten gelegt werden.
Sobald eine 11 vom Stapel gezogen wird muss diese gelegt werden.

Sofern beim Ziehen der jeweils ersten Karte pro Durchgang vom Stapel keine 11 gezogen wird, müssen zwei Karten nachgezogen werden.
Karten mit einer 11 die beim Nachziehen (zwei Karten ziehen) gezogen werden, können erst in der nächsten Runde gelegt werden.
Der Wert der Karte im System stimmt mit dem angezeigten Wert überein.
Die Karten des Gegners sind nicht einsehbar.

### 1.8. Glossar

Eclipse ↗IDE auf der Basis von Java
IDE Integrated Development Environment = Entwicklungsumgebung zum Programmieren von Software
GUI Graphical User Interface (Grafische Benutzeroberfläche)

### 1.9. Literatur

#### 1.1.1. Hinweis zu dieser Vorlage

Die Vorlage für dieses Pflichtenheft wurde Balzert (2009), S. 492 ff. entnommen.

#### 1.1.2. Literaturliste

Balzert, Helmut (2009). Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering. 3. Auflage. Heidelberg: Spektrum, Seite 492 ff.

## 2. Projektmanagement

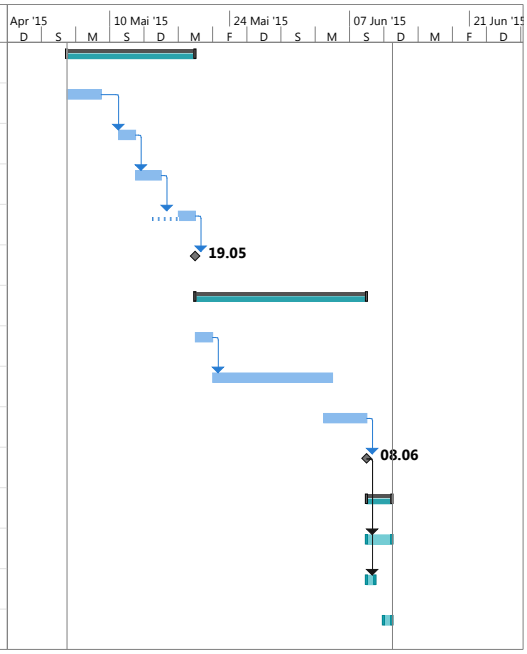
### 1.1. Rollen

Die Teammitglieder übernahmen folgende Rollen:
Lara Sievers: Dokumentation, Softwareentwicklung
Adrian Schmidt: Softwareentwicklung, Tester
Fabian Schneider: Softwareentwicklung, Teamleiter

# 3. Projektplan

## 1.1. Beginn des Projektes

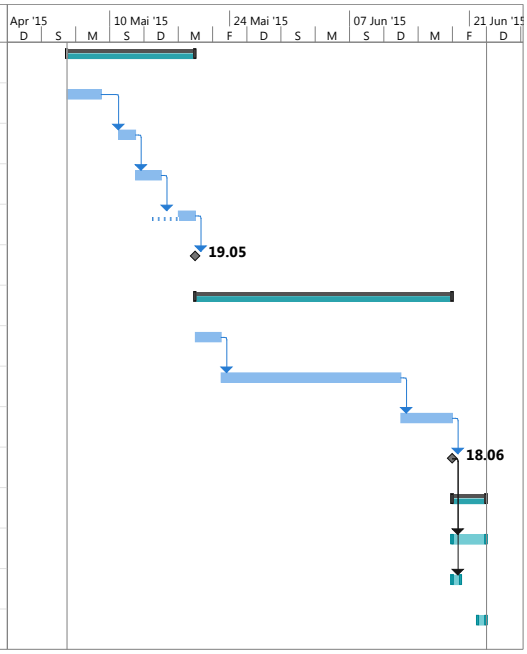
| Nr. | Vorgangs | Task Name | Dauer | Anfang | Ende |
|---|---|---|---|---|---|
| 1 | | **Analysephase** | **11 Tage** | **05.05.2015** | **19.05.2015** |
| 2 | | Pflichtenheft | 4 Tage | 05.05.2015 | 08.05.2015 |
| 3 | | Use-Case | 2 Tage | 11.05.2015 | 12.05.2015 |
| 4 | | Klassendiagramm | 3 Tage | 13.05.2015 | 15.05.2015 |
| 5 | | Sequenzdiagramm | 2 Tage | 15.05.2015 | 19.05.2015 |
| 6 | | Analaysephase abgeschlossen | 0 Tage | 19.05.2015 | 19.05.2015 |
| 7 | | **Programmierphase** | **14 Tage** | **20.05.2015** | **08.06.2015** |
| 8 | | Pseudocode | 2 Tage | 20.05.2015 | 21.05.2015 |
| 9 | | Volle Implementierung | 10 Tage | 22.05.2015 | 04.06.2015 |
| 10 | | Bugfixing | 3 Tage | 04.06.2015 | 08.06.2015 |
| 11 | | Programmierphase abgeschlossen | 0 Tage | 08.06.2015 | 08.06.2015 |
| 12 | | **Dokumentation** | **3 Tage** | **09.06.2015** | **11.06.2015** |
| 13 | | Präsentation erstellen | 3 Tage | 09.06.2015 | 11.06.2015 |
| 14 | | Screencast | 1 Tag | 09.06.2015 | 09.06.2015 |
| 15 | | Gesamtdokumenta zusammenfügen | 1 Tag | 11.06.2015 | 11.06.2015 |




Page 1

## 1.2. Ende des Projektes

| Nr. | Vorgangs | Task Name | Dauer | Anfang | Ende |
|---|---|---|---|---|---|
| 1 | | **Analysephase** | **11 Tage** | **05.05.2015** | **19.05.2015** |
| 2 | | Pflichtenheft | 4 Tage | 05.05.2015 | 08.05.2015 |
| 3 | | Use-Case | 2 Tage | 11.05.2015 | 12.05.2015 |
| 4 | | Klassendiagramm | 3 Tage | 13.05.2015 | 15.05.2015 |
| 5 | | Sequenzdiagramm | 2 Tage | 15.05.2015 | 19.05.2015 |
| 6 | | Analaysephase abgeschlossen | 0 Tage | 19.05.2015 | 19.05.2015 |
| 7 | | **Programmierphase** | **25 Tage** | **20.05.2015** | **18.06.2015** |
| 8 | | Pseudocode | 3 Tage | 20.05.2015 | 22.05.2015 |
| 9 | | Volle Implementierung | 16 Tage | 23.05.2015 | 12.06.2015 |
| 10 | | Bugfixing | 6 Tage | 13.06.2015 | 18.06.2015 |
| 11 | | Programmierphase abgeschlossen | 0 Tage | 18.06.2015 | 18.06.2015 |
| 12 | | **Dokumentation** | **3 Tage** | **19.06.2015** | **22.06.2015** |
| 13 | | Präsentation erstellen | 3 Tage | 19.06.2015 | 22.06.2015 |
| 14 | | Screencast | 1 Tag | 19.06.2015 | 19.06.2015 |
| 15 | | Gesamtdokumenta zusammenfügen | 1 Tag | 22.06.2015 | 22.06.2015 |



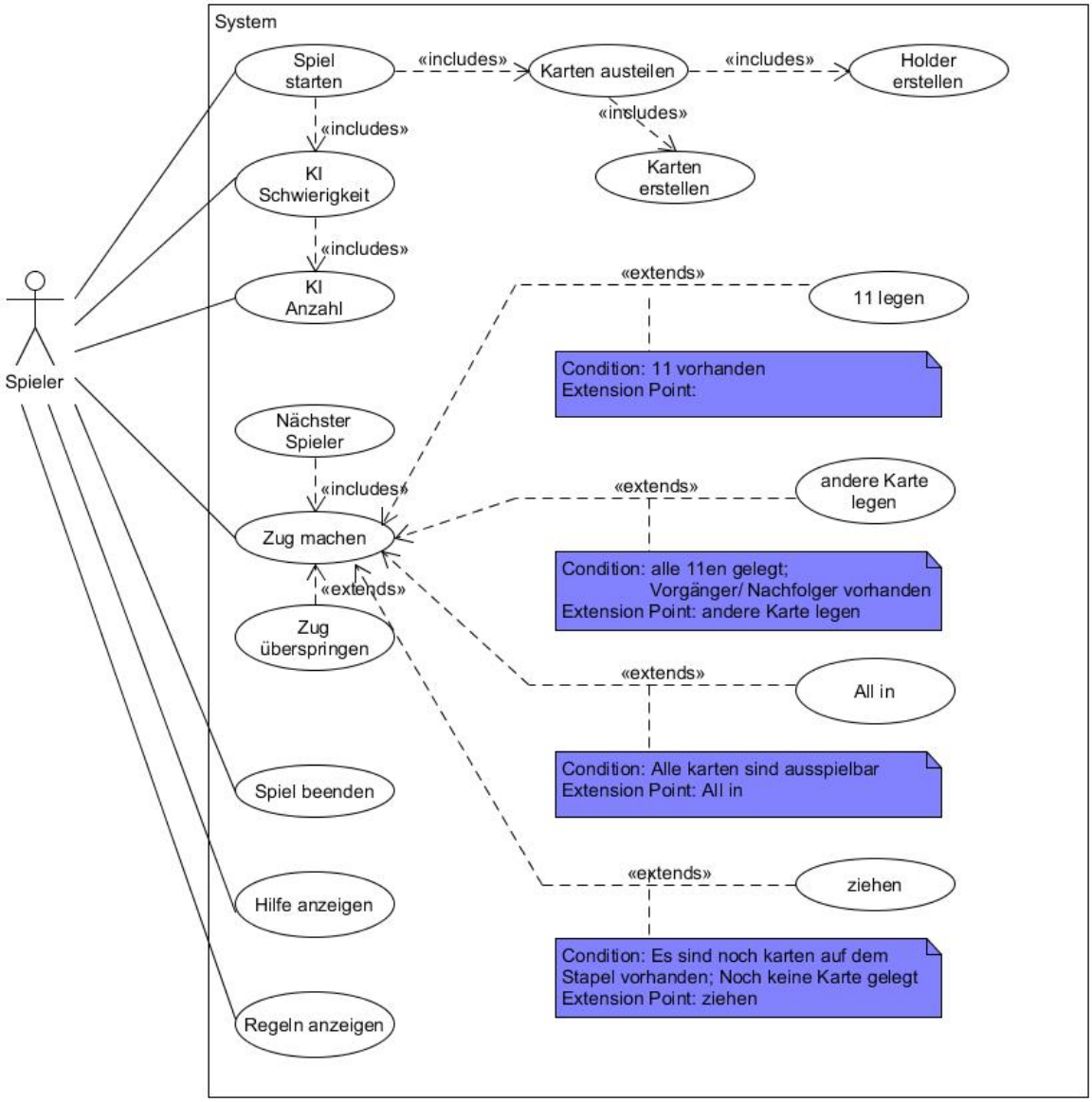

Page 1

## 1.3. Erläuterungen zu den Abweichungen

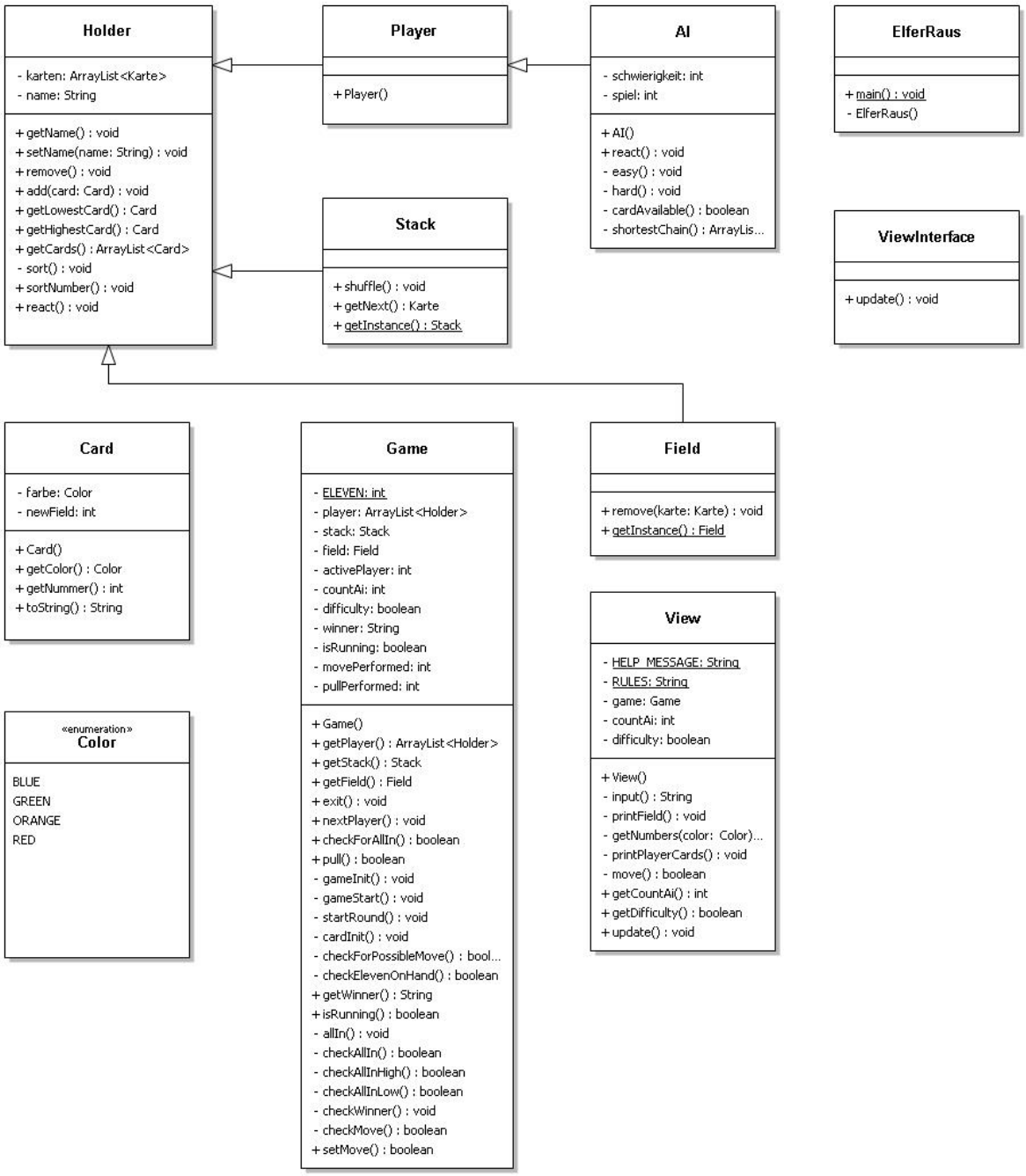
Die Abweichungen kamen in der Programmierphase zustande. Die Zeitplanung war gewollt enger gesetzt um das Ziel schneller zu erreichen. Dadurch wurde ein Puffer generiert, um Zeit für die Lösung unvorhersehbarer Probleme zur Verfügung zu haben. Diese Zeit wurde im Nachhinein benötigt.

# 4. Diagramme

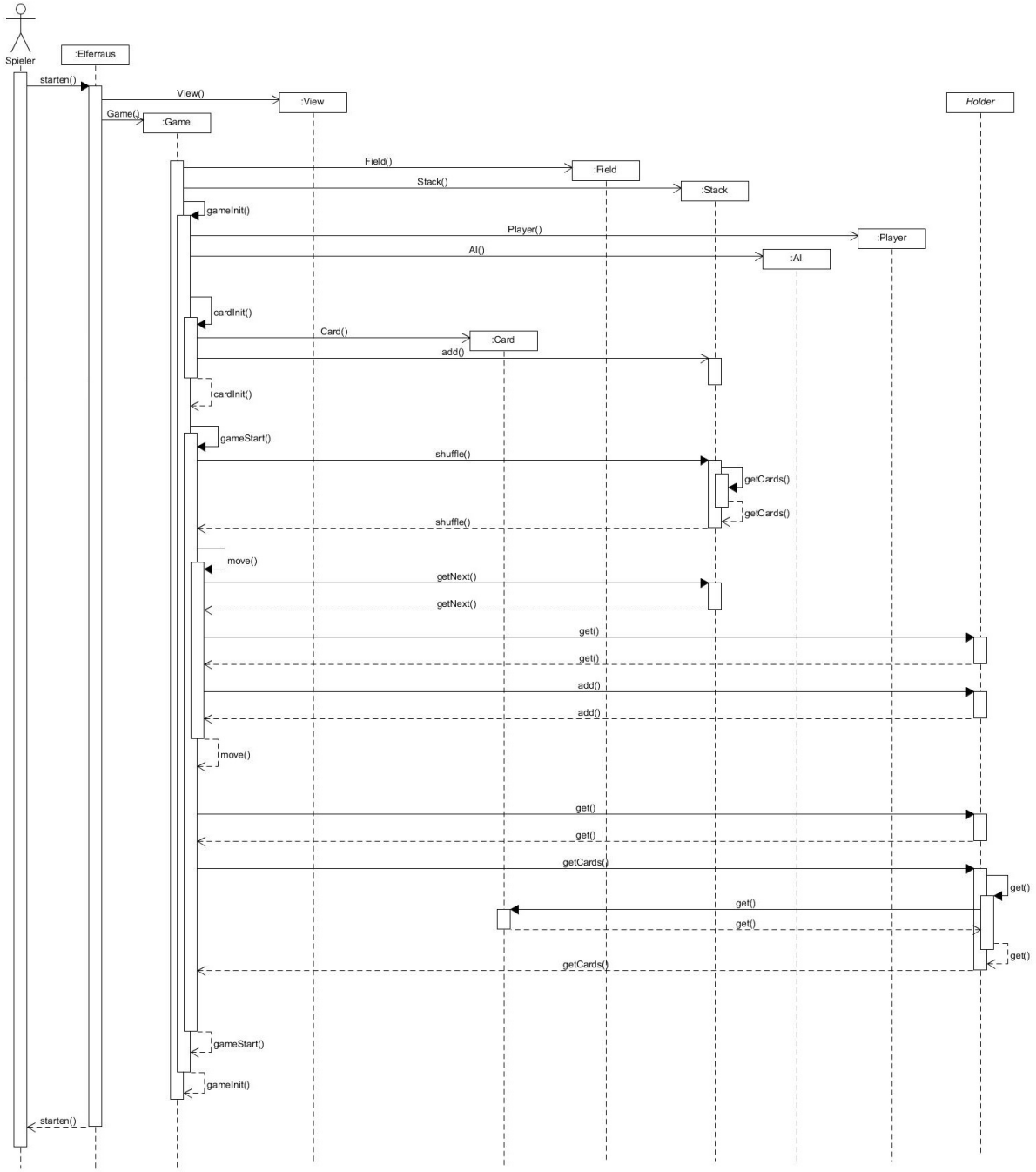
## 1.1. Use-Case Diagramm



## 1.2. Klassendiagramm



## 1.3. Sequenzdiagramm



# 5. Akzeptanztests

| Test | A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

1: Bestanden 0: nicht Bestanden

A: Das Programm startet und erstellt ein Set aus 20 Karten pro Farbe, bei vier Farben.
B: Die Karten werden gemischt.
C: Jeder Spieler erhält 11 Karten.
D: Eine Karte kann nur auf einen Stapel der gleichen Farbe gelegt werden.
E: Karten gleicher Farbe können nur zu Karten auf einen Stapel gelegt werden, wo vorher eine Karte um 1 kleiner der zu legenden Karte lag, bei Zahlen größer 11.
F: Karten gleicher Farbe können nur zu Karten auf einen Stapel gelegt werden, wo vorher eine Karte um 1 größer der zu legenden Karte lag, bei Zahlen kleiner 11.
G: Bei gelegten finalen Karten (1 oder 20) können keine weiteren Karten gelegt werden.
H: Sobald eine 11 vom Stapel gezogen wird muss diese gelegt werden.
I: Sofern beim Ziehen der jeweils ersten Karte pro Durchgang vom Stapel keine 11 gezogen wird, müssen zwei Karten nachgezogen werden.
J: Karten mit einer 11, die beim Nachziehen (zwei Karten ziehen) gezogen werden, können erst in der nächsten Runde gelegt werden.
K: Der Wert der Karte im System stimmt mit dem angezeigten Wert überein.
L: Die Karten des Gegners sind nicht einsehbar.

# 6. Quellcode

## 1.1. ElferRaus

```
/** Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

import control.Game;
import view.View;

/**
 * Hauptklasse des Spiels ElferRaus. Startet das Spiel.
 */
public final class ElferRaus {

	/**
	 * Hauptmethode des Spiels. Erstellt eine View und weist diese dem Spiel zu.
	 * @param args nicht genutzt.
	 */
	public static void main(String... args) {
		View view = new View();
		new Game(view);
	}

	private ElferRaus() {
	}
}
```

## 1.2. Game

```
package control;

/** Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

import java.util.ArrayList;

import view.View;
import data.Color;
import data.Holder;
import data.Card;
import data.AI;
import data.Player;
```

```
import data.Field;
import data.Stack;

public class Game {
	/** Die Schluesselzahl 11. */
	private static final int ELEVEN = 11;
	/** Spieler-ArrayList. */
	private final ArrayList<Holder> player = new ArrayList<Holder>();
	/** Stapel fuer die Karten. */
	private final Stack stack = Stack.getInstance();
	/** Spielfeld. */
	private final Field field = Field.getInstance();
	/** Index des aktiven Spielers. */
	private int activePlayer;
	/** Anzahl der vorhandenen Kis */
	private final int countKi;
	/** Schwierigkeit der KIs. */
	private boolean difficulty;
	/** Name des Gewinners. */
	private String winner = "niemand";
	/** true solange das Spiel laeuft. */
	private boolean isRunning = true;
	/** Haeufigkeit des Aufraufs der Methode setMove(). */
	private int movePerformed;
	/** Haeufigkeit des Aufraufs der Methode pull(). */
	private int pullPerformed;

	/**
	 * Erstellt ein Spiel und updatet die View.
	 * @param view Die TUI des Spiels
	 */
	public Game(final View view) {
		countKi = view.getCountKi();
		difficulty = view.getDifficulty();
		gameInit();
		while (isRunning) {
			view.update(this);
		}
	}

	/**
	 * Getter fuer die Spieler.
	 * @return Gibt die Spieler zurueck.
	 */
	public ArrayList<Holder> getPlayer() {
		return player;
	}

	/**
```

```
	 * Getter fuer den Stapel.
	 * @return Gibt den Stapel zurueck.
	 */
	public Stack getStack() {
		return stack;
	}

	/**
	 * Getter fuer das Spielfeld.
	 * @return Gibt das Spielfeld zurueck.
	 */
	public Field getField() {
		return field;
	}

	/**
	 * Getter fuer den Namen des Gewinners.
	 * @return Gibt den Namen des Gewinners zurueck.
	 */
	public String getWinner() {
		return winner;
	}

	/**
	 * Getter fuer den Spielzustand.
	 * @return Gibt den Spielzustand zurueck. True wenn das Spiel laeuft, sonst
	 *	 false.
	 */
	public boolean isRunning() {
		return isRunning;
	}

	/**
	 * Beendet das Spiel.
	 */
	public void exit() {
		isRunning = false;
	}

	/**
	 * Wechselt den aktiven Spieler, falls kein Zug mehr moeglich ist, und
	 * startet eine neue Runde.
	 */
	public void nextPlayer() {
		if (movePerformed == 0 && pullPerformed == 0 &&
checkForPossibleMove()) {
			System.out.println("Sie muessen eine Aktion ausfuehren!");
		} else {
			if (activePlayer < countKi) {
```

```
                        activePlayer++;
                } else {
                        activePlayer = 0;
                }

                movePerformed = 0;
                pullPerformed = 0;
                startRound();
        }
}

/**
 * Der gewuenschte Zug fuer den aktiven Spieler wird gesetzt.
 * @param color Farbe der zu bewegenden Karte.
 * @param number Nummer der zu bewegenden Karte.
 * @return true wenn der Zug durchgefuehrt wurde. Sonst false.
 */
public boolean setMove(final Color color, final int number) {
        for (Card k : player.get(activePlayer).getCards()) {
                if (k.getColor().equals(color) && k.getNumber() == number) {
                        if (k.getNumber() == ELEVEN) {
                                if (move(k, field)) {
                                        player.get(activePlayer).remove(k);
                                        movePerformed++;
                                        checkWinner();
                                        return true;
                                }
                        }
                        if (!checkElevenOnHand()) {
                                if (move(k, field)) {
                                        player.get(activePlayer).remove(k);
                                        movePerformed++;
                                        checkWinner();
                                        return true;
                                }
                        } else {
                                movePerformed++;
                                checkWinner();
                                return true;
                        }
                }

        }
        return false;
}

/**
 * Der aktive Spieler zieht eine Karte vom Stapel, falls er vorher Keine
 * gelegt hat oder noch eine Elf auf der Hand hat.
```

```
	 * @return true wenn die Karte vom Stapel gezogen wurde, wenn noch eine
Elf
	 *         auf der Hand ist, wenn schon gelegt worden ist. Sonst false
	 *         (=leer).
	 */
	public boolean pull() {
		boolean result = true;
		if (movePerformed == 0) {
			if (stack.getCards().size() != 0) {
				if (!checkElevenOnHand()) {
					Card card = stack.getNext();
					if (card.getNumber() == ELEVEN) {
						result = move(card, field);
					} else {
						result = move(card, player.get(activePlayer));
						result = move(stack.getNext(),
player.get(activePlayer));
						result = move(stack.getNext(),
player.get(activePlayer));
					}
					pullPerformed++;
					nextPlayer();
					return result;
				} else {
					pullPerformed++;
					nextPlayer();
					return true;
				}
			} else {
				return false;
			}
		} else {
			System.out.println("Sie koennen nach einem Zug nicht ziehen!");
			return true;
		}

	}

	/**
	 * Prueft alle Farben, ob alle Karten abgelegt werden koennen.
	 * @return true, wenn alle Karten abgelegt werden koennen, sonst false.
	 */
	public boolean checkForAllIn() {
		if (checkAllIn(Color.BLUE) && checkAllIn(Color.GREEN)
				&& checkAllIn(Color.ORANGE) && checkAllIn(Color.RED))
{
			allIn();
			return true;
		}
```

```
		if (activePlayer == 0) {
			System.out.println("Sie koennen nicht alle Karten ablegen!");
		}
		return false;
	}

	/**
	 * Leert die Hand des Spielers.
	 */
	private void allIn() {
		player.get(activePlayer).getCards().clear();
		checkWinner();
	}

	/**
	 * Prueft ob gleichzeitig alle Karte <&>11 gelegt werden koennen.
	 * @param color Farbe die geprueft werden soll.
	 * @return true, wenn alle Karte <&>11 gelegt werden koennen, sonst false.
	 */
	private boolean checkAllIn(final Color color) {
		if (checkAllInLow(color) && checkAllInHigh(color)) {
			return true;
		}
		return false;
	}

	/**
	 * Prueft ob alle Karten auf der Hand (>11) legbar sind.
	 * @param color Farbe der Karten die ueberprueft werden sollen.
	 * @return true, wenn alle Karten >11 abgelegt werden koennen, sonst false
	 */
	private boolean checkAllInHigh(final Color color) {
		int temp;
		int i = 0;
		ArrayList<Card> cards = player.get(activePlayer).getCards(color);
		int max = cards.size() - 1;

		if (max + 1 != 0) {
			while (i <= max) {
				if (cards.get(i).getNumber() == field.getHighestCard(color)
						.getNumber() + 1) {
					temp = i;
					if (i == max) {
						return true;
					} else {
						for (int x = i; x < max; x++) {
							if (cards.get(temp + 1).getNumber() ==
cards.get(
temp).getNumber() + 1) {
```

```
                                                temp++;
                                                i++;
                                                if (temp == max) {
                                                        return true;
                                                }
                                        } else {
                                                return false;
                                        }
                                }
                        }
                } else {
                        i++;
                }
        }
        return false;
} else {
        return true;
}
}

/**
 * Prueft ob alle Karten auf der Hand (<11) legbar sind.
 * @param color Farbe der Karten die ueberprueft werden sollen.
 * @return true, wenn alle Karten <11 abgelegt werden koennen, sonst false
 */
private boolean checkAllInLow(final Color color) {
        int temp;
        ArrayList<Card> cards = player.get(activePlayer).getCards(color);
        int i = cards.size() - 1;
        int max = 0;

        if (i + 1 != 0) {
                while (i >= max) {
                        if (cards.get(i).getNumber() == field.getLowestCard(color)
                                        .getNumber() - 1) {
                                temp = i;
                                if (i == max) {
                                        return true;
                                } else {
                                        for (int x = i; x > max; x--) {
                                                if (cards.get(temp - 1).getNumber() ==
cards.get(
                                                                temp).getNumber() - 1) {
                                                temp--;
                                                i--;
                                                if (temp == max) {
                                                        return true;
                                                }
                                        } else {
```

```
                                                return false;
                                        }
                                }
                        }
                } else {
                        i--;
                }
        }
        return false;
    } else {
        return true;
    }
}

/**
 * Prueft ob der aktive Spieler noch einen Zug ausfuehren kann, oder ob er
 * noch Ziehen oder Legen kann.
 * @return true, wenn noch Karten auf dem Stapel sind, oder wenn ein Zug
 *         moeglich ist, sonst false.
 */
private boolean checkForPossibleMove() {
    if (!stack.getCards().isEmpty()) {
        return true;
    }
    if (field.getHighestCard(Color.BLUE) != null) {
        for (Card k : player.get(activePlayer).getCards(Color.BLUE)) {
            if (checkMove(k, field)) {
                return true;
            }
        }
    }
    if (field.getHighestCard(Color.GREEN) != null) {
        for (Card k : player.get(activePlayer).getCards(Color.GREEN)) {
            if (checkMove(k, field)) {
                return true;
            }
        }
    }
    if (field.getHighestCard(Color.ORANGE) != null) {
        for (Card k : player.get(activePlayer).getCards(Color.ORANGE)) {
            if (checkMove(k, field)) {
                return true;
            }
        }
    }
    if (field.getHighestCard(Color.RED) != null) {
        for (Card k : player.get(activePlayer).getCards(Color.RED)) {
            if (checkMove(k, field)) {
                return true;
```

```
                        }
                    }
                }
                return false;
        }

        /**
         * Prueft ob der aktive Spieler gewonnen hat.
         */
        private void checkWinner() {
                if (player.get(activePlayer).getCards().size() == 0) {
                        winner = player.get(activePlayer).getName();
                        exit();
                }
        }

        /**
         * Prueft ob der aktive Spieler eine Elf auf der Hand hat, wenn ja wird die
         * Elf automatisch auf das Spielfeld gelegt.
         * @return true, wenn Spieler eine Elf auf der Hand hat, sonst false.
         */
        private boolean checkElevenOnHand() {
                for (Card k : player.get(activePlayer).getCards()) {
                        if (k.getNumber() == ELEVEN) {
                                if (activePlayer == 0) {
                                        System.out
                                                        .println("Zug nicht moeglich! Elf wird
automatisch gelegt!");
                                }
                                move(k, field);
                                player.get(activePlayer).remove(k);
                                return true;
                        }
                }

                return false;
        }

        /**
         * Methode um eine Karte einem neuen Holder zu ueberschreiben.
         * @param card die Karte die verschoben werden soll.
         * @param target Holder an den die Karte geht.
         * @return true wenn der Zug erfolgreich ist. Sonst false (Karte null).
         */
        private boolean move(final Card card, final Holder target) {
                if (card != null) {
                        if (checkMove(card, target)) {
                                target.add(card);
                                return true;
```

```
                }
            }
            return false;
        }

        /**
         * Initialisiert das Spiel.
         */
        private void gameInit() {
            player.add(new Player("Layer 8"));

            for (int y = 1; y <= countKi; y++) {
                player.add(new AI("KI " + y, difficulty));
            }

            cardInit();
            gameStart();
        }

        /**
         * Der Stapel wird gemischt, danach werden jeweils 11 Karten an die Spieler
         * verteilt. Der Spieler mit einer 11 beginnt.
         */
        private void gameStart() {
            stack.shuffle();

            // Verteilen von je 11 Karten an jeden Spieler
            for (int i = 0; i < ELEVEN; i++) {
                for (int y = 0; y <= countKi; y++) {
                    move(stack.getNext(), player.get(y));
                }
            }

            // Sucht den ersten Spieler mit einer 11 raus und setzt ihn als aktiven
            // Spieler
            for (int i = 0; i < ELEVEN; i++) {
                for (int j = 0; j <= countKi; j++) {
                    if (player.get(j).getCards().get(i).getNumber() == ELEVEN) {
                        activePlayer = j;
                        // Beenden der Schleifen bei gefundener 11
                        i = ELEVEN + 1;
                        j = countKi + 1;
                    } else {
                        activePlayer = 0;
                    }
                }
            }

            startRound();
```

```
	}

	/**
	 * Startet eine neue Runde.
	 */
	private void startRound() {
		if (isRunning && activePlayer > 0) {
			player.get(activePlayer).react(this);
		}
	}

	/**
	 * Karten werden erstellt, danach zum Stapel hinzugefuegt.
	 */
	private void cardInit() {
		for (int number = 1; number <= 20; number++) {
			Card card = new Card(Color.BLUE, number);
			stack.add(card);
		}
		for (int number = 1; number <= 20; number++) {
			Card card = new Card(Color.GREEN, number);
			stack.add(card);
		}
		for (int number = 1; number <= 20; number++) {
			Card card = new Card(Color.ORANGE, number);
			stack.add(card);
		}
		for (int number = 1; number <= 20; number++) {
			Card card = new Card(Color.RED, number);
			stack.add(card);
		}
	}

	/**
	 * Prueft ob der aktuelle Zug gueltig ist.
	 * @param card die Karte die verschoben werden soll.
	 * @param target Holder an den die Karte geht.
	 * @return true wenn ziel kein Spielfeld ist, Farbe und Nummer stimmen.
	 *         Sonst false.
	 */
	private boolean checkMove(final Card card, final Holder target) {
		if (card.getNumber() == ELEVEN)
			return true;

		boolean result = false;

		if (target.equals(field)) {
			for (int i = 0; i < Color.values().length; i++) {
				if (card.getColor().equals(Color.values()[i])) {
```

```
                                        if (card.getNumber() == target.getHighestCard(
                                                Color.values()[i]).getNumber() + 1) {
                                            result = true;
                                        } else if (card.getNumber() ==
target.getLowestCard(
                                                Color.values()[i]).getNumber() - 1) {
                                            result = true;
                                        }
                                    }
                                }
                            } else {
                                result = true;
                            }

                            return result;
                        }
}
```

### 1.3. AI

```
package data;

/** Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

import java.util.ArrayList;

import control.Game;

public class AI extends Player {
        /** Schwierigkeit der KI. */
        private final boolean difficult;

        /**
         * Erstellt eine KI.
         * @param name Name der KI.
         * @param difficult Schwierigkeit der KI.
         */
        public AI(final String name, final boolean difficult) {
                setName(name);
                this.difficult = difficult;
        }

        /**
         * Reaktion der KI.
         * @param spiel Das aktuelle Spiel.
```

```
	 */
	public void react(final Game spiel) {
		if (!spiel.getStack().getCards().isEmpty()) {
			spiel.pull();
		} else if (!difficult) {
			easy(spiel);
		} else {
			hard(spiel);
		}
	}

	/**
	 * Die KI als leichter Gegner.
	 * @param game Das Spiel.
	 */
	private void easy(final Game game) {
		for (int i = 0; i < Color.values().length; i++) {
			// Fuer Jede Farbe wird geprueft ob auf der Hand eine Karte
groesser
			// oder kleiner als die auf dem Spielfeld vorhanden ist. Diese
wird
			// dann gelegt.
			if (game.getField().getHighestCard(Color.values()[i]) != null) {
				int hiNum =
game.getField().getHighestCard(Color.values()[i])
						.getNumber() + 1;
				if (cardAvailable(Color.values()[i], hiNum)) {
					game.setMove(Color.values()[i], hiNum);
				}
				int loNum =
game.getField().getLowestCard(Color.values()[i])
						.getNumber() - 1;
				if (cardAvailable(Color.values()[i], loNum)) {
					game.setMove(Color.values()[i], loNum);
				}
			}
		}
		game.nextPlayer();
	}

	/**
	 * Die KI als schwerer Gegner.
	 * @param game Das Spiel.
	 */
	private void hard(final Game game) {
		if (!game.checkForAllIn()) {
			ArrayList<Card> shortestChainCards = shortestChain(game);
			// Legen der kuerzesten Kette sofern eine Karte nicht auf der
			// Backhand liegt
```

```
                        for (Card c : shortestChainCards) {
                                game.setMove(c.getColor(), c.getNumber());
                        }
                }
                game.nextPlayer();
        }

        /**
         * Prueft ob eine bestimmte Karte auf der Hand verfuegbar ist.
         * @param color Die Farbe der gesuchten Karte.
         * @param number Die Nummer der gesuchten Karte.
         * @return true wenn die Karte vorhanden ist, sonst false.
         */
        private boolean cardAvailable(final Color color, final int number) {
                for (Card k : getCards()) {
                        // Pruefen auf alle Suchparameter
                        if (k.getColor().equals(color) && k.getNumber() == number) {
                                return true;
                        }
                }
                return false;
        }

        /**
         * Gibt die kuerzeste Kartenkette zurueck die auf der Hand liegt und auf dem
         * Spielfeld gelegt werden kann.
         * @param game Das Spiel.
         * @return ArrayList mit den Karten der kuersesten Kette.
         */
        private ArrayList<Card> shortestChain(final Game game) {
                ArrayList<Card> result = new ArrayList<Card>();
                ArrayList<Card> tempLo = new ArrayList<Card>();
                ArrayList<Card> tempHi = new ArrayList<Card>();
                Card tempCard;
                result.addAll(this.getCards());

                for (int i = 0; i < Color.values().length; i++) {
                        Color c = Color.values()[i];

                        Card hiCard = game.getField().getHighestCard(c);
                        if (hiCard != null) {
                                // Suche nach allen Karten < 11 die an das Spielfeld
passen
                                if (cardAvailable(c, game.getField().getHighestCard(c)
                                                .getNumber() + 1)) {
                                        // Rueckwaerts zusamenfuegen der Karten solange
keine
                                        // Unterbrechung der Kette vorhanden ist
                                        int j = getCards().indexOf(
```

```
                                                getCard(c,
game.getField().getHighestCard(c)
                                                        .getNumber() + 1));
                                do {
                                        tempCard = getCards().get(j++);
                                        tempHi.add(tempCard);
                                } while (cardAvailable(c, tempCard.getNumber() +
1));
                        }
                }

                Card loCard = game.getField().getLowestCard(c);
                if (loCard != null) {
                        // Suche nach allen Karten > 11 die an das Spielfeld
passen
                        if (cardAvailable(c, game.getField().getLowestCard(c)
                                        .getNumber() - 1)) {
                                // Vorwaerts zusamenfuegen der Karten solange
keine
                                // Unterbrechung der Kette vorhanden ist
                                int j = getCards().indexOf(
                                                getCard(c,
game.getField().getLowestCard(c)
                                                                .getNumber() - 1));
                                do {
                                        tempCard = getCards().get(j--);
                                        tempLo.add(tempCard);
                                } while (cardAvailable(c, tempCard.getNumber() -
1));
                        }
                }

                // Wenn keine HiKarten vorhanden sind
                int tempHiSize = tempHi.size() == 0 ? tempLo.size() + 1 : tempHi
                                .size();
                // Zuweisen des kleinsten Stapels
                if (tempLo.size() > 0 && tempHiSize > tempLo.size()
                                && result.size() > tempLo.size()) {
                        result.clear();
                        result.addAll(tempLo);
                } else if (tempHi.size() > 0 && result.size() > tempHi.size()) {
                        result.clear();
                        result.addAll(tempHi);
                }

                tempHi.clear();
                tempLo.clear();
        }
```

```
                // Leeren des Ergebnisses wenn keine Kette gefunden wurde
                if (result.size() == getCards().size()) {
                        result = new ArrayList<Card>();
                }

                return result;
        }
}
```

### 1.4. Card

```
package data;

/**
 * Hochschule Hamm-Lippstadt Praktikum Informatik II (ElferRaus) (C) 2015 Lara
 * Sievers, Adrian Schmidt, Fabian Schneider 22.05.2015
 */

public class Card {
        /** Farbe der Karte. */
        private final Color color;
        /** Nummer der Karte. */
        private final int number;

        /**
         * Erstellt eine Karte.
         * @param color Farbe der Karte.
         * @param number Nummer der Karte.
         */
        public Card(final Color color, final int number) {
                this.color = color;
                this.number = number;
        }

        /**
         * Getter fuer die Farbe der Karte.
         * @return farbe Farbe der Karte.
         */
        public Color getColor() {
                return color;
        }

        /**
         * Getter fuer die Nummer
         * @return nummer Nummer der Karte.
         */
        public int getNumber() {
                return number;
        }
```

```
	/**
	 * Gibt die farbe und die Nummer der Karte als String zurueck.
	 */
	public String toString() {
		return color + " " + number + "\n";
	}

}
```

### 1.5. Color

```
package data;

/**
 * Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

/**
 * Fuer das Spiel verfuegbare Farben.
 */
public enum Color {
	BLUE, GREEN, ORANGE, RED
}
```

### 1.6. Field

```
package data;

/** Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

/**
 * Das Spielfeld auf dem die Karten abgelegt werden. Kann nur Karten aufnehmen
 * und nicht mehr abgeben.
 */
public class Field extends Holder {
	/** Enthaelt die Stapel Instanz. */
	private static Field field;

	/** Ctor fuer das Spielfeld. */
	private Field() {
	}
```

```
	/**
	 * Getter fuer die Spielfeld Instanz.
	 * @return Gibt die Spielfeld Instanz zurueck.
	 */
	public static Field getInstance() {
		if (field == null) {
			field = new Field();
		}
		return field;
	}

	/**
	 * Inaktive Methode.
	 */
	public void remove(final Card card) {
		// Leer, damit nicht geloescht werden kann.
	}
}
```

### 1.7. Game

```
package data;

import java.util.ArrayList;

import control.Game;

/** Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

/**
 * Holder-Klasse definiert die Grundfunktionen jedes Kartenhalters.
 */
public abstract class Holder {
	/** Name des Holders. */
	private String name;

	/** Karten im Besitz des Holders. */
	private final ArrayList<Card> cards = new ArrayList<Card>();

	/**
	 * Getter fuer den Namen des Holders
	 * @return name Name des Holders
	 */
	public String getName() {
```

```
		return name;
	}

	/**
	 * Setter fuer den Namen
	 * @param name Zukuenftiger Name des Holders
	 */
	public void setName(final String name) {
		this.name = name;
	}

	/**
	 * Reaktion des Holders.
	 * @param spiel Das aktuelle Spiel.
	 */
	public void react(final Game spiel) {
	}

	/**
	 * Gibt die kleinste Karte unterhalb von 12 der gesuchten Farbe zurueck.
	 * @param color Die Farbe (BLUE, GREEN, ORANGE, RED) fuer welche die
	 *            kleinste Karte gesucht wird.
	 * @return Die kleinste Karte der Farbe, sofern kleiner gleich 11. Sonst
	 *         null.
	 */
	public Card getLowestCard(final Color color) {
		Card lowestCard = null;

		for (Card k : cards) {
			if (k.getColor().equals(color) && k.getNumber() < 12) {
				// Sofern noch keine Karte gesetzt wurde passt der erste
Treffer
				// immer
				if (lowestCard == null) {
					lowestCard = k;
				}
				// Sofern eine kleinere Karte gefunden ist, wird sie als
				// Kleinste gesetzt
				else if (lowestCard.getNumber() > k.getNumber()) {
					lowestCard = k;
				}
			}
		}

		return lowestCard;
	}

	/**
	 * Gibt die kleinste Karte unterhalb von 12 der gesuchten Farbe zurueck.
```

```
	 * @param cards Die ArrayList in der die kleinste Karte gesucht werden soll.
	 * @param color Die Farbe (BLUE, GREEN, ORANGE, RED) fuer welche die
	 *          kleinste Karte gesucht wird.
	 * @return Die kleinste Karte der Farbe, sofern kleiner gleich 11. Sonst
	 *         null.
	 */
	public Card getLowestCard(final ArrayList<Card> cards, final Color color) {
		Card lowestCard = null;

		for (Card k : cards) {
			if (k.getColor().equals(color) && k.getNumber() < 12) {
				// Sofern noch keine Karte gesetzt wurde passt der erste
Treffer
				// immer
				if (lowestCard == null) {
					lowestCard = k;
				}
				// Sofern eine kleinere Karte gefunden ist, wird sie als
				// Kleinste gesetzt
				else if (lowestCard.getNumber() > k.getNumber()) {
					lowestCard = k;
				}
			}
		}

		return lowestCard;
	}

	/**
	 * Gibt die hoechste Karte oberhalb von 10 der gesuchten Farbe zurueck.
	 * @param color Die Farbe (BLUE, GREEN, ORANGE, RED) fuer welche die
	 *          hoechste Karte gesucht wird.
	 * @return Die hoechste Karte der Farbe, sofern groesser gleich 11. Sonst
	 *         null.
	 */
	public Card getHighestCard(final Color color) {
		Card highestCard = null;

		for (Card k : cards) {
			if (k.getColor().equals(color) && k.getNumber() > 10) {
				// Sofern noch keine Karte gesetzt wurde passt der erste
Treffer
				// immer
				if (highestCard == null) {
					highestCard = k;
				}
				// Sofern eine hoehere Karte gefunden ist, wird sie als
Hoechste
				// gesetzt
```

```
                        else if (highestCard.getNumber() < k.getNumber()) {
                                highestCard = k;
                        }
                }
        }

        return highestCard;
}

/**
 * Gibt die hoechste Karte oberhalb von 10 der gesuchten Farbe zurueck.
 * @param cards Die ArrayList in der die hoechste Karte gesucht werden soll.
 * @param color Die Farbe (BLUE, GREEN, ORANGE, RED) fuer welche die
 *          hoechste Karte gesucht wird.
 * @return Die hoechste Karte der Farbe, sofern groesser gleich 11. Sonst
 *         null.
 */
public Card getHighestCard(final ArrayList<Card> cards, final Color color) {
        Card highestCard = null;

        for (Card k : cards) {
                if (k.getColor().equals(color) && k.getNumber() > 10) {
                        // Sofern noch keine Karte gesetzt wurde passt der erste
Treffer
                        // immer
                        if (highestCard == null) {
                                highestCard = k;
                        }
                        // Sofern eine hoehere Karte gefunden ist, wird sie als
Hoechste
                        // gesetzt
                        else if (highestCard.getNumber() < k.getNumber()) {
                                highestCard = k;
                        }
                }
        }

        return highestCard;
}

/**
 * Getter fuer eine bestimmte Karte.
 * @param color Farbe der gesuchten Karte.
 * @param number Nummer der gesuchten Karte.
 * @return Die Karte zu den gesuchten Parametern, sonst null.
 */
public Card getCard(final Color color, final int number) {
        for (Card k : getCards()) {
                // Pruefen auf alle Suchparameter
```

```
                if (k.getColor().equals(color) && k.getNumber() == number) {
                        return k;
                }
        }
        return null;
}

/**
 * Getter fuer die Karten des Holders.
 * @return Gibt die Karten des Holders zurueck.
 */
public ArrayList<Card> getCards() {
        return cards;
}

/**
 * Gibt alle Karten einer bestimmten Farbe zurueck.
 * @return ArrayList mit den Karten der gesuchten Farbe.
 */
public ArrayList<Card> getCards(final Color color) {
        ArrayList<Card> searchedColor = new ArrayList<Card>();
        for (Card k : cards) {
                if (k.getColor().equals(color)) {
                        searchedColor.add(k);
                }
        }
        return searchedColor;
}

/**
 * Fuegt eine neue Karte hinzu.
 * @param card karte die hinzugefuegt werden soll.
 */
public void add(final Card card) {
        cards.add(card);
        sort(cards);
}

/**
 * Entfernt eine Karte.
 * @param card Karte die entfernt werden soll.
 */
public void remove(final Card card) {
        cards.remove(card);
}

/**
 * Sortiert die Karten nach Farbe.
 */
```

```
	protected void sort(final ArrayList<Card> cards) {
		ArrayList<Card> blue = new ArrayList<Card>();
		ArrayList<Card> green = new ArrayList<Card>();
		ArrayList<Card> orange = new ArrayList<Card>();
		ArrayList<Card> red = new ArrayList<Card>();

		// Aufsplitten der Farben im Stapel
		for (Card k : cards) {
			if (k.getColor().equals(Color.BLUE)) {
				blue.add(k);
			} else if (k.getColor().equals(Color.GREEN)) {
				green.add(k);
			} else if (k.getColor().equals(Color.ORANGE)) {
				orange.add(k);
			} else {
				red.add(k);
			}
		}

		// Sortieren der Nummern
		sortNumber(blue);
		sortNumber(green);
		sortNumber(orange);
		sortNumber(red);

		cards.clear();

		// Zusammenfuegen der Farben in einen Stapel
		for (Card k : blue) {
			cards.add(k);
		}
		for (Card k : green) {
			cards.add(k);
		}
		for (Card k : orange) {
			cards.add(k);
		}
		for (Card k : red) {
			cards.add(k);
		}
	}

	/**
	 * Sortiert die Nummern einer Kartenfarbe aufsteigend.
	 * @param cards ArrayList der zu sortierenden Kartenfarbe.
	 */
	private void sortNumber(final ArrayList<Card> cards) {
		for (int i = 0; i < cards.size(); i++) {
			Card temp = cards.get(i);
```

```
                        int j = i;
                        while (j > 0 && cards.get(j - 1).getNumber() > temp.getNumber())
{
                                cards.add(j, cards.get(j - 1));
                                cards.remove(j + 1);
                                j--;
                        }
                        cards.add(j, temp);
                        cards.remove(j + 1);
                }
        }
}
```

### 1.8. Holder

```
package data;

/** Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

/**
 * Spieler ist ein Holder.
 */
public class Player extends Holder {

        /**
         * Erstellt einen neuen Spieler.
         */
        public Player() {

        }

        /**
         * Erstellt einen neuen Spieler.
         * @param Name des Spielers.
         */
        public Player(final String name) {
                setName(name);
        }
}
```

### 1.9. Stack

```
package data;

/** Hochschule Hamm-Lippstadt
```

```
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

import java.util.Collections;

/**
 * Stapel erweitert den Holder. Haelt die noch nicht ausgeteilten Karten.
 */
public class Stack extends Holder {
	/** Enthaelt die Stapel Instanz. */
	private static Stack stack;

	/**
	 * Ctor fuer den Stapel.
	 */
	private Stack() {
	}

	/**
	 * Getter fuer die Stapel Instanz.
	 * @return Gibt die Stapel Instanz zurueck.
	 */
	public static Stack getInstance() {
		if (stack == null) {
			stack = new Stack();
		}
		return stack;
	}

	/**
	 * Mischt den Stapel.
	 */
	public void shuffle() {
		Collections.shuffle(this.getCards());
	}

	/**
	 * Nimmt eine Karte vom Stapel und gibt diese zurueck. Die Karte wird auf
	 * dem Stapel geloescht.
	 * @return Unterste Karte vom Stapel. null bei leerem Stapel.
	 */
	public Card getNext() {
		Card karte = null;
		try {
			karte = this.getCards().get(0);
		} catch (IndexOutOfBoundsException e) {
			// e.printStackTrace();
```

```
                }
                this.remove(karte);
                return karte;
        }
}
```

## 1.10. View

```
/** Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
 * 22.05.2015
 */

package view;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

import control.Game;
import data.Color;
import data.Card;

/**
 * Stellt das Spiel auf Kommandozeile dar und wartet auf Befehle des Spielers.
 */
public class View implements ViewInterface {
        /** Hilfetext. */
        private static final String HELP_MESSAGE = "\r\n"
                        + "Um eine Karte zu ziehen geben Sie 'pull' ein.\r\n"
                        + "Um eine Karte zu legen geben Sie 'put', den Farbbuchstaben
und die Nummer ein. Bsp: put R9.\r\n"
                        + "Mit 'next' kann der Zug beendet werden."
                        + "Das Kommando 'allin' legt alle Karten sofern sie gelegt
werden koennen (gegen Spielende)."
                        + "Diese Hilfe kann mit 'help' angezeigt werden.\r\n"
                        + "Die Regeln erhalten Sie ueber den Befehl 'rules'.\r\n"
                        + "Und wenn Sie dann doch keine Lust mehr haben sagen Sie
'bye'.";
        /** Regeltext. */
        private static final String RULES = "\r\n"
                        + "Spielregeln:\r\n"
                        + "Ziel des Spiels ist es alle Karten abzulegen.\r\n"
                        + "Jeder Spieler erhaelt 11 Karten. Es gibt 4 Farben mit jeweils
20 Karten, die von 1-20 nummeriert sind.\r\n"
                        + "Karten duerfen nur auf Karten mit der gleichen Farbe auf
einen Stapel des Spielfelds gelegt werden.\r\n"
```

```
                        + "Bei jeder Farbe wird jeweils von 11 bis 1 ab- bzw. von 12 bis
20 aufwaerts gelegt\r\n"
                        + "Der Spieler mit einer 11 faengt an.\r\n"
                        + "Elfen muessen immer zuerst gelegt werden.\r\n"
                        + "Es ist nicht verpflichtend die anderen Karten zu legen,
solange noch Karten auf dem Stapel sind.\r\n"
                        + "Es koennen so viele Karten gelegt werden wie man moechte,
jedoch mindestens Eine sofern man kann.\r\n"
                        + "Es wird entweder gezogen oder gelegt, danach ist der
naechste Spieler dran.";
        /** Das Spiel. */
        private Game game;
        /** Anzahl der KIs. */
        private int countKi;
        /** Schwierigkeit der KIs. */
        private boolean difficulty;

        /**
         * Erstellt eine neue View fuer das Spiel. Muss an Spiel uebergeben werden.
         */
        public View() {
                System.out.println("ElferRaus");
                System.out.println(HELP_MESSAGE);
                System.out.println();

                do {
                        try {
                                countKi = Integer.parseInt(input("Anzahl der Gegner [1-3]:
"));
                        } catch (NumberFormatException e) {
                        }
                } while (countKi < 1 || countKi > 3);

                boolean result = false;
                do {
                        String input = input("Leicht oder schwer [easy / hard]: ")
                                        .toLowerCase();
                        if (input.equals("easy")) {
                                difficulty = false;
                                result = true;
                        } else if (input.equals("hard")) {
                                difficulty = true;
                                result = true;
                        }
                } while (!result);
        }

        /**
         * Getter fuer die Anzahl der KIs.
```

```
	 * @return Gibt die Anzahl der KIs zurueck.
	 */
	public int getCountKi() {
		return countKi;
	}

	/**
	 * Getter fuer die Schwierigkeit.
	 * @return Gibt die Schwierigkeit zurueck.
	 */
	public boolean getDifficulty() {
		return difficulty;
	}

	/**
	 * Aktualisiert das Spiel auf der Kommandozeile und wartet auf neue Befehle.
	 */
	@Override
	public void update(final Game game) {
		this.game = game;

		boolean result = true;
		do {
			System.out.println();
			for (int i = 1; i < game.getPlayer().size(); i++) {
				System.out.println("Karten von "
						+ game.getPlayer().get(i).getName() + ": "
						+ game.getPlayer().get(i).getCards().size() +
".");
			}

			System.out.println();
			printField();
			System.out.println();

			System.out.println("Karten auf dem Stapel: "
					+ game.getStack().getCards().size());
			System.out.println();
			System.out.println("Ihre Karten ("
					+ game.getPlayer().get(0).getCards().size() + "):");
			printPlayerCards(game.getPlayer().get(0).getCards());

			System.out.println();
			String[] input = input("Ihr Zug: ").split(" ");
			switch (input[0]) {
			case "help":
				System.out.println(HELP_MESSAGE);
				update(game);
				break;
```

```
                        case "rules":
                                System.out.println(RULES);
                                update(game);
                                break;
                        case "pull":
                                if (!game.pull()) {
                                        System.out.println("Keine Karten mehr auf dem
Stapel.");
                                }
                                break;
                        case "put":
                                try {
                                        result = move(input[1]);
                                } catch (Exception e) {
                                        System.out.println("Bitte geben Sie eine Karte
an.");
                                }
                                break;
                        case "next":
                                System.out.println("Bitte Warten.");
                                game.nextPlayer();
                                result = true;
                                break;
                        case "allin":
                                game.checkForAllIn();
                                break;
                        case "bye":
                        case "exit":
                                game.exit();
                                break;
                        default:
                                System.out.println("Ich verstehe die Eingabe nicht.");
                                update(game);
                                break;
                        }
                } while (!result);

                if (!game.isRunning()) {
                        System.out.println("Gewonnen hat: " + game.getWinner());
                }
        }

        /**
         * Gibt eine Nachricht auf der Kommandozeile aus und liesst eine Eingabe.
         * @param message Nachricht, welche ausgegeben werden soll.
         * @return String mit dem eingegebenen Text.
         */
        private String input(final String message) {
                System.out.print(message);
```

```
		String input = "";
		try {
			input = new BufferedReader(new InputStreamReader(System.in))
					.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
		return input.toLowerCase();
	}

	private void printField() {
		System.out.println("Spielfeld:");
		int[] blue = getNumbers(Color.BLUE);
		int[] orange = getNumbers(Color.ORANGE);
		int[] green = getNumbers(Color.GREEN);
		int[] red = getNumbers(Color.RED);

		System.out.print("Blau   " + blue[0] + "|" + blue[1]);
		System.out.println();
		System.out.print("Gruen  " + green[0] + "|" + green[1]);
		System.out.println();
		System.out.print("Orange " + orange[0] + "|" + orange[1]);
		System.out.println();
		System.out.print("Rot    " + red[0] + "|" + red[1]);
		System.out.println();
	}

	/**
	 * Gibt die Nummern der Karten auf dem Spielfeld fuer eine Farbe zurueck.
	 * @param color Die Farbe der Karten fuer die die Nummern gesucht sind.
	 * @return Ein int Array mit der niedrigsten, der elf und der hoechsten
	 *         Nummer pro Farbe auf dem Spielfeld.
	 */
	private int[] getNumbers(final Color color) {
		int[] result = new int[2];
		// Wert der niedrigsten Karte
		try {
			result[0] = game.getField().getLowestCard(color).getNumber();
		} catch (NullPointerException e) {
			result[0] = 0;
		}

		// Wert der hoechsten Karte
		try {
			result[1] = game.getField().getHighestCard(color).getNumber();
		} catch (NullPointerException e) {
			result[1] = 0;
		}
```

```
		return result;
	}

	private void printPlayerCards(final ArrayList<Card> cards) {
		String output = "";
		if (cards.size() > 0) {
			Card temp = cards.get(0);
			String[] farbe = { "B", "G", "O", "R" };

			boolean firstRound = true;
			for (Card k : cards) {
				if (firstRound) {
					for (int i = 0; i < farbe.length; i++) {
						if (k.getColor().equals(Color.values()[i])) {
							output += farbe[i];
						}
					}
					firstRound = false;
				}
				if (!temp.getColor().equals(k.getColor())) {
					for (int i = 0; i < farbe.length; i++) {
						if (k.getColor().equals(Color.values()[i])) {
							output += "\r\n" + farbe[i];
						}
					}
				}
				output += " " + k.getNumber();
				temp = k;
			}
		}

		System.out.println(output);
	}

	/**
	 * Prueft den gewuenschsten Zug vor und laesst ihn ausfuehren.
	 * @param card Kartencode fuer die Karte die bewegt werden soll. Bsp: r9
	 * @return true wenn der Zug durchgefuehrt wurde. Sonst false.
	 */
	private boolean move(final String card) {
		String colorCode = card.substring(0, 1);
		int number;

		// Konvertieren der Nummer
		try {
			number = Integer.parseInt(card.substring(1));
		} catch (NumberFormatException e) {
			System.out.println("Keine gueltige Zahl eingegeben.");
			return false;
```

```
                }

                // Check ob Nummer im gueltigen Bereich liegt
                if (number < 1 && number > 20) {
                        System.out.println("Zahl nicht im gueltigen Wertebereich.");
                        return false;
                }

                // Zuweisen der Farbe passend zum Code
                Color color = null;

                switch (colorCode) {
                case "r":
                        color = Color.RED;
                        break;
                case "g":
                        color = Color.GREEN;
                        break;
                case "b":
                        color = Color.BLUE;
                        break;
                case "o":
                        color = Color.ORANGE;
                        break;

                default:
                        System.out
                                        .println("Falscher Farbcode. Moeglich sind: Rot,
Gruen, Blau, Orange");
                        return false;
                }

                // Ausfuehren des Zugs
                if (game.setMove(color, number)) {
                        return true;
                } else {
                        System.out.println("Zug nicht moeglich!");
                        return false;
                }
        }
}
```

## 1.11. ViewInterface

```
package view;

/** Hochschule Hamm-Lippstadt
 * Praktikum Informatik II (ElferRaus)
 * (C) 2015 Lara Sievers, Adrian Schmidt, Fabian Schneider
```

```
 * 22.05.2015
 */

import control.Game;

/**
 * Interface der View. Gibt die Schnittstelle zur control vor.
 */
public interface ViewInterface {

	/**
	 * Aktualisiert die View auf den momentanen Spielstatus.
	 * @param game Das Spiel welches update aufruft.
	 */
	public void update(final Game game);
}
```