Programmentwurf
UNO

Name: Rehan, Richard
Martrikelnummer: 3549839

Abgabedatum: 31.05.2023

Allgemeine Anmerkungen:

- *Gesamt-Punktzahl: 60P (zum Bestehen mit 4,0 werden 30P benötigt)*
- *die Aufgabenbeschreibung (der blaue Text) und die mögliche Punktanzahl muss im Dokument erhalten bleiben*
- *es darf nicht auf andere Kapitel als alleiniger Leistungsnachweis verwiesen werden (z.B. in der Form "XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung")*
- *alles muss in UTF-8 codiert sein (Text und Code)*
- *das Dokument muss als PDF abgegeben werden*
- *es gibt keine mündlichen Nebenabreden / Ausnahmen – alles muss so bearbeitet werden, wie es schriftlich gefordert ist*
- *alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)*
- *die Beispiele sollten wenn möglich vom aktuellen Stand genommen werden*
  - *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
  - *Ausnahme: beim Kapitel "Refactoring" darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*
- *falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden*
  - ***Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele sondern 0,5P Abzug für das fehlende Negativ-Beispiel***
  - *Beispiel*
    - *"Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt." (2P)*
      - *Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]*
      - *Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: 2P ODER falls im Code mind. eine Klasse SRP verletzt: 0,5P*
- *verlangte Positiv-Beispiele müssen gebracht werden – im Zweifel müssen sie extra für die Lösung der Aufgabe implementiert werden*
- *Code-Beispiel = Code in das Dokument kopieren (inkl. Syntax-Highlighting)*
- *falls Bezug auf den Code genommen wird: entsprechende Code-Teile in das Dokument kopieren (inkl Syntax-Highlighting)*

- *bei UML-Diagrammen immer die öffentlichen Methoden und Felder angegeben – private Methoden/Felder nur angeben, wenn sie zur Klärung beitragen*
- *bei UML-Diagrammen immer unaufgefordert die zusammenspielenden Klassen ergänzen, falls diese Teil der Aufgabe sind*
- *Klassennamen/Variablennamen/etc im Dokument so benennen, wie sie im Code benannt sind (z.B. im Dokument nicht anfangen, englische Klassennamen zu übersetzen)*
- *die Aufgaben sind von vorne herein bekannt und müssen wie gefordert gelöst werden – z.B. ist es keine Lösung zu schreiben, dass es das nicht im Code gibt*
    - *Beispiel 1*
        - *Aufgabe: Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten*
        - *Antwort: Es wurden keine Fake/Mock-Objekte gebraucht.*
        - *Punkte: 0P*
    - *Beispiel 2*
        - *Aufgabe: UML, Beschreibung und Begründung des Einsatzes eines Repositories*
        - *Antwort: Die Applikation enthält kein Repository*
        - *Punkte*
            - *falls (was quasi nie vorkommt) die Fachlichkeit tatsächlich kein Repository hergibt: volle Punktzahl*
            - *falls die Fachlichkeit in irgendeiner Form ein Repository hergibt (auch wenn es nicht implementiert wurde): 0P*
    - *Beispiel 3*
        - *Aufgabe: UML von 2 implementierte unterschiedliche Entwurfsmuster aus der Vorlesung*
        - *Antwort: es wurden keine Entwurfsmuster gebraucht/implementiert*
        - *Punkt: 0P*

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

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

Die Applikation soll das Kartenspiel UNO in einer Kommandozeile umsetzen. Es gibt sowohl menschliche Spieler als auch Computergegner. Es gibt auch 2 Spielmodi: Standard und Time. Dabei wird sowohl der Stapel in der Mitte als auch die eigenen Karten angezeigt, von denen man eine Karte zum Spielen auswählen kann. Es implementiert die Standard UNO Regeln. Der erste, der keine Karten mehr hat, gewinnt.

Starten der Applikation (1P)

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

- Java (JRE) 17 installieren
- Maven installieren
- Terminal/CMD im Root-Verzeichnis vom Repository öffnen
- Im Terminal:
    - **Linux**
        - erst das Skript ausführbar machen: *chmod +x run.sh*
        - *run.sh*
    - **Windows**
        - *./run.sh*

Technischer Überblick (2P)

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

Java – bekannte Programmiersprache, plattformunabhängig durch die JVM, persönliche Erfahrungen durch andere Vorlesungen

Maven – Buildtools, vereinfacht und organsiert den gesamten Buildprozess, erlaubt durch einfache Kommandos beispielsweise das Projekt zu kompilieren, zu testen oder aber auch zu bereinigen (heißt kompilierten Code zu entfernen)

JUnit 5 – Unit Testing Bibliothek, die das Unit Testing ermöglicht, speziell für Java

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

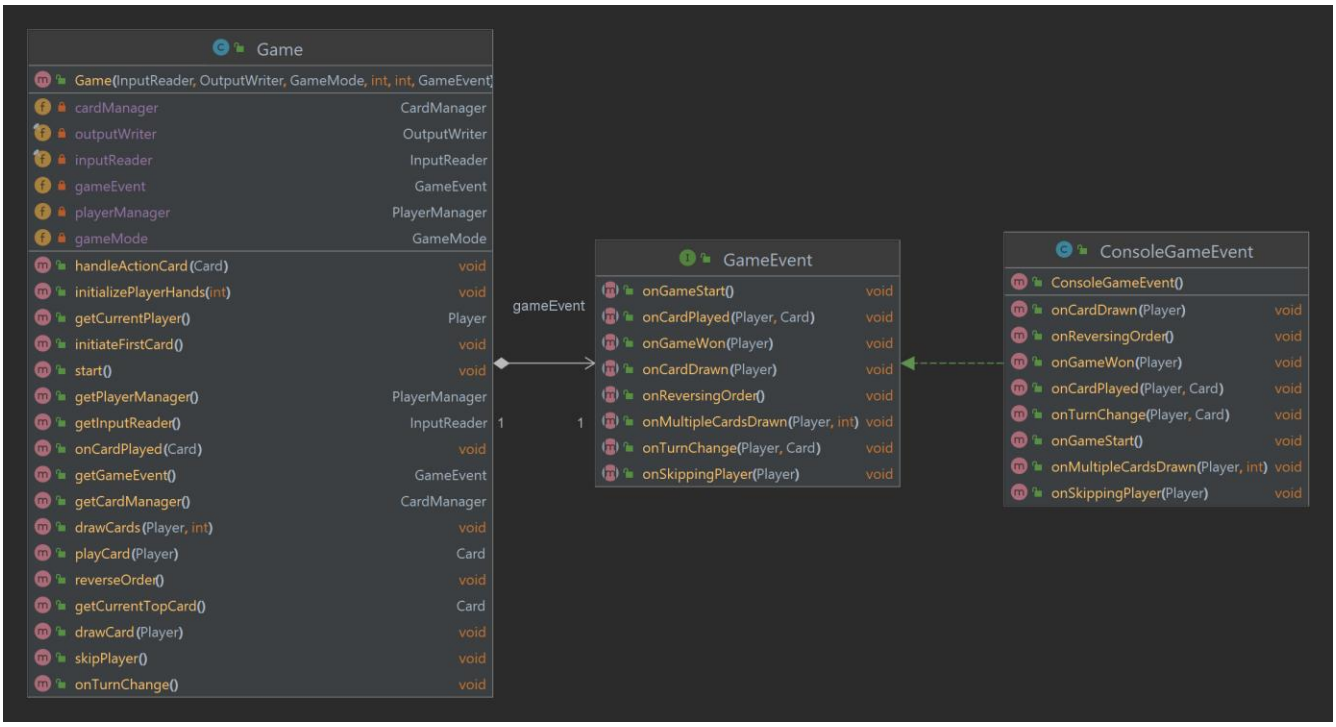
[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

Die Clean Architecture ist ein Prinzip zur Strukturierung von Software auf eine Art und Weise, die ihre Verständlichkeit, Flexibilität und Wartbarkeit maximiert. Der Hauptgedanke ist, dass die Software in mehrere Schichten aufgeteilt ist, wobei jede Schicht von innen nach außen auf die Schichten davor abhängt. Auf diese Weise entsteht eine klare Trennung von Aufgaben.

Analyse der Dependency Rule (3P)

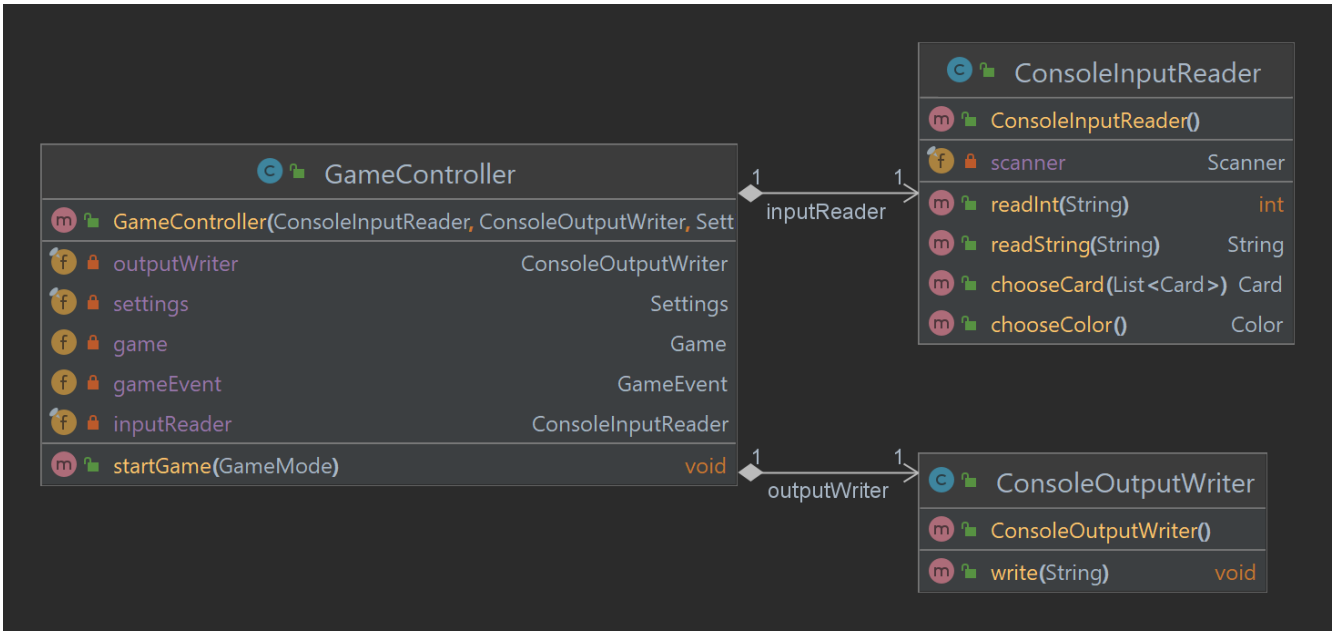
[1 Klasse, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

Positiv-Beispiel: Dependency Rule



Das Interface *GameEvent* ist dazu da, um bei spielspezifischen Events wie dem Kartenspielen oder wenn eine Karte gezogen wird eine Ausgabe in dem User Interface zu machen. Es gibt eine Implementation davon, welche speziell für Konsolenausgaben existiert. Die Game Klasse ist von dem *GameEvent* Interface abhängig, aber nicht von der Implementierung, wodurch die Dependency Rule eingehalten wird, weil die Abhängigkeiten die richtige Richtung hat. *ConsoleGameEvent* liegt in der Adapters-Schicht (UI-Schicht), und die *Game* Klasse in der Domain Schicht. Außerdem gibt es im *GameEvent* nur Übergabeparameter aus der Domain-Schicht.

Negativ-Beispiel: Dependency Rule

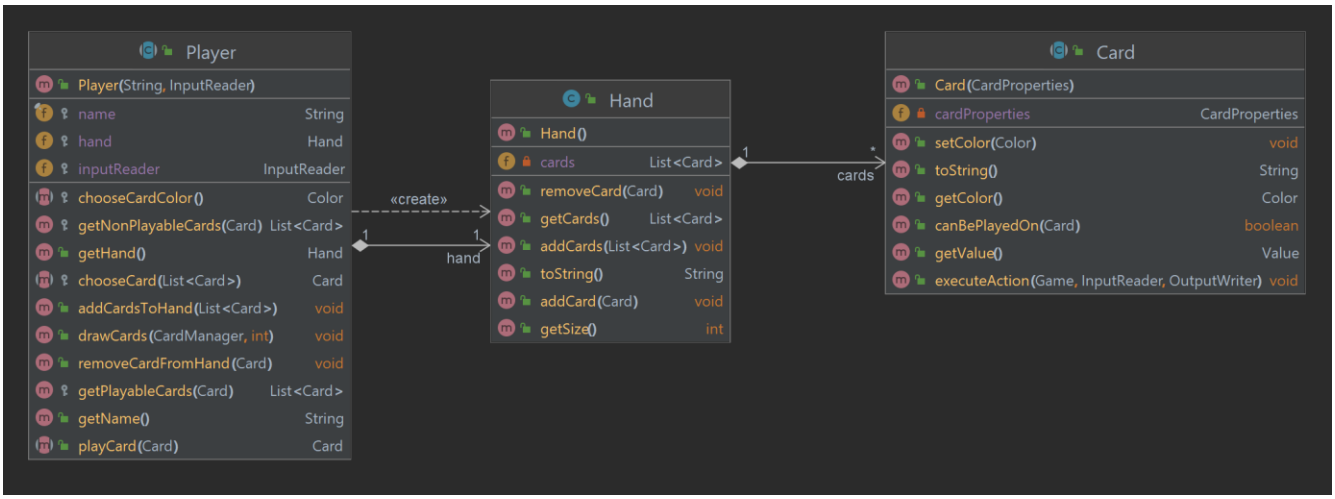


Die *GameController* Klasse hat hingegen Abhängigkeiten von den *ConsoleInputReader* sowie *ConsoleOutputWriter*. Der *GameController* liegt in der Application-Schicht, wohingegen die *ConsoleInputReader* sowie *ConsoleOutputWriter* in der Adapter-Schicht liegen. Somit verletzt es die Dependency Rule, weil die Abhängigkeit von *GameController* zu einer höheren Schicht liegt.

Analyse der Schichten (4P)

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

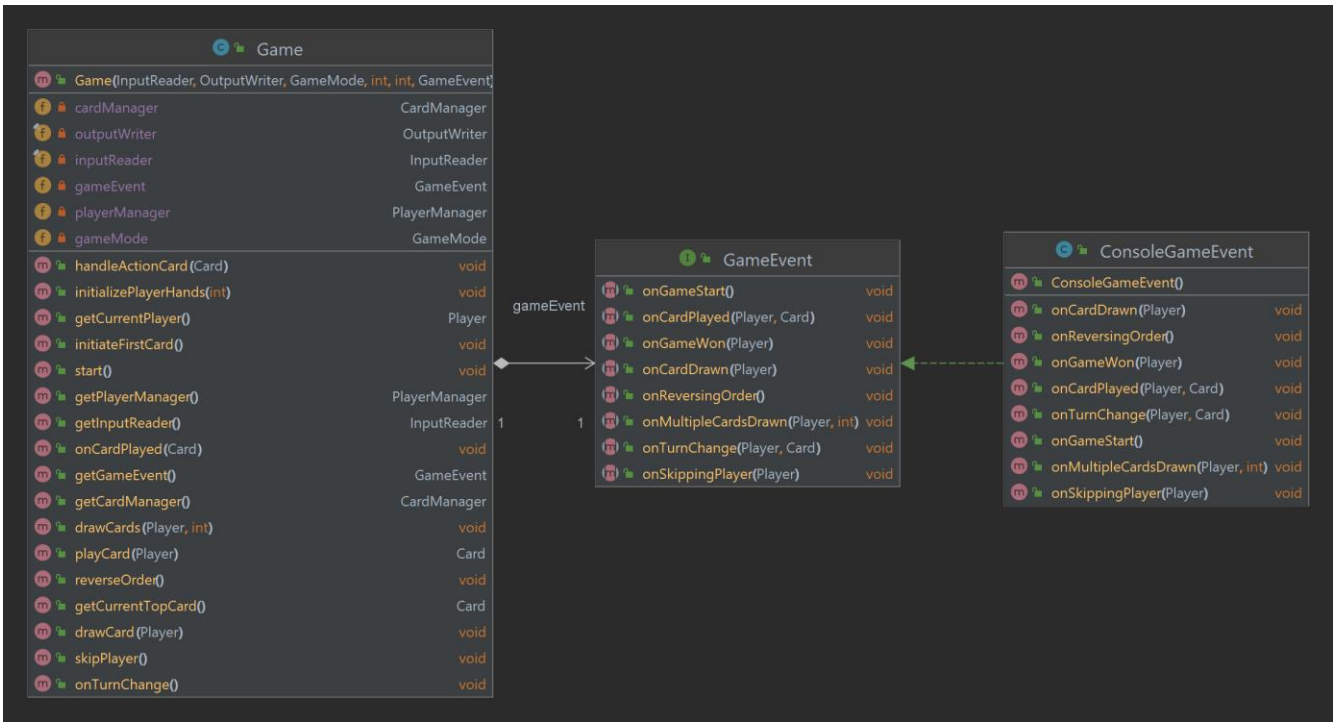
Schicht: Domain



Die *Hand* Klasse ist dafür zuständig, die Karten eines Spielers zu halten und zu verwalten. Die *Hand* ist grundlegender Bestandteil von Kartenspielen und somit auch von UNO. Daher

gehört diese Klasse in die Domain-Schicht, da sie Domänenspezifische Informationen und Aktionen enthält.

Schicht: Adapters



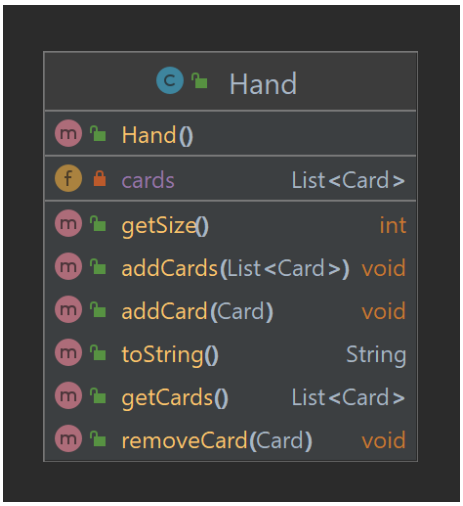
Die Klasse *ConsoleGameEvent* ist für das User Interface zuständig und gehört somit in die Adapterschicht (welche hier die UI Schicht übernimmt). In diesem Fall ist das eine konkrete Implementation vom Interface *GameEvent*, welches für jegliche Events beim Spielen Methoden anbietet, wie beispielsweise, wenn eine Karte gezogen oder ein Spiel gestartet wird. Diese Methoden werden von der *Game* Klasse ausgeführt und die nötigen Parameter werden gegeben. Die Klasse *ConsoleGameEvent* hat die Aufgabe, die gewollten Informationen in der Konsole auszugeben. Ähnlich könnte eine andere Implementierung von *GameEvent* eine GUI nutzen und dabei noch irgendwelche Effekte und Animationen abspielen, die bei einem Event sein sollen.

Kapitel 3: SOLID (8P)

Analyse SRP (3P)

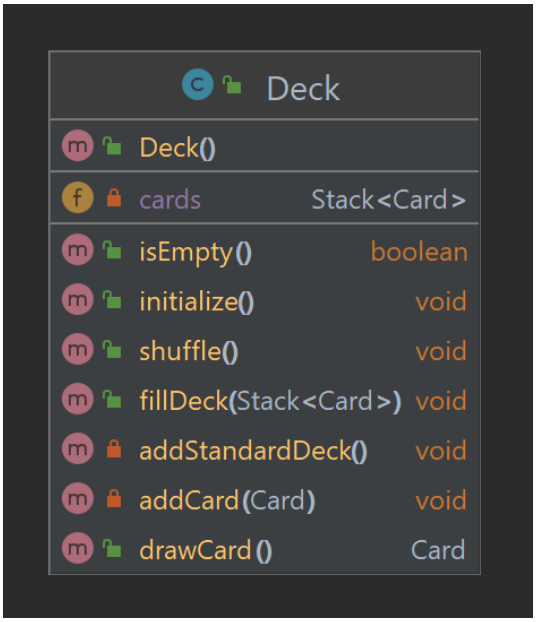
[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

Positiv-Beispiel



Die *Hand* Klasse ist nur für die Verwaltung der Karten eines Spielers zuständig. Es besitzt die Karten als eine Liste *cards* und bietet Methoden zum Hinzufügen und entfernen von Karten an. Außerdem gibt es noch Getter-Methoden und eine eigene *toString()* Methode. Die Klasse hält SRP ein, weil ihre einzige Verantwortung das Verwalten der Kartenliste *cards* ist.

Negativ-Beispiel



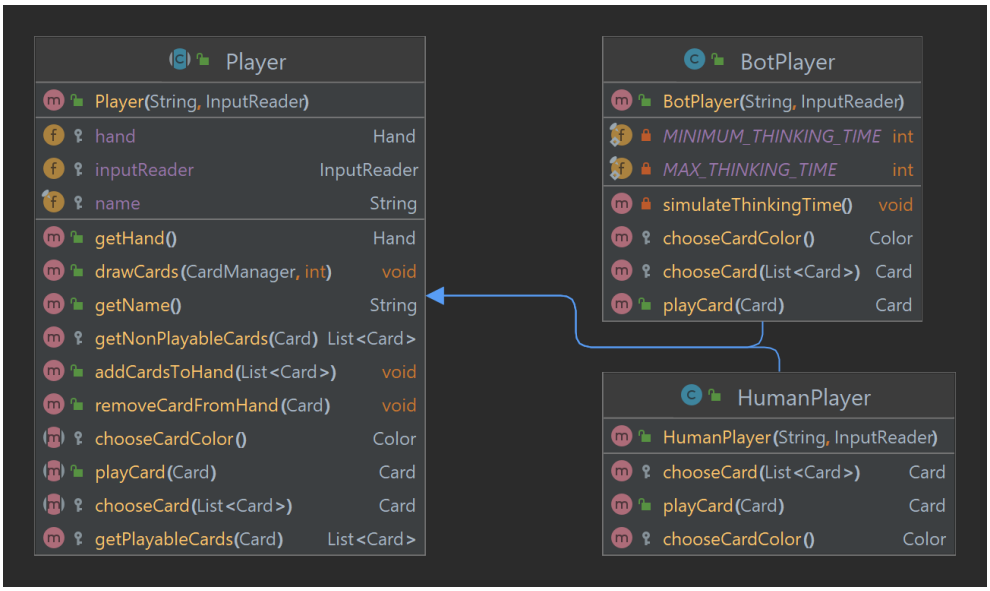
Die *Deck* Klasse ist zwar, ähnlich zur *Hand* Klasse, hauptsächlich für die Verwaltung der Karten im *Deck* zuständig, jedoch gibt es auch die *addStandardDeck()* Methode, welche in das *Deck* alle Karten eines Standard-Decks erstellt und hinzufügt. Diese Methode sollte

ausgelagert werden. Ein Standard Deck könnte von einer anderen Klasse erstellt werden und mit Verwendung der *fillDeck()* Methode von der *Deck*-Klasse dem *Deck* hinzugefügt werden. Die Klasse könnte ein DeckCreator oder ähnliches sein, welcher auch eine bestimmte DeckConfiguration annehmen könnte, um verschiedene Decks zu kreieren. So würde das *Deck* nur noch für die Verwaltung der Karten in einem *Deck* zuständig sein.

Analyse OCP (3P)

[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML 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



Durch die Abstraktion der *Player*-Klasse und mehrerer Methoden wird das Open-Closed-Principle eingehalten. Die *Player* Klasse erlaubt neue Implementationen und dadurch neues Verhalten, ohne etwas am bestehenden Code verändern zu müssen.

Negativ-Beispiel

Analyse [LSP/ISP/DIP] (2P)

[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP; jeweils UML und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird; beim Negativ-Beispiel UML einer möglichen Lösung hinzufügen]

[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

Negativ-Beispiel

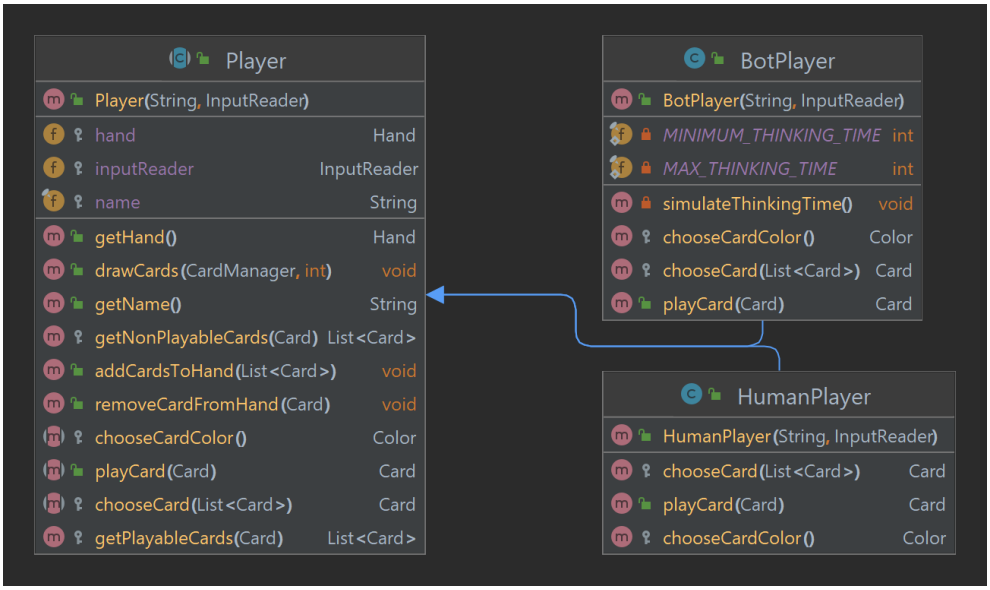
Kapitel 4: Weitere Prinzipien (8P)

Analyse GRASP: Geringe Kopplung (3P)

*[eine **bis jetzt noch nicht behandelte** Klasse als positives Beispiel geringer Kopplung; UML mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt]*

Analyse GRASP: [Polymorphismus/Pure Fabrication] (3P)

[eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt]



Die Klasse Player hat zwei Klassen, die von ihr erben: *HumanPlayer* und *BotPlayer*. Hier liegt also Polymorphismus vor. Es wird die *playCard()* Methode implementiert (sie ist abstract), weil ein Computergegner eine andere Logik braucht, als ein menschlicher Spieler. Beim Computergegner wird eine Karte mit einer bestimmten Logik ausgewählt, und dabei sogar die Zeit simuliert, die es für das Auswählen braucht. Der menschliche Spieler braucht aber hingegen eine Eingabe in der UI, um die Karte auszuwählen. Durch Polymorphismus kann dies umgesetzt werden, indem die *playCard()* Methode bei den beiden Klassen *BotPlayer* und *HumanPlayer* mit der benötigten Logik implementiert wird.

DRY (2P)

[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher) einfügen; begründen und Auswirkung beschreiben – ggf. UML zum Verständnis ergänzen]

Commit: 6efd7a59ade299e8ff768ff59aa316acb965fe32 (Added gamemodes and gamemode selection menu)

Die beiden Klassen *WildCard* und *WildDrawFourCard* nutzen beide den gleichen Code bei der *executeAction()* Methode.

Vorher

WildCard

```
@Override
    public void executeAction(Game game, InputReader inputReader, OutputWriter outputWriter) {
        super.executeAction(game, inputReader, outputWriter);

        outputWriter.write("Choosing new color!");

        Color color;

        if(game.getCurrentPlayer().isBot())
        {
            color = Color.values()[new Random().nextInt(Card.Color.values().length)];
        }
        else
        {
            color =  inputReader.chooseColor();
        }

        this.setColor(color);

        outputWriter.write("New color: " + color);
    }
```

WildDrawFourCard

```
@Override
    public void executeAction(Game game, InputReader inputReader, OutputWriter outputWriter) {
        super.executeAction(game, inputReader, outputWriter);

        outputWriter.write("Choosing new color!");

        Color color;

        if(game.getCurrentPlayer().isBot())
        {
            color = Color.values()[new Random().nextInt(Card.Color.values().length)];
        }
        else
        {
            color =  inputReader.chooseColor();
        }

        this.setColor(color);

        outputWriter.write("New color: " + color);
```

```
        // Next player has to draw 4 cards
        outputWriter.write(game.getPlayerManager().getNextPlayer().getName() + " has to draw 4
cards!");
        List<Card> drawnCards = game.getCardManager().drawCards(4);
        game.getPlayerManager().getNextPlayer().addCardsToHand(drawnCards);

        //Skip next player
        game.getPlayerManager().nextPlayer();
    }
```

Nachher

WildCard

```
@Override
    public void executeAction(Game game, InputReader inputReader, OutputWriter outputWriter) {
        super.executeAction(game, inputReader, outputWriter);
    }
```

WildDrawFourCard

```
@Override
    public void executeAction(Game game, InputReader inputReader, OutputWriter outputWriter) {
        super.executeAction(game, inputReader, outputWriter);

        // Next player has to draw 4 cards
        outputWriter.write(game.getPlayerManager().getNextPlayer().getName() + " has to draw 4
cards!");
        List<Card> drawnCards = game.getCardManager().drawCards(4);
        game.getPlayerManager().getNextPlayer().addCardsToHand(drawnCards);

        //Skip next player
        game.getPlayerManager().nextPlayer();
    }
```

Da die Logik bei jeder *WildCard* passieren muss, ist die Auswahl der Farbe nun nicht mehr in der Kartenklasse selbst, sondern bei der *Player*-Klasse.

Player

playCard(Card currentCard) Methode

```
// Choose a color when a wild card is played
if(chosenCard.getColor().equals(Card.Color.WILD)) {
    Card.Color color;

    if(this.isBot)
    {
        color = Card.Color.values()[new Random().nextInt(Card.Color.values().length - 1)];
    }
    else
    {
        color =  inputReader.chooseColor();
    }

    chosenCard.setColor(color);
}
```

Da der Spieler selbst die Karte auswählt, die er spielen möchte und bei einer *WildCard* immer eine Farbe gewählt werden muss, bevor sie quasi gespielt wird, macht es Sinn den Code für die Auswahl in der Player-Klasse zu schreiben. Dadurch ist auch der duplizierte Code aufgelöst und es wird immer derselbe Code ausgeführt bei einer Wild Card, selbst wenn eigene Wild Karten in der Zukunft hinzugefügt werden sollten.

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

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

Unit Test	Beschreibung
PlayerFactoryTest #shouldCreateHumanPlayer	Testet, ob die Methode createPlayer() mit dem Parameter PlayerType.HUMAN auch wirklich einen HumanPlayer generiert.
DeckTest #initialize_shouldFillDeck	Testet, ob nach der initialize() Methode von Deck das Deck nicht leer ist.
DeckTest #fillDeck_shouldAddCardsToDeck	Testet, ob bei fillDeck() die Karten wirklich in das Deck geschoben werden.
StashTest #getSupplyCards_shouldReturnAllCardsButTopCardAndClearStash	Testet, ob die getSupplyCards() Methode alle Karten bis auf die oberste zurückgibt und aus dem Stash löscht. Es sollte nur die oberste Karte im Stash zurückbleiben.
SettingsTest #settingNumPlayersShouldWork	Testet, ob die Einstellungen für die Anzahl an Spielern wirklich gesetzt werden.
StashTest #addCard_shouldAddCardToStash	Testet, ob addCard() wirklich eine Karte zum Stash hinzufügt und sie die oberste Karte ist.

ATRIP: Automatic (1P)

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

ATRIP: Thorough (1P)

[Code Coverage im Projekt analysieren und begründen]

ATRIP: Professional (1P)

[1 positves Beispiel zu 'Professional'; Code-Beispiel, Analyse und Begründung, was professionell ist]

Fakes und Mocks (3P)

[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten (die Fake/Mocks sind ohne Dritthersteller-Bibliothek/Framework zu implementieren); zusätzlich jeweils UML Diagramm mit Beziehungen zwischen Mock, zu mockender Klasse und Aufrufer des Mocks]

Kapitel 6: Domain Driven Design (8P)

Ubiquitous Language (2P)

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

Bezeichung	Bedeutung	Begründung
Deck	Der Kartenstapel zum Ziehen der Handkarten.	Das Deck ist ein zentraler Bestandteil eines Kartenspiels. Es ist die Quelle aller Karten, die im Spiel verwendet werden.
Hand	Die Karten eines Spielers.	Hand ist ein spezieller Begriff bei Kartenspielen. Hand ist eigentlich ein Körperteil, jedoch werden hier die Karten eines Spielers gemeint.
Player	Eine Person, die aktives Mitglied einer Runde UNO ist.	Domäne ist Spiel: In den meisten Spielen gibt es Spieler. Der Begriff wird nur im Kontext von „Spielen" benutzt.
Turn	Eine Aktion, die ein Spieler während seiner Spielrunde ausführt. (Zug)	Der Zug (Turn) ist vor allem bei Spielen ein wichtiger Begriff. Besonders bei turn-based Spielen.

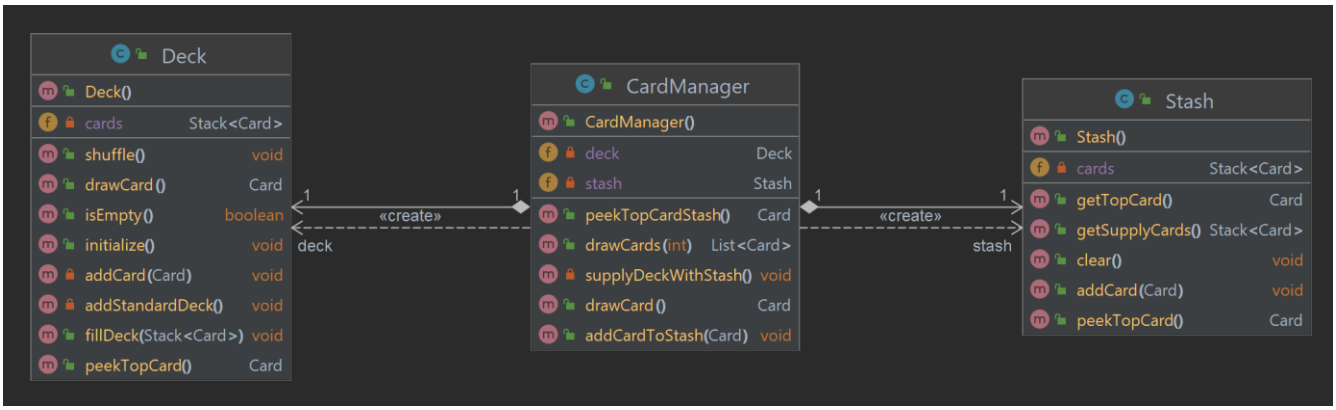
Repositories (1,5P)

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

Es gibt das *SettingsRepository*, welches das Speichern und Laden von Einstellungen vornimmt. Die Einstellungsmöglichkeiten sind in diesem Fall die Anzahl an menschlichen Spielern und die Anzahl an Computer-Spielern.

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– NICHT, warum es nicht implementiert wurde]



Der CardManager ist ein Aggregate, welches die Karten auf dem Spielfeld verwaltet. Sowohl Deck als auch Stash (der Kartenstapel in der Mitte) sind Entities, welche vom CardManager verwaltet werden. Es wird eingesetzt, weil zum einen beim Starten eines Spiels die erste Karte direkt als erstes auf den Stash gelegt wird. Dafür gibt es dann die Methode „*addCardToStash(Card)*“ Außerdem wird das Deck mit allen Karten aus dem Stash (außer der obersten Karte) gefüllt, wenn das Deck beim Ziehen leer geht („*supplyDeckWithStash()*“). Somit ist die komplexe Beziehung der beiden Entities Deck und Stash durch den CardManager vereinfacht worden.

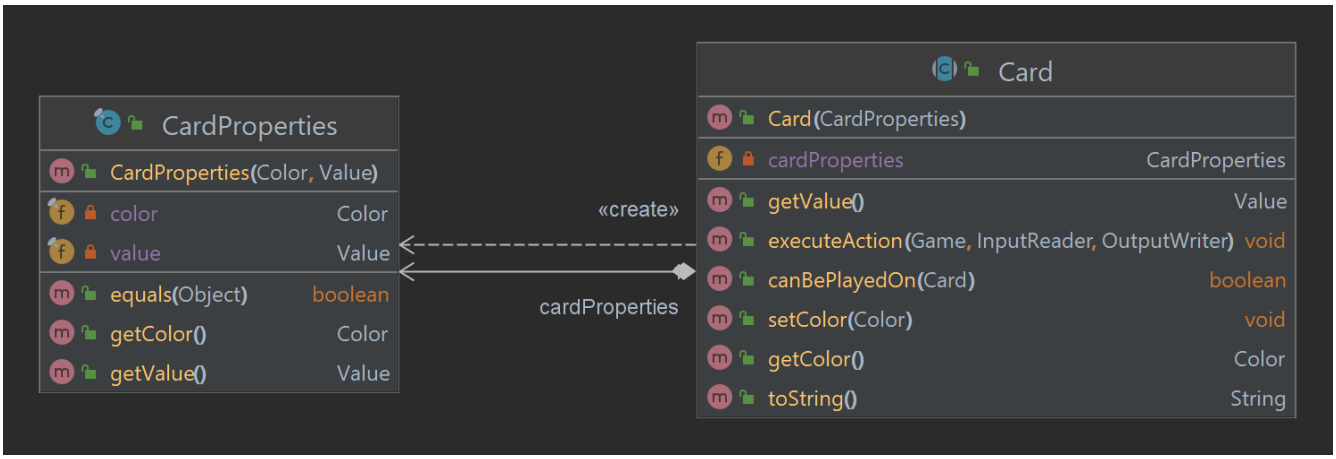
Entities (1,5P)

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

-

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– NICHT, warum es nicht implementiert wurde]



Die *CardProperties* Klasse ist ein ValueObject, welches die Information über Farbe und Wert einer Karte enthält. Sie hat eine eigene *equals()* Methode und nur Getter Methoden. Wenn ein Wert geändert werden muss, muss ein neues *CardProperties* Objekt erstellt werden. Dies geschieht bei *setColor(Color)* in der *Card*-Klasse. Es wird benutzt, weil eine Karte normalerweise nicht plötzlich seine Farbe oder seinen Wert ändert. Eine definierte Karte, wie die Rote Acht, wird niemals ihre Farbe oder ihren Wert ändern. Außerdem wird durch die *CardProperties* zusammengehörige Eigenschaften vereint. Einzig und allein die Wild Karten müssen ihre Farbe „ändern“, sobald sie gespielt werden.

Kapitel 7: Refactoring (8P)

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)]

[CODE SMELL 1]

[CODE SMELL 2]

2 Refactorings (6P)

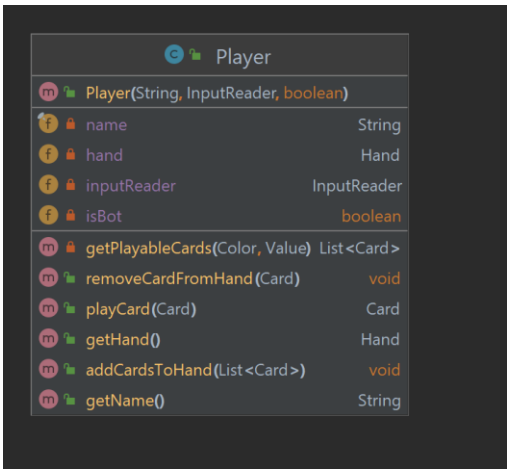
[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]

1. Replace conditional with polymorphism

Commit: a63ebe19f0dd276ed6cb5b8260896a96c3e8f2be (Added GameEvent and some tests, split Player into Human and Bot)

Anstatt zu prüfen, ob das Attribut *isBot* bei einem Player *true* ist, um zu entscheiden, welche Aktion bei *playCard()* ausgeführt werden soll, wurde Polymorphismus eingeführt. Sonst liegt die Logik für einen Computer gesteuerten Gegner und die Logik für einen menschlichen Spieler in einer Methode und wird durch ein Conditional gewählt. Mit Polymorphismus hingegen kann die Logik einfach in einer eigenen Implementation der *playCard()* Methode liegen. Somit ist der spezifische Programmcode sauber getrennt zwischen *BotPlayer* und *HumanPlayer*.

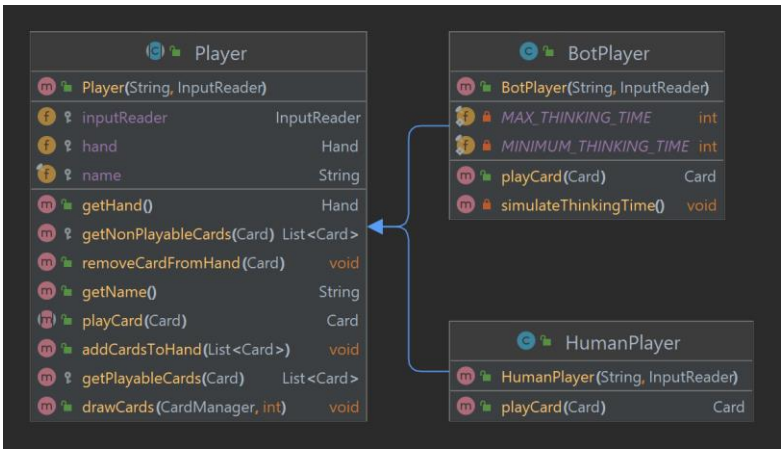
Vorher




```java
public Card playCard(Card currentCard) {
        List<Card> playableCards = getPlayableCards(currentCard.getColor(), currentCard.getValue());
        if (playableCards.isEmpty()) {
            return null;
        } else {
            Card chosenCard;

            if(this.isBot) {
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                chosenCard = playableCards.get(0);
            } else {
                chosenCard = inputReader.chooseCard(playableCards);
            }

            // Choose a color when a wild card is played
            if(chosenCard.getColor().equals(Card.Color.WILD)) {
                Card.Color color;

                if(this.isBot)
                {
                    color = Card.Color.values()[new Random().nextInt(Card.Color.values().length - 1)];
                }
                else
                {
                    color =  inputReader.chooseColor();
                }

                chosenCard.setColor(color);
```

```
            }

            this.removeCardFromHand(chosenCard);
            return chosenCard;
        }
    }
```

Nachher



Player

```
public abstract Card playCard(Card currentCard);
```

HumanPlayer

```
@Override
    public Card playCard(Card currentCard)
    {
        List<Card> nonPlayableCards = getNonPlayableCards(currentCard);
        System.out.println(nonPlayableCards);
        List<Card> playableCards = getPlayableCards(currentCard);
        if (playableCards.isEmpty())
        {
            return null;
        } else
        {
            Card chosenCard = inputReader.chooseCard(playableCards);

            // Choose a color when a wild card is played
            if (chosenCard.getColor().equals(Card.Color.WILD))
            {
                Card.Color color = inputReader.chooseColor();

                chosenCard.setColor(color);
            }

            this.removeCardFromHand(chosenCard);
            return chosenCard;
        }
    }
```

BotPlayer

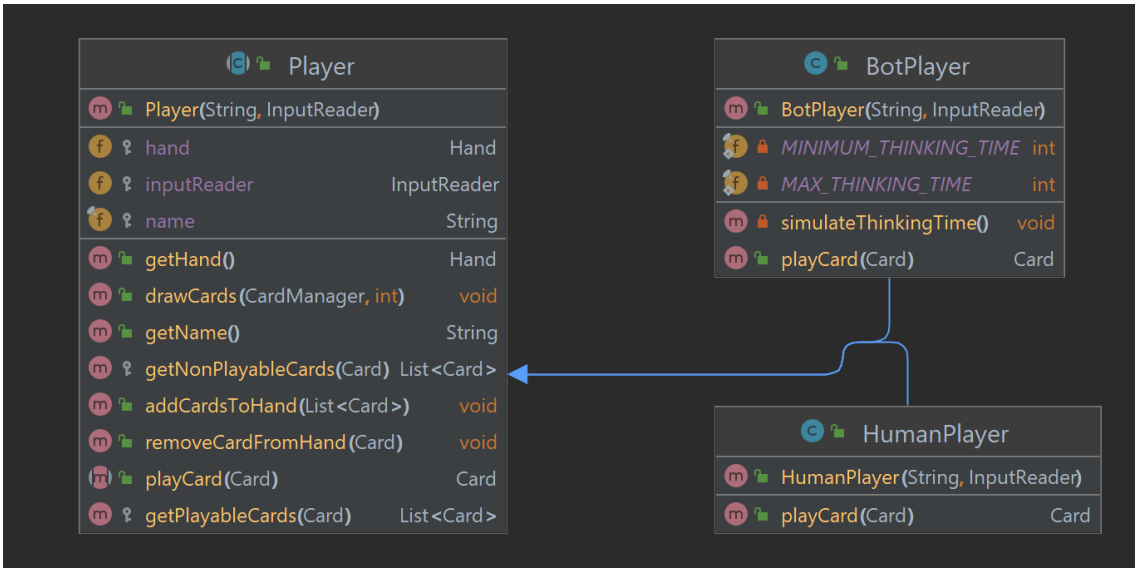
```
@Override
    public Card playCard(Card currentCard)
    {
        List<Card> playableCards = getPlayableCards(currentCard);
        if (playableCards.isEmpty())
        {
            return null;
        } else
        {
            simulateThinkingTime();

            Card chosenCard = playableCards.get(0);

            // Choose a color when a wild card is played
            if (chosenCard.getColor().equals(Card.Color.WILD))
            {
                Card.Color color = Card.Color.values()[new Random().nextInt(Card.Color.values().length
- 1)];

                chosenCard.setColor(color);
            }

            this.removeCardFromHand(chosenCard);
            return chosenCard;
        }
    }
```

2. Extract Method

Commit: [51f95413ffee164b10e7053b448873a870e9b2f5] (Extracted methods out of Player playCard() Method)

Die *playCard()* Methode hat bisher innerhalb der Methode das Auswählen der Karte sowie die Farbauswahl bei Wild-Karten implementiert. Lieber werden diese verschiedenen Aktionen in eigene Methoden ausgelagert. So wurden die zwei Methoden *chooseCard()* und *chooseCardColor()* hinzugefügt. Damit ist die *playCard()* Methode viel kleiner geworden und der Code auch sauberer getrennt.

Vorher



Player

```
public abstract Card playCard(Card currentCard);
```

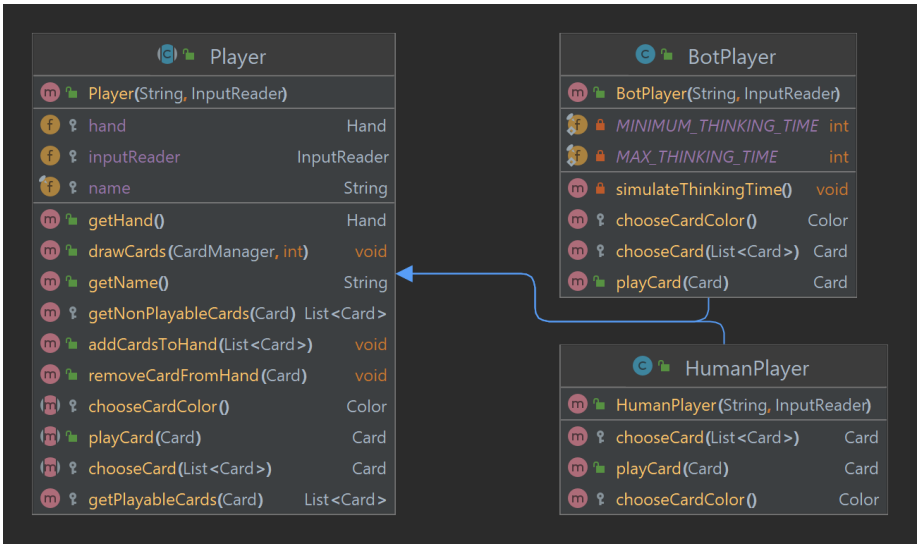
HumanPlayer

```
@Override
    public Card playCard(Card currentCard)
    {
        List<Card> nonPlayableCards = getNonPlayableCards(currentCard);
        System.out.println(nonPlayableCards);
        List<Card> playableCards = getPlayableCards(currentCard);
        if (playableCards.isEmpty())
        {
            return null;
        } else
        {
            Card chosenCard = inputReader.chooseCard(playableCards);

            // Choose a color when a wild card is played
            if (chosenCard.getColor().equals(Card.Color.WILD))
            {
                Card.Color color = inputReader.chooseColor();

                chosenCard.setColor(color);
            }

            this.removeCardFromHand(chosenCard);
            return chosenCard;
        }
    }
```

BotPlayer

```
@Override
    public Card playCard(Card currentCard)
    {
        List<Card> playableCards = getPlayableCards(currentCard);
        if (playableCards.isEmpty())
        {
            return null;
        } else
        {
            simulateThinkingTime();

            Card chosenCard = playableCards.get(0);

            // Choose a color when a wild card is played
            if (chosenCard.getColor().equals(Card.Color.WILD))
            {
                Card.Color color = Card.Color.values()[new Random().nextInt(Card.Color.values().length
- 1)];

                chosenCard.setColor(color);
            }

            this.removeCardFromHand(chosenCard);
            return chosenCard;
        }
    }
```

Nachher



Player

```
public abstract Card playCard(Card currentCard);

protected abstract Card chooseCard(List<Card> playableCards);

protected abstract Card.Color chooseCardColor();
```

HumanPlayer

```
@Override
    public Card playCard(Card currentCard)
    {
        List<Card> nonPlayableCards = getNonPlayableCards(currentCard);
        System.out.println(nonPlayableCards);
        List<Card> playableCards = getPlayableCards(currentCard);
        if (playableCards.isEmpty())
        {
            return null;
        } else
        {
            Card chosenCard = chooseCard(playableCards);

            this.removeCardFromHand(chosenCard);

            return chosenCard;
        }
    }

    @Override
    protected Card chooseCard(List<Card> playableCards)
    {
        Card chosenCard = inputReader.chooseCard(playableCards);

        // Choose a color when a wild card is played
```

```
        if (chosenCard.getColor().equals(Card.Color.WILD))
        {
            Card.Color color = chooseCardColor();
            chosenCard.setColor(color);
        }

        return chosenCard;
    }

    @Override
    protected Card.Color chooseCardColor()
    {
        return inputReader.chooseColor();
    }
```

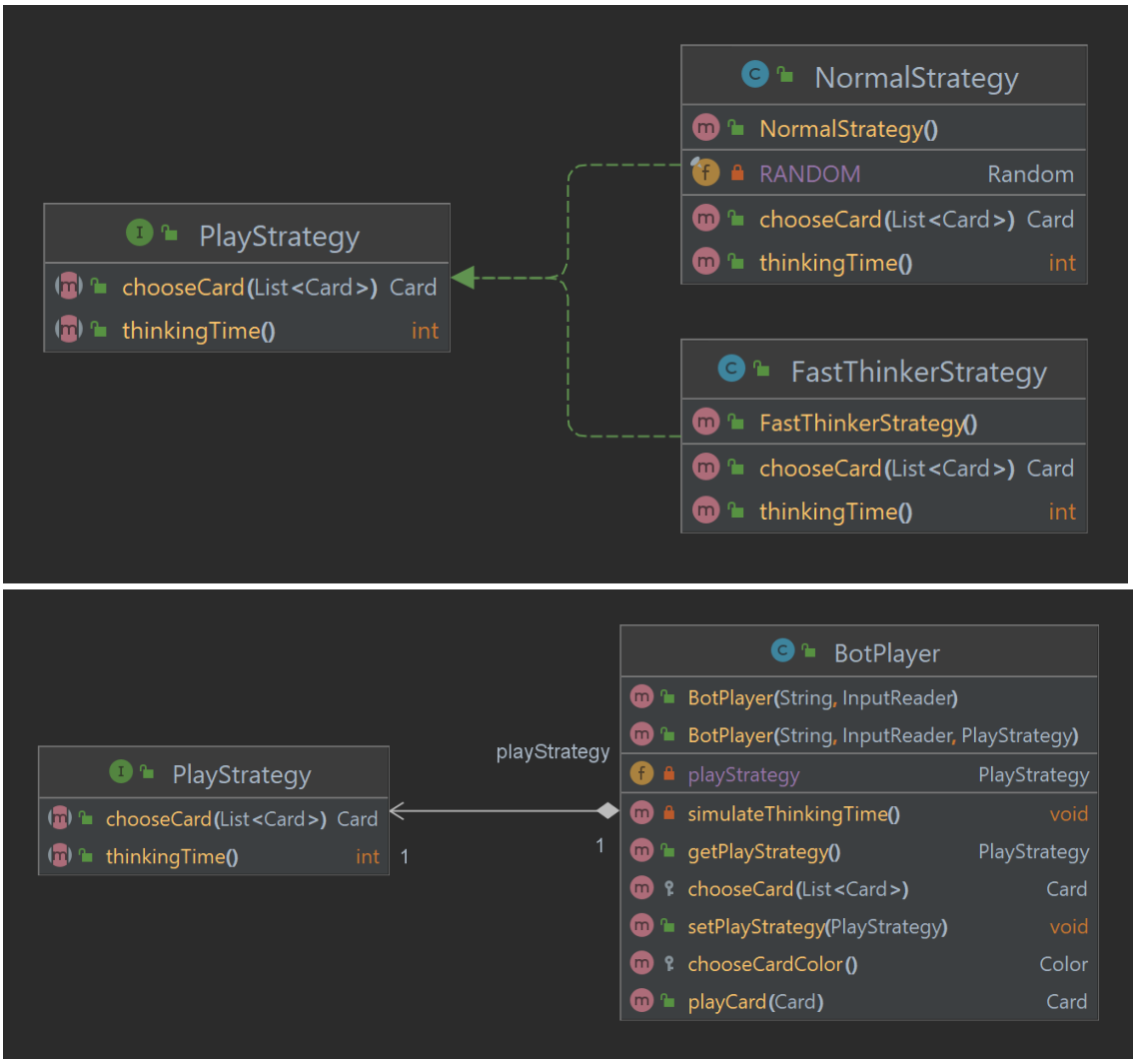
BotPlayer

```
@Override
    public Card playCard(Card currentCard)
    {
        List<Card> playableCards = getPlayableCards(currentCard);
        if (playableCards.isEmpty())
        {
            return null;
        } else
        {
            Card chosenCard = chooseCard(playableCards);

            this.removeCardFromHand(chosenCard);

            return chosenCard;
        }
    }

    @Override
    protected Card chooseCard(List<Card> playableCards)
    {
        simulateThinkingTime();

        Card chosenCard = playableCards.get(0);

        // Choose a color when a wild card is played
        if (chosenCard.getColor().equals(Card.Color.WILD))
        {
            Card.Color color = chooseCardColor();
            chosenCard.setColor(color);
        }

        return chosenCard;
    }

    @Override
    protected Card.Color chooseCardColor()
    {
        return Card.Color.values()[new Random().nextInt(Card.Color.values().length - 1)];
    }
```

Kapitel 8: Entwurfsmuster (8P)

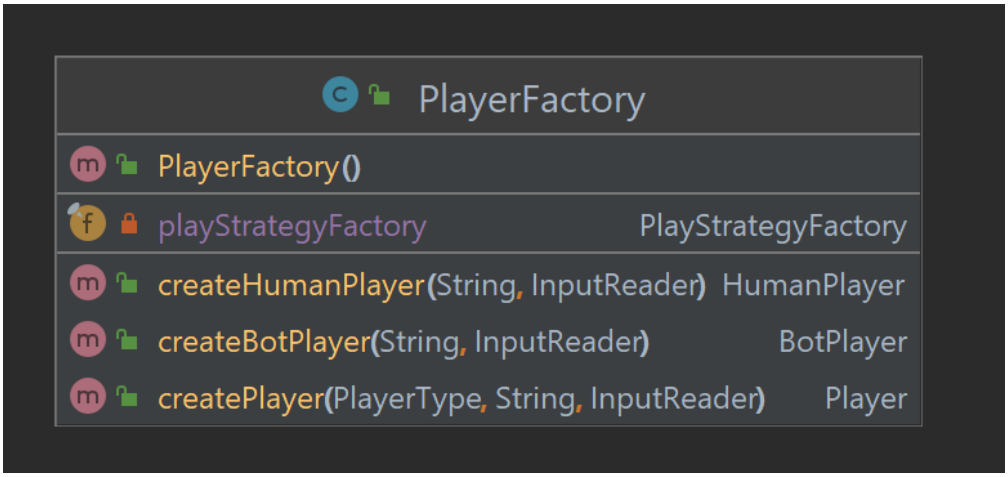
[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils sinnvoll einsetzen, begründen und UML-Diagramm]

Entwurfsmuster: Strategy (4P)



Die *PlayStrategy* ist eine Klasse, welche sowohl die Denkzeit als auch den Auswahlalgorithmus für die Karte bereitstellt. Dies ist hauptsächlich für *BotPlayer* gedacht. Hiermit können verschiedene Verhaltensmuster implementiert werden und bei Erstellung des *BotPlayer* mitgegeben werden.

Entwurfsmuster: Factory (4P)



Die *PlayerFactory* erstellt Player-Instanzen. Dabei wird bei *BotPlayer* gleich eine *PlayStrategy* vorgegeben, die durch die *PlayStrategyFactory* erstellt wird (zufällig). Durch die Factory wird das Kreieren von Players vereinfacht und lässt einen auch mit einem *PlayerType* den korrespondierenden Player erstellen. *PlayerType* ist dabei HUMAN oder BOT.