Dokumentation Projekt Card Game App - OOP

Projektarbeit für
FH Masterstudium DDB Programieren Java

Gruppe International Beasty Boys
- Markus Hilbert (Lead/Architecture)
- Hannes Brottrager (Logging/Data/Exceptions/Console)
- Felix Ossmann (Testing)
- Eric Langer (Docs for Code)

Inhaltsverzeichnis

1 Einführung

1.1 Idee

Ein Projekt entwerfen von Grund auf wo die gelernten Fähigkeiten zum Einsatz kommen auf Basis von Java als OOP Objekt Orientiertes Programmieren. Aufgabe ist auch alles so aufzubauen das der meiste Lerneffekt entsteht.
Nach einigen Ideen wie z.b.: Auto Datenbank usw. wurde darauf geeinigt ein Karten Spiel zu Entwickeln das mehrere Karten spiele Ausgeben kann – zwecks Demonstration wurde Black Jack ausgewählt.

1.2 Ziele

Das Karten Spiel sollte erweiterbar sein und einer einfachen Logik folgen.

Prozeduren	-	if/else – case – while/for – early return
OOP	-	Encaspulation – Inheritance – Abtraction – Types – Enumaration/Primitive Complex Types
OOP-Klassen	-	Constructors – Setters&Getters – Modifiers/public private protected abstract final – Return
Java STD	-	Scanner/Terminal – File I/O – Date
Frameworks	-	Logging via java.util.logging – Testing via Junit – Code Versin via GIT
Dokumentation	-	MarkDown - Javadoc

1.3 Nicht Ziele

Gedanken zwecks Erweiterungen wurden gemacht aber aufgrund des Zeitlichen Limits und dem nicht in Unnötigen Ideen sich zu Verlieren wurden auch gezielt nicht Ziele gesetzt um die Produktion zu Konzentrienen.

Keine GUI (Graphical User Interface)
Keine AI (Artifizielle Intelligenz)
Keine Produktions Fähige Anwendung

2 Setup

2.1 Vorbereitung

Dadurch das 4 Personen mit Unterschiedlichen Betriebssysteme und Ortschaften zusammen arbeiten brauchte es eine Lösung damit das Fehlerfreie Arbeiten so gut wie möglich von statten gehen kann.

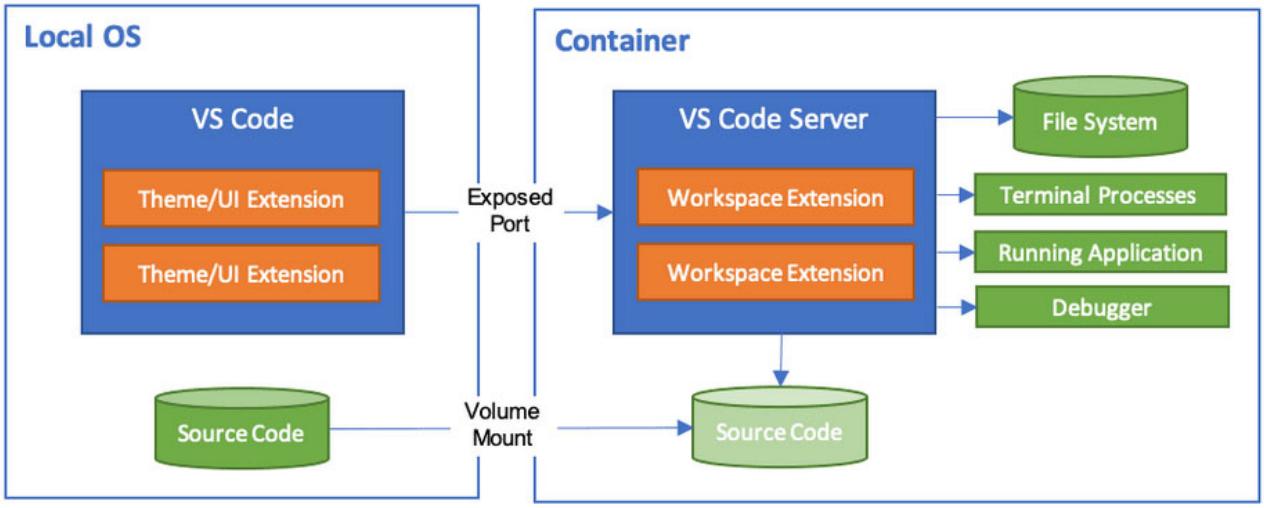
Zuerst wurde entschieden sich für eine Docker Lösung zu entscheiden für die Vereinfachung der Bereitstellung von einer Anwendung (Java). Dabei ist auch zu beachten das Docker auf dem Betriebsystem auch ausführbar ist. Wir hatten z.b.: ein Problem das Docker auf WIN10 Home sich nur sehr umständlich installieren lässt. Zusätzlich wird über GIT alles verwaltet.

Danach wurde festgelegt das das Spiel Black Jack verwendet wird und deren Regel Recherchiert nach welcher Logik und Muster man vorgehen soll.

2.2 Setup

Auf dem Docker läuft VS-Code zusammen mit einem Devcontainer um alle Dateien zusammen führen zu können um überall darauf zugreifen zu können.

Docker / VS-Code / Dev-Container / GIT



3 Theoretische Betrachtung

3.1 Prototyp

Zuerst wurden verschiedene vorgehensweisen getestet wie auf was zueinander greift. Für das Prototyping wurde mal eine Basis Implementierung angedacht Java mit seiner Daten Struktur die erweiterbar sein soll. Dafür wurde eine Einfache Lösung angedacht.

Man musste bedenken das Black Kack ein Runden Basiertes Spiel ist wo der Dealer ein Spieler ist der gegen jeden spielt und der Spieler (Person) mehrfach geben kann die immer alle zusammen gegen den Dealer Spielen.

Features:	class / constructor / modifiers-private-public-static / getters-Setters
Struktur Daten:	int / boolean / String / ArrayList / (nested) LinkedHashMap / Random Scanner / Collections (shuffle)

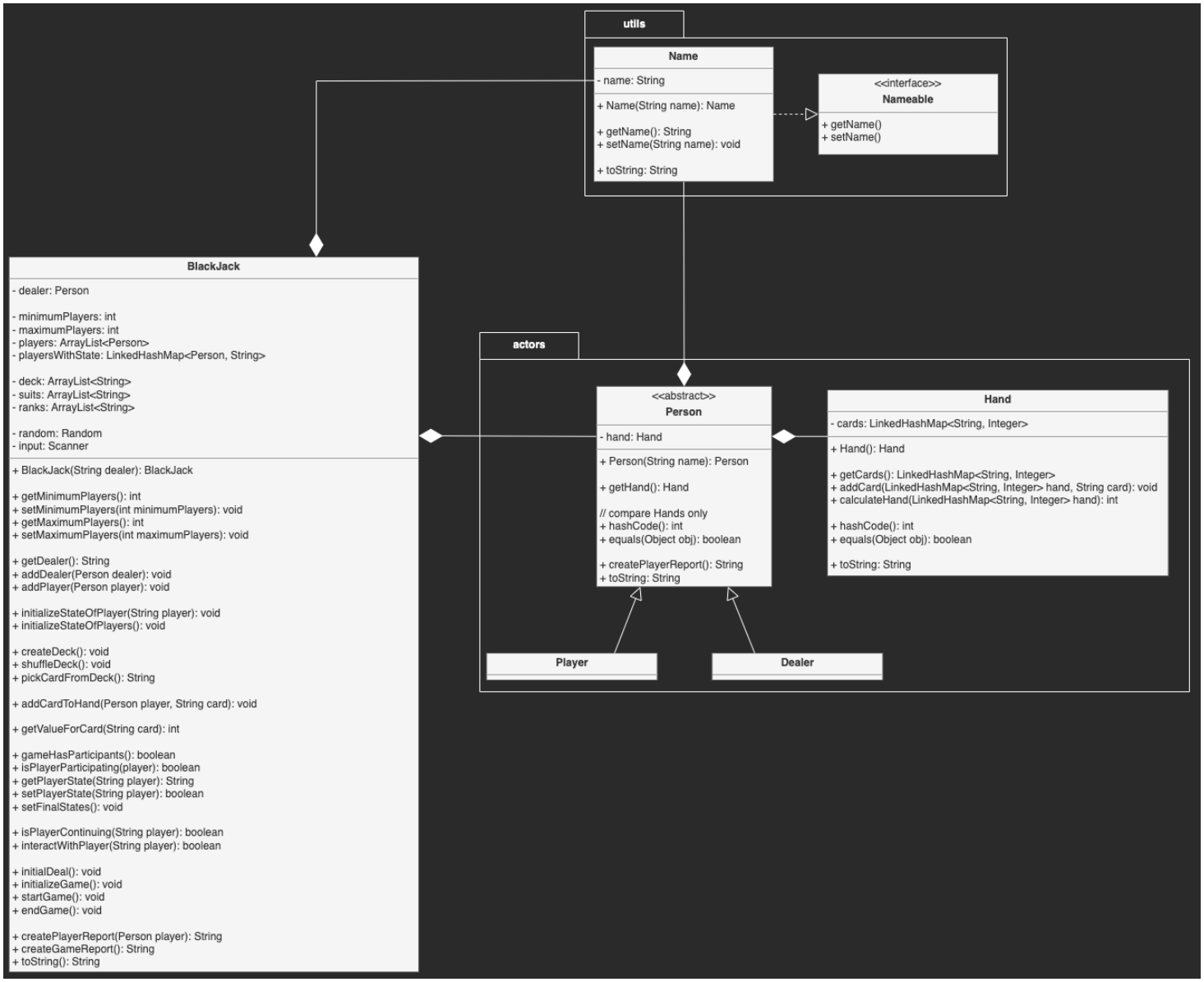
3.2 Architektur

Daraufhin konnte man anfangen die Umsetzung der Personen und Abstrakte Umzusetzen.

Eine *Abstrakte* „Person“ Klasse erstellen und „Dealer“ und „Player“ erstellen die eine Erbung davon bekommen.

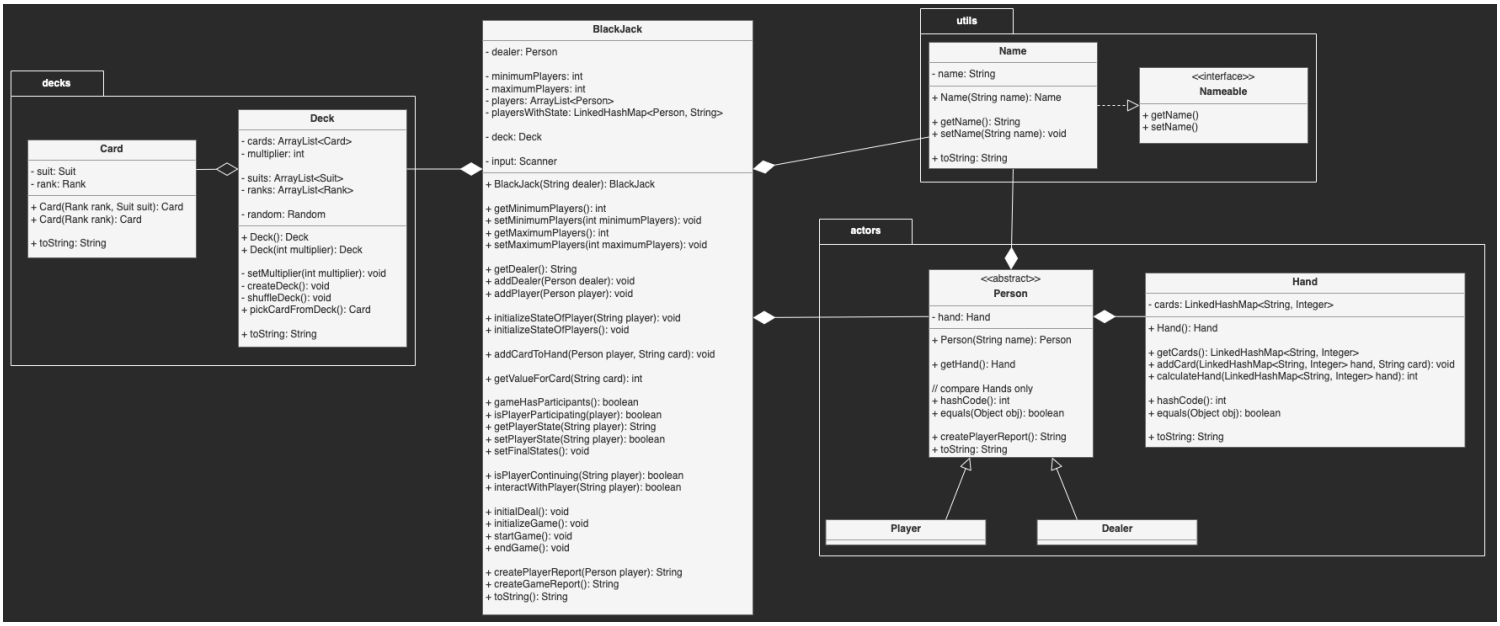
Das Feld „Name“ (*Abstrakt*) wird öfters benutzt werden um „Dealer“ „Player“ „Game“ usw. zu implementieren, damit wir einen Kontakt/Verbindung haben zu allem was einen „Name“ beinhaltet

Features:	abstract / encapsulation / interface / additional modifiers (final)
Neue Typen:	Person / Player / Dealer / Hand

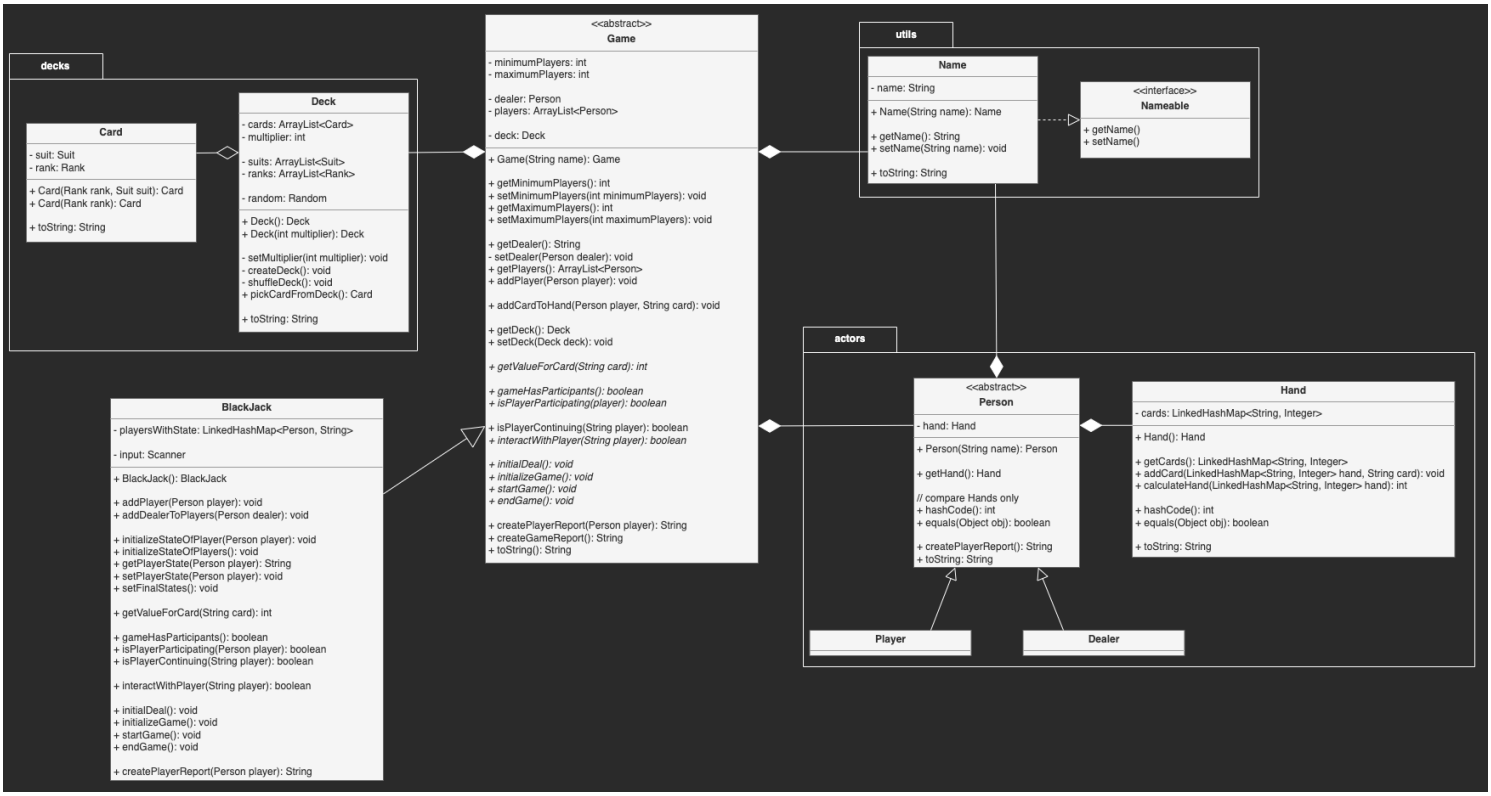


Abstrakt und Hinzufügen von „Deck“ und „Card“, ein „Deck“ zu erstellen das die Separate Klasse „Cards“ hat.

Features: switch / case (für Wertigkeiten in Ränge bei spielen)
Types: Deck / Card



Abstrakt „Game“ ist eine Implementation und hat Black Jack Spezifische Regeln zum Ausführen.

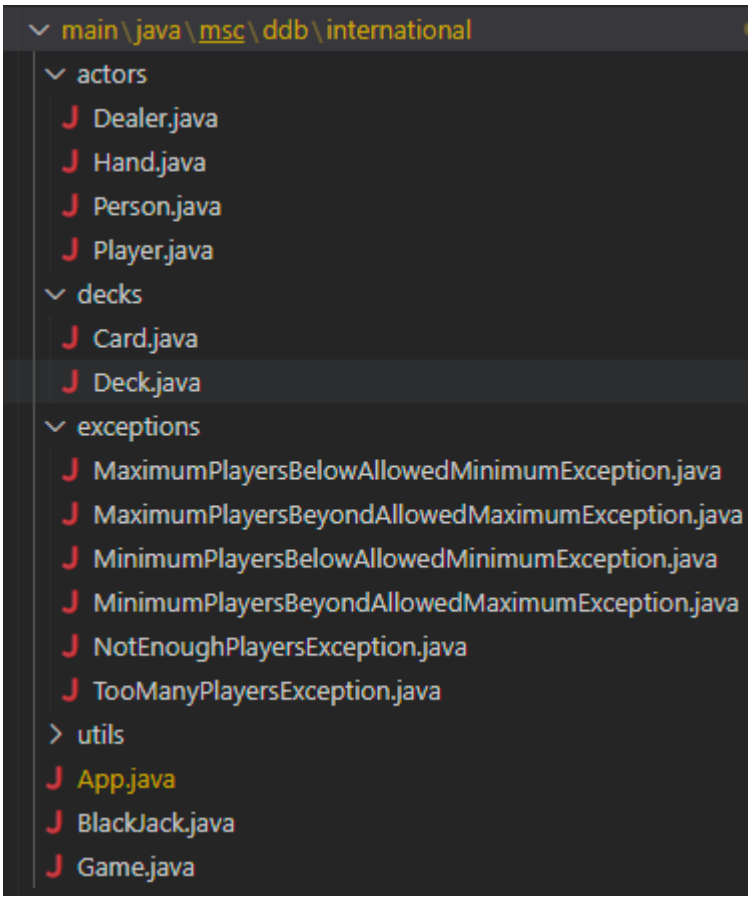


Erweitere Anpassungs Möglichkeiten
Mann kann die Muster „Player“ und „Dealer“ übernehmen oder anpassen.
Die „Command Pattern“ ist die Basis für andere Spiele.
Für das Muster „Command Pattern“ kann man neue Regeln erstellen und diese dann zu „Game“ hinzufügen.
Abstrakte Runden für ein Muster „Decorator Pattern“ könnte dann mehrere Spiele per Sitzung erlauben z.b.: UNO

4 Programm Card Game App

4.1 Aufbau

Aufbau des Java Programms



4.2 Übersicht

> Dealer

src > main > java > msc > ddb > international > actors > Dealer.java > Dealer > Dealer(String)

```java
package msc.ddb.international.actors;

public class Dealer extends Person {

    public Dealer(String name) {
        super(name);
        //TODO Auto-generated constructor stub
    }

    public Dealer(String name, Hand hand) {
        super(name, hand);
    }

}
```

> Hand

src > main > java > msc > ddb > international > actors > Hand.java > Hand

```java
package msc.ddb.international.actors;

import java.util.LinkedHashMap;
import java.util.StringJoiner;

import msc.ddb.international.decks.Card;

public class Hand {
    LinkedHashMap<Card, Integer> cards = new LinkedHashMap<Card, Integer>();

    public Hand() {}

    public LinkedHashMap<Card, Integer> getCards() {
        return cards;
    }

    public void addCard(Card card, int value) {
        cards.put(card, value);
    }

    public void setCards(LinkedHashMap<Card, Integer> cards) {
        this.cards = cards;
    }

    public int calculateHand() {
        return cards.values().stream().reduce(0, (a, b) -> a + b);
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((cards == null) ? 0 : cards.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Hand other = (Hand) obj;
        if (cards == null) {
            if (other.cards != null)
                return false;
        } else if (!cards.equals(other.cards))
            return false;
        return true;
    }

    @Override
    public String toString() {
        StringJoiner output = new StringJoiner(", ");
        cards.forEach((Card card, Integer value) -> {
            output.add(String.format("%s (%d)", card.getName(), value));
        });
        return String.format("Hand:\n%s\nTotal: %d", output.toString(), calculateHand());
    }


}
```

> Person

src > main > java > msc > ddb > international > actors > Person.java > {} msc.ddb.international.actors

```java
package msc.ddb.international.actors;

import msc.ddb.international.utils.Name;

public abstract class Person extends Name {
    private Hand hand = new Hand();

    public Person(String name) {
        super(name);
        setHand(hand);
        hand = new Hand();
    }

    public Person(String name, Hand hand) {
        super(name);
        setHand(hand);
    }

    public Hand getHand() {
        return hand;
    }

    public void setHand(Hand hand) {
        this.hand.setCards(hand.getCards());
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((getName() == null) ? 0 : getName().hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        // DEALER and PLAYER are different Classes - we only care about the name
        // if (getClass() != obj.getClass())
        //     return false;
        Person other = (Person) obj;
        if (getName() == null) {
            if (other.getName() != null)
                return false;
        } else if (!getName().equals(other.getName()))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + " [" + super.toString() + ", hand=" + hand + "]";
    }

}
```

> Player

src > main > java > msc > ddb > international > actors > Player.java > {} msc.ddb.international.actors

```java
package msc.ddb.international.actors;

public class Player extends Person {

    public Player(String name) {
        super(name);
        //TODO Auto-generated constructor stub
    }

    public Player(String name, Hand hand) {
        super(name, hand);
    }

}
```

> Cards

src > main > java > msc > ddb > international > decks > Card.java > {} msc.ddb.international.decks

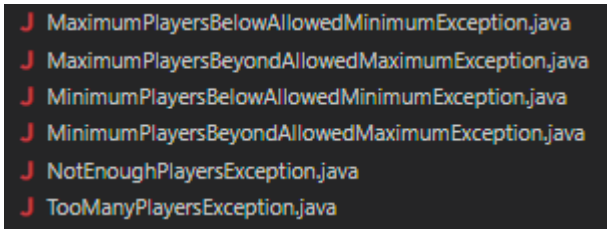
```java
package msc.ddb.international.decks;

import java.util.StringJoiner;

import msc.ddb.international.utils.Name;

public class Card extends Name {
    private String rank;
    private String suit;

    private Card() {
        super(name: "");
    }

    public Card(String rank, String suit) {
        super(name: "");
        this.rank = rank;
        this.suit = suit;
        setName();
    }

    public Card(String rank) {
        super(name: "");
        this.rank = rank;
        setName();
    }

    public String getRank() {
        return rank;
    }

    public void setRank(String rank) {
        this.rank = rank;
    }

    public String getSuit() {
        return suit;
    }

    public void setSuit(String suit) {
        this.suit = suit;
    }

    public void setName() {
        StringJoiner name = new StringJoiner(" of ");
        name.add(rank);
        name.add(suit);
        super.setName(name.toString());
    }

    @Override
    public String toString() {
        return "Card [rank=" + rank + ", suit=" + suit + "]";
    }


}
```

> Deck

src > main > java > msc > ddb > international > decks > Deck.java > {} msc.ddb.international.decks

```java
package msc.ddb.international.decks;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;

public class Deck {
    private ArrayList<Card> cards = new ArrayList<Card>();
    private int multiplier = 1;

    private Random random = new Random();

    private static final String[] suits = {
        "Hearts",
        "Diamonds",
        "Spades",
        "Clubs"
    };
    private static final String[] ranks = {
        "Two",
        "Three",
        "Four",
        "Five",
        "Six",
        "Seven",
        "Eight",
        "Nine",
        "Ten",
        "Jack",
        "Queen",
        "King",
        "Ace"
    };

    public Deck() {
        createDeck();
        shuffleDeck();
    }

    public Deck(int multiplier) {
        setMultiplier(multiplier);
        createDeck();
        shuffleDeck();
    }

    private void createDeck() {
        // lets create a deck of 52 cards
        ArrayList<Card> temp = new ArrayList<Card>();
        for (String suit : suits) {
            for (String rank : ranks) {
                temp.add(new Card(rank, suit));
            }
        }
        // BlackJack has 6 decks, so lets add 5 copies
        for (int i = 0; i < 6; i++) {
            cards.addAll(temp);
        }
    }

    private void shuffleDeck() {
        Collections.shuffle(cards, random);
    }

    public Card pickCard() {
        // remove the last card from deck and return it
        return cards.remove(cards.size() - 1);
    }

    private void setMultiplier(int multiplier) {
        if(multiplier > 0)
            this.multiplier = multiplier;
        else {
            // TODO: throw Exception
            System.out.println("no negativ values allowed.");
        }
    }

    @Override
    public String toString() {
        return "Deck [multiplier=" + multiplier + ", cardAmount=" + cards.size() + "]";
    }
```

> Exceptions



src > main > java > msc > ddb > international > exceptions > MaximumPlayersBelowAllowedMinimumException.java > ...

```java
package msc.ddb.international.exceptions;

public class MaximumPlayersBelowAllowedMinimumException extends Exception {
    public MaximumPlayersBelowAllowedMinimumException(String errorMessage) {
        super(errorMessage);
    }
}

```

>utils>Name

src > main > java > msc > ddb > international > utils > Name.java > {} msc.ddb.international.utils

```java
package msc.ddb.international.utils;

public class Name implements Nameable {
    String name;

    public Name(String name) {
        setName(name);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if(name.isEmpty())
            name = "Unknown";
        this.name = name;
    }

    @Override
    public String toString() {
        return "Name [name=" + name + "]";
    }

}
```

>utils>nameable

src > main > java > msc > ddb > international > utils > Nameable.java > {} msc.ddb.international.utils

```java
package msc.ddb.international.utils;

/*
 * @FunctionalInterface (called by class Name)
 * <p><i>This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference</i></p>
 */
public interface Nameable {
    void setName(String name);
    String getName();
}
```

5 Dokumentation

5.1 Logging

Unter *Logging* versteht man das Protokollieren eines Softwareablaufes. Ein solches Protokoll entsteht nicht automatisch, sondern muss eingerichtet werden. Üblicherweise werden dabei Zwischenzustände eines Programmablaufes notiert. Für die Ausgabe des Logprotokolls können dabei unterschiedliche Ziele (Konsole, Datei, etc.) verwendet werden.

5.2 Data

6 Testing

6.1 Unit Testing

Testing mit Junit in Java.
Testen ist ein unverzichtbarer Bestandteil der Entwicklung von Software. Dabei sollte man schon sehr frühzeitig im Entwicklungsprozess mit der Erstellung von Tests zu beginnen. Idealerweise erfolgt die Programmierung einer Java Klasse und des zugehörigen Tests gleichzeitig, bei uns war es danach da wir noch geübt haben damit.

Ziel davon – Code isolieren und auf Funktion Prüfen

z.b.: test\java\msc\ddb\international
> Dealer Test

7 Spiel

7.1 Spielregeln

Black Jack (17+4)

„Dealer“ spielt gegen „Player“

Ziel ist es wer die Summer 21 so genau wie möglich bekommt.
Es werden so lange Karten abgehoben bis man über 21 ist oder so nah wie möglich. Ist man über 21 hat man automatisch verloren. Solange man nähre an 21 ist wie der Dealer hat man gewonnen.
Wenn mehre „Player“ spielen alle gegen den „Dealer“

7.2 Manual

Es wird über den Terminal gespielt

```
Dez. 14, 2022 5:49:50 PM msc.ddb.international.App getEnvironment
INFORMATION: Write Environments: FogFrog


Standard Language      = Deutsch
Standard Language (EN) = German

---------------------------------------------
Choose a language: (German|English [X=Exit])
---------------------------------------------

```

Nachdem man eine Sprache ausgewählt hat startet das spiel und man bekommt seine Karten

```
Dealer is in state "HIT" and has Hand:
Ten of Spades (10), Five of Clubs (5)
Total: 15
Dealer is in state "LOST" and has Hand:
Ten of Spades (10), Five of Clubs (5), King of Diamonds (10)
Total: 25



BlackJack (17+4) is finished.
Dealer is in state "LOST" and has Hand:
Ten of Spades (10), Five of Clubs (5), King of Diamonds (10)
Total: 25

Harald is in state "WON" and has Hand:
Jack of Spades (10), Queen of Diamonds (10)
Total: 20


Card game finished.
```

Danach kann man mit 1 Karten ziehen oder mit 2 das ziehen beenden

8 Fazit

- Markus Hilbert (Lead/Architecture)

- Hannes Brottrager (Logging/Data/Exceptions/Console)

- Felix Ossmann (Testing)

- Eric Langer (Docs for Code)