TopTrumps – Beschreibung der Klassen

- Der Packagename für Ihr Projekt lautet **aMatrikelnummer**.
- Instanzvariablen sind `private` zu deklarieren.
- Die Signaturen der vorgegeben Methoden dürfen nicht verändert werden.
- Java Version am almighty ist Java 11; verwenden Sie diese um Import-Probleme beim Test zu vermeiden!

Zum Test dürfen Sie nur die **Basisimplementierung** mitbringen. Speichern Sie sich also gegebenenfalls einen Zwischenstand bevor Sie die Erweiterung für den Zusatzpunkt implementieren. **Abgabe der Basisimplementierung bis 16.01.22**

1 Basis

Folgendes ist Teil der Basisimplementierung

1.1 Klasse: VehicleCard

```
public class VehicleCard implements Comparable<VehicleCard> {
  public enum Category {...}

  private String name;
  private Map<Category, Double> categories;

  public VehicleCard(final String name, final Map<Category, Double>
      categories) {
   // throws IllegalArgumentException if name is null or empty.
   // throws IllegalArgumentException if categories is null or categories
       does not contain all Category values.
   // throws IllegalArgumentException if categories contains any null
       value or values less than 0.
   // set member variables - important: create a shallow copy of
       categories argument before assignment to this.categories (!)
  }
  // getters for _immutable_ class, no setters (!)
  public String getName() {/*return name*/}

  public Map<Category, Double> getCategories() {/*returns shallow copy (!)
       of this.categories*/}

  public static Map<Category, Double> newMap(double economy, double
      cylinders, double displacement, double power, double weight, double
      acceleration, double year) {/*factory method to create a new vehicle
      card's categories map*/}
```

```
  @Override
  public int compareTo(final VehicleCard other) {
    // compare by totalBonus
  }
  public int totalBonus(){
    // return total bonus of card, i.e. sum up the bonus per category and
        its corresponding category value using Category#bonus. Note that
        the calculated return value might be negative.
  }
  @Override
  public int hashCode() {
    // hash name and totalBonus (hint: Objects-class)
  }
  @Override
  public boolean equals(Object obj) {
    // true if obj is instance of VehicleCard and name and totalBonus
        match, false otherwise
  }
  @Override
  public String toString() {
    /* "- <name>(totalBonus) -> {<categories>}" e.g.:
  - Telsa(2) -> {Miles/Galon=1.0, Hubraum[cc]=1.0, Gewicht[lbs]=1.0,
      Beschleunigung=1.0, Zylinder=1.0, Baujahr[19xx]=0.0, Leistung[hp
      ]=1.0}*/
  }
}
```

1.2 Enumeration: VehicleCard – Category

```
public class VehicleCard implements Comparable<VehicleCard> {
  public enum Category {
    // values:
    // ECONOMY_MPG, CYLINDERS_CNT, DISPLACEMENT_CCM, POWER_HP, WEIGHT_LBS,
         ACCELERATION, YEAR;

    // ("Miles/Galon"), ("Zylinder"), ("Hubraum[cc]"), ("Leistung[hp]"),
        ("Gewicht[lbs]"), ("Beschleunigung"), ("Baujahr[19xx]")

    private final String categoryName;

    private Category(final String categoryName) {/*throws
        IllegalArgumentException if categoryName null or empty*/}

    // @Override for WEIGHT_LBS, ACCELERATION to return true (i.e. less is
         better)
    public boolean isInverted() {
        return false;
    }

    public int bonus(final Double value) { /*returns -value if this.
        isInverted, value otherwise*/ }

    @Override
    public String toString() { /*categoryName*/ }
  }

}
```

1.3 Klasse: FoilVehicleCard

```
public class FoilVehicleCard extends VehicleCard {

  private Set<Category> specials;

  public FoilVehicleCard(final String name, final Map<Category, Double>
      categories,final Set<Category> specials) {
   // calls super constructor
   // throws IllegalArgumentException if specials contains more than 3
       items or is null or empty
   // set member variables - important: create a shallow copy of specials
       argument before assignment to this.specials (!)
  }

  public Set<Category> getSpecials() {
    // returns shallow copy (!) of this.specials
  }

  @Override
  public int totalBonus(){
    // calculates the base classes totalBonus
    // for each special category add the (*absolute*) value of the
       corresponding category to the previously calculated totalBonus.
  }

  public String toString(){ // add * before and after special categories,
      e.g.
  // - Ladla(4) -> {*Miles/Galon*=1.0, Hubraum[cc]=1.0, Gewicht[lbs]=1.0,
      Beschleunigung=1.0, Zylinder=1.0, Baujahr[19xx]=0.0, *Leistung[hp
      ]*=1.0}
  }
}
```

1.4 Klasse: Player

```
public class Player implements Comparable<Player>{
  private String name;
  private Queue<VehicleCard> deck = new ArrayDeque();

  public Player(final String name) {
    // throw IllegalArgumentException if name is null or empty
  }

  public String getName() {...}
  public int getScore() {/*return sum of totalBonus of deck's cards (maybe
       negative)*/}

  public void addCards(final Collection<VehicleCard> cards) {/*add cards
      to end*/}
  public void addCard(final VehicleCard card) {/*add card to end*/}
  public void clearDeck() {...}
  public List<VehicleCard> getDeck() {/*returns a shallow copy (!) of this
      .deck*/}

  protected VehicleCard peekNextCard() {/*peek next card*/ }
  public VehicleCard playNextCard() {/*poll next card from deck*/}
```

```
  public int compareTo(final Player other) {
    // compare by name[case insensitive]
  }
  @Override
  public int hashCode() { /*hash(name[case insensitive])*/}
  @Override
  public boolean equals(Object obj) {/*auto generate but cmp name case
      insensitive*/}

  @Override
  public String toString() {/*format: name(score), one card per line, e.g
      .:
    Maria(73214):
      - Porsche 911(73054) -> {Beschleunigung=<val>, Zylinder=<val>, ...}
      - Renault Clio(160)-> {...}
  */}

  public boolean challengePlayer(Player p) {
    //throws IllegalArgumentException if p is null or p is this.
    //playNextCard from this and p.
    //Player who has higher scoring card, adds both of them to the end of
        his deck. Order is not important.
    //Player who has lower scoring card, loses card.
    //If draw, repeat until winner is found.
    //Winner gets all cards played. Order is not important.
    //Loser loses all cards played.
    //If one of the decks is empty before winner is found,
    //cards are returned to the original decks and the method returns
        false.
    //Returns true if this wins, false otherwise.
  }


  public static Comparator<Player> compareByScore() {....}
  public static Comparator<Player> compareByDeckSize() {....}
}
```

1.5 Tests

Erstellen Sie ein `Main.java`-File in dem Sie möglichst alle Funktionalitäten Ihrer Implementierung testen.
Hinweis: Die Korrektheit der Implementierung wird im Rahmen der Java-Klausur überprüft und ist für eine positive Bewertung zwingend erforderlich.

2 Zusatzaufgabe

- Sofern Sie den Zusatzpunkt erhalten möchten, beachten Sie bitte, dass in diesem Fall sowohl für die Basisimplementierung, als auch für die Zusatzimplementierung eine automatische **Plagiatsüberprüfung** durchgeführt wird.
- Projektpunkte tragen **nicht** zu den für einen positiven Abschluss erforderlichen Testpunkten bei, führen aber im Fall einer positiven Bewertung gegebenenfalls zu einer Verbesserung der Note.
- **Abgabe des Zusatzes bis 31.01.22**
- Denken Sie daran, sich Ihre Basisimplementierung für den Test zwischenzuspeichern bevor Sie mit dem Zusatz beginnen.

Die Zusatzaufgabe besteht darin das Spiel der Realität anzugleichen:

- Anstatt einen fiktiven `totalBonus` einer VehicleCard zu berechnen und dadurch den Gewinner zu ermitteln, soll der Gewinner der letzten Spielrunde bei der aktuellen Runde eine `Category` bestimmen, anhand derer die gegnerische Karte gestochen werden soll.
- Spieler können auch zusammengesetzte Strategien (i.e.: nested) implementieren, um eine möglichst gute `Category` zu wählen.

2.1 Erweiterung VehicleCard

Erstellen Sie eine Methode
`public int compareByCategory(VehicleCard other, VehicleCard.Category category)`
welche abhängig davon ob `category` invertiert ist oder nicht (`isInverted()`) den korrekten Wert retourniert (-1, 0, 1).

2.2 Strategy Interface

Erstellen Sie als Basis zur Auswahl einer `Category` das Interface `Strategy`:

```
public interface Strategy {
    VehicleCard.Category chooseCategory(final VehicleCard vehicleCard);
}
```

Implementieren Sie die zwei Strategien `AvgStrategy` sowie `RndStrategy`.

2.2.1 RndStrategy

Bei der `RndStrategy` handelt es sich um einen äußerst naiven Ansatz der per Zufallszahlengenerator (siehe z.B. `Random`) eine `Category` auswählt. Welche durchschnittliche Gewinnwahrscheinlichkeit erwarten Sie wenn drei Spieler:innen mit dieser Strategie ausreichend viele Spiele spielen? Gleichen Sie Ihre Erwartung mit Ihrem Ergebnis ab.

2.2.2 AvgStrategy

Bei der `AvgStrategy` besitzt der/die Spieler:in Vorwissen über andere `VehicleCard`s die über einen Konstruktor gesetzt werden können (i.e. `knownCards`). Die Implementierung der Methode `public VehicleCard.Category chooseCategory(VehicleCard card)` soll der Reihe nach alle `Category`-Werte durchgehen, zur entsprechenden `Category` den Durchschnitt aller bekannten `VehicleCard`s aus `knownCards` ermitteln und mit dem Wert `v` der entsprechenden `Category` aus

`card` vergleichen. Sollte der Wert `v` für invertierte Kategorien kleiner bzw. für nicht-invertierte Kategorien größer sein als der Durchschnitt, soll die `Category` retourniert werden.

Sollten alle Kategorien dem Durchschnitt entsprechen soll als Fallback die `RndStrategy` verwendet werden.

2.3 Erweiterung Player

- Fügen Sie der `Player`-Klasse eine Instanzvariable `strategy` hinzu. Sofern nicht anders definiert soll die `RndStrategy` als Default angenommen werden.
- Erstellen Sie eine Methode `public VehicleCard.Category chooseNextCategory()` welche für die oberste Karte im Deck des/der Spieler:in anhand der `strategy` die nächste `Category` auswählt ohne die Karte vom Deck des/der Spieler:in zu entfernen.
- Erstellen Sie Setter und Getter für `strategy`.
- Erstellen Sie einen Konstruktor der zusätzlich zu den bisherigen Parametern auch ein `Strategy`-Objekt entgegennimmt und `this.strategy` entsprechend setzt bzw. für `null`-Werte eine `IllegalArgumentException` wirft.

2.4 Vorgegebene Klassen & Dateien

Laden Sie sich die vorgegebenen Dateien für den Zusatz von Moodle herunter und lesen Sie sich in die Funktionsweise der jeweiligen Klassen ein.

cars.csv CSV Datei welche Daten zu den einzelnen `VehicleCard`s enthält.

SimpleCsvParser.java Vorgegebener und zu ergänzender CSV-Parser zum Einlesen der CSV Datei sowie zum Erzeugen der `VehicleCard`-Objekte.

SimpleTablePrinter.java Vorgegebene Klasse um die Ergebnisse tabellarisch anzuzeigen.

Game.java Simuliert für eine vorgegebene Anzahl von Spielern und Karten den Spielablauf.

2.5 Eingabeformat – CSV

Machen Sie sich mit dem vorgegebenen CSV[1] Format vertraut: Das CSV-File `cars.csv`[2] beginnt mit einer "Header"-Zeile welche das Format nochmals erklärt und beim Einlesen ignoriert werden soll.

```
name,economy (mpg),cylinders,displacement (cc),power (hp),weight (lb),0-60
     mph (s),year
AMC Ambassador Brougham,13,8,360,175,3821,11,73
AMC Ambassador DPL,15,8,390,190,3850,8.5,70
AMC Ambassador SST,17,8,304,150,3672,11.5,72
AMC Concord DL 6,20.2,6,232,90,3265,18.2,79
AMC Concord DL,18.1,6,258,120,3410,15.1,78
...
```

Wie im obigen Beispiel ersichtlich, besteht jede Zeile aus bis zu 8 Spalten, die durch Kommata (,) voneinander getrennt sind. Beachten Sie, dass die Tabelle auch unvollständige Datensätze beinhaltet die ignoriert werden sollen.

[1] https://de.wikipedia.org/wiki/CSV_(Dateiformat)
[2] Quelle: https://github.com/syntagmatic/parallel-coordinates

2.6 Klasse: SimpleCsvParser

Machen Sie sich mit der vorgegebenen Klasse `SimpleCsvParser` vertraut. Diese muss nur noch um das Erstellen der Objekte aus einer Zeile ergänzt werden (`parseLine(...)`). Falls notwendig, ergänzen Sie die Klasse um weitere Helper-Methoden. Beispielhafte-Verwendung der Klasse[3]:

```
public static void main() {

    final List<String> allLines = SimpleCsvParser.readAllLinesFrom("src/
        a123456/cars.csv");
    //... begin iterate over each line
    //..... SimpleCsvParser.parseLine(...) -> returns VehicleCard
    //... end
}
```

2.7 Klasse: Game

Implementieren Sie die Methode `public void writeStatistics(final OutputStream) throws IOException`. Die Methode soll aus den `playerStats` des `Game`-Objektes unter Verwendung der vorgegebenen `SimpleTablePrinter`-Klasse die folgende Statistik erzeugen[4]:

```
+--------+-------------+---------+----------+---------------------+
| Player | Strategy    | success | failures | success rate        |
+--------+-------------+---------+----------+---------------------+
| Paula  | RndStrategy | ?72     | ?886     | 0.?4009797060881736 |
+--------+-------------+---------+----------+---------------------+
| Alex   | AvgStrategy | ?460    | ?988     | 0.?530575539568345  |
+--------+-------------+---------+----------+---------------------+
| Franz  | RndStrategy | ?76     | ?918     | 0.?1352899069434503 |
+--------+-------------+---------+----------+---------------------+
```

Hinweise:

- Die Spalte **Strategy** wird über den Klassennamen der jeweils verwendeten Strategie mittels **Reflection** befüllt (siehe JavaDoc: Class::getSimpleName[5]).
- Die Spalte **success rate** ist das Verhältnis aus **success** zu den insgesamt durchgeführten Spielzügen des Spielers (unabhängig davon ob erfolgreich oder fehlgeschlagen).

2.8 Klasse: ExtensionMain

Erstellen Sie eine neue Java Datei `ExtensionMain.java` und entsprechenden Code um die CSV Datei einzulesen. Erstellen Sie ein neues `Game`-Objekt und fügen Sie diesem sowohl die zuvor eingelesenen `VehicleCard`s als auch 3 `Player`-Objekte hinzu. Führen Sie das Spiel in einer Schleife 100-mal hintereinander aus (`play`) und lassen Sie sich danach die Statistik für jeden Spieler ausgeben (`Game::writeStatistics`).

```
public class ExtensionMain {
    public static void main(String[] args) throws Exception {
        // read csv file and generate VehicleCards and save them to a deck
        // create new Game g
        // add deck to g
        // add three Players to g
        // play() the Game g 100 times
        // write the player stats to the console (Game::writeStatistics)
    }
}
```

[3] normalerweise werden Artefakte **nicht** im src-Folder abgelegt
[4] die tatsächlichen Werte wurden hier mit einem ? maskiert
[5] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Class.html

Beantworten Sie die folgende Fragen 3 & 4 in `ExtensionMain` als Kommentar:

1. Gleichen Sie Ihre vorige Annahme bzgl. der erwarteten Success-Rate mit dem Ergebnis ab und schätzen Sie ab ob Ihre Implementierung korrekt ist.
2. Ändern Sie die Strategie einer Spieler:in auf die `AvgStrategy` ab und überprüfen Sie das Ergebnis.
3. Wie groß muss das Sample der `knownCards` mindestens sein, damit **success rate** im Mittel ≥ 0.5 wird.
4. Hat es eine Auswirkung ob Sie das Sample für 3 der `knownCards` aus den ersten n Karten des CSV Files ziehen oder das Kartendeck zuvor durchmischen. **Geben Sie zusätzlich zum Kommentar den entsprechenden Code mit ab**.

2.9 Weitere Strategie

Erzeugen Sie eine weitere Strategie die besser funktioniert als `AvgStrategie`.

Hinweis: Bei Kartenspielen soll es sich generell bewährt haben sich die Karten zu merken die bereits gespielt wurden bzw. diejenigen welche die Gegner noch ausspielen könnten.