Spieltheorie

Ziel der Aufgabe

Ziel der Aufgabe ist es, ein Framework zu schreiben, mithilfe dessen sich rundenbasierte Spiele simulieren lassen, bei denen die Spieler eine endliche Menge an möglichen Spielzügen zur Verfügung haben. Dabei verfolgt jeder Spieler (also Bot) eine bestimmte Strategie, die ihm vorgibt, welchen Zug er als nächstes ausführt. Diese Entscheidung kann entweder zufällig, oder aber auf Basis der Züge der Spieler in den vorherigen Runden getroffen werden.

Exemplarisch soll zudem das Spiel Schere-Stein-Papier implementiert werden.

Hinweise zur Implementierung:

- Falls ein Ihrer Methode übergebener Parameter den Wert `null` hat, soll Ihre Methode eine `NullPointerException` mit aussagekräftiger Fehlermeldung werfen.
- Achten Sie darauf, dass keine Ihrer Methoden `null` zurückgibt, außer die Aufgabenstellung fordert es explizit.
- Falls nicht anders angegeben, überschreiben Sie Ihre `toString()` Methode so, dass Sie das gleiche zurückgibt wie die Methode `name()`.
- In dieser Aufgabe müssen Sie Java Generics benutzen. Lesen Sie sich also ein, zum Beispiel hier (https://docs.oracle.com/javase/tutorial/java/generics/index.html).
- Für `toString()` Methoden sind oft Formatierungen angegeben. Dabei sind zu ersetzenden Parameter durch '<' und '>' und der komplette String durch Anführungszeichen begrenzt. Diese Zeichen dienen nur zum Abgrenzen, haben also in der Ausgabe nichts zu suchen, z.B. "<Name>" => Herbert. Hier kann `String.format()` sehr helfen. Sollte sich das Format über mehrere Zeilen erstrecken, wird als Trennzeichen immer '\n' genutzt (nicht `System.lineSeparator`!).
- Nutzen Sie für Listen keine `LinkedList` sondern `ArrayList`, da diese deutlich effizienter sind. Zu den Gründen dafür sei auf Stackoverflow verwiesen, da gibt es einen sehr schönen Beitrag zu dem Thema.

Hinweise zu den Tests:

- Die Tests kompilieren erst, wenn alle Klassen und Methoden erstellt wurden.
- Für jede in der Aufgabenstellung mit Punkten annotierte Klasse/Methode existieren ein oder mehrere Tests. Dabei ist die Zuordnung der Tests zum jeweiligen Aufgabenteil eindeutig durch die Namensgebung der Tests gegeben.
- Ein roter Test bedeutet einen Fehler in Ihrem Code. Die Umkehrung gilt nicht!

1. Generische Klassen (12 Punkte)

Zunächst sollen die beiden Klassen `Player` und `GameRound` im Paket `jpp.gametheory.generic` implementiert werden, die unabhängig vom Typ des Spiels sind.
Im Repository sind im Paket `jpp.gametheory.generic` bereits die Interfaces `IPlayer` und `IGameRound` vorgegeben.
Außerdem finden Sie dort noch weitere Interfaces, deren Dokumentation für die Implementierung der beiden Klassen hilfreich sein kann.
Die vorgegebenen Interfaces dürfen nicht verändert werden.

public class Player<C extends IChoice> implements IPlayer<C> (4 Punkte)

Repräsentiert einen Spieler in einem Spiel, der eine bestimmte Strategie verfolgt.

- Schreiben Sie mindestens einen Konstruktor der Form `public Player(String name, IStrategy<C> strategy)`, der einen Spieler mit Namen und Strategie, die er zu verfolgen hat, erzeugt.
- Implementieren Sie die im Interface vorgegebenen Methoden.
- Implementieren Sie die Methode `compareTo()` so, dass Spieler aufsteigend nach ihren Namen sortiert werden.
- Überschreiben Sie die Methoden `equals()` und `hashCode()` gemäß der Java-Konventionen so, dass zwei Spieler genau dann gleich sind, wenn Sie den gleichen Namen haben.
- Des Weiteren soll die Methode `toString()` einen String der Form "<Spielername>(<Strategiename>)" zurückgeben.
- Achten Sie bei Ihrer Implementation darauf, dass der Name eines Spielers im Nachhinein nicht mehr geändert werden kann.

public class GameRound<C extends IChoice> implements IGameRound<C> (8 Punkte)

Stellt eine Runde eines Spiels dar und speichert die Spielzüge ab, die die Spieler in dieser Runde gemacht haben.
Achten Sie darauf, dass Ihre Klasse nicht von außen modifiziert werden kann, indem auf die Rückgabewerte der Methoden zugegriffen wird.

- Schreiben Sie mindestens einen Konstruktor der folgenden Form: `public GameRound(Map<IPlayer<C>, C> playerChoices)`. Dieser soll eine neue Runde mit den entsprechenden Player->Choice Mappings erstellen. Falls die Map weniger als einen Spieler enthält, soll eine aussagekräftige `IllegalArgumentException` geworfen werden.
- Überschreiben Sie außerdem die Methode `toString()`, sodass ein String der folgenden Form "(<Spieler-Name> -> <Choice-Name>, <Spieler-Name 2> -> <Choice-Name 2>, ...)" zurückgegeben wird. Die Spieler sollen nach ihren Namen sortiert ausgegeben werden.

2. Schere-Stein-Papier (48 Punkte)

Nun geht es darum, konkret das Spiel Schere-Stein-Papier zu simulieren. Dabei liegt das Hauptaugenmerk auf den verschiedenen möglichen Strategien.
Implementieren Sie die restlichen Interfaces in den folgenden Klassen. Richten Sie sich auch hier wieder nach der Dokumentation in den Interfaces.
Beachten Sie, dass bei dieser Implementation mehr als zwei Spieler spielen können.

Grundklassen (12 Punkte)

Die folgenden Klassen sollen im Paket `jpp.gametheory.rockPaperScissors` implementiert werden. Dazu finden Sie im Paket `jpp.gametheory.generic` bereits entsprechende Interfaces, in denen die zu implementierenden Methoden vorgegeben sind.

- **public enum RPSChoice implements IChoice (2 Punkte)**
  Stellt die Auswahlmöglichkeiten dar, die ein Spieler in einer Runde hat.
  Die Enum soll die Felder `ROCK`, `PAPER` und `SCISSORS` beinhalten.

- **public class RPSReward implements IReward<RPSChoice> (10 Punkte)**
  Klasse die eine Methode zur Berechnung des Profits bereitstellt.
  Nach den üblichen Regeln von Schere-Stein-Papier (https://de.wikipedia.org/wiki/Schere,_Stein,_Papier) gewinnt Papier gegen Stein, Stein gegen Schere und Schere gegen Papier.
  Der Profit einer Runde für einen Spieler A berechnet sich wie folgt:

Für jeden Spieler, gegen den A gewonnen hat, erhält er 2 Punkte.
Für jeden Spieler, gegen den A verloren hat, verliert er 1 Punkt.
Bei einem Unentschieden zwischen zwei Spielern bekommt keiner von beiden einen Punkt.

Strategien (36 Punkte)

Für die Aufgabe sind nur einige wenige Strategien verlangt, die im Paket `jpp.gametheory.rockPaperScissors.strategies` implementiert werden sollen.
Sie dürfen aber zu Testzwecken gerne noch weitere entwerfen.

- **public class SingleChoice implements IStrategy<RPSChoice> (4 Punkte)**
  - Diese Klasse soll einen Konstruktor der Form `public SingleChoice(RPSChoice choice)` besitzen.
  - `String name()` soll "Always <Name der Choice>" zurückgeben.
  - `RPSChoice getChoice(IPlayer<RPSChoice> player, List<IGameRound<RPSChoice>> previousRounds)` soll immer den im Konstruktor übergebenen Zug zurückgeben.
- **public class CircleChoice implements IStrategy<RPSChoice> (8 Punkte)**
  - Diese Klasse soll einen Default-Konstruktor besitzen.
  - `String name()` soll "Circle Choice" zurückgeben.
  - `RPSChoice getChoice(IPlayer<RPSChoice> player, List<IGameRound<RPSChoice>> previousRounds)` soll die Folge ROCK -> PAPER -> SCISSORS -> ... spielen. D.h. wenn der Spieler in der vorherigen Runde ROCK gespielt hat soll er als nächstes PAPER, bei PAPER als nächstes SCISSORS und bei SCISSORS als nächstes ROCK spielen (usw). Falls noch keine Runden gespielt wurden (d.h. es wird eine leere Liste übergeben), soll ROCK zurückgegeben werden.
    Bestimmen Sie Ihre neue Entscheidung immer auf Basis der übergebenen Runden, damit Sie nicht mögliche Undo-Vorgänge übersehen.
- **public class MostCommon implements IStrategy<RPSChoice> (12 Punkte)**
  - Diese Klasse soll einen Konstruktor der Form `public MostCommon(IStrategy<RPSChoice> alternate)` besitzen.
  - `String name()` soll "Most Common Choice (Alternate: <Name der Alternativstrategie>)" zurückgeben.
  - `RPSChoice getChoice(IPlayer<RPSChoice> player, List<IGameRound<RPSChoice>> previousRounds)` soll die Wahl zurückgeben, die bisher am häufigsten unter allen Spielern gespielt wurde. Bei Gleichstand soll die im Konstruktor übergebene alternative Strategie ausgeführt werden.
- **public class MostSuccessful implements IStrategy<RPSChoice> (12 Punkte)**
  - Diese Klasse soll einen Konstruktor der Form `public MostSuccessful(IStrategy<RPSChoice> alternate, IReward<RPSChoice> reward)` besitzen.
  - `String name()` soll "Most Successful Choice (Alternate: <Name der Alternativstrategie>)" zurückgeben.
  - `RPSChoice getChoice(IPlayer<RPSChoice> player, List<IGameRound<RPSChoice>> previousRounds)` soll die Wahl zurückgeben, die im bisherigen Spiel am erfolgreichsten unter allen Spielern war (Bewertet nach der übergebenen Reward-Klasse). Dabei sollen die Profite der einzelnen Runden aufsummiert werden. Bei Gleichstand soll stattdessen die im Konstruktor übergebene alternative Strategie ausgeführt werden.

3. Spiellogik (40 Punkte)

Das Herzstück des Frameworks bildet die Klasse **`Game<C extends IChoice>`**, die im Paket `jpp.gametheory.generic` implementiert werden soll.
Die Klasse soll die folgenden Methoden unterstützen:

- **public Game(Set<IPlayer<C>> players, IReward<C> reward) (2 Punkte)**
  Erstellt eine neues Game. Dabei enthält `players` alle Spieler, die an diesem Spiel teilnehmen und `reward` eine Ertragsfunktion, die zur Berechnung des Profits verwendet werden soll. Falls `players` weniger als einen Spieler enthält, soll eine aussagekräftige `IllegalArgumentException` geworfen werden.
- **public Set<IPlayer<C>> getPlayers() (1 Punkt)**
  Gibt ein Set aller Spieler zurück.
- **public IGameRound<C> playRound() (8 Punkte)**
  Simuliert eine Runde des Spiels und gibt diese zurück.
- **public void playNRounds(int n) (1 Punkt)**
  Simuliert mehrere Runden des Spiels. Soll eine `IllegalArgumentException` werfen, falls `n < 1`.
- **public Optional<IGameRound<C>> undoRound() (2 Punkte)**
  Macht die letzte Runde des Spiels rückgängig und gibt diese aus. Falls noch keine Runde gespielt wurde soll ein leeres Optional zurückgegeben und nichts verändert werden.
- **public void undoNRounds(int n) (1 Punkt)**
  Macht mehrere Runden des Spiels rückgängig. Falls es weniger als `n` Runden gibt, sollen alle Runden entfernt werden, aber keine Exception geworfen werden. Es soll jedoch eine `IllegalArgumentException` geworfen werden, falls `n < 1`.
- **public List<IGameRound<C>> getPlayedRounds() (4 Punkte)**
  Gibt eine Liste aller gespielten Runden aus, wobei die letzte Runde an letzter Stelle steht.
- **public int getPlayerProfit(IPlayer<C> player) (8 Punkte)**
  Gibt den momentanen Profit für einen bestimmten Spieler aus. Dieser ist die Summe der Profits aus den einzelnen Runden. Zu Beginn ist der Profit aller Spieler 0. Achten Sie außerdem darauf, dass auch der Profit korrekt zurückgesetzt wird, wenn Runden rückgängig gemacht werden. Die Methode soll eine `IllegalArgumentException` werfen, falls der Spieler nicht am Spiel teilnimmt.
- **public Optional<IPlayer<C>> getBestPlayer() (5 Punkte)**
  Gibt den Spieler mit dem zur Zeit höchsten Profit aus oder ein leeres Optional, falls es keinen eindeutigen Besten gibt.
- **public String toString (8 Punkte)**
  Soll einen String der unten dargestellten Form zurückgeben. Des Weiteren sollen die Spieler nach ihrem Profit sortiert ausgegeben werden. (Der beste Spieler als erstes, bei gleichem Profit soll aufsteigend nach Namen der Spielers sortiert werden). Achten Sie auch auf eine effiziente Implementation, z.B. bietet es sich hier an, einen `StringBuilder` zu nutzen, um sehr viele String-Kopien zu sparen.
  ```
  Spiel nach <N> Runden:
  Profit : Spieler
  <Profit von Spieler A> : <String von Spieler A>
  <Profit von Spieler B> : <String von Spieler B>
  ...
  ```

Hinweis: Es empfiehlt sich, für die Speicherung der gespielten Runden eine Implementation von List zu wählen, die gleichzeitig eine LIFO-Datenstruktur unterstützt, da so Runden einfach hinzugefügt und entfernt werden können.

Hinweis 2: Achten Sie darauf, dass Instanzen Ihrer Klasse nicht von außen modifiziert werden können, indem auf die Rückgabewerte der Methoden zugegriffen wird.

Viel Erfolg!
