Programmentwurf
UNO

Namen: Kilders, Noah / Leuck, Jens
Matrikelnummern: 7305144 / 8535877

Abgabedatum: 28. Mai 2023

Kapitel 1: Einführung

Übersicht über die Applikation

Die Applikation realisiert das Kartenspiel UNO. Da sowohl ein Server als auch ein Client implementiert wurde, können auch Personen auf verschiedenen Geräten über das Internet miteinander spielen. Zuerst muss dafür jemand die Anwendung im Server-Modus starten, danach können sich beliebig viele Mitspieler über den Client-Modus der Anwendung mit dem Server verbinden.

Wie startet man die Applikation?

Voraussetzungen

- JRE 19 oder höher

Anwendung starten

Server

1. `mvn install`
2. `java -jar ./main/target/main-1.0.jar`
3. Spielernamen in der Konsole eingeben
4. „0“ eingeben, um die Anwendung im Server-Modus zu starten

Client

1. `mvn install`
2. `java -jar ./main/target/main-1.0.jar`
3. Spielernamen in der Konsole eingeben
4. „1“ eingeben, um die Anwendung im Client-Modus zu starten
5. IP-Adresse des Servers eingeben, mit dem eine Verbindung hergestellt werden soll

Wie testet man die Applikation?

```
mvn test
```

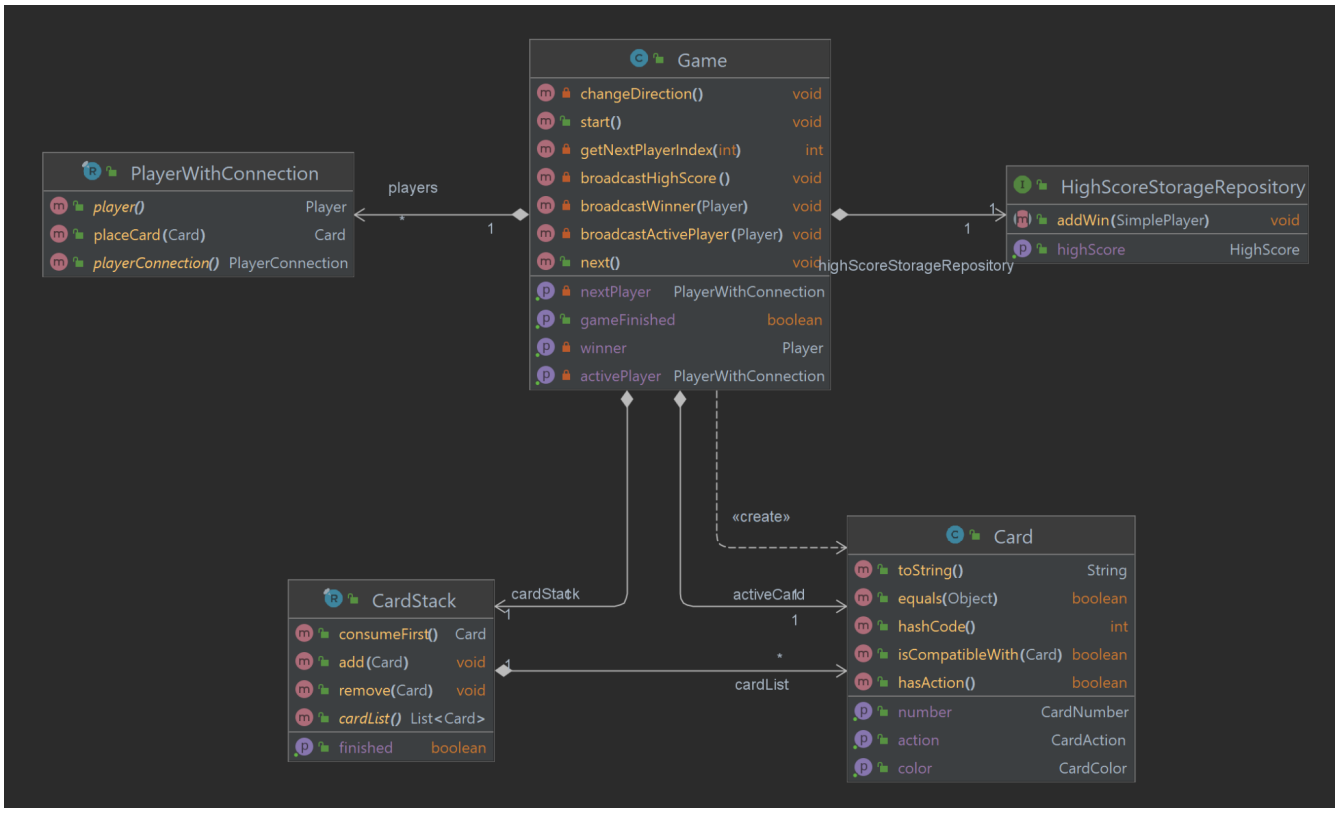
Kapitel 2: Clean Architecture

Was ist Clean Architecture?

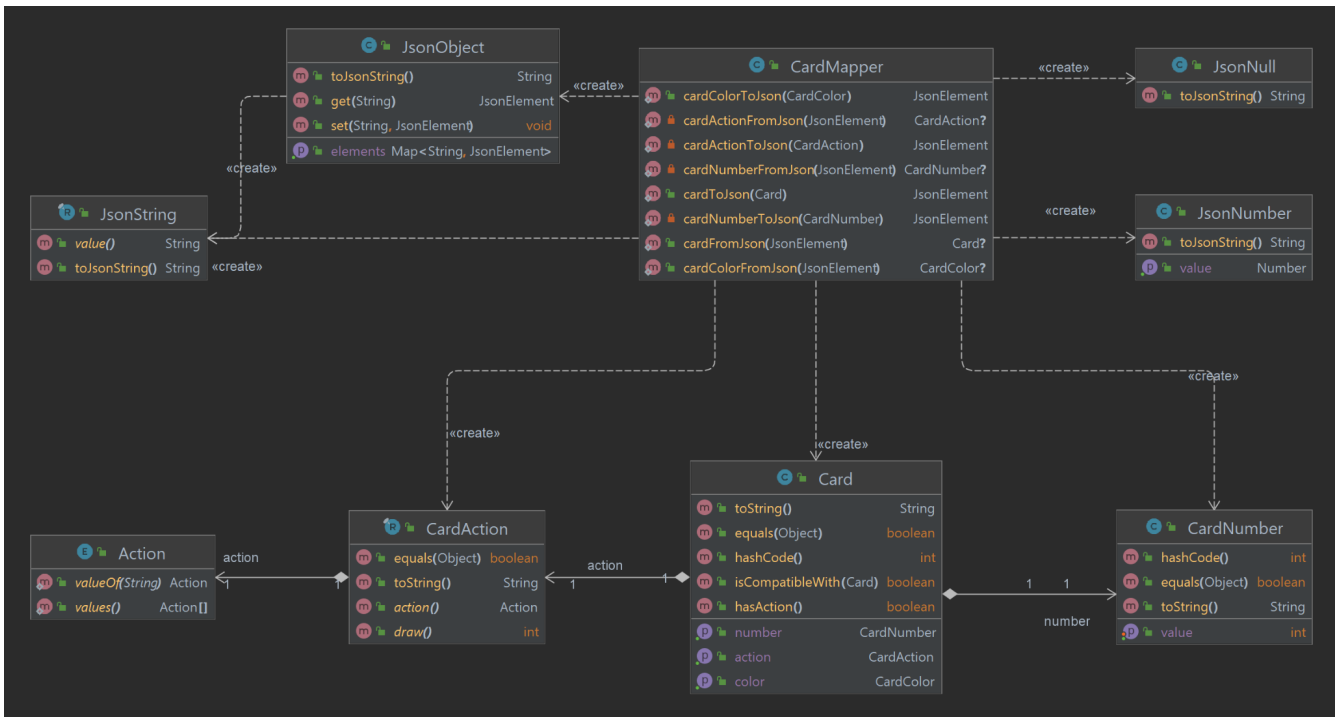
Bei Clean Architecture wird Software wie eine Zwiebel in Schichten unterteilt. Diese Schichten heißen von innen nach außen: Domain, Application, Adapters und Plugins. Die Domain-Schicht beinhaltet Dinge wie Objekte und fachliche Regeln, die innerhalb einer Organisation immer gelten und somit selten bis gar nicht verändert werden. In der Application-Schicht wird die anwendungsspezifische Geschäftslogik (Use Cases) implementiert sowie der Fluss von Daten und Aktionen gesteuert. Die Adapters-Schicht dient zur Vermittlung von Aufrufen und Daten zwischen den ihr unter-/überlegenen Schichten, um diese voneinander zu entkoppeln. Die Plugins-Schicht besteht aus Komponenten wie Datenbanken, Benutzeroberflächen oder sonstigen Frameworks, enthält gar keine Geschäftslogik und kommuniziert ausschließlich mit der Adapters-Schicht.
Der Vorteil von Clean Architecture ist, dass Komponenten, die nur entfernt mit dem Kerngeschäft (Domain) in Verbindung stehen, “ganz am Rand” platziert werden, damit sie mit möglichst geringem Aufwand ersetzt werden können.

Analyse der Dependency Rule

Positiv-Beispiel: Dependency Rule



Das obige Bild zeigt einen Ausschnitt des UML-Diagramms der Anwendung. Die oberen drei Klassen gehören zur Application-Schicht, die unteren beiden zur Domain-Schicht. Die Dependency Rule ist eingehalten, da die Abhängigkeiten zwischen den Schichten ausschließlich “von oben nach unten” gehen.



Dieses Bild zeigt einen weiteren Ausschnitt, diesmal die Adapter-Schicht (oben) und die Domain-Schicht (unten). Da die Abhängigkeiten erneut ausschließlich “von oben nach unten” verlaufen, ist auch hier die Dependency Rule eingehalten.

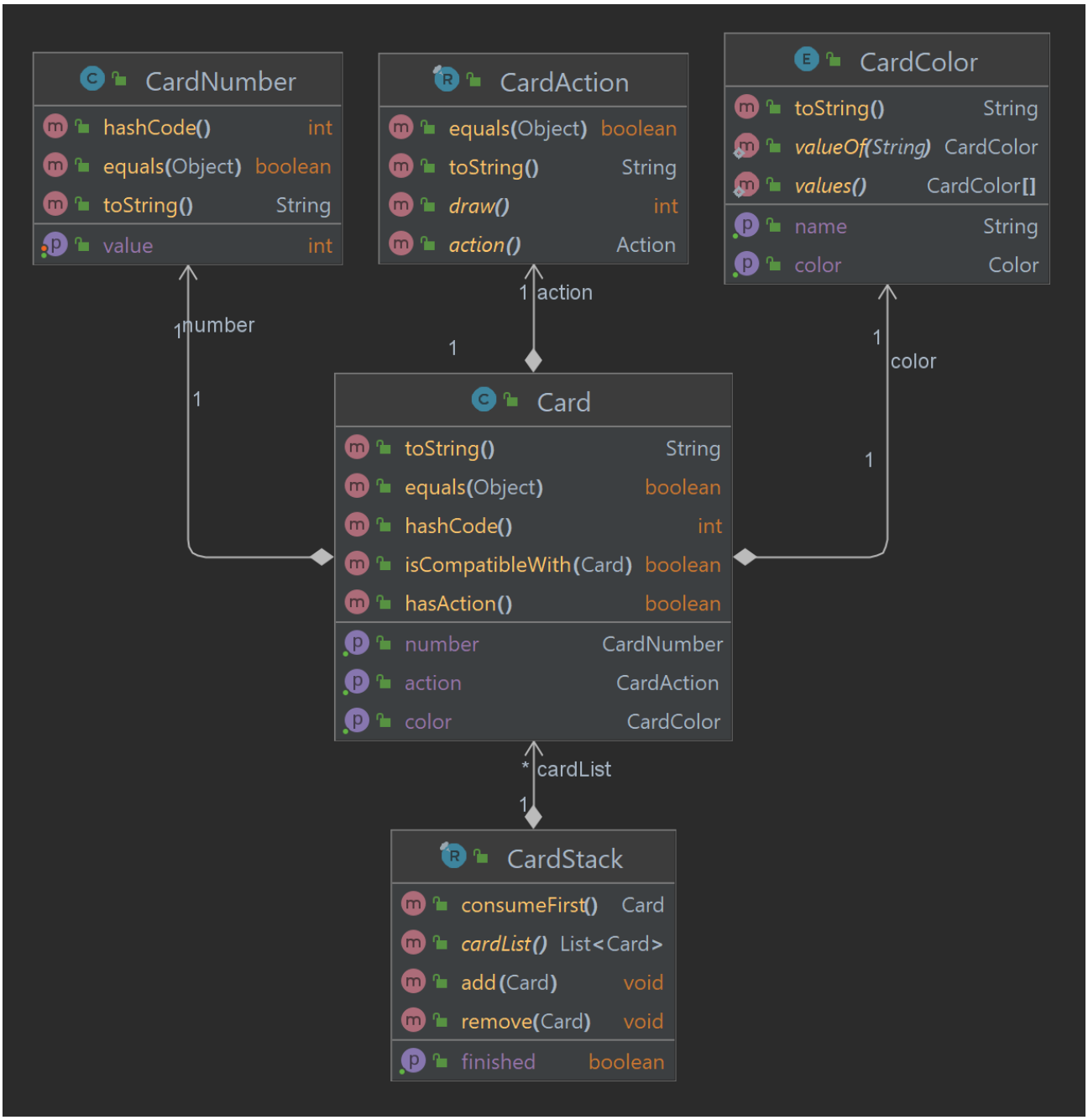
Negativ-Beispiel: Dependency Rule

Aufgrund der Dependencies in den `pom.xml`-Dateien unseres Projekts ist es nicht möglich, dass niedere Schichten von höheren Schichten abhängen. Die Dependency Rule kann somit nicht verletzt werden.

Analyse der Schichten

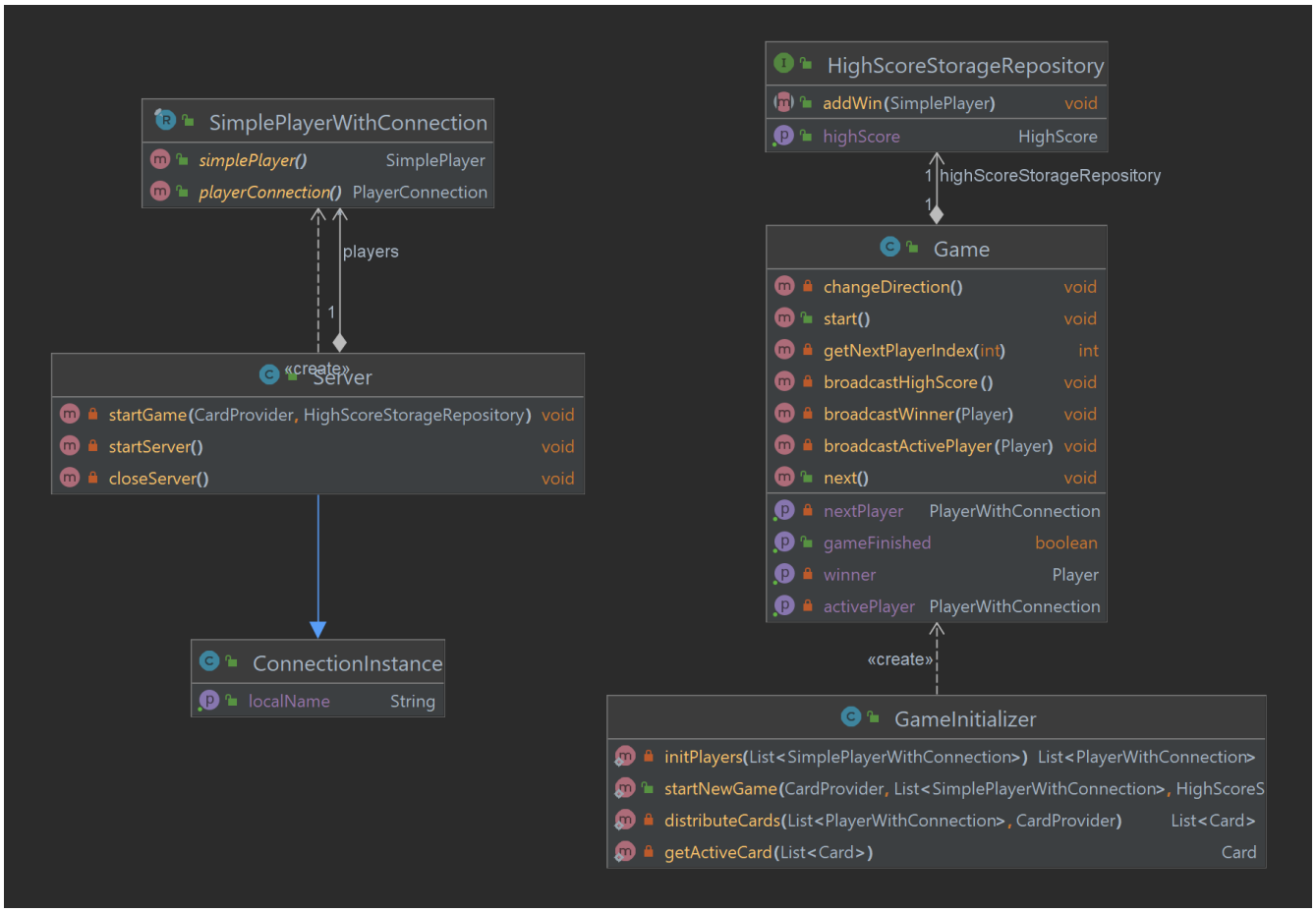
Schicht: Domain

Klasse: `Card`
Aufgabe: Die Klasse repräsentiert eine UNO-Karte mit Farbe, Nummer und Aktion (Farbe wechseln, Karten ziehen). Mit der Klasse `CardStack` können mehrere Karten zusammengefasst werden (bspw. Nachziehstapel).
Einordnung: Karten sind der zentrale Bestandteil des Spiels und werden sich niemals ändern. Aus diesem Grund befinden sie sich in der innersten Schicht.



Schicht: Application

Klasse: `Server`
Aufgabe: Der Server ist “Gastgeber” jeder Spielrunde. Vor Beginn der Runde können sich Clients mit ihm verbinden, bis der Server das Spiel startet.
Einordnung: Der Server ist ein zentraler Baustein der Anwendung, jedoch nicht so, dass er zur Domain-Schicht gehören könnte. Sollten in Zukunft neue Funktionalitäten (bspw. Countdown vor Start einer Runde) hinzugefügt werden, so ist es gut möglich, dass der Server bearbeiten werden muss.

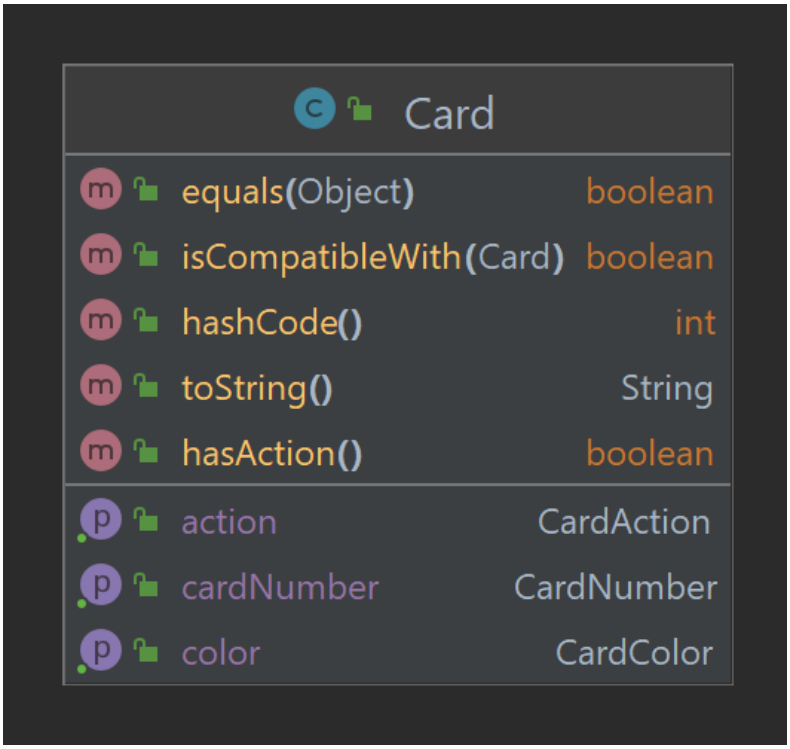


Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

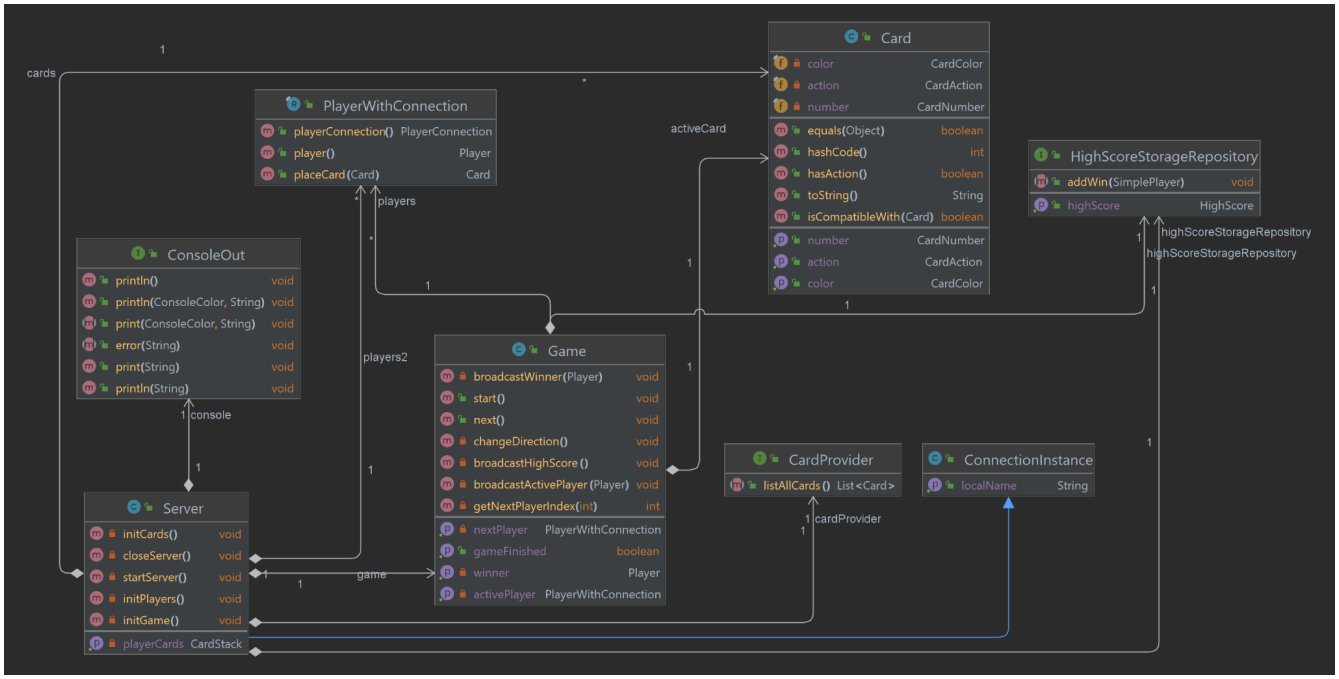
Positiv-Beispiel

Klasse: `Card`
Aufgabe: Informationen einer UNO-Karte (Farbe, Nummer, Aktion) speichern; Überprüfung, ob zwei Karten miteinander kompatibel sind (aufeinander gelegt werden können)



Negativ-Beispiel

Klasse: `Server`

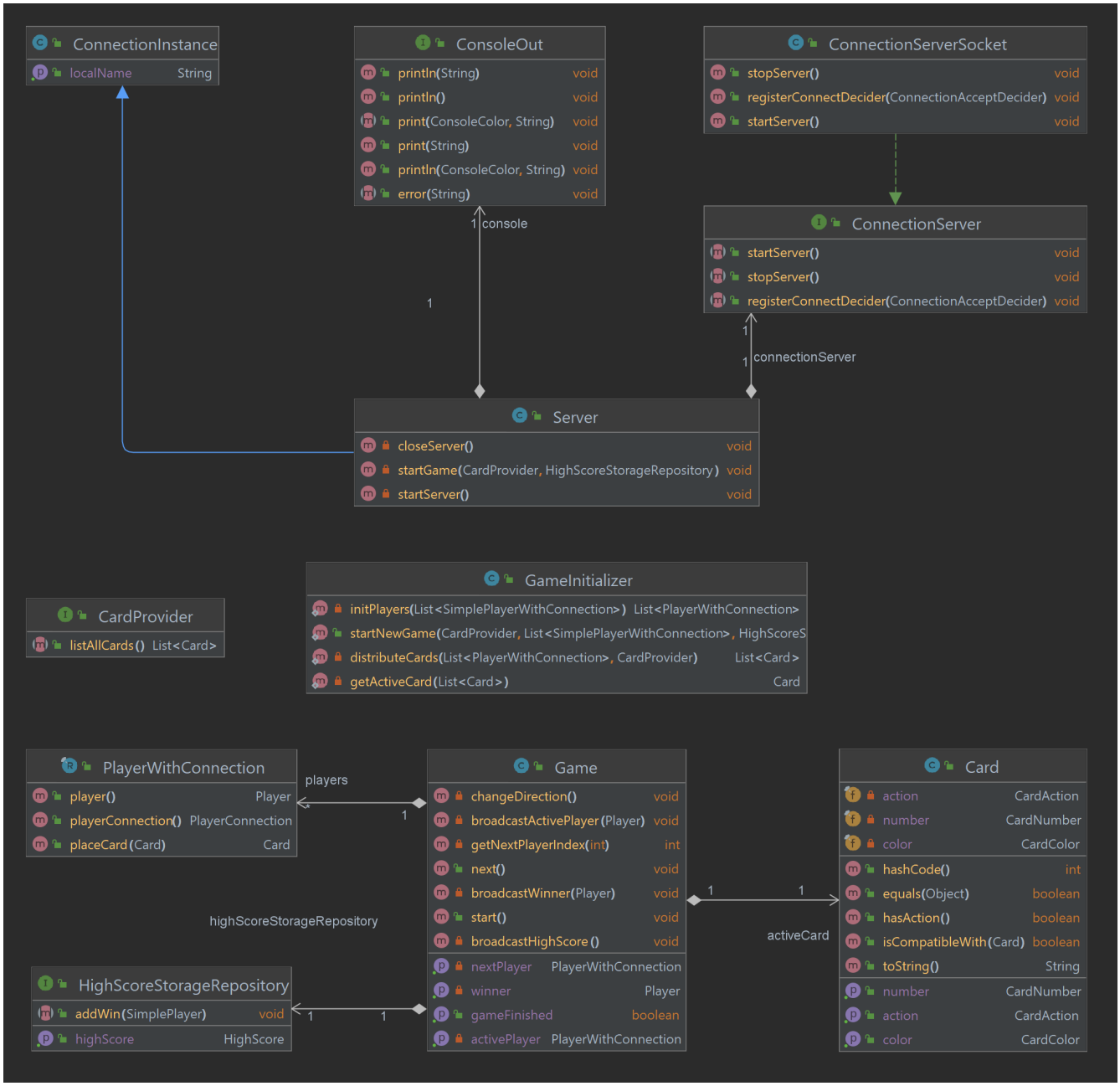


Aufgaben:

- Server Socket verwalten (widerspricht auch der Dependency Rule → Socket gehört in Plugins-Schicht)
- Karten an Mitspieler verteilen
- Spiel erstellen und starten

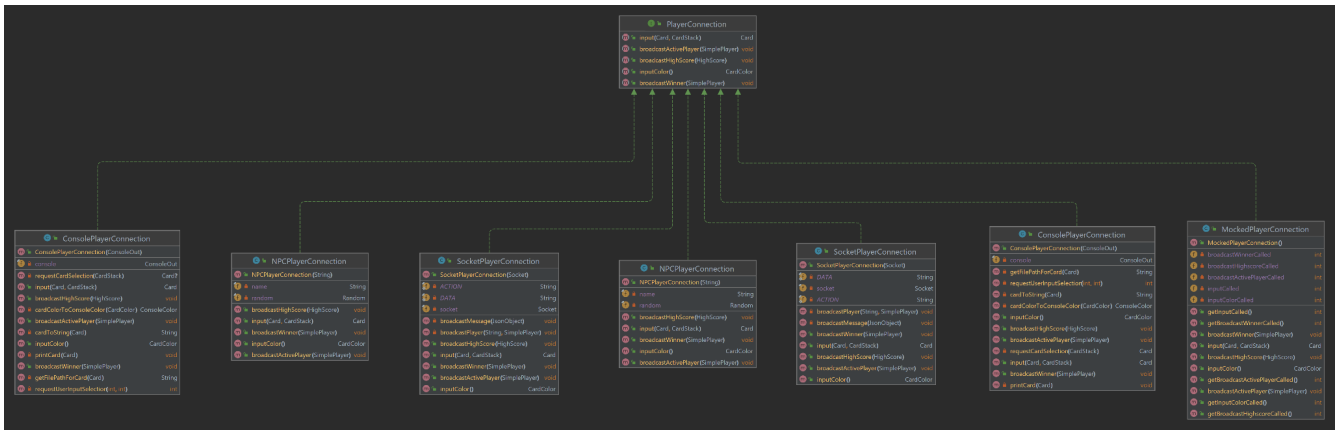
Lösung:

- Server Socket in Plugins-Schicht verschieben (ConnectionServerSocket) und für Application-Schicht abstrahieren (`ConnectionServer`)
- `GameInitializer` einführen, der Kartenverteilung und Spielerstellung übernimmt



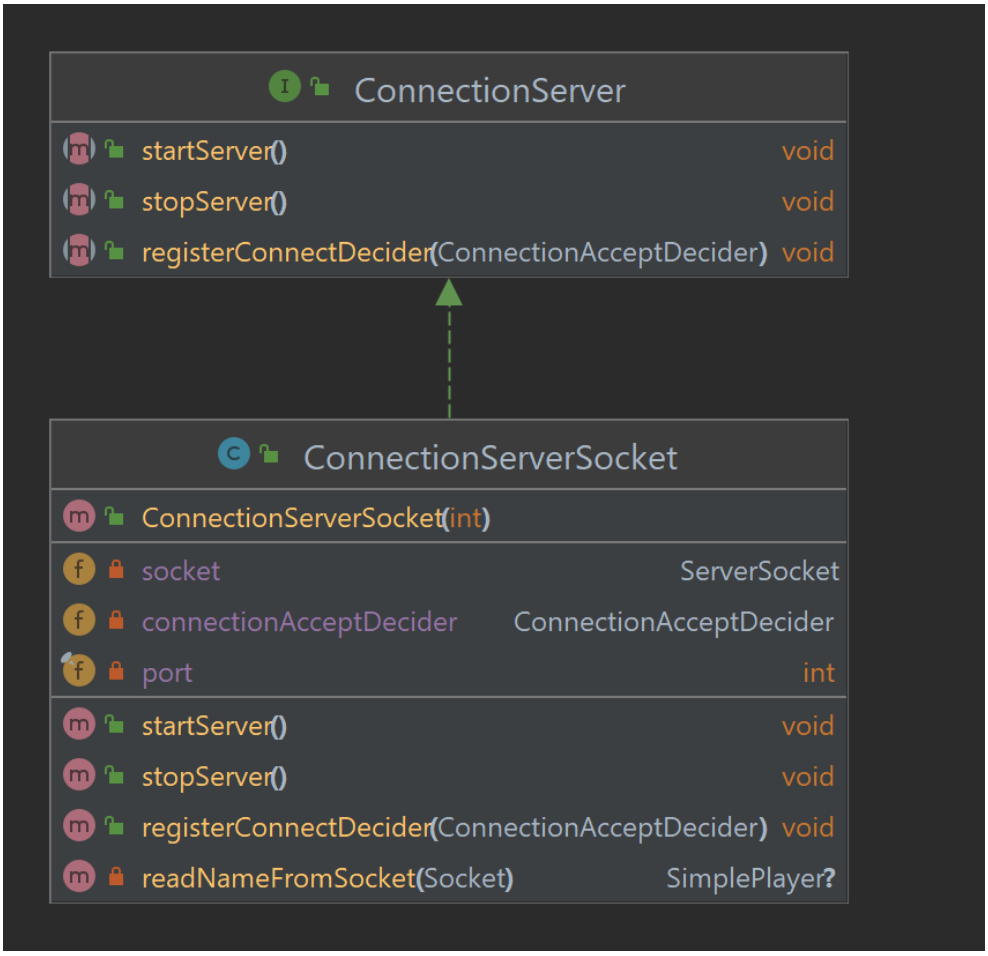
Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel

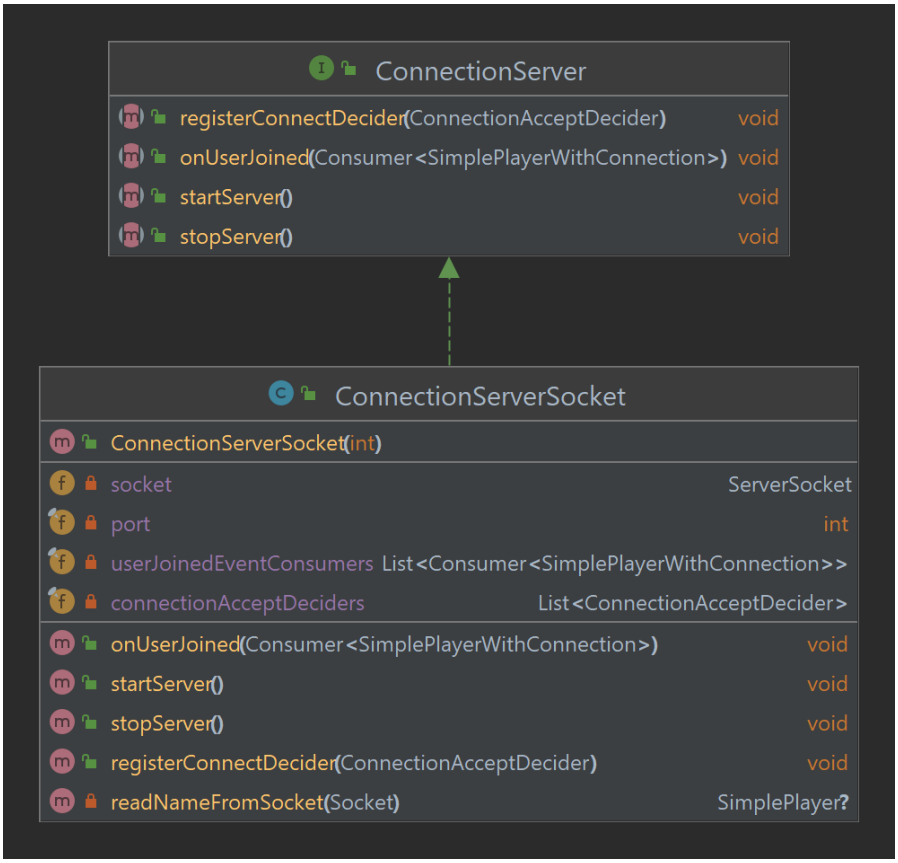


Das Interface `PlayerConnection` beinhaltet alle Funktionen, die für die Benutzerinteraktion implementiert werden müssen. Wenn neue Schnittstellen, wie beispielsweise einen Spielpartner als KI, oder eine Technologie anstatt eines Websockets implementiert werden soll, ist eine Erweiterung extrem einfach und ohne Veränderung von anderem Code möglich.

Negativ-Beispiel



Aktuell ist eine Erweiterung der Logik zum Akzeptieren oder Ablehnen von Verbindungen nicht möglich, so muss, um dies zu erweitern, um beispielsweise einen Login, oder weitere Prüfungen (nicht mehr als x Spieler, Spieler dürfen nicht gleich heißen, usw.) nicht möglich. Dies ist nicht möglich, da keine Registrierung von mehreren `ConnectionAcceptDecider` möglich ist und kein eigenständiges Event gesendet wird, wenn alle Decider erfolgreich die Verbindung akzeptieren.

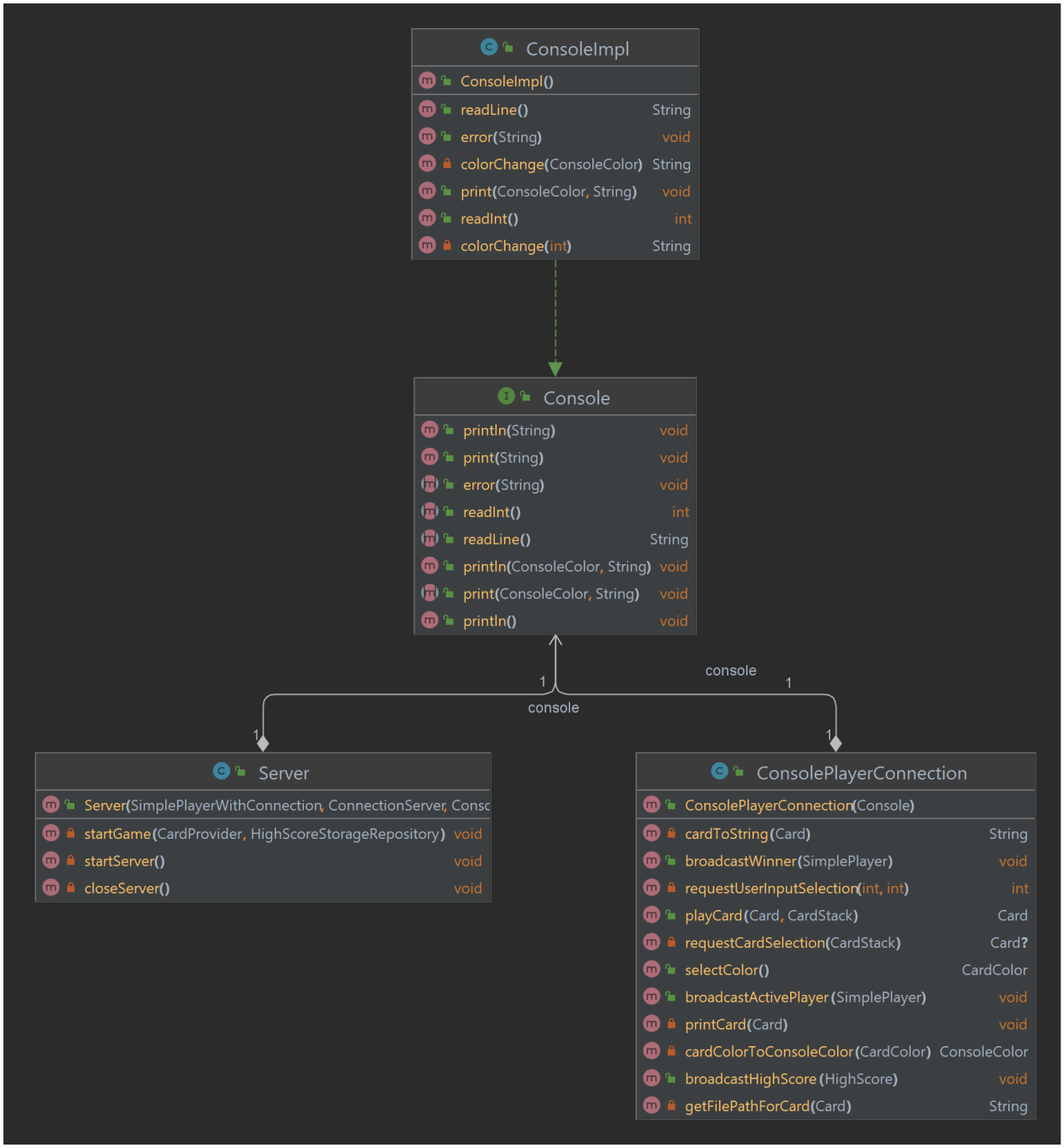


Umgesetzt wurde das, indem mehrere `ConnectionAcceptDecider` als Liste registrierbar sind. Wenn alle Decider akzeptiert haben, wird über die Consumer, welche durch `onUserJoined` registrierbar sind, ein Event gesendet, so ist eine Erweiterung einfacher möglich.

Analyse Liskov-Substitution- (LSP), Interface-Segregation (ISP), Dependency-Inversion-Principle (DIP)

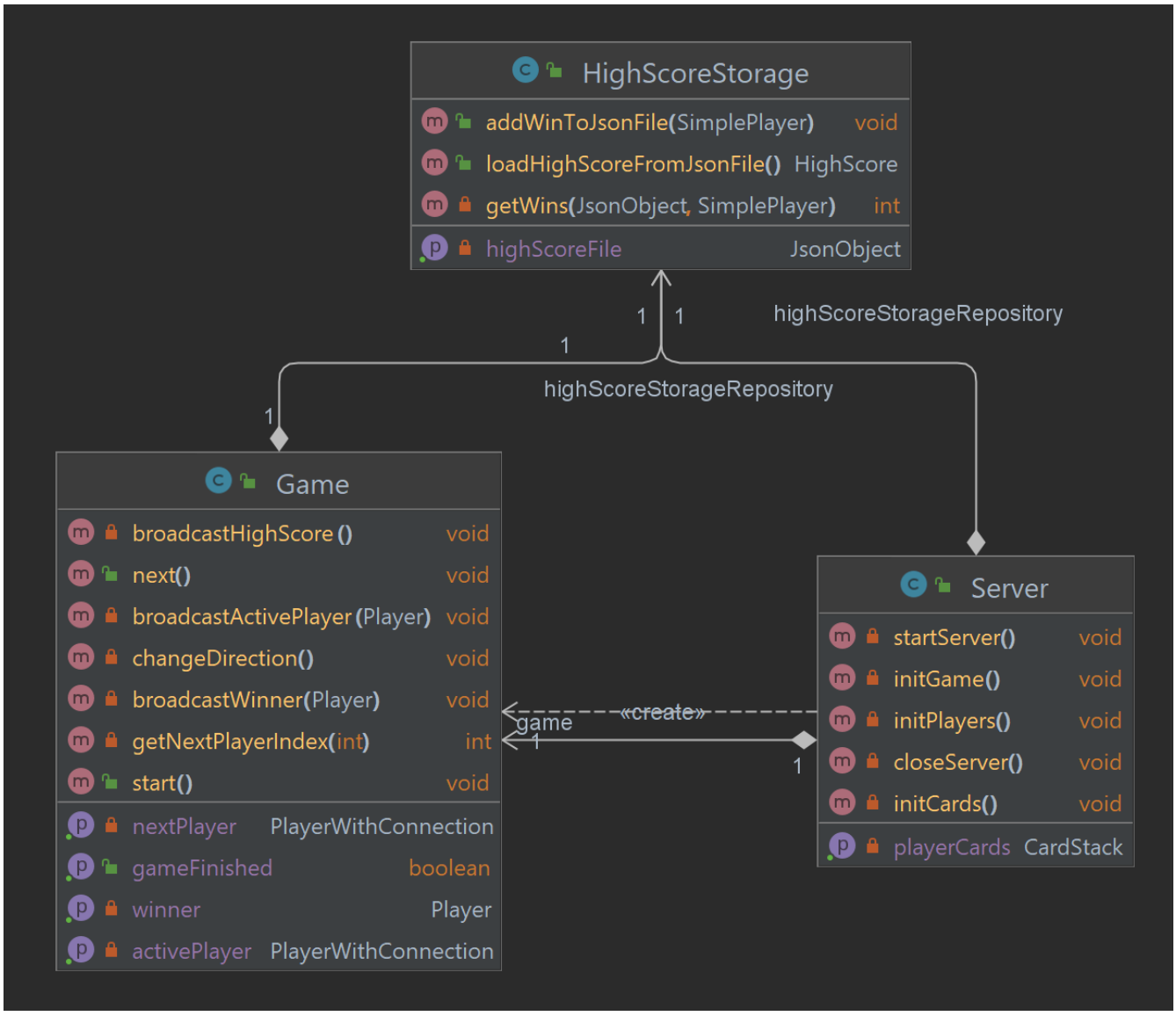
Positiv-Beispiel

Klassen: `ConsoleAdapter` in Verbindung mit `Server`, `Client` und `ConsolePlayerConnection`
Begründung: DIP ist hier erfüllt, da die Klassen nicht direkt, sondern über das Interface `ConsoleOut` abstrahiert mit dem `ConsoleAdapter` interagieren. Sie müssen sich somit nicht an die Implementierung einer bestimmten Konsole anpassen, sondern nur an die abstrakte Schnittstelle `ConsoleOut`.



Negativ-Beispiel

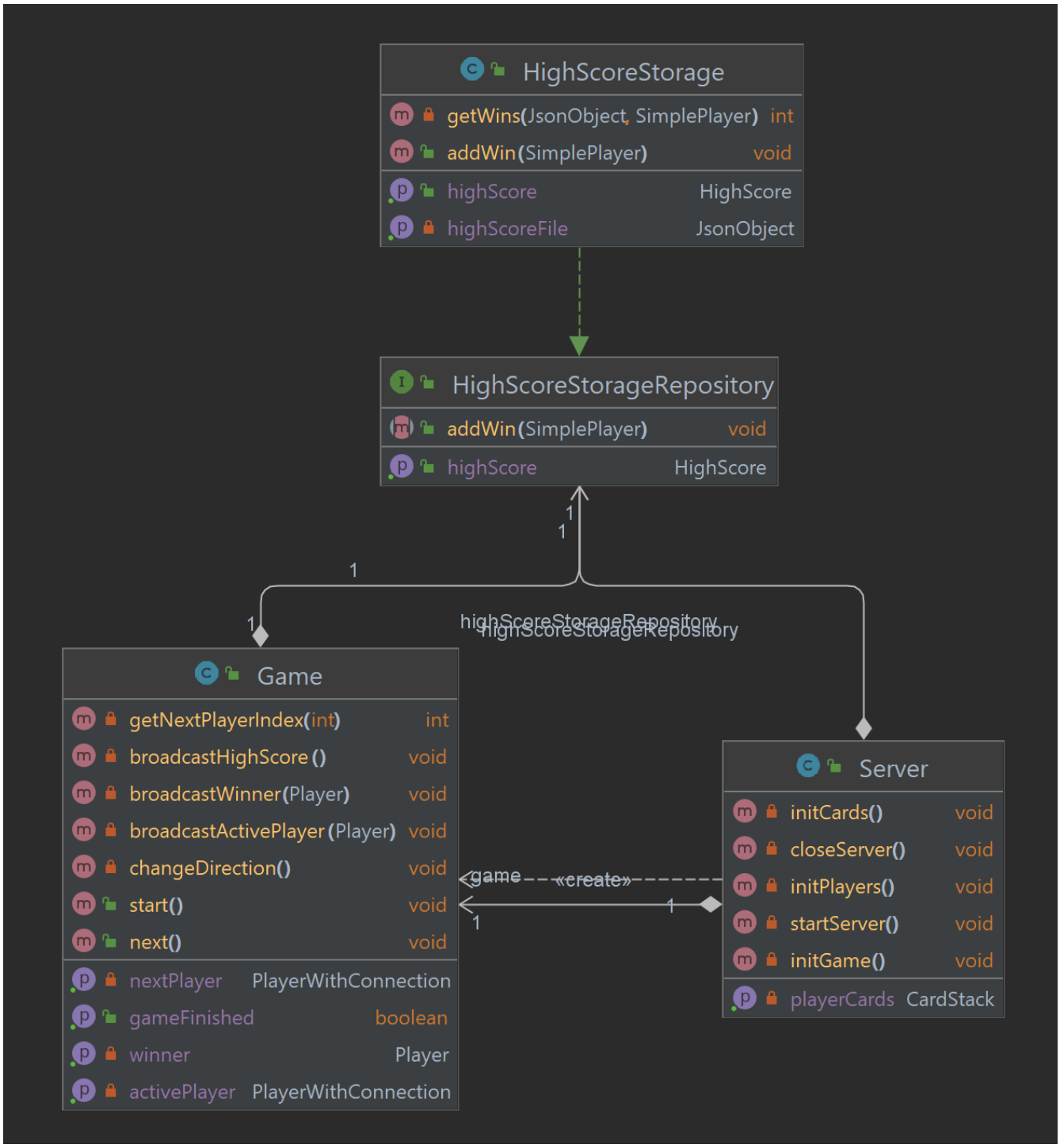
Klasse: `HighScoreStorage` in Verbindung mit `Game` und `Server`
Problem: Die Klassen `Game` und `Server` benutzen die Methoden `addWinToJsonFile()` und `loadHighScoreFromJsonFile()` des `HighScoreStorage`. Falls in Zukunft eine andere Art von Storage eingebaut werden sollte, müssen sowohl die Implementierung von `Game` als auch von `Server` bearbeitet werden.



Lösung:

https://github.com/ase-uno/uno/commit/b1bb3180d9ec01ce749536a02b9c175cb90cb834

Mit dem Interface `HighScoreStorageRepository` wurde eine Abstraktionsschicht eingeführt, sodass `Game` und `Server` standardisierte Schnittstellen nutzen können und nicht verändert werden müssen, wenn die eigentliche Implementierung ausgetauscht wird.

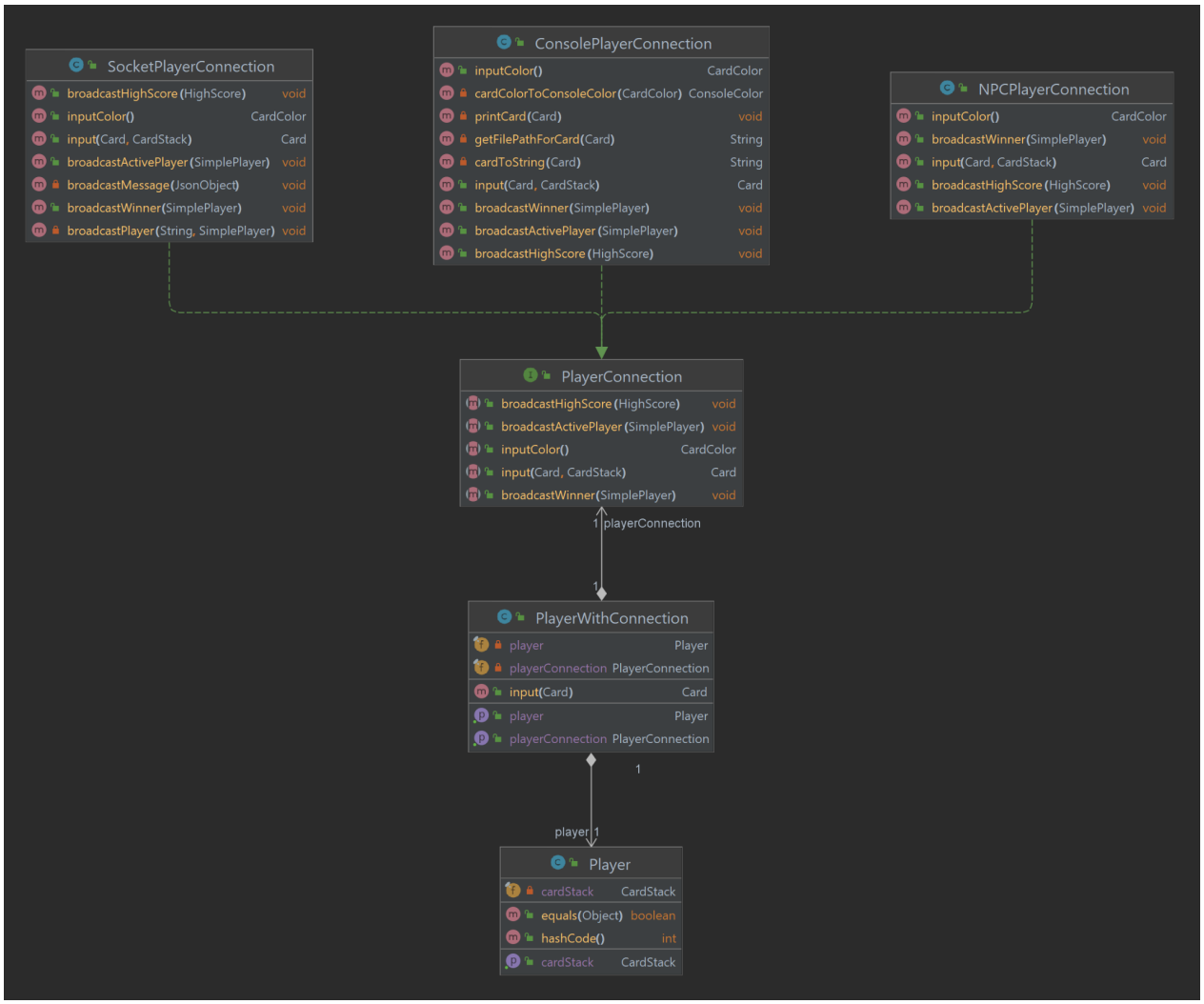


Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

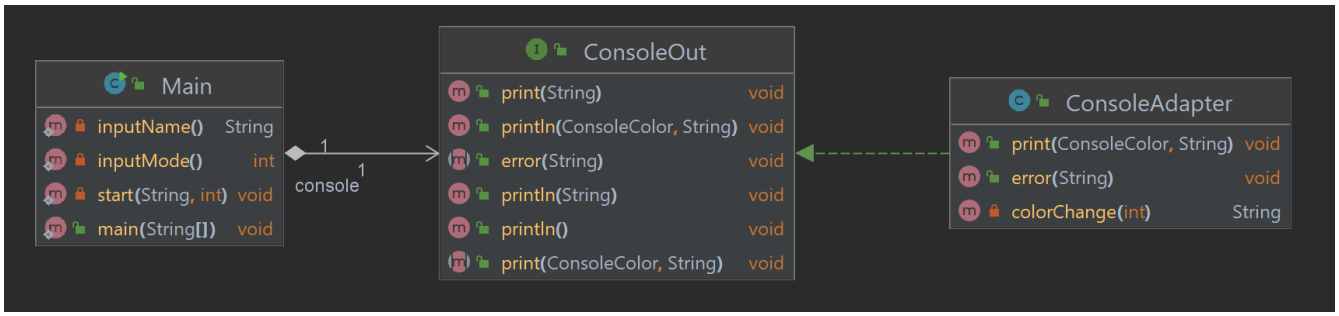
Positiv-Beispiel

Klassen: `SocketPlayerConnection`, `ConsolePlayerConnection` und `NPCPlayerConnection`
Aufgabe: Die Klassen stellen unterschiedliche Möglichkeiten dar, wie ein Spieler am Spiel teilnehmen kann. So kann er etwa über die Konsole spielen (`ConsolePlayerConnection`), extern über eine Netzwerkverbindung (`SocketPlayerConnection`) oder automatisiert als Bot (`NPCPlayerConnection`).
Begründung: Die Kopplung der Klassen ist niedrig, da sie ausschließlich über das Interface `PlayerConnection` mit dem Rest der Anwendung verbunden sind und somit leicht verändert oder ersetzt werden können.



Negativ-Beispiel

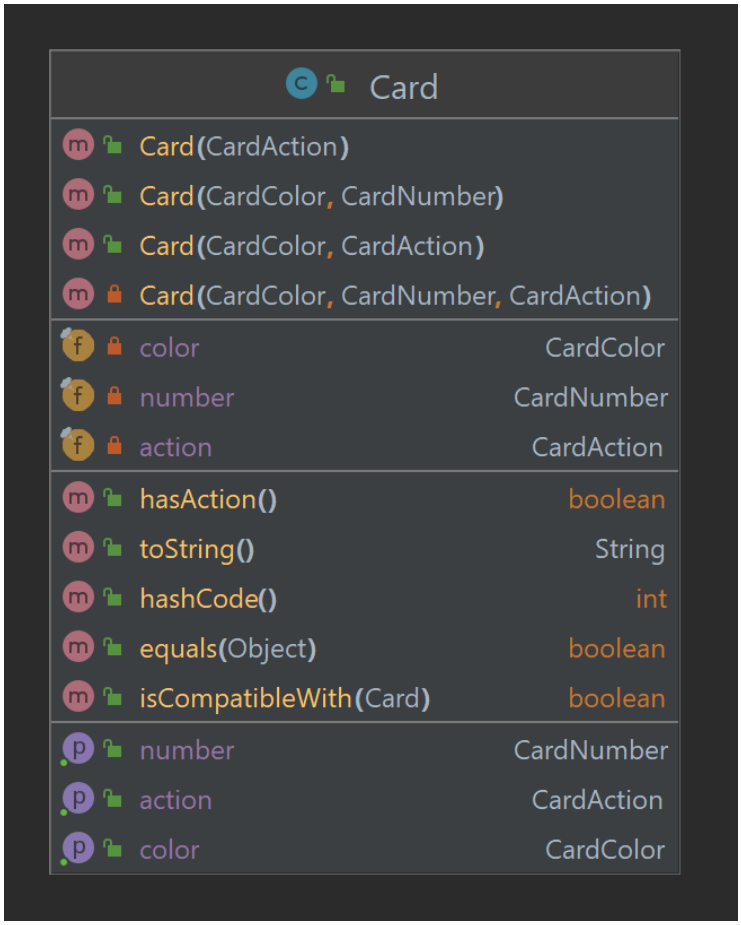
Klasse: `Main`
Aufgabe: Für den Start der Anwendung benötigte Eingaben vom Nutzer entgegennehmen und die Anwendung starten.
Begründung: Um Nutzereingaben entgegenzunehmen, wird hier die Konsole (`System.in`) verwendet (vgl. Quellcode). Um die Kopplung zu verringern, wäre es sinnvoll, ein Interface einzuführen, welches die Konsole sowie mögliche alternative Eingabeformen abstrahiert.




```java
private static String inputName() {
    console.println("Wie heißt du?");

    Scanner scanner = new Scanner(System.in);

    // ...
}

private static int inputMode() {
    Scanner scanner = new Scanner(System.in);

    // ...
}
```

Analyse GRASP: Hohe Kohäsion

Die Kohäsion der Klasse `Card` ist hoch, da alle Attribute und Methoden zur Repräsentation und Realisierung der Spielkarten dienen. Wenn externe Klassen genutzt werden, sind diese zwingend nötig und unverzichtbar.



Don’t Repeat Yourself (DRY)

https://github.com/ase-uno/uno/commit/bfcb4100a258eabe757a15808f41bbc62555873a

Vorher

de.dhbwka.uno.adapters.persistence.HighScoreStorage

```
37
38      @Override
39      public HighScore getHighScore() {
40
41          JsonObject jsonObject = getHighScoreFile();
42          return highScoreFromJson(jsonObject);
43
44      }
45
46      private HighScore highScoreFromJson(JsonElement jsonElement) {
47          if(jsonElement instanceof JsonNull) return new HighScore();
48
49          JsonObject jsonObject = (JsonObject) jsonElement;
50
51          HashMap<SimplePlayer, Integer> data = jsonObject.getElements()
52                  .entrySet()
53                  .stream()
54                  .map(e -> new AbstractMap.SimpleEntry<>(new SimplePlayer(e.getKey()), ((JsonNumber) e.getValue()).getValue().intValue()))
55                  .collect(Collectors.toMap(
56                          AbstractMap.SimpleEntry::getKey,
57                          AbstractMap.SimpleEntry::getValue,
58                          Integer::sum,
59                          HashMap::new
60                  ));
61
62          return new HighScore(data);
63      }
64
```

de.dhbwka.uno.adapters.client.SocketConnection

```
            }
            case "broadcastHighScore" -> {
                JsonObject jsonObject1 = (JsonObject) element;
                HighScore highScore = highScoreFromJson(jsonObject1.get("highScore"));
                playerConnection.broadcastHighScore(highScore);
            }
            default -> console.error("Error, invalid message received from Server");
        }
    }

    private HighScore highScoreFromJson(JsonElement jsonElement) {
        if(jsonElement instanceof JsonNull) return new HighScore();

        JsonObject jsonObject = (JsonObject) jsonElement;

        HashMap<SimplePlayer, Integer> data = jsonObject.getElements()
                .entrySet()
                .stream()
                .map(e -> new AbstractMap.SimpleEntry<>(new SimplePlayer(e.getKey()), ((JsonNumber) e.getValue()).getValue().intValue()))
                .collect(Collectors.toMap(
                        AbstractMap.SimpleEntry::getKey,
                        AbstractMap.SimpleEntry::getValue,
                        Integer::sum,
                        HashMap::new
                ));

        return new HighScore(data);
    }
```

Nachher

de.dhbwka.uno.adapters.mapper.HighScoreMapper

```

    public static HighScore highScoreFromJson(JsonElement jsonElement) {
        if(jsonElement instanceof JsonNull) return new HighScore();

        JsonObject jsonObject = (JsonObject) jsonElement;

        HashMap<SimplePlayer, Integer> data = jsonObject.getElements()
                .entrySet()
                .stream()
                .map(e -> new AbstractMap.SimpleEntry<>(new SimplePlayer(e.getKey()), ((JsonNumber) e.getValue()).getValue().intValue()))
                .collect(Collectors.toMap(
                        AbstractMap.SimpleEntry::getKey,
                        AbstractMap.SimpleEntry::getValue,
                        Integer::sum,
                        HashMap::new
                ));

        return new HighScore(data);
    }
```

de.dhbwka.uno.adapters.persistence.HighScoreStorage

```
33
34      @Override
35      public HighScore getHighScore() {
36
37          JsonObject jsonObject = getHighScoreFile();
38          return HighScoreMapper.highScoreFromJson(jsonObject);
39
40      }
```

de.dhbwka.uno.adapters.client.SocketConnection

```
84              }
85              case "broadcastHighScore" -> {
86                  JsonObject jsonObject1 = (JsonObject) element;
87                  HighScore highScore = HighScoreMapper.highScoreFromJson(jsonObject1.get("highScore"));
88                  playerConnection.broadcastHighScore(highScore);
89              }
90              default -> console.error("Error, invalid message received from Server");
91          }
```

Begründung

Die Methode `highScoreFromJson()` wurde zuvor an zwei unterschiedlichen Stellen exakt identisch implementiert. Falls die Methode in Zukunft verändert werden sollte, ist es sehr wahrscheinlich, dass nur eine Implementierung angepasst und die andere vergessen wird.

Auswirkung

Durch die Änderungen wurde redundanter Code entfernt und das Risiko für potenzielle Inkonsistenzen durch zukünftige Änderungen an der Methode minimiert. Außerdem kann die neue Methode einfach an weiteren Stellen wiederverwendet werden, falls dies in Zukunft nötig sein sollte.

Kapitel 5: Unit-Tests

10 Unit-Tests

Unit-Test	Beschreibung
`CardTest #compatibleSameColor`	Testet, ob zwei Karten als kompatibel (können aufeinander gelegt werden) erkannt werden, wenn sie die gleiche Farbe haben
`CardTest #compatibleSameNumber`	Testet, ob zwei Karten als kompatibel erkannt werden, wenn sie die gleiche Nummer haben
`CardTest #compatibleIncompatible`	Testet, ob zwei Karten als nicht-kompatibel erkannt werden, wenn sie gänzlich unterschiedlich sind
`CardStackTest #add`	Testet, ob Karten korrekt zu einem Kartenstapel hinzugefügt werden können
`CardStackTest #remove`	Testet, ob Karten korrekt von einem Kartenstapel entfernt werden können
`CardStackTest #isFinished`	Testet, ob Kartenstapel korrekt als “fertig” (Aus Spielersicht: Keine Karten mehr übrig → Spieler hat gewonnen) markiert werden
`CardNumberTest #setNumberFailPos`	Testet, ob Kartennummern den Wert 9 nicht überschreiten können
`CardNumberTest #setNumberFailNeg`	Testet, ob Kartennummern den Wert 0 nicht unterschreiten können
`JsonObjectTest #toJson`	Testet, ob `JsonObject`-Objekte zu korrektem JSON konvertiert werden
`JsonArrayTest #toJson`	Testet, ob `JsonArray`-Objekte zu korrektem JSON konvertiert werden

ATRIP: Automatic

Sämtliche Unit-Tests können mithilfe des Befehls `mvn test` ausgeführt werden. Dieser Prozess kann auch einfach, beispielsweise in einer CI/CD-Pipeline, automatisiert werden.

ATRIP: Thorough

Positiv:

```
public CardNumber(int value) {
   setValue(value);
}

public void setValue(int value) {
   if(value < 0 || value > 9) {
       throw new IllegalArgumentException("Numbers only allowed in
                                                    Interval [0, 9]");
   }
   this.value = value;
}
```

```
@Test
public void setNumberSuccess() {
   for(int i = 0; i ⩽ 9; i++) {
       CardNumber number = new CardNumber(i);
       assertEquals(number.getValue(), i);
   }
}


@Test
public void setNumberFailNeg() {
   assertThrows(IllegalArgumentException.class, () → new CardNumber(-1));
}

@Test
public void setNumberFailPos() {
   assertThrows(IllegalArgumentException.class, () → new CardNumber(10));
}
```

Spielkarten können ausschließlich Nummern von 0 bis 9 haben. Wird beim Erzeugen einer `CardNumber` ein ungültiger Wert eingegeben, so löst dies eine `IllegalArgumentException` aus. Der Test setNumberSuccess() überprüft, ob alle gültigen Werte verwendet werden können. Die anderen beiden Tests überprüfen, ob korrekterweise ein Fehler ausgelöst wird, wenn der gültige Wertebereich unter-/überschritten wird. Auf diese Weise wird jeder erdenkliche Fall beim Ausführen der zu testenden Methode überprüft.

Negativ:

```
public Card input(Card active, CardStack cardStack) {
    for(Card card: cardStack.getCardList()) {
        if(active.isCompatibleWith(card)) {
            return card;
        }
    }

    return null;
}
```

```
@Test
public void input() {
    Card activeCard = new Card(CardColor.BLUE, new CardNumber(1));

    List<Card> cards = new ArrayList<>();
    cards.add(new Card(CardColor.RED, new CardNumber(2)));

    CardStack cardStack = new CardStack(cards);

    NPCPlayerConnection con = new NPCPlayerConnection("Bob der Baumeister");

    assertNull(con.input(activeCard, cardStack));
}
```

Die Methode `input()` gibt die erste Karte eines Kartenstapels (`cardStack`) zurück, die mit einer gegebenen Karte (`active`) kompatibel ist. Wird keine kompatible Karte gefunden, so wird `null` zurückgegeben. Der dargestellte Test überprüft lediglich den Fall, dass keine Karte gefunden wurde. Die rot markierte Zeile wird somit nie ausgeführt und die Methode nie vollständig getestet. Es wird jedoch kaum Aufwand benötigt, um diesen Fall ebenfalls abzudecken.

ATRIP: Professional

Positiv:

```
class GameTest {

   @Test
   void gameFinishesWhenOnePlayerHasNoCards() {
       Game game = new TestGameBuilder()
               .oneCard()
               .build();
       game.start();
       assertTrue(game.isGameFinished());
   }

   // ...
}
```

```
public class TestGameBuilder {

   // ...

   public TestGameBuilder() {
       // ...
   }

   public TestGameBuilder oneCard() {
       // ...
   }

   // ...

   public Game build() {
       // ...
   }
}
```

```java
public class MockedPlayerConnection implements PlayerConnection {

    // ...


    @Override
    public Card input(Card active, CardStack cardStack) {
        inputCalled++;

        // ...
    }

    // ...

    public int getInputCalled() {
        return inputCalled;
    }

    // ...
}
```

```java
public class TestCardProvider implements CardProvider {

    // ...

    @Override
    public List<Card> listAllCards() {
        List<Card> list = new ArrayList<>();

        for(int i = 0; i<amountCards; i++) {
            list.add(cardToProvide);
        }

        return list;
    }
}
```

Die Tests der Hauptlogik in der Klasse “Game”, wird mit einigen Hilfsklassen zur Erzeugung von Elementen und von Testimplementierungen unterstützt. So bietet beispielsweise die Klasse “TestGameBuilder” einen Builder an, mit dem unterschiedliche Szenarien für Spiele erzeugt werden können, ohne jedes Mal den Code dafür neu zu schreiben. So ist es extrem einfach, ein Spiel mit einer vorgegebenen Kartenzahl zu starten.
Eine weitere Hilfsklasse des Spieles ist die “MockedPlayerConnection”. Hier wird die Funktionalität, die eigentlich durch Benutzerinput und die Konsole durchgeführt wird, gemockt. Das bedeutet, eine Standardfunktionalität ist eingebunden und die Anzahl der Aufrufe wird mitgezählt, um in den Tests eine Auswertung durchzuführen, ob der Benutzerinput korrekt angefordert wurde. Durch diese Klasse sind die Eingaben jedes Mal identisch und voraussehbar, somit ist das Spiel einfacher testbar.
Auch zur einfacheren Testbarkeit trägt die Klasse “TestCardProvider” bei. Es wird bei Tests kein vollständiger Kartenstapel bereitgestellt, sondern nur die Karten, die für den Test benötigt werden, damit entfällt der zufällige Charakter des Spieles und die Karten und deren Generator sind deterministisch.

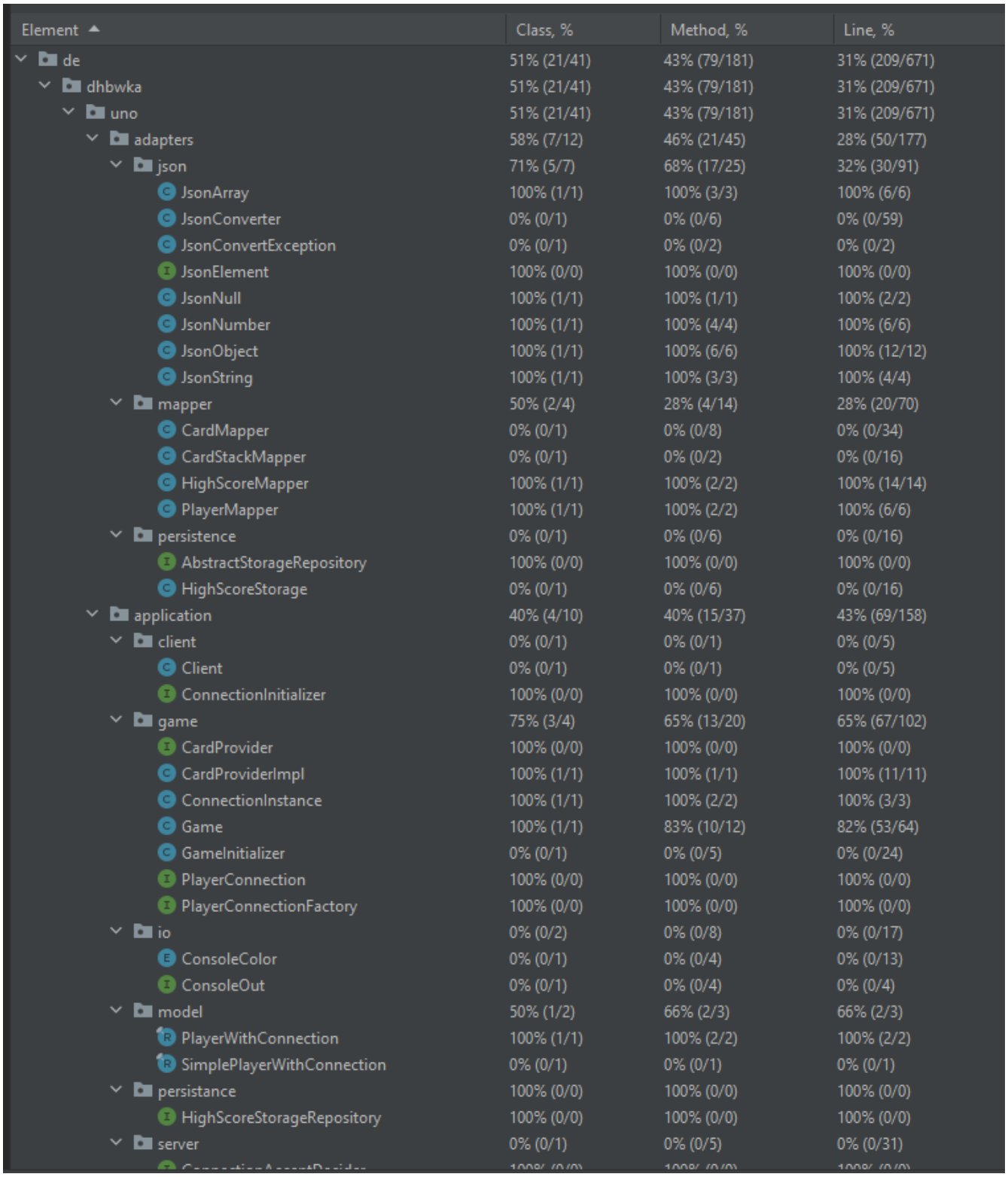
Negativ:

```java
public class ConnectionInstance {
   private final String name;

   public ConnectionInstance(String name) {
       this.name = name;
   }

   public String getLocalName() {
       return name;
   }
}
```

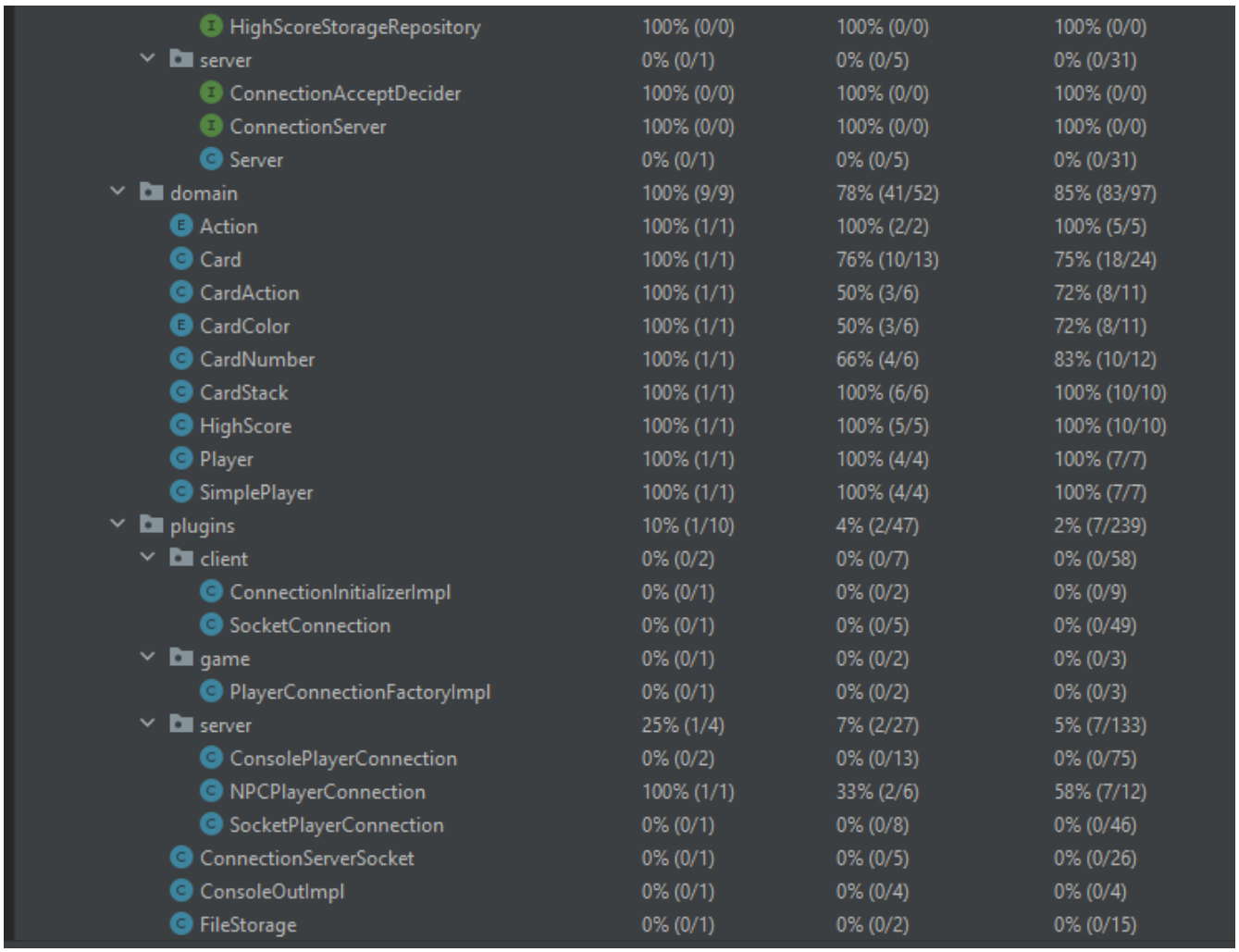
```java
@Test
void getLocalName() {
   final String name = "Peter Lustig";

   ConnectionInstance con = new ConnectionInstance(name);
   assertEquals(con.getLocalName(), name);
}
```

Der gezeigte Test testet lediglich eine Getter-Methode. Dies ist unprofessionell, da in solchen Methoden nur triviale Logik vorhanden ist. Der Test ist somit unnötig.

Code Coverage



Element	Class, %	Method, %	Line, %
de	51% (21/41)	43% (79/181)	31% (209/671)
dhbwka	51% (21/41)	43% (79/181)	31% (209/671)
uno	51% (21/41)	43% (79/181)	31% (209/671)
adapters	58% (7/12)	46% (21/45)	28% (50/177)
json	71% (5/7)	68% (17/25)	32% (30/91)
JsonArray	100% (1/1)	100% (3/3)	100% (6/6)
JsonConverter	0% (0/1)	0% (0/6)	0% (0/59)
JsonConvertException	0% (0/1)	0% (0/2)	0% (0/2)
JsonElement	100% (0/0)	100% (0/0)	100% (0/0)
JsonNull	100% (1/1)	100% (1/1)	100% (2/2)
JsonNumber	100% (1/1)	100% (4/4)	100% (6/6)
JsonObject	100% (1/1)	100% (6/6)	100% (12/12)
JsonString	100% (1/1)	100% (3/3)	100% (4/4)
mapper	50% (2/4)	28% (4/14)	28% (20/70)
CardMapper	0% (0/1)	0% (0/8)	0% (0/34)
CardStackMapper	0% (0/1)	0% (0/2)	0% (0/16)
HighScoreMapper	100% (1/1)	100% (2/2)	100% (14/14)
PlayerMapper	100% (1/1)	100% (2/2)	100% (6/6)
persistence	0% (0/1)	0% (0/6)	0% (0/16)
AbstractStorageRepository	100% (0/0)	100% (0/0)	100% (0/0)
HighScoreStorage	0% (0/1)	0% (0/6)	0% (0/16)
application	40% (4/10)	40% (15/37)	43% (69/158)
client	0% (0/1)	0% (0/1)	0% (0/5)
Client	0% (0/1)	0% (0/1)	0% (0/5)
ConnectionInitializer	100% (0/0)	100% (0/0)	100% (0/0)
game	75% (3/4)	65% (13/20)	65% (67/102)
CardProvider	100% (0/0)	100% (0/0)	100% (0/0)
CardProviderImpl	100% (1/1)	100% (1/1)	100% (11/11)
ConnectionInstance	100% (1/1)	100% (2/2)	100% (3/3)
Game	100% (1/1)	83% (10/12)	82% (53/64)
GameInitializer	0% (0/1)	0% (0/5)	0% (0/24)
PlayerConnection	100% (0/0)	100% (0/0)	100% (0/0)
PlayerConnectionFactory	100% (0/0)	100% (0/0)	100% (0/0)
io	0% (0/2)	0% (0/8)	0% (0/17)
ConsoleColor	0% (0/1)	0% (0/4)	0% (0/13)
ConsoleOut	0% (0/1)	0% (0/4)	0% (0/4)
model	50% (1/2)	66% (2/3)	66% (2/3)
PlayerWithConnection	100% (1/1)	100% (2/2)	100% (2/2)
SimplePlayerWithConnection	0% (0/1)	0% (0/1)	0% (0/1)
persistance	100% (0/0)	100% (0/0)	100% (0/0)
HighScoreStorageRepository	100% (0/0)	100% (0/0)	100% (0/0)
server	0% (0/1)	0% (0/5)	0% (0/31)

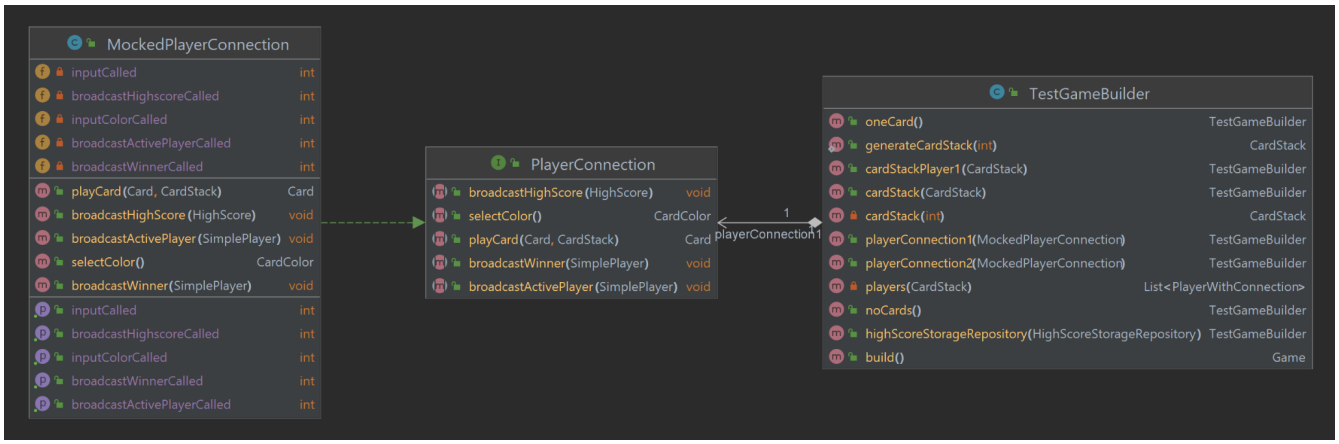


HighScoreStorageRepository	100% (0/0)	100% (0/0)	100% (0/0)
server	0% (0/1)	0% (0/5)	0% (0/31)
ConnectionAcceptDecider	100% (0/0)	100% (0/0)	100% (0/0)
ConnectionServer	100% (0/0)	100% (0/0)	100% (0/0)
Server	0% (0/1)	0% (0/5)	0% (0/31)
domain	100% (9/9)	78% (41/52)	85% (83/97)
Action	100% (1/1)	100% (2/2)	100% (5/5)
Card	100% (1/1)	76% (10/13)	75% (18/24)
CardAction	100% (1/1)	50% (3/6)	72% (8/11)
CardColor	100% (1/1)	50% (3/6)	72% (8/11)
CardNumber	100% (1/1)	66% (4/6)	83% (10/12)
CardStack	100% (1/1)	100% (6/6)	100% (10/10)
HighScore	100% (1/1)	100% (5/5)	100% (10/10)
Player	100% (1/1)	100% (4/4)	100% (7/7)
SimplePlayer	100% (1/1)	100% (4/4)	100% (7/7)
plugins	10% (1/10)	4% (2/47)	2% (7/239)
client	0% (0/2)	0% (0/7)	0% (0/58)
ConnectionInitializerImpl	0% (0/1)	0% (0/2)	0% (0/9)
SocketConnection	0% (0/1)	0% (0/5)	0% (0/49)
game	0% (0/1)	0% (0/2)	0% (0/3)
PlayerConnectionFactoryImpl	0% (0/1)	0% (0/2)	0% (0/3)
server	25% (1/4)	7% (2/27)	5% (7/133)
ConsolePlayerConnection	0% (0/2)	0% (0/13)	0% (0/75)
NPCPlayerConnection	100% (1/1)	33% (2/6)	58% (7/12)
SocketPlayerConnection	0% (0/1)	0% (0/8)	0% (0/46)
ConnectionServerSocket	0% (0/1)	0% (0/5)	0% (0/26)
ConsoleOutImpl	0% (0/1)	0% (0/4)	0% (0/4)
FileStorage	0% (0/1)	0% (0/2)	0% (0/15)

Viele Tests sowie eine hohe Code Coverage sind immer wünschenswert und bringen viele Vorteile mit sich. Die über 40 implementierten Tests, welche ungefähr ein Drittel der vorhandenen Zeilen abdecken, sind jedoch völlig ausreichend, um die in der Vorlesung behandelten Konzepte anzuwenden. Aus diesem Grund soll es bei diesem Stand bleiben.

Fakes und Mocks

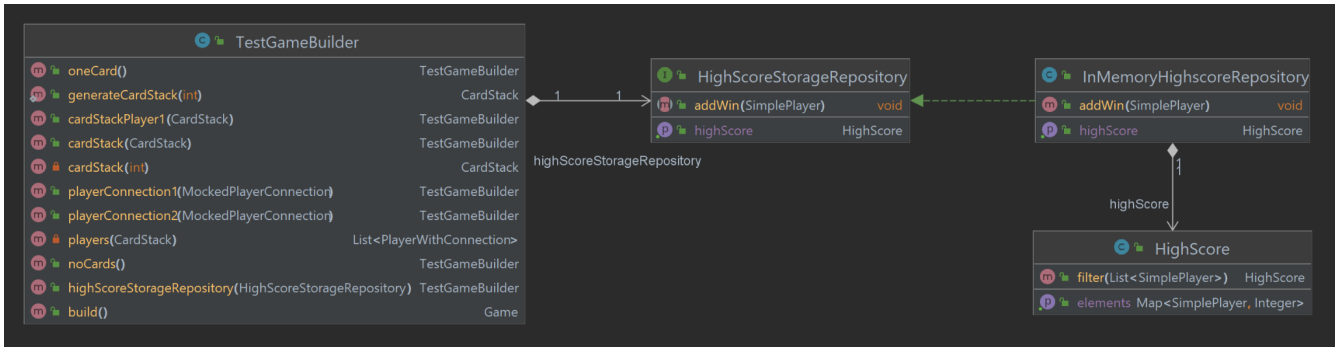
1) `MockedPlayerConnection`



Beim Testen des Spiels wäre es unpraktisch, wenn sich reale Personen als Mitspieler verbinden müssten. Aus diesem Grund wurde die Klasse `MockedPlayerConnection` eingeführt, welche bestimmte Prozesse automatisiert durchführen kann und weitere hilfreiche Funktionen zum Testen anbietet. Dazu zählen beispielsweise:

- Zählvariablen, wie häufig eine bestimmte Methode aufgerufen wurde
- Eine Methode, um Karten auszuspielen
- Eine Methode, um eine Kartenfarbe für den weiteren Spielverlauf auszuwählen

2) `InMemoryHighScoreRepository`



Das Abspeichern von High Scores findet normalerweise in Datenbanken oder einem Dateisystem statt. Damit die Anwendung auch ohne diese Komponenten getestet werden kann, wurde die Klasse `InMemoryHighScoreRepository` eingeführt. Diese übernimmt die gleiche Funktionalität, speichert die Daten jedoch nur im Arbeitsspeicher ab.

Kapitel 6: Domain Driven Design

Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
Card	Spielkarte	Die Spielkarten sind ein elementarer Bestandteil von UNO und kommen somit auch häufig in der Domäne sowie dem Quellcode vor
Player	Mitspieler	Mitspieler sind ebenfalls ein wichtiger Bestandteil jedes Kartenspiels
Action	Aktion einer Spielkarte	Spielkarten können bei UNO bestimmte Aktionen haben (Farbe wechseln, Karten ziehen, Richtung umkehren, Spieler überspringen)
CardStack	Spielkartenstapel	Spielkarten kommen bei UNO meist im Verbund zum Einsatz (Nachziehstapel, Karten eines Mitspielers, bereits gelegte Karten)

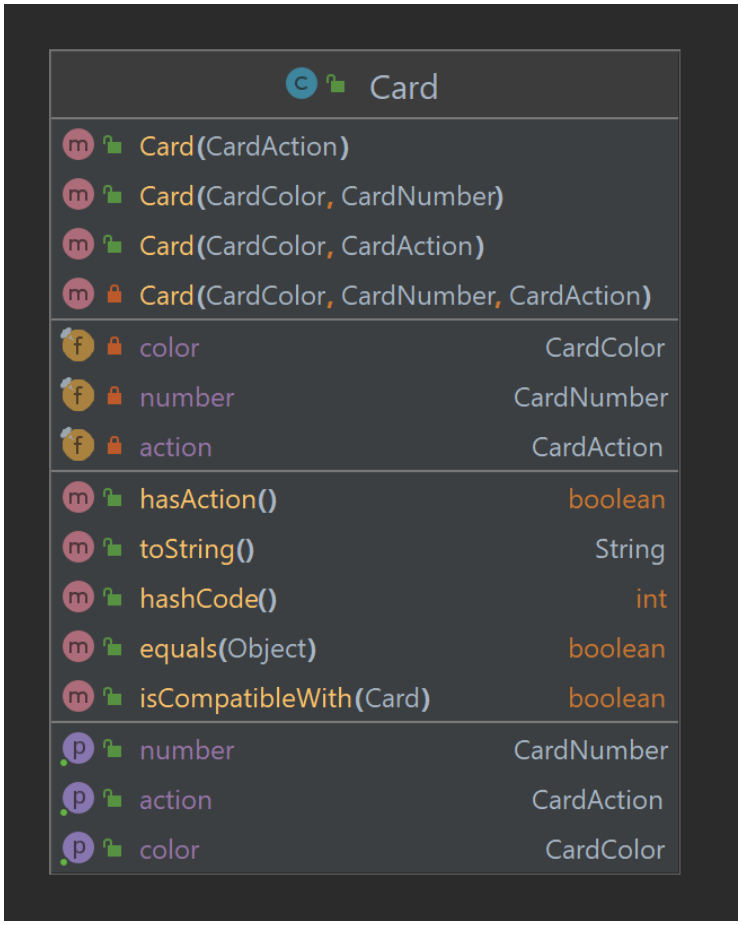
Entities

In der Problemdomäne eines UNO Spieles gibt es zwar Objekte, die eindeutig identifiziert werden, die beispielsweise Spieler durch deren Namen, doch diese Objekte haben keine veränderbare Eigenschaft. Eindeutig identifizierbare Objekte haben keinen Lebenszyklus in der Domain.

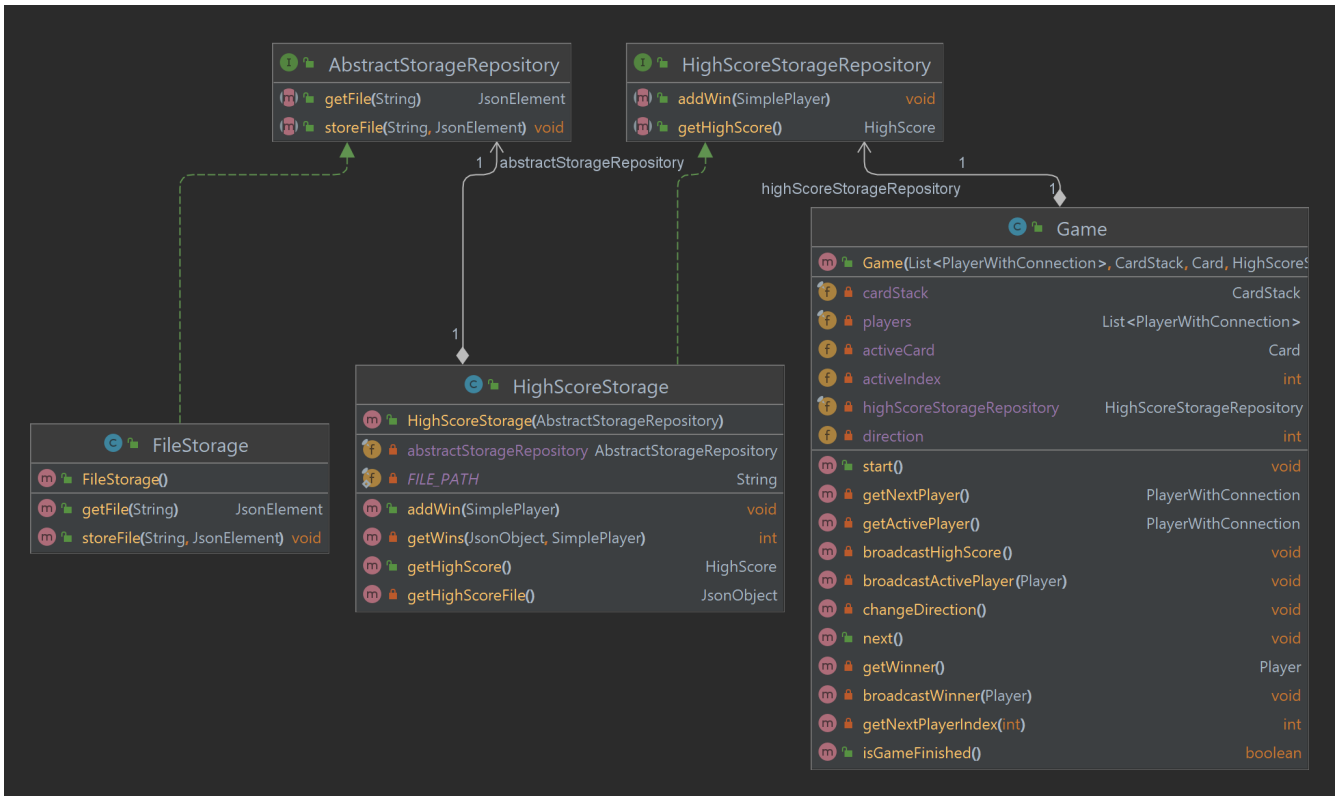
Value Objects

Karten werden einmalig zu Beginn jeder Spielrunde erstellt und erfahren danach keine Änderungen mehr. Lediglich der Kartenstapel, der auf die Karte verweist, kann sich ändern. Dies hat jedoch keinen Einfluss auf die Karte selbst.

.



Repositories



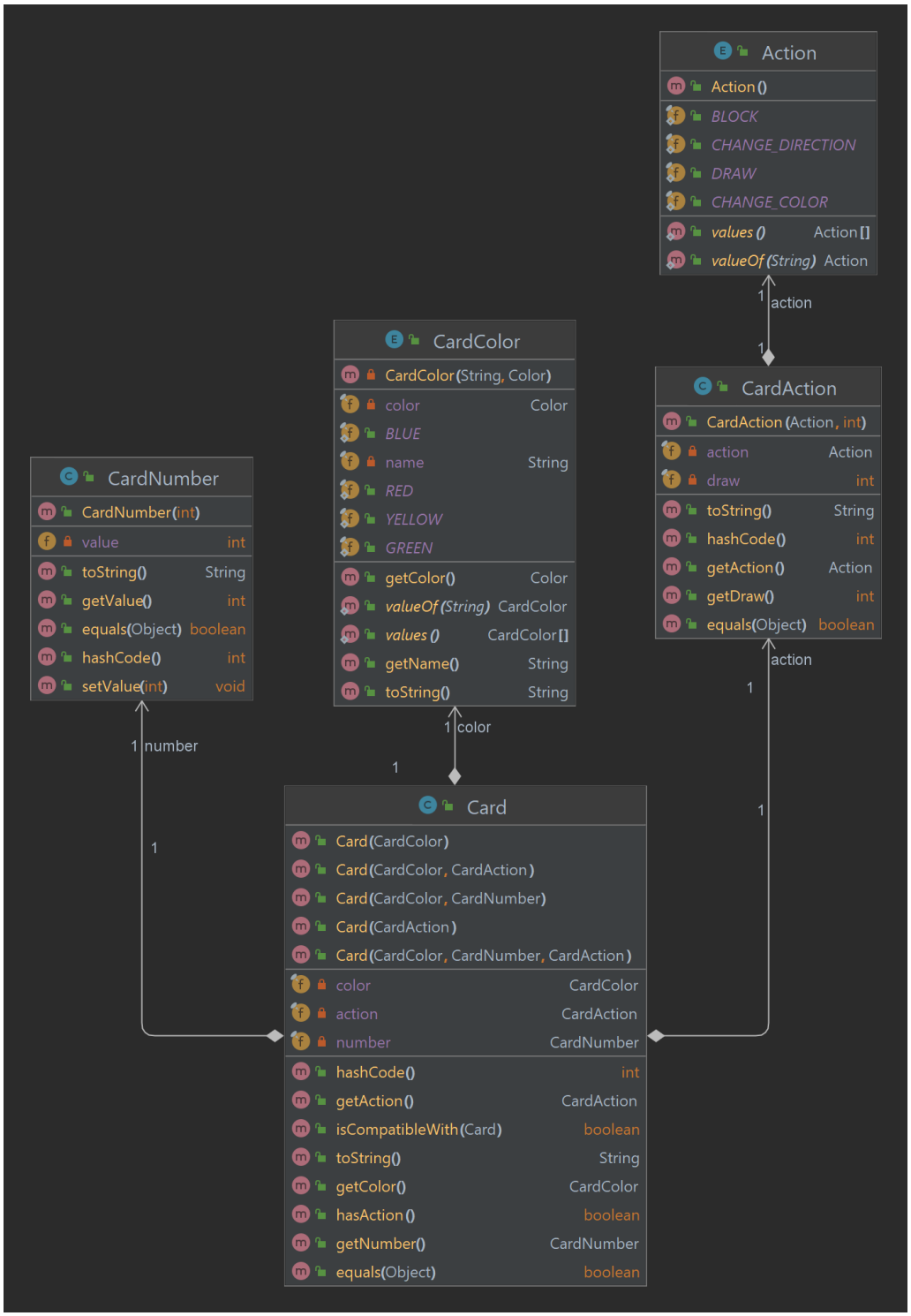
Beschreibung:

Um Highscores über Sitzungen hinweg zu speichern, werden diese Statistiken mit einem Repository persistiert. Hierzu wurden zwei Ebenen gewählt, in der ersten Ebene wird das `HighScoreStorageRepository` (implementiert durch `HighScoreStorage`) benutzt, um in einem fachlichen Ablauf die Statistiken zu ändern oder abzurufen. In der zweiten Ebene werden durch das `AbstractStorageRepository` (implementiert durch `FileStorage`) Aufrufe zu speichern und laden von Dateien abstrahiert.

Begründung:

Der Einsatz eines oder mehrerer Repositories ist hier geeignet, da Daten von der Festplatte des Programms im Server-Modus gelesen und geschrieben werden müssen. Falls in Zukunft die Speicherung der Highscores nicht mehr in einer lokalen Datei des Servers angedacht ist, ist eine Änderung der Technologie durch das Austauschen der Implementierung ohne Anpassung des Applikation-Codes relativ einfach möglich.

Aggregates



Das Objekt `Card` ist ein Aggregat aus einer `CardNumber`, `CardColor` und `CardAction`. Da das Verwalten der einzelnen Aspekte kompliziert wäre, wurde es zusammengefasst.

Kapitel 7: Refactoring

Code Smells

1) Duplikation

<u>Vorher</u>

de.dhbwka.uno.adapters.persistence.HighScoreStorage

```
37
38      @Override
39      public HighScore getHighScore() {
40
41          JsonObject jsonObject = getHighScoreFile();
42          return highScoreFromJson(jsonObject);
43
44      }
45
46      private HighScore highScoreFromJson(JsonElement jsonElement) {
47          if(jsonElement instanceof JsonNull) return new HighScore();
48
49          JsonObject jsonObject = (JsonObject) jsonElement;
50
51          HashMap<SimplePlayer, Integer> data = jsonObject.getElements()
52                  .entrySet()
53                  .stream()
54                  .map(e -> new AbstractMap.SimpleEntry<>(new SimplePlayer(e.getKey()), ((JsonNumber) e.getValue()).getValue().intValue()))
55                  .collect(Collectors.toMap(
56                          AbstractMap.SimpleEntry::getKey,
57                          AbstractMap.SimpleEntry::getValue,
58                          Integer::sum,
59                          HashMap::new
60                  ));
61
62          return new HighScore(data);
63      }
64
```

de.dhbwka.uno.adapters.client.SocketConnection

```
            }
            case "broadcastHighScore" -> {
                JsonObject jsonObject1 = (JsonObject) element;
                HighScore highScore = highScoreFromJson(jsonObject1.get("highScore"));
                playerConnection.broadcastHighScore(highScore);
            }
            default -> console.error("Error, invalid message received from Server");
        }
    }

    private HighScore highScoreFromJson(JsonElement jsonElement) {
        if(jsonElement instanceof JsonNull) return new HighScore();

        JsonObject jsonObject = (JsonObject) jsonElement;

        HashMap<SimplePlayer, Integer> data = jsonObject.getElements()
                .entrySet()
                .stream()
                .map(e -> new AbstractMap.SimpleEntry<>(new SimplePlayer(e.getKey()), ((JsonNumber) e.getValue()).getValue().intValue()))
                .collect(Collectors.toMap(
                        AbstractMap.SimpleEntry::getKey,
                        AbstractMap.SimpleEntry::getValue,
                        Integer::sum,
                        HashMap::new
                ));

        return new HighScore(data);
    }
```

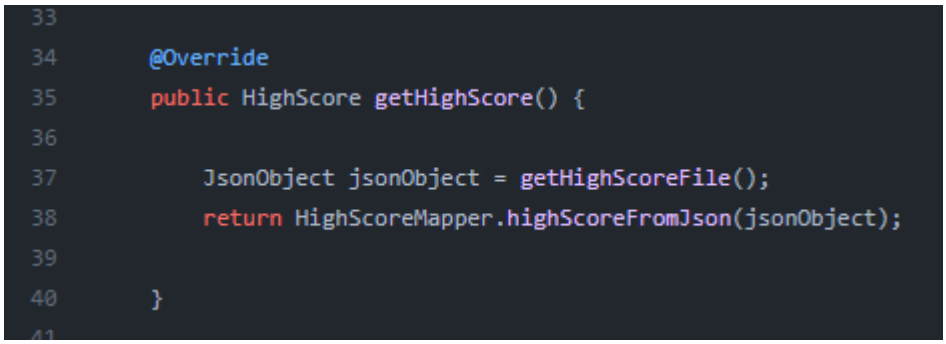
Lösung

Methode an zentrale Stelle auslagern

de.dhbwka.uno.adapters.mapper.HighScoreMapper

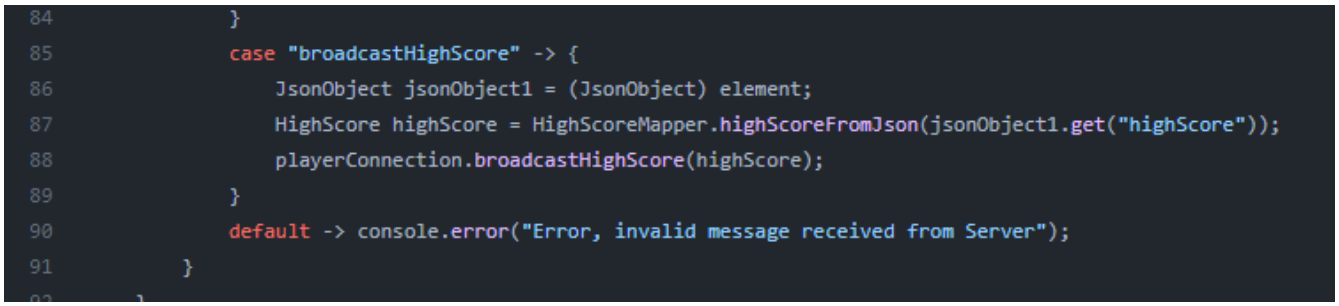
```
    public static HighScore highScoreFromJson(JsonElement jsonElement) {
        if(jsonElement instanceof JsonNull) return new HighScore();

        JsonObject jsonObject = (JsonObject) jsonElement;

        HashMap<SimplePlayer, Integer> data = jsonObject.getElements()
                .entrySet()
                .stream()
                .map(e -> new AbstractMap.SimpleEntry<>(new SimplePlayer(e.getKey()), ((JsonNumber) e.getValue()).getValue().intValue()))
                .collect(Collectors.toMap(
                        AbstractMap.SimpleEntry::getKey,
                        AbstractMap.SimpleEntry::getValue,
                        Integer::sum,
                        HashMap::new
                ));

        return new HighScore(data);
    }
```

de.dhbwka.uno.adapters.persistence.HighScoreStorage



```
33
34      @Override
35      public HighScore getHighScore() {
36
37          JsonObject jsonObject = getHighScoreFile();
38          return HighScoreMapper.highScoreFromJson(jsonObject);
39
40      }
```

de.dhbwka.uno.adapters.client.SocketConnection



```
84              }
85              case "broadcastHighScore" -> {
86                  JsonObject jsonObject1 = (JsonObject) element;
87                  HighScore highScore = HighScoreMapper.highScoreFromJson(jsonObject1.get("highScore"));
88                  playerConnection.broadcastHighScore(highScore);
89              }
90              default -> console.error("Error, invalid message received from Server");
91          }
```

2) Shotgun Surgery

https://github.com/ase-uno/uno/commit/fe75f6c1ccee961165b58363df2874d28f66595c

de.dhbwka.uno.adapters.client.SocketConnection

```
private void connect(String ip, String name) throws IOException {
   socket = new Socket(ip, 9999);

   // ...
}
```

de.dhbwka.uno.application.server.Server

```
private void startServer() throws IOException {
   serverSocket = new ServerSocket(9999);

   // ...
}
```

Problem: Der Port zur Verbindung von Client und Server wird an zwei völlig unterschiedlichen Stellen im Code verwendet. Sollte es zu einer Änderung kommen, ist es nicht unwahrscheinlich, dass eine der beiden Stellen vergessen wird.

Lösung: Port als statische Variable in eine Config-Datei auslagern.

de.dhbwka.uno.adapters.client.SocketConnection

```
private void connect(String ip, String name) throws IOException {
   socket = new Socket(ip, ConnectionConfig.SOCKET_PORT);

   // ...
}
```

de.dhbwka.uno.application.server.Server

```
private void startServer() throws IOException {
   serverSocket = new ServerSocket(ConnectionConfig.SOCKET_PORT);

   // ...
}
```

de.dhbwka.uno.application.config.ConnectionConfig

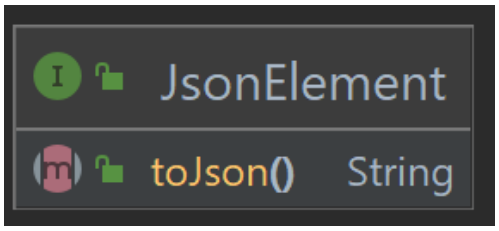
```
public class ConnectionConfig {

   public static final int SOCKET_PORT = 9999;

}
```

2 Refactorings

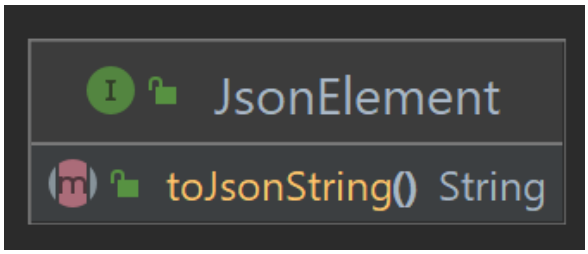
1) Rename Method

JsonElement:

Vorher:

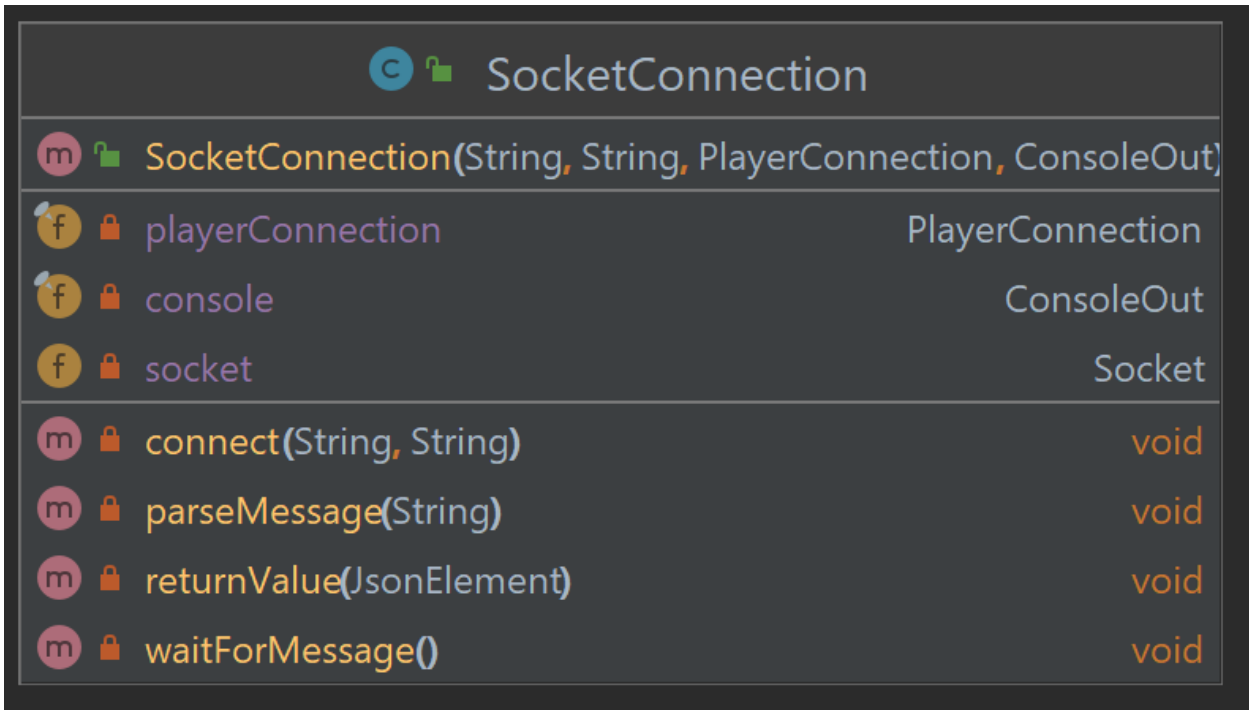


Nachher:

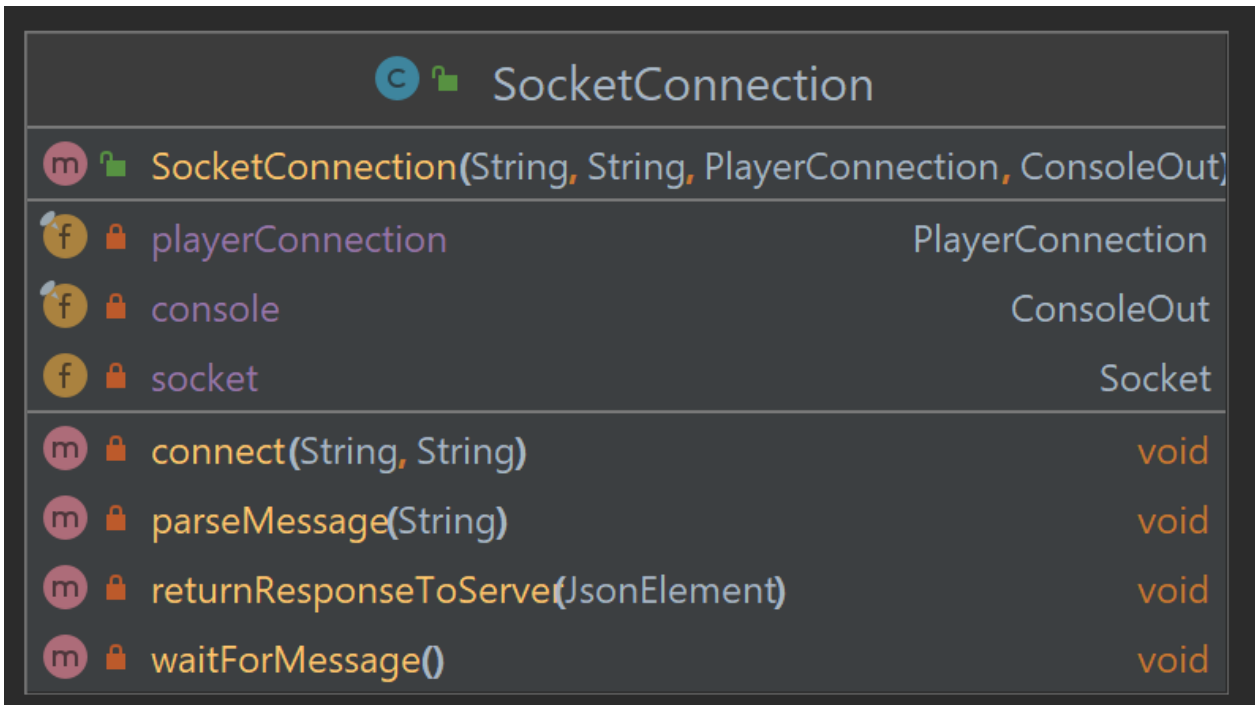


SocketConnection:

Vorher:



Nachher:

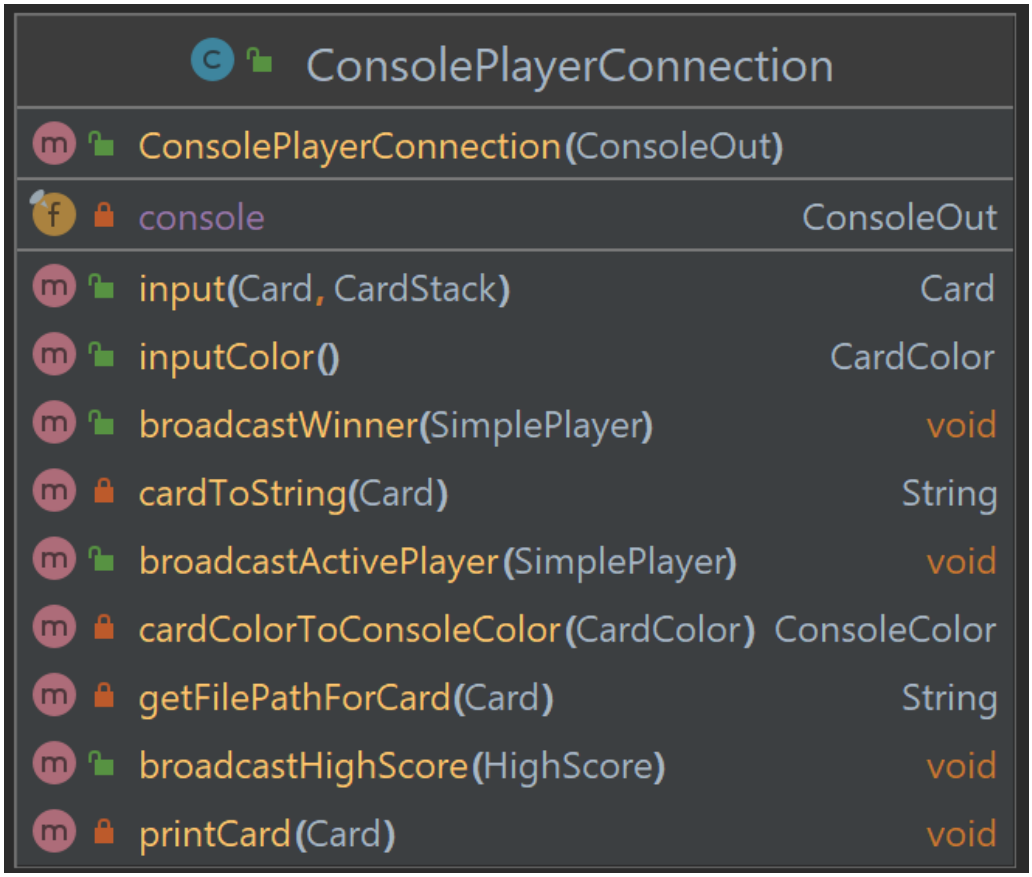


Commit:
[refactoring 1 - rename methods · ase-uno/uno@321cae0 (github.com)]

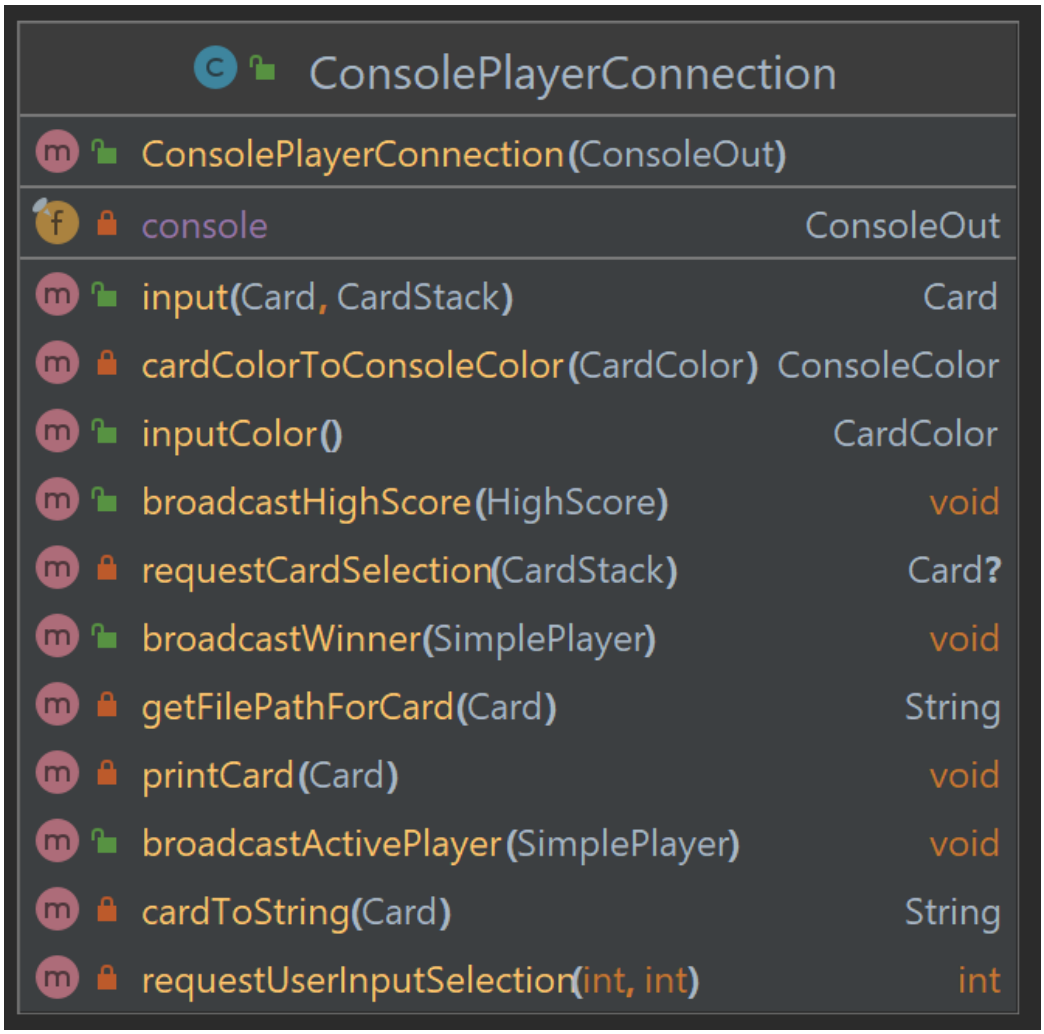
Begründung:
“`toJson`” (und auch “`fromJson`”) ist nicht eindeutig, da als JSON im Allgemeinen im Code die JSON-Elemente und nicht die String-Repräsentation von JSON-Werten gemeint ist. Damit musste diese Methode umbenannt werden
“`returnValue`” ist nicht sprechend und kann zu Verwirrungen führen, es ist nicht klar ersichtlich, dass eine Antwort mit den Werten an den Server zurückgeschickt wird. Hier wurde der neue Name “`returnResponseToServer`” gewählt, um die Aufgabe der Methode eindeutig zu beschreiben

2) Extract Method

Vorher:



Nachher:



Commit:

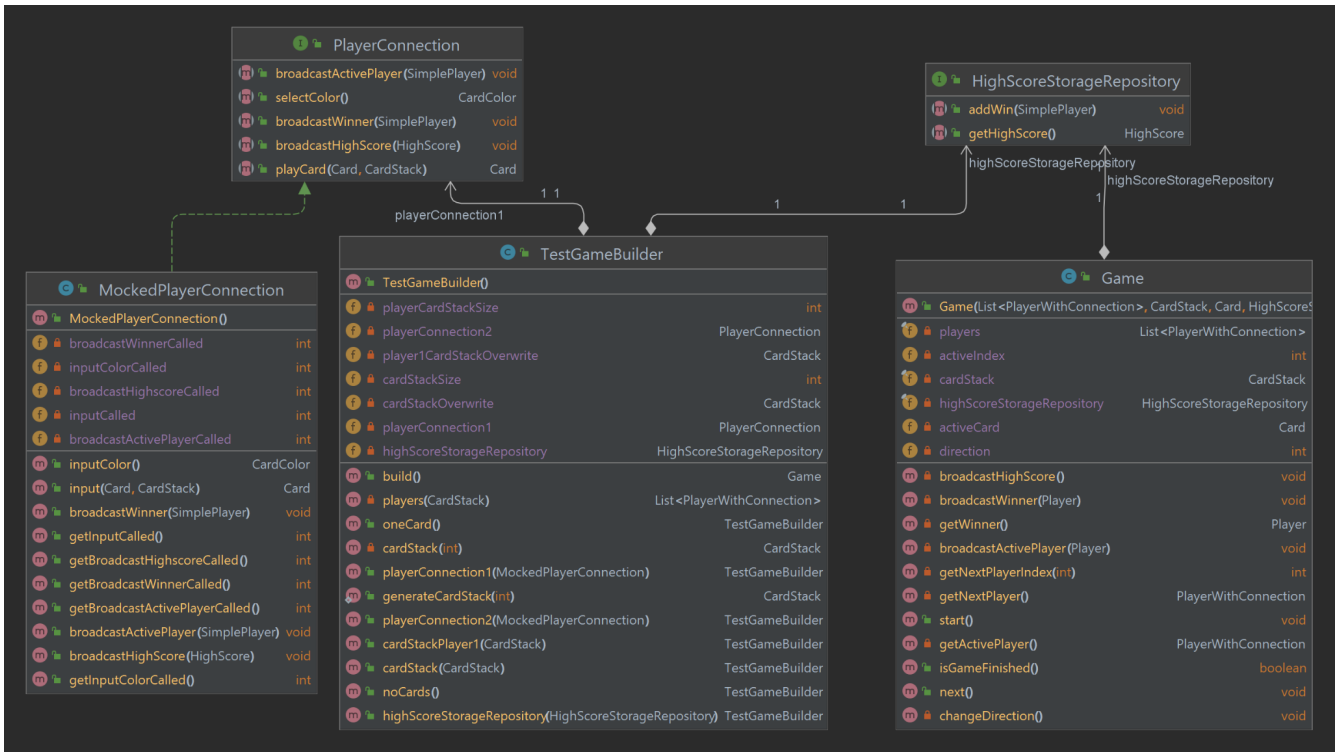
refactoring 2 - extract method · ase-uno/uno@407f24d (github.com)

Begründung:

Der User Input wurde von zwei Methoden unabhängig implementiert, “`input`” und “`inputColor`”, jetzt wird die gewählte Karte von “`input`” von “`requestCardSelection`” geliefert, welche die zentral benutzte Methode “`requestUserInputSelection`” aufruft, welche die Auswahl der Karte vom Benutzer übernimmt. Diese Methode wird auch von “`inputColor`” verwendet, somit gibt es keine doppelte Implementierung mehr und Code wurde in eine extra Methode extrahiert.

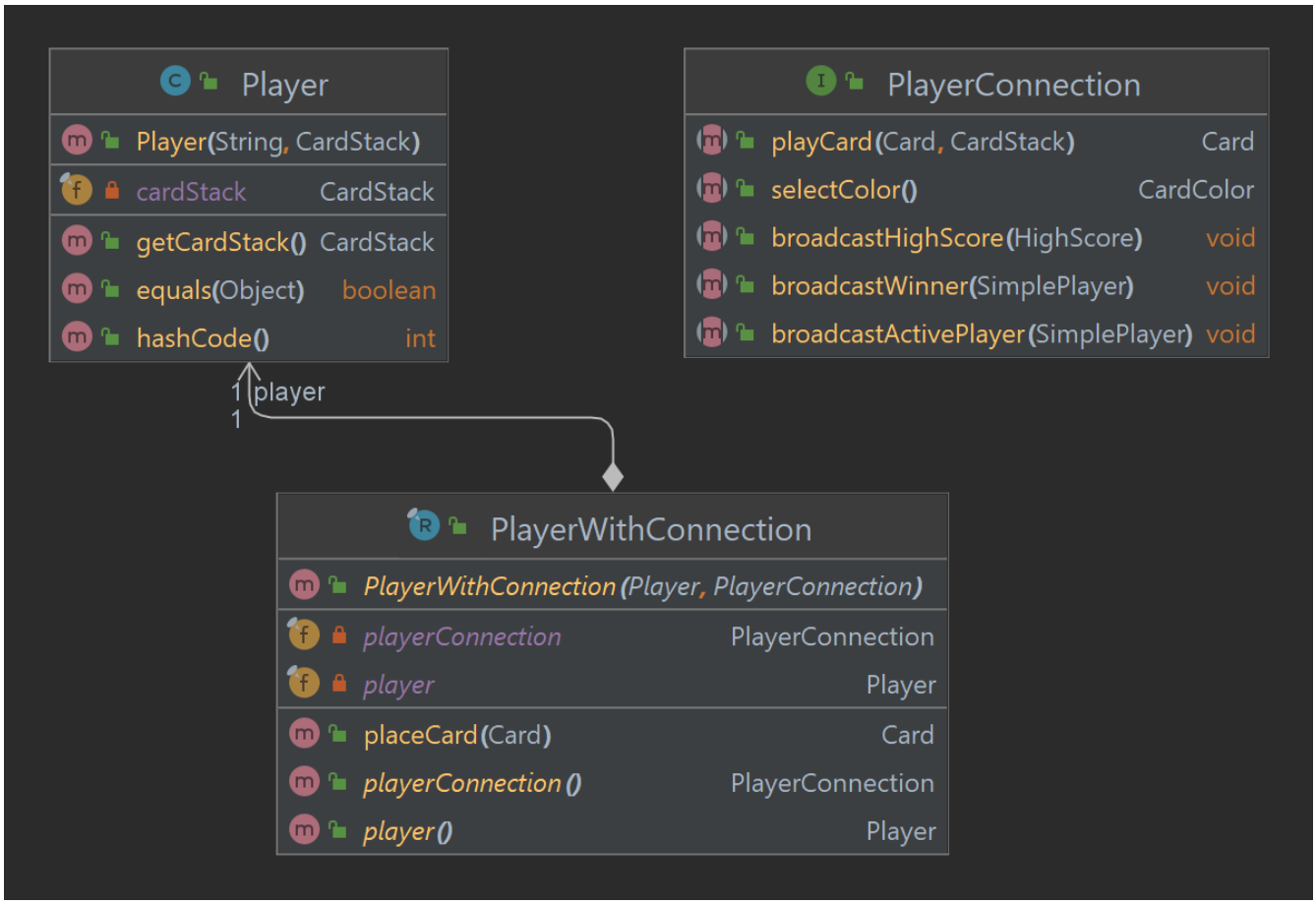
Kapitel 8: Entwurfsmuster

Entwurfsmuster: Erbauer / Builder



Der `TestGameBuilder` ist ein Erbauer für die im Test benutzten Instanzen für Spiele. Dass nicht der Konstruktor von `Game` benutzt werden muss, für den weitere Abhängigkeiten bestehen. So ist das Erbauen von Testspiel Instanzen erleichtert. Der `TestGameBuilder` ist auch sehr kurzlebig, da er nach dem Aufruf der `.build()` Methode nicht mehr benötigt wird.

Entwurfsmuster: Dekorierer / Wrapper



Die Klasse `PlayerWithConnection` dekoriert eine Spieler-Instanz mit der dazugehörigen Verbindung (`PlayerConnection`). Dies erweitert das Domainobjekt Spieler um die Funktion, mit Transferschichten zu interagieren.