

# Programmentwurf Minigolf Punkteerfassung

<https://github.com/Kronnox/ase-minigolf>

Name: Adams, Jan Luca  
Martrikelnummer: 1012734

Abgabedatum: 29. Mai 2022

### Allgemeine Anmerkungen:

- *es darf nicht auf andere Kapitel als 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)*
- *sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)*
- *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*
  - *Beispiele*
    - *"Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt."*
      - *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: volle Punktzahl ODER falls im Code mind. eine Klasse SRP verletzt: halbe Punktzahl*
- *verlangte Positiv-Beispiele müssen gebracht werden*
- *Code-Beispiel = Code in das Dokument kopieren*

# Kapitel 1: Einführung

## Übersicht über die Applikation

Die entwickelte Anwendung erlaubt das digitale Festhalten der Punktestände zu einer Runde Minigolf.

Dabei werden die verschiedenen Bahnen der Minigolfanlage in einer CSV-Datei fest in die Anwendung integriert. In einer weiteren CSV-Datei sind nun eine Reihe vordefinierter Spielrouten definiert, welche sich aus einer Kombination dieser Bahnen zusammensetzen. Jeder Bahn ist dabei ein sogenannter „Par“-Wert zugeordnet, welcher angibt, wie viel Schläge durchschnittlich zum Absolvieren der Bahn benötigt werden. Die verschiedenen Spielrouten wiederum haben jeweils einen Namen zur einfachen Identifikation und eine für die Route festgelegten maximale Schlaganzahl. Das Loch muss in der angegebenen Maximalanzahl an Schlägen getroffen werden, ansonsten erhält der Spieler eine Strafe von zwei weiteren benötigten Schlägen. Bei einer Maximalanzahl von 12 erlaubten Schlägen, ergibt dies also zum Beispiel 14 benötigte Schläge.

Nach Auswahl der gewünschten Route gibt jeder Spieler einen Namen für die Punkteübersicht an. Dieser darf maximal 15 Zeichen lang sein.

Während dem Spiel ist dann dauerhaft eine Übersicht der „Par“-Werte aller Bahnen, der bei vorherigen Bahnen benötigten Schläge, als auch der Gesamtanzahl an Schlägen jedes Spielers aufgeführt. Ebenfalls wird angezeigt, beim wievielen Loch der gewählten Route man sich befindet. Das Mitzählen der benötigten Schläge im Kopf ist nicht mehr erforderlich, da dies ebenfalls durch die Software übernommen wird. Sollte ein Spieler eine Bahn nicht weiter spielen wollen kann er auch aufgeben und erhält sofort die maximale Schlaganzahl vermerkt.

Ziel der Applikation ist es eine umweltfreundliche Alternative beim Punktefassen zu den herkömmlichen und weit verbreiteten Papier-Blöcken zu bieten.

## Wie startet man die Applikation?

Zum Starten der Anwendung muss Java 18 auf dem jeweiligen System installiert sein.

- 1) Gehe zu <https://github.com/Kronnox/ase-minigolf/releases/latest> und lade die dort verfügbare „ase-minigolf.jar“ herunter
- 2) Navigiere in das Verzeichnis der JAR-Datei
- 3) Führe die JAR-Datei mit „java -jar ase-minigolf.jar“ aus

## Wie testet man die Applikation?

Zum Testen der Anwendung wird Java 18 und eine aktuelle Maven Installation benötigt.

- 1) Klone das Repository von <https://github.com/Kronnox/ase-minigolf>
- 2) Navigiere in das geklonte Verzeichnis
- 3) Führe „mvn clean test“ aus

## Kapitel 2: Clean Architecture

### Was ist Clean Architecture?

Der Begriff Clean-Architecture bezeichnet eine Design-Philosophie bei welcher der Code in Geltungsbereiche oder visuell dargestellte Ringebenen unterteilt wird. Hauptziel ist es wenig wandelnde Domänen-Logik von der schnellebigen Außenwelt zu trennen. Während sich die Implementation von Schnittstellen nach außen aufgrund von technischem Wandel oder anderen Einflüssen häufig ändern mag, bleibt die Kernlogik der Domäne meistens gleich.

Umgesetzt wird dies durch entsprechendes Abhängigkeits-Management. So kann eine Schicht nur Abhängigkeiten auf weiter innen liegende Schichten haben. Sollte es nun Änderungen an äußeren Schichten geben, so sind die inneren davon nicht betroffen. Sinnbildlich schützen die äußeren Schichten also die Kernlogik.

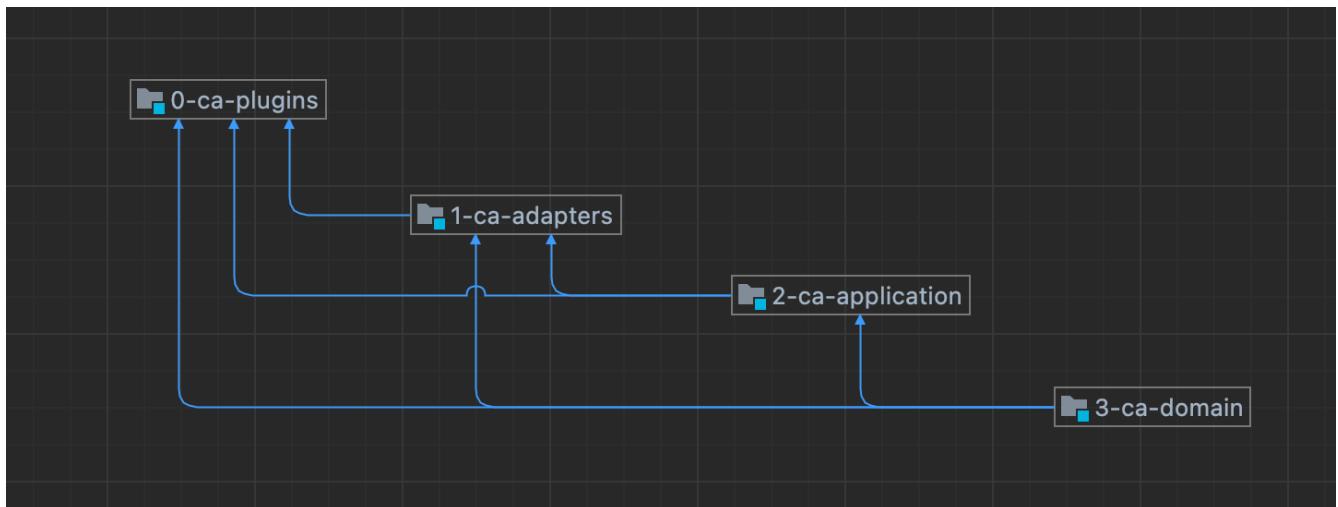
Um logische Zugriffe zwischen den Schichten zu ermöglichen, können in innenliegenden Schichten Abmachungen für Schnittstellen getroffen werden, welche in den äußeren Schichten erst konkret ausimplementiert werden. So können diese Schnittstellen trotzdem in inneren Schichten verwendet werden.

### Analyse der Dependency Rule

Die Einhaltung der Dependency-Rule wird durch den Einsatz einer entsprechenden Maven Projektstruktur „erzwungen“. Die jeweiligen Klassen sind nur in der eigenen unter untergeordneten Modulen verfügbar. Es ist also nicht möglich auf eine Klasse entgegen der Dependency-Rule zu verweisen. So dürfte es eigentlich keinen Bruch dieser und entsprechend kein Negativbeispiel geben.

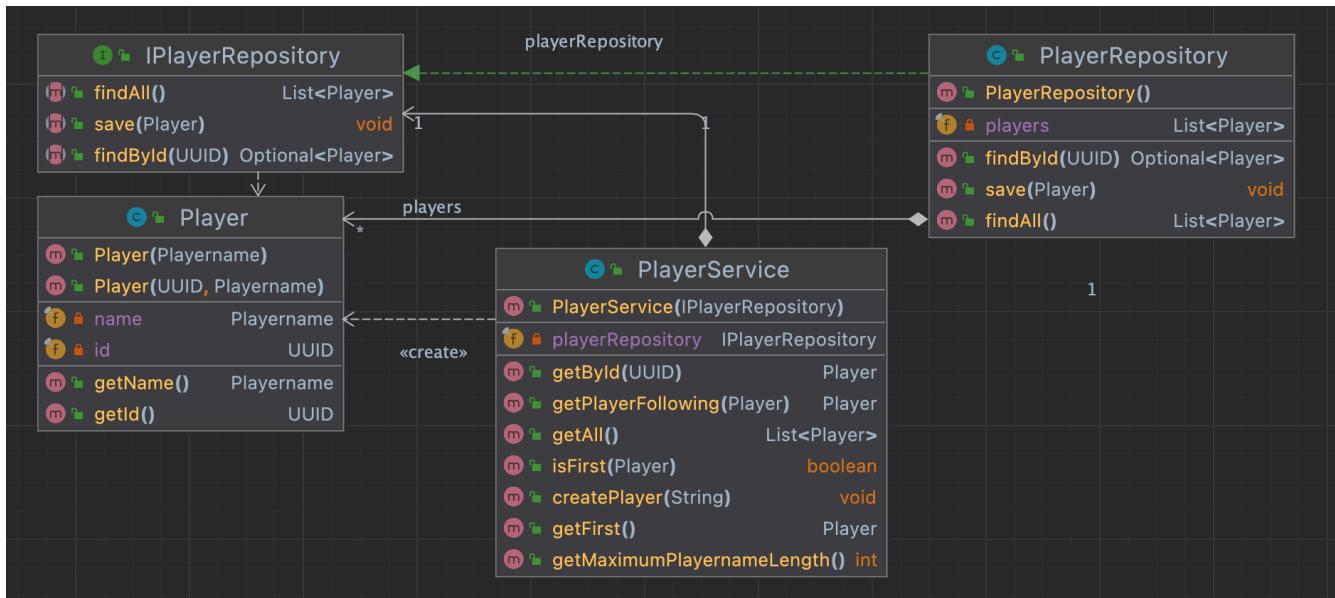
### Positiv-Beispiel 1

Bereits auf Modul-Ebene ist eine klare lineare Hierarchie zwischen den einzelnen Modulen zu verzeichnen. Dies garantiert die Einhaltung der Dependency-Rule zwischen diesen Modulen.



## Positiv-Beispiel 2

Feingranularer ist die Einhaltung der Regel im Dependency-Tree der Spieler-Verwaltung zu erkennen. Von den Klassen PlayerService (Applikations-Schicht) und PlayerRepository (Plugin-Schicht) wird nur auf Klassen der Domänen-Schicht zugegriffen. Es findet allerdings nie ein Zugriff in umgekehrter Richtung statt. Die Dependency-Rule wird also eingehalten.

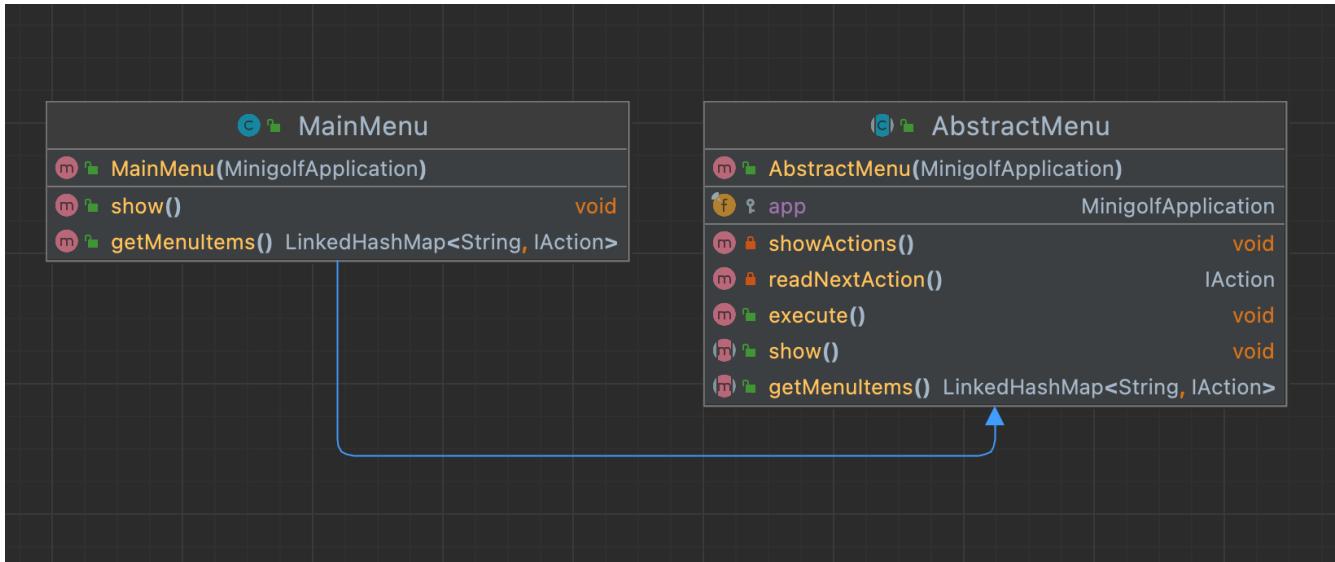


## Analyse der Schichten

### Schicht 0: Plugins

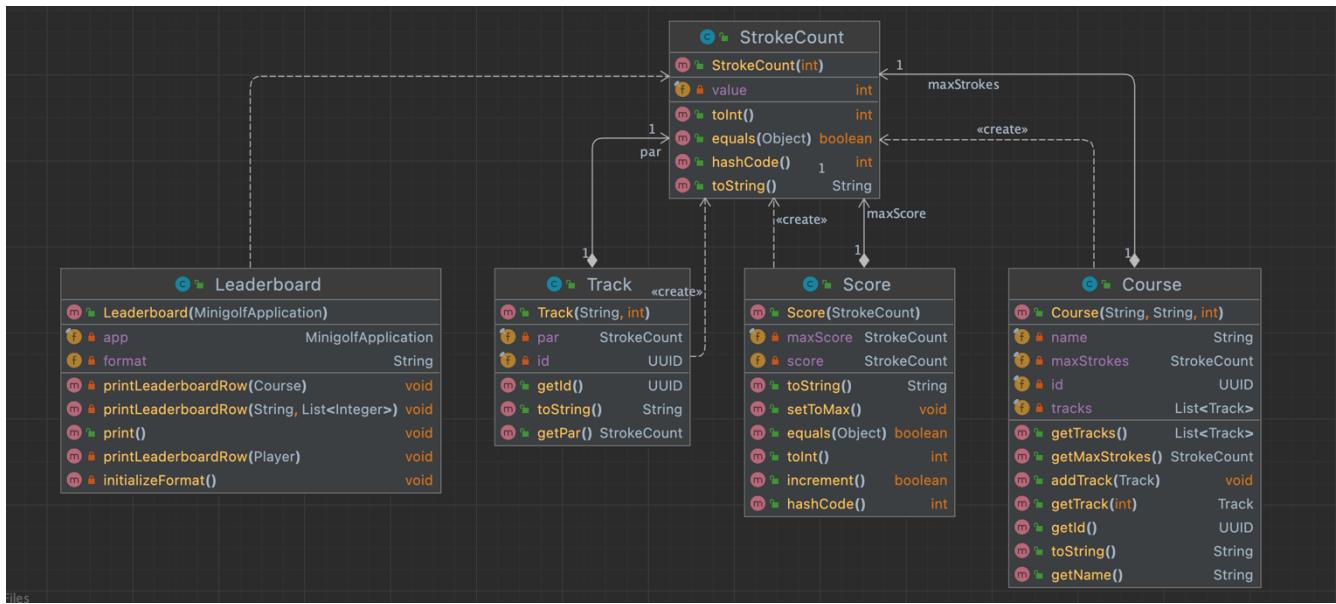
Die Main-Menu Klasse und die dazu gehörige AbstractMenu Klasse befindet sich in Schicht 0 der Clean-Architecture, da sie die Implementation für die Interaktion mit dem Nutzer über eine Oberfläche darstellt. Sie ist stark abhängig von der technologischen Umgebung, in diesem Fall von der Konsole.

Sie ist nicht weiter relevant für den Domänen-Kontext oder die Applikations-Logik und könnte beliebig gegen eine andere Umsetzung getauscht werden.



### Schicht 3: Domain

Bei der StrokeCount Klasse handelt es sich um ein ValueObject aus dem inhaltlichen Kontext der Domäne. Es bezeichnet eine Anzahl von Schlägen. Aufgrund seiner Zugehörigkeit zur Domäne befindet es sich in Schicht 3 der Clean-Architecture und damit sehr weit oben in der Hierarchie. Sie kann also in der eigenen und allen unter ihr liegenden Schichten verwendet werden. Es ist äußerst selten, dass sich etwas an den Domänen-Objekten ändert.



## Kapitel 3: SOLID

### Analyse Single-Responsibility-Principle (SRP)

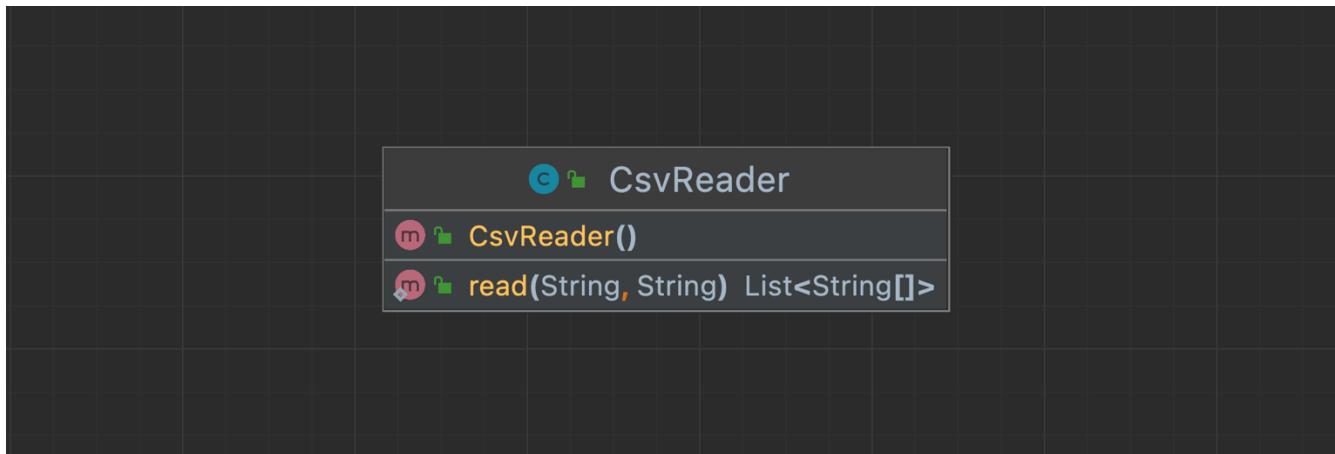
Das SRP kann auf verschiedenste Scopes und Granularitätsstufen angewendet werden. Das bedeutet zum Beispiel auf Modul-, Klassen-, Methoden- oder sogar Variablenebene.

Generell geht es darum, dass jede dieser Entitäten jeweils nur für die Erfüllung genau einer Aufgabe zuständig ist und von anderen Aufgabengebieten klar separiert ist.

Eigenständige und unabhängige Entitäten sind besser wiederzuverwenden und lassen sich einfacher testen.

#### **Positiv-Beispiel**

Die Klasse CsvReader besitzt nur eine Verantwortlichkeit, nämlich das Einlesen von CSV-Dateien (in eine Liste von String-Arrays).



## Negativ-Beispiel

Die Klasse SessionManager mag vielleicht alles Inhalte des selben groben Anwendungsgebietes enthalten, was allerdings nichts daran ändert, dass es grundlegend verschiedene und unabhängige Aufgaben sind.

SessionManager	
<b>m</b> SessionManager(PlayerService, ScoreService)	
f	playerService PlayerService
f	currentPlayer Player
f	currentScore Score
f	scoreService ScoreService
f	currentTrackIndex int
f	currentCourse Course
m	goToNextTrack() void
m	getCurrentCourse() Course
m	getCurrentTrackIndex() int
m	getCurrentPlayer() Player
m	setCurrentCourse(Course) void
m	getCurrentScore() Score
m	goToNextPlayer() void

Sie enthält Methoden zur Verwaltung der Spielerrotation, des aktuellen Punktestandes, der aktuellen Spielroute und der aktuell bespielten Bahn.

Um das SRP zu wahren, wäre es empfehlenswert die Klasse auf mehrere Klassen aufzuteilen. Anstatt einem zentralen SessionManager würde die zuvor genannten Aufgaben auf einzelne Klassen für die Zustandsspeicherung aufgeteilt werden.

Eine Aufteilung könnte dabei wie folgt aussehen:

PlayerSessionManager	CourseSessionManager	ScoreSessionManager	
<b>m</b> PlayerSessionManager(PlayerService)	<b>m</b> CourseSessionManager()	<b>m</b> ScoreSessionManager(ScoreService)	
f	currentCourse Course	f	scoreService ScoreService
f	currentTrackIndex int	f	currentScore Score
m	getCurrentCourse() Course	m	getCurrentScore() Score
m	getCurrentTrackIndex() int		
m	setCurrentCourse(Course) void		
m	goToNextTrack() void		

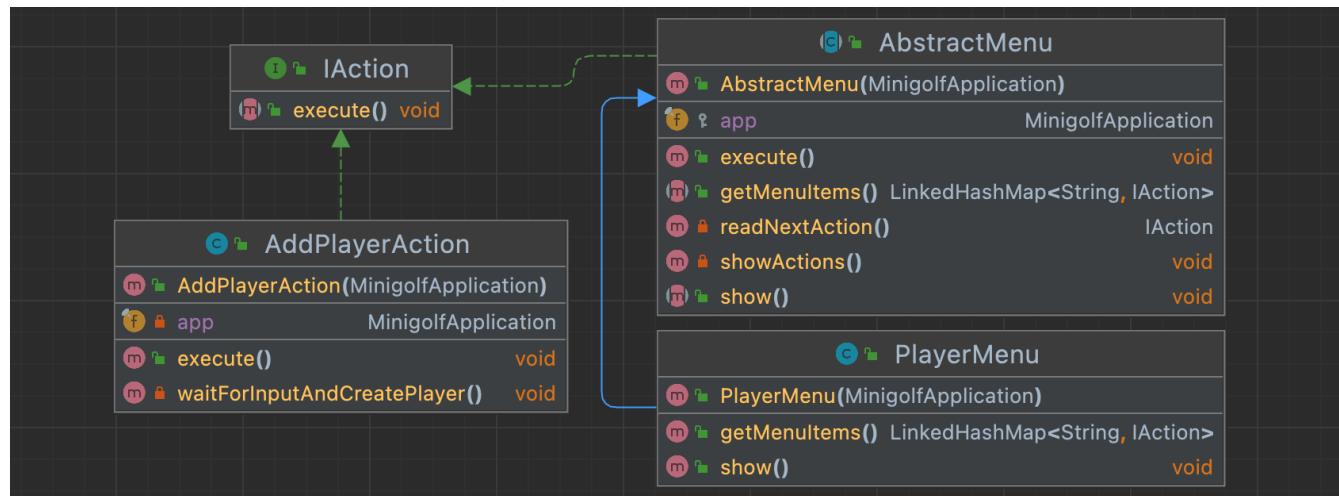
## Analyse Open-Closed-Principle (OCP)

Klassen sollten offen für Erweiterungen und geschlossen für Modifikationen sein.

### Positiv-Beispiel

Für die Interaktion auf der Konsolenoberfläche wird eine spezielle Datenstruktur für das Anlegen von ausführbaren Aktionen (Actions) verwendet. Eine Aktion kann dabei sehr verschieden ausschauen. Es mag das öffnen eines Menüs bzw. Dialogfensters sein, oder zum Beispiel das Ausführen von jeglichem Code.

Daher wird hier das IAction Interface verwendet. Dieses definiert die nötige Schnittstelle für das interagieren mit einer Action.



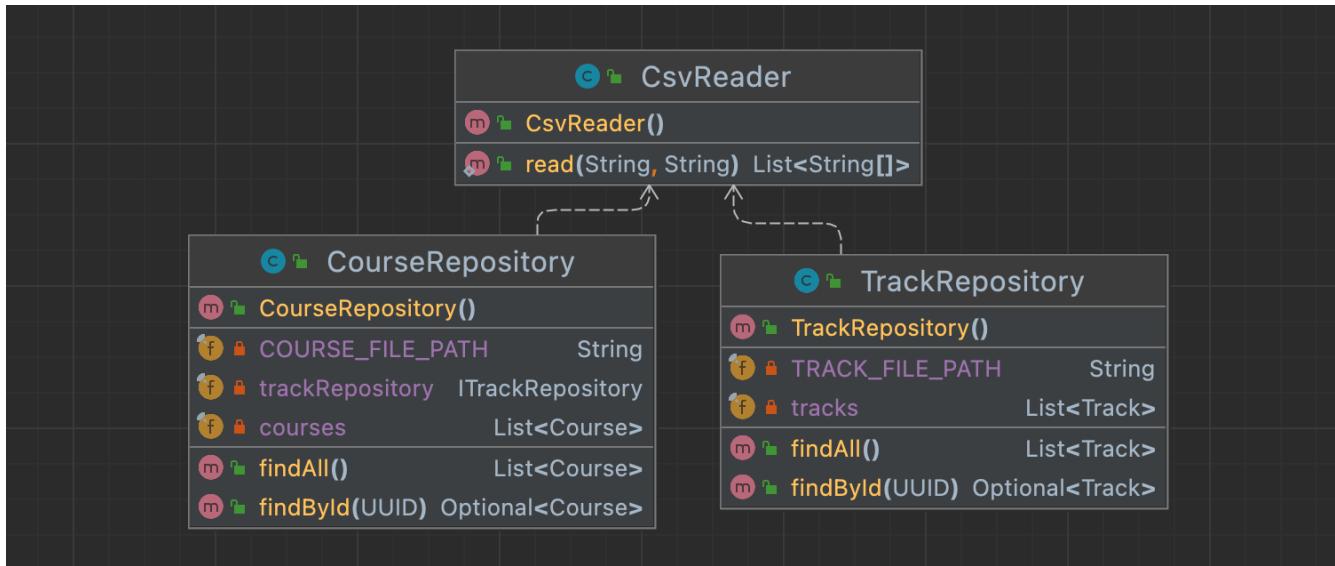
Sowohl das AbstractMenu, als auch eigenständige Aktionen, wie die AddPlayerAction verfügen nun jeweils über Implementierungen den Interfaces. So bietet die execute Methode eine einheitliche Schnittstelle unabhängig von der dahinterstehenden Implementation.

Eine Action kann also unabhängig von ihrer Funktion aufgerufen werden:

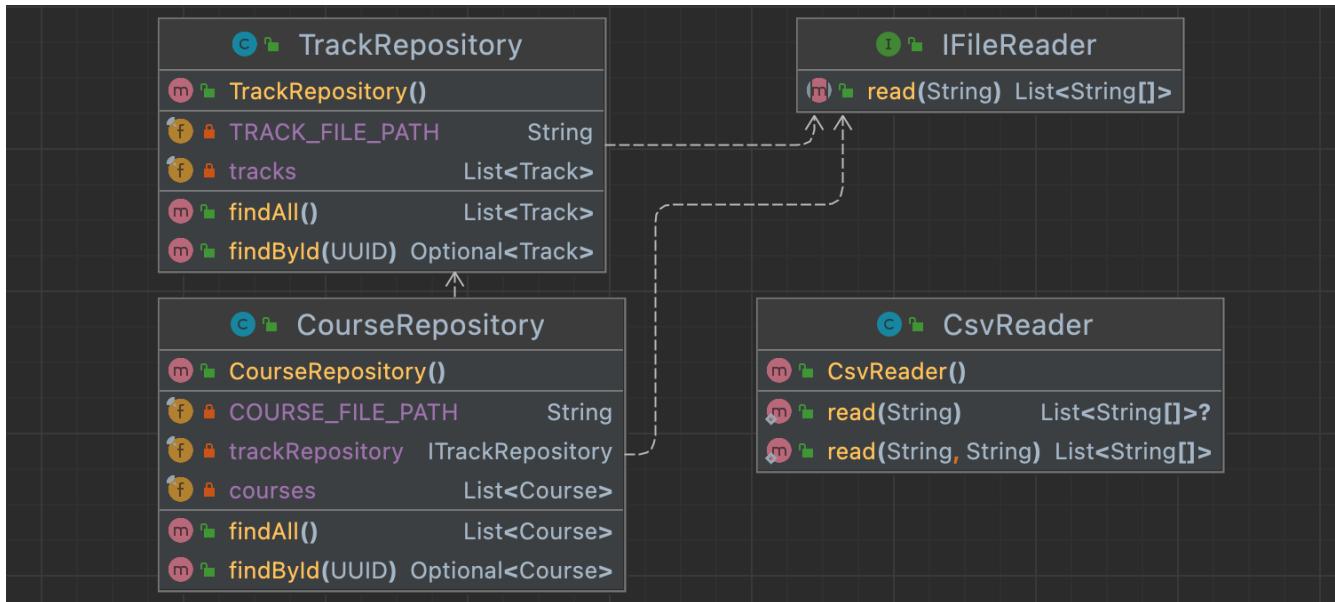
```
IAction action = readNextAction();
action.execute();
```

## Negativ-Beispiel

Ein negatives Beispiel für OCP finden wir beim CsvReader.



Wie im Diagramm zu erkennen referenzieren beide Repository Klassen die CsvReader Implementation direkt. Bei Änderungen kann es also schnell zu Problemen kommen. Da es hier nur eine Reader Implementation gibt ist das aktuell weniger problematisch, generell wäre es allerdings sinnvoll ein Interface zwischenzuhängen:

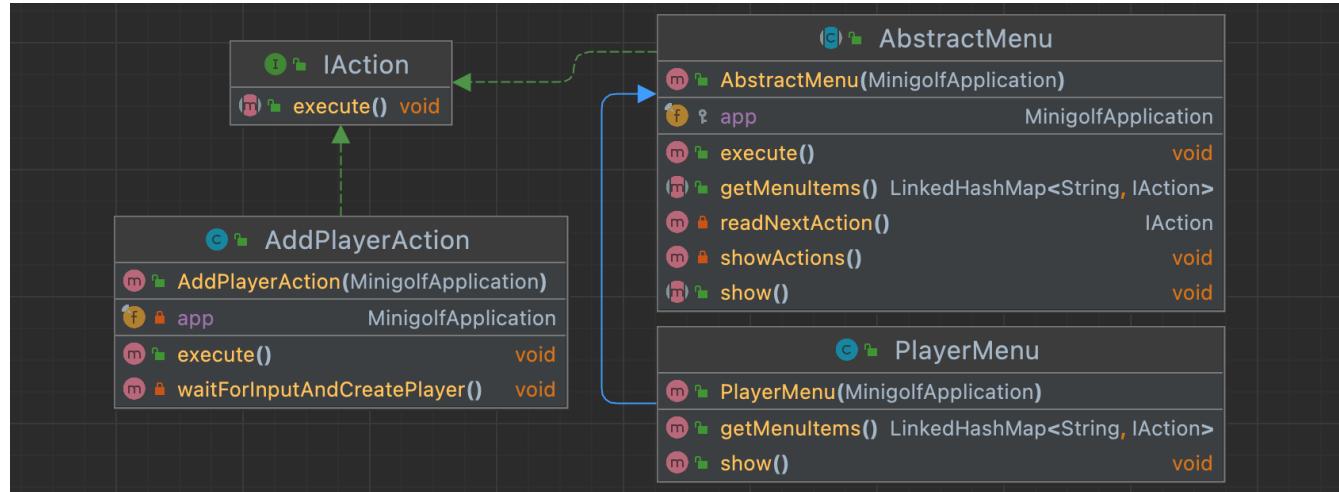


# Analyse Liskov-Substitution- (LSP), Interface-Segregation- (ISP), Dependency-Inversion-Principle (DIP)

Bei dem Dependency-Inversion-Principle geht es darum, dass

## Positiv-Beispiel

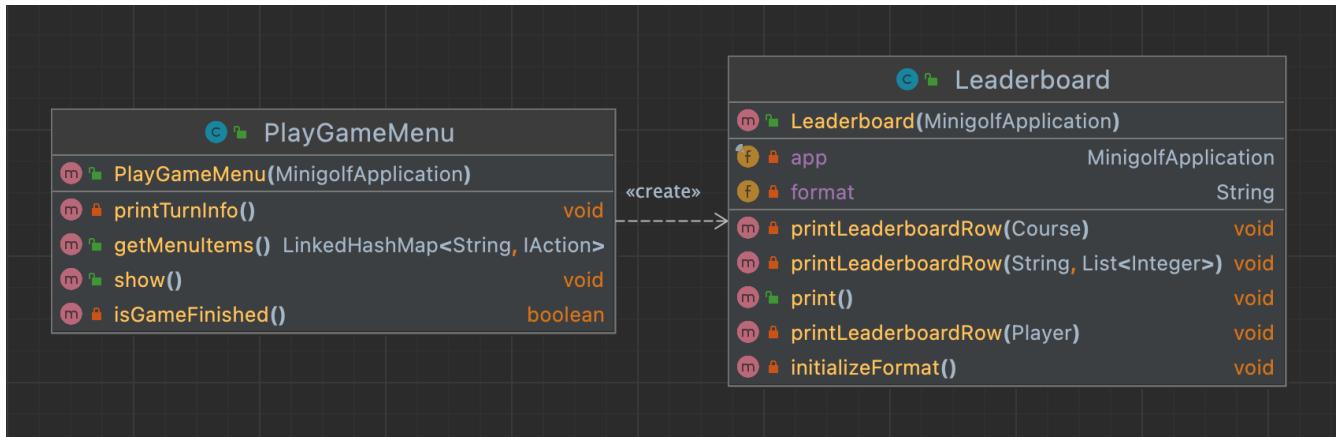
Die in einem Menü auswählbaren Interaktionen verwenden ein Interface IAction. Dieses enthält eine execute Methode.



Klassen, welche nun, eine Action aufrufen sind nicht an eine konkrete Action Klasse gebunden. Eine High-Level Klasse kann hierbei unabhängig von einer Low-Level Klasse agieren und ist nicht direkt von dieser abhängig.

## Negativ-Beispiel

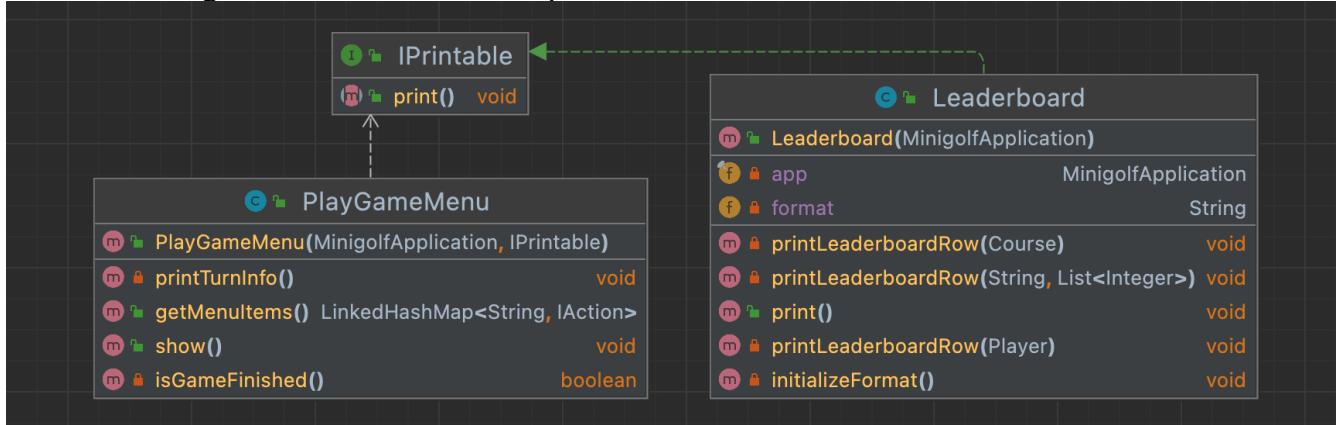
Ein Negativ Beispiel bezüglich der DIP finden wird beim in der PlayGameMenu Klasse bezüglich der Leaderboard Klasse.



Die Leaderboard Klasse wird hier direkt referenziert und ist damit direkt abhängig von dieser.

```
new Leaderboard(app).print();
```

Um dem zu entgehen, müsste hier ein entsprechendes Interface verwendet werden:



## Kapitel 4: Weitere Prinzipien

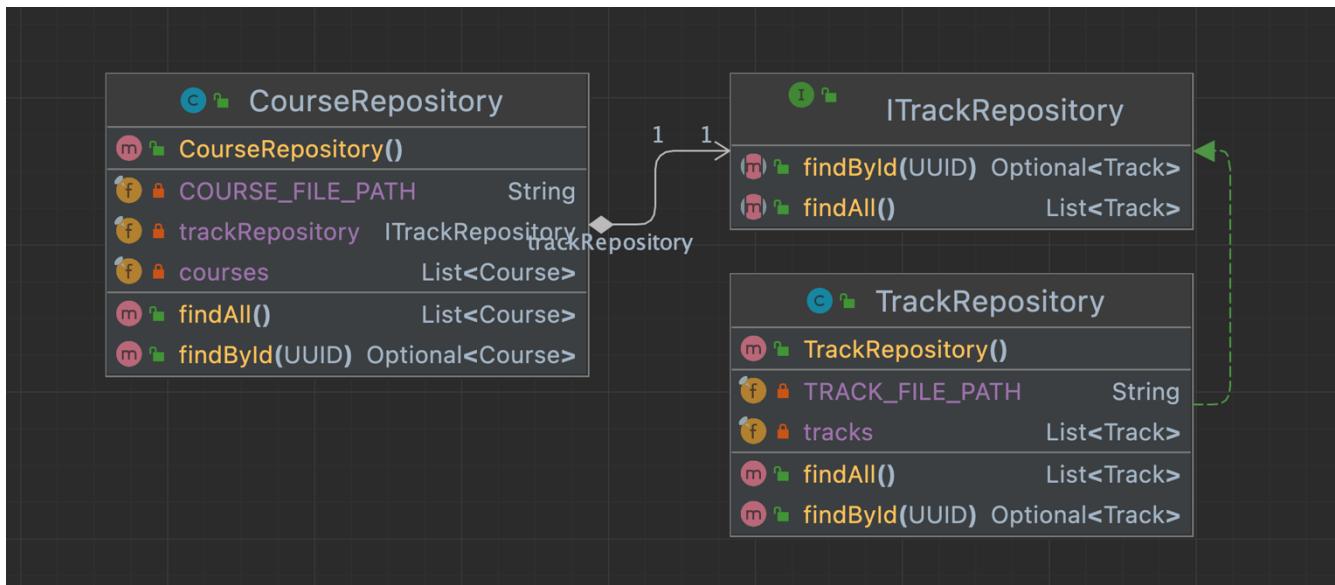
### Analyse GRASP: Geringe Kopplung

#### Positiv-Beispiel

Alle Interaktionen mit dem Repository für Golfbahnen finden mit dem dazugehörigen ITrackRepository statt und nicht durch direkte Zugriffe auf die Implementation TrackRepository. Dadurch sind Schnittstellen-Definition und Implementation „entkoppelt“. Die Implementation kann also ohne Bedenken angepasst werden, da alle Zugriffe über das Interface geschehen.

Das Repository generell wird verwendet, um die hinterlegten Golfbahnen abrufen zu können.

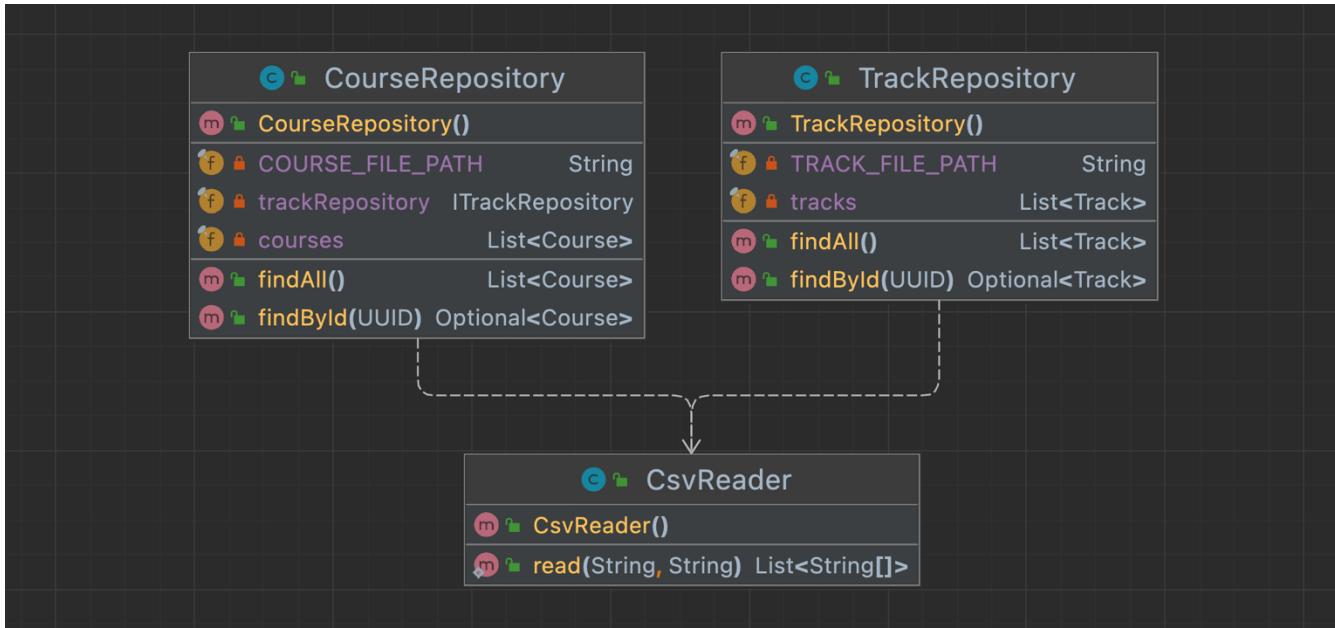
Jegliche Komplexität liegt rein bei der Implementierung und nicht im Interface, welches von anderen Klassen verwendet wird.



#### Negativ-Beispiel

Die Klassen CourseRepository und TrackRepository machen direkte Zugriffe auf die Implementation der CsvReader Klasse. Bei der CsvReader Klasse handelt es sich folglich um eine Klasse mit hoher Komplexität.

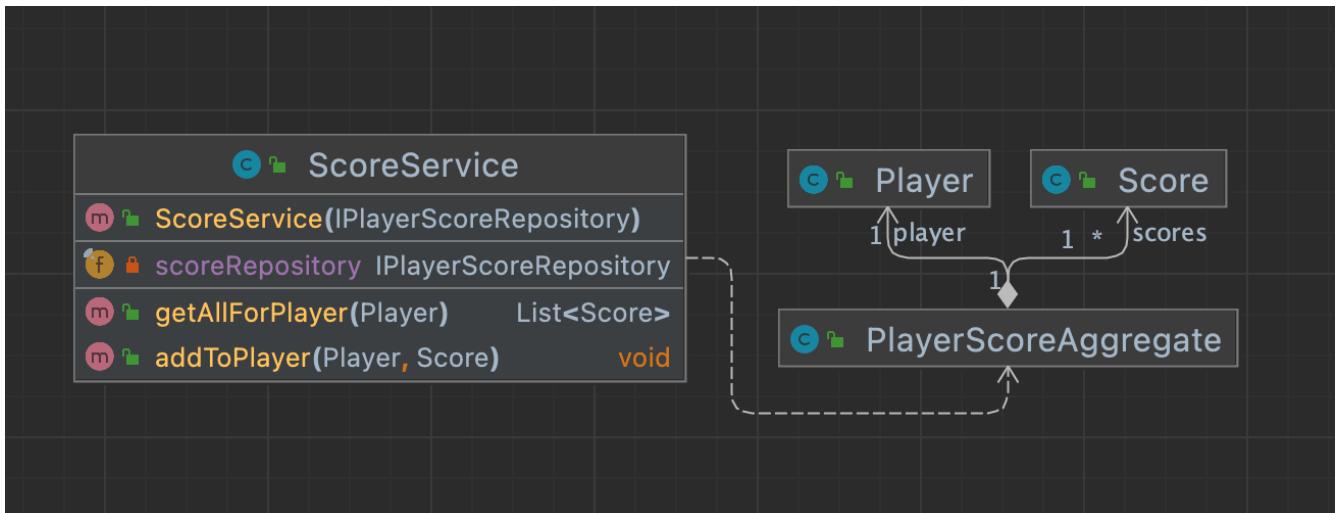
Die Kopplung könnte deutlich reduziert werden, in dem nicht direkt auf die Implementation zugegriffen wird, sondern auf ein dazugehöriges Interface. Dieses würde als Schnittstellen-Vereinbarung zwischen Repositories und Implementation dienen, sodass Änderungen an der Implementation im Rahmen dieser Vereinbarung weniger potentiell fatale Auswirkungen haben würden.



## Analyse GRASP: Hohe Kohäsion

Aufgrund der generell niedrigen Komplexität der im Projekt verwendeten Klassen fällt es schwer ein wirklich aussagekräftiges Beispiel für hohe Kohäsion zu finden.

Die Klasse PlayerScoreAggregate kommt dem wohl noch am nächsten.



Anstatt die Eigenschaften von Spieler, Punkteanzahl und Schlaganzahl in der PlayerScoreAggregate Klasse als Felder zu definieren, sind diese auf die jeweiligen Unterklassen ausgelagert. Dies folgt dem Single Responsibility Prinzip, stärkt zugleich allerdings auch die Kohäsion der PlayerScoreAggregate Klasse. Daher haben wir es hier mit einer hohen Kohäsion zu tun

## Don't Repeat Yourself (DRY)

<https://github.com/Kronnox/ase-minigolf/commit/e032908ed28f62b479cea35a13643c046b1ca4a3>

In der Klasse Leaderboard enthielten die Methoden printLeaderboardRow(Course course) und private void printLeaderboardRow(Player player) ähnliche Code-Blöcke.

```
private void printLeaderboardRow(Course course) {
    List<Integer> scores = course.getTracks().stream().map(Track::getPar)
        .map(StrokeCount::toInt).toList();
    List<Object> parameters = new ArrayList<>();

    // Label section
    parameters.add("(Par)");

    // Score section
    int totalScore = 0;
    for (int i = 0; i < app.getSession().getCurrentCourse().getTracks().size(); i++) {
        if (i >= scores.size()) {
            parameters.add("-");
            continue;
        }
        int score = scores.get(i);
        parameters.add(score);
        totalScore += score;
    }

    // Total score section
    parameters.add(totalScore);

    // Final print out
    System.out.format(format, parameters.toArray());
}

private void printLeaderboardRow(Player player) {
    List<Integer> scores = app.getScoreService().getAllForPlayer(player).stream()
        .map(Score::toInt).toList();
    List<Object> parameters = new ArrayList<>();

    // Label section
    parameters.add(player.getName().toString());

    // Score section
    int totalScore = 0;
    for (int i = 0; i < app.getSession().getCurrentCourse().getTracks().size(); i++) {
        if (i >= scores.size()) {
            parameters.add("-");
            continue;
        }
        int score = scores.get(i);
        parameters.add(score);
        totalScore += score;
    }

    // Total score section
    parameters.add(totalScore);
    // Final print out
    System.out.format(format, parameters.toArray());
}
```

Im Kern schreiben Beide Methoden eine Zeile des Leaderboards in die Konsole. In einem Fall zum Anzeigen der „Par“-Werte, im anderen für die Punkteanzahlen eines Spielers. Sie machen also das selbe nur mit einem abweichenden Label und einer unterschiedlichen Liste an Zahlenwerten.

Durch Auslagerung des gemeinsamen Codes in eine separate Methode kann dies deutlich optimiert werden:

```
private void printLeaderboardRow(Course course) {
    printLeaderboardRow(
        "(Par)",
        course.getTracks().stream().map(Track::getPar).map(StrokeCount::toInt).toList()
    );
}

private void printLeaderboardRow(Player player) {
    printLeaderboardRow(
        player.getName().toString(),
        app.getScoreService().getAllForPlayer(player).stream().map(Score::toInt).toList()
    );
}
private void printLeaderboardRow(String label, List<Integer> scores) {
    List<Object> parameters = new ArrayList<>();

    // Label section
    parameters.add(label);

    // Score section
    int totalScore = 0;
    for (int i = 0; i < app.getSession().getCurrentCourse().getTracks().size(); i++) {
        if (i >= scores.size()) {
            parameters.add("-");
            continue;
        }
        int score = scores.get(i);
        parameters.add(score);
        totalScore += score;
    }

    // Total score section
    parameters.add(totalScore);

    // Final print out
    System.out.format(format, parameters.toArray());
}
```

Durch die Reduktion des übereinstimmenden Codes ist die Klasse Leaderboard deutlich besser lesbar und die Wiederverwendbarkeit der Methoden wurde erhöht.

# Kapitel 5: Unit Tests

## 10 Unit Tests

Unit Test	Beschreibung
TrackRepositoryTest#findAll	Testet das Abfragen aller Bahnen auf Vollständigkeit
TrackRepositoryTest#findOne	Testet das Abfragen einer einzelnen Bahn via ID
TrackRepositoryTest#findInvalid	Testet das invalide IDs ein leere Optional zurückgeben
ScoreServiceTest#getAllForPlayer	Testet das Abrufen aller zu einem Spieler gehörenden Punktestände
PlayernameTest#invalidValues	Testet das Fehlschlagen der Namenserstellung bei zu langer oder zu kurzer Eingabe
PlayernameTest#methods	Testet die Methoden des Value Objects zum Konvertieren in String und Integer
ScoreTest#setToMax	Testet, ob der Wert beim Aufruf der Methode setToMax() zwei höher als das Maximum gesetzt wird
ScoreTest#increment	Testet, ob die Methode increment den Wert um 1 erhöht, bis zum Maximum, wo der Wert 2 höher als dieses gesetzt wird
CsvReaderTest#invalidPath	Testet, ob eine FileNotFoundException geworfen wird, bei Angabe eines ungültigen Pfades
CsvReaderTest#validPath	Testet das korrekte Parsen einer gültigen CSV-Datei

### ATRIP: Automatic

Durch Integration der Testbibliotheken in Maven wird der Prozess zum Durchlaufen aller Tests mit jedem vollständigen Build-Vorgang ausgeführt. (Wäre mein monatliches Kontingent für GitHub-Actions Zyklen nicht aufgebraucht, würde ich den dazugehörigen Maven Job ebenfalls mit jedem Push ausführen...)

### ATRIP: Thorough

Die Tests sollen alle relevanten und kritischen Funktionen abdecken. Generell ist dabei jedoch viel Interpretations-Spielraum gelassen und es liegt letztlich im Ermessen eines jeden Entwicklers, was er für sinnvoll hält.

Dies umfasst in jedem Fall die Kernfunktionalität in Bezug auf den Auftrag bzw. die Domäne. Daher wird von vielen auch das Prinzip des „Test before Code“ angewandt, bei dem zunächst Tests für alle domänenrelevanten Funktionen angelegt werden und anschließend erst die entsprechende Implementierung angegangen wird.

### Positiv Beispiel:

Am Beispiel des StrokeCount Value Objects sehen wir eine gute Abdeckung.

```

@Test
void validValues() {
    assertDoesNotThrow(() -> new StrokeCount(1));
    assertDoesNotThrow(() -> new StrokeCount(12));
    assertDoesNotThrow(() -> new StrokeCount(Integer.MAX_VALUE));
}

@Test
void invalidValues() {
    assertThrows(IllegalArgumentException.class, () -> new StrokeCount(-1));
    assertThrows(IllegalArgumentException.class, () -> new StrokeCount(0));
}

@Test
public void methods(){
    assertDoesNotThrow(() -> {
        assertEquals(10, new StrokeCount(10).toInt());
        assertEquals("10", new StrokeCount(10).toString());
    });
}

```

Für jede Variante einer invaliden Eingabe von Konstruktor-Parametern existiert ein Test-Fall. Gleiches gilt für Varianten von akzeptierten Konstruktor-Parametern. Darüber hinaus wird die Funktion jeder der Methoden des Value-Objects getestet. Bei den verwendeten Werten handelt es sich immer um die Grenzwerte des jeweiligen akzeptierten bzw. nicht akzeptierten Bereiches und um einen frei gewählten realistischer Wert aus der echten Welt.

Sicherlich wäre es möglich auf noch mehr Eingaben zu testen, dies würde allerdings in den meisten Fällen keine bessere Abdeckung und damit Erkennung von potenziellen Fehlfunktionen bieten.

Man kann also sagen, dass das Value-Object StrokeCount vollständig getestet wird.

### Negativ Beispiel:

```

@Test
void validValues() {
    assertDoesNotThrow(() -> new StrokeCount(1));
}

```

Würden wir z.B. nur einen akzeptierten Wert testen, um die „geradeaus Funktion“ der Klasse zu evaluieren, riskieren wir Fehlverhalten bei sogenannten Edge-Cases. Durch die zuvor genannten weiteren Test-Fälle könnte dem vorgebeugt werden.

Eine solche Testabdeckung wäre nicht „Thorough“.

Das komplette Auslassen von jeglichen Tests, wie es beim PlayerService der Fall ist, wäre natürlich ebenfalls ungenügend.

## ATRIP: Professional

Nur weil es sich bei Tests nicht um Produktiv-Code handelt, sollte dieser in seiner Qualität nicht minderwertiger als jener sein. Es gelten die selbigen Anforderungen an die Sauberkeit, Übersichtlichkeit und Effizienz, wie beim Produktiv-Code.

Die Tests sollten einfach von Menschen gelesen und verstanden werden können. Sprechende Variablennamen und eine strukturierte Gliederung sind also unabdingbar. Auch ist es möglich Tests in mehrere Submethoden aufzuteilen, um lange Testmethoden zu vermeiden.

### Positiv Beispiel:

```
@Test
void findAll() {
    LinkedList<String[]> tracks = new LinkedList<>();
    tracks.add(new String[]{"13935c14-0929-40ee-9379-c7aa002c163e", "4"});
    tracks.add(new String[]{"448ed1b6-8ea3-44b9-afaa-446715bde61a", "6"});

    List<Track> findAll;
    try (MockedStatic<CsvReader> mockedReader = Mockito.mockStatic(CsvReader.class)) {
        mockedReader.when(() → CsvReader.read(any(), any())).thenReturn(tracks);

        TrackRepository repo = new TrackRepository();
        findAll = repo.findAll();
    }
    assertEquals(2, findAll.size());
}
```

Trotz der komplexeren Funktionsweise ist auch für außenstehende Betrachter schnell erkenntlich, wie der vorliegende Test funktioniert. Variablennamen wie „tracks“ und „findAll“ sind sprechend gewählt und die Methode ist keine 15 Zeilen lang.

### Negativ Beispiel:

Alle der im Projekt enthaltenen Tests folgen dem „Professional“ Standard. Eine weniger professionelle Variante der zuvor genannten Test-Methode wäre wie folgt:

```
@Test
void findAll() {
    LinkedList<String[]> a;
    a = new LinkedList<>();
    a.add(new String[]{"13935c14-0929-40ee-9379-c7aa002c163e", "4"});
    List<Track> b;
    a.add(new String[]{"448ed1b6-8ea3-44b9-afaa-446715bde61a", "6"});
    try (MockedStatic<CsvReader> c = Mockito.mockStatic(CsvReader.class)) {
        c.when(() → CsvReader.read(any(), any())).thenReturn(a);

        TrackRepository d = new TrackRepository();
        d = c.findAll();
    }
    assertTrue(b.size() = 2);
}
```

Variablennamen aus jeweils einem wahllos gewählten Zeichen und unsinnvolle Gliederung verhindert die Lesbarkeit erheblich. Ebenfalls können einige der verwendeten Ausdrücke verkürzt werden. Ein Beispiel hierfür wäre „assertTrue(b.size() = 2)“, was auch als „assertEquals(2, b.size())“ geschrieben werden könnte.

## Code Coverage

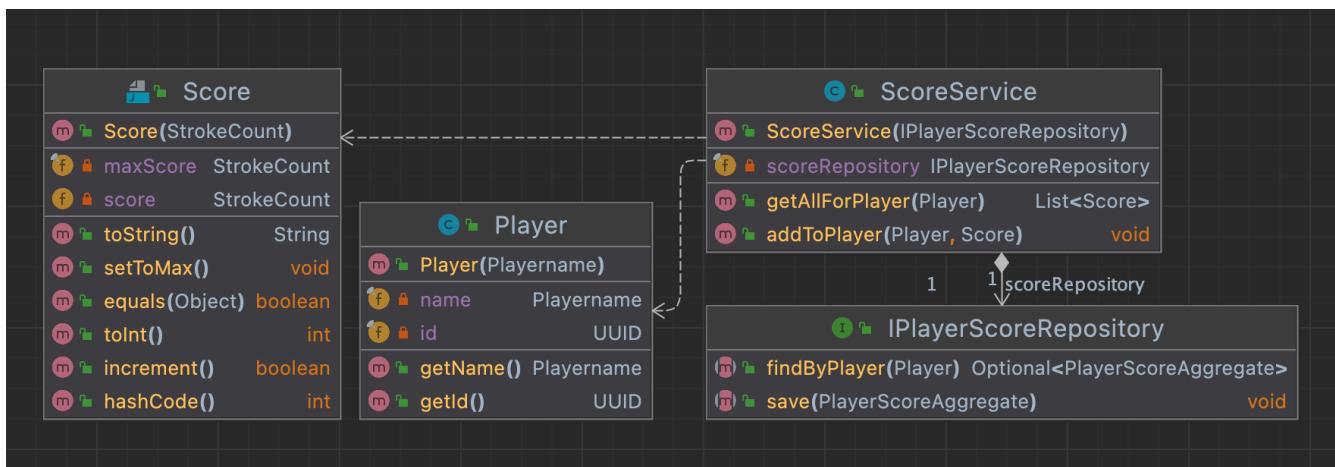
Aufgrund der geringen Anzahl an umgesetzten Tests in diesem Projekt sehen wir eine miserable Code Coverage bezogen auf das gesamte Projekt.

Die einzelnen Klassen, welche tiefgehend getestet werden, wie Playername und TrackRepository wiederum haben eine sehr gute Abdeckung. Bei anderen Klassen, für welche keine Tests geschrieben wurden, kann es auch keine gute Abdeckung geben.

## Fakes und Mocks

### Beispiel 1:

Der ScoreService referenziert auf eine Menge anderer Entitäten, das IPlayerScoreRepository, den PlayerScoreAggregate, den Player und den Score.



Um Tests der ScoreService Klasse von all diesen Entitäten zu entkoppeln, müssen diese gemockt werden. Dies verhindert, dass sich Fehler im Verhalten dieser Objekte auf die Tests auswirken würden und somit, dass die Ergebnisse verfälscht werden.

Ebenfalls ist es notwendig, da in der Schicht, in welcher der Test liegt, keine Implementation des IPlayerScoreRepository vorhanden ist. Durch den Mock ist dies aber nicht weiter schlimm, da Werte bestimmt werden können, welchem beim Aufruf der Interface Methoden zurückgegeben werden.

Der Code um die Mockobjekte zu erzeugen ist nachfolgen aufgeführt. Die Objekte können im Rahmen der gemockten Methoden, wie normale Objekte im Test verwendet werden.

```
Score score = mock(Score.class);
when(score.toInt()).thenReturn(6);

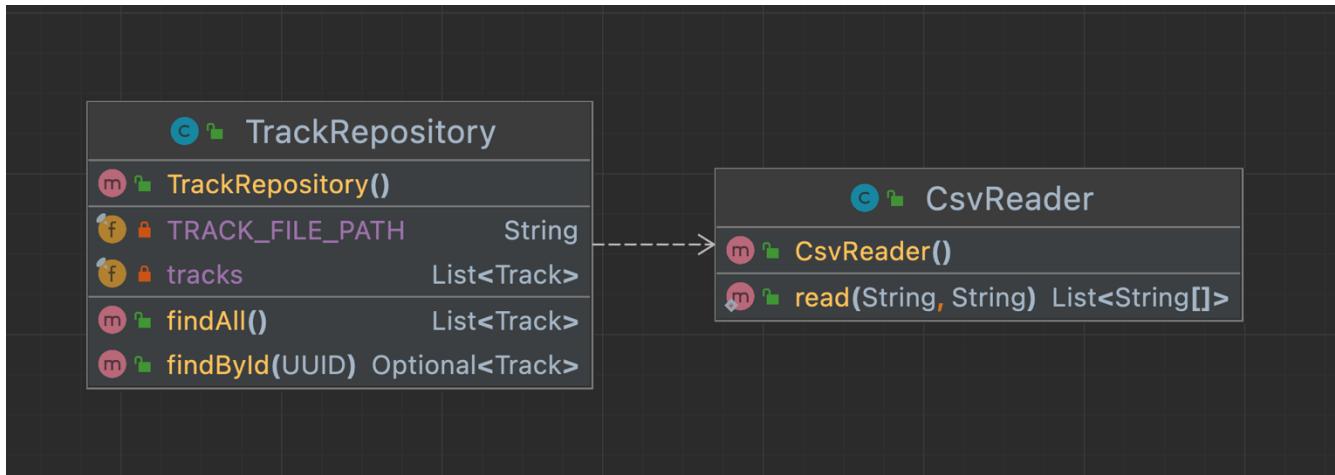
player = mock(Player.class);

PlayerScoreAggregate playerScore = mock(PlayerScoreAggregate.class);
when(playerScore.getScores()).thenReturn(new ArrayList<>(){
    add(score);
});

repo = mock(IPlayerScoreRepository.class);
when(repo.findByPlayer(any())).thenReturn(Optional.of(playerScore));
```

## Beispiel 2:

Da die Klasse TrackRepository die Klasse CsvReader referenziert müssen wir diese mocken, um auszuschließen, dass Fehler an dieser zu einem fehlgeschlagenen Test beim TrackRepository führen.



Auf diese Weise können wir die beiden Klassen für die Tests entkoppeln.

Eine Implementierung des gemockten CsvReader sieht dann wie folgt aus:

```
@Test
void findAll() {
    LinkedList<String[]> tracks = new LinkedList<>();
    tracks.add(new String[]{"13935c14-0929-40ee-9379-c7aa002c163e", "4"});
    tracks.add(new String[]{"448ed1b6-8ea3-44b9-afaa-446715bde61a", "6"});

    List<Track> findAll;
    try (MockedStatic<CsvReader> mockedReader = Mockito.mockStatic(CsvReader.class)) {
        mockedReader.when(() -> CsvReader.read(any(), any())).thenReturn(tracks);

        TrackRepository repo = new TrackRepository();
        findAll = repo.findAll();
    }
    assertEquals(2, findAll.size());
}
```

Da es sich beim CsvReader um eine statische Klasse handelt, muss er entgegen dem ersten Beispiel durch das generische MockedStatic Objekt gemockt werden.

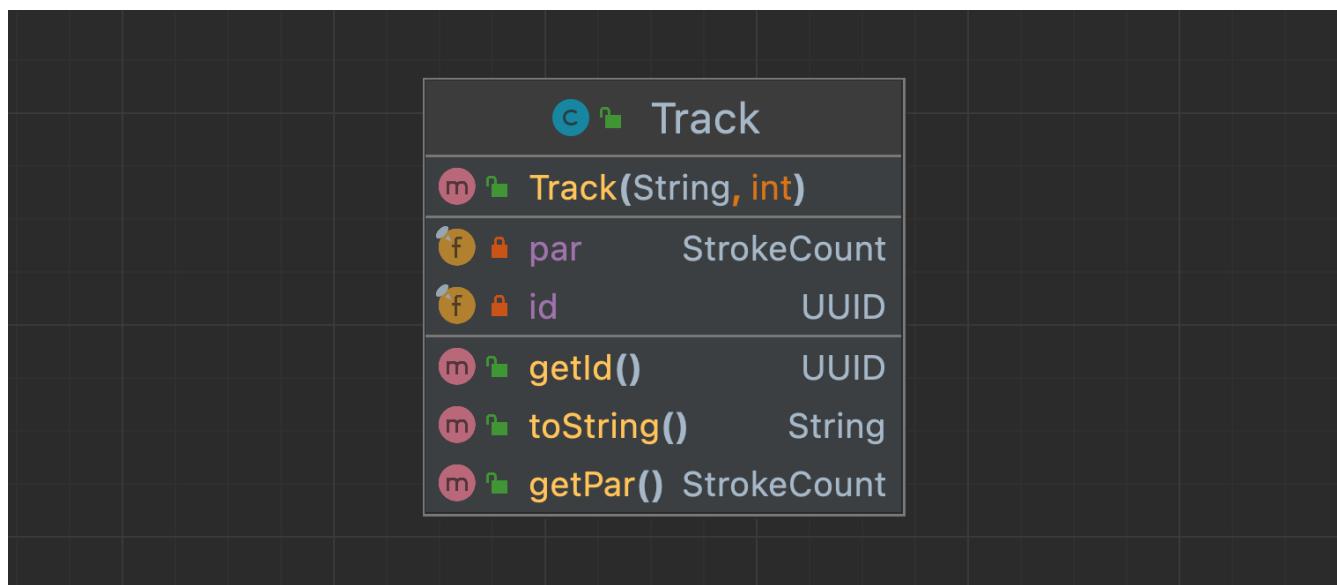
## Kapitel 6: Domain Driven Design

### Ubiquitous Language

Bezeichung	Bedeutung	Begründung
StrokeCount	Anzahl an Schlägen	Anzahl an Schlägen mit dem Golfschläger
Player	Mitspieler	Tägt Schläge und hat Ergebnisse auf den Bahnen
Course	Spielroute	Zusammensetzung mehrerer Bahnen mit Metadaten
Track	Minigolf-Bahn	Eine Bahn, die bespielt werden kann

### Entities

Ein durch eine ID klar identifizierbares Objekt mit Eigenschaften. Änderungen, welche an dieser vollzogen werden, sind nicht in der Lage ihre Werte zu invalidieren.

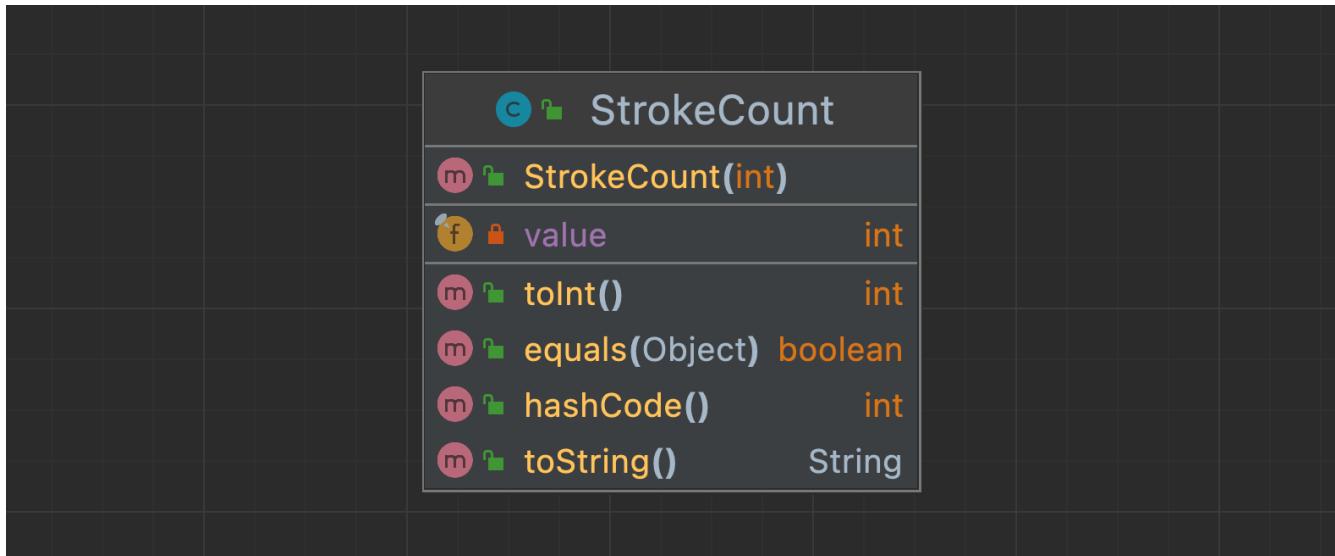


Eine Minigolfbahn wird nur durch ihre ID von einer anderen Bahn unterschieden und kann mit dieser klar zugeordnet werden. Es können zwei Bahnen mit denselben Metadaten, aber abweichender ID existieren. Auf dem Minigolfplatz handelt es sich um unterschiedliche physische Bahnen, welche lediglich dieselben Eigenschaften aufweisen.

## Value Objects

Werden ausschließlich durch den in ihnen enthaltenen Wert identifiziert. Sie können verschiedene Schnittstellen beim Erstellen haben, sind aber klar auf einen Wert zu führen. Zwei Value Objects desselben Typen mit gleichen Werten beschreiben dasselbe Objekt und sind daher identisch.

Es kann nur in validen Zuständen existieren, kann also keine invaliden Werte enthalten.

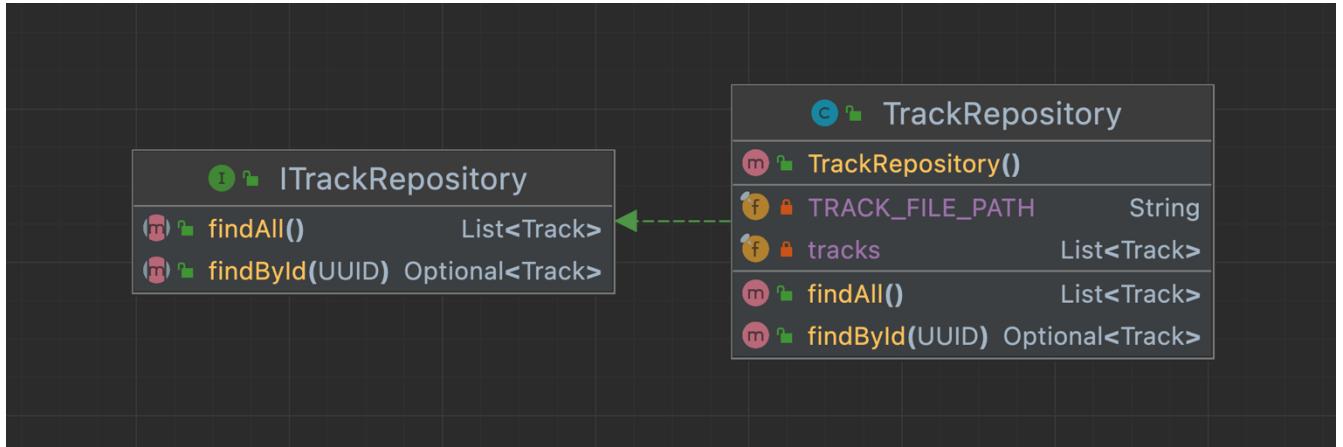


Um eine Anzahl an Schlägen zu beschreiben, wird das `StrokeCount` Objekt verwendet. Dieses validiert dabei, dass es sich beim Wert um eine positive Zahl größer 0 handelt, denn eine Schlaganzahl besteht immer aus mindestens einem Schlag.

Um die Eigenschaften eines Value Objects zu erfüllen, ist die Klasse „final“ und alle enthaltenen Werte „blank final“. Sie können nur bei Instanziierung des Objektes gesetzt werden. Dazu wurden die „equals“ und „hashcode“ Methode entsprechend den oben genannten Bedingungen angepasst.

## Repositories

Sie dienen der Verwaltung von Instanzen eines Objektes. So kann ihre Aufgabe das Speichern, sowie Abrufen von Ressourcen sein. Das Abrufen kann dabei auch durch spezielle Bezeichner und Filter geschehen oder Aggregate der zu verwaltenden Objekte, wie zum Beispiel die Anzahl aller Objekte, bilden. Durch die Implementation nah an der verwendeten Technologie, sind hier effizientere Abfragen möglich als in einer höheren Schicht.

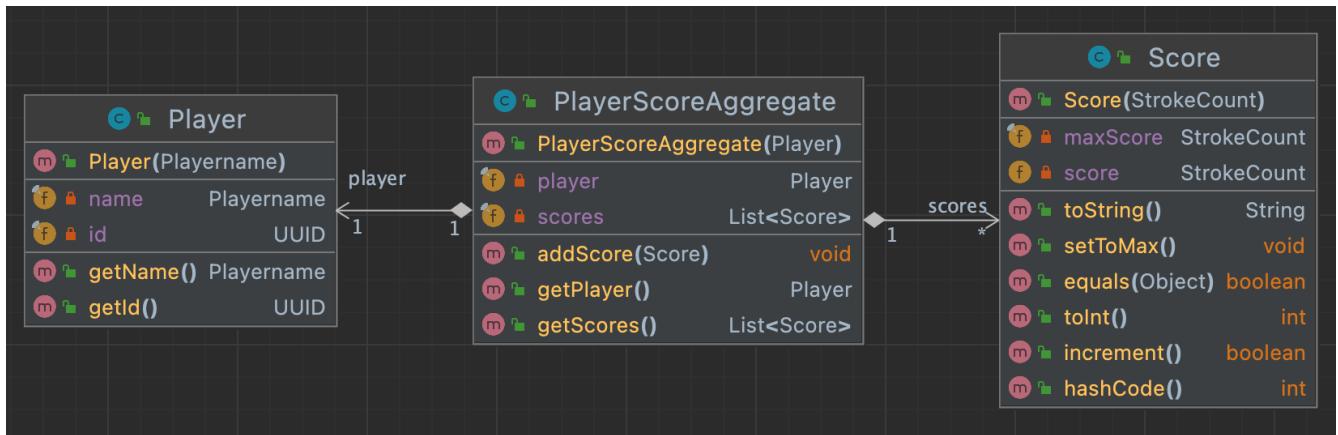


Das `TrackRepository` ermöglicht die zielgerichtete Abfrage der verfügbaren Spielbahnen. Für Auflistungen ist es möglich alle Routen abzufragen, es können aber auch einzelne über ihre ID angefragt werden. So muss nicht immer das gesamte Objekt übergeben oder hinterlegt werden, sondern es reicht die ID.

In der Implementation wird dann Gebrauch des CSV Readers gemacht, welcher die Inhalte der dazugehörigen Datei liest. Beim Repository handelt es sich um ein `ReadOnly`-Repository, bei dem sich die Inhalte der Datei während der Laufzeit nicht ändern. Daher können die Datei-Inhalte auch schon im Konstruktor geladen werden und nicht erst mit jedem Methodenaufruf.

## Aggregates

Ein Aggregate schließt mehrere Domänenobjekte in einem gebündelten Objekt zusammen. So kann es als eine Entität behandelt werden, was die Verwaltung deutlich vereinfacht. Innerhalb dieses Verbundes wird eines der Domänenobjekte als sogenannter Aggregate-Root verwendet. Dieser dient als Identität des gesamten Aggregates bei Zugriffen.



Das PlayerScoreAggregate verbindet einen Spieler (Player) mit einer Liste an ihm zugeordneten Punkteständen (Scores).

```
@Override  
public Optional<PlayerScoreAggregate> findByPlayer(Player player) {  
    return scores.retrieve().stream().filter(s → s.getPlayer().equals(player)).findAny();  
}
```

Der Spieler dient dabei als Root-Entity und dient zur Identifizierung des Aggregates.

## Kapitel 7: Refactoring

### Code Smells

#### Beispiel 1: Long Method

Während bei den meisten in der Anwendung verwendeten Menü Klassen die Methode zum Anzeigen des Inhaltes kurz ausfällt, ist dies beim PlayGameMenu nicht der Fall. Dieses ist die Hauptansicht während eines Spiels und zeigt sowohl das aktuelle Leaderboard, als auch eine Reihe an Informationen zum aktuellen Spielzug.

```
public void show() {
    app.getPlayerService().getAll().forEach(player -> {
        String scores = app.getScoreService().getAllForPlayer(player.getId()).stream()
            .map(Score::toString).collect(Collectors.joining(" "));
        int totalScore = app.getScoreService().getTotalForPlayer(player.getId());

        int nameColumnWidth = app.getPlayerService().getMaximumPlayernameLength();
        int scoreColumnWith = Math.max(1, app.getSession().getCurrentCourse().getTracks().size()*2-1);
        String format = "%-"+nameColumnWidth+"s | %-"+scoreColumnWith+"s | %s%n";
        System.out.format(format, player.getName(), scores, totalScore);
    });

    if (isGameFinished())
        return;
}

System.out.println("-----"); // divider
System.out.printf(
    "Playing: %s | Hole %s | Stroke %s / %s\n",
    app.getSession().getCurrentPlayer().getName(),
    app.getSession().getCurrentTrackIndex(),
    app.getSession().getCurrentScore(),
    app.getSession().getCurrentCourse().getMaxStrokes()
);
}
```

Diese eine lange Methode kann jedoch zu Zwecken der Übersicht und Lesbarkeit einfach in mehrere Submethoden aufgeteilt werden.

<https://github.com/Kronnox/ase-minigolf/commit/678f25285b175d33475936d51d29c728d02824ee>

```

public void show() {
    app.getPlayerService().getAll().forEach(this::printLeaderboardRow);
    if (!isGameFinished()) {
        printTurnInfo();
    }
}

private void printLeaderboardRow(Player player) {
    String scores = app.getScoreService().getAllForPlayer(player.getId()).stream()
        .map(Score::toString).collect(Collectors.joining(" "));
    int totalScore = app.getScoreService().getTotalForPlayer(player.getId());

    int nameColumnWidth = app.getPlayerService().getMaximumPlayernameLength();
    int scoreColumnWith = Math.max(1, app.getSession().getCurrentCourse().getTracks().size()*2-1);
    String format = "%" + nameColumnWidth + "s | %-" + scoreColumnWith + "s | %s%n";
    System.out.format(format, player.getName(), scores, totalScore);
}

private void printTurnInfo() {
    System.out.println("-----"); // divider
    System.out.printf(
        "Playing: %s | Hole %s | Stroke %s / %s\n",
        app.getSession().getCurrentPlayer().getName(),
        app.getSession().getCurrentTrackIndex(),
        app.getSession().getCurrentScore(),
        app.getSession().getCurrentCourse().getMaxStrokes()
    );
}

```

## Beispiel 2: Large Class

Aufbauend auf Beispiel 1, bekommt das Leaderboard noch einige weitere Funktionen, wie das Anzeigen der „Par“-Werte für alle Tracks und eine gleichmäßige Einrückung.

<https://github.com/Kronnox/ase-minigolf/commit/22d520c6c676991a76aedf87d841da37eb83a2ea>

Die dazugehörigen Methoden lauten wie folgt:

```

public void show() {
    printLeaderboardRow(app.getSession().getCurrentCourse());
    app.getPlayerService().getAll().forEach(this::printLeaderboardRow);
    if (!isGameFinished()) {
        printTurnInfo();
    }
}

private void printLeaderboardRow(Course course) {
    printLeaderboardRow(
        "(Par)",
        course.getTracks().stream().map(Track::getPar).map(StrokeCount::toInt).toList(),
        course.getTracks().stream().map(Track::getPar).mapToInt(StrokeCount::toInt).sum()
    );
}

private void printLeaderboardRow(Player player) {
    printLeaderboardRow(
        player.getName().toString(),
        app.getScoreService().getAllForPlayer(player.getId()).stream().map(Score::toInt).toList(),
        app.getScoreService().getTotalForPlayer(player.getId())
    );
}

private void printLeaderboardRow(String label, List<Integer> scores, int totalScore) {
    String scoreString = scores.stream().map(i → Integer.toString(i)).collect(Collectors.joining(" "));
    int nameColumnWidth = Math.max(5, app.getPlayerService().getMaximumPlayernameLength());
    int scoreColumnWith = Math.max(4, app.getSession().getCurrentCourse().getTracks().size()*2-1);
    String format = "%-"+nameColumnWidth+"s | %-"+scoreColumnWith+"s | %s%n";
    System.out.format(format, label, scoreString, totalScore);
}

private void printTurnInfo() {
    System.out.println("-----"); // divider
    System.out.printf(
        "Playing: %s | Hole %s | Stroke %s / %s\n",
        app.getSession().getCurrentPlayer().getName(),
        app.getSession().getCurrentTrackIndex(),
        app.getSession().getCurrentScore(),
        app.getSession().getCurrentCourse().getMaxStrokes()
    );
}

```

Wie in Commit [554a50ce935ed993a38d76910f7316e7af022d49](#) ersichtlich können jene Methoden aus der PlayGameMenu Klasse in eine separate Leaderboard Klasse ausgegliedert werden, welche dann einfach aus der PlayGameMenu Klasse aufgerufen werden kann. Konkret wurde dies durch ein instanziierbares Leaderboard Objekt mit öffentlicher print Methode umgesetzt.

```

public void show() {
    new Leaderboard(app).print();
    if (!isGameFinished()) {
        printTurnInfo();
    }
}

```

(Die vollständige gewählte Umsetzung ist im verlinkten Commit nachzuvollziehen, da ihre Länge den Rahmen des Dokumentes sprengen würde)

## 2 Refactorings

### Rename Method

<https://github.com/Kronnox/ase-minigolf/commit/0059b140ffafe56246b71eb6f779d134c490d113>

Auch geringe Änderungen zu Zwecken der Lesbarkeit sind wichtige Refactorings.

Leaderboard		
m	Leaderboard	(MinigolfApplication)
f	app	MinigolfApplication
f	format	String
m	printLeaderboardRow	(Course) void
m	print()	void
m	printLeaderboardRow	(Player) void
m	printLeaderboardRow	(String, List<Integer>) void
m	initializeFormat()	void

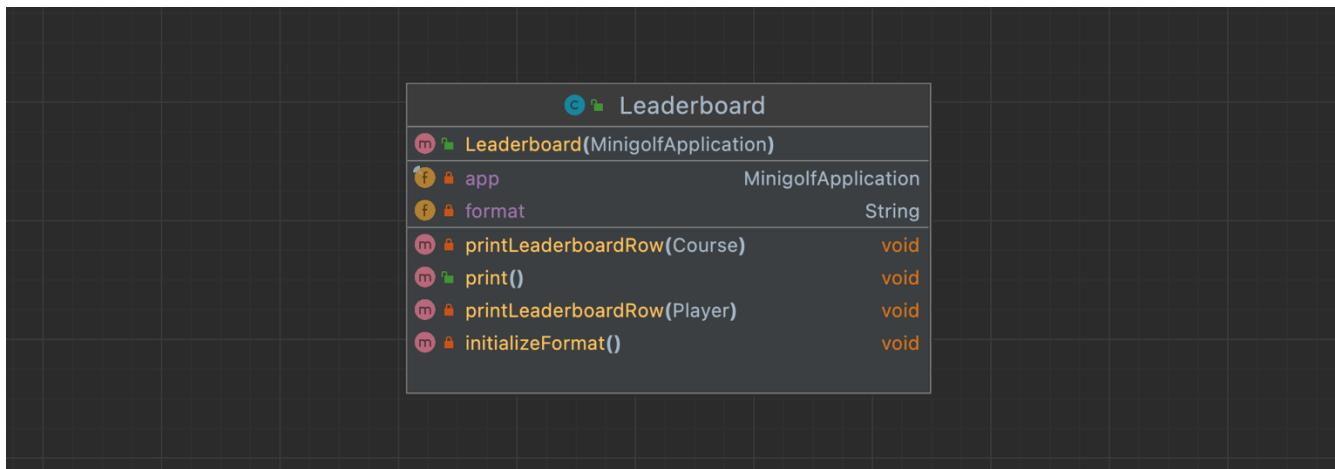
Durch das ergänzen des Wortes „for“ kann der Methodenname inklusive der Parameter flüssig gelesen werden.

Leaderboard		
m	Leaderboard	(MinigolfApplication)
f	app	MinigolfApplication
f	format	String
m	printLeaderboardRowFor	(Course) void
m	print()	void
m	printLeaderboardRowFor	(Player) void
m	printLeaderboardRow	(String, List<Integer>) void
m	initializeFormat()	void

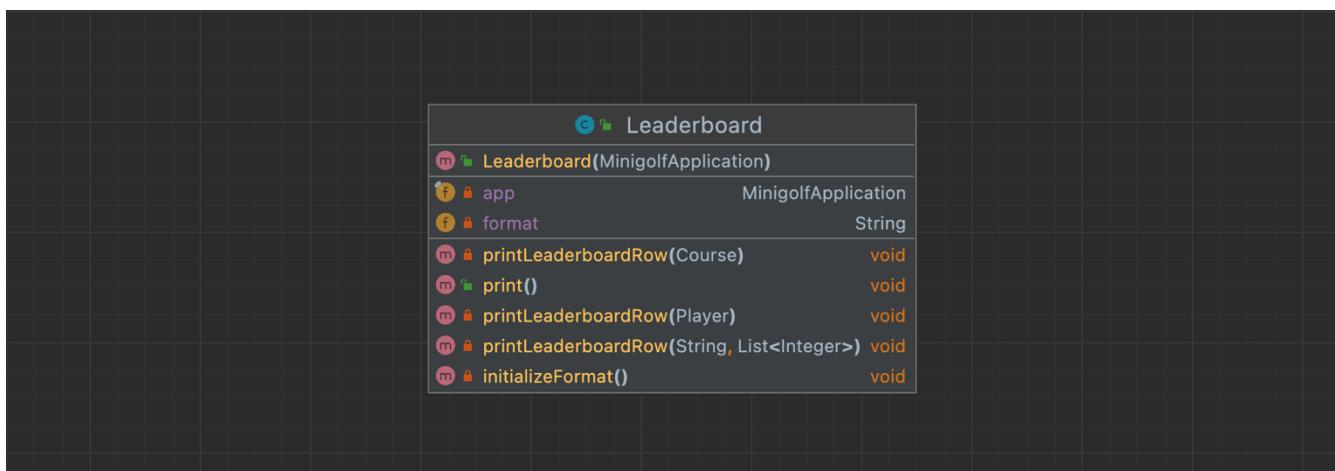
## Extract Method

<https://github.com/Kronnox/ase-minigolf/commit/e032908ed28f62b479cea35a13643c046b1ca4a3>

Das Auslangern von äquivalenten Code-Bausteinen in eine wiederverwendbare Methode reduziert die Gesamtanzahl der Zeilen und erhöht so die Übersichtlichkeit, reduziert aber auch die Gefahr von Inkonsistenzen bei Änderungen an einer der Einsatzpunkte.



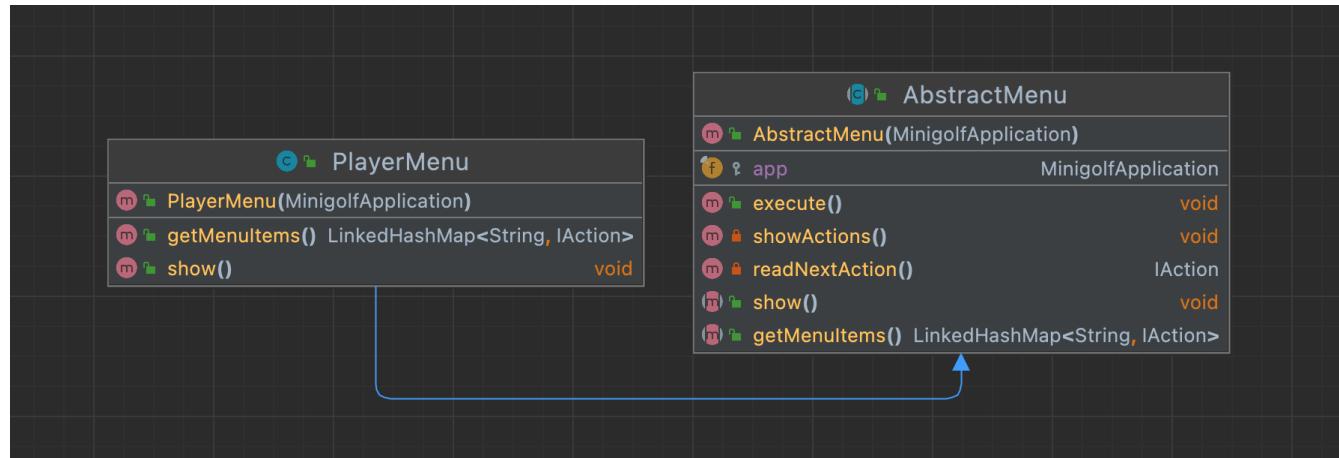
Auslagerung in eine dritte printLeaderboardRow Methode:



## Kapitel 8: Entwurfsmuster

### Entwurfsmuster: Prototyp

Für die Erstellung von interaktiven Menüs auf der Konsolenoberfläche wird das Prototyp Entwurfsmuster verwendet. Die abstrakte Klasse `AbstractMenu` bietet dabei Funktionalitäten, wie das Anzeigen eines einheitlichen Interaktionsmenüs inklusive der dazugehörigen Abfrage der Eingabe.

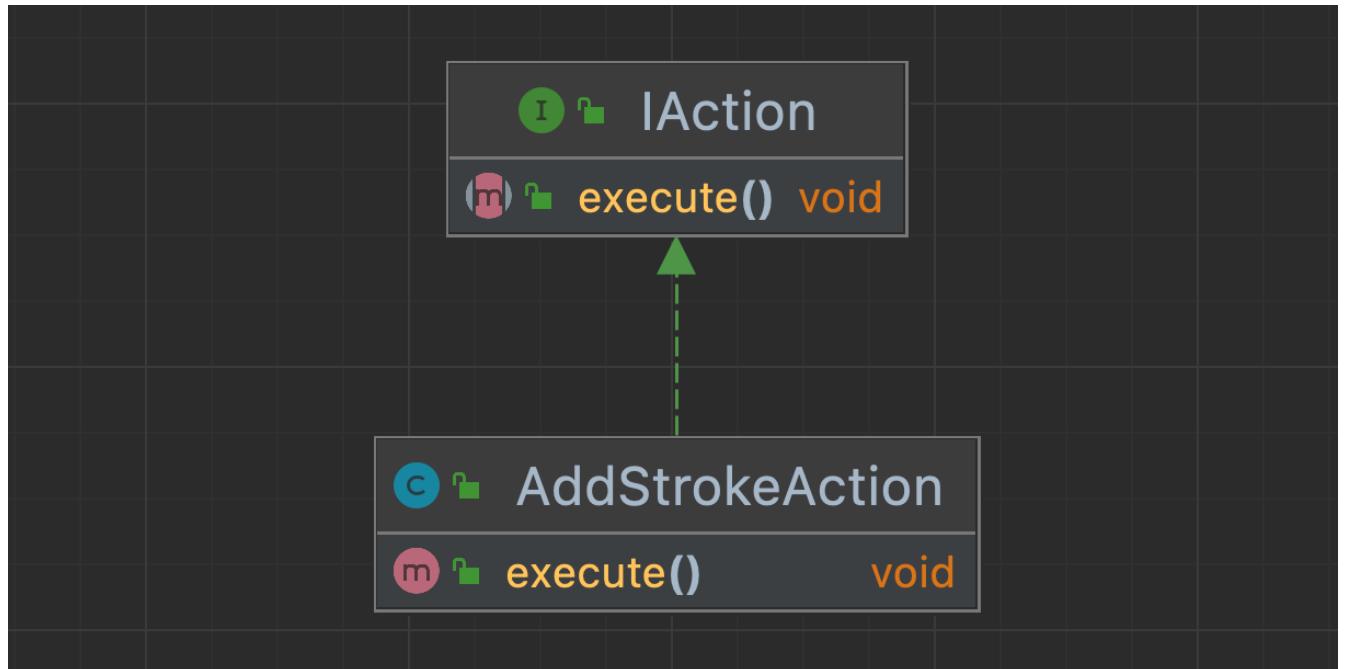


Die konkreten Implementationen einzelner Menüs, wie dem `PlayerMenu`, müssen dann nur noch die Methoden `show` und `getMenuItems` implementieren, welche bestimmen, was im jeweiligen Menü angezeigt werden soll und welche Aktionen zur Verfügung stehen und wie diese benannt sind.

Auf diese Weise kann eine Menge von immer wieder benötigtem Quelltext eingespart werden und die Klassen werden lesbarer und enthalten nur noch den für die jeweilige Aufgabe benötigten Code.

## Entwurfsmuster: Kommando

Um eine möglichst große Flexibilität bei den Interaktions-Optionen in den Menüs auf der Konsolenoberfläche zu bieten, wird das Entwurfsmuster Kommando verwendet. Eine Aktion nach dem Auswählen einer Option kann dabei sehr vielfältig sein. Beispielsweise kann es der Aufruf eines weiteren Menüs sein, die Verarbeitung eingegebener Daten oder die Ausführung eines beliebigen Code-Ausschnitts sein.



Die Kernklasse des Entwurfsmusters ist in diesem Fall `IAction`. Sie enthält eine `execute` Methode, welche in den konkreten Umsetzungen implementiert werden muss. Dadurch existiert eine einheitliche Schnittstelle für das Ausführen von Kommandos (in diesem Projekt Aktion bzw. Action genannt). Es ist also nicht relevant, welcher eigentliche Befehl sich dahinter verbirgt.