Programmentwurf
F1-Analyzer

Name: Helmrich, Ian
Martrikelnummer: 4891102

Abgabedatum: 02. Juni 2022

Kapitel 1: Einführung

Übersicht über die Applikation

Formel-1-Analysen werden nur durch AWS während der Rennen eingeblendet und sind nachträglich nicht mehr anschaubar. Damit auch danach Aussagen und Analysen über Rennen gegeben werden können, wurde der F1-Analyzer entwickelt. Der F1-Analyzer ist eine Konsolenanwendung, mit welchem Rennen ausgewählt werden können, zu welchem sich die Rennergebnisse, Fahrervergleiche und andere Informationen ausgegeben werden können.

Die Renndaten über CSV-Dateien bezogen werden, ist die Rennauswahl begrenzt. Als auswählbares Rennen kann bisher folgendes ausgewählt werden (Eingabe in Konsole):

Jahr: 2022

Rennen: 1

Wie startet man die Applikation?

Die Grundvoraussetzungen, um die Software zu starten, sind folgende:

1. Installation von Java 11
2. Repository von Github herunterladen
3. Jetbrains Intellij IDE herunterladen und über den Ordner 1-adapters die main.java kompilieren und starten

Wie testet man die Applikation?

Die Applikation wurde mit JUnit getestet. Wie die Tests gestartet werden können, wird folgend erläutert:

1. Voraussetzung: Java 11 wurde installiert und das Repository heruntergeladen
2. Vorhandensein von Maven
3. Projekt in Jetbrains Intellij IDE öffnen und die Tests manuell ausführen. Die Tests liegen im Ordner 2-Application/src/test.

Mir tut es leid, dass Sie durch mich einen Mehraufwand haben, aber ich habe leider keine .jars erzeugen können.

Kapitel 2: Clean Architecture

Was ist Clean Architecture?

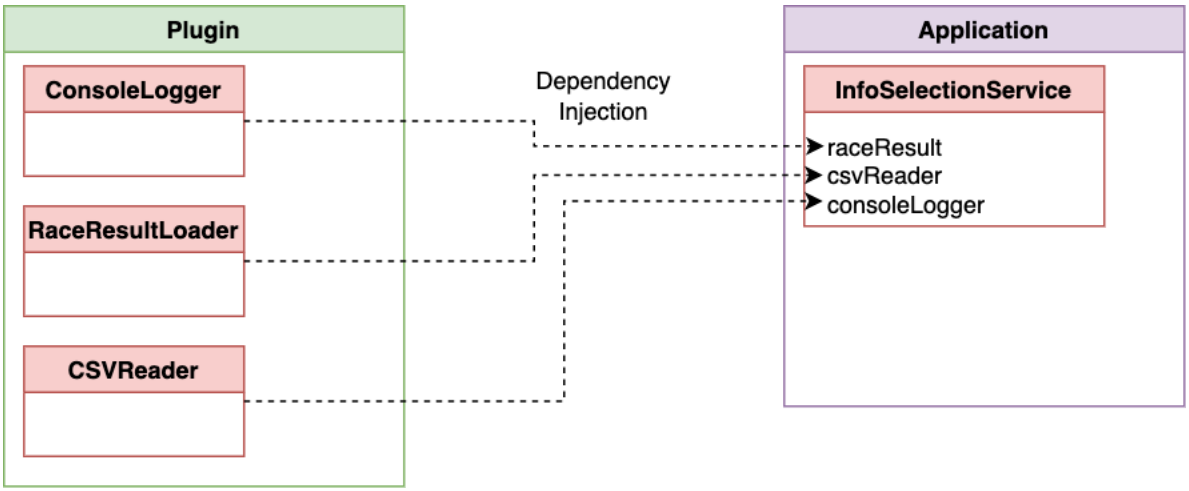
Unter Clean Architecture versteht man einen speziellen Architekturstil in der Softwareentwicklung. Die Software wird als eine Art Zwiebel in Schichten aufgebaut. Dabei kann die äußere Schicht immer nur

auf die innere zugreifen, wodurch eine bessere Entkopplung und Testfähigkeit der einzelnen Schichten entsteht.

Analyse der Dependency Rule

Im Folgenden wird ein positives sowie negatives Beispiel der Dependency Rule gezeigt.

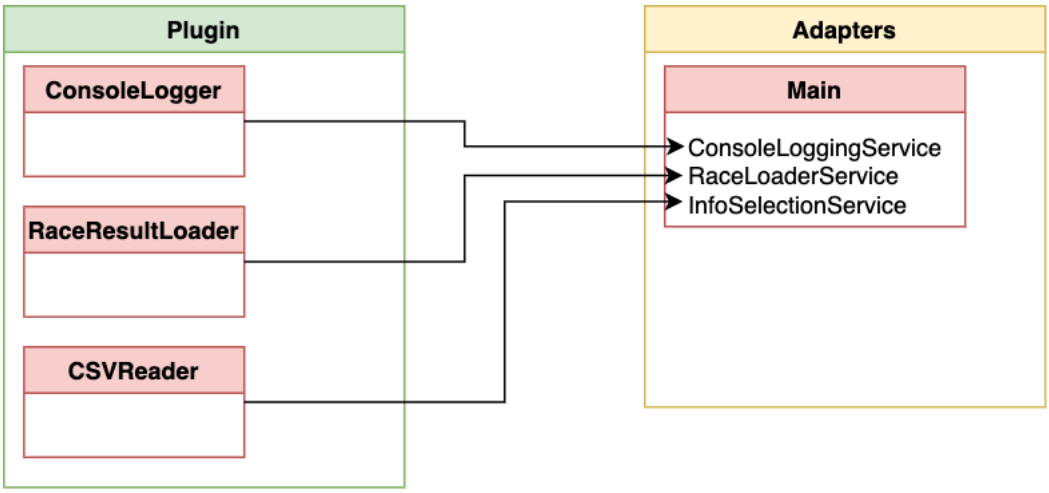
Positiv-Beispiel: Dependency Rule



Im obigen Beispiel wird aufgezeigt, wie die Dependency-Rule eingehalten wird. In der Application-Schicht befindet sich der InfoSelectionService, welcher in drei Variablen Objekte aus der Plugin-Schicht zugewiesen bekommt. Diese Objekte werden jedoch nicht im InfoSelectionService erzeugt, da dies die Dependency-Rule brechen würde. Aus diesem Grund erhält der InfoSelectionService die Objekte via Dependency-Injection von der Main-Klasse, wodurch die Dependency-Rule eingehalten wird.

Negativ-Beispiel: Dependency Rule

Im Folgenden Schaubild wird ein Beispiel zur Nichteinhaltung der Dependency Rule gezeigt. Die Main-Klasse befindet sich im Adapter. Aus der Main-Klasse werden die verschiedenen Services der Anwendung gestartet. In der Main-Klasse befinden sich drei Klassenvariablen, die zur Zuweisung von Werten auf die Plugin-Schicht zugreifen, um Objekte aus den ConsoleLogger-, RaceResultLoader- und CSVReader-Klassen zu generieren. Dadurch wurde die Dependency-Rule nicht mehr eingehalten, da von der Adapter-Schicht auf die Plugin-Schicht zugegriffen wurde.



Analyse der Schichten

Schicht: Application

Im oberen UML-Diagramm wird die Application-Schicht mit dem Beispiel von zwei Klassen gezeigt. In der Application-Schicht befindet sich die Business-Logik der Anwendung. Zu der Business-Logik gehört die Klasse FastestDriverSelectionService. Der Benutzer kann diesen Service auswählen, nachdem er angegeben hat, zu welchem Rennen er den Service ausführen möchte. Der Service gibt dem Benutzer den Rennfahrer mit der schnellsten Rennrunde, sowie seiner Zeit für diese Runde aus. Der FastestDriverSelectionService hängt mit der InfoSelectionService-Klasse zusammen, da die Klasse dort initialisiert wird, denn der InfoSelectionService bietet dem Benutzer die Chance auszuwählen, welchen Service er nutzen will. Entsprechend der Auswahl startet der InfoSelectionService den entsprechenden anderen Service.

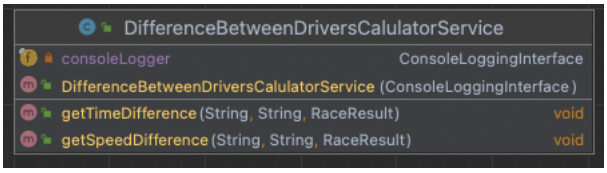
Schicht: Plugin

Unter der Plugin-Schicht versteht man die äußerste Schicht im Clean-Architecture Modell. Die Schicht besteht aus Plugins, wie zum Beispiel die Anbindung an eine Datenbank oder die Funktion einer Konsolenausgabe. In der F1-Analyzer-Software werden die Ergebnisse der Rennen und Analysen in der Konsole ausgegeben, weshalb auch eine Konsolenausgabe gebaut werden musste. Diese Konsolenausgabe befindet sich im Repository in der Plugin-Schicht. Die Services loggen ihre Ergebnisse über dieses Plugin in die Konsole.

Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel

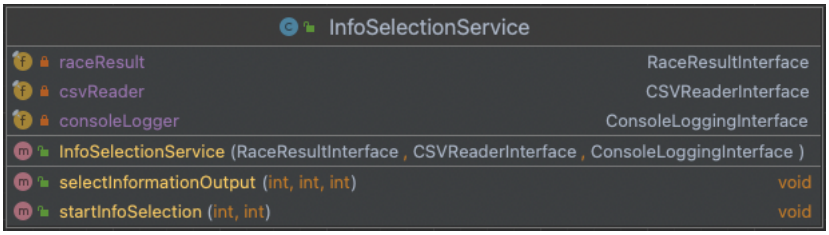


Im positiven Beispiel geht es um die DifferenceBetweenDriversCalculatorService-Klasse. Die Aufgabe der Klasse ist es, Methoden zur Berechnung von Unterschieden zwischen zwei Fahrern bereitzustellen.

Die Klasse hat keine andere Aufgabe, weshalb das Single-Responsibility-Principle erfüllt ist.

Negativ-Beispiel

Als Negativ-Beispiel dient der InfoSelectionService. Die Aufgabe des Services ist es, dem Nutzer die Auswahl zwischen den verschiedenen Services wie die Ausgabe des Rennergebnisses zu geben. Das Problem hierbei ist, dass in der selectInformationOutput-Methode das Rennergebnis aus der csv-Datei gelesen wird, um es anderen Services direkt mitzugeben. Dadurch hat die Klasse nicht nur eine Aufgabe. Eine Lösung des Problems wäre es, das Lesen der CSV-Datei in den jeweiligen Services zu machen, wo das Rennergebnis wirklich gebraucht wird.



Analyse Open-Closed-Principle (OCP)

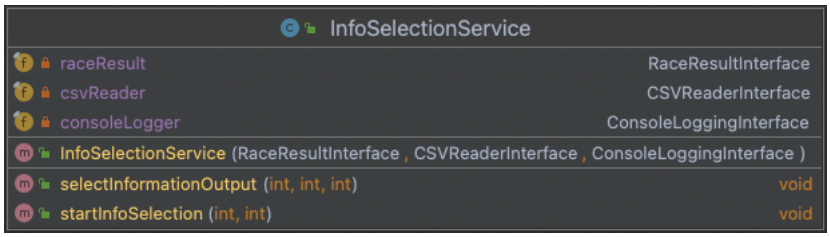
Positiv-Beispiel

Durch die Einführung von Interfaces wird eine leichtere Erweiterung gewährleistet. Außerdem ist eine Anpassung in allen Klassen, die das Interface implementieren, ziemlich umständlich, da sonst auch das Interface geändert werden muss.

Negativ-Beispiel

Als Negativ-Beispiel gilt es die InfoSelectionService-Klasse zu nennen. Wie schon genannt ist es die Aufgabe der Klasse, dem Benutzer die verschiedenen Services zur Auswahl anzubieten. Kommen nun neue Services hinzu, muss in den Quellcode eingegriffen werden, da die Auswahl in der Methode

selectInformationOutput über ein Switch-Case Statement geschieht. Dadurch kann es passieren, dass bei der Einführung weiterer Services diese Methode jedes mal auf neue angepasst werden muss. Dieses Problem lässt sich lösen, indem das switch-case Statement mithilfe der Aufteilung auf weitere Klassen und der Einführung eines Interfaces beseitigt wird.



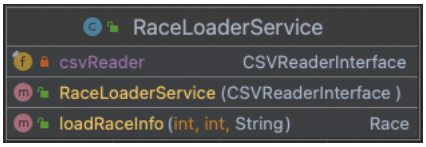
Analyse Dependency-Inversion-Principle (DIP)

[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP); jeweils UML der Klasse und Begründung, warum man hier das Prinzip erfüllt/nicht erfüllt wird]

[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

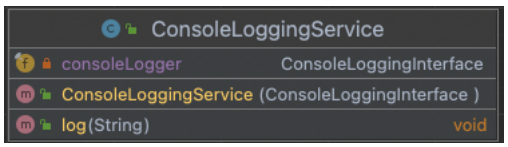
Positiv-Beispiel

Als positives Beispiel wird die Klasse RaceLoaderService aus der Application-Schicht verwendet. Die Klasse benötigt den CSVReader aus der Plugin-Schicht. Mittels Dependency-Inversion wird der CSVReader über den Konstruktor in die Klasse eingeführt. Aus dem UML-Diagram geht diese Technik nicht sonderlich hervor, weshalb die Klasse oben verlinkt ist.



Weiteres Positiv-Beispiel

Das Prinzip der Dependency-Inversion wurde nicht nur in der obigen Klasse, sondern ebenfalls in der ConsoleLoggingService-Klasse verwendet. Das dazugehörige UML-Diagram befindet sich in folgender Abbildung:



Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

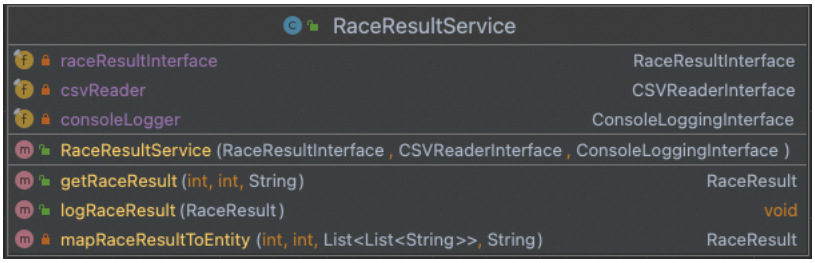
Positiv-Beispiel

Ein positives Beispiel ist in der CSVReader-Klasse zu erkennen. Die Klasse nur an das Interface gekoppelt. Dadurch ist die Klasse gut wiederverwendbar und kann schnell angepasst werden. Die

Klasse hat nur die Aufgabe eine Datei auszulesen und in Form einer zweidimensionalen Liste zurückzugeben.

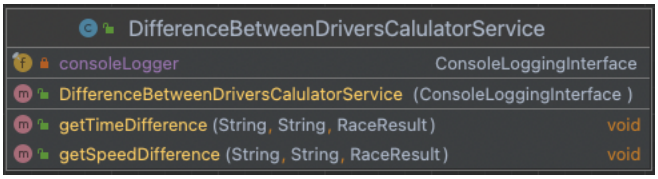
Negativ-Beispiel

Ein Negativ-Beispiel ist in der Klasse RaceResultService zu finden. Die Klasse hat Abhängigkeiten zu drei verschiedenen Plugins. Erstens zum RaceResultLoader, zweitens zum CSVReader und drittens zum ConsoleLogger. Wird an einem der drei Plugins eine Anpassung gemacht, muss diese wahrscheinlich auch in der RaceResultService-Klasse gemacht werden. Die RaceResultService-Klasse hat die Aufgabe, die Rennergebnisse (in die dazugehörigen Objekte) zu laden und auszugeben.



Analyse GRASP: Hohe Kohäsion

Die Klasse DifferenceBetweenDriversCalculatorService bietet eine hohe Kohäsion. Die Klasse ist verantwortlich für die Berechnung von Unterschieden zwischen zwei Fahrern im Rennen. Dies können zum Beispiel Zeitabstände oder Geschwindigkeitsunterschiede sein. Die hohe Kohäsion wird eingehalten, da die Klasse mit den zwei Methoden überschaubar organisiert ist und thematisch zusammengehören.



Don’t Repeat Yourself (DRY)

In der Klasse DifferenceBetweenDriversCalculatorService wird für zwei Methoden nach den angebenen Fahrern aus der Result-Liste gesucht. Dieser Code ist in beiden Methoden identisch und kann in eine neue Methode ausgelagert werden. Im Commit delete repeating code ist ersichtlich, welcher Code gelöscht werden konnte. Durch die Anwendung dieses Prinzips konnten aus beiden Methoden 7 Zeilen Code gelöscht werden.

Kapitel 5: Unit Tests

10 Unit Tests

Unit Test	Beschreibung
ConsoleLoggingService#log	Es wurde getestet, ob die Methode die Eingaben in die Konsole ausgibt.
FastestDriverSelectionService#GetFastestDriver	Es wurde getestet, ob die Methode den schnellsten Fahrer ausgibt.
FastestDriverSelectionService#logResult	Es wurde getestet, ob die Methode den schnellsten Fahrer mit seiner Zeit richtig in die Konsole ausgegeben wird.
DifferenceBetweenDriversCalculator#getSpeedDifference	Es wurde getestet, ob die Differenz der Geschwindigkeit zwischen zwei Fahrern richtig berechnet wurde.
DifferenceBetweenDriversCalculator#getTimeDifference	Es wurde getestet, ob die Differenz der Zeit zwischen zwei Fahrern richtig berechnet wurde.
RaceLoaderService#loadRaceInfo	Es wurde getestet, ob das Rennen mit dem aus der CSV geladenen übereinstimmt

ATRIP: Automatic

Die Tests können durch die IDE automatisch ausgeführt werden. Dafür wird ein Test selektiert, kompiliert und ausgeführt. Außerdem können die Tests nur mit bestanden oder fehlgeschlagen enden. Die wird durch JUnit und die Verwendung von Assert gewährleistet.

ATRIP: Thorough

Positives Beispiel: Es wird in der Testmethode testConsoleOutput genau getestet, ob der consoleLoggingService die gewünschte Ausgabe in der Konsole macht. Dabei wurde auch genauestens darauf geachtet, dass die Ausgabe auf die Konsole nicht verfälscht wird und mithilfe eines ByteArrayOutputStreams ausgelesen.

Negatives Beispiel:

ATRIP: Professional

Positives Beispiel: Der Test generiert zu Beginn die später in den Tests benötigten Rennergebnisse. Das ist wichtig und professionell, da man sich an bereits bestehenden Hilfsfunktionen und Klassen bedient.

Negatives Beispiel: Unter dem selben obigen Link kann der Test testLogResult gefunden werden. Dieser Test ist unnötig, da am Ende nur das Konsolenergebnis getestet wird. Dies kann hilfreich sein, wird aber in diesem Fall als unnötiger Test betrachtet.

Code Coverage

Die Code Coverage wurde über das Tool Sonarcloud analysiert. Unter folgendem Link können genauere Informationen entnommen werden. Es gibt insgesamt 16 Findings im Projekt und die Coverage liegt bei 0%, da SonarCloud nicht richtig eingerichtet ist.

Fakes und Mocks

Es wurden zwei Mock-CSV-Dateien eingeführt, mit welchen in den Tests gearbeitet wird. Dadurch können sich die anderen CSV-Dateien verändern, aber die CSV-Dateien für die Tests bleiben gleich, wodurch die Tests unabhängig von den produktiven Dateien gestartet werden können.

Kapitel 6: Domain Driven Design

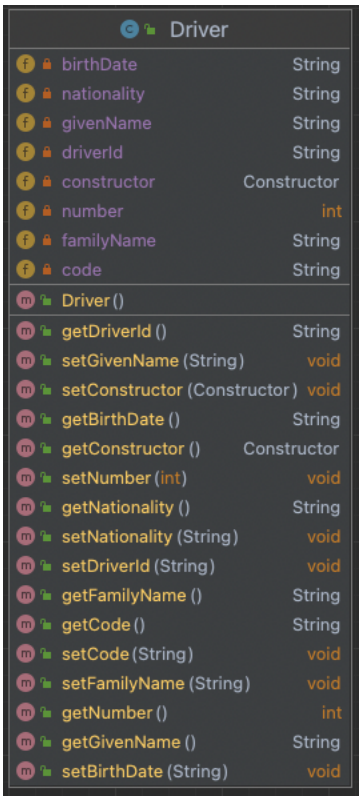
Ubiquitous Language

[4 Beispiele für die Ubiquitous Language; jeweils Bezeichung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]

Bezeichung	Bedeutung	Begründung
RaceResult	Übergeordneter Begriff, welcher Rennergebnis und Information über Rennen enthält	Sinnvoll, damit klar wird, dass es sich hier nicht nur um das Ergebnis handelt
Result	Hierbei handelt es sich nur um das Ergebnis eines einzelnen Fahrers	-
Constructor	Fahrer-Konstruktor wie Ferrari oder Mercedes	Hierbei handelt es sich nicht um den constructor in der Informatik, sondern um die „Marke“ des Autos
Driver	Fahrer des Autos im Konstrukteursteam	Ist kein Treiber auf deutsch, sondern der Fahrer. Ergibt im Kontext der Software Sinn.

Entities

Folgend ist die Fahrer-Entität zu erkennen. Die wird eingesetzt, damit Informationen zu dem Fahrer abgespeichert werden, die interessant für den Endbenutzer sind.



Value Objects

Es hätte ein Value Object für die Geschwindigkeit oder Zeit eingeführt werden können. Da diese Attribute aber nicht für Berechnungen verwendet werden, sondern nur als String auf der Konsole ausgegeben werden, wurde sich der Aufwand für die Implementierung erspart.

Repositories

Es hätte ein Repository für den RaceResult eingeführt werden können, wodurch vor jedem CSV-Read des Rennergebnis geschaut wird, ob das Ergebnis bereits eingelesen wurde. Aus Zeitgründen wurde auf die Implementierung verzichtet. Wäre das Repository implementiert worden, hätte die Performance der Software verbessert werden können, da nicht jedes mal ein Read auf eine Datei, sowie das Mapping auf die Entitäten erfolgt wäre.

Aggregates

Es wurden keine Aggregates eingeführt, da die Software dadurch eher eingeschränkt wird. Dadurch, dass zum Beispiel der Driver (Fahrer) vom Result (Ergebnis) entkoppelt ist, kann schnell eine Funktion eingebaut werden, in der der Benutzer seine Lieblingsfahrer speichern kann.

Kapitel 7: Refactoring

Code Smells

Duplicated Code: In der Klasse DifferenceBetweenDriversCalculatorService wird für zwei Methoden nach den angebenen Fahrern aus der Result-Liste gesucht. Dieser Code ist in beiden Methoden identisch und kann in eine neue Methode ausgelagert werden. Im Commit delete repeating code ist ersichtlich, welcher Code gelöscht werden konnte. Durch die Anwendung dieses Prinzips konnten aus beiden Methoden 7 Zeilen Code gelöscht werden.

Vorher:

```
public void getTimeDifference(String firstDriverCode, String secondDriverCode, RaceResult raceResult) {
    List<Result> results = raceResult.getResultList();
    Result firstResult = null;
    Result secondResult = null;
    for (int i = 0; i < results.size(); i++) {
        String driverCode = results.get(i).getDriver().getCode();
        if (Objects.equals(driverCode, firstDriverCode)) {
            firstResult = results.get(i);
        } else if (Objects.equals(driverCode, secondDriverCode)) {
            secondResult = results.get(i);
        }
    }
    if (firstResult == null) consoleLogger.log("Driver Code of first driver is wrong");
    if (secondResult == null) consoleLogger.log("Driver Code of second driver is wrong");
    else {
        int timeDifferenceInMillis = firstResult.getRaceTimeInMilli() - secondResult.getRaceTimeInMilli();
        if (timeDifferenceInMillis < 0) timeDifferenceInMillis *=-1;
        consoleLogger.log("Die Zeitdifferenz der beiden Fahrer liegt bei " +
                TimeUnit.MILLISECONDS.toSeconds(timeDifferenceInMillis) + " Sekunden");
    }
}

public void getSpeedDifference(String firstDriverCode, String secondDriverCode, RaceResult raceResult) {
    List<Result> results = raceResult.getResultList();
    Result firstResult = null;
    Result secondResult = null;
    for (int i = 0; i < results.size(); i++) {
        String driverCode = results.get(i).getDriver().getCode();
        if (Objects.equals(driverCode, firstDriverCode)) {
            firstResult = results.get(i);
        } else if (Objects.equals(driverCode, secondDriverCode)) {
            secondResult = results.get(i);
        }
    }
    if (firstResult == null) consoleLogger.log("Driver Code of first driver is wrong");
    if (secondResult == null) consoleLogger.log("Driver Code of second driver is wrong");
    else {
        double speedDifference = firstResult.getFastestLapAvgSpeed() - secondResult.getFastestLapAvgSpeed();
        if (speedDifference < 0) speedDifference *=-1;
        speedDifference = Math.floor(speedDifference * 100) / 100;
        consoleLogger.log("Die Geschwindigkeitsdifferenz der beiden Fahrer liegt bei " +
                speedDifference + "Km/h");
    }
}
```

Nachher:

```
public void getTimeDifference(String firstDriverCode, String secondDriverCode, RaceResult raceResult) {
    List<Result> driverResults = getDriverCodes(firstDriverCode, secondDriverCode, raceResult);
    firstResult = driverResults.get(0);
    secondResult = driverResults.get(1);
    if (firstResult == null) consoleLogger.log("Driver Code of first driver is wrong");
    if (secondResult == null) consoleLogger.log("Driver Code of second driver is wrong");
    else {
        int timeDifferenceInMillis = firstResult.getRaceTimeInMilli() - secondResult.getRaceTimeInMilli();
        if (timeDifferenceInMillis < 0) timeDifferenceInMillis *=-1;
        consoleLogger.log("Die Zeitdifferenz der beiden Fahrer liegt bei " +
                TimeUnit.MILLISECONDS.toSeconds(timeDifferenceInMillis) + " Sekunden");
    }
}

public void getSpeedDifference(String firstDriverCode, String secondDriverCode, RaceResult raceResult) {
    List<Result> driverResults = getDriverCodes(firstDriverCode, secondDriverCode, raceResult);
    firstResult = driverResults.get(0);
    secondResult = driverResults.get(1);
    if (firstResult == null) consoleLogger.log("Driver Code of first driver is wrong");
    if (secondResult == null) consoleLogger.log("Driver Code of second driver is wrong");
    else {
        double speedDifference = firstResult.getFastestLapAvgSpeed() - secondResult.getFastestLapAvgSpeed();
        if (speedDifference < 0) speedDifference *=-1;
        speedDifference = Math.floor(speedDifference * 100) / 100;
        consoleLogger.log("Die Geschwindigkeitsdifferenz der beiden Fahrer liegt bei " +
                speedDifference + "Km/h");
    }
}
```

2 Refactorings

Erstes Refactoring: Rename Method

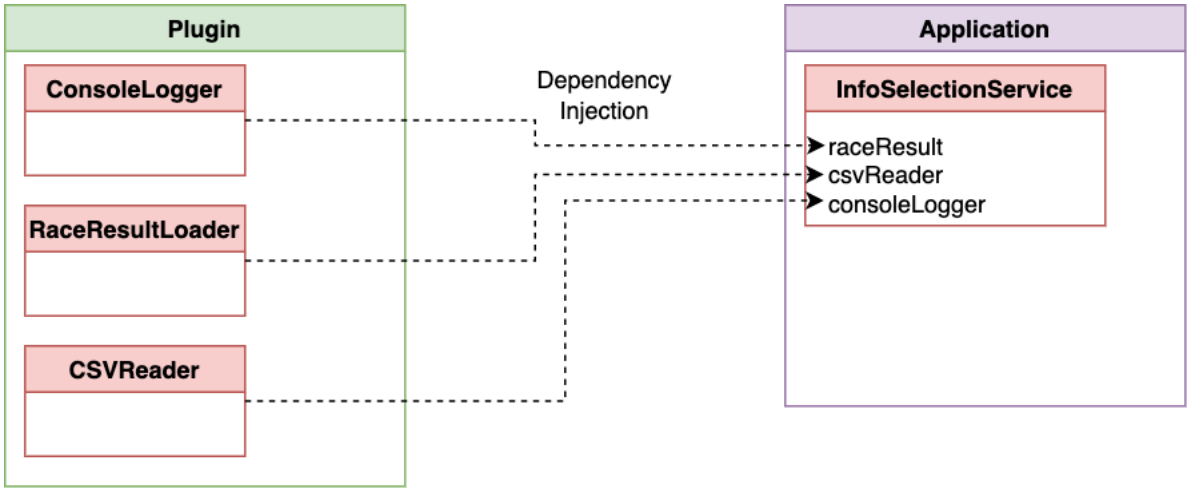
Es wurde die private Methode mapRaceResultToEntity aus der Klasse RaceResultService zu mapCSVToRaceResultObject umbenannt. Diese Umbenennung macht klarer, was die Methode macht und hilft anderen Entwicklern dabei, diese besser zu verstehen.

Zweites Refactoring: Delete unnecessary imports

Im obigen Commit werden unnötige Imports gelöscht, die für Verwirrung sorgen können oder auf legacy-Code hinweisen können.

Kapitel 8: Entwurfsmuster

Entwurfsmuster: Dependency Injection



Im obigen Schaubild ist der Einsatz von Dependency Injection ablesbar. Dadurch konnte gewährleistet werden, dass keine Abhängigkeiten zwischen verschiedenen Schichten entstehen und der Code widerverwendbarer wird.

Entwurfsmuster: Builder Pattern

Das Builder Pattern sollte in die Driver-Entität eingebaut werden, um eine bessere Erweiterbarkeit der Software in der Zukunft zu ermöglichen. Da allerdings momentan kein Use-Case für dieses Pattern gebraucht wird, wurde es nicht implementiert. Außerdem haben sich keine anderen in der Vorlesung behandelten Patterns auf das Softwaredesign angeboten, weshalb nur die Dependency Injection umgesetzt wurde.