Advanced Software Engineering

Projektarbeitsdokumentation

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges

Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Philipp Reichert

Abgabedatum:	XX. Mai 2023
Bearbeitungszeitraum:	01.10.2022 - XX.05.2023
Matrikelnummer, Kurs:	1758822, TINF20B2
Gutachter der Dualen Hochschule:	Dr. Lars Briem

Contents

Kapitel 1:Einführung

Übersicht über die Applikation

Bei der Applikation handelt es sich um ein Programm zur Auswertung von GPS Exchange Format (GPX) Dateien.
Allgemein wird Unterschieden zwischen geplanten Strecken (Track) und bereits bestrittenen Touren welche Zeitstempel an allen Koordinaten haben (Tour). Für beide kann die Höhendifferenz und die Strecke berechnet werden. Außerdem kann ein Höhenprofil generiert und in der Konsole Angezeigt werden.
Mit Hilfe von Bewegungsgeschwindigkeiten kann die voraussichtliche Dauer einer Begehung eines Tracks vorhergesagt werden. Dabei kann die Bewegungsgeschwindigkeit entweder aus bereits begangen Touren kalkuliert werden oder einer Auswahl an Sportarten gewählt werden (Wandern, Radfahren, ...). Um die eigene Geschwindigkeit herauszufinden kann man sich die aus Touren gewonnene Bewegungsgeschwindigkeit auch isoliert anzeigen lassen oder ein Geschwindigkeit-Zeit-Diagramm der Tour anzeigen lassen.
Bei langen Touren ist es häufig nötig, Umwege zur Übernachtung oder zum Auffüllen von Vorräten nach einer gewissen Zeit einzulegen. Da die Lösung des Problems nicht trivial ist (nach einstündiger, ergebnisloser Überlegung wurde auf die Erstellung einer Reduzierung auf das Knapsack-Problem verzichtet, da es über den Umfang des Projekts hinausgeht) wurde ein Evolutionärer Hillclimb-Algorithmus zur möglichst optimalen Wahl der Stützpunkte gewählt. Hierfür muss eine Tour oder ein Track anhand einer Auswahl an Wegpunkten (etwa Hütten oder Supermärkte), einer Bewegungsgeschwindigkeit und einer Dauer, die man ohne die Ressource auskommt angegeben werden, um die zu besuchenden Wegpunkte zu erhalten.

Wie startet man die Applikation?

Benötigt wird eine IDE die Java 19 ausführen kann.

Zum Starten der Applikation muss die Main-Methode ausgeführt werden. Diese liegt im Pfad *src/GPXrechner/Main.java*.

Erste Schritte

Nun läuft das Command Line Interface der Applikation und man wird aufgefordert, eine Instruktion einzugeben. Gibt man eine nicht zulässigen Befehl oder *help* ein, so bekommt man eine Übersicht über alle möglichen Instruktionen.
Um etwas über mit einer Tour oder einem Track zu tun, muss sie geladen werden. Dies funktioniert mit dem Befehl *load gpx*.

Da alle GPX-Dateien in dem dafür vorgesehenen Ordner abgelegt sind, muss der Pfad zu ihnen relativ zu diesem Ordner angegeben werden. Beispielhaft zum Laden eines Tracks der die berühmte Watzmann Überschreitung beinhaltet wäre der Pfad *Track/Watzmann.gpx*.

Wie testet man die Applikation?

Die Tests befinden sich unter *src/test/*.
Zum Testen der Applikation führt man diese mithilfe der IDE aus, in IntelliJ mit rechtem Mausklick auf das Directory und Auswahl von 'Run 'Tests in 'test".

Kapitel 2: Clean Architecture

Was ist Clean Architecture?

Clean Architecture ist ein Softwarearchitekturmuster welches darauf abzielt Code klar zu organisieren und leicht wartbar, testbar und erweiterbar zu machen. Hierfür werden die Bestandteile einer Anwendung in verschiedenen hierarchische Schichten so gekapselt, dass außen liegende Schichten von inneren abhängen könne, innere aber nicht von äußeren. Tiefere Schichten sind langlebiger als außenliegende.

Analyse der Dependency Rule

Positiv-Beispiel: Dependency Rule

Das Positivbeispiel ist die Klasse GetDistance, die eine Implementation einer Instruction ist. Sie liegt in der Anwendungsschicht. Um den Weg zu erhalten, für den die Strecke berechnet werden soll greift sie auf den Anwendungszustand auf der Anwendungsschicht zu. Die Strecke wird über das UserOutput Interface mitgeteilt, welches auch in der Anwendungsschicht liegt. Die eigentliche Berechnung findet mithilfe der Klasse DistanceCalculator statt, welche in der Domänenschicht liegt auf die im Sinne der clean architecture Abhängigkeiten bestehen dürfen.

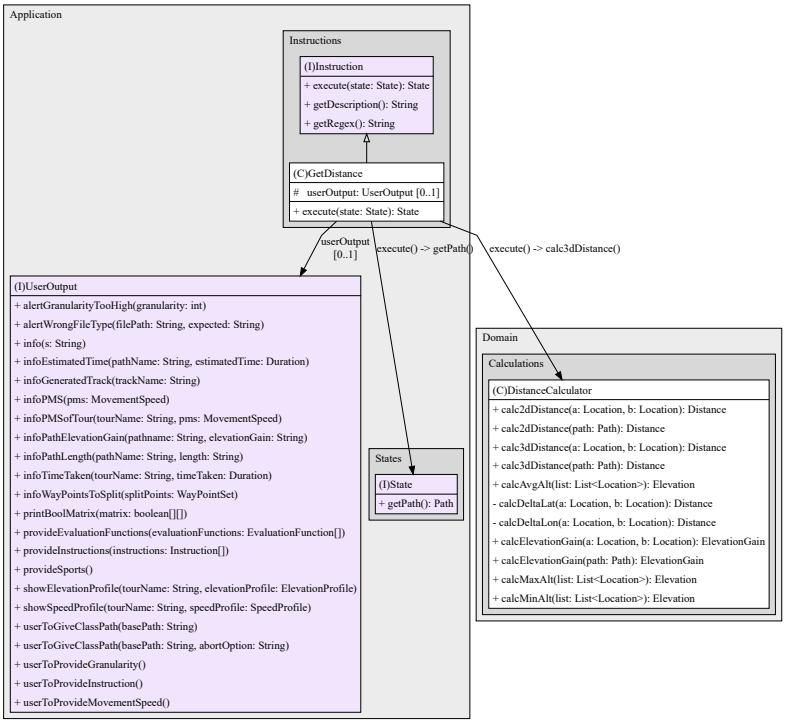

Figure 1: Abhängigkeiten der Klasse GetDistance

Negativ-Beispiel 2: Dependency Rule

Die Dependency Rule wird beim Zugriff auf die Klasse GPXToTour verletzt, die in der Plugin Schicht liegt und aus GPX Dateien ein Tour Objekt generiert. Sie wird von der Instruction ReadPath verwendet, welche in der Anwendungsschicht liegt. Somit besteht eine Abhängigkeit von der inneren Anwendungsschicht zur ausserhalb liegenden Plugin Schicht, was eine Verletzung der Dependency Rule darstellt.

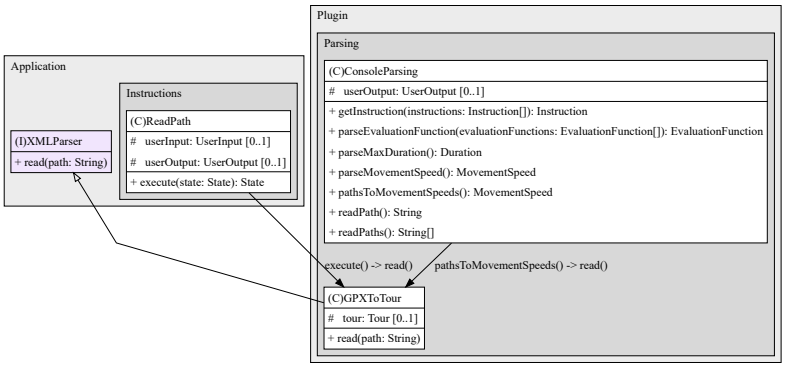

Figure 2: Abhängigkeiten auf die Klasse GPXToTour

Schicht: Domain Code

Die Klasse DistanceCalculator ist dafür Zuständig verschiedene Distanzen zwischen Orten oder einer chronologischen Abfolgen von Orten im Sexalsystem zu berechnen. Die (angemessen genaue) Berechnung von Distanzen im Sexalsystem basieren auf grundlegenden geometrischen Zusammenhängen, welche sich in absehbarer Zeit nicht ändern. Diese Berechnungen Grundlegend für alle Auswertungen von Daten die im Sexalsystem gespeichert sind, so wie beispielsweise GPS Daten im GPS Exchange format(GPX).

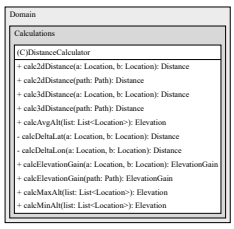

Figure 3: UML Diagramm der Klasse DistanceCalculator

Schicht: Plugins

Die Klasse ConsoleParsing ist dafür Zuständig verschiedene Formen Input von Benutzern zu erfassen. Dies umfasst den Pfad zu GPX Dateien, die Wahl einer Sportart oder Geschwindigkeit oder die Eingabe einer Zeit. Somit stellt die Klasse einen wesentlichen Bestandteil der Benutzerschnittstelle dar.

Ein Austausch der Klasse durch eine Grafische Benutzerschnittstelle wäre denkbar.

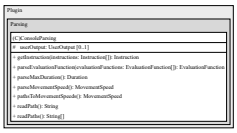

Figure 4: UML Diagramm der Klasse ConsoleParsing

Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel

Die Klasse Latitude und repräsentiert eine Breite im Sexalsystem, also etwa 49.00 für Karlsruhe. Die Aufgabe ist die Überprüfung ob die Breite im Erlaubten Wertebereich ist und die Ausgabe als Double, was in eigenen Methoden umgesetzt ist.

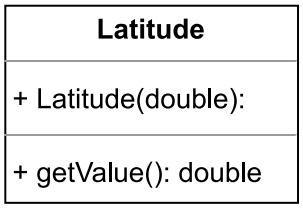

Figure 5: UML Diagramm der Klasse Latitude

Negativ-Beispiel

Die Klasse SpeedCalculator ist eine Klasse die statische Methoden zu Berechnungen mit Geschwindigkeiten umsetzt. Konkrete Aufgaben sind die Berechnung von Geschwindigkeitskomponenten aus einer oder mehreren Touren sowie die in Verhältnis Stellung von der Geschwindigkeit einer Tour mit einer Liste von Tourpunkten.

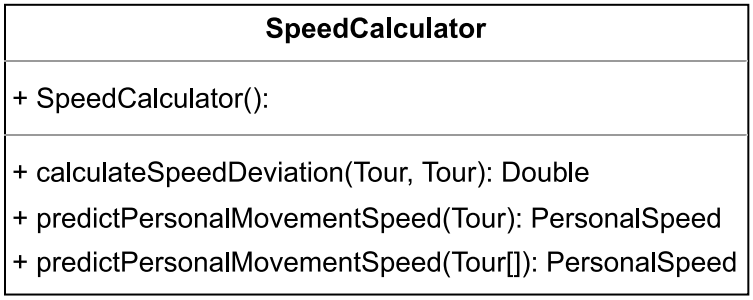

Figure 6: UML Diagramm der Klasse SpeedCalculator

Hier könnte das SRP (zumindest auf Klassenebene) umgesetzt werden indem die Berechnung der Geschwindigkeitsabweichung eine eingene Klasse

bekommt die von der neuen Klasse, welche lediglich die Aufgabe hat Geschwindigkeitskomponenten zu berechnen abhängt.

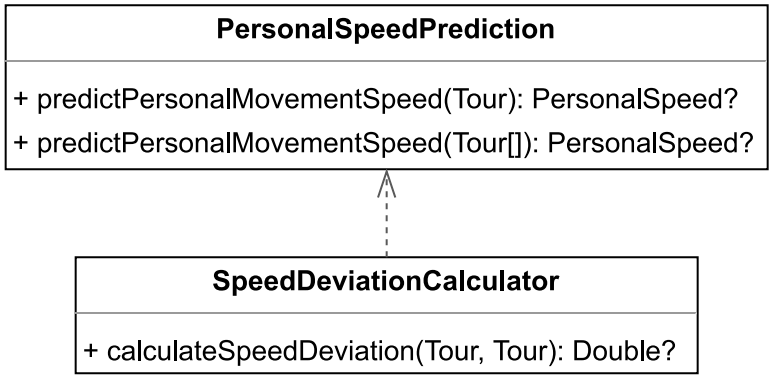

Figure 7: UML Diagramm mit Umsetzung des SRP

Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel

Ein Beispiel wo das OCP angewandt wurde ist im Instruction Interface. Es ist der Zentrale Punkt in der Anwendungslogik, das den Matcher einer Eingabe darstellt. In der execute() Methode wird der zugehörige Use Case ausgeführt. Durch die Implementierung des Interfaces können neue Befehle einfach hinzugefügt werden ohne bestehende Befehle zu verändern.

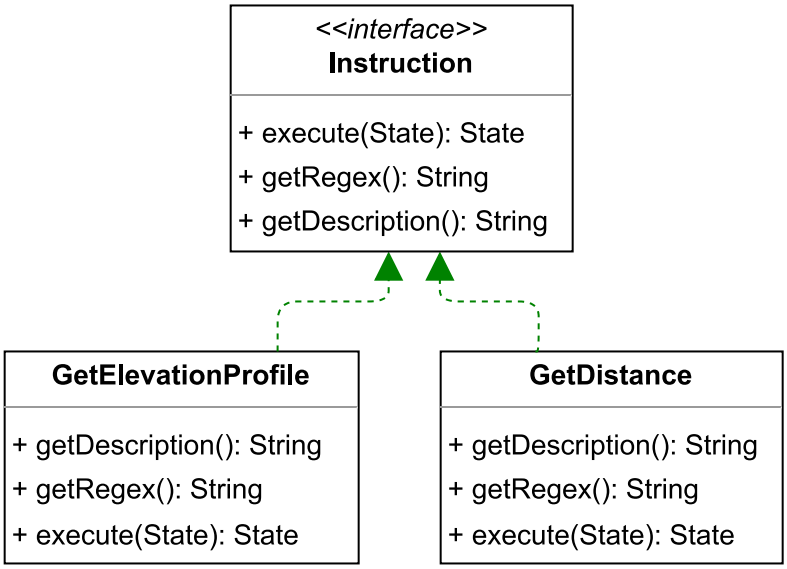

Figure 8: UML Diagramm des Instruction Interface mit 2 Implementierungen

Negativ-Beispiel

Ein Beispiel wo das OCP nicht angewandt wurde ist bei der Klasse Console. Sie ist dazu zuständig die Verbindung zwischen verschiedenen Befehlen zu gewährleisten. Wollte man diese anderst Umsetzten, etwa mithilfe von Event listenern oder ähnlichem, müsste man die bestehende Implementierung ändern.

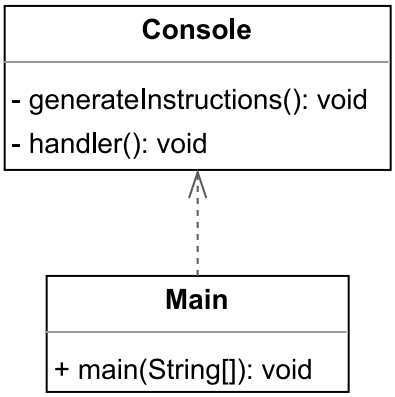

Figure 9: UML Diagramm der Klasse SpeedCalculator

Mithilfe eines ProgramFlow Interfaces könnte hier das OCP verwendet werden.

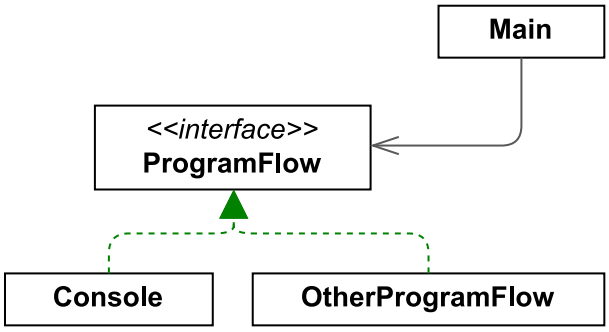

Figure 10: UML Diagramm mit Umsetzung des SRP

Dependency-Inversion-Principle (DIP))

Positiv-Beispiel

Bei der TimePrediction wurde das Dependency Inversion Principle angewandt, da verhindert wird, dass diese von den Details eines Bestimmten Pfades abhängt. Durch die Einsetzun des Path Interfaces hängt nun die Detailimplementation(Track) von der Abstraktion(Path) ab.

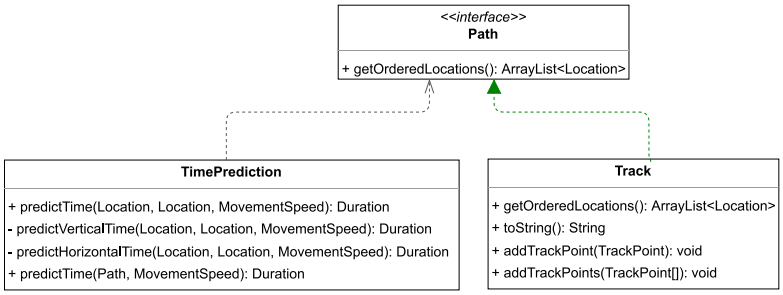

Figure 11: Dependency Inversion von Vorhersagen bei verschiedenen Pfaden

Negativ-Beispiel

Dadurch, dass der Befehl ReadPath neben der Abstraktion auch von der Detailimplementierung abhängt ist das DIP hier nicht erfüllt. Besser wäre,

wenn in einer abstrakteren Schicht der detaillierte DOMParser als XML-Parser übergeben wird statt ihn bei der Instanziierung zu erzeugen.

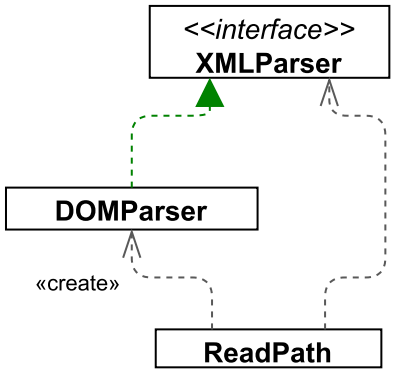

Figure 12: Keine Dependency Inversion beim Lesen eines Pfads

Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

Negativ-Beispiel

Analyse GRASP: Hohe Kohäsion

Don't Repeat Yourself (DRY)

Erstellen der Helferklasse ProfileCalcultion für die Berechnung von Profilen, welchen von den Klassen (damals)ElevationProfile un d SpeedProfile verwendet wird (commit 8ffd648d794563fea2c8662debe12ca1277b1b3e). Da die Methoden jeweils unabhängig vom Inhalt des jeweiligen Profils ausgeführt werden und genau Dasselbe tun können sie ausgelagert und dann darauf zugegriffen werden. Dies verhindert eine wiederholte Implementierung, hat aber keinerlei Auswirkungen auf das Ergebnis. In einem Anschliessenden Commit wird dieser Effekt sogar noch Ausgeweitet, indem anstelle von Minima und Maxima zur Normalisierung nur die Liste selbst angegeben wird und diese Werte dann in der ausgelagerten Methode berechnet werden (commit 996f066a8f26f78852df00c85888f7236b87b458).

Vorher

```
1  public class SpeedProfile {
2          private boolean[][] calculateSpeedProfile(Tour
               tour, int xGranularity) {
3                  ...
4                  int[] sectionLength = split(tourPoints.
                       size() , xGranularity);
5                  ...
6                  speeds = normalize(speeds,min,max);
7                  ...
8          }
9
10     private List<Double> normalize(List<Double> list ,
           double min, double max){
```

```
11          double diff = max - min;
12          for (int i = 0; i < list.size(); i++) {
13              double val = list.get(i);
14              double normalizedVal = (val - min) / diff;
15              list.set(i,normalizedVal);
16          }
17          return list;
18      }
19
20      private int[] split(int pool,int sections){
21          int[] output = new int[sections];
22          int base = pool/sections;
23          int remainder = pool % sections;
24          for (int i = 0; i < output.length; i++){
25              if (remainder <= i){
26                  output[i] = base;
27              }
28              if (remainder > i){
29                  output[i] = 1+base;
30              }
31          }
32          return output;
33      }
34  }
35
36  public class ElevationProfile {
37          private boolean[][] calculateSpeedProfile(Tour
                tour, int xGranularity) {
38                  ...
39                  int[] sectionLength = split(locations.
                        size() , xGranularity);
40                  ...
41                  heights = normalize(heights,min,max);
42                  ...
43          }
44
45      private List<Double> normalize(List<Double> list ,
            double min, double max){
```

```
46          double diff = max - min;
47          for (int i = 0; i < list.size(); i++) {
48              double val = list.get(i);
49              double normalizedVal = (val - min) / diff;
50              list.set(i,normalizedVal);
51          }
52          return list;
53      }
54
55      private int[] split(int pool,int sections){
56          int[] output = new int[sections];
57          int base = pool/sections;
58          int remainder = pool % sections;
59          for (int i = 0; i < output.length; i++){
60              if (remainder <= i){
61                  output[i] = base;
62              }
63              if (remainder > i){
64                  output[i] = 1+base;
65              }
66          }
67          return output;
68      }
69  }
```

Nacher

```
1  public class SpeedProfile {
2          private boolean[][] calculateSpeedProfile(Tour
               tour, int xGranularity) {
3                  ...
4                  int[] sectionLength =
                       ProfileCalculation.split(tourPoints.
                       size() , xGranularity);
5                  ...
6                  speeds = normalize(speeds,min,max);
7                  ...
```

```
8           }
9   }
10
11  public class ElevationProfile {
12          private boolean[][] calculateSpeedProfile(Tour
                tour, int xGranularity) {
13                  ...
14                  int[] sectionLength =
                        ProfileCalculation.split(locations.
                        size() , xGranularity);
15                  ...
16                  heights = ProfileCalculation.normalize(
                        heights.stream().map(e->e.getValue()
                        ).toList(),min.getValue(),max.
                        getValue());
17                  ...
18          }
19  }
20
21  public class ProfileCalculation {
22      public static List<Double> normalize(List<Double>
            list, double min, double max){
23          double diff = max - min;
24          for (int i = 0; i < list.size(); i++) {
25              double val = list.get(i);
26              double normalizedVal = (val - min) / diff;
27              list.set(i,normalizedVal);
28          }
29          return list;
30      }
31
32      public static int[] split(int pool,int sections){
33          int[] output = new int[sections];
34          int base = pool/sections;
35          int remainder = pool % sections;
36          for (int i = 0; i < output.length; i++){
37              if (remainder <= i){
38                  output[i] = base;
```

```
39                  }
40                  if (remainder > i){
41                      output[i] = 1+base;
42                  }
43              }
44              return output;
45          }
46  }
```

Kapitel 5: Unit Tests

10 Unit Tests

Unit Test	Beschreibung
ElevationGainTest#addElevation	Test auf korrekte Summierung von a Auf- und Abstieg
ElevationGainTest#getManhattenNorm	Test auf korrekte Berechnung der Manhattennorm eines Elevationgains
DistanceCalculatorTest#calc3dDistance	Test auf korrekt genuge heuristische Berechnung im dreidimensionalen Raum
DistanceCalculatorTest#calcElevationGain	Test auf korrekte Berechnung einer Höhendifferenz zwische 2 Punken und eines gesamten GPX Tracks
ProfileCalculationTest#split	Test auf die Korrekte Aufteilung von Wegpunkten in Balken für Profile
ProfileCalculationTest#normalize	Test auf die Korrekte Normalisierung von Datenpunkten für die Erstellung von Profilen
SpeedHeuristicsTest#calculateTime	Test auf die Korrekte Auswertung der tatsächlich benötigten Zeit aus Toursegmenten
ElevationProfileTest#getProfile	Test auf die Korrekte erstellung einer Matrix die ein Höhenprofil repräsentiert
SpeedCalculatorTest#predictPMSSingle	Test auf die Korrekte erstellung eines Personal Movement Speeds (PMS) aus einer gegangenen Tour
SpeedHeuristicsTest#getClimbingHeuristic	Test auf genau genuge heuristische Berechnung einer Geschwindigkeit mit der Steigungen bezwungen werden

ATRIP: Automatic

Automatic wurde realisiert indem per rechtem Mausklick auf Verzeichnis src/test und Auswahl der Option 'Run 'Tests in test" alle Tests ausgeführt werden.

ATRIP: Thorough

Positivbeispiel

ProfileCalculationTest der die Grundfunktionen für die Erstellung von Profilen bereitstellt. Das Thoroughe dabei ist dass alle Fehlerfälle abgedeckt sind die Mathematisch auftreten können, wie etwa die Eingabe von Nullparametern oder die Unterrepräsentation von Daten.

```
1
2  @Test
3      void normalize(){
4          List<Double> list = new ArrayList<>();
5          list.add(-10.0);
6          list.add(0.0);
7          list.add(10.0);
8          list = ProfileCalculation.normalize(list);
9          assertEquals(0,list.get(0));
10         assertEquals(0.5,list.get(1));
11         assertEquals(1,list.get(2));
12     }
13
14     @Test
15     void normalizeFlatDiff(){
16         List<Double> list = new ArrayList<>();
17         list.add(4.0);
18         list.add(4.0);
19         list.add(4.0);
20         list = ProfileCalculation.normalize(list);
21         assertEquals(1,list.get(0));
22         assertEquals(1,list.get(1));
23         assertEquals(1,list.get(2));
24     }
25
26     @Test
27     void split(){
28         int[] sections = ProfileCalculation.split(22,5)
               ;
29         assertEquals(5,sections[0]);
```

```
30          assertEquals(5,sections[1]);
31          assertEquals(4,sections[2]);
32          assertEquals(4,sections[3]);
33          assertEquals(4,sections[4]);
34      }
35
36      @Test
37      void splitTooSmallGranularity(){
38          int[] sections = ProfileCalculation.split
                (22,25);
39          assertEquals(1,sections[19]);
40          assertEquals(1,sections[20]);
41          assertEquals(1,sections[21]);
42          assertEquals(0,sections[22]);
43      }
```

Negativbeispiel

Instructions für die Konsole
Dadurch, dass diese Klassen die Äußerste Schicht im Sinne der Clean Architecture darstellen und aufgrund der Anforderung auf Fokus ausserhalb der User Experience liegt sind diese Klassen weder besonders komplex noch in einem vollständig ausgearbeiteten Zustand, welcher außerhalb einer Konsolenanwendung läge.
Da die Hauptlogik in der Interaktion mit dem Benutzer liegt, wurden hier manuelle acceptence Tests angewandt

ATRIP: Professional

Positivbeispiel

The distanceCalculatortest Tests the correct interpretation of Distances from a real Track with an reduced amount of Waypoints. Those waypoints are created in an extra Helper Class, since It's creation is not in the responsibility of the DistanceCalculator to be tested. They are Called before each Test, which ensures the tests are isolated from one another. Additionally, there are no big code smells in the Classes to be tested.

```
1   Track mountainTrack;
```

```
2
3       @BeforeEach
4       void init() {
5           mountainTrack = GetTracks.getMountainTrack();
6       }
7
8       @Test
9       void calc2dDistance() {
10          Distance distanceToFirstHut2D =
                DistanceCalculator.calc2dDistance(
                mountainTrack.getOrderedLocations().get(0),
                mountainTrack.getOrderedLocations().get(1));
11          assertEquals(308, distanceToFirstHut2D.getValue
                (), 1);
12
13          Distance distanceOfWholeTrack2D =
                DistanceCalculator.calc2dDistance(
                mountainTrack);
14          assertEquals(7795, distanceOfWholeTrack2D.
                getValue(), 10);
15      }
16
17      @Test
18      void calc3dDistance() {
19          Distance distanceToFirstHut2D =
                DistanceCalculator.calc3dDistance(
                mountainTrack.getOrderedLocations().get(0),
                mountainTrack.getOrderedLocations().get(1));
20          assertEquals(310, distanceToFirstHut2D.getValue
                (), 1);
21
22          Distance distanceOfWholeTrack3D =
                DistanceCalculator.calc3dDistance(
                mountainTrack);
23          assertEquals(8315, distanceOfWholeTrack3D.
                getValue(), 10);
24      }
25
```

```
26      @Test
27      void calcElevationGain() {
28          ArrayList<Location> locations = mountainTrack.
                getOrderedLocations();
29          ElevationGain uphillSection =
                DistanceCalculator.calcElevationGain(
                locations.get(1), locations.get(2)); //
                uphill
30          assertEquals(569, uphillSection.getUp(), 1);
31          assertEquals(0, uphillSection.getDown(), 1);
32
33          ElevationGain downhillSection =
                DistanceCalculator.calcElevationGain(
                locations.get(4), locations.get(5)); //
                downhill
34          assertEquals(0, downhillSection.getUp(), 1);
35          assertEquals(539, downhillSection.getDown(), 1)
                ;
36
37          ElevationGain wholeTrack = DistanceCalculator.
                calcElevationGain(mountainTrack); //whole
                Track
38          assertEquals(1346, wholeTrack.getUp(), 1);
39          assertEquals(1493, wholeTrack.getDown(), 1);
40      }
41
42      @Test
43      void calcAltitudeData() throws
           InsufficientDataException {
44          List<Location> locations = mountainTrack.
                getOrderedLocations();
45          Elevation lowestPoint = DistanceCalculator.
                calcMinAlt(locations);
46          assertEquals(1096, lowestPoint.getValue(), 1);
47
48          Elevation highestPoint = DistanceCalculator.
                calcMaxAlt(locations);
49          assertEquals(2589, highestPoint.getValue(), 1);
```

```
50
51          Elevation averageAltitude = DistanceCalculator.
                calcAvgAlt(locations);
52          assertEquals(1778, averageAltitude.getValue(),
                1);
53      }
```

Negativbeispiel

Ein negativbeispiel ist die Methode evaluationFunction der Klasse EvolutionaryDistTest. Aufgrund der hohen komplexität des Algorithmus und der Eingabeparameter wird auf höchstem Level quasi eine Instruction mit gegebenen Usereingaben ausgeführt und der Status der Klasse danach überprüft. Allerdings werden auch DOMParser und Bewertungsfunktion mitgetested, was nicht professionell ist.

```
1   Track mountainTrack;
2
3       @BeforeEach
4       void init() {
5           mountainTrack = GetTracks.getMountainTrack();
6       }
7
8       @Test
9       void calc2dDistance() {
10          Distance distanceToFirstHut2D =
                DistanceCalculator.calc2dDistance(
                mountainTrack.getOrderedLocations().get(0),
                mountainTrack.getOrderedLocations().get(1));
11          assertEquals(308, distanceToFirstHut2D.getValue
                (), 1);
12
13          Distance distanceOfWholeTrack2D =
                DistanceCalculator.calc2dDistance(
                mountainTrack);
14          assertEquals(7795, distanceOfWholeTrack2D.
                getValue(), 10);
15      }
```

```
16
17      @Test
18      void calc3dDistance() {
19          Distance distanceToFirstHut2D =
                DistanceCalculator.calc3dDistance(
                mountainTrack.getOrderedLocations().get(0),
                mountainTrack.getOrderedLocations().get(1));
20          assertEquals(310, distanceToFirstHut2D.getValue
                (), 1);
21
22          Distance distanceOfWholeTrack3D =
                DistanceCalculator.calc3dDistance(
                mountainTrack);
23          assertEquals(8315, distanceOfWholeTrack3D.
                getValue(), 10);
24      }
25
26      @Test
27      void calcElevationGain() {
28          ArrayList<Location> locations = mountainTrack.
                getOrderedLocations();
29          ElevationGain uphillSection =
                DistanceCalculator.calcElevationGain(
                locations.get(1), locations.get(2)); //
                uphill
30          assertEquals(569, uphillSection.getUp(), 1);
31          assertEquals(0, uphillSection.getDown(), 1);
32
33          ElevationGain downhillSection =
                DistanceCalculator.calcElevationGain(
                locations.get(4), locations.get(5)); //
                downhill
34          assertEquals(0, downhillSection.getUp(), 1);
35          assertEquals(539, downhillSection.getDown(), 1)
                ;
36
37          ElevationGain wholeTrack = DistanceCalculator.
                calcElevationGain(mountainTrack); //whole
```

```
            Track
38          assertEquals(1346, wholeTrack.getUp(), 1);
39          assertEquals(1493, wholeTrack.getDown(), 1);
40      }
41
42      @Test
43      void calcAltitudeData() throws
            InsufficientDataException {
44          List<Location> locations = mountainTrack.
                getOrderedLocations();
45          Elevation lowestPoint = DistanceCalculator.
                calcMinAlt(locations);
46          assertEquals(1096, lowestPoint.getValue(), 1);
47
48          Elevation highestPoint = DistanceCalculator.
                calcMaxAlt(locations);
49          assertEquals(2589, highestPoint.getValue(), 1);
50
51          Elevation averageAltitude = DistanceCalculator.
                calcAvgAlt(locations);
52          assertEquals(1778, averageAltitude.getValue(),
                1);
53      }
```

Code Coverage

Aktuell liegt die Testabdeckung bei 78% class coverage und 73% line coverage. Der Grund hierfür ist hauptsächlich die geringe Testabdeckung der äußeren Schichten im Sinne der clean architecture die in den Verzeichnissen Application und Interfaces liegen, während die Verzeichnisse mit den komplizierten Berechnungen und der Grundstruktur bei 91 bzw. 94% line coverage liegen. Die restlichen Prozente lassen sich durch auslassen trivialster Testfälle, etwa getter und setter, erklären.

Fakes und Mocks

1. Mockobjekt: DirectWayHeuristik

Hier wurde beim Test der Erstellung von Umwegen ein Mockobjekt für die Bestimmung der Zeit für die Umleitung verwendet. Dies ist Sinnvoll, da hier die Korrekte Initialisierung der Umwege an der richtigen Stelle getestet werden soll und nicht die Bestimmung der Zeit, die man benötigt um sich eine Gewisse Strecke zu bewegen.

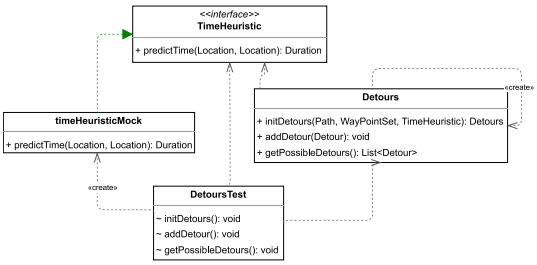

Figure 13: UML Diagramm des Mocks für DetoursTests

2. Mockobjekt: EvaluationFunction

Hier wurde beim Test des Hillclimbingalgorithmus die Bewertung der einzelnen Tests gemockt. Die Bewertungsfunktion bewertet alle Lösungen gleich gut bis auf die Lösung wahr,wahr,falsch, die bevorzugt bewertet wird. Dies macht deutlich, dass diese Lösung als Lokales (und globales) Optimum beim Hillclimbingalgorithmus herauskommen soll, da es das einzige existierende Optimum ist. Das Mock Objekt ist sinnvoll, da die Bewertungsfunktion in anderen Tests abgedeckt ist und es lediglich um das Korrekte finden Lokaler Optimums geht bei der Hillclimbing-Klasse.

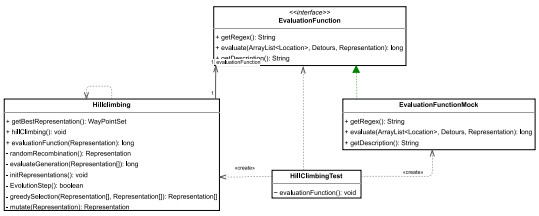

Figure 14: UML Diagramm des Mocks für eine Bewertungsfunktion

Kapitel 6: Domain Driven Design

Ubiquitous Language

Track : sortierte Wegpunkte die chronologisch zusammenhängen zu einem gesamten Weg
Tour : sortierte Wegpunkte die in der Vergangenheit zusammenhängen zu einem zu einem festgelegten begangenen gesamten Weg

Entities

Klasse Hillclimbing Verbindet eine Anzahl an Lösungen für ein spezielles Problem. Zur Erstellung der Entität sind die vorgeschlagenen Lösungen noch trivial, mit der Lebenszeit der Entity verbessern sich die Qualitäten der Lösungen.

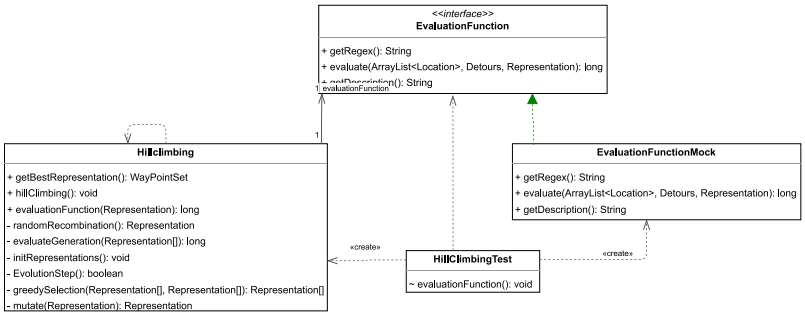

Figure 15: UML der Entity Hillclimbing

Value Objects

Die Klasse Elevation setzt den Werte einer Höhe über dem Meeresspiegel um. Die klasse ist immutable und bei Erstellung wird geprüft ob sich der Wert in einem Auf der Erde Sinnvollen Rahmen bewegt (-500 bis 9000). Eine Überschreibung der Hashfunktion oder der equals Methode wurde aufgrund von mangelnder Nötigkeit nicht Implementiert.

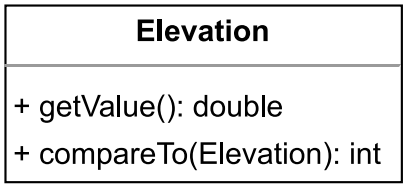

Figure 16: UML des Elevation Value Objects

Repositories

Wegpunkt + Lat + Lon + Elevation als Ortsbeschreibung ?? wahrscheinlich kein aggregate?

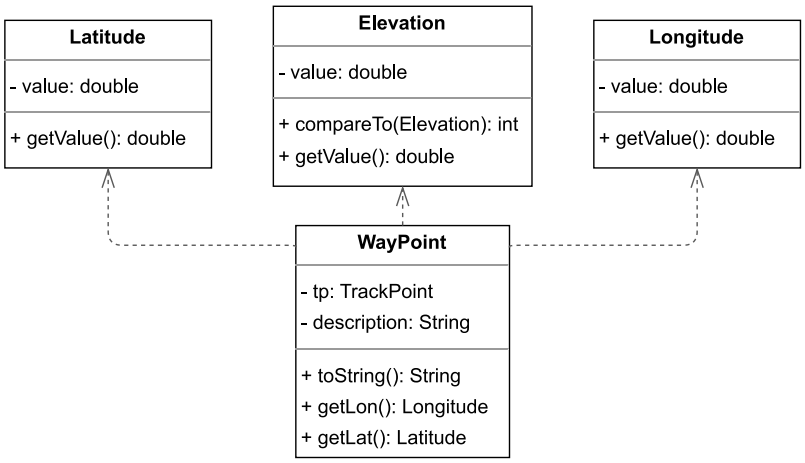

Figure 17: UML des Wegpunkt Repositories

Aggregates

Keine, da keine persistente Datenspeicherung

Kapitel 7: Refactoring

Code Smells

Code Smell: Duplicated Code

(Ausgebessert in commit https://github.com/reichert-p/GPXrechner/commit/4039bc31f631050887
)
Um die Strafe von schlechten Aufteilungen im Toursplitting zu berechnen, müss bei allen Bewertungsfunktionen erst gefiltert werden welche Umwege tatsächlich genommen werden.
Vorheriger Zustand:

```
1          class StayNightEvaluation {
2          private double getWeightedOvershoot(ArrayList<
               Location> path, Detours detours,
               Representation representation) {
3           List<Detours.Detour> visitedDetours = new
               ArrayList<>();
4          for (int i = 0; i < detours.getPossibleDetours
               ().size();i++){
5           if (representation.getBitstring()[i]){
6                visitedDetours.add(detours.
                    getPossibleDetours().get(i));
7           }
8          }
9          List<Detours.Detour> orderedVisitedDetours =
               visitedDetours.stream().sorted(Comparator.
               comparing(Detours.Detour::getPosition)).
               toList();
10 ...
11         }
12         }
```

```
1
2          class SupplyEvaluation {
3          private double getOvershoot(ArrayList<Location>
               path, Detours detours, Representation
               representation) {
```

```
4           List<Detours.Detour> visitedDetours = new
                ArrayList<>();
5           for (int i = 0; i < detours.getPossibleDetours
                ().size();i++){
6               if (representation.getBitstring()[i]){
7                   visitedDetours.add(detours.
                        getPossibleDetours().get(i));
8               }
9           }
10          List<Detours.Detour> orderedVisitedDetours =
                visitedDetours.stream().sorted(Comparator.
                comparing(Detours.Detour::getPosition)).
                toList();
11 ...
12          }
13          }
```

Lösung: Auslagerung dieser Funktionalität in externe statische Klasse EvaluationHelper.

```
1
2 public class EvaluationHelper {
3     public static List<Detours.Detour>
          getRepresentedDetoursOrdered(Detours detours,
          Representation representation){
4           List<Detours.Detour> visitedDetours = new
                ArrayList<>();
5           for (int i = 0; i < detours.getPossibleDetours
                ().size();i++){
6               if (representation.getBitString()[i]){
7                   visitedDetours.add(detours.
                        getPossibleDetours().get(i));
8               }
9           }
10          return visitedDetours.stream().sorted(
                Comparator.comparing(Detours.Detour::
                getPosition)).toList();
11    }
12 }
```

Die Umsetzung in den einzelnen Bewertungsfunktionen sieht dann wie folgt aus:

```
1
2         class SupplyEvaluation {
3  var orderedVisitedDetours = EvaluationHelper.
       getRepresentedDetoursOrdered(detours,
       representation);
4 ...
5         }
6 }
```

Code Smell: Code Comments

remove runaways Kommentar in der percentileandaverage Methode der Klasse SpeedHeuristics.

```
1
2 for (double d:paceValues) {
3         if (i + 1 > paceValues.size() * 0.25 && i <
              paceValues.size() * 0.9) // remove runaways
4         sum += d;
5         instances ++;
6         i++;
7 }
```

Die Lösung ist, die Funktionalität in eine Methode mit sprechendem Namen auszulagern.

```
1         ...
2         var consideredPaceValues =
              removeRunawaysfromSortedList(paceValues,
              0.25, 0.9);
3         ...
4
5 private static List<Double>
      removeRunawaysfromSortedList(List<Double> input,
      double lowerBound, double upperBound){
6         if(lowerBound < 0 || lowerBound > 1 ||
              upperBound < 0 || upperBound > 1){
```

```
7                 throw new RuntimeException("Bounds in
                      removeRunaways should be a value between
                      0 and 1");
8             }
9             int length = input.size();
10            return input.subList((int)(length*lowerBound),(
                  int)Math.ceil(length*upperBound));
11        }
```

2 Refactorings

1. Refactoring: Rename Class

https://github.com/reichert-p/GPXrechner/commit/7947597903210e73647ffbc20d07e6b5b257cff2#
a3b6bacc77a02d2a1746cbbbd776a467bf3fcd7920bfefbef1e892af10172e50
Umbenennung der Klasse EvolutionaryDist zu Hillclimbing, da der Name besser zum Inhalt der Klasse passt. Da sich beim UML nur der Name ändert wird auf ein zweites Diagramm verzichtet.

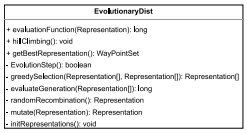

Figure 18: UML Diagramm der Klasse EvolutionaryDist

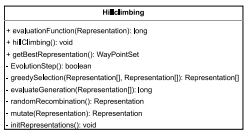

Figure 19: UML Diagramm der Klasse unter dem Namen HillClimbing

2. Refactoring: Extract Method

commit 59f9045a2ac73496111bba87c35016c2b26108e2 und an selber Stelle im Anschluss 7d57943bb5fcdf2d23e76f05c0157f7753f6c05e.
Die Methode generateGPX zur Erstellung einer .gpx Datei aus gespeicherten Touren und Tracks war 35 Zeilen lang und Spaghetticode in Reinform. Mithilfe

der Extraction der beiden Methoden wurde Struktur in den Code gebracht und die Sprechenden Methodennamen helfen später beim Verständniß des Codes.

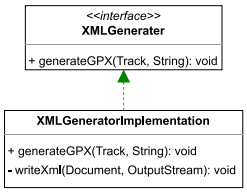

Figure 20: UML Diagramm der Klasse XMLGeneratorImplementation vor dem Refactoring

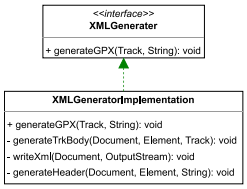

Figure 21: UML Diagramm der Klasse XMLGeneratorImplementation vor dem Refactoring

Kapitel 8: Entwurfsmuster

Entwurfsmuster Fabrik

Zur Erzeugung verschiedener Objekte aus GPX Dateien wurde das Entwurfsmuster Fabrik eingesetzt. Da aus der GPX nicht eindeutig erkennbar ist welches Objekt erzeugt werden muss, muss diese Logik vom Nutzer mitgegeben werden.
FÜr das Entwurfsmuster wurde sich entschieden um die Lesbarkeit des Programmcodes zu erhöhen und die Logik in eigene Klassen zu Kapseln.

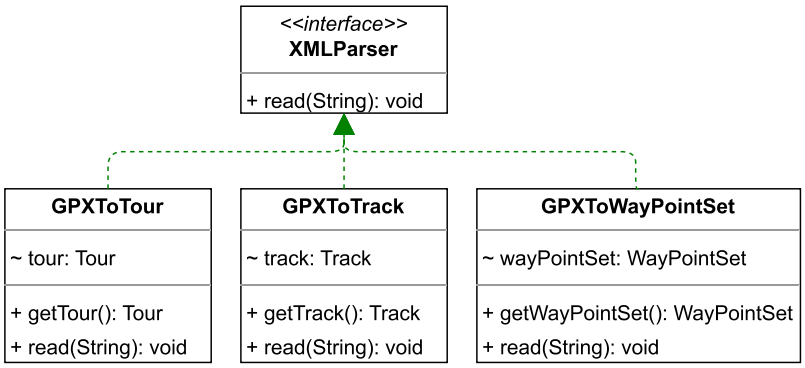

Figure 22: UML Diagramm der Fabrik für die Erzeugung von Objekten aus GPX Dateien

Entwurfsmuster Strategie

Bei der Evolutionären Optimierung von Umwegen wird müssen erzeugte Lösungen bewertet werden. Abhängig vom Anwendungsfall können diese Bewertungsalgorithmen stark voneinander Abweichen. Um dies flexibel umzusetzen und weitere Bewertungsfunktionen zukünftig gut umsetzen zu können und die Übersichtlichkeit sowie Testbarkeit zu verbessern wurde auf das Strategie-Entwurfsmuster zurückgegriffen.

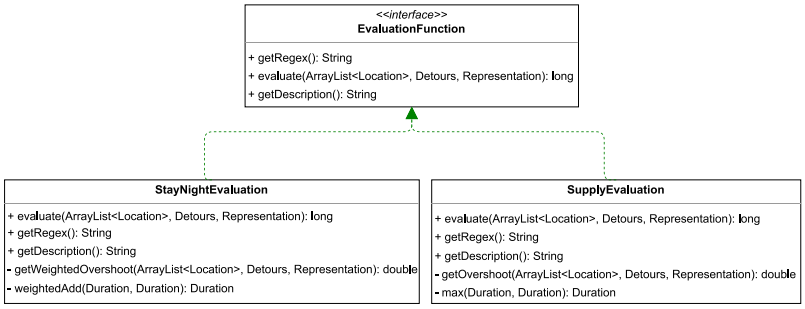

Figure 23: UML Diagramm der Strategie für die Bewertung von Umwegsoptimierungen