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

Inhaltsverzeichnis

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 Strecken welche an allen Koordinaten Zeitstempel haben (Tour). Für beide kann die Höhendifferenz und die Strecke berechnet werden. Ein Höhenprofil von Strecken kann in der Konsole angezeigt werden.

Mithilfe von Bewegungsgeschwindigkeiten kann die voraussichtliche Dauer einer Begehung einer Strecke vorhergesagt werden. Dabei kann die Bewegungsgeschwindigkeit entweder aus bereits begangenen Strecken berechnet werden oder es kann eine Sportart aus einer Auswahl (Wandern, Radfahren, ...) gewählt werden. Um Schlüsse über die eigene Geschwindigkeit herauszufinden, kann man sich entweder die aus Touren gewonnene Bewegungsgeschwindigkeit anzeigen lassen oder ein Geschwindigkeit-Zeit-Diagramm einer Tour anzeigen lassen.

Auf langen Touren kann es notwendig sein, Umwege einzulegen, um zu Übernachten oder Vorräte aufzufüllen. Da die Wahl der optimalen Umwege nicht trivial ist[1], wurde ein evolutionärer Hillclimb-Algorithmus zur möglichst optimalen Wahl der Stützpunkte gewählt. Hierfür muss bei eine Tour oder ein Track anhand einer Auswahl an Wegpunkten (etwa Hütten oder Supermärkten) entschieden werden, welche davon besucht werden müssen. Mithilfe einer Bewegungsgeschwindigkeit und einer maximale Dauer zwischen den Stützpunkten können die (möglichst) optimalen zu wählenden Stützpunkte berechnet werden.

Wie startet man die Applikation?

Benötigt wird eine IDE die Java 19 ausführen kann.

[1] nach einstündiger, ergebnisloser Überlegung wurde auf die Erstellung einer Reduzierung auf das Knapsack-Problem verzichtet, da dies über den Umfang des Projekts hinausginge

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, einen Befehl einzugeben. Gibt man eine nicht zulässigen Befehl oder *help* ein, so bekommt man eine Übersicht über alle möglichen Befehle.

Der erste Befehl ist üblicherweise *load gpx*, um eine Tour oder Track aus einer GPX Datei zu laden. Dies wird benötigt, um Informationen oder Analysen der Tour oder des Tracks zu bekommen.

Da alle GPX-Dateien in dem dafür vorgesehenen Ordner abgelegt sind, muss der Pfad zu ihnen relativ zu diesem Ordner angegeben werden.
Möchte man beispielsweise einen Track der berühmten (und im Repository vorhandenen) Watzmann Überschreitung laden, gibt man den Pfad *Track/Watzmann.gpx* an.

Wie testet man die Applikation?

Die Tests befinden sich unter *src/test/*.
Zum Testen der Applikation führt man diese mithilfe seiner 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, das darauf abzielt, Code klar zu organisieren und leicht wartbar, testbar und erweiterbar zu machen. Hierfür werden die Bestandteile einer Anwendung in verschiedenen hierarchischen Schichten gekapselt, sodass äußere Schichten von inneren abhängen können, jedoch nicht umgekehrt. Tiefere Schichten sind langlebiger als äußere Schichten.

Analyse der Dependency Rule

Positiv-Beispiel: Dependency Rule

Das Positivbeispiel ist die Klasse *GetDistance*, die eine Implementation eines *Instruction* Interfaces ist. Sie liegt in der Anwendungsschicht. Sie implementiert den Anwendungsfall, dass Benutzer die Strecke eines Weges erfahren wollen.
Um den Weg zu erhalten, für den die Strecke berechnet werden soll, greift sie auf den *State* auf der Anwendungsschicht zu. Die eigentliche Berechnung findet mithilfe der Klasse *DistanceCalculator* statt, welche in der Domain-Schicht liegt. Die Strecke wird über das *UserOutput* Interface mitgeteilt, welches in der Anwendungsschicht liegt. Damit hat die Klasse *GetDistance* lediglich Abhängigkeiten auf die eigene und tiefere Schichten und erfüllt die Dependency Rule.

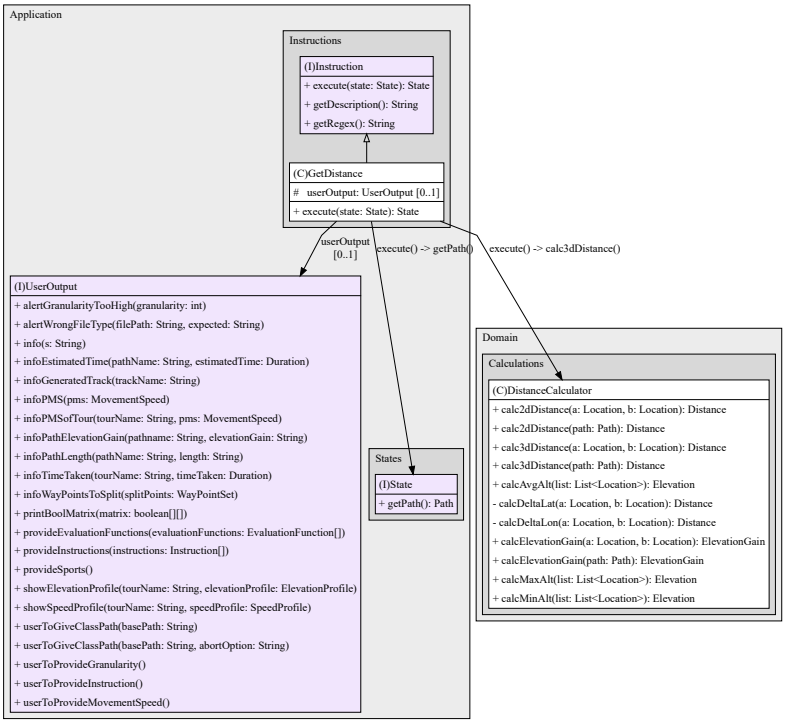

Abbildung 1: Abhängigkeiten der Klasse GetDistance

Negativ-Beispiel: 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 Klasse *ReadPath* verwendet, welche in der Anwendungsschicht liegt. Somit besteht eine Abhängigkeit von der inneren Anwendungsschicht zur äußeren Plugin-Schicht, was eine Verletzung der Dependency Rule darstellt.

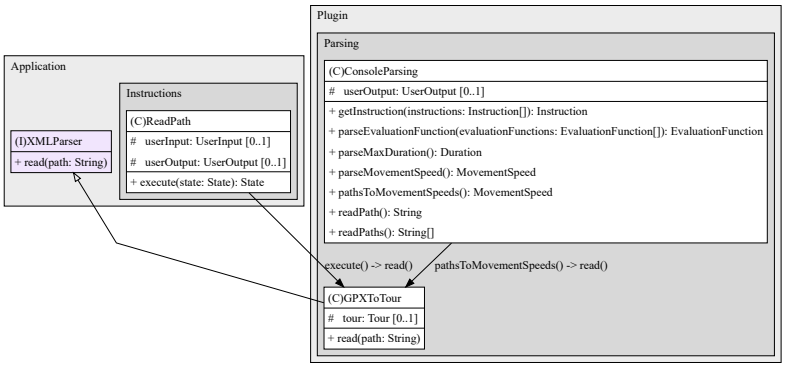

Abbildung 2: Abhängigkeiten auf die Klasse GPXToTour

Schicht: Domain Code

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

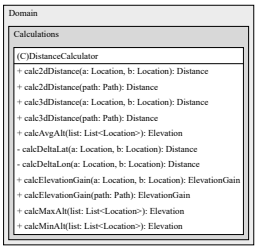

Abbildung 3: UML Diagramm der Klasse DistanceCalculator

Schicht: Plugins

Die Klasse *ConsoleParsing* ist dafür zuständig, verschiedene Formen Input von Benutzern zu erfassen. Dies umfasst beispielsweise den Pfad zu GPX Dateien, die Wahl einer Sportart oder Geschwindigkeit sowie die Eingabe

einer Zeit. Somit stellt die Klasse einen wesentlichen Bestandteil der Benutzerschnittstelle dar. Es wäre denkbar, die Klasse durch eine grafische Benutzerschnittstelle zu ersetzen.

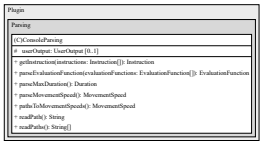

Abbildung 4: UML Diagramm der Klasse ConsoleParsing

Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel

Die Klasse *Latitude* repräsentiert eine Breite im Sexagesimalsystem. Die einzige Aufgabe ist dabei die korrekte Repräsentation eines Breitengrades, womit sie das Single-Responsibility-Principle einhält.

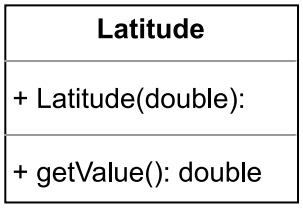

Abbildung 5: UML Diagramm der Klasse Latitude

Negativ-Beispiel

Die Klasse *SpeedCalculator* implementiert verschiedene Berechnungen von Geschwindigkeiten. Zwei der Methoden berechnen die Bewegungsgeschwindigkeit (Zusammengesetzt aus horizontaler, auf- und absteigender Geschwindigkeit) einer oder mehrerer begangener Touren. Eine weitere die relative Geschwindigkeit zweier Touren zueinander. Dies wird benötigt um Geschwindigkeit-Zeit-Diagramme zu erstellen. Die Klasse hat also zwei Aufgaben und verletzt somit das Single-Responsibility-Principle.

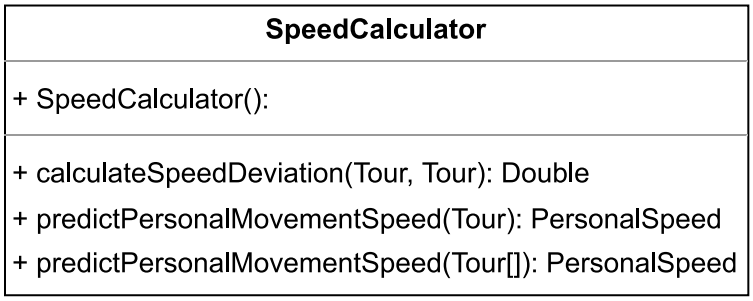

Abbildung 6: UML Diagramm der Klasse SpeedCalculator

Um hier das Single-Responsibility-Principle umzusetzen könnte die Klasse aufgeteilt werden. Dadurch entstünden zwei neuen Klassen, welche jeweils nur eine Aufgabe hätten.

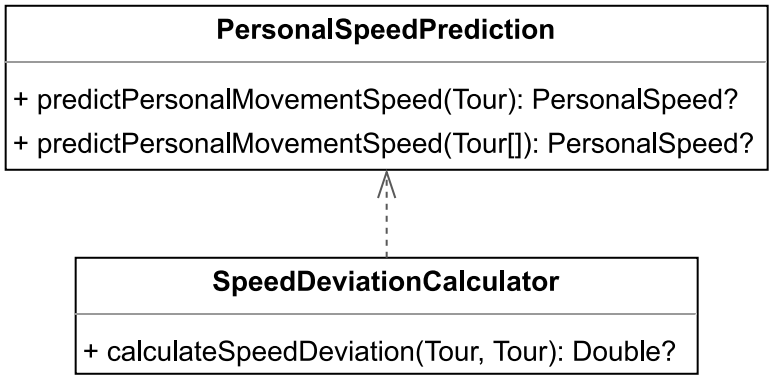

Abbildung 7: Abbildung mit umgesetzten SRP

Analyse Open-Closed-Principle (OCP)

Positiv-Beispiel

Ein Positivbeispiel für die Anwendung des Open-Closed-Prinzips (OCP) findet sich im *Instruction* Interface. Dieses Interface bildet den zentralen Punkt in der Anwendungslogik der Anwendungsschicht. Implementierungen einer *Instruction* implementieren die Logik für die jeweiligen Anwendungsfälle. Durch Ausführung der *execute* Methode wird der jeweilige Anwendungsfall ausgeführt. Durch die Umsetzung mithilfe des Interfaces können neue Anwendungsfälle problemlos hinzugefügt werden, ohne die Implementierung bestehender Befehle zu verändern.

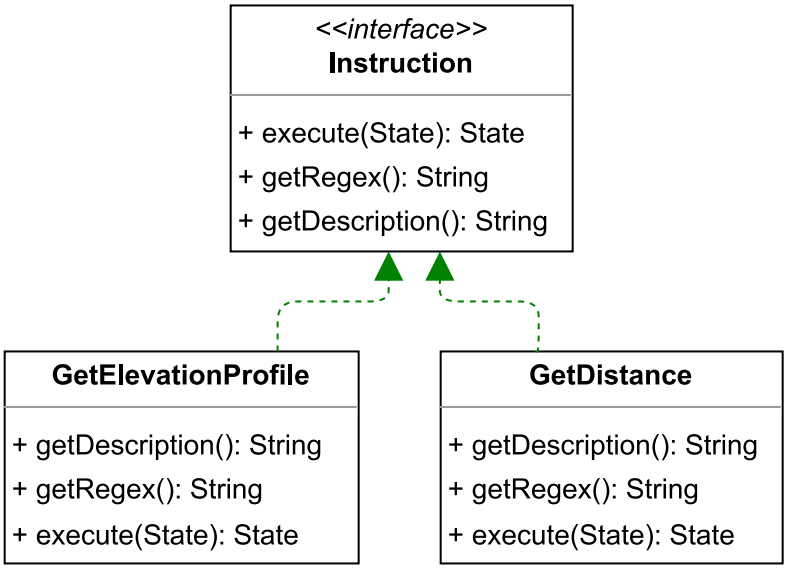

Abbildung 8: UML Diagramm des Interfaces *Instruction* mit 2 beispielhaften Implementierungen

Negativ-Beispiel

Die Umsetzung des Programmflusses, welcher die Reihenfolge der wählbaren Anweisungen bestimmt, verletzt das Open-Closed-Prinzip. Derzeit wird lediglich die Klasse *Console* implementiert und aufgerufen. Um das OCP einzuhalten und beispielsweise Event Listener zu nutzen oder den Aufruf von sinnlosen Anweisungen zu vermeiden, müsste die bestehende Implementierung in der Klasse *Console* oder der *Main*-Methode angepasst werden.

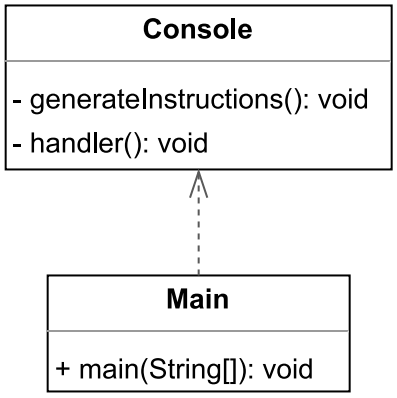

Abbildung 9: UML Diagramm der Klasse *Console*

Das Open-Closed-Prinzip könnte hier angewendet werden, indem man ein *ProgramFlow* Interface nutzt. Dadurch ist es möglich, alternative Implementierungen als Umsetzung des *ProgramFlow* Interfaces zu erstellen, ohne dass man die bestehende *Console* Klasse ändern muss. Mit dieser Methode kann der Programmfluss flexibel erweitert werden, ohne dass man den vorhandenen Code ändern muss.

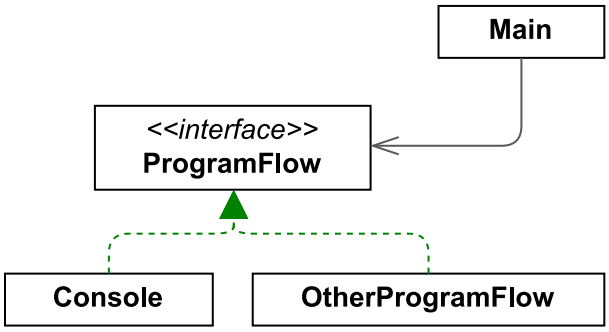

Abbildung 10: UML Diagramm für die Umsetzung des SRP mittels *ProgramFlow* Interface

Dependency-Inversion-Principle (DIP))

Positiv-Beispiel

Das Dependency Inversion Principle wurde bei der Implementierung der Klasse *TimePrediction* angewendet, um zu verhindern, dass diese von den

Details eines bestimmten Weges abhängig ist. Durch die Verwendung des *Path* Interfaces ist die Detailimplementierung *Track* von der Abstraktion *Path* entkoppelt. Das bedeutet, dass für verschiedene Implementierungen von *Path* eine benötigte Zeit vorhergesagt werden kann, ohne Änderungen an der Klasse *TimePrediction* vornehmen zu müssen. Dies ermöglicht eine flexible Gestaltung der Software mit reduzierten Abhängigkeiten von konkreten Implementierungen.

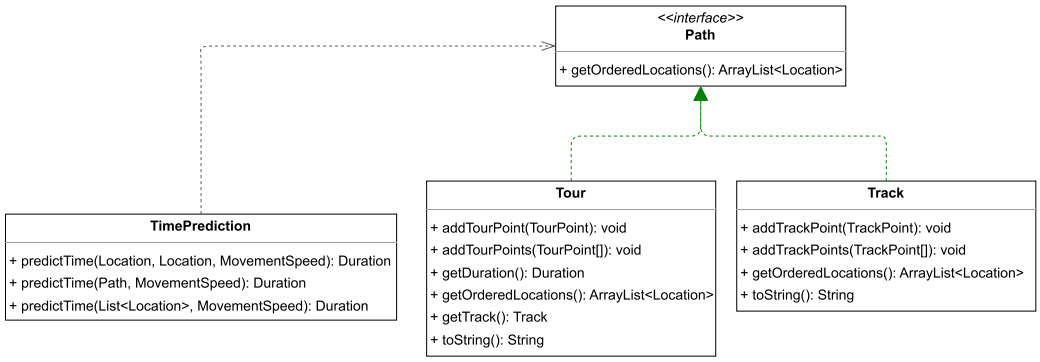

Abbildung 11: Dependency Inversion bei der Zeitvorhersagen von verschiedenen Arten von Wegen

Negativ-Beispiel

Das Dependency-Inversion-Principle wird beim Zugriff auf die Klasse *GPXToTour* verletzt, welche aus GPX Dateien ein Tour Objekt generiert. Sie ist eine Implementation des *XMLParser* Interfaces. Die Klasse *ReadPath* verwendet aber diese konkrete Implementierung anstelle des Interfaces, da beim Interface nicht klar ist welcher genaue Datentyp zurückgegeben werden soll. Da hier eine Abhängigkeit auf eine konkrete Implementierung anstelle des abstrakten Interfaces besteht ist das Dependency Inversion Principle verletzt.

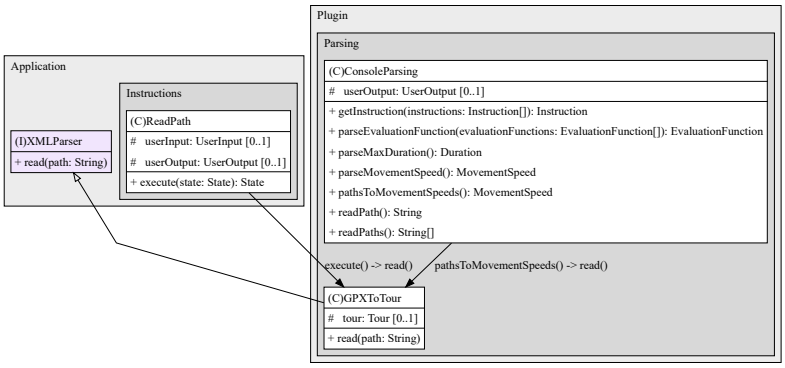

Abbildung 12: Abhängigkeiten auf die Klasse *GPXToTour*

Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

Die Abbildung zeigt das Positiv-Beispiel für geringe Kopplung. Die verschiedenen Klassen im Package *Instructions* implementieren die Abläufe verschiedener Anwendungsfälle. Dabei müssen Entscheidungen von Anwendern während der Nutzung berücksichtigt werden. Die Klasse *ConsoleParsing* nimmt Nutzereingaben von der Konsole entgegen. Das Interface *UserInput* entkoppelt die *Instructions* von den Benutzereingaben über die Konsole. Damit wird eine direkte Abhängigkeit der Anwendungslogik vermieden. Das erleichtert eine Änderung auf eine nicht-Konsolen-Applikation, was einen Vorteil der darstellt.

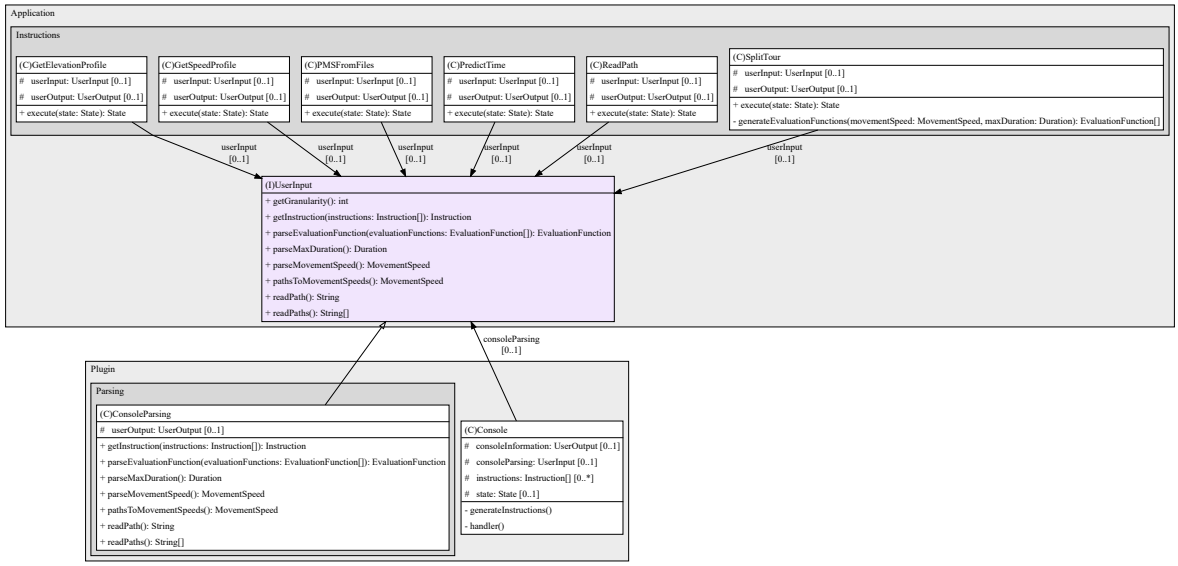

Abbildung 13: UML Diagramm des Interfaces *UserInput*

Negativ-Beispiel

Die Abbildung zeigt das Negativ-Beispiel für geringe Kopplung. Die Klasse *Hillclimbing* implementiert eine Annäherung[2] an eine optimale Lösung zur Mitnahme von wichtigen Wegpunkten, die eine Abweichung vom eigentlichen Wege erfordern.

Die Klasse *Detours* stellt alle in Betracht zu ziehenden Umwege dar. Die Klasse *Representation* stellt in Kombination mit Detours eine Lösung des Problems in Form eines Bitstrings dar. Eine *EvaluationFunction* bewertet Lösungen des Problems. *MovementSpeed* stellt die Geschwindigkeit dar, mit der sich auf dem Weg bewegt wird.
Die Klasse *Hillclimbing* besitzt zwar eine geringe Kopplung zu *Evaluation-Function*, *MovementSpeed* und den zugrundeliegenden Wegpunkten über das *Location* Interface. Allerdings besteht eine starke Kopplung zwischen der *Hillclimbing* Klasse und sowohl der Repräsentation der Lösungen und den mögliche Umwegen (Realisiert in den Klassen *Representation* und *Detours*), da die Klasse *Hillclimbing* von der Detailumsetzung der Umwege und insbesondere der Darstellung von Lösungen abhängt.

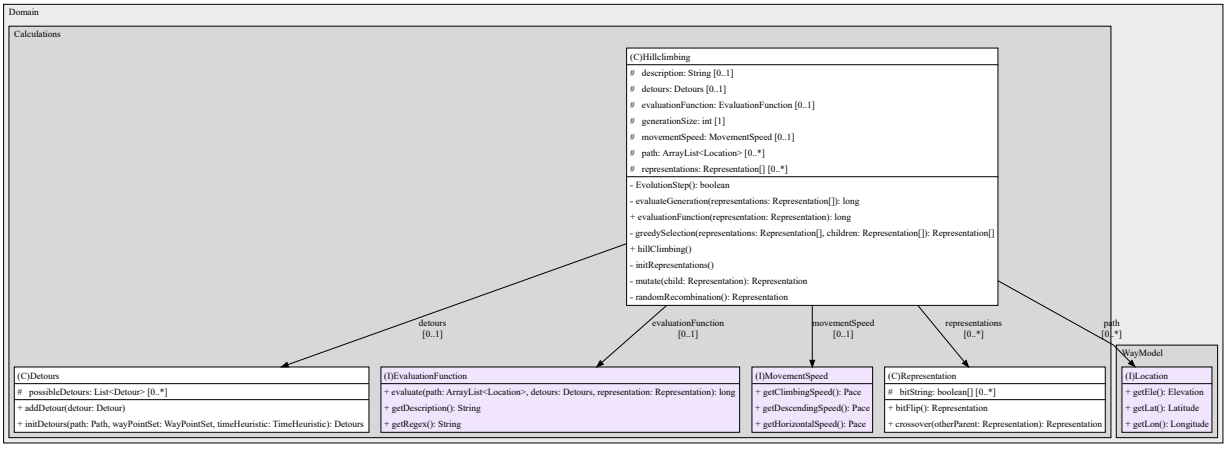

Abbildung 14: UML Diagramm der Klasse *Hillclimbing*

Die Kopplungen könnten gelöst werden, indem zwischen *Detours* und *Hill-*

[2]Insbesondere bei kleinen Datenmengen ist auch ein Erreichen möglich. Dies kann aber aufgrund der Komplexität des Problems (oder möglicherweise meinem fehlenden Verständnis) nicht garantiert werden

climbing sowie zwischen *Representation* und *Hillclimbing* jeweils ein Interface genutzt würde. Diese Interfaces müssten dann auch in die *EvaluationFunction* übergeben werden, da auch diese eine hohe Kopplung zu den beiden Klassen hat. Mithilfe dieser Umsetzung könnte sowohl eine neue Repräsentation für Lösungen sowie eine neue Umsetzung möglicher Lösungsbestandteilen aufgrund der geringeren Kopplung implementiert werden.
Dies würde allerdings zu einigen Problemen führen. Beide Klassen nur miteinander sinnvoll von einer Bewertungsfunktion auswertbar sind, da beide Bestandteil einer Lösung sind und damit eine hohe Kohäsion haben. Sinnvoller wäre wohl, das Prinzip der hohen Kohäsion anzuwenden, und die so erstellte Lösungskombination gering zu koppeln.

Analyse GRASP: Hohe Kohäsion

Die Klasse *Trackpoint* referenziert eine Länge, eine Breite und eine Höhe über dem Meeresspiegel. Als Gesamtheit repräsentiert ein *Trackpoint* also einen präzisen Ort auf der Erde. Die Kohäsion ist also sehr hoch, da genau diese Werte zusammen ein wohldefinierten Punkt im dreidimensionalen Raum abbilden.

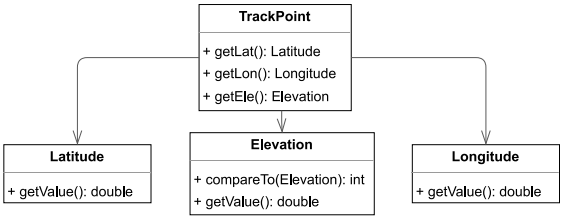

Abbildung 15: UML Diagramm der Klasse *Hillclimbing*

Don't Repeat Yourself (DRY)

Im Commit 8ffd648 wird der duplizierte Code aus den Klassen *ElevationProfile* und *SpeedProfile* aufgelöst.
In beiden Klassen werden Diagramme berechnet. Es gibt Berechnungen, die unabhängig von dem Inhalt des Profils durchgeführt werden müssen. Ein Beispiel hierfür ist die Methode *normalize*, welche den Wertebereich der Profile auf Zahlen zwischen 0 und 1 normiert.

Listing 1: Sich wiederholender Code vor dem Commit

```
1
2 // SpeedProfile.java (Zeile 39)
3
4 speeds = normalize(speeds,min,max);
5
6 // (Zeile 50-58)
7
8 private List<Double> normalize(List<Double> list,double
      min, double max){
9           double diff = max - min;
10          for (int i = 0; i < list.size(); i++) {
11                  double val = list.get(i);
12                  double normalizedVal = (val - min) /
                        diff;
13                  list.set(i,normalizedVal);
14          }
15          return list;
16 }
17
18 // ElevationProfile.java (Zeile 37)
19
20 heights = normalize(heights,min,max);
21
22 // (Zeile 48-56)
23
24 private List<Double> normalize(List<Double> list,double
      min, double max){
25          double diff = max - min;
26          for (int i = 0; i < list.size(); i++) {
27                  double val = list.get(i);
28                  double normalizedVal = (val - min) /
                        diff;
29                  list.set(i,normalizedVal);
30          }
31          return list;
32 }
```

Listing 2: Angewandtes DRY-Prinzip nach dem Commit

```
1
2 // SpeedProfile.java (Zeile 39)
3
4 speeds = ProfileCalculation.normalize(speeds,min,max);
5
6 // ElevationProfile.java (Zeile 37)
7
8 heights = ProfileCalculation.normalize(heights.stream()
      .map(e->e.getValue()).toList(), min.getValue(), max.
      getValue());
9
10 // ProfileCalculation.java (Zeile 6-14)
11
12 public static List<Double> normalize(List<Double> list,
       double min, double max){
13         double diff = max - min;
14         for (int i = 0; i < list.size(); i++) {
15                 double val = list.get(i);
16                 double normalizedVal = (val - min) /
                       diff;
17                 list.set(i,normalizedVal);
18         }
19         return list;
20 }
```

Kapitel 5: Unit Tests

10 Unit Tests

Unit Test	Beschreibung
ElevationGainTest#addElevation	Test auf korrekte Aufsummierung von Auf- und Abstiegen
ElevationGainTest#getManhattenNorm	Test auf korrekte Berechnung der Manhattennorm von Auf- und Abstieg
DistanceCalculatorTest#calc3dDistance	Test auf ausreichend genaue Berechnung von Distanzen im dreidimensionalen Raum
DistanceCalculatorTest#calcElevationGain	Test auf korrekte Berechnung der Höhendifferenz von mehreren Punkten im GPX Track
ProfileCalculationTest#split	Test auf korrekte Aufteilung von Wegpunkten in Wertegruppen für Profile
ProfileCalculationTest#normalize	Test auf korrekte Normalisierung von Werten auf winen Wertebereich zwischen 0 und 1
SpeedHeuristicsTest#calculateTime	Test auf die korrekte Berechnung der benötigten Zeit einer Tour
ElevationProfileTest#getProfile	Test auf korrekte Erstellung einer einem Höhenprofil entsprechenden Matrix
SpeedCalculatorTest#predictPMSSingle	Test auf korrekte Erstellung eines Personal Movement Speeds (PMS) aus einer bestrittenen Tour
SpeedHeuristicsTest#getClimbingHeuristic	Test auf ausreichend genaue Berechnung der Steigungsgeschwindigkeit

ATRIP: Automatic

Die Unit Tests sind beliebig oft wiederholbar und unabhängig voneinander ausführbar. Durch die Nutzung des JUnit Test Frameworks können sie mithilfe der IDE einfach gestartet werden. Mithilfe der *IntelliJ IDEA* beispielsweise können per rechtem Mausklick auf das Verzeichnis *src/test* und Ausführung

der Option *Run 'Tests in test'* alle Tests automatisch ausgeführt werden. Nach der Ausführung erscheint eine Übersicht über alle durchgeführten und erfolgreichen bzw. fehlgeschlagenen Tests.

ATRIP: Thorough

Positiv-Beispiel

Im Test *normalize* der Klasse *ProfileCalculationTest* wird auf korrekte Normalisierung von Werten auf den Wertebereich von 0 bis 1 getestet. Der Test ist thorough, da der Test als ParameterizedTest mit der zugehörigen MethodSource *normalizeValues* viele Fälle abgedeckt.

```
1
2  @ParameterizedTest
3  @MethodSource("normalizeValues")
4  void normalize(Double[] input, Double[] expected){
5          List<Double> list = new ArrayList<>(Arrays.
               asList(input));
6          list = ProfileCalculation.normalize(list);
7          assertEquals(Arrays.asList(expected), list.
               stream().toList());
8  }
9
10 private static List<Arguments> normalizeValues() {
11         return List.of(
12                 Arguments.of(new Double[]{4.0,4.0,4.0},
                       new Double[]{1.0,1.0,1.0}),
13                 Arguments.of(new Double
                       []{-10.0,0.0,10.0}, new Double
                       []{0.0,0.5,1.0}),
14                 Arguments.of(new Double
                       []{2.0,3.0,3.0,4.0}, new Double
                       []{0.0,0.5,0.5,1.0}),
15                 Arguments.of(new Double
                       []{0.0,3.0,6.0,2.0,8.0,9.0,10.0,2.0,3.0,4.0,5.0,6.0},
                        new Double
                       []{0.0,0.3,0.6,0.2,0.8,0.9,1.0,0.2,0.3,0.4,0.5,0.6})
                       ,
```

```
16                  Arguments.of(new Double[]{4.0}, new
                        Double[]{1.0}),
17                  Arguments.of(new Double[]{}, new Double
                        []{}),
18                  Arguments.of(new Double
                        []{9000.0,-500.0, 0.0}, new Double
                        []{1.0,0.0,500.0/9500}),
19                  Arguments.of(new Double[]{60.75,
                        105.75, 240.75}, new Double
                        []{0.0,0.25,1.0})
20          );
21  }
```

Negativ-Beispiel

Das Negativ-Beispiel zu thorough Test Code ist der Test *calc2dDistance* der Klasse *DistanceCalculatorTest*. Hier wird die berechnete Distanz zweier Punkte sowie eines Tracks (ohne Berücksichtigung der Höhe) getestet. Der Test ist nicht thorough, da lediglich ein einziger Fall pro Überladung der Methode getestet wird, und keine besonderen Fälle vorkommen.

```
1
2  void calc2dDistance() {
3           Distance distanceToFirstHut2D =
                DistanceCalculator.calc2dDistance(
                mountainTrack.getOrderedLocations().get(0),
                mountainTrack.getOrderedLocations().get(1));
4           assertEquals(308, distanceToFirstHut2D.getValue
                (), 1);
5
6           Distance distanceOfWholeTrack2D =
                DistanceCalculator.calc2dDistance(
                mountainTrack);
7           assertEquals(7795, distanceOfWholeTrack2D.
                getValue(), 10);
8   }
```

ATRIP: Professional

Positivbeispiel

Im Test *split* der Klasse *ProfileCalculationTest* wird auf korrektes Aufteilen von Werten für die Erstellung von Profilen getestet. Die zugrundeliegende Methode berechnet also, wie viele tatsächliche Werte in einem x-Wert gebündelt werden. Der Test ist professionell, da der Test als Parameterized-Test kurz und übersichtlich ist, keine Code Smells enthält und ausführlich relevante Testfälle abdeckt.

```
1  private static List<Arguments> splitValues() {
2    return List.of(
3      Arguments.of(25, 0, new int[]{}),
4      Arguments.of(25, 1, new int[]{25}),
5      Arguments.of(25, 2, new int[]{13,12}),
6      Arguments.of(25, 3, new int[]{9,8,8}),
7      Arguments.of(25, 4, new int[]{7,6,6,6}),
8      Arguments.of(0, 5, new int[]{0,0,0,0,0}),
9      Arguments.of(3, 5, new int[]{1,1,1,0,0}),
10     Arguments.of(Integer.MAX_VALUE, 4, new int[]{
           Integer.MAX_VALUE/4 +1,Integer.MAX_VALUE/4 +1,
           Integer.MAX_VALUE/4 +1,Integer.MAX_VALUE/4}),
11     Arguments.of(-3, 5, new int[]{}),
12     Arguments.of(42, -5, new int[]{})
13   );
14 }
15
16 @ParameterizedTest
17 @MethodSource("splitValues")
18 void split(int pool, int sections, int[] expected ){
19   int[] result = ProfileCalculation.split(pool,sections
       );
20   assertArrayEquals(expected, result);
21 }
```

Negativbeispiel

Das Negativ-Beispiel für professionelle Tests liefert der Test *calcElevationGain* der Klasse *DistanceCalculatorTest*. Der Test testet auf korrekte Berechnung von Höhendifferenzen zwei oder mehreren Punkten, die als Track gespeichert sind.
Bei dem Test wird das SRP verletzt, da mehrere Überladungen der Methode im einem Test geprüft werden. Bei fehlgeschlagenen Assertions ist zudem nicht leicht ersichtlich, wo genau der Fehler auftritt. Die zu testenden Punkte werden aus hardcoded Stellen einer Liste geholt. Das ist beim Lesen nicht nachvollziehbar, da kaum nachvollzogen werden kann welche Werte hier warum getestet werden. Es besteht viel duplizierter Code, der durch Methoden-Extraktion behoben werden könnte. Außerdem werden viele (nach Methoden Extraktion überflüssige) Variablen deklariert, die keine treffenden Namen haben (In Z.13 handelt es sich um wholeTrack nicht um eine Räpresentation des gesamten Tracks, sondern lediglich um die Höhendifferenz des Selbigen).
All dies sind Merkmale von Unprofessioneller Programmierung und folglich eines unprofessionellen Tests.

```
1
2 @Test
3 void calcElevationGain() {
4   ArrayList<Location> locations = mountainTrack.
        getOrderedLocations();
5   ElevationGain uphillSection = DistanceCalculator.
        calcElevationGain(locations.get(1), locations.get
        (2));
6   assertEquals(569, uphillSection.getUp(), 1);
7   assertEquals(0, uphillSection.getDown(), 1);
8
9   ElevationGain downhillSection = DistanceCalculator.
        calcElevationGain(locations.get(4), locations.get
        (5));
10  assertEquals(0, downhillSection.getUp(), 1);
11  assertEquals(539, downhillSection.getDown(), 1);
12
13  ElevationGain wholeTrack = DistanceCalculator.
        calcElevationGain(mountainTrack);
```

```
14    assertEquals(1346, wholeTrack.getUp(), 1);
15    assertEquals(1493, wholeTrack.getDown(), 1);
16 }
```

Code Coverage

Es werden 69% der Klassen und 59% der Zeilen getestet. Dies wird als ausreichend für den aktuellen Stand des Projekts angesehen, da die Testabdeckung sich auf die relevanten Teile der Applikation fokussiert.

Da die sich Plugin-Schicht aufgrund der Anforderungen an des Projekts mit der Konsole umgesetzt ist, welche simpel zu implementieren und gut durch manuelle Ende-zu-Ende Test testbar ist, liegt kein Fokus auf deren Unit-Testabdeckung. Entsprechend liegt hier die Testabdeckung bei 50% der Klassen und lediglich 27% der Zeilen.
Die verschiedenen Anwendungsfälle der Anwendungsschicht wurden sehr ausführlich durch manuelle Ende-zu-Ende Tests getestet. Demnach wurden nur drei der Anwendungsfall Implementierungen von Unit-Test abgedeckt.

Die meisten Unit Test testen also den Domain Code, da er komplizierte und langlebige Berechnungen enthält. Hier liegt die Testabdeckung bei 93% der Klassen und 87% der Zeilen, mit der (experimentellen) Klasse *SupplyEvaluation* als ungetestete Ausnahme[3]. Die restlichen Ungetesteten Codezeilen sind überwiegend einzeilige Methoden wie *getter*- und *toString* Methoden sowie die Behandlung offensichtlicher Fehler.

Fakes und Mocks

1. Mockobjekt: DirectWayHeuristik

Im Test *initDetours* der Klasse *DetoursTest* wird die korrekte Erzeugung von *Detours* getestet. Die Klasse hängt maßgeblich von der Bestimmung der Zeit

[3]Da die Zeilen an reinem Quellcode zum Stand der Dokumentation bei über 2300 liegen und das Projekt alleine implementiert wurde, wurde sich entschieden die Fortsetzung der komplizierten Bewertungsfunktion auf nach den Klausuren zu legen. Nichtsdestotrotz können Touren anhand von Übernachtungsorten geteilt werden. Bei der Nutzung sei aber gewarnt dass die Bewertung vergleichsweise wenig Rücksicht auf die gewünschte Gehzeit gibt um eventuell ein paar Tage zu sparen

einzelne Umwege (Detour) ab, welche über eine Implementierung des Interfaces *TimeHeuristic* berechnet wird.

Es wurde ein Mockobjekt für die Bestimmung der Zeit für eine einzelne Detour 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 (unbekannte) Strecke zu bewegen.

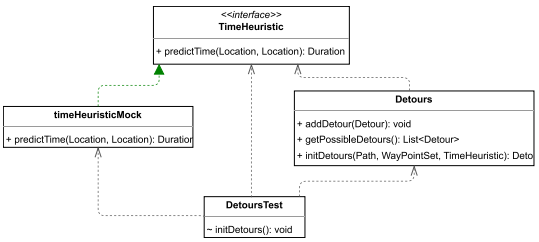

Abbildung 16: UML Diagramm des Mocks für *DetoursTests*

2. Mockobjekt: EvaluationFunction

Beim Test *evaluatinFunction* in der Klasse *HillClimbingTest* wurde beim Test des Hillclimbing-Algorithmus die Bewertung der einzelnen Lösungen des Algorithmus gemockt.

Die so erstellte Bewertungsfunktion bewertet alle Lösungen bis auf eine gleich gut. Besser bewertetf wird die, die gefunden werden soll. Dies führt dazu, dass diese Lösung als Lokales (und globales) Optimum beim Hillclimbing-Algorithmus herauskommen sollte, da sie das einzige existierende Optimum ist.

Ein Test über eine Praxisnahe Bewertungsfunktion wäre sehr umständlich, da man ein Beispiel mit möglichst gleichmäßigen Gradienten finden müsste, um ein Steckenbleiben in einem unvorhergesehenen lokalen Optimum zu vermeiden. Zudem ist das Mock-Objekt sinnvoll, da die Bewertungsfunktion in anderen Tests abgedeckt ist und sich die Aufgabe der Klasse und des Tests *Hillclimbing* auf das korrekte finden Lokaler Optima beschränkt.

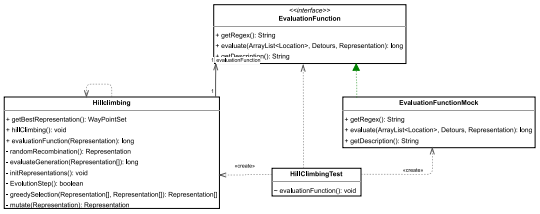

Abbildung 17: UML Diagramm des Mocks für eine Bewertungsfunktion

Kapitel 6: Domain Driven Design

Ubiquitous Language

Sport

Bedeutung Eine Ansammlung an Sportarten, die das *Movementspeed* Interface implementieren. Eine Sportart kann ausgeführt werden, wenn man einem GPX-Weg folgt. Eine Sportart beeinflusst die erwartete Zeit, die beim Folgen des Weges benötigt wird.

Begründung Eine Sportart ist Bestandteil der Ubiquitous Language, da sich alle Stakeholder vorstellen können, was die groben Geschwindigkeiten sind, die üblicherweise in den jeweiligen Sportarten erreicht werden können.

Track

Bedeutung Eine Chronologische Menge and Orten, die zusammen einen Weg ergeben.

Begründung Domänen-Experten sprechen üblicherweise von GPX-Tracks, wenn sie vom GPX-äquivalent eines Weges sprechen. Damit Entwickler und Domänen-Experten hier unmissverständlich über die jeweils vorliegenden Informationen sprechen können, ist Track Bestandteil der Ubiquitous Language.

Hillclimbing

Bedeutung *Hillclimbing* ist ein einfaches, heuristisches Optimierungsverfahren zum Finden lokaler Maxima.

Begründung *Hillclimbing* ist Bestandteil der Ubiquitous Language, damit Domänenexperten (hier Optimierungsexperten) und Entwickler präzise über komplizierte Optimierungsverfahren kommunizieren können, wenn bestehende Verfahren verstanden oder neue, bessere Verfahren implementiert werden sollen.

ElevationProfile

Bedeutung Ein Höhenprofil ist ein Zweidimensionaler Schnitt einer Strecke, der die Höhe an den jeweiligen Positionen darstellt. In der Regel werden diese überhöht dargestellt.

Begründung Ein Höhenprofil ist Bestandteil der Ubiquitous Language, da der Begriff bei Domänenexperten etabliert ist und eine technischere Bezeichnung, etwa *exaggeratedAltitudeAtDistanceFigure*, schwer treffend zu formulieren ist.

Entities

Die Klasse *Hillclimbing* verbindet eine Anzahl an Lösungen für ein spezielles Problem. Bei der Erstellung der Entity sind die vorgeschlagenen Lösungen noch trivial, mit der Lebenszeit der Entity verbessern sich die Qualitäten der Lösungen. Eine Instanz der Klasse *Hillclimbing* stellt einen konvergierten Lösungsvorschlag dar. Mehrere gleiche Hillclimbing Objekte wären trotzdem einzeln zu betrachten, da sie lediglich aussagen würden, dass mehrere Lösungsversuche zur selben Lösung konvergiert sind. Die Eindeutigkeit ist implizit über den Hashwertes des Objekts in Java umgesetzt. Somit ist Hillclimbing eine Entity.

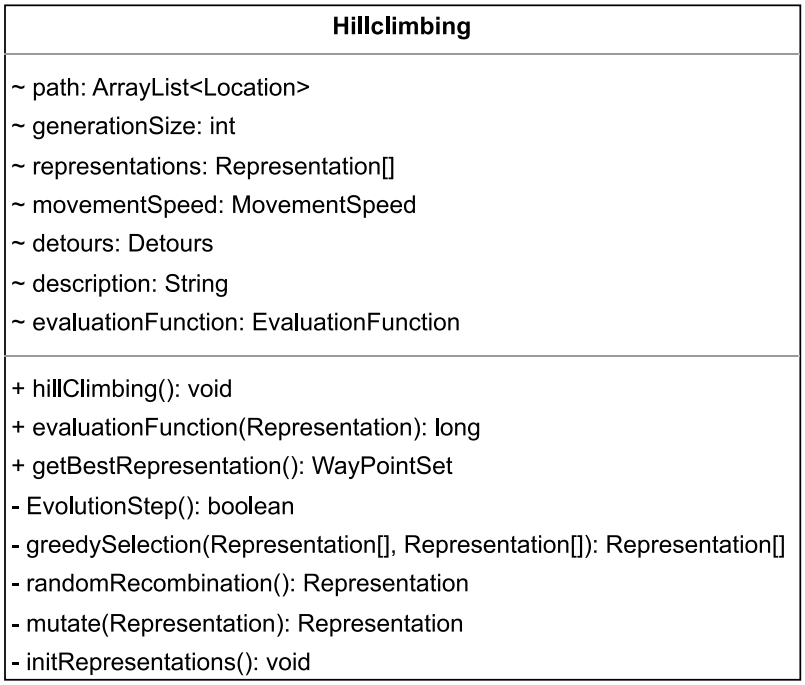

Abbildung 18: UML der Entity *Hillclimbing*

Value Objects

Die Klasse *Elevation* stellt den Wert einer Höhe über dem Meeresspiegel dar. Die Klasse ist immutable und bei Erstellung wird geprüft, ob sich der Wert in einem auf der Erde sinnvollen Rahmen bewegt (-500 bis 9000). Auf eine Überschreibung der Hashfunktion oder der equals Methode wurde aufgrund von mangelnder Notwendigkeit verzichtet.

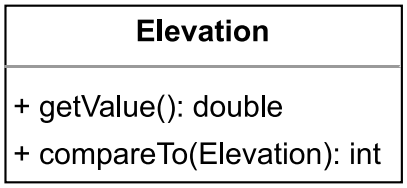

Abbildung 19: UML des Value Objects *Elevation*

Repositories

Das Interface *XMLGenerator* bietet dem Domain Code die Möglichkeit Tracks persistent zu speichern. Durch die Kapselung durch das Interface ist das Anti-Corruption-Layer implementiert. Somit ist der *XMLGenerator* der Adapter zwischen den Tracks und der persistenten Datenspeicherung in Form von Dateien auf der Festplatte.

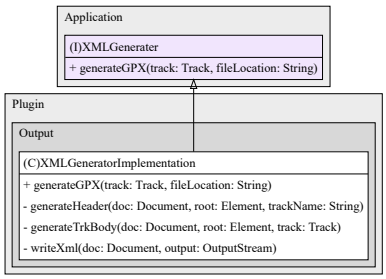

Abbildung 20: UML der Klasse *XMLGeneratorImplementation*

Aggregates

Ein Track bietet eine Zusammenfassung von *Trackpoint*s, die wiederum aus *Elevation*, *Latitude* und *Longitude* bestehen. Sie bilden eine eigene Einheit und werden üblicherweise auf diesem Level in GPX Dateien gespeichert und aus ihnen geladen.

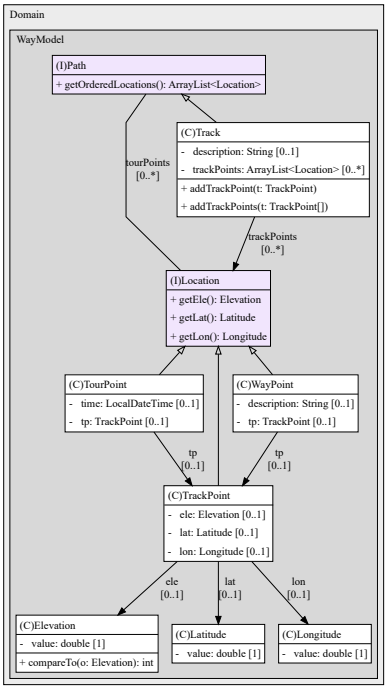

Abbildung 21: UML des Aggregates *Track*

Kapitel 7: Refactoring

Code Smells

Code Smell: Duplicated Code

Um Lösungen zu bewerten, muss bei Bitstringcodierung (welche in der Klasse *Representation* verwendet wird) gefiltert werden, welche Umwege tatsächlich genommen werden.
Vorheriger Zustand:

```
1  // in Klasse StayNightEvaluation
2
3  private double getWeightedOvershoot(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 }
```

```
1  // in Klasse SupplyEvaluation
2
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  }
```

Der in Commit 4039bc3 gewählte Lösungsweg lagert die duplizierte Funktionalität aus in die externe statische Klasse *EvaluationHelper*. In den einzelnen Bewertungsfunktionen wird die Funktionalität aus der Klasse *EvaluationHelper* aufgerufen.

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

Code Smell: Code Comments

Der Kommentar *// remove runaways* in der Methode *percentileandaverage* der Klasse SpeedHeuristics ist ein Anzeichen dafür, dass die Funktionalität

des Codes ohne den Kommentar nicht verständlich ist.

```
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    }
8 }
```

Der gewählte Lösungsweg ist die Auslagerung der Funktionalität in eine Methode mit sprechendem Namen.

```
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

Die Klasse *EvolutionaryDist* implementiert die evolutionäre Suche nach lokalen Optima nach dem Hillclimbing-Verfahren. Das Refactoring aus commit

a3b6bac ist die Umbenennung der Klasse *EvolutionaryDist* zu *Hillclimbing*, da der Name besser zum Inhalt der Klasse passt.

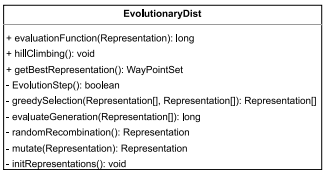

Abbildung 22: UML Diagramm der Klasse *EvolutionaryDist*

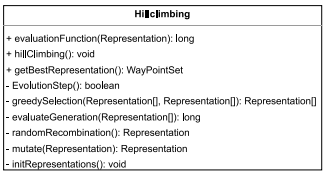

Abbildung 23: UML Diagramm der Klasse unter dem neuen Namen *Hill-Climbing*

2. Refactoring: Extract Method

Die Methode *generateGPX* zur Erstellung einer .gpx Datei aus gespeicherten Touren und Tracks besteht aus 35 Zeilen langem Spaghetticode. In den Commits 59f9045 und 7d57943 wurde mithilfe von des Refactorings *Extract Method* Struktur in den Code gebracht. Die Länge der einzelnen Methoden wird reduziert und das Verständnis des Codes wird durch Sprechenden Methodennamen verbessert.

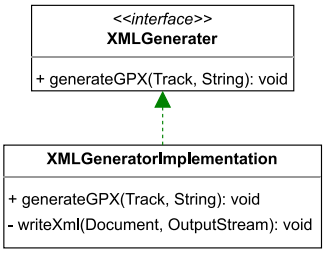

Abbildung 24: UML Diagramm der Klasse *XMLGeneratorImplementation* vor dem Refactoring

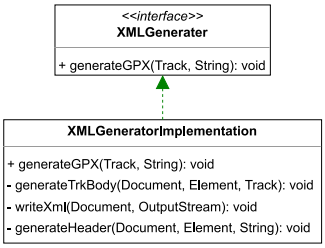

Abbildung 25: UML Diagramm der Klasse *XMLGeneratorImplementation* nach dem Refactoring mit zwei extracteden Methoden

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.

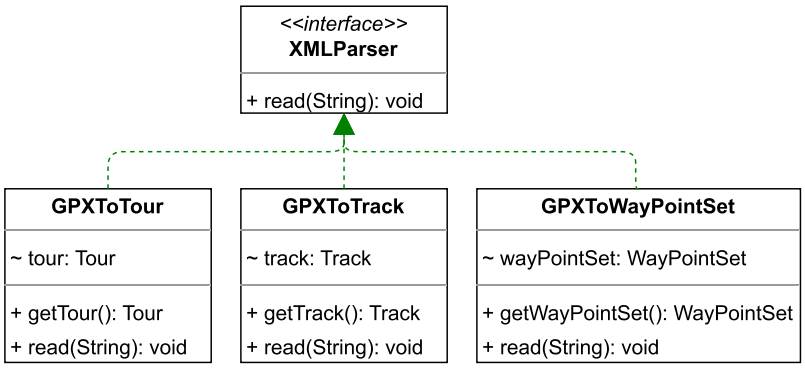

Abbildung 26: UML Diagramm der Fabrik für die Erzeugung von Objekten aus GPX Dateien

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.

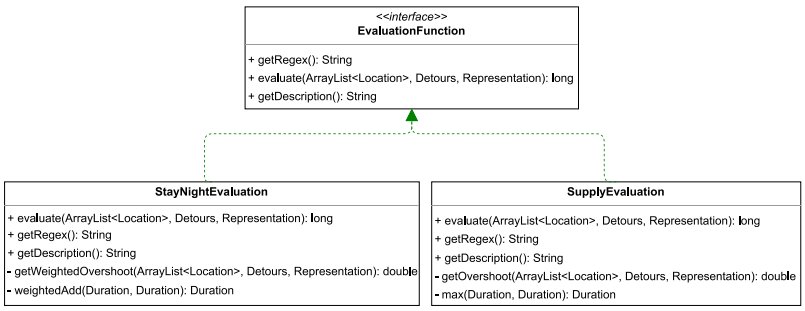

Abbildung 27: UML Diagramm der Strategie für die Bewertung von Umwegsoptimierungen