Programmentwurf GPX Rechner

Name: Philipp Reichert
Matrikelnummer:1758822
Abgabedatum: 28. Mai 2023

Kapitel 1:Einführung

Übersicht über die Applikation

Der GPX Rechner ist ein Programm zur Auswertung von GPX Dateien. Mit seiner Hilfe können Strecke, Dauer und Höhenprofil geplanter Touren vorhergesagt werden und bereits gegangene Touren ausgewertet werden auf Konsistenz der Geschwindigkeit und Geschwindigkeitsheuristiken für weitere Planungen.
Ein besonderes Feature bei geplanten Strecken ist die Aufteilung dieser anhand wichtiger Punkte die regelmäßig besucht werden müssen, wie etwa Unterkünfte oder Wasserquellen.
Die Benutzung der Applikation erfolgt indem man seine Dateien im Format dateiname.gpx in den Projektpfad kopiert. Mit starten der Main() Methode wird das Terminal gestartet und man wird aufgefordert einen Befehl einzugeben. Bei falscher Eingabe (bzw. "hilfe") werden alle verfügbaren Befehle aufgelistet.

Wie startet man die Applikation?

Wie testet man die Applikation?

Kapitel 2: Clean Architecture

Was ist Clean Architecture?

Clean Architecture ist der Aufbau von Anwendungen in verschiedenen Schichten, die nach innen hin immer beständiger werden. Äussere Schichten können dabei von inneren Schichten Abhängen, innere jedoch nicht von äusseren.

Analyse der Dependency Rule

Positiv-Beispiel: Dependency Rule

Das Positivbeispiel ist die Klasse Speedprofile. Die einzige Methode mit Abhängigkeiten ist

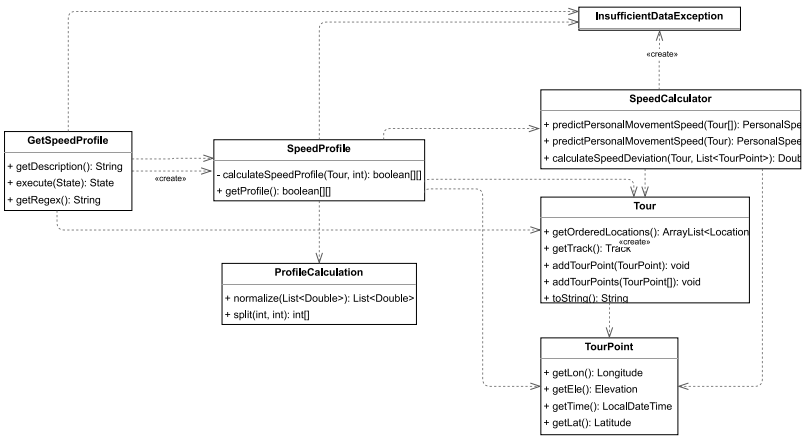

Figure 1: Abhängigkeiten der Klassse Speedprofile

Negativ-Beispiel: Dependency Rule

Schicht: Domain Code

Die Klasse DistanceCalculator ist dafür Zuständig verschiedene Distanzen zwischen Orten oder einer chronologischen Abfolgen von Orten im Sexalsystem zu berechnen. Die (zugegeben heuristische) Berechnung von Distanzen im Sexalsystem basieren auf grundlegenden Zusammenhängen und werden sich in absehbarer Zeit nicht ändern und sind Grundlegend für alle Auswertungen von Daten, welche im Sexalsystem abgespeichert sind.

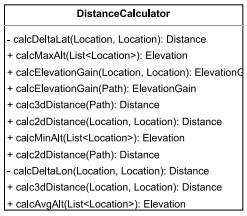

Figure 2: UML Diagramm der Klasse DistanceCalculator

Schicht: Plugins

Die Klasse ConsoleParsing ist dafür Zuständig bestimmte vorgegebene Werte von der Konsole zu lesen. Bei Eingabe unvorhergesehener Werte werden Vorschläge ausgegeben. Die Klasse stellt einen wesentlichen Bestandteil der Benutzerschnittstelle dar. Ein Austausch der Klasse durch beispielsweise ein tolles GUI mit Drop-Downs in Zukunft ist wahrscheinlich und lässt sich entsprechend einfach umsetzen.

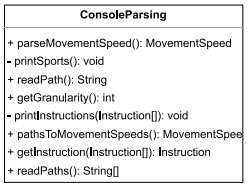

Figure 3: UML Diagramm der Klasse ConsoleParsing

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.

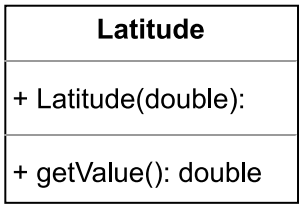

Figure 4: UML Diagramm der Klasse Latitude

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.

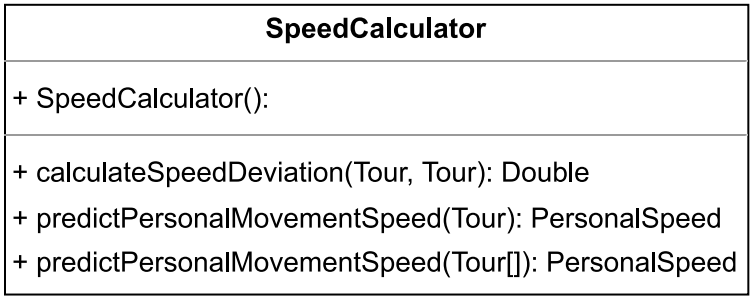

Figure 5: UML Diagramm der Klasse SpeedCalculator

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.

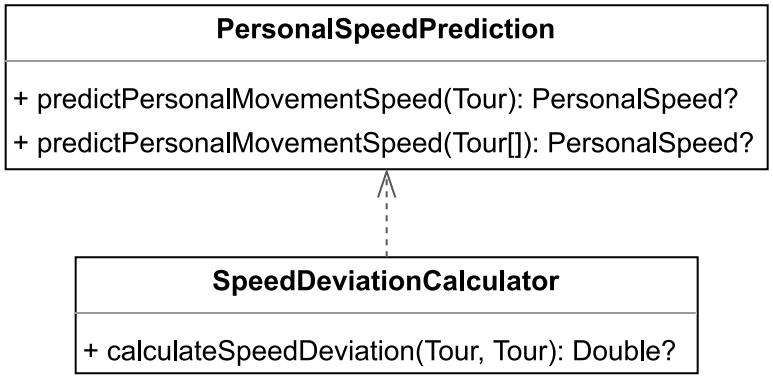

Figure 6: UML Diagramm mit Umsetzung des SRP

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.

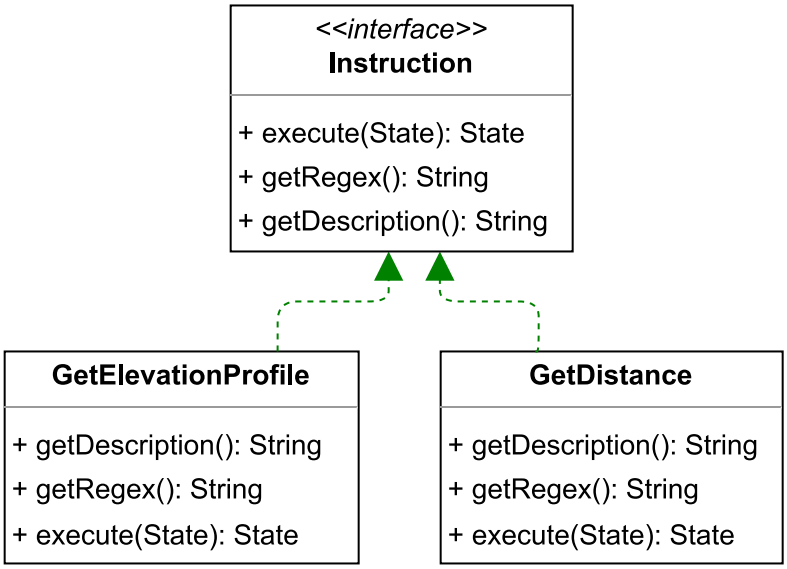

Figure 7: UML Diagramm des Instruction Interface mit 2 Implementierungen

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.

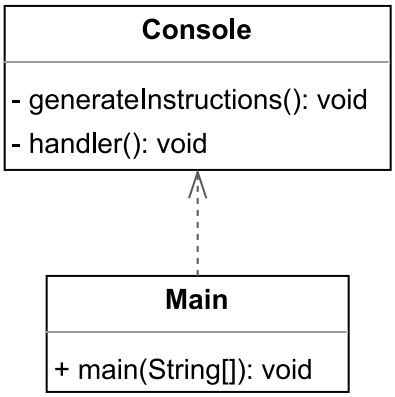

Figure 8: UML Diagramm der Klasse SpeedCalculator

Mithilfe eines ProgramFlow Interfaces könnte hier das OCP verwendet werden.

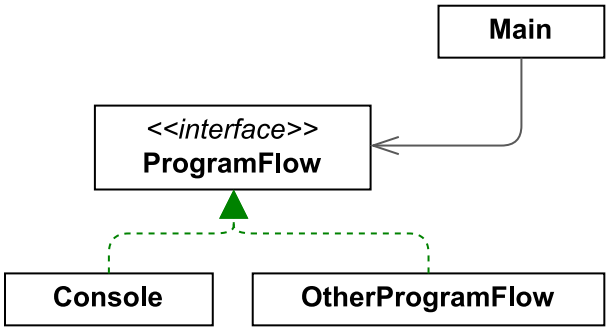

Figure 9: UML Diagramm mit Umsetzung des SRP

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.

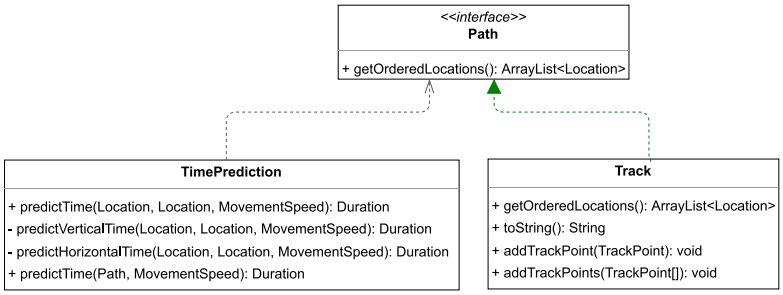

Figure 10: Dependency Inversion von Vorhersagen bei verschiedenen Pfaden

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.

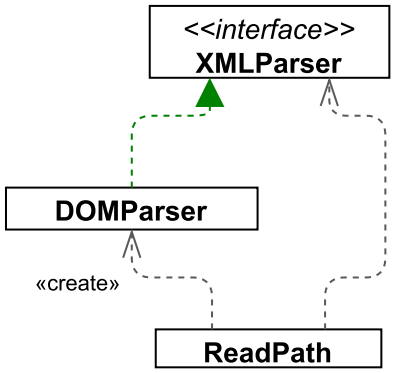

Figure 11: Keine Dependency Inversion beim Lesen eines Pfads

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	beschreibung
ElevationGainTest#getManhattenNorm	beschreibung
DistanceCalculatorTest#calc3dDistance	beschreibung
DistanceCalculatorTest#calcElevationGain	beschreibung
ProfileCalculationTest#split	beschreibung
ProfileCalculationTest#normalize	beschreibung
TimePredictionTest#predictTime	beschreibung
ElevationProfileTest#getProfile	beschreibung
SpeedCalculatorTest#predictPMSSingle	beschreibung
SpeedHeuristicsTest#getClimbingHeuristic	beschreibung