Dokumentation des Programmes in Runde 1 des 41. BWINF von Valentin Ahrend

Inhaltsangabe

Grundlegende Beschreibung des vorliegenden Programms

Verwendetes System / verwendete Programmiersprache

Das Programm basiert auf dem Flutter-Framework. Flutter ermöglicht es Apps (und Websites) für alle möglichen Betriebssysteme (und Web) zu entwickeln. Flutter basiert auf der open-source Programmiersprache Dart. In **Dart** wurde also das gesamte Projekt entwickelt. Obwohl Flutter Multi-Platform Apps ermöglicht, habe ich mich in diesem Fall auf **macOS** beschränkt. Bei dem ausführbaren Programm (in ../programm/bwinf41valentin_ahrend) handelt es sich also um eine **macOS-Applikation**.

Grundlegender Aufbau

Das Projekt (der Code) ist in mehrere Ordner unterteilt, die dem typischen Aufbau von Flutter-Applikationen entsprechen. Die für das Entwickeln ausschlaggebenden Ordner sind die Folgenden: `/lib`, `/assets` und die Datei `pubspec.yaml` (in ../code/bwinf41valentin_ahrend/...). Im Lib Ordner sind die dart-Dateien gespeichert. Im Assets Ordner befinden sich die asset-Dateien. Das sind einerseits die Beispiele aus Aufgabe 1, sowie die Beispiele aus Aufgabe 5. (Zudem mein Logo-Img). Die pubspec.yaml Datei ist das Manifest der Flutter-Applikation. Dort sind wichtige Daten, wie der Name oder die Version der App, sowie die Implementierungen (Dart und Flutter Packages) und die verwendeten Assets notiert.

```
  cupertino_icons: ^1.0.2 # Für Standard-Icons (macOS,iOS)
  url_launcher: ^6.1.6 # Für das Launchen von URLs (Öffnen des Browsers)
  image: ^3.2.0 # Für das Bearbeiten von Pixeln in Bildern, PixelArt. (für Aufgabe 2)
```

Das sind die genutzten Implementierungen für die App.

Der Ordner für diese Abgabe hat folgende Struktur:

```
--- bwinf_valentin_ahrend
----- dokumentation
------- DOKUMENTATION.md (diese Datei)
----- code
------- bwinf41valentin_ahrend
--------- lib
--------- ...
```

----- programm
------- bwinf41valentin_ahrend.app

Einzelne Aufgaben innerhalb des Programms

Einbettung einzelner Aufgaben

Die Starter-Klasse des Programms ist `/lib/main.dart`. Dort wird das UI-Interface initialisiert. Die UI von main.dart beinhaltet die linke Spalte, mit dem Selektor für die einzelnen Aufgaben. Die Flutter-UI ist in Widgets unterteilt. Die einzelne Aufgaben sind auch Widgets. Diese Widgets werden von main.dart eingebunden. Der Quellcode für die einzelnen Aufgaben beinhaltet also auch gleichzeitig dessen UI-Interface.

Aufgabe 1: Störung (lib/task1.dart)

Problem: Sätze aus "Alice im Wunderland" bestehen teilweise aus Lücken. Diese Lücken müssen durch ein Computerprogramm ergänzt werden. Zuerst habe ich folgendes festgelegt: Die Lücken entsprechen Wörtern aus dem Text "Alice im Wunderland". Diese Wörter werden ohne ihre Satzzeichen ersetzt. Die Wörter müssen also von ihren umschließenden Satzzeichen getrennt in die Lücken eingefügt werden. Im Beispiel: [das _ mir _ _ _ vor -> das kommt mir gar nicht richtig vor] werden schließlich auch die Satzzeichen weggelassen.
Lösungsidee: Für jedes feststehende Wort werden die möglichen Indexe des Wortes innerhalb des gesamten Textes festgelegt. Die möglichen Indexe der jeweiligen Wörter werden verlichen und es wird mit Berücksichtigung der Lücken überprüft, ob Indexe aufeinander folgen. Wenn diese Reihe / Kette auf jeweils einen der Indexe der festehenden Wörter zutrifft. So wurde der Satz im Text gefunden und kann nun ergänzt werden.

Um unabhängig von Satzzeichen und Formatierung zu sein, wird der Gesamttext sowie der Lückentext mittelt der Methode `textToWords()` in eine Liste aus Strings unterteilt. Diese Methode

```
List<String> textToWords(String text) {
    final lines = text.split("\n");
    List<String> words = [];
    for (var element in lines) {
      words.addAll(element.split(" ").map((e) => e.toLowerCase().replaceAll(RegExp(r"
[^a-z0-9üäöß_\n]"), "")).toList());
    }
    return words;
}
```

teilt den Text zunächst (druch `split("\n")`) in Zeilen ein. Anschließend wird jede Zeile durch `split(" ")` in Wörter aufgeteilt. Diese Wörter enthalten allerdings noch Satzzeichen (etc.). Um die Satzzeichen zu entfernen, werden, nachdem das Wort in kleinbuchstaben umtransformiert wurde, alle Characters des Strings, welche die Regular Expression `RegExp(r"[^a-z0-9üäöß_\n]"` inkludiert, gelöscht, sodass ein Wort ohne Satzzeichen entsteht. Die RegExp trifft auf alle Characters zu, die nicht im Bereich a-z, 0-9 liegen oder nicht den Zeichen üäöß_\n entsprechen.

Das Entscheidende an diesem Lösungsweg ist die `List<int> alleIndexe` Liste (l. 204-295). Als nächstes iteriert das Programm durch die Liste des Wörter des Lösungssatzes.

Falls es sich um ein Wort und nicht um eine Lücke handelt, so wird von der alleIndexe Liste Gebrauch gemacht.
Wenn es sich um das erste feststehende Wort handelt, so ist die Liste leer. Es werden alle Indexe also alle Stellen, wo das festehende Wort in der Liste der Wörter des Gesamttextes vorkommt, ermittelt (l. 219-245). Da die Liste leer ist, werden nun alle Indexe der Liste hinzugefügt.
Bei der nächsten Iteration und dem nächsten feststehenden Wort werden wieder die Indexe des aktuellen Wortes bestimmt. Diese werden allerdings nun mit den Werten aus der Liste (alleIndexe) verglichen. Falls ein Index 1 größer ist als der alte Index, so wird der alte Index durch den neue Index in der Liste ersetzt. Wenn dies nicht der Fall ist, wird der Index in der Liste mit -1 ersetzt.
In dem Fall, das es sich bei dem aktuellen Wort um eine Lücke handelt, werden alle Indexe (von alleIndexe) um 1 erhöht.

Beispiel:
Eingabe: das _ mir _ _ _ vor Der String "das" kommt an vielen verschiedenen Position in aliceWoerter vor.
Alle Positionen bzw. Indexe werden nun in alleIndexe gelistet. Z.B. an Index 100 und Index 500
Als nächstes kommt die Lücke _, hier werden alle vorherigen Indexe um 1 erhöht. -> Index 101 und Index 501. Der String "mir" wird nun gesucht. Er hat bspw. die Indexe 102 und 300 in aliceWoerter.
Vergleicht man nun die Liste (alleIndexe) mit den Indexen von "mir" so wird deutlich, dass der Index 102 auf den Index 101 (aus alleIndexe) folgt. Demnacht wird der Index 101 mit 102 aktualisiert, während der Index 501 verworfen wird.
Bei den nöchsten 3 Lücken erhöhen sich alleIndexe wieder um 3, d.h. Index 102 -> Index 105. Der String "vor" wird nun gesucht, falls dieser einen Index von 106 in aliceWoerter aufweist, so ist die Reihe komplett und die Endposition der Eingabe steht fest.

```
    List<int> alleIndexe = [];
    /// Mit einem For loop wird durch die woerter Liste iteriert.
    for (var i = 0; i < woerter.length; i++) {
      String eingabeWort = woerter[i];
      /// aliceWoerter wird nach eingabeWort durchsucht, falls das eingabeWort keine
Lücke ist
      if(eingabeWort != "_") {
        int indexWortInAlice = aliceWoerter.indexOf(eingabeWort);
        if(indexWortInAlice == -1){
          /// wenn indexWortInAlice == -1, so befindet sich das eingabeWort nicht in
dem Text
          /// demnach ist die Eingabe nicht korrekt, da sich jedes eingabeWort im Text
befinden muss.
          logResult("Eingabe ist nicht korrekt, da nicht alle sichtbaren Wörter sich
im Text befinden.");
          return;
        }
        /// ueberpruefe ob es das erste sichtbare Wort ist, also alleIndexe leer
ist...
        if(alleIndexe.isEmpty) {
          /// da aliceWoerter mehrmals das eingabeWort beinhaltet kann, und bei
indexOf nur der erste Wert
```

```
          /// als Index in der Liste ausgegeben wird, müssen alle anderen Indexe auch
gefunden bzw. überprüft
          /// werden.
          alleIndexe.add(indexWortInAlice);
          /// currentIndex ist der akutelle Wert des Indexes von eingabeWort in
aliceWoerter
          int currentIndex = indexWortInAlice;
          /// in dieser while-Scheleife werden alle Indexe von eingabeWort in
aliceWoerter festgestellt,
          /// wenn ein Index gefunden wurde, anfangs indexWortInAlice, so wird dieser
verwendet um den nächsten
          /// Index des eingabeWorts zu bestimmen. Der zweite Parameter von indexOf
gibt an, ab welchem Index die Suche
          /// nach dem nächsten Index startet. Der Start-Index liegt hier bei
currentIndex + 1, sodass nur ein neuer
          /// Index gefunden werden kann. Ist dieser neue Index -1, so wurden alle
Vorkommnisse von eingabeWort in
          /// aliceWoerter gefunden, und in alleIndexe aufgelistet.
          while(true) {
            int naechsterIndexWortInAlice = aliceWoerter.indexOf(eingabeWort,
currentIndex + 1);
            if(naechsterIndexWortInAlice == -1) {
              break;
            }
            alleIndexe.add(naechsterIndexWortInAlice);
            currentIndex = naechsterIndexWortInAlice;
          }
        }else{
          List<int> neueAlleIndexeProWort = [];
          neueAlleIndexeProWort.add(indexWortInAlice);
          /// copy (siehe oben)
          int currentIndex = indexWortInAlice;
          while(true) {
            int naechsterIndexWortInAlice = aliceWoerter.indexOf(eingabeWort,
currentIndex + 1);
            if(naechsterIndexWortInAlice == -1) {
              break;
            }
            neueAlleIndexeProWort.add(naechsterIndexWortInAlice);
            currentIndex = naechsterIndexWortInAlice;
          }
          /// nun wird überprüft, welche Indexe aus neueAlleIndexeProWort 1 größer als
ein Wert von alleIndexe sind, denn eine Wort folgt schließlich auf das nächste
(index+1)
          for (var i = 0; i < alleIndexe.length; i++) {
            if(alleIndexe[i] >= 0){
              bool neuerWert = false;
              for (var j = 0; j < neueAlleIndexeProWort.length; j++) {
                if(neueAlleIndexeProWort[j] == alleIndexe[i] + 1){
                  /// wenn der neue Index 1 größer ist als der alte Index (Beispiel
oben, 101 auf 102)
                  /// -> alleIndexe wird bei i, der neue Index zugeschrieben
```

```
                  alleIndexe[i] = neueAlleIndexeProWort[j];
                  neuerWert = true;
                  break;
                }
              }
              /// wenn nichts verändert wurde, ist dieser Index falsch und wird mit -1
ersetzt.
              if(!neuerWert){
                alleIndexe[i] = -1;
              }
            }
          }
        }
      }else{
        /// wenn Indexe bereits in alleIndexe eingetragen wurden, dann wird jedem
Index ein Wert von 1
        /// hinzugefügt, entsprechend der einzelnen Lücke.
        if(alleIndexe.isNotEmpty){
          for (var i = 0; i < alleIndexe.length; i++) {
            /// jedem Index in alleIndexe wird die Lücke (als +1) hinzugefügt.
            /// außer wenn der zugeordnete Index zu große für aliceWoerter ist oder
der Index mit -1 markiert wurde (also falsch ist)
            if(alleIndexe[i] + 1 >= aliceWoerter.length || alleIndexe[i] == -1){
              continue;
            }
            alleIndexe[i] = alleIndexe[i] + 1;
          }
        }
      }
    }
```

Nun werden alle Werte mit -1 aus der Liste gelöscht (l.301-306).

Die Liste der Wörter des Lösungssatzes wird von hinten nach vorne iteriert. Dabei wird für jeden ermittelten Endindex des Satzes innerhalb der Liste der Wörter des Gesamttextes ein Satz erstellt. Die Lösungssätze befinden sich schließlich in `List<String> valideLoesungen`:

```
    List<String> valideLoesungen = List.filled(valideIndexe.length, "");
    for (var i = woerter.length - 1; i >= 0; i--) {
      if(woerter[i] == "_"){
        for (var j = 0; j < valideIndexe.length; j++) {
          /// String der jeweiligen Lücke in woerter wird bestimmt, indem man
rückwärts durch die Liste der Eingabewörter iteriert.
          /// Der Index für die jeweilige Lücke beträgt nun valideIndexe[j] -
(woerter.length - i - 1)
          String? lueckeX = aliceWoerter[valideIndexe[j] - (woerter.length - i - 1)];
          /// Es werden gleichzeitig mehrere Lösungen bereitsgestellt, da es mehrere
korrekte Endindexe geben kann.
          /// trim() wird gecallt, um die Lösung ins optimale Format ohne überflüssige
Leerzeichen zu bringen.
          valideLoesungen[j] = "$lueckeX ${valideLoesungen[j].trim()}";
        }
```

```
      }else{
        for (var j = 0; j < valideIndexe.length; j++) {
          /// hier muss die Lücke nicht aus aliceWoerter erschlossen werden, da sich
das Wort bereits in der Eingabe befindet.
          valideLoesungen[j] = "${woerter[i]} ${valideLoesungen[j]}";
        }
      }
    }
```

Am Ende werden die validen Lösungen durch `join(",")` zu einem String verbunden und an die UI als `result` weitergegeben.

Beispiel (stoerung0.txt):

- Als Eingabe wird: "das _ mir _ _ _ vor" verwendet.
- Die Eingabe wird nun in eine Liste aus Strings aufgeteilt: ["das", "", *"mir",* "", ", "", "vor"]
- Nach der 1. Iteration (durch die Liste an Wörtern) ist folgende Liste enstatnden: [86, 291, 339, 478, 683, 720, 839, 841, 929, 970, 1044, 1307, 1355, 1410, 1438, 1453, 1482, 1730, 1783, 1908, 1936, 2043, 2125, 2462, 2507, 2534, 2564, 2826, 2879, 2882, 2988, 3118, 3230, 3233, 3289, 3307, 3423, 3425, 3549, 3580, 3657, 3705, 3756, 3800, 3880, 3948, 3989, 4005, 4173, 4177, 4271, 4445, 4555, 4589, 4840, 5172, 5492, 5554, 5635, 5656, 5796, 6000, 6034, 6045, 6075, 7634, 7912, 7953, 7981, 8064, 8334, 8337, 8449, 8736, 8746, 8823, 8859, 8989, 9162, 9281, 9284, 9294, 9311, 9326, 9375, 9389, 9424, 9486, 9507, 9543, 9664, 9765, 9792, 10016, ...]
- Nach der 2. Iteration wurde jeder Wert in der Liste um 1 erhöht, da es sich um eine Lücke handelt "_".
- Nach der 3. Iteration ("mir") sieht die Liste wie folgt aus: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 3707, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 10207, -1, ...]
- Nach der letzten Iteration ("vor") bleiben in der Liste nur -1 und 3711 übrig.
- Der Index von dem Wort "vor" im Gesamttext ist also 3711.
- D.h. der Lösungssatz besteht aus den Indexen: [3705, 3706, 3707 ,3708 ,3709, 3710, 3711].
- Der Satz wird mit den Indexen ergänzt. Damit entsteht der Satz: "das kommt mir gar nicht bekannt vor" Das Programm kann genutzt werden um eigenen Lückensätze oder alle anderen Beispielsätze zu testen.

Aufgabe 2: Verzinkt (lib/task2.dart)

Problem: Es soll die Kristallisierung von Keimen simuliert werden. Jeder Keim bzw. Kristall hat ein bestimmtes Wachstum in alle 4 Richtungen, eine Orientierung (Farbe), eine Spawn-Zeit (Zeit bis zum Spawn) und eine Position (x, y Koordinate). Kristalle dürfen sich nicht gegenseitig berühren oder überdecken.
Lösungsidee: Jeder Kristall bzw. Keim ist eine Klasse, welche die nötigen Informationen (Wachstum etc.) enthält. Eine sich nach einer bestimmten Zeit (Tick-Dauer) wiederholende Methode leitet bei jedem Keim das Wachstum ein. Ein Wachstum eines Kristalls ist dabei die Kreation eines neuen Kristalls, der anschließend auch weiter wächst und neue Kristalle spawnt.

Ich habe als erstes die bereits erwähnte Klasse (Crystal) konstruiert. Der Nutzer hat die Möglichkeit Grund-Kristalle oder Grund-Keime zu bestimmen. Der zentrale Bestandteil der Simulation ist die `List<List<int>> simulationMap`. Dies ist eine Liste, welche 500 Listen mit jeweils 500 Elementen hält. Sie fungiert wie ein Koordinatensystem mit jeweils 500 Elementen auf der x und y Achse. Die Position eines Kristalles ist in diesem Raster festgelegt. Dabei wird nur die Orientierung also der spätere Grau-Ton der Kristalls dem Raster hinzugefügt. Das Set `Set<Crystal> aktuelleKristalle` beinhaltet die aktuellen Kristalle, also die Kristalle, welche von der Simulation aktiv simuliert werden. Die vom Benutzer zu Beginn erstellten Keime sind auch Kristalle dieses Sets.
Wenn die Simulation gestartet wird, wird die `runTick(1)` Methode ausgeführt. Ein Tick entspricht einem Durchlauf der Simulation bzw. einem Ausführen der Methode und kann mit dem `tickDuration` Parameter zeitlich (also die Dauer) festgelegt werden.
Tick 0: Bei Tick 0 werden nur die vom Nutzer festgelegten Keime dargestellt und noch nicht simuliert. Die Simulation startet ab **Tick 1** und setzt sich solange fort, bis keine Kristalle mehr im Set `aktuelleKristalle` sind. In der `runTick` Methode wird durch das Set `aktuelleKristalle` iteriert und bei jedem Kristall wird die Methode `neueKristalle` ausgeführt:

```
      Set<'Crytal>  neueKirstalle =
          kristall.neueKristalle(simulationMap, tickNummer);
      if (neueKirstalle.length == 1 &&
          neueKirstalle.first.creationStatus == 4) {
        aktuelleKristalle.remove(kristall);
      } else {
        aktuelleKristalle.addAll(neueKirstalle);
      }
```

Jeder Kristall enthält die Variable `creationStatus`. Diese sagt aus, wie oft aus dem Kristall schon neue Kristalle hervorgegangen sind. Wenn der Kristall schon 4 Kristalle erzeugt hat also in alle Richtungen gewachsen ist, so wird dieser Kristall vom Set gelöscht. Die neu erzuegten Kristalle werden dem Set hinzugefügt. Die Methode `neueKristalle` funktioniert wie folgt:

```
 if (creationStatus == 4) {
      ausgabe.add(this);.
      return ausgabe;
 }
```

Es wird zunächst überprüft, ob der Kristall schon in alle Richtungen gewachsen ist (creationStatus == 4). Wenn ja wird der Kristall als Set zurückgegeben.

```
if (positionY > 0) {
      if (wachstumsGeschwindigkeiten[0] + startTickNummer <= currentTick) {
        /// wenn die Bedingung erfüllt ist, kann ein Kristall wachsen (oben)
        /// überprüfen, ob an dieser Stelle bereits ein Kristall exisitiert:
        if (simulationMap[positionY - 1][positionX] == -1) {
          /// frei
          /// in der simulationMap wird der Platz mit der Orientierung belegt
          simulationMap[positionY - 1][positionX] = orientierung;

          Crystal neuerKristall = Crystal(currentTick, orientierung, positionX,
              positionY - 1, wachstumsGeschwindigkeiten);
```

```
          ausgabe.add(neuerKristall);
        }else{
          if(simulationMap[positionY - 1][positionX] != orientierung){
            creationStatus += 1;
          }
        }
      }
    }
```

Als nächstes wird das "Wachsen" in alle 4 Richtungen probiert. Im Code oben handelt es sich um das Wachstum nach oben. Es wird zunächst überprüft, ob der Kristall, welcher durch das Wachstum eine neue Position im Raster hat, immernoch innerhalb des Rasters ist. Anschließend wird überprüft, ob ein Kristall (hier: nach oben) bei diesem Tick wachsen kann:
Jeder Kristall hat eine Liste mit der Anzahl an Ticks pro Richtung, welche für ein Wachstum erforderlich sind.
`wachstumsGeschwindigkeiten[0]` entspricht hier der Anzahl an Ticks, die erfolderlich sind, damit ein Kristall oben entsteht. Die Anzahl an Ticks (`startTickNummer`) bei der Erstellung des Kristalls wird als Grundwert genommen. D.h.
`wachstumsGeschwindigkeiten[0] + startTickNummer` entspricht dem Tick, ab dem das Wachstum möglich ist. Als nächstes wird überprüft, ob die Position, an der der neue Kristalle (der neu gewachsene Kristall) entstehen würde, in dem Raster (`simulationMap`) schon vergeben ist. Wenn nicht, wird an der Position der Orientierungs-Wert hinzugefügt. Nun wird der neue Kristall (`Crystal()`) erstellt und dem Set, das später ausgegeben wird, hinzugefügt. Nachdem alle Richtungen auf Wachstum überprüft wurden, werden ein Set mit neuen Kristallen zurückgegeben. Die neuen Kristalle haben hierbei die gleiche Orientierung und Wachstumsparamter, wie der ursprüngliche Kristall. Es ändern sich Position und die `startTickNummer` (zu `currentTick`). Wenn ein neuer Kristall erzeugt wurde oder wenn das Erstellen eines neuen Kristalls auf Grund von einem bereits dort existierenden Wert nicht möglich war, so wird der `creationStatus` um 1 erhöht (je nach Richtung, also maximal um 4).

Ein wichtiger Bestandteil der Simulation ist die Methode `sendToUI` :

```
img.Image image = img.Image.rgb(500, 500);
    for (var i = 0; i < simulationMap.length; i++) {
      for (var j = 0; j < simulationMap[i].length; j++) {
        final c = simulationMap[i][j];
        if (c == -1) {
          image.setPixelSafe(j, i, hexOfRGB(255, 255, 255));
        } else {
          image.setPixelSafe(j, i, hexOfRGB(c, c, c));
        }
      }
    }
```

Dort wird die `simulationMap` in ein Bild umgewandelt. Dieses Bild wird anschließend in die UI (mit `setState`) übertragen. Nach dem Druchlauf von `runTick()` wird die runTick Methode mit der Ticknummer + 1 als Paramter eingeleitet (Rekursion).

Ich habe ungefähr 30 zufällige Keime mit dem Programm erstellt und die Simulation ergibt ungefähr ein ähnliches Bild wie in der Aufgabenstellung. Das Programm kann frei für jede beispielhafte Simulation genutzt werden.

Aufgabe 5: Hüpfburg (lib/task5.dart)

Problem: Ein Parcours wird von zwei Spielern gleichzeitig absolviert. Können die beiden Spieler sich treffen und wenn ja nach welcher Abfolge von Sprüngen? Der Parcours besteht aus Feldern. Jedes Feld ist zu bestimmten anderen Felder verbunden. Somit kann ein Spieler von diesem Feld nur zu den anderen verbundenen Feldern hüfpen.
Lösungsidee: Um zu bestimmen, ob der Parcours sich beide Spieler treffen und wenn ja wie, wird der Parcours "simuliert". Jeder Spieler startet auf einem Spielfeld. Ein Feld entspricht der Klasse `Punkt`. Ein Spieler entspricht der Klasse `Runner`. Es gibt zwei Arten von Runnern: `Runner` des Typ 0 (Spieler 1) und `Runner` des Typ 1 (Spieler 2). Falls auf einem Feld (`Punkt`) zwei `Runner`, welche nicht den gleichen Typ haben, stehen, haben die Spieler sich getroffen und der Parcours wurde bewältigt. Jeder `Runner` wird gleichzeitig auf das nächste Feld verschoben. Wenn es mehrere Felder zur Auswahl gibt, dann wird pro möglichem Feld ein `Runner` erzeugt, sodass jeder mögliche Bewegungsablauf gleichzeitig existiert.

Zunächst habe ich die Klasse `Punkt` (also ein Feld) konstruiert. Diese hält ein Set an Verbindungen, eine id (Nummer des Punktes) und ein Set mit den aktuellen Runnern, die sich auf dem Punkt befinden. Die `test` Methode überprüft, ob sich in dem Set der aktuellen Runner des Punktes `Runner` unterschiedlicher Typen befinden. Wenn ja, werden diese `Runner` als Set ausgegeben (wenn nicht, wird nichts ausgegeben). Runner können dem Punkt hinzugefügt und wieder entfernt werden. Beim Hinzufügen wird dem Set ein Runner hinzugefügt. Beim Entfernen wird die `runnerAbmelden` Methode genutzt. Dort wird die Anzahl an Runnern, die sich auf dem Feld / Punkt befindenen, mit der Anzahl an Verbindungen des Feldes geleichgesetzt. D.h. es werden entweder Runner entfernt oder neu hinzugefügt. Wenn ein Runner neu hinzugefügt wird, nutzt er die Werte eines anderen Runners als Basis (siehe l. 496).

```
    int i = 0;
    for (Runner runner in aktuelleRunner) {
      int verbindung = verbindungen.elementAt(i);
      runner.gelaufeneFelder.add(runner.naechsterPunkt);
      runner.naechsterPunkt = verbindung;
      ausgabeRunners.add(Runner(runner.typ, runner.gelaufeneFelder,
runner.naechsterPunkt));
      i++;
    }
```

Im nächsten Abschnitt (siehe oben) wird den aktuellen Runnern ihre Verbindung zugeorndet und die Liste der gelaufenen Felder aktualsiert. Dem Set `ausgabeRunners` werden nun Kopien der aktuellen Runner hinzugefügt. Die Liste der aktuellen Runner wird nun geklärt, sodass neue Runner das Feld neu besetzen können.

Die Punkte des Parcours werden zu Beginn der Analyse (`analysieren()`) samt ihrer Verbindungen erstellt und in die Liste `aktuellePunkte` aufgenommen.

Die Klasse des Runners beinhaltet die Liste an gelaufenen Feldern und den Typ des Runners. Außerdem gibt die Variable naechsterPunkt an, welchem Punkt der Runner als nächstes zugeorndet wird.

Die Simulation wird von der rekrusiven `handleRunner` Methode gesteuert. Diese wiederholt sich solange, bis ein Runner-Paar gefunden wurde oder die Anzahl an gelaufenen Feldern größer ist als die Anzahl an Punkten zum Quadrat. Dieser Wert wurde als Maximalwert für die Weglänge bestimmt. Bei nicht rekursiven Wegen (also Wegen, bei

denen sich kein Feld wiederholt) entspricht die maximale Weglänge die Anzahl an Feldern. Wenn sich die Felder allerdings wiederholen können, erhöht sich diese Anzahl. Beispiel: Bei huepfburg1.txt treffen sich beide Spieler nur nach mehreren Durchläufen (durch den Parcours). Wenn Sasha auf Feld 1 und Mika auf Feld 2 startet, so gelingt es nach 121 Feldern, dass beide auf ein Feld kommen. Bei 17 Feldern sind 121 Felder weniger als der Maximalwert 17 * 17 (=289).

Die Methode `handleRunner` iteriert durch die Liste der aktuellen Runner und fügt dem Punkt (aus `aktuellePunkte`), welcher bei Runner als naechsterPunkt bestimmt ist, den Runner hinzu (`runnerAddieren`). Dann iteriert das Programm durch die aktuellen Punkte (`aktuellePunkte`). Bei jedem Punkt wird die `test()` Methode gerunnt. Es wird bei jedem Punkt überprüft, ob sich Runner mit unterschiedlichen Typen gelichzeitig dort befinden. Wenn dies bei keinem Punkt der Fall ist, werden alle Runner wieder von dem jeweiligen Punkt getrennt (`runnerAbmelden`) und die Methode wiederholt sich.

Am Ende wird das Runner-Paar, falls eines gefunden wurde, in die UI übertragen.

Beispiel: In huepfburg0.txt startet Sasha an Feld 1 und Mika an Feld 2. Zunächst wird die Liste der Punkte erstellt (Methode: `analysieren`):

```
[Punkt: 1, {18, 8, 4}, Punkt: 2, {3, 19}, Punkt: 3, {19, 6}, Punkt: 4, {}, Punkt: 5, {15, 12, 13}, Punkt: 6, {2, 12}, Punkt: 7, {5, 8}, Punkt: 8, {18, 4}, Punkt: 9, {4, 1, 14}, Punkt: 10, {16, 3, 12, 18}, Punkt: 11, {19}, Punkt: 12, {3}, Punkt: 13, {7, 10}, Punkt: 14, {17, 16, 10, 18, 1}, Punkt: 15, {12}, Punkt: 16, {11, 20, 17, 19}, Punkt: 17, {11, 9}, Punkt: 18, {13, 7}, Punkt: 19, {20}, Punkt: 20, {3, 10}]
```

[Punkt: Nummer, Set der Verbindungen]

Als nächstes werden beide Runner erstellt (Methode `runnerErstellen`):

```
[Runner: 0 [] 2, Runner: 1 [] 1]
```

[Runner: Typ GelaufeneFelder StartFeld]
Dann wird die "Simulation" gestartet... Nach 4 Iterationen liegen folgende Runner vor:

```
[Runner: 0 [2, 3, 6] 2, Runner: 0 [2, 3, 6] 12, Runner: 1 [1, 18, 7] 5, Runner: 1 [1, 18, 7] 8, Runner: 1 [1, 18, 13] 7, Runner: 1 [1, 18, 13] 10, Runner: 1 [1, 8, 18] 13, Runner: 1 [1, 8, 18] 7, Runner: 0 [2, 3, 19] 20, Runner: 0 [2, 19, 20] 3, Runner: 0 [2, 19, 20] 10]
```

Die `test()` Methode stellt nun fest, dass `Runner: 1 [1, 18, 13] 10` und `Runner: 0 [2, 19, 20] 10` sich auf Feld 10 treffen. -> Damit ist der Parcours möglich und der Weg beider Runner wird in der UI dargstellt:

```
Ausgabe:
Mika: 2 -> 19 -> 20 -> 10
Sasha: 1 -> 18 -> 13 -> 10
```

Ob ein Parcours möglich ist, wird also durch das Ausführen der Simulation überprüft. Das Programm kann auf alle Beispielaufgaben angewendet werden.

Aufbau des Programms (GUI)

Das Programm hat eine möglichst einfache UI, die das Individuelle Ausprobieren der einzelnen Lösungen ermöglicht. Das Programm startet in einer Übersichtsseite und die Aufgaben können über das Menu in der linken Spalte ausgewählt werden. Es wurden neben den eigen programmierten Widgets nur Widgets der Flutter Bibliothek genutzt.

Autor

Das Programm wurde von mir, Valentin Ahrend, designed und programmiert.