Praktische Informatik II | SoSe 2017 | Jana Göken
TutorIn: Nikhil Bhardwaj | Gruppe: 09 | Alex Rink
Meira Iske

Übungsblatt 2

Aufgabenlösung
Abgabe: 14.05.2017

Aufgabe 1 TheaPlusPlus

Aufgabe 1.1 TheaPP

Im Quellcode der Klasse TheaPP wurden die Methoden createInitialConfiguration(), distributeStudis(), nextConfiguration() und constraintsSatisfied() implementiert bzw. vervollständigt. In createinitialConfiguration() entsteht eine leere Startkonfiguration, indem ein leeres Feld von Listen erzeugt wird, die den Tutorien entsprechen, in die Später Studenten eingefügt werden können. In nextConfiguration() werden die vier Fälle umgesetzt, die in der Aufgabenstellung beschrieben sidn und eintreten können. Fälle 1-3 wurden von uns direkt implementiert, wobei wir für Fall vier die Hilfsmethode searchK() implementiert haben, welche das K sucht, ab dem alle Studenten aus den Tutorien entfernt werden. ContraintsSatisfied() unterscheidet eine gültige von einer ungültigen Konfiguration, indem die Bewertungen bezüglich der Mindestbewertungen der einzelnen Studenten überprüft werden. Sobald ein Student bezüglich der Mindestbewertung nicht in das Tutorium eingetragen werden kann, ist die Konfiguration nicht gültig. DistributeStudis() greift schließlich auf alle anderen Methoden zu und erzeugt eine möglichst gültige Verteilung der Studenten.

```
1  public class TheaPP {
2
3    /**
4     * Die maximal mögliche Bewertung von Studierenden für ein Tutorium. Die möglichen
5     * Werte für die Bewertung liegen damit zwischen 0 und diesem Wert (beide Werte
6     * inklusive).
7     */
8    public static final int MAX_RATING = 5;
9
10   /**
11    * Die aktuelle Konfiguration als Liste von Tutorien. Ein Tutorium wird hier als Liste
12    * von Studi-Objekten realisiert.
13    */
14   private List<List<Studi>> tutorials;
15
16   /**
17    * Die Anzahl der Tutorien.
18    */
19   private final int noOfTutorials;
20
21   /**
22    * Die maximale Anzahl von Studierenden in einem Tutorium.
23    */
24   private final int maxTutorialSize;
25
26   // TODO: evtl. eigene Attribute hinzufügen
27
28   /**
29    * Erzeugt ein neues Objekt zur Verteilung von Studierenden auf die gegebene Anzahl
30    * von Tutorien mit der gegebenen maximalen TeilnehmerInnenzahl pro Tutorium.
31    *
32    * @param pNoOfTutorials
```

```
33    *          Die Anzahl der Tutorien.
34    * @param pMaxTutorialSize
35    *          Die maximale Anzahl an Studierenden pro Tutorium.
36    * @throws IllegalArgumentException
37    *           Falls die Anzahl der Tutorien oder die maximale Anzahl der Studierenden
38    *           pro Tutorium negativ ist.
39    */
40   public TheaPP(final int pNoOfTutorials, final int pMaxTutorialSize) {
41     // TODO: evtl. Code ergänzen
42     if (pNoOfTutorials < 0) {
43       throw new IllegalArgumentException("Number of tutorials must not be negative!");
44     }
45     if (pMaxTutorialSize < 0) {
46       throw new IllegalArgumentException(
47         "Maximum number of students in tutorial must not be negative!");
48     }
49     noOfTutorials = pNoOfTutorials;
50     maxTutorialSize = pMaxTutorialSize;
51
52     tutorials = createInitialConfiguration();
53   }
54
55   /**
56    * Erzeugt die Startkonfiguration, d.h. eine Liste von Tutorien mit der korrekten
57    * Größe. Ein Tutorium ist dabei eine Liste von Studi-Objekten mit der maximalen
58    * Anzahl an Studierenden plus Eins als Größe.
59    *
60    * @return die Startkonfiguration als Liste von Studi-Listen
61    */
62   private List<List<Studi>> createInitialConfiguration() {
63     List<List<Studi>> btutorials = new ArrayList<>(noOfTutorials);
64     List<Studi> tutorial = new ArrayList<Studi>(maxTutorialSize+1);
65
66     for(int i = 0; i<noOfTutorials; i++){
67       btutorials.add(tutorial);
68     }
69
70     tutorials = btutorials;
71
72     return tutorials;
73
74   }
75
76   /**
77    * Verteilt die Studierenden aus der gegebenen Studi-Liste auf die Tutorien unter
78    * Berücksichtigung des gegebenen Wertes für die Mindestbewertung. Das bedeutet, dass
79    * Studierende niemals einem Tutorium zugeordnet werden dürfen, das sie mit einer
80    * Bewertung niedriger als die Mindestbewertung bewertet haben. Gleichzeitig dürfen
81    * pro Tutorium höchstens so viele Studierende zugeordnet werden, wie zuvor durch den
82    * Konstruktorparameter festgelegt (allgemein auf keinen Fall mehr als
83    * {@link TheaPP#MAX_STUDENTS_IN_TUTORIAL}).
84    *
85    * Wenn eine Verteilung der gegebenen Studierenden auf die Tutorien mit der gegebenen
86    * Mindestbewertung möglich und durchgeführt worden ist, wird {@code true}
87    * zurückgegeben, ansonsten {@code false}.
88    *
89    * @param pStudis
90    *          Die Liste mit den zu verteilenden Studis.
91    * @param pMinRating
92    *          Die Mindestbewertung für die Verteilung auf die Tutorien.
93    * @return {@code true} falls eine Verteilung möglich war, ansonsten {@code false}.
94    * @throws IllegalArgumentException
95    *           Falls die gegebene Mindestbewertung kleiner als 1 oder größer als
96    *           {@link TheaPP#MAX_RATING} ist, die Liste der zu verteilenden Studis
97    *           {@code null} ist oder nicht genug Platz in den Tutorien für die Studis
98    *           ist.
99    */
100  public final boolean distributeStudis(final List<Studi> pStudis,
101    final int pMinRating) {
102    if (pMinRating < 1 || pMinRating > TheaPP.MAX_RATING) {
103        throw new IllegalArgumentException("invalid minimum rating");
```

```
104       }
105       lastStudi =pStudis.get(0);
106       tutorials.get(0).add(lastStudi);
107       lastStudi.setTutorial(0);
108       int j = 0;
109       while (!(constraintsSatisfied(tutorials, pMinRating))) {
110         tutorials.get(j).remove(lastStudi);
111         tutorials.get(j+1).add(lastStudi);
112         lastStudi.setTutorial(j+1);
113         j++;
114       }
115       int counter = 0;
116       while (counter != -1) {
117         counter = nextConfiguration(tutorials, pStudis, counter, pMinRating);
118         if (counter == pStudis.size()) {
119           return true;
120         }
121       }
122         resetConfiguration();
123         return false;
124   }
125
126   /**
127    * Berechnet die Folgekonfiguration der gegebenen Konfiguration für die gegebene Liste
128    * von Studierenden, den Index des zuletzt zugeordneten Studis der gegebenen
129    * Konfiguration und die Mindestbewertung. Die Berechnung der Folgekonfiguration
130    * geschieht "in-place", d.h. die gegebene Konfiguration wird direkt durch diese
131    * Methode verändert. Gibt den Index des bei der Berechnung der Folgekonfiguration
132    * zuletzt zugeordneten Studis zurück.
133    *
134    * Die Methode ist nicht private, sondern package-private, da sie durch JUnit-Tests
135    * getestet werden soll. Da das Paket vor einer Auslieferung versiegelt wird, ist sie
136    * damit von außen nicht aufrufbar. Aus diesem Grund werden die Parameterwerte auch
137    * nicht auf sinnvolle Werte überprüft. Insbesondere darf diese Methode nicht mit
138    * einer Konfiguration aufgerufen werden, für die es keine Folgekonfiguration gibt.
139    *
140    * @param pConfiguration
141    *          Die Konfiguration, deren Folgekonfiguration errechnet werden soll.
142    * @param pStudents
143    *          Die Liste der Studierenden.
144    * @param pLastStudiIndex
145    *          Der Index der zuletzt zugeordneten StudentIn der gegebenen Konfiguration.
146    * @param pMinRating
147    *          Die Mindestbewertung.
148    * @return Den Index des zuletzt zugeordneten Studis oder -1 wenn es keine
149    *         Folgekonfiguration gibt und nicht die letzte StudentIn zugeordnet wurde
150    *         oder die Anzahl der StudentInnen wenn alle StudentInnen erfolgreich
151    *         zugeordnet wurden.
152    */
153   final int nextConfiguration(final List<List<Studi>> pConfiguration,
154     final List<Studi> pStudents, final int pLastStudiIndex, final int pMinRating) {
155     throw new UnsupportedOperationException(); // TODO: implementieren
156
157     //Fall 1
158     if(constraintsSatisfied(pConfiguration, pMinRating) && (pLastStudiIndex == pStudents.
        size())){
159
160     }
161     //Fall 2
162     if(constraintsSatisfied(pConfiguration, pMinRating) && (pLastStudiIndex < pStudents.
        size())){
163       pConfiguration.get(1).add(pStudents.get(pLastStudiIndex+1));
164     }
165     //Fall 3 ???
166     if(!this.constraintsSatisfied(pConfiguration, pMinRating) && (pConfiguration.indexOf(
        pLastStudiIndex) < noOfTutorials)) { // && j < t
167           int j;
168
169             for(int i = noOfTutorials; i>-1; i--){
170           if( (pConfiguration.get(i)).contains(pStudents.get(pLastStudiIndex)) ){
171             j = i;
```

```
172           }
173             }
174
175       pConfiguration.get(j).remove(pStudents.get(pLastStudiIndex));
176       pConfiguration.get(j+1).add(pStudents.get(pLastStudiIndex));
177
178
179
180     }
181     //Fall 4
182     else {
183         k = this.searchK(pStudents,pConfiguration);
184         if(k < 0){
185           this.resetConfiguration();
186           return -1;
187         }
188         else{
189           for(int i = k+1, i <= pLastStudiIndex, i++){
190             (pConfiguration.get(noOfTutorials-2)).remove(pStudents.get(i));
191           }
192           for(int i = noOfTutorials-2, i > -1 , i--){
193               if( (pConfiguration.get(i)).contains(pStudents.get(k)) ){
194                 (pConfiguration.get(i)).remove(pStudents.get(k));
195               }
196           }
197         (pConfiguration.get(i+1)).add(pStudents.get(k));
198         tutorials = pConfiguration;
199         return k
200   }
201
202   /**
203    * Prüft, ob die gegebene Konfiguration hinsichtlich der gegebenen Mindestbewertung
204    * gültig ist. Wenn das so ist, wird {@code true} zurückgegeben, sonst {@code false}.
205    *
206    * Die Methode ist nicht private, sondern package-private, da sie durch JUnit-Tests
207    * getestet werden soll. Da das Paket vor einer Auslieferung versiegelt wird, ist sie
208    * damit von außen nicht aufrufbar.
209    *
210    * @param pConfiguration
211    *          Die zu prüfende Konfiguration als Liste von Studi-Listen.
212    * @param pMinRating
213    *          Die Mindestbewertung der Studis für ihre Tutorien.
214    * @return {@code true} falls kein Tutorium mehr als die erlaubte Maximalzahl von
215    *         Studis enthält und kein Studi einem Tutorium zugeordnet ist, für das sie
216    *         oder er eine geringere Bewertung als die gegebene Mindestbewertung vergeben
217    *         hat, ansonsten {@code false}.
218    */
219   final boolean constraintsSatisfied(final List<List<Studi>> pConfiguration,
220     final int pMinRating) {
221
222
223     for(int i= 0; i<noOfTutorials; i++) {
224      for(int j = 0; j<maxTutorialSize; j++){
225
226       Studi student = pConfiguration.get(i).get(j);
227        if(student.getRating(i) < pMinRating){
228          return false;
229        }
230      }
231     }
232     return true;
233   }
234
235   /**
236    * Setzt die Konfiguration auf die Startkonfiguration zurück.
237    */
238   private void resetConfiguration() {
239     for (final List<Studi> tutorial : tutorials) {
240       tutorial.clear();
241     }
242   }
```

```
243
244   /**
245    * Gibt die aktuelle Verteilung der Studierenden in der Konsole aus. Dabei steht in
246    * jeder Zeile ein Tutorium und die Teilnehmenden werden mit ihrem Account und der
247    * Bewertung für ihr aktuell zugeordnetes Tutorium durch Komma getrennt aufgeführt.
248    */
249   private void printDistribution() {
250     int tutNo = 1;
251     for (final List<Studi> tutorial : tutorials) {
252       System.out.print(String.format("Tut %02d:", tutNo));
253       System.out.println(Arrays.toString(tutorial.toArray()));
254       tutNo++;
255     }
256   }
257
258   /**
259    * Hilfsmethode
260    * Sucht das k' aus Fall 4 und gibt dieses zurück, falls es eines gibt.
261    * Ansonsten wird -1 zurückgegeben.
262    */
263   private int searchK(List<Studi> pStudents, List<List<Studi>> pConfiguration, int
       pLastStudiIndex){
264     int k = -1;
265
266     for(int i = noOfTutorials-2; i > -1; i--){
267       for(int j = 0; j < maxTutorialSize; j++){
268         if(
269           pConfiguration.get(i)).get(j) != null
270           && k < pStudents.indexOf( ((pConfiguration.get(i)).get(j))
271           && for( int l = k+1, l <= pLastStudiIndex, l++) {
272               (pConfiguration.get(noOfTutorials-1)).contains(pStudents.get(l));
273              }
274           ){
275             k = pStudents.indexOf( ((pConfiguration.get(i)).get(j)) );
276         }
277       }
278     }
279     return k;
280   }
```

Aufgabe 1.2 Main

Für die Main-Methode haben wir zunächst die Hilfsmethode validSetup angelegt, die die in der JavaDoc beschrieben Funktion hat. Die Methode war leider nicht ausfürbar, da es an anderer Stelle Probleme zu geben scheint, die Main-Methode an sich sollte jedoch funktionstüchtig sein.

```
1
2   /**
3    * Hilfmethode. Dient zur Überpruefung der Parameter, also ob
4    * es ausreichend viele Plätze in den Tutorien für alle Studenten gibt und ob jeder
5    * Student mindestens ein Tutorium mit dem Minimum-Rating bewertet hat. In diesem Fall
6    * wird {@code true} zurueckgegeben.
7    * Ist eine dieser beiden Bedingungen verletzt, wird {@code false} zurueckgegeben.
8    *
9    * @param pStudis
10   * @param pNoOfTutorials
11   * @param pMaxTutorialSize
12   * @param pMinRating
13   * @return
14   */
15  private boolean validSetup(final List<Studi> pStudis, final int pNoOfTutorials,
16      final int pMaxTutorialSize, final int pMinRating) {
17      if (pStudis.size() > pNoOfTutorials * pMaxTutorialSize) {
18      return false; }
19      boolean hasMinRat;
20      for (Studi studi :pStudis) {
21      hasMinRat = false;
22      for (int k = 0 ; k < pNoOfTutorials ; k++) {
```

```
23          if (studi.getRating(k) >= pMinRating) {
24              hasMinRat = true;
25              break;
26              }
27            }
28          if (hasMinRat) {
29            continue;
30          }
31          return false;
32      }
33        return true;
34        }
35
36          /**
37     * Liest die Dateien FiveTutorials.csv, SevenTutorials.csv und
38     * TenTutorials.csv ein, parst sie und versucht die Studierenden auf die Tutorien zu
39     * verteilen. Wenn die Verteilung gelingt, wird sie auf der Konsole ausgegeben.
40     *
41     * @param args
42     *           Werden ignoriert.
43     * @throws TheaPPParseException
44     *            Falls eine der Dateien nicht eingelesen werden kann.
45     */
46    public static void main(final String[] args)  {
47      final String filename = args[0];
48        final int noOfTutorials = Integer.parseInt(args[1]);
49        final int maxTutorialSize = Integer.parseInt(args[2]);
50        final int minRating = Integer.parseInt(args[3]);
51        final TheaPP thea = new TheaPP(noOfTutorials, maxTutorialSize);
52        try {
53          final List<Studi> studiList = (new TheaPPParser(noOfTutorials )).parseStudents(
              filename);
54          if(!(thea.validSetup(studiList, noOfTutorials, maxTutorialSize, minRating))) {
55            throw new IllegalArgumentException("invalid setup");
56          }
57          if (thea.distributeStudis(studiList, minRating)) {
58            thea.printDistribution();
59          }
60          else {
61            throw new IllegalArgumentException("ivalid setup");
62          }
63        }
64          catch (Exception e) {
65            System.out.println(e);
66          }
67        }
68    }
```

Aufgabe 2 Bonus

Aufgrund der Probleme bei der Ausführung der Main-Methode erwies es sich als äußerst schwierig, Vermutungen zu den angeführten Fragestellungen aufzustellen.

Es könnte sich als problematisch erweisen, dass einige der Studenten die Mindestbewertung von 4 für kein Tutorium angegeben haben und deshalb nie alle Studenten zufriedengestellt verteilt sind.