9 Anhang: Programmcode

9.1 Package main

9.1.1 Klasse Main

```
1  package main;
2
3  import java.io.File;
4  import java.io.IOException;
5
6  import controller.Controller;
7  import io.AusgabeInDatei;
8  import io.LeseAusDatei;
9  import model.Model;
10
11 /**
12  * @author M. Leonard Haufs Prüflingsnummer: 101-20540
13  *
14  */
15 public class Main {
16     public static void main(String[] args) {
17         String dateiendung;
18         String verzeichnis;
19
20         // Nur zum Testen- wird später aus über die Argumente args der Jar
21         // übergeben
22         args = new String[2];
23         args[0] = ".in";
24         args[1] = "/Users/hfs23/Dropbox/MATSE/Programmieraufgaben/2018_GrosseProg/Testfaelle";
25
26         // Parameterübergabe prüfen
27         if (args.length != 2) {
28             // keine korrekte Parameterübergabe
29             System.out.println(
30                     "Es müssen 2 Parameter übergeben werden. Paramter 1: Endung der
                           Eingabedateien (z.B.: .in)\nParameter 2: Verzechnis aus dem die
                           Eingabedateien gelesen werden soll.");
31             return;
32         }
33         dateiendung = args[0];
34         verzeichnis = args[1];
35
36         File f;
37         try {
38             try {
39                 f = new File(verzeichnis);
40             } catch (Exception ex) {
41                 throw new IOException("Der Angegebene Pfad existiert nicht");
42             }
43
44             if (f.isDirectory() && f.canRead()) {
45                 File[] dateien = f.listFiles();
46                 for (int i = 0; i < dateien.length; i++) {
47                     // Prüfe ob die Datei gelesen werden kann
48                     if (dateien[i].isFile() && dateien[i].canRead()) {
49                         String tempEndung =
                                dateien[i].getName().substring(dateien[i].getName().lastIndexOf("."),
50                                    dateien[i].getName().length());
51                         // wenn die Dateiendung der gewählten entspricht
52                         // wird die Datei eingelesen
53                         if (dateiendung.equals(tempEndung)) {
54
55                             // Eingabe
56
57                             LeseAusDatei in = new LeseAusDatei();
58                             Model model = in.getModelAusDatei(dateien[i]);
59
60                             // Berechnung
61                             Controller c = new Controller(model);
62                             c.calculate();
63
64                             // Ausgabe
65                             AusgabeInDatei out = new AusgabeInDatei(model);
66
67                             String outputPath = verzeichnis + "/" +
                                    (dateien[i].getName().replace(dateiendung, ".out"));
68                             out.schreibeModelInDatei(outputPath);
69
70                             // OutputConsole out = new OutputConsole();
71                             // out.printEntireOutputString(model);
72                         }
73                     }
74                 }
```

```
75                      System.out.print("Vorgang abgeschlossen.");
76                  } else {
77                      throw new IOException("Der Angegebene Pfad ist kein Ordner oder kann nicht
                            geöffnet werden.");
78                  }
79              } catch (IOException ex) {
80                  System.out.println(ex.getMessage());
81              }
82
83          }
84      }
```

9.2 Package io

9.2.1 Klasse LeseAusDatei

```java
package io;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

import model.Knoten;
import model.Model;

/**
 * Ermöglicht das Einlesen der Daten eines Models aus einer Datei
 *
 * @author M. Leonard Haufs Prüflingsnummer: 101-20540
 *
 */
public class LeseAusDatei {

    /**
     * Liefert die Daten eines Models, die in einer Datei gespeichert sind.
     *
     * @param file
     *            Datei, aus der gelesen werden soll.
     * @return Model mit dem gekapselten Daten. Falls eine ungültige Eingabe
     *         erfolgt, wird ein leeres Model zurückgegeben.
     */
    public Model getModelAusDatei(File file) {
        ArrayList<Knoten> knoten = new ArrayList<>();
        String kommentar = "Fehler beim Einlesen.";
        ArrayList<Integer> vorgangsnummern = new ArrayList<>();

        BufferedReader br;
        try {
            br = new BufferedReader(new InputStreamReader(new FileInputStream(file)));
        } catch (FileNotFoundException ex) {
            System.out.println(ex);
            return new Model();
        }

        try {
            String aktZeile = "";
            while ((aktZeile = br.readLine()) != null) {
                if (aktZeile.startsWith("//")) {
                    // Zeile beginnt mit "//+ " ?
                    if (aktZeile.startsWith("//+ ")) {
                        if (aktZeile.length() > 4) {
                            kommentar = aktZeile.substring(4, aktZeile.length());
                        }
                        continue;
                    }
                    continue;
                }

                String aktZeileOhneLeer = aktZeile.replace(" ", "");
                String[] zeileSplit = aktZeileOhneLeer.split(";");
                if (zeileSplit.length != 5) {
                    System.out.println("In Datei " + file.getName()
                            + ": Ungenügende Eingabe. Es müssen je Zeile genau 5 Argumente
                                getrennt mit einem Semikolon übergeben werden: "
                            + aktZeile);
                    br.close();
                    return new Model();
                }
                int nr = Integer.parseInt(zeileSplit[0]);
                vorgangsnummern.add(nr);
                String beschr = aktZeile.split("; ")[1];
                int dauer = Integer.parseInt(zeileSplit[2]);

                ArrayList<Integer> vorgaengerNummern = new ArrayList<>();
                if (!zeileSplit[3].equals("-")) {
                    String[] vorgaengerNummernArr = zeileSplit[3].split(",");
                    for (int i = 0; i < vorgaengerNummernArr.length; i++) {
                        String string = vorgaengerNummernArr[i];
                        int number = Integer.parseInt(string);
                        vorgaengerNummern.add(number);
```

```
78                      }
79                  }
80
81                  ArrayList<Integer> nachfolgerNummern = new ArrayList<>();
82                  if (!zeileSplit[4].equals("-")) {
83                      String[] nachfolgerNummernArr = zeileSplit[4].split(",");
84                      for (int i = 0; i < nachfolgerNummernArr.length; i++) {
85                          String string = nachfolgerNummernArr[i];
86                          int number = Integer.parseInt(string);
87                          nachfolgerNummern.add(number);
88                      }
89                  }
90                  // Prüfe, ob vorgangsnummern nicht doppelt vorliegen
91                  if (!vorgangsnummernNichtDoppelt(vorgangsnummern)) {
92                      System.out.println("In Datei " + file.getName()
93                              + ": Ungenügende Eingabe: Es kommt mindestens eine Vorgangsnummer
                                    mehrfach vor.");
94                      br.close();
95                      return new Model();
96                  }
97                  Knoten k = new Knoten(nr, beschr, dauer, vorgaengerNummern, nachfolgerNummern);
98                  knoten.add(k);
99              }
100             br.close();
101         } catch (IOException ex) {
102             System.out.println(ex);
103             new Model();
104         } catch (NumberFormatException e) {
105             System.out.println("In Datei " + file.getName()
106                     + ": Ungenügende Eingabe. Es wurde mindestens eine ungültige Zahl
                            eingeben.");
107             return new Model();
108         }
109         if (!alleKnotenVerweisenAufExistierendenKnoten(knoten, vorgangsnummern)) {
110             System.out.println("In Datei " + file.getName()
111                     + ": Ungenügende Eingabe: Es existieren ungültige Referenzen, da
                            mindestens ein Knoten auf einen nicht existenten Knoten
                            referenziert.");
112             return new Model();
113         }
114         Model model = new Model(knoten, kommentar);
115         return model;
116     }
117
118     /**
119      * Prüft, ob die Vorgangsnummern nicht doppelt vorliegen
120      *
121      * @param vorgangsnummern
122      *            die zu Prüfen sind
123      * @return true, falls die Vorgangsnummern nicht doppelt vorliegen
124      */
125     private boolean vorgangsnummernNichtDoppelt(ArrayList<Integer> vorgangsnummern) {
126         @SuppressWarnings("unchecked")
127         ArrayList<Integer> copyOfVorgangsnummern = (ArrayList<Integer>)
                vorgangsnummern.clone();
128
129         for (int i = 0; i < copyOfVorgangsnummern.size(); i++) {
130             Integer vorgangsnummer = copyOfVorgangsnummern.get(i);
131             copyOfVorgangsnummern.remove(vorgangsnummer);
132             if (copyOfVorgangsnummern.contains(Integer.valueOf(vorgangsnummer))) {
133                 return false;
134             }
135         }
136         return true;
137     }
138
139     /**
140      * Prüft, ob alle Knoten auf einen existierenden Knoten verweisen.
141      *
142      * @param knoten
143      *            Knotenliste, der zu prüfenden Knoten
144      * @param vorgangsnummern
145      *            Liste der Vorgangsnummern aller Knoten
146      * @return true, falls alle Knoten auf einen existierenden Knoten verweisen,
147      *         sonst false
148      */
149     private boolean alleKnotenVerweisenAufExistierendenKnoten(ArrayList<Knoten> knoten,
150             ArrayList<Integer> vorgangsnummern) {
151         for (Knoten k : knoten) {
152             for (int nachfolgernummer : k.getNachfolgerNummern()) {
153                 if (!vorgangsnummern.contains(Integer.valueOf(nachfolgernummer))) {
154                     return false;
155                 }
156             }
```

```
157
158                 for (int vorgaengernummer : k.getVorgaengerNummern()) {
159                     if (!vorgangsnummern.contains(Integer.valueOf(vorgaengernummer))) {
160                         return false;
161                     }
162                 }
163
164             }
165             return true;
166         }
167     }
```

9.2.2 Klasse Ausgabe

```
1  package io;
2
3  import java.util.ArrayList;
4
5  import model.Knoten;
6  import model.Model;
7
8  /**
9   * Ermöglicht zu einem Model die Ausgabe der kenngrößen und kritischen Pfade
10  * auszugeben
11  *
12  * @author M. Leonard Haufs Prüflingsnummer: 101-20540
13  *
14  */
15 public abstract class Ausgabe {
16     private Model model;
17
18     /**
19      * Konstruktor, der Ausgabe mit einem Model initialisiert
20      *
21      * @param model
22      *            model, welches die auszugebenen Daten enthällt
23      */
24     public Ausgabe(Model model) {
25         super();
26         this.model = model;
27     }
28
29     /**
30      * Gibt den Ausgabestring zurück.
31      *
32      * Falls nicht zusammenhängend oder falls Zyklen enthalten sind, wird ein
33      * entsprechender Fehler ausgegeben.
34      *
35      * @return Ausgabestring
36      */
37     protected String getAusgabeString() {
38         StringBuilder sb = new StringBuilder();
39
40         if (this.model.getKnoten().size() == 0) {
41             sb.append("Berechnung nicht möglich.");
42             sb.append("\n");
43             sb.append("Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen
                   zu erhalten.");
44         } else if (this.model.getZyklus().size() != 0) {
45             sb.append(this.model.getName());
46             sb.append("\n");
47             sb.append("\n");
48             sb.append("Berechnung nicht möglich.");
49             sb.append("\n");
50             sb.append("Zyklus erkannt: ");
51             this.getZyklusString(sb);
52         } else if (!this.model.isZusammenhaengend()) {
53             sb.append(this.model.getName());
54             sb.append("\n");
55             sb.append("\n");
56             sb.append("Berechnung nicht möglich.");
57             sb.append("\n");
58             sb.append("Nicht zusammenhängend.");
59         } else if (!this.model.isGueltigeReferenzen()) {
60             sb.append(this.model.getName());
61             sb.append("\n");
62             sb.append("\n");
63             sb.append("Berechnung nicht möglich.");
64             sb.append("\n");
65             sb.append(
66                     "Referenzen der Eingabe sind nicht gültig! Es gibt also mindestens einen
                           Knoten,\ndessen Nachfolger den Knoten selbst nicht als Vorgänger
                           hat\nbzw. dessen Vorgänger den Knoten selbst nicht als Nachfolger
                           hat.");
67         } else {
68             sb.append("Vorgangsnummer; Vorgangsbeschreibung; D; FAZ; FEZ; SAZ; SEZ; GP; FP");
69             sb.append("\n");
70             this.getKnotenbeschreibung(sb);
71             sb.append("\n");
72             this.getVorgangString(sb);
73             sb.append("\n");
74             this.getGesamtdauer(sb);
75             sb.append("\n");
76             sb.append("\n");
77             this.getKritischerPfadString(sb);
```

```
78              }
79
80              return sb.toString();
81          }
82
83          /**
84           * Gibt die Beschreibung eines Knotens im Netzplan. Dabei wird der übergebene
85           * StringBuilder verändert.
86           *
87           * @param sb
88           *              Stringbuilder, an den die Beschreibung angehängt werden soll
89           */
90          private void getKnotenbeschreibung(StringBuilder sb) {
91              for (Knoten knoten : model.getKnoten()) {
92                  sb.append(knoten.getVorgangsnummer());
93                  sb.append("; ");
94                  sb.append(knoten.getVorgangsbezeichnung());
95                  sb.append("; ");
96                  sb.append(knoten.getDauer());
97                  sb.append("; ");
98                  sb.append(knoten.getFaz());
99                  sb.append("; ");
100                 sb.append(knoten.getFez());
101                 sb.append("; ");
102                 sb.append(knoten.getSaz());
103                 sb.append("; ");
104                 sb.append(knoten.getSez());
105                 sb.append("; ");
106                 sb.append(knoten.getGp());
107                 sb.append("; ");
108                 sb.append(knoten.getFp());
109                 sb.append("\n");
110             }
111         }
112
113         /**
114          * Gibt die Beschreibung von Anfangs- und Endvorgang zurück
115          *
116          * @param sb
117          *              Stringbuilder, an den die Beschreibung von Anfangs- und Endvorgang
118          *              angehängt werden soll
119          */
120         private void getVorgangString(StringBuilder sb) {
121             sb.append("Anfangsvorgang: ");
122             for (int i = 0; i < model.getStartknoten().size(); i++) {
123                 Knoten startK = model.getStartknoten().get(i);
124
125                 sb.append(startK.getVorgangsnummer());
126                 if (i != model.getStartknoten().size() - 1) {
127                     sb.append(",");
128                 }
129             }
130             sb.append("\n");
131             sb.append("Endvorgang: ");
132             for (int i = 0; i < model.getEndknoten().size(); i++) {
133                 Knoten endK = model.getEndknoten().get(i);
134
135                 sb.append(endK.getVorgangsnummer());
136                 if (i != model.getEndknoten().size() - 1) {
137                     sb.append(",");
138                 }
139             }
140         }
141
142         /**
143          * Gibt die Gesamtdauer des kritischen Pfades zurück. Sind mehrere Kritische
144          * Pfade enthalten, so wird "Nicht eindeutig" zurückgegeben
145          *
146          * @return Gesamtdauer des kritischen Pfades. Sind mehrere Kritische Pfade
147          *         enthalten, so wird "Nicht eindeutig" zurückgegeben
148          */
149         private void getGesamtdauer(StringBuilder sb) {
150             sb.append("Gesamtdauer: ");
151             if (this.model.getKritischePfade().size() == 0) {
152                 sb.append(0);
153             } else if (this.model.getKritischePfade().size() > 1) {
154                 sb.append("Nicht eindeutig");
155             } else {
156                 int gesamtdauer = 0;
157                 ArrayList<Knoten> firstKritPfad = this.model.getKritischePfade().get(0);
158                 for (Knoten knoten : firstKritPfad) {
159                     gesamtdauer += knoten.getDauer();
160                 }
161                 sb.append(gesamtdauer);
```

```
162             }
163
164         }
165
166         /**
167          * Hängt die String- Repräsentation des/der Kritischen Pfade(s) an einen
168          * übergebenen Stringbuilder an
169          *
170          * @param sb
171          *             StringBuilder, an den die String- Repräsentation des/der
172          *             Kritischen Pfade(s) angehängt werden soll
173          */
174         private void getKritischerPfadString(StringBuilder sb) {
175             if (this.model.getKritischePfade().size() > 1) {
176                 sb.append("Kritische Pfade");
177             } else {
178                 sb.append("Kritischer Pfad");
179             }
180             sb.append("\n");
181
182             for (ArrayList<Knoten> kritischerPfad : this.model.getKritischePfade()) {
183                 for (int i = 0; i < kritischerPfad.size(); i++) {
184                     Knoten knoten = kritischerPfad.get(i);
185                     sb.append(knoten.getVorgangsnummer());
186                     if (i != kritischerPfad.size() - 1) {
187                         sb.append("->");
188                     }
189                 }
190                 sb.append("\n");
191             }
192         }
193
194         /**
195          * Hängt die String- Repräsentation eines Zyklus an einen übergebenen
196          * Stringbuilder an
197          *
198          * @param sb
199          *             StringBuilder, an den die String- Repräsentation des/der Zyklus
200          *             angehängt werden soll
201          */
202         private void getZyklusString(StringBuilder sb) {
203             int posDerErstenWiederholung = this.posDerErstenWiederholung(this.model.getZyklus());
204
205             for (int i = posDerErstenWiederholung; i < this.model.getZyklus().size(); i++) {
206                 Knoten knoten = this.model.getZyklus().get(i);
207                 sb.append(knoten.getVorgangsnummer());
208                 if (i != this.model.getZyklus().size() - 1) {
209                     sb.append("->");
210                 }
211             }
212             sb.append("\n");
213         }
214
215         /**
216          * Gibt die Position des ersten Elementes in einer ArrayList von Knoten zurück,
217          * die doppelt vorkommt
218          *
219          * @param knoten
220          *             ArrayList<Knoten>, die überprüft werden soll
221          * @return Position des ersten Elementes in einer ArrayList von Knoten, die
222          *         doppelt vorkommt
223          */
224         private int posDerErstenWiederholung(ArrayList<Knoten> knoten) {
225             ArrayList<Knoten> ks = new ArrayList<>();
226             for (int i = 0; i < knoten.size(); i++) {
227                 Knoten k = knoten.get(i);
228                 if (ks.contains(k)) {
229                     return ks.indexOf(k);
230                 }
231                 ks.add(k);
232             }
233             return 0;
234         }
235
236     }
```

9.2.3 Klasse Ausgabe

```
1  package io;
2
3  import java.io.File;
4  import java.io.FileWriter;
5  import java.io.IOException;
6
7  import model.Model;
8
9  /**
10  *
11  * @author M. Leonard Haufs Prüflingsnummer: 101-20540
12  *
13  */
14 public class AusgabeInDatei extends Ausgabe {
15
16     public AusgabeInDatei(Model model) {
17         super(model);
18     }
19
20     public void schreibeModelInDatei(String path) {
21         String outputString = super.getAusgabeString();
22
23         File file = new File(path);
24         FileWriter writer;
25         try {
26             writer = new FileWriter(file, false);
27         } catch (IOException ex) {
28             System.out.println("Fehler beim öffnen/erstellen der Datei!");
29             return;
30         }
31         try {
32             writer.write(outputString);
33             writer.close();
34         } catch (IOException ex) {
35             System.out.println("Fehler beim schreiben in die Datei!");
36             ex.printStackTrace();
37         }
38     }
39
40 }
```

9.3 Package controller

9.3.1 Klasse Ausgabe

```
1  package controller;
2
3  import java.util.ArrayList;
4
5  import model.Knoten;
6  import model.Model;
7
8  /**
9   * Hauptberechnungsklasse.
10  *
11  * @author M. Leonard Haufs Prüflingsnummer: 101-20540
12  *
13  */
14 public class Controller {
15     private Model model;
16     private ArrayList<Knoten> validationsListe;
17
18     // Konstruktor
19     public Controller(Model model) {
20         super();
21         this.model = model;
22     }
23
24     /**
25      * Hauptberechnungsmethode des Controllers.
26      *
27      * Falls noch nicht initialisiert wurde, wird auf Zyklen und Zusammenhängigkeit
28      * geprüft. Falls der Netzplan Zyklen enthällt, wird im Model in zyklus ein
29      * zyclus gespeichert. Wenn der Netzplan nicht nicht zusammenhängend ist, wird
30      * im Model isZusammenhaengend auf false gesetzt. Sonst auf true.
31      *
32      * Anschließend wird das Model initialisiert, also die kenngrößen berechnet und
33      * anschließend der kritische Pfad, falls er existiert, berechnet
34      */
35     public void calculate() {
36         // Prüfe, ob der im Model gekapselte Netzplan keine Zyklen enthällt
37         boolean hatKeineZyklen = this.hatKeineZyklen();
38         if (!hatKeineZyklen) {
39             System.out.println(this.model.getName() + ": Zyklen enthalten");
40             return;
41         }
42
43         // Prüfe, ob der im Model gekapselte Netzplan zusammenhängend ist
44         boolean isZusammenhaengend = this.isZusammenhaengend();
45         if (!isZusammenhaengend) {
46             System.out.println(this.model.getName() + ": Fehler (Nicht zusammenhängend)");
47             model.setZusammenhaengend(false);
48             return;
49         } else {
50             model.setZusammenhaengend(true);
51         }
52
53         // Prüft, ob alle Referenzen in model.knoten korrekt sind, also ob jeder
54         // Nachfolger auch in dessen Vorgaengern enthalten ist bzw. umgekehrt.
55         boolean hatGueltigeReferenzen = this.hatGueltigeReferenzen();
56         if (!hatGueltigeReferenzen) {
57             System.out.println(this.model.getName()
58                     + "Referenzen der Eingabe sind ungenügend. nicht jeder Nachfolger auch in
                           dessen Vorgaengern enthalten bzw. umgekehrt ist.");
59             model.setGueltigeReferenzen(false);
60         } else {
61             model.setGueltigeReferenzen(true);
62         }
63
64         // Initialisiere das Model
65         if (!this.model.isInitialized()) {
66             initModel();
67         }
68     }
69
70     /**
71      * Prüft, ob der im Model gekapselte Netzplan keine Zyklen enthällt
72      *
73      * @return true, falls der Netzplan im Model keine Zyklen enthällt, sonst true
74      */
75     boolean hatKeineZyklen() {
76         ArrayList<Boolean> check = new ArrayList<>();
77
```

```
78          /*
79           * Rufe für ausgehend von allen Startknoten die Helpermethode
80           * hatKeineZyklenHelper auf. Falls ein Ergebnis negativ ausfällt wird false
81           * zurückgegeben
82           */
83          for (Knoten s : this.model.getStartknoten()) {
84              this.validationsListe = new ArrayList<>();
85              check.add(hatKeineZyklenHelper(s));
86              if (check.contains(Boolean.valueOf(false))) {
87                  model.setZyklus(this.validationsListe);
88                  return false;
89              }
90          }
91          return true;
92      }
93
94      /**
95       * Hilfsfunktion zur Überprüfung, ob keine Zyklen existieren
96       *
97       * @param aktKnoten
98       * @return
99       */
100     private boolean hatKeineZyklenHelper(Knoten aktKnoten) {
101         // Abbruchbedingung
102         if (this.validationsListe.contains(aktKnoten)) {
103             // Falls aktueller Knoten bereits in ValidationListe enthaöten ist, füge
104             // aktuellen Knoten zu ValidationListe zu und gebe false zurück
105             this.validationsListe.add(aktKnoten);
106             return false;
107         }
108         // Füge aktuellen Knoten zur Validationliste hinzu
109         this.validationsListe.add(aktKnoten);
110         // Für jeden nachfolger des aktuellen Knotens führe rekursiv
111         // hatKeineZyklenHelper aus und gebe den Wert zurück.
112         for (Knoten nachfolger : aktKnoten.getNachfolger()) {
113             return this.hatKeineZyklenHelper(nachfolger);
114         }
115         return true;
116     }
117
118     /**
119      * Prüft, ob der Netzplan zusammenhängend ist.
120      *
121      * @return true, falls der Netzplan zusammenhängend ist, sonst false
122      */
123     boolean isZusammenhaengend() {
124         this.validationsListe = new ArrayList<>();
125
126         for (Knoten startK : this.model.getStartknoten()) {
127             isZusammenhaengendHelper(startK);
128         }
129         if (this.validationsListe.size() == model.getKnoten().size()) {
130             return true;
131         } else {
132             return false;
133         }
134     }
135
136     /**
137      * Helper-Funktion zur Bestimmung, ob der Netzplan zusammnhängend ist
138      *
139      * @param aktKnoten
140      *             aktuell betrachteter Knoten
141      */
142     private void isZusammenhaengendHelper(Knoten aktKnoten) {
143         // Falls die ValidationListe den aktuellen Knoten noch nicht enthällt, füge
144         // diesen ein.
145         if (!this.validationsListe.contains(aktKnoten)) {
146             this.validationsListe.add(aktKnoten);
147         }
148         // rufe isZusammenhaengendHelper für jeden Nachfolger des aktuellen Knotens auf
149         for (Knoten nachfolger : aktKnoten.getNachfolger()) {
150             isZusammenhaengendHelper(nachfolger);
151         }
152     }
153
154     /**
155      * Initialisiert das Model. Dabei werden drei Phasen durchlaufen:
156      *
157      * 1. Phase: Vorwa rtsrechnung Bei gegebenem Anfangstermin werden aufgrund der
158      * angegebenen Dauer eines Vorganges die fru hestmo glichen Anfangs- und
159      * Endzeiten eingetragen. Weiterhin la sst sich die Gesamtdauer eines Projekts
160      * bestimmen.
161      *
```

```
162         * 2. Phase: Ru ckwa rtsrechnung: Bei der Ru ckwa rtsrechnung wird ermittelt,
163         * wann die einzelnen Vorga nge spa testens begonnen und fertiggestellt sein
164         * mu ssen, damit die Gesamtprojektzeit nicht gefa hrdet ist.
165         *
166         *
167         * 3. Phase: Ermittlung der Zeitreserven und des kritischen Pfades: In dieser
168         * Phase wird ermittelt, welche Zeitreserven existieren und welche Vorga nge
169         * besonders problematisch sind (kritischer Vorgang), weil es bei diesen keine
170         * Zeitreserven gibt. Dazu wird fu r alle Knoten der Gesamtpuffer (GP)
171         * berechnet, sowie der freie Puffer (FP).
172         */
173        private void initModel() {
174            // Prüfe, ob das Model bereits initialisiert wurde
175            if (this.model.isInitialized()) {
176                return;
177            }
178
179            /*
180             * 1. Phase: Vorwärtsrechnung
181             *
182             * Setze FAZ der Startknoten
183             */
184            for (Knoten startK : this.model.getStartknoten()) {
185                // Der Startknoten hat als FAZ immer den Wert 0
186                startK.setFaz(0);
187            }
188
189            // Setze FEZ aller Knoten als FEZ = FAZ + Dauer
190            for (Knoten startK : this.model.getStartknoten()) {
191                // startK.setFez(startK.getFaz() + startK.getDauer());
192                this.setFezAndFaz(startK);
193            }
194
195            // /*
196            // * Setze FAZ für alle Nachfolger der Startknoten als der gro ßte
197            // * (spa teste) FEZ der unmittelbaren Vorga nger.
198            // */
199            // for (Knoten startK : this.model.getStartknoten()) {
200            // for (Knoten nachfolger : startK.getNachfolger()) {
201            // setFaz(nachfolger);
202            // }
203            // }
204
205            /*
206             * 2. Phase: Ru ckwa rtsrechnung
207             *
208             * Bei der Ru ckwa rtsrechnung wird ermittelt, wann die einzelnen Vorga nge
209             * spa testens begonnen und fertiggestellt sein mu ssen, damit die
210             * Gesamtprojektzeit nicht gefa hrdet ist.
211             *
212             * Fu r den letzten Vorgang ist der fru heste Endzeitpunkt (FEZ) auch der
213             * spa teste Endzeitpunkt (SEZ), also SEZ = FEZ.
214             */
215            for (Knoten endK : this.model.getEndknoten()) {
216                endK.setSez(endK.getFez());
217            }
218
219            /*
220             * Fu r den spa testen Anfangszeitpunkt gilt: SAZ = SEZ     Dauer.
221             */
222            for (Knoten endKnoten : this.model.getEndknoten()) {
223                this.setSazAndSez(endKnoten);
224            }
225
226            // /*
227            // * Der SAZ eines Vorgangs wird SEZ aller unmittelbarer Vorga nger
228            // *
229            // * Haben mehrere Vorga nge einen gemeinsamen Vorga nger, so ist dessen SEZ der
230            // * fru heste (kleinste) SAZ aller Nachfolger.
231            // **/
232            // for (Knoten endK : this.model.getEndknoten()) {
233            // setSez(endK);
234            // }
235
236            // 3. Phase: Ermittlung der Zeitreserven
237            for (Knoten startK : this.model.getStartknoten()) {
238                /*
239                 * Berechnung des Gesamtpuffers fu r jeden Knoten
240                 */
241                this.setGp(startK);
242
243                /*
244                 * Berechnung des freien Puffers
245                 */
```

```
246             this.setFp(startK);
247         }
248
249         /*
250          * Bestimmung der kritischen Vorga nges
251          */
252         this.setKritischePfade();
253
254         this.model.initialize();
255     }
256
257     /**
258      * Setzt FEZ und FAZ ausgehend von einem aktuellen Knoten für diesen und alle
259      * Nachfolger dieses Knotens
260      *
261      * @param aktKnoten
262      */
263     private void setFezAndFaz(Knoten aktKnoten) {
264         // Fu r den FEZ gilt: FEZ = FAZ + Dauer
265         aktKnoten.setFez(aktKnoten.getFaz() + aktKnoten.getDauer());
266
267         // Wenn Endknoten wird FAZ auf den maximalen FEZ der Vorgängerknoten gesetzt
268         if (aktKnoten.getNachfolger().size() == 0) {
269             aktKnoten.setFaz(this.getMaxFezOfVorgaenger(aktKnoten));
270         }
271
272         for (Knoten nachfolger : aktKnoten.getNachfolger()) {
273             nachfolger.setFaz(this.getMaxFezOfVorgaenger(nachfolger));
274             setFezAndFaz(nachfolger);
275         }
276     }
277
278     /**
279      * Berechnet SAZ für den aktuell betrachteten Knoten sowie alle Vorgängerknoten,
280      * ausgehend vom aktuell betrachteten Knoten
281      *
282      * @param aktKnoten
283      *             aktuell betrachteter Knoten
284      */
285     private void setSazAndSez(Knoten aktKnoten) {
286         // Wenn aktueller Knoten ein Anfangsknoten ist, so wird Sez als minimaler SAZ
287         // der Nachfolger gesetzt
288         if (aktKnoten.getVorgaenger().size() == 0) {
289             aktKnoten.setSez(this.getMinSazOfNachfolger(aktKnoten));
290         }
291
292         // SAZ = SEZ      Dauer.
293         aktKnoten.setSaz(aktKnoten.getSez() - aktKnoten.getDauer());
294
295         for (Knoten vorgaenger : aktKnoten.getVorgaenger()) {
296             /*
297              * Der SAZ eines Vorgangs wird SEZ aller unmittelbarer Vorga nger
298              *
299              * Haben mehrere Vorga nge einen gemeinsamen Vorga nger, so ist dessen SEZ der
300              * fru heste (kleinste) SAZ aller Nachfolger.
301              */
302
303             vorgaenger.setSez(this.getMinSazOfNachfolger(vorgaenger));
304             // Rufe setSazAndSez rekursiv fpr alle vorgänger vom aktuellen Knoten auf
305             setSazAndSez(vorgaenger);
306         }
307     }
308
309     // /**
310     // * Setzt FAZ für alle Knoten ausgehend von einem aktuellen Knoten
311     // *
312     // * @param aktKnoten
313     // * aktuell betrachteter Knoten
314     // */
315     // private void setFaz(Knoten aktKnoten) {
316     // /*
317     // * Der FEZ eines Vorga ngers ist FAZ aller unmittelbar nachfolgenden Knoten.
318     // * Mu nden mehrere Knoten in einen Vorgang, dann ist der FAZ der gro ßte
319     // * (spa teste) FEZ der unmittelbaren Vorga nger.
320     // */
321     // aktKnoten.setFaz(this.getMaxFezOfVorgaenger(aktKnoten));
322     //
323     // // Rufe setFaz für alle nachfolgenden Knoten von aktKnoten auf
324     // for (Knoten nachfolger : aktKnoten.getNachfolger()) {
325     // setFaz(nachfolger);
326     // }
327     // }
328
329     /**
```

```
330         * Berechnet den Maximalen FEZ aller Vorgänger eines Knotens
331         *
332         * @param aktKnoten
333         *            aktuell betrachteter Knoten
334         * @return maximalen FEZ aller Vorgänger des Knoten
335         */
336        private int getMaxFezOfVorgaenger(Knoten aktKnoten) {
337            int max = Integer.MIN_VALUE;
338            for (Knoten vorgaenger : aktKnoten.getVorgaenger()) {
339                if (vorgaenger.getFez() > max) {
340                    max = vorgaenger.getFez();
341                }
342            }
343            return max;
344        }
345
346        // /**
347        // * Berechnet SEZ ausgehend von einem aktuellen Knoten
348        // *
349        // * @param aktKnoten
350        // * aktuell betrachteter Knoten
351        // */
352        // private void setSez(Knoten aktKnoten) {
353        // /*
354        // * Der SAZ eines Vorgangs wird SEZ aller unmittelbarer Vorga nger
355        // *
356        // * Haben mehrere Vorga nge einen gemeinsamen Vorga nger, so ist dessen SEZ der
357        // * fru heste (kleinste) SAZ aller Nachfolger.
358        // **/
359        // aktKnoten.setSez(this.getMinSazOfNachfolger(aktKnoten));
360        //
361        // for (Knoten vorgaenger : aktKnoten.getVorgaenger()) {
362        // setSez(vorgaenger);
363        // }
364        // }
365
366        /**
367         * Berechnet den minimalen SAZ der Nachfolgenden Knoten eines betrachteten
368         * Knoten
369         *
370         * @param aktKnoten
371         *            aktuell betrachteter Knoten
372         * @return minimaler SAZ der Nachfolgenden Knoten eines betrachteten Knoten
373         */
374        private int getMinSazOfNachfolger(Knoten aktKnoten) {
375            int min = Integer.MAX_VALUE;
376            for (Knoten nachfolger : aktKnoten.getNachfolger()) {
377                if (nachfolger.getSaz() < min) {
378                    min = nachfolger.getSaz();
379                }
380            }
381            return min;
382        }
383
384        /**
385         * Berechnet den GP aller Knoten ausgehend vom aktuell betrachteten Knoten
386         *
387         * @param aktKnoten
388         *            aktuell betrachteter Knoten
389         */
390        private void setGp(Knoten aktKnoten) {
391            /*
392             * Berechnung des Gesamtpuffers fu r jeden Knoten: GP = SAZ     FAZ = SEZ     FEZ
393             */
394            aktKnoten.setGp(aktKnoten.getSaz() - aktKnoten.getFaz());
395            for (Knoten nachfolger : aktKnoten.getNachfolger()) {
396                setGp(nachfolger);
397            }
398        }
399
400        /**
401         * Berechnet den FP aller Knoten ausgehend vom aktuell betrachteten Knoten
402         *
403         * @param aktKnoten
404         *            aktuell betrachteter Knoten
405         */
406        private void setFp(Knoten aktKnoten) {
407            /*
408             * Fu r die Berechnung des freien Puffers gilt: FP= (kleinster FAZ der
409             * nachfolgenden Knoten) - FEZ Ist der aktuelle Knoten der Endknoten, so ist der
410             * Freie Puffer 0, da FAZ==FEZ
411             */
412            aktKnoten.setFp(this.getMinFazOfNachfolger(aktKnoten) - aktKnoten.getFez());
413            for (Knoten nachfolger : aktKnoten.getNachfolger()) {
```

```
414                 setFp(nachfolger);
415             }
416         }
417 
418         /**
419          * Berechnet den kleinsten FAZ aller Nachfolger eines betrachteten Knoten
420          *
421          * @param aktKnoten
422          *            aktuell betrachteter Knoten
423          * @return kleinste FAZ aller Nachfolger eines betrachteten Knoten
424          */
425         private int getMinFazOfNachfolger(Knoten aktKnoten) {
426             int min = Integer.MAX_VALUE;
427             if (aktKnoten.getNachfolger().size() == 0) {
428                 return aktKnoten.getFez();
429             }
430             for (Knoten nachfolger : aktKnoten.getNachfolger()) {
431                 if (nachfolger.getFaz() < min) {
432                     min = nachfolger.getFaz();
433                 }
434             }
435             return min;
436         }
437 
438         /**
439          * Berechnet die Kritischen Pfade eines Netzplans und setzt sie im Model als
440          * kritischePfade
441          */
442         private void setKritischePfade() {
443             this.model.setKritischePfade(new ArrayList<>());
444             /*
445              * Bestimmung der kritischen Vorga nge ausgehend von jedem Startknoten
446              */
447             for (Knoten startK : this.model.getStartknoten()) {
448                 ArrayList<Knoten> pfad = new ArrayList<>();
449                 setKritischePfadeHelper(pfad, startK);
450             }
451         }
452 
453         /**
454          * Rekursive Hilfsmethode zur Berechnung der Kritischen Pfade nach dem Prinzip
455          * des Backtracking. Fügt bei erreichen des Endknotens den berechneten Pfad zum
456          * kritischePfade-Array im Model hinzu
457          *
458          * @param pfad
459          *            aktuell berechneter Pfad
460          * @param aktKnoten
461          *            aktuell betrachteter Knoten
462          */
463         private void setKritischePfadeHelper(ArrayList<Knoten> pfad, Knoten aktKnoten) {
464             /*
465              * Abbruchkriterium:Endknoten ist erreicht
466              */
467             if (aktKnoten.getNachfolger().size() == 0) {
468                 // Füge aktuellen Knoten in pfad ein
469                 pfad.add(aktKnoten);
470                 // Erstell Kopie des kritischen Pfades
471                 @SuppressWarnings("unchecked")
472                 ArrayList<Knoten> pfadKopie = (ArrayList<Knoten>) pfad.clone();
473                 // Füge errechneten Kritischen Pfad zu den im Model gekapselten Kritischen
474                 // Pfaden hinzu
475                 model.getKritischePfade().add(pfadKopie);
476                 // Breche die Mathode ab
477                 return;
478             }
479             /*
480              * Bestimmung der kritischen Vorga nge, d.h. GP = 0 und FP = 0
481              */
482             if (aktKnoten.getGp() == 0 && aktKnoten.getFp() == 0) {
483                 // füge aktuellen Knoten zum kritischen Pfad hinzu
484                 // pfad.add(aktKnoten);
485                 @SuppressWarnings("unchecked")
486                 ArrayList<Knoten> pfadKopie = (ArrayList<Knoten>) pfad.clone();
487                 pfadKopie.add(aktKnoten);
488                 // Führe für alle Nachfolger rekursiv die Methode setKritischePfadehelper aus
489                 // und durchlaufe so nach Backtraking den virtuellen Baum
490                 for (Knoten nachfolger : aktKnoten.getNachfolger()) {
491                     this.setKritischePfadeHelper(pfadKopie, nachfolger);
492                 }
493                 // // Entferne den zuletzt hinzugefügten Knoten aus dem Pfad-Array
494                 // pfad.remove(pfad.size() - 1);
495             }
496         }
497 
```

```
498         /**
499          * Prüft, ob alle Referenzen in model.knoten korrekt sind, also ob jeder
500          * Nachfolger auch in dessen Vorgaengern enthalten ist bzw. umgekehrt.
501          *
502          * Darf erst nach der Prüfung der Zyklen aufgerufen werden!
503          *
504          * @return true, falls alle Referenzen korrekt sind, sonst false.
505          */
506         boolean hatGueltigeReferenzen() {
507             for (Knoten k1 : this.model.getKnoten()) {
508                 for (Knoten nachfolger : k1.getNachfolger()) {
509                     if (!nachfolger.getVorgaenger().contains(k1)) {
510                         return false;
511                     }
512                 }
513             }
514 
515             for (Knoten k1 : this.model.getKnoten()) {
516                 for (Knoten vorgaenger : k1.getVorgaenger()) {
517                     if (!vorgaenger.getNachfolger().contains(k1)) {
518                         return false;
519                     }
520                 }
521             }
522 
523             return true;
524         }
525     }
```

9.4 Package model

9.4.1 Klasse Knoten

```
1   package model;
2
3   import java.util.ArrayList;
4
5   /**
6    *
7    * @author M. Leonard Haufs Prüflingsnummer: 101-20540
8    *
9    */
10  public class Knoten {
11
12      private int vorgangsnummer;
13      private String vorgangsbezeichnung;
14
15      private int faz;
16      private int fez;
17      private int dauer;
18      private int gp;
19      private int fp;
20      private int saz;
21      private int sez;
22
23      ArrayList<Knoten> vorgaenger;
24      ArrayList<Integer> vorgaengerNummern;
25
26      ArrayList<Knoten> nachfolger;
27      ArrayList<Integer> nachfolgerNummern;
28
29      // Getter und Setter
30      public int getVorgangsnummer() {
31          return vorgangsnummer;
32      }
33      public void setVorgangsnummer(int vorgangsnummer) {
34          this.vorgangsnummer = vorgangsnummer;
35      }
36      public String getVorgangsbezeichnung() {
37          return vorgangsbezeichnung;
38      }
39      public void setVorgangsbezeichnung(String vorgangsbezeichnung) {
40          this.vorgangsbezeichnung = vorgangsbezeichnung;
41      }
42      public int getFaz() {
43          return faz;
44      }
45      public void setFaz(int faz) {
46          this.faz = faz;
47      }
48      public int getFez() {
49          return fez;
50      }
51      public void setFez(int fez) {
52          this.fez = fez;
53      }
54      public int getDauer() {
55          return dauer;
56      }
57      public void setDauer(int dauer) {
58          this.dauer = dauer;
59      }
60      public int getGp() {
61          return gp;
62      }
63      public void setGp(int gp) {
64          this.gp = gp;
65      }
66      public int getFp() {
67          return fp;
68      }
69      public void setFp(int fp) {
70          this.fp = fp;
71      }
72      public int getSaz() {
73          return saz;
74      }
75      public void setSaz(int saz) {
76          this.saz = saz;
77      }
78      public int getSez() {
```

```
79          return sez;
80      }
81      public void setSez(int sez) {
82          this.sez = sez;
83      }
84      public ArrayList<Knoten> getVorgaenger() {
85          return vorgaenger;
86      }
87      public void setVorgaenger(ArrayList<Knoten> vorgaenger) {
88          this.vorgaenger = vorgaenger;
89      }
90      public ArrayList<Integer> getVorgaengerNummern() {
91          return vorgaengerNummern;
92      }
93      public void setVorgaengerNummern(ArrayList<Integer> vorgaengerNummern) {
94          this.vorgaengerNummern = vorgaengerNummern;
95      }
96      public ArrayList<Knoten> getNachfolger() {
97          return nachfolger;
98      }
99      public void setNachfolger(ArrayList<Knoten> nachfolger) {
100         this.nachfolger = nachfolger;
101     }
102     public ArrayList<Integer> getNachfolgerNummern() {
103         return nachfolgerNummern;
104     }
105     public void setNachfolgerNummern(ArrayList<Integer> nachfolgerNummern) {
106         this.nachfolgerNummern = nachfolgerNummern;
107     }
108
109     // Konstruktor
110     public Knoten(int vorgangsnummer, String vorgangsbezeichnung, int dauer,
            ArrayList<Integer> vorgaengerNummern, ArrayList<Integer> nachfolgerNummern) {
111         super();
112
113         this.vorgangsnummer = vorgangsnummer;
114         this.vorgangsbezeichnung = vorgangsbezeichnung;
115         this.dauer = dauer;
116         this.vorgaengerNummern = vorgaengerNummern;
117         this.nachfolgerNummern = nachfolgerNummern;
118
119         this.vorgaenger = new ArrayList<>();
120         this.nachfolger = new ArrayList<>();
121     }
122 }
```

9.4.2 Klasse Model

```
1   package model;
2
3   import java.util.ArrayList;
4
5   /**
6    *
7    * @author M. Leonard Haufs Prüflingsnummer: 101-20540
8    *
9    */
10  public class Model {
11      private boolean initialized;
12
13      private ArrayList<Knoten> startknoten;
14      private ArrayList<Knoten> endknoten;
15
16      private ArrayList<Knoten> knoten;
17
18      private ArrayList<ArrayList<Knoten>> kritischePfade;
19      private ArrayList<Knoten> zyklus;
20      private boolean isZusammenhaengend;
21      private boolean gueltigeReferenzen;
22
23      private String name;
24
25      public boolean isInitialized() {
26          return initialized;
27      }
28
29      public void initialize() {
30          this.initialized = true;
31      }
32
33      public boolean isZusammenhaengend() {
34          return isZusammenhaengend;
35      }
36
37      public void setZusammenhaengend(boolean isZusammenhaengend) {
38          this.isZusammenhaengend = isZusammenhaengend;
39      }
40
41      public boolean isGueltigeReferenzen() {
42          return gueltigeReferenzen;
43      }
44
45      public void setGueltigeReferenzen(boolean gueltigeReferenzen) {
46          this.gueltigeReferenzen = gueltigeReferenzen;
47      }
48
49      // Getter und Setter
50      public ArrayList<ArrayList<Knoten>> getKritischePfade() {
51          return kritischePfade;
52      }
53
54      public void setKritischePfade(ArrayList<ArrayList<Knoten>> kritischePfade) {
55          this.kritischePfade = kritischePfade;
56      }
57
58      public ArrayList<Knoten> getZyklus() {
59          return zyklus;
60      }
61
62      public void setZyklus(ArrayList<Knoten> zyklus) {
63          this.zyklus = zyklus;
64      }
65
66      public ArrayList<Knoten> getStartknoten() {
67          return startknoten;
68      }
69
70      public ArrayList<Knoten> getEndknoten() {
71          return endknoten;
72      }
73
74      public ArrayList<Knoten> getKnoten() {
75          return knoten;
76      }
77
78      public String getName() {
79          return name;
80      }
81
```

```
82      // Konstruktoren
83
84      public Model() {
85          super();
86          this.knoten = new ArrayList<>();
87          this.name = "not set";
88
89          this.startknoten = new ArrayList<>();
90          this.endknoten = new ArrayList<>();
91
92          this.kritischePfade = new ArrayList<>();
93          this.zyklus = new ArrayList<>();
94          this.gueltigeReferenzen = true;
95      }
96
97      public Model(ArrayList<Knoten> knoten, String name) {
98          this();
99          this.knoten = knoten;
100         this.name = name;
101
102         this.initKnoten(knoten);
103         this.startknoten = this.getStartknoten(knoten);
104         this.endknoten = this.getEndknoten(knoten);
105     }
106
107     private ArrayList<Knoten> getStartknoten(ArrayList<Knoten> knoten) {
108         ArrayList<Knoten> startknoten = new ArrayList<>();
109         for (Knoten k : knoten) {
110             if (k.getVorgaengerNummern().size() == 0) {
111                 startknoten.add(k);
112             }
113         }
114
115         return startknoten;
116     }
117
118     private ArrayList<Knoten> getEndknoten(ArrayList<Knoten> knoten) {
119         ArrayList<Knoten> endknoten = new ArrayList<>();
120         for (Knoten k : knoten) {
121             if (k.getNachfolgerNummern().size() == 0) {
122                 endknoten.add(k);
123             }
124         }
125
126         return endknoten;
127     }
128
129     private void initKnoten(ArrayList<Knoten> knoten) {
130         for (Knoten k : knoten) {
131             for (int vorgaengerNr : k.getVorgaengerNummern()) {
132                 for (Knoten k2 : knoten) {
133                     if (k2.getVorgangsnummer() == vorgaengerNr) {
134                         k.getVorgaenger().add(k2);
135                     }
136                 }
137             }
138
139             for (int nachfolgerNr : k.getNachfolgerNummern()) {
140                 for (Knoten k2 : knoten) {
141                     if (k2.getVorgangsnummer() == nachfolgerNr) {
142                         k.getNachfolger().add(k2);
143                     }
144                 }
145             }
146         }
147     }
148
149 }
```