# 9 Anhang: Programmcode

## 9.1 Package main

### 9.1.1 Klasse Main

```java
package main;

import java.io.File;
import java.io.IOException;

import controller.Controller;
import io.AusgabeInDatei;
import io.LeseAusDatei;
import model.Model;

/**
 * @author M. Leonard Haufs Prüflingsnummer: 101-20540
 *
 */
public class Main {
    public static void main(String[] args) {
        String dateiendung;
        String verzeichnis;

        // Parameterübergabe prüfen
        if (args.length != 2) {
            // keine korrekte Parameterübergabe
            System.out.println(
                    "Es müssen 2 Parameter übergeben werden. Paramter 1: Endung der Eingabedateien (z.B.: .in)\nParameter 2: Verzechnis aus dem die Eingabedateien gelesen werden soll.");
            return;
        }
        dateiendung = args[0];
        verzeichnis = args[1];

        File f;
        try {
            try {
                f = new File(verzeichnis);
            } catch (Exception ex) {
                throw new IOException("Der Angegebene Pfad existiert nicht");
            }

            if (f.isDirectory() && f.canRead()) {
                File[] dateien = f.listFiles();
                for (int i = 0; i < dateien.length; i++) {
                    // Prüfe ob die Datei gelesen werden kann
                    if (dateien[i].isFile() && dateien[i].canRead()) {
                        String tempEndung =
                                dateien[i].getName().substring(dateien[i].getName().lastIndexOf("."),
                                        dateien[i].getName().length());
                        // wenn die Dateiendung der gewählten entspricht
                        // wird die Datei eingelesen
                        if (dateiendung.equals(tempEndung)) {

                            // Eingabe

                            LeseAusDatei in = new LeseAusDatei();
                            Model model = in.getModelAusDatei(dateien[i]);

                            // Berechnung
                            Controller c = new Controller(model);
                            c.calculate();

                            // Ausgabe
                            AusgabeInDatei out = new AusgabeInDatei(model);

                            String outputPath = verzeichnis + "/" +
                                    (dateien[i].getName().replace(dateiendung, ".out"));
                            out.schreibeModelInDatei(outputPath);

                            // OutputConsole out = new OutputConsole();
                            // out.printEntireOutputString(model);
                        }
                    }
                }
                System.out.println(args[1] + ": Vorgang abgeschlossen.");
            } else {
                throw new IOException("Der Angegebene Pfad ist kein Ordner oder kann nicht geöffnet werden.");
            }
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }

    }
}
```

## 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("In Datei " + file.getName() + "Ungenügende Eingabe: Datei konnte
                         nicht geöffnet werden");
            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);
                    }
                }

                ArrayList<Integer> nachfolgerNummern = new ArrayList<>();
                if (!zeileSplit[4].equals("-")) {
                    String[] nachfolgerNummernArr = zeileSplit[4].split(",");
                    for (int i = 0; i < nachfolgerNummernArr.length; i++) {
                        String string = nachfolgerNummernArr[i];
                        int number = Integer.parseInt(string);
                        nachfolgerNummern.add(number);
                    }
                }
                // Prüfe, ob vorgangsnummern nicht doppelt vorliegen
                if (!vorgangsnummernNichtDoppelt(vorgangsnummern)) {
                    System.out.println("In Datei " + file.getName()
                            + ": Ungenügende Eingabe: Es kommt mindestens eine Vorgangsnummer
                                 mehrfach vor.");
                    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("In Datei " + file.getName() + "Ungenügende Einabe: Eingabestruktur
                    nicht erfüllt");
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         } catch (Exception e) {
109             System.out.println("In Datei " + file.getName() + ": Ungenügende Eingabe.");
110             return new Model();
111         }
112         if (!alleKnotenVerweisenAufExistierendenKnoten(knoten, vorgangsnummern)) {
113             System.out.println("In Datei " + file.getName()
114                     + ": Ungenügende Eingabe: Es existieren ungültige Referenzen, da mindestens ein
                            Knoten auf einen nicht existenten Knoten referenziert.");
115             return new Model();
116         }
117         if (knoten.size() == 0) {
118             System.out.println(
119                     "In Datei " + file.getName() + ": Ungenügende Eingabe: Es wurden keinerlei
                            Vorgänge angegeben.");
120             return new Model();
121         }
122         Model model = new Model(knoten, kommentar);
123         return model;
124     }
125
126     /**
127      * Prüft, ob die Vorgangsnummern nicht doppelt vorliegen
128      *
129      * @param vorgangsnummern
130      *            die zu Prüfen sind
131      * @return true, falls die Vorgangsnummern nicht doppelt vorliegen
132      */
133     private boolean vorgangsnummernNichtDoppelt(ArrayList<Integer> vorgangsnummern) {
134         @SuppressWarnings("unchecked")
135         ArrayList<Integer> copyOfVorgangsnummern = (ArrayList<Integer>) vorgangsnummern.clone();
136
137         for (int i = 0; i < copyOfVorgangsnummern.size(); i++) {
138             Integer vorgangsnummer = copyOfVorgangsnummern.get(i);
139             copyOfVorgangsnummern.remove(vorgangsnummer);
140             if (copyOfVorgangsnummern.contains(Integer.valueOf(vorgangsnummer))) {
141                 return false;
142             }
143         }
144         return true;
145     }
146
147     /**
148      * Prüft, ob alle Knoten auf einen existierenden Knoten verweisen.
149      *
150      * @param knoten
151      *            Knotenliste, der zu prüfenden Knoten
152      * @param vorgangsnummern
153      *            Liste der Vorgangsnummern aller Knoten
154      * @return true, falls alle Knoten auf einen existierenden Knoten verweisen,
155      *         sonst false
156      */
157     private boolean alleKnotenVerweisenAufExistierendenKnoten(ArrayList<Knoten> knoten,
158             ArrayList<Integer> vorgangsnummern) {
159         for (Knoten k : knoten) {
160             for (int nachfolgernummer : k.getNachfolgerNummern()) {
161                 if (!vorgangsnummern.contains(Integer.valueOf(nachfolgernummer))) {
162                     return false;
163                 }
164             }
165
166             for (int vorgaengernummer : k.getVorgaengerNummern()) {
167                 if (!vorgangsnummern.contains(Integer.valueOf(vorgaengernummer))) {
168                     return false;
169                 }
170             }
171
172         }
173         return true;
174     }
175 }
```

### 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 AusgabeInDatei

```
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 Controller

```java
package controller;

import java.util.ArrayList;

import model.Knoten;
import model.Model;

/**
 * Hauptberechnungsklasse.
 *
 * @author M. Leonard Haufs Prüflingsnummer: 101-20540
 *
 */
public class Controller {
    private Model model;
    private ArrayList<Knoten> validationsListe;

    // Konstruktor
    public Controller(Model model) {
        super();
        this.model = model;
    }

    /**
     * Hauptberechnungsmethode des Controllers.
     *
     * Falls noch nicht initialisiert wurde, wird auf Zyklen und Zusammenhängigkeit
     * geprüft . Falls der Netzplan Zyklen enthällt, wird im Model in zyklus ein
     * zyclus gespeichert. Wenn der Netzplan nicht nicht zusammenhängend ist, wird
     * im Model isZusammenhaengend auf false gesetzt. Sonst auf true.
     *
     * Anschließend wird das Model initialisiert, also die kenngrößen berechnet und
     * anschließend der kritische Pfad, falls er existiert, berechnet
     */
    public void calculate() {
        // Prüfe, ob der im Model gekapselte Netzplan keine Zyklen enthällt
        boolean hatKeineZyklen = this.hatKeineZyklen();
        if (!hatKeineZyklen) {
            System.out.println(this.model.getName() + ": Zyklen enthalten");
            return;
        }

        // Prüfe, ob der im Model gekapselte Netzplan zusammenhängend ist
        boolean isZusammenhaengend = this.isZusammenhaengend();
        if (!isZusammenhaengend) {
            System.out.println(this.model.getName() + ": Fehler (Nicht zusammenhängend)");
            model.setZusammenhaengend(false);
            return;
        } else {
            model.setZusammenhaengend(true);
        }

        // Prüft, ob alle Referenzen in model.knoten korrekt sind, also ob jeder
        // Nachfolger auch in dessen Vorgaengern enthalten ist bzw. umgekehrt.
        boolean hatGueltigeReferenzen = this.hatGueltigeReferenzen();
        if (!hatGueltigeReferenzen) {
            System.out.println(this.model.getName()
                    + "Referenzen der Eingabe sind ungenügend. nicht jeder Nachfolger auch in
                        dessen Vorgaengern enthalten bzw. umgekehrt ist.");
            model.setGueltigeReferenzen(false);
        } else {
            model.setGueltigeReferenzen(true);
        }

        // Initialisiere das Model
        if (!this.model.isInitialized()) {
            initModel();
        }
    }

    /**
     * Prüft, ob der im Model gekapselte Netzplan keine Zyklen enthällt
     *
     * @return true, falls der Netzplan im Model keine Zyklen enthällt, sonst true
     */
    boolean hatKeineZyklen() {
        ArrayList<Boolean> check = new ArrayList<>();

        /*
         * Rufe für ausgehend von allen Startknoten die Helpermethode
         * hatKeineZyklenHelper auf. Falls ein Ergebnis negativ ausfällt wird false
         * zurückgegeben
         */
        for (Knoten s : this.model.getStartknoten()) {
            this.validationsListe = new ArrayList<>();
            check.add(hatKeineZyklenHelper(s));
            if (check.contains(Boolean.valueOf(false))) {
                model.setZyklus(this.validationsListe);
                return false;
            }
        }
        return true;
    }

    /**
     * Hilfsfunktion zur Überprüfung, ob keine Zyklen existieren
     *
```

```
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     }
```

### 9.3.2 Unittest Klasse Controller

```
1  package controller;
2
3  import static org.junit.Assert.assertEquals;
4
5  import java.util.ArrayList;
6
7  import org.junit.Test;
8
9  import model.Knoten;
10 import model.Model;
11
12 public class ControllerTest {
13     @Test
14     public void hatKeineZyklen_ModelOhneZyklen_RueckgabeTrue() {
15         // Arrangieren
16         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
17         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
18         ersterKnotenNachfolger.add(2);
19         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, new ArrayList<>(),
               ersterKnotenNachfolger);
20         knotenliste.add(ersterKnoten);
21         //
22         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
23         zweiterKnotenVorgaenger.add(1);
24         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger, new
               ArrayList<>());
25         knotenliste.add(zweiterKnoten);
26         Model model = new Model(knotenliste, "Testliste");
27         //
28         Controller controller = new Controller(model);
29
30         // Ausführen
31         boolean keineZyklen = controller.hatKeineZyklen();
32
33         // Auswerten
34         assertEquals(true, keineZyklen);
35     }
36
37     @Test
38     public void hatKeineZyklen_ZweiterKnotenHatErstenKnotenAlsNachfolger_RueckgabeFalse() {
39         // Arrangieren
40         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
41         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
42         ersterKnotenNachfolger.add(2);
43         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, new ArrayList<>(),
               ersterKnotenNachfolger);
44         knotenliste.add(ersterKnoten);
45         //
46         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
47         zweiterKnotenVorgaenger.add(1);
48         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
49         zweiterKnotenNachfolger.add(1);
50         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
               zweiterKnotenNachfolger);
51         knotenliste.add(zweiterKnoten);
52         Model model = new Model(knotenliste, "Testliste");
53         //
54         Controller controller = new Controller(model);
55
56         // Ausführen
57         boolean keineZyklen = controller.hatKeineZyklen();
58
59         // Auswerten
60         assertEquals(false, keineZyklen);
61     }
62
63     @Test
64     public void
           hatKeineZyklen_ErsterKnotenHatZweitenKnotenAlsVorgaengerSowieNachfolgerUndZweiterKnotenHatErstenKnotenAlsVor
           {
65         // Arrangieren
66         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
67         //
68         ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
69         ersterKnotenVorgaenger.add(2);
70         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
71         ersterKnotenNachfolger.add(2);
72         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
               ersterKnotenNachfolger);
73         knotenliste.add(ersterKnoten);
74         //
75         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
76         zweiterKnotenVorgaenger.add(1);
77         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
78         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
               zweiterKnotenNachfolger);
79         knotenliste.add(zweiterKnoten);
80         //
81         Model model = new Model(knotenliste, "Testliste");
82         //
83         Controller controller = new Controller(model);
84
85         // Ausführen
86         boolean keineZyklen = controller.hatKeineZyklen();
87
88         // Auswerten
89         assertEquals(true, keineZyklen);
90     }
91
92     @Test
93     public void
```

```
            hatKeineZyklen_ZweiKnotenHabenSichGegenseitigAlsNachfolgerSowieVorgaenger_RueckgabeTrueDaKeinExistierenderSt
            {
94          // Arrangieren
95          ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
96          //
97          ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
98          ersterKnotenVorgaenger.add(2);
99          ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
100         ersterKnotenNachfolger.add(2);
101         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
                ersterKnotenNachfolger);
102         knotenliste.add(ersterKnoten);
103         //
104         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
105         zweiterKnotenVorgaenger.add(1);
106         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
107         zweiterKnotenNachfolger.add(1);
108         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                zweiterKnotenNachfolger);
109         knotenliste.add(zweiterKnoten);
110         //
111         Model model = new Model(knotenliste, "Testliste");
112         //
113         Controller controller = new Controller(model);
114
115         // Ausführen
116         boolean keineZyklen = controller.hatKeineZyklen();
117
118         // Auswerten
119         assertEquals(true, keineZyklen);
120     }
121
122     @Test
123     public void hatKeineZyklen_DritterKnotenHatZweitenKnotenAlsNachfolger_RueckgabeFalse() {
124         // Arrangieren
125         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
126         //
127         ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
128         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
129         ersterKnotenNachfolger.add(2);
130         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
                ersterKnotenNachfolger);
131         knotenliste.add(ersterKnoten);
132         //
133         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
134         zweiterKnotenVorgaenger.add(1);
135         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
136         zweiterKnotenNachfolger.add(3);
137         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                zweiterKnotenNachfolger);
138         knotenliste.add(zweiterKnoten);
139         //
140         ArrayList<Integer> dritterKnotenVorgaenger = new ArrayList<Integer>();
141         dritterKnotenVorgaenger.add(2);
142         ArrayList<Integer> dritterKnotenNachfolger = new ArrayList<Integer>();
143         dritterKnotenNachfolger.add(2);
144         Knoten dritterKnoten = new Knoten(3, "Dritter Schritt", 10, dritterKnotenVorgaenger,
                dritterKnotenNachfolger);
145         knotenliste.add(dritterKnoten);
146         //
147         Model model = new Model(knotenliste, "Testliste");
148         //
149         Controller controller = new Controller(model);
150
151         // Ausführen
152         boolean keineZyklen = controller.hatKeineZyklen();
153
154         // Auswerten
155         assertEquals(false, keineZyklen);
156     }
157
158     @Test
159     public void isZusammenhaengend_ZusammenhaengendeKnoten_RueckgabeTrue() {
160         // Arrangieren
161         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
162         //
163         ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
164         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
165         ersterKnotenNachfolger.add(2);
166         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
                ersterKnotenNachfolger);
167         knotenliste.add(ersterKnoten);
168         //
169         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
170         zweiterKnotenVorgaenger.add(1);
171         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
172         zweiterKnotenNachfolger.add(3);
173         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                zweiterKnotenNachfolger);
174         knotenliste.add(zweiterKnoten);
175         //
176         ArrayList<Integer> dritterKnotenVorgaenger = new ArrayList<Integer>();
177         dritterKnotenVorgaenger.add(2);
178         ArrayList<Integer> dritterKnotenNachfolger = new ArrayList<Integer>();
179         Knoten dritterKnoten = new Knoten(3, "Dritter Schritt", 10, dritterKnotenVorgaenger,
                dritterKnotenNachfolger);
180         knotenliste.add(dritterKnoten);
181         //
182         Model model = new Model(knotenliste, "Testliste");
183         //
184         Controller controller = new Controller(model);
185
186         // Ausführen
```

```
187             boolean zusammenhaengend = controller.isZusammenhaengend();
188
189             // Auswerten
190             assertEquals(true, zusammenhaengend);
191         }
192
193         @Test
194         public void
                isZusammenhaengend_DritterKnotenHatEinenVorgaengerAberDieserKeinenNachfolger_RueckgabeFalse()
                {
195             // Arrangieren
196             ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
197             //
198             ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
199             ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
200             ersterKnotenNachfolger.add(2);
201             Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
                    ersterKnotenNachfolger);
202             knotenliste.add(ersterKnoten);
203             //
204             ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
205             zweiterKnotenVorgaenger.add(1);
206             ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
207             Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                    zweiterKnotenNachfolger);
208             knotenliste.add(zweiterKnoten);
209             //
210             ArrayList<Integer> dritterKnotenVorgaenger = new ArrayList<Integer>();
211             dritterKnotenVorgaenger.add(2);
212             ArrayList<Integer> dritterKnotenNachfolger = new ArrayList<Integer>();
213             Knoten dritterKnoten = new Knoten(3, "Dritter Schritt", 10, dritterKnotenVorgaenger,
                    dritterKnotenNachfolger);
214             knotenliste.add(dritterKnoten);
215             //
216             Model model = new Model(knotenliste, "Testliste");
217             //
218             Controller controller = new Controller(model);
219
220             // Ausführen
221             boolean zusammenhaengend = controller.isZusammenhaengend();
222
223             // Auswerten
224             assertEquals(false, zusammenhaengend);
225         }
226
227         @Test
228         public void
                hatGueltigeReferenzen_dreiKnotenMitFehlenderReferenzVomZweitenZumDrittenKnoten_nichtGueltig()
                {
229             // Arrangieren
230             ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
231             //
232             ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
233             ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
234             ersterKnotenNachfolger.add(2);
235             Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
                    ersterKnotenNachfolger);
236             knotenliste.add(ersterKnoten);
237             //
238             ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
239             zweiterKnotenVorgaenger.add(1);
240             ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
241             Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                    zweiterKnotenNachfolger);
242             knotenliste.add(zweiterKnoten);
243             //
244             ArrayList<Integer> dritterKnotenVorgaenger = new ArrayList<Integer>();
245             dritterKnotenVorgaenger.add(2);
246             ArrayList<Integer> dritterKnotenNachfolger = new ArrayList<Integer>();
247             Knoten dritterKnoten = new Knoten(3, "Dritter Schritt", 10, dritterKnotenVorgaenger,
                    dritterKnotenNachfolger);
248             knotenliste.add(dritterKnoten);
249             //
250             Model model = new Model(knotenliste, "Testliste");
251             //
252             Controller controller = new Controller(model);
253
254             // Ausführen
255             boolean gueltig = controller.hatGueltigeReferenzen();
256
257             // Auswerten
258             assertEquals(false, gueltig);
259         }
260
261         @Test
262         public void hatGueltigeReferenzen_dreiKnotenMitKorrektGesetztenReferenzen_istGueltig() {
263             // Arrangieren
264             ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
265             //
266             ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
267             ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
268             ersterKnotenNachfolger.add(2);
269             Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
                    ersterKnotenNachfolger);
270             knotenliste.add(ersterKnoten);
271             //
272             ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
273             zweiterKnotenVorgaenger.add(1);
274             ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
275             zweiterKnotenNachfolger.add(3);
276             Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                    zweiterKnotenNachfolger);
277             knotenliste.add(zweiterKnoten);
```

```
278             //
279             ArrayList<Integer> dritterKnotenVorgaenger = new ArrayList<Integer>();
280             dritterKnotenVorgaenger.add(2);
281             ArrayList<Integer> dritterKnotenNachfolger = new ArrayList<Integer>();
282             Knoten dritterKnoten = new Knoten(3, "Dritter Schritt", 10, dritterKnotenVorgaenger,
                    dritterKnotenNachfolger);
283             knotenliste.add(dritterKnoten);
284             //
285             Model model = new Model(knotenliste, "Testliste");
286             //
287             Controller controller = new Controller(model);
288
289             // Ausführen
290             boolean gueltig = controller.hatGueltigeReferenzen();
291
292             // Auswerten
293             assertEquals(true, gueltig);
294         }
295     }
```

## 9.4 Package model

### 9.4.1 Klasse Knoten

```java
package model;

import java.util.ArrayList;

/**
 *
 * @author M. Leonard Haufs Prüflingsnummer: 101-20540
 *
 */
public class Knoten {

    private int vorgangsnummer;
    private String vorgangsbezeichnung;

    private int faz;
    private int fez;
    private int dauer;
    private int gp;
    private int fp;
    private int saz;
    private int sez;

    ArrayList<Knoten> vorgaenger;
    ArrayList<Integer> vorgaengerNummern;

    ArrayList<Knoten> nachfolger;
    ArrayList<Integer> nachfolgerNummern;

    // Getter und Setter
    public int getVorgangsnummer() {
        return vorgangsnummer;
    }
    public void setVorgangsnummer(int vorgangsnummer) {
        this.vorgangsnummer = vorgangsnummer;
    }
    public String getVorgangsbezeichnung() {
        return vorgangsbezeichnung;
    }
    public void setVorgangsbezeichnung(String vorgangsbezeichnung) {
        this.vorgangsbezeichnung = vorgangsbezeichnung;
    }
    public int getFaz() {
        return faz;
    }
    public void setFaz(int faz) {
        this.faz = faz;
    }
    public int getFez() {
        return fez;
    }
    public void setFez(int fez) {
        this.fez = fez;
    }
    public int getDauer() {
        return dauer;
    }
    public void setDauer(int dauer) {
        this.dauer = dauer;
    }
    public int getGp() {
        return gp;
    }
    public void setGp(int gp) {
        this.gp = gp;
    }
    public int getFp() {
        return fp;
    }
    public void setFp(int fp) {
        this.fp = fp;
    }
    public int getSaz() {
        return saz;
    }
    public void setSaz(int saz) {
        this.saz = saz;
    }
    public int getSez() {
        return sez;
    }
    public void setSez(int sez) {
        this.sez = sez;
    }
    public ArrayList<Knoten> getVorgaenger() {
        return vorgaenger;
    }
    public void setVorgaenger(ArrayList<Knoten> vorgaenger) {
        this.vorgaenger = vorgaenger;
    }
    public ArrayList<Integer> getVorgaengerNummern() {
        return vorgaengerNummern;
    }
    public void setVorgaengerNummern(ArrayList<Integer> vorgaengerNummern) {
        this.vorgaengerNummern = vorgaengerNummern;
    }
    public ArrayList<Knoten> getNachfolger() {
        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

```java
package model;

import java.util.ArrayList;

/**
 *
 * @author M. Leonard Haufs Prüflingsnummer: 101-20540
 *
 */
public class Model {
    private boolean initialized;

    private ArrayList<Knoten> startknoten;
    private ArrayList<Knoten> endknoten;

    private ArrayList<Knoten> knoten;

    private ArrayList<ArrayList<Knoten>> kritischePfade;
    private ArrayList<Knoten> zyklus;
    private boolean isZusammenhaengend;
    private boolean gueltigeReferenzen;

    private String name;

    public boolean isInitialized() {
        return initialized;
    }

    public void initialize() {
        this.initialized = true;
    }

    public boolean isZusammenhaengend() {
        return isZusammenhaengend;
    }

    public void setZusammenhaengend(boolean isZusammenhaengend) {
        this.isZusammenhaengend = isZusammenhaengend;
    }

    public boolean isGueltigeReferenzen() {
        return gueltigeReferenzen;
    }

    public void setGueltigeReferenzen(boolean gueltigeReferenzen) {
        this.gueltigeReferenzen = gueltigeReferenzen;
    }

    // Getter und Setter
    public ArrayList<ArrayList<Knoten>> getKritischePfade() {
        return kritischePfade;
    }

    public void setKritischePfade(ArrayList<ArrayList<Knoten>> kritischePfade) {
        this.kritischePfade = kritischePfade;
    }

    public ArrayList<Knoten> getZyklus() {
        return zyklus;
    }

    public void setZyklus(ArrayList<Knoten> zyklus) {
        this.zyklus = zyklus;
    }

    public ArrayList<Knoten> getStartknoten() {
        return startknoten;
    }

    public ArrayList<Knoten> getEndknoten() {
        return endknoten;
    }

    public ArrayList<Knoten> getKnoten() {
        return knoten;
    }

    public String getName() {
        return name;
    }

    // Konstruktoren

    public Model() {
        super();
        this.knoten = new ArrayList<>();
        this.name = "not set";

        this.startknoten = new ArrayList<>();
        this.endknoten = new ArrayList<>();

        this.kritischePfade = new ArrayList<>();
        this.zyklus = new ArrayList<>();
        this.gueltigeReferenzen = true;
    }

    public Model(ArrayList<Knoten> knoten, String name) {
        this();
        this.knoten = knoten;
        this.name = name;

```

```
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     }
```