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      * @param sb
147      *             StringBuilder, an den der Gesamtdauerstring angehängt werden soll.
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              * 2. Phase: Ru ckwa rtsrechnung
197              *
198              * Bei der Ru ckwa rtsrechnung wird ermittelt, wann die einzelnen Vorga nge
199              * spa testens begonnen und fertiggestellt sein mu ssen, damit die
```

```
200             * Gesamtprojektzeit nicht gefa hrdet ist.
201             *
202             * Fu r den letzten Vorgang ist der fru heste Endzeitpunkt (FEZ) auch der
203             * spa teste Endzeitpunkt (SEZ), also SEZ = FEZ.
204             */
205            for (Knoten endK : this.model.getEndknoten()) {
206                endK.setSez(endK.getFez());
207            }
208
209            /*
210             * Fu r den spa testen Anfangszeitpunkt gilt: SAZ = SEZ     Dauer.
211             */
212            for (Knoten endKnoten : this.model.getEndknoten()) {
213                this.setSazAndSez(endKnoten);
214            }
215
216            // 3. Phase: Ermittlung der Zeitreserven
217            for (Knoten startK : this.model.getStartknoten()) {
218                /*
219                 * Berechnung des Gesamtpuffers fu r jeden Knoten
220                 */
221                this.setGp(startK);
222
223                /*
224                 * Berechnung des freien Puffers
225                 */
226                this.setFp(startK);
227            }
228
229            /*
230             * Bestimmung der kritischen Vorga nges
231             */
232            this.setKritischePfade();
233
234            this.model.initialize();
235        }
236
237        /**
238         * Setzt FEZ und FAZ ausgehend von einem aktuellen Knoten für diesen und alle
239         * Nachfolger dieses Knotens
240         *
241         * @param aktKnoten
242         */
243        private void setFezAndFaz(Knoten aktKnoten) {
244            // Fu r den FEZ gilt: FEZ = FAZ + Dauer
245            aktKnoten.setFez(aktKnoten.getFaz() + aktKnoten.getDauer());
246
247            // Wenn Endknoten wird FAZ auf den maximalen FEZ der Vorgängerknoten gesetzt
248            if (aktKnoten.getNachfolger().size() == 0) {
249                aktKnoten.setFaz(this.getMaxFezOfVorgaenger(aktKnoten));
250            }
251
252            for (Knoten nachfolger : aktKnoten.getNachfolger()) {
253                nachfolger.setFaz(this.getMaxFezOfVorgaenger(nachfolger));
254                setFezAndFaz(nachfolger);
255            }
256        }
257
258        /**
259         * Berechnet SAZ für den aktuell betrachteten Knoten sowie alle Vorgängerknoten,
260         * ausgehend vom aktuell betrachteten Knoten
261         *
262         * @param aktKnoten
263         *            aktuell betrachteter Knoten
264         */
265        private void setSazAndSez(Knoten aktKnoten) {
266            // Wenn aktueller Knoten ein Anfangsknoten ist, so wird Sez als minimaler SAZ
267            // der Nachfolger gesetzt
268            if (aktKnoten.getVorgaenger().size() == 0) {
269                aktKnoten.setSez(this.getMinSazOfNachfolger(aktKnoten));
270            }
271
272            // SAZ = SEZ     Dauer.
273            aktKnoten.setSaz(aktKnoten.getSez() - aktKnoten.getDauer());
274
275            for (Knoten vorgaenger : aktKnoten.getVorgaenger()) {
276                /*
277                 * Der SAZ eines Vorgangs wird SEZ aller unmittelbarer Vorga nger
278                 *
279                 * Haben mehrere Vorga nge einen gemeinsamen Vorga nger, so ist dessen SEZ der
280                 * fru heste (kleinste) SAZ aller Nachfolger.
281                 */
282
283                vorgaenger.setSez(this.getMinSazOfNachfolger(vorgaenger));
284                // Rufe setSazAndSez rekursiv fpr alle vorgänger vom aktuellen Knoten auf
285                setSazAndSez(vorgaenger);
286            }
287        }
288
289        /**
290         * Berechnet den Maximalen FEZ aller Vorgänger eines Knotens
291         *
292         * @param aktKnoten
293         *            aktuell betrachteter Knoten
294         * @return maximalen FEZ aller Vorgänger des Knoten
295         */
296        private int getMaxFezOfVorgaenger(Knoten aktKnoten) {
297            int max = Integer.MIN_VALUE;
298            for (Knoten vorgaenger : aktKnoten.getVorgaenger()) {
299                if (vorgaenger.getFez() > max) {
300                    max = vorgaenger.getFez();
301                }
302            }
```

```
303             return max;
304         }
305 
306         /**
307          * Berechnet den minimalen SAZ der Nachfolgenden Knoten eines betrachteten
308          * Knoten
309          *
310          * @param aktKnoten
311          *            aktuell betrachteter Knoten
312          * @return minimaler SAZ der Nachfolgenden Knoten eines betrachteten Knoten
313          */
314         private int getMinSazOfNachfolger(Knoten aktKnoten) {
315             int min = Integer.MAX_VALUE;
316             for (Knoten nachfolger : aktKnoten.getNachfolger()) {
317                 if (nachfolger.getSaz() < min) {
318                     min = nachfolger.getSaz();
319                 }
320             }
321             return min;
322         }
323 
324         /**
325          * Berechnet den GP aller Knoten ausgehend vom aktuell betrachteten Knoten
326          *
327          * @param aktKnoten
328          *            aktuell betrachteter Knoten
329          */
330         private void setGp(Knoten aktKnoten) {
331             /*
332              * Berechnung des Gesamtpuffers fu r jeden Knoten: GP = SAZ     FAZ = SEZ     FEZ
333              */
334             aktKnoten.setGp(aktKnoten.getSaz() - aktKnoten.getFaz());
335             for (Knoten nachfolger : aktKnoten.getNachfolger()) {
336                 setGp(nachfolger);
337             }
338         }
339 
340         /**
341          * Berechnet den FP aller Knoten ausgehend vom aktuell betrachteten Knoten
342          *
343          * @param aktKnoten
344          *            aktuell betrachteter Knoten
345          */
346         private void setFp(Knoten aktKnoten) {
347             /*
348              * Fu r die Berechnung des freien Puffers gilt: FP= (kleinster FAZ der
349              * nachfolgenden Knoten) - FEZ Ist der aktuelle Knoten der Endknoten, so ist der
350              * Freie Puffer 0, da FAZ==FEZ
351              */
352             aktKnoten.setFp(this.getMinFazOfNachfolger(aktKnoten) - aktKnoten.getFez());
353             for (Knoten nachfolger : aktKnoten.getNachfolger()) {
354                 setFp(nachfolger);
355             }
356         }
357 
358         /**
359          * Berechnet den kleinsten FAZ aller Nachfolger eines betrachteten Knoten
360          *
361          * @param aktKnoten
362          *            aktuell betrachteter Knoten
363          * @return kleinste FAZ aller Nachfolger eines betrachteten Knoten
364          */
365         private int getMinFazOfNachfolger(Knoten aktKnoten) {
366             int min = Integer.MAX_VALUE;
367             if (aktKnoten.getNachfolger().size() == 0) {
368                 return aktKnoten.getFez();
369             }
370             for (Knoten nachfolger : aktKnoten.getNachfolger()) {
371                 if (nachfolger.getFaz() < min) {
372                     min = nachfolger.getFaz();
373                 }
374             }
375             return min;
376         }
377 
378         /**
379          * Berechnet die Kritischen Pfade eines Netzplans und setzt sie im Model als
380          * kritischePfade
381          */
382         private void setKritischePfade() {
383             this.model.setKritischePfade(new ArrayList<>());
384             /*
385              * Bestimmung der kritischen Vorga nge ausgehend von jedem Startknoten
386              */
387             for (Knoten startK : this.model.getStartknoten()) {
388                 ArrayList<Knoten> pfad = new ArrayList<>();
389                 setKritischePfadeHelper(pfad, startK);
390             }
391         }
392 
393         /**
394          * Rekursive Hilfsmethode zur Berechnung der Kritischen Pfade nach dem Prinzip
395          * des Backtracking. Fügt bei erreichen des Endknotens den berechneten Pfad zum
396          * kritischePfade-Array im Model hinzu
397          *
398          * @param pfad
399          *            aktuell berechneter Pfad
400          * @param aktKnoten
401          *            aktuell betrachteter Knoten
402          */
403         private void setKritischePfadeHelper(ArrayList<Knoten> pfad, Knoten aktKnoten) {
404             /*
405              * Abbruchkriterium:Endknoten ist erreicht
```

```
406             */
407             if (aktKnoten.getNachfolger().size() == 0) {
408                 // Füge aktuellen Knoten in pfad ein
409                 pfad.add(aktKnoten);
410                 // Erstell Kopie des kritischen Pfades
411                 @SuppressWarnings("unchecked")
412                 ArrayList<Knoten> pfadKopie = (ArrayList<Knoten>) pfad.clone();
413                 // Füge errechneten Kritischen Pfad zu den im Model gekapselten Kritischen
414                 // Pfaden hinzu
415                 model.getKritischePfade().add(pfadKopie);
416                 // Breche die Mathode ab
417                 return;
418             }
419             /*
420              * Bestimmung der kritischen Vorga nge, d.h. GP = 0 und FP = 0
421              */
422             if (aktKnoten.getGp() == 0 && aktKnoten.getFp() == 0) {
423                 // füge aktuellen Knoten zum kritischen Pfad hinzu
424                 // pfad.add(aktKnoten);
425                 @SuppressWarnings("unchecked")
426                 ArrayList<Knoten> pfadKopie = (ArrayList<Knoten>) pfad.clone();
427                 pfadKopie.add(aktKnoten);
428                 // Führe für alle Nachfolger rekursiv die Methode setKritischePfadehelper aus
429                 // und durchlaufe so nach Backtraking den virtuellen Baum
430                 for (Knoten nachfolger : aktKnoten.getNachfolger()) {
431                     this.setKritischePfadeHelper(pfadKopie, nachfolger);
432                 }
433             }
434         }
435 
436         /**
437          * Prüft, ob alle Referenzen in model.knoten korrekt sind, also ob jeder
438          * Nachfolger auch in dessen Vorgaengern enthalten ist bzw. umgekehrt.
439          *
440          * Darf erst nach der Prüfung der Zyklen aufgerufen werden!
441          *
442          * @return true, falls alle Referenzen korrekt sind, sonst false.
443          */
444         boolean hatGueltigeReferenzen() {
445             for (Knoten k1 : this.model.getKnoten()) {
446                 for (Knoten nachfolger : k1.getNachfolger()) {
447                     if (!nachfolger.getVorgaenger().contains(k1)) {
448                         return false;
449                     }
450                 }
451             }
452 
453             for (Knoten k1 : this.model.getKnoten()) {
454                 for (Knoten vorgaenger : k1.getVorgaenger()) {
455                     if (!vorgaenger.getNachfolger().contains(k1)) {
456                         return false;
457                     }
458                 }
459             }
460 
461             return true;
462         }
463     }
```

9.3.2 Unittest Klasse Controller

```java
package controller;

import static org.junit.Assert.assertEquals;

import java.util.ArrayList;

import org.junit.Test;

import model.Knoten;
import model.Model;

/**
 * Unittest der Klasse Controllers
 *
 * @author M. Leonard Haufs Prüflingsnummer: 101-20540
 *
 */
public class ControllerTest {
    @Test
    public void hatKeineZyklen_ModelOhneZyklen_RueckgabeTrue() {
        // Arrangieren
        ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
        ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
        ersterKnotenNachfolger.add(2);
        Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, new ArrayList<>(),
                ersterKnotenNachfolger);
        knotenliste.add(ersterKnoten);
        //
        ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
        zweiterKnotenVorgaenger.add(1);
        Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger, new
                ArrayList<>());
        knotenliste.add(zweiterKnoten);
        Model model = new Model(knotenliste, "Testliste");
        //
        Controller controller = new Controller(model);

        // Ausführen
        boolean keineZyklen = controller.hatKeineZyklen();

        // Auswerten
        assertEquals(true, keineZyklen);
    }

    @Test
    public void hatKeineZyklen_ZweiterKnotenHatErstenKnotenAlsNachfolger_RueckgabeFalse() {
        // Arrangieren
        ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
        ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
        ersterKnotenNachfolger.add(2);
        Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, new ArrayList<>(),
                ersterKnotenNachfolger);
        knotenliste.add(ersterKnoten);
        //
        ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
        zweiterKnotenVorgaenger.add(1);
        ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
        zweiterKnotenNachfolger.add(1);
        Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                zweiterKnotenNachfolger);
        knotenliste.add(zweiterKnoten);
        Model model = new Model(knotenliste, "Testliste");
        //
        Controller controller = new Controller(model);

        // Ausführen
        boolean keineZyklen = controller.hatKeineZyklen();

        // Auswerten
        assertEquals(false, keineZyklen);
    }

    @Test
    public void
            hatKeineZyklen_ErsterKnotenHatZweitenKnotenAlsVorgaengerSowieNachfolgerUndZweiterKnotenHatErstenKnotenAlsVor
            {
        // Arrangieren
        ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
        //
        ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
        ersterKnotenVorgaenger.add(2);
        ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
        ersterKnotenNachfolger.add(2);
        Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
                ersterKnotenNachfolger);
        knotenliste.add(ersterKnoten);
        //
        ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
        zweiterKnotenVorgaenger.add(1);
        ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
        Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                zweiterKnotenNachfolger);
        knotenliste.add(zweiterKnoten);
        //
        Model model = new Model(knotenliste, "Testliste");
        //
        Controller controller = new Controller(model);

        // Ausführen
        boolean keineZyklen = controller.hatKeineZyklen();

```

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

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

```
279             zweiterKnotenVorgaenger.add(1);
280             ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
281             zweiterKnotenNachfolger.add(3);
282             Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
                    zweiterKnotenNachfolger);
283             knotenliste.add(zweiterKnoten);
284             //
285             ArrayList<Integer> dritterKnotenVorgaenger = new ArrayList<Integer>();
286             dritterKnotenVorgaenger.add(2);
287             ArrayList<Integer> dritterKnotenNachfolger = new ArrayList<Integer>();
288             Knoten dritterKnoten = new Knoten(3, "Dritter Schritt", 10, dritterKnotenVorgaenger,
                    dritterKnotenNachfolger);
289             knotenliste.add(dritterKnoten);
290             //
291             Model model = new Model(knotenliste, "Testliste");
292             //
293             Controller controller = new Controller(model);
294
295             // Ausführen
296             boolean gueltig = controller.hatGueltigeReferenzen();
297
298             // Auswerten
299             assertEquals(true, gueltig);
300         }
301     }
```

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         /**
108          * Bestimmt die Startknoten einer Liste von Knoten
109          *
110          * @param knoten
111          *             Liste von Knoten
112          * @return Startknoten einer Liste von Knoten
113          */
114         private ArrayList<Knoten> getStartknoten(ArrayList<Knoten> knoten) {
115             ArrayList<Knoten> startknoten = new ArrayList<>();
116             for (Knoten k : knoten) {
117                 if (k.getVorgaengerNummern().size() == 0) {
118                     startknoten.add(k);
119                 }
120             }
121 
122             return startknoten;
123         }
124 
125         /**
126          * Bestimmt die Endknoten einer Liste von Knoten
127          *
128          * @param knoten
129          *             Liste von Knoten
130          * @return Endknoten einer Liste von Knoten
131          */
132         private ArrayList<Knoten> getEndknoten(ArrayList<Knoten> knoten) {
133             ArrayList<Knoten> endknoten = new ArrayList<>();
134             for (Knoten k : knoten) {
135                 if (k.getNachfolgerNummern().size() == 0) {
136                     endknoten.add(k);
137                 }
138             }
139 
140             return endknoten;
141         }
142 
143         /**
144          * Initialisiert eine liste von Knoten, setzt also die Vorgänger und Nachfolger
145          * der Knoten
146          *
147          * @param knoten
148          *             Liste von Knoten
149          */
150         private void initKnoten(ArrayList<Knoten> knoten) {
151             for (Knoten k : knoten) {
152                 for (int vorgaengerNr : k.getVorgaengerNummern()) {
153                     for (Knoten k2 : knoten) {
154                         if (k2.getVorgangsnummer() == vorgaengerNr) {
155                             k.getVorgaenger().add(k2);
156                         }
157                     }
158                 }
159 
160                 for (int nachfolgerNr : k.getNachfolgerNummern()) {
161                     for (Knoten k2 : knoten) {
162                         if (k2.getVorgangsnummer() == nachfolgerNr) {
163                             k.getNachfolger().add(k2);
164                         }
165                     }
166                 }
167             }
168         }
169 
170     }
```