Abschlussprüfung Sommer 2018

zum

Mathematisch-technischer Softwareentwickler/-in (IHK)

vor der IHK Aachen

Entwicklung eines Softwaresystems

Thema:

Netzplanerstellung

Prüfling:	Haufs, Martin Leonard
Prüflingsnummer:	101 20540
Bearbeitungszeitraum:	14.05.2018 – 18.05.2018
Ausbildungsbetrieb:	Werkzeugmaschinenlabor RWTH Aachen Steinbachstraße 19 52074 Aachen

I Inhaltsverzeichnis

I Inhaltsverzeichnis i

1 Eigenständigkeitserklärung 1

2 Benutzeranleitung 2

2.1 Systemvoraussetzungen und Hinweise zum Aufruf 2

2.2 Installation des Programms 2

2.3 Programmstart 2

2.4 Externe Programme 3

3 Aufgabenanalyse 4

3.1 Allgemeine Problemstellung 4

3.2 Format der Eingabedatei 5

3.3 Format der Ausgabedatei 5

3.4 Algorithmus 6

3.5 Verbale Beschreibung des Verfahrens 7

3.6 Einlesen der Eingabedatei 7

3.7 Überführung der Eingabedaten ins Datenmodel 7

3.8 Berechnung im Controller 7

4 Programmkonzeption 10

4.1 UML Klassendiagramm 10

4.2 Programmablauf im Sequenzdiagramm 11

4.3 Nassi-Shneiderman-Diagramme 11

4.3.1 Main 11

4.3.2 EinlesenAusDatei 12

4.3.3 Model - Erzeugung des Models 13

4.3.4 Ausgabe 14

4.3.5 Controllermethoden 15

5 Abweichung von der handschriftlichen Ausarbeitung 18

5.1 Datenmodell 18

5.1.1 Die Sichtbarkeiten der Methoden 18

5.1.2 Klasse `Model` 18

5.1.3 Klasse `Knoten` 18

5.1.4 Klasse `Controller` 18
5.1.5 Klasse `LeseAusDatei` (Ursprünglich `InputFromFile`) 19
5.1.6 Abstrakte Klasse `Ausgabe` (ursprünglich `Output`) 19
5.1.7 Klasse `AusgabeInDatei` (ursprünglich `OutputToFile`) 20
6 Unittests 21
6.1 Prüfung der Methode `hatKeineZyklen()` 21
6.2 Prüfung der Methode `isZusammenhaengend()` 22
6.3 Prüfung der Methode `hatGueltigeReferenzen()` 22
7 Blackbox- Testfälle 23
7.1 Besonderheiten der Beispiele 2, 3 und 5 der durch die IHK verbesserten Aufgabenstellung 23
7.2 Normalfälle 24
7.2.1 Beispiele aus der durch die IHK verbesserten Aufgabenstellung 24
7.2.2 Eigene Normalfälle 25
7.3 Sonderfälle 29
7.3.1 Eigene Sonderfälle 29
7.4 Fehlerfälle 30
7.4.1 Beispiele der IHK 30
7.4.2 Eigene Fehlerfälle 32
8 Zusammenfassung und Ausblick 37
8.1 Ausblick 37
9 Anhang: Programmcode 38

1 Eigenständigkeitserklärung

Ich erkläre verbindlich, dass das vorliegende Prüfprodukt von mir selbstständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. Ich versichere, dass der vorgelegte Ausdruck mit dem Inhalt der von mir erstellten digitalen Version identisch ist. Weder ganz noch in Teilen wurde die Arbeit bereits als Prüfungsleistung vorgelegt. Mir ist bewusst, dass jedes Zuwiderhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung des Prüfungsprodukts als Prüfungsleistung ausschließt.

Aachen, der 18.05.2018 Martin Leonard Haufs

2 Benutzeranleitung

2.1 Systemvoraussetzungen und Hinweise zum Aufruf

Das Programm wurde unter MacOS 10 (64-Bit) in der Programmiersprache Java geschrieben und getestet. Das Programm sollte daher plattformunabhängig laufen, jedoch wird aufgrund der betriebssystemspezifischen Testumgebung und der beiliegenden Shellscript-Dateien eine Benutzung unter einem Unix-System (bestenfalls MacOS) empfohlen. Benötigt wird außerdem eine Java Runtime Environment, die mindestens in der Version 1.8 sein muss.

2.2 Installation des Programms

Für die Installation des Programms werden Ausführungsrechte für die Datei „PATH/Netzplanerstellung.jar“ und ggf. auch für alle Dateien im Ordner „Shellscripte“ benötigt und zudem ein Lese- und Schreibrecht für alle Dateien im Verzeichnis „Testfaelle“ und all seinen Unterverzeichnissen. Die Shellscripte .sh im Ordner Shellscripte müssen ausführbar sein.

Dies lässt sich mittels des Konsolenbefehls chmod +x SHELLSCRIPTNAME.sh realisieren.

2.3 Programmstart

Es gibt grundsätzlich zwei Möglichkeiten das Programm aufzurufen.

Die erste Methode geht folgendermaßen und bezieht sich auf die Benutzung von Unix-Betriebssystemen:

1. Eingabedaten in folgende Verzeichnisse einfügen:
   - „Testfaelle/Normalfaelle“
   - „Testfaelle/Sonderfaelle“
   - „Testfaelle/Fehlerfaelle“
2. Entsprechende Shellscript-Datei im Verzeichnis „Shellscripte “ ausführen.

Anzumerken ist, dass im Verzeichnis „Testfaelle“ noch weitere Verzeichnisse existieren, in denen schon fertige Testfälle vorhanden sind. Auf diese Fälle wird im 7. Kapitel Bezug genommen.

Eine weitere Möglichkeit besteht darin, die betriebsspezifische Konsole zu verwenden. Der Aufruf erfolgt dabei über:

```
$ java –jar
ABLAGEVERZEICHNIS/GrosseProg_101201540/Netzplanerstellung.jar ENDUNG VERZEICHNIS
```

Dabei steht der Platzhalter ENDUNG für die Endung der Dateien, die im Verzeichnis VERZEICHNIS durch das Programm eingelesen und verarbeitet werden.
ABLAGEVERZEICHNIS ist der Ordner, in dem der Ordner *GrosseProg_101201540* liegt.

2.4 Externe Programme

Für die Nassi Shneidermann- Diagramme wurde das Programm *Structorizer*[1] verwendet.

Das Klassendiagramm wurde aus dem Progammcode mittels des Eclipse- Plugins *ObjectAid UML Explorer*[2] *erzeugt.*

Für die Erstellung des Sequenzdiagramms und der anderen Abbildungen dieses Dokumentes wurde das Programm Umlet[3] verwendet.

Die im Hauptverzeichnis *GrosseProg_101201540* unter dem Ordner *Diagramme* gespeicherten Dateien lassen sich mit diesen Programmen öffnen.

[1] http://structorizer.fisch.lu

[2] http://www.objectaid.com

[3] http://www.umlet.com

3 Aufgabenanalyse

3.1 Allgemeine Problemstellung

Zu erstellen war ein Programm zur Generierung und Analyse eines Netzplans.

Ein Netzplan ist eine Verkettung von Knotenpunkten mit definierten Eigenschaften, die sich zum Teil aus ihren Nachfolgern und Vorgängern berechnen lassen. Jeder Knoten hat dabei folgende Eigenschaften: Die Dauer (D) eines Vorgangs, den frühesten Anfangszeitpunkt (FAZ), den frühesten Endzeitpunkt (FEZ), den spätesten Anfangszeitunkt (SAZ), den spätesten Endzeitpunkt (SEZ), den Gesamtpuffer (GP) und den freien Puffer (FP).

Jeder Knoten hat, mit Ausnahme des Startknotens, mindestens einen Vorgänger und, mit Ausnahme des Endknotens, mindestens einen Nachfolger.

Zyklen innerhalb des Netzplans sind nicht erlaubt und sollen bei der Prüfung der Daten zu einem Abbruch führen. Ebenso soll auf Zusammengehörigkeit der Knoten geprüft werden. Das bedeutet, dass alle Knoten direkt oder indirekt miteinander verbunden sind.

Der FAZ des Startknotens ist 0. Für den FEZ eines Knotens gilt: FEZ = FAZ + D. FEZ eines Vorgängers ist der FAZ aller nachfolgenden Knoten, wobei bei mehreren Vorgängern der mit dem größten FEZ gewählt wird. Für den Endknoten gilt, dass der FEZ dem SEZ entspricht (SEZ=FEZ). SAZ eines Knotens ist wie folgt definiert: SAZ = SEZ – Dauer. Der SAZ eines Knotens ist der SEZ des Vorgängers. Haben mehrere Knoten einen gemeinsamen Vorgänger, ist der SEZ dieses Knotens der kleinste SAZ aller Nachfolger. Der Gesamtpuffer eines Knotens ist wie folgt definiert: GP = SAZ - FAZ (also auch GP = SEZ - FEZ). Der freie Puffer eines Knotens ist (kleinster FAZ der Nachfolgeknoten) - FEZ.

Die Daten der unter Kapitel 3.2 beschriebenen Eingabedatei sollen eingelesen werden und auf ihre Korrektheit hin überprüft werden. Existieren mindestens ein Start- und ein Endpunkt, so sollen, nach Prüfung auf Zusammenhang der Knoten und Ausschluss von Zyklen, alle Kenngrößen und die möglichen kritischen Pfade berechnet werden.

Kritische Pfade sind die Reihenfolge des Netzplans, ausgehend von einem Startknoten und endend in einem Endknoten, bei dem alle durchlaufenen Knoten keine Zeitreserven haben, also GP = 0 und FP = 0. Es kann mehrere Kritische Pfade geben. Ist dies der Fall, so sollen alle kritischen Pfade bestimmt werden.

3.2 Format der Eingabedatei

Die Eingabe der Strategie erfolgt über eine Datei, die wie folgt strukturiert ist:

```
// beliebige Anzahl Kommentarzeilen, eingeleitet mit „//“
//+ Überschrift
// beliebige Anzahl Kommentarzeilen, eingeleitet mit „//“
Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
…
```

Kommentare innerhalb der Eingabe sind Zeilen, die mit einem „//“ beginnen. Es gibt genau einen solchen Kommentar, der die für das Programm relevante Überschrift beinhaltet. Dieser Kommentar beginnt mit „//+ “. Sonstige Kommentare werden ignoriert.

Jede nicht-Kommentarzeile besteht aus folgender Struktur: Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger (getrennt durch Komma); Nachfolger (getrennt durch Komma). Nach jedem Semikolon folgt zusätzlich ein Leerzeichen. Existiert kein Vorgänger bzw. Nachfolger, so wird stattdessen ein Minuszeichen eingefügt. Alle Zahlen müssen ganzzählige Werte annehmen.

3.3 Format der Ausgabedatei

Die Ausgabe erfolgt in eine Datei und soll folgende Struktur haben, wenn das Programm fehlerfrei mit gültigen Daten gestartet wird:

```
Überschrift

Vorgangsnummer; Vorgangsbezeichnung; D; FAZ; FEZ; SAZ; SEZ; GP; FP
Vorgangsnummer; Vorgangsbezeichnung; D; FAZ; FEZ; SAZ; SEZ; GP; FP
Vorgangsnummer; Vorgangsbezeichnung; D; FAZ; FEZ; SAZ; SEZ; GP; FP
...

Anfangsvorgang: Startknoten
Endvorgang: Endknoten
Gesamtdauer: Gesamtdauer des Kritischen Pfades
Kritischer Pfad
Vorgangsnummer->Vorgangsnummer->Vorgangsnummer->...
Vorgangsnummer->Vorgangsnummer->Vorgangsnummer->...
...
```

Zunächst wird der in der Eingabe mit „//+ “ gekennzeichneten Überschrift ausgegeben, wobei auf das einleitende „//+ “ verzichtet wird. Nach einem Absatz folgt eine Beschreibende Zeile „Vorgangsnummer; Vorgangsbezeichnung; D; FAZ; FEZ; SAZ; SEZ; GP; FP“.

Darauffolgende Zeilen haben immer die gleiche Struktur: Es wird also zeilenweise für jeden Knoten zunächst die Vorgangsnummer, dann die Vorgangsbezeichnung, dann die Dauer, dann die FAZ, dann die FEZ, dann die SAZ, dann die SEZ, dann der GP und anschließend der FP angegeben. Getrennt werden diese Werte mit Semikolon und einem Leerzeichen. Nach jedem Knoten folgt ein Absatz.

Nachdem alle Knoten ausgegeben wurden, folgt ein Absatz. Es wird „Anfangsvorgang: “ gefolgt von einer durch Komma getrennten Auflistung der Startpunkte. Es folgt ein Absatz.

Es wird „Endvorgang: “ gefolgt von einer durch Komma getrennten Auflistung der Endpunkte. Es folgt ein Absatz. Es wird „Gesamtdauer: “ gefolgt von der Gesamtdauer des kritischen Pfades. Gibt es mehrere kritische Pfade, wird „*Nicht eindeutig*“ angegeben. Nach einem Absatz folgt „Kritischer Pfad“ bzw. bei mehreren Kritischen Pfaden „Kritische Pfade“. Nach einem Absatz wird jeder kritische Pfad durch eine Auflistung der Vorgangsnummern, getrennt durch „->“, angegeben.

3.4 Algorithmus

Der eigentliche Hauptalgorithmus des Controllers besteht aus drei Teilen.

Zunächst wird überprüft, ob der Netzplan (im folgenden Graph genannt) aus zusammenhängenden Knoten besteht und ob er keine Zyklen hat. Dies wird mittels Backtracking überprüft, wo jeweils ein virtueller Graph (Baum) ausgehend von allen Startpunkten durchlaufen wird.

Im Falle der Prüfung auf Zusammenhängigkeit der Knoten wird jeder Knoten durchlaufen und die einzelnen Knoten in einer Validation-Liste gesammelt, falls diese noch nicht enthalten sind. Falls nach Durchlauf des gesamten Graphen alle Knoten des Graphen in der Validation-Liste enthalten sind, ist der Graph zusammenhängend.

Im Falle der Prüfung auf Zykelfreiheit wird ähnlich verfahren. Alle Knoten des Graphen werden durchlaufen. Erreicht die Funktion zum zweiten Mal einen Knoten (hier ebenfalls durch eine Validation-Liste geregelt), so wird ein Zykel festgestellt. Falls jeder Knoten nur einmal durchlaufen wird, so wird die Zykelfreiheit festgestellt.

Beide Methoden verlaufen nach dem Prinzip des Backtrackings, bei dem der Graph bis zu den Blättern durchlaufen wird und im Falle des Erreichen eines Abbruchkriteriums am Blatt das Ergebnis in einem externen Korb gespeichert wird.

In der zweiten Hauptfunktion des Controllers wird das Model initialisiert nach drei Schritten:

1) Vorwärtsrechnung:
   Bei gegebenem Anfangstermin werden aufgrund der angegebenen Dauer eines Vorganges die frühestmöglichen Anfangs- und Endzeiten eingetragen. Weiterhin lässt sich die Gesamtdauer eines Projekts bestimmen. Dabei wird der Baum von allen Startknoten aus vorwärts durchlaufen:
   Der Startknoten hat als FAZ immer den Wert 0. Für den FEZ gilt: FEZ = FAZ + Dauer. Der FEZ eines Vorgängers ist FAZ aller unmittelbar nachfolgenden Knoten. Münden mehrere Knoten in einen Vorgang, dann ist der FAZ der größte FEZ der unmittelbaren Vorgänger.
2) Rückwärtsrechnung:
   Bei der Rückwärtsrechnung wird ermittelt, wann die einzelnen Vorgänge spätestens begonnen und fertiggestellt sein müssen, damit die Gesamtprojektzeit nicht gefährdet ist. Dazu wird der Graph von allen Endpunkten aus durchlaufen.
   Für die Startpunkte ist der früheste Endzeitpunkt (FEZ) auch der späteste Endzeitpunkt (SEZ), also SEZ = FEZ. Für den spätesten Anfangszeitpunkt gilt: SAZ = SEZ – Dauer. Der SAZ eines Vorgangs wird SEZ aller unmittelbarer Vorgänger.

Haben mehrere Vorgänge einen gemeinsamen Vorgänger, so ist dessen SEZ der kleinste SAZ aller Nachfolger.

3) Ermittlung der Zeitreserven und der kritischen Pfade:
   Für alle Knoten wird der Gesamtpuffer (GP) sowie der freie Puffer (FP) berechnet.
   GP = SAZ – FAZ = SEZ – FEZ und FP= (kleinster FAZ der nachfolgenden Knoten) – FEZ
   Die Kritischen Pfade sind die Abfolgen von Knoten, bei der FP=0 und GP=0 sind.

3.5 Verbale Beschreibung des Verfahrens

3.6 Einlesen der Eingabedatei

Das Programm wird mit zwei Argumenten gestartet. Es enthält neben dem Verzeichnis, aus dem Eingabedateien eingelesen werden sollen, eine Dateiendung, die spezifiziert, welche Dateien aus diesem Verzeichnis gelesen werden. Falls das Verzeichnis nicht gefunden wird, ist kein gültiges Verzeichnis vorhanden oder existiert der Pfad nicht, wird das Programm abgebrochen und eine Fehlermeldung auf der Konsole ausgegeben. Bei einer fehlerfreien Überprüfung wird für jede der Dateien in diesem Verzeichnis überprüft, ob die Dateiendung der dem Programm übergebenen Endung entspricht. Falls dies nicht der Fall ist, wird die nächste Datei überprüft. Für jede Datei mit entsprechender Dateiendung wird zusätzlich die Lesbarkeit dieser Datei festgestellt. Kann die Datei nicht gelesen werden, wird anschließend die nächste Datei untersucht und ein entsprechender Fehler auf der Konsole ausgegeben. Es wird zudem geprüft, ob Vorgangsnummern mehrfach vorkommen, da dies ansonsten zu einem Fehler führen würde. Kommen Vorgangsnummern mehrfach vor, so wird ein entsprechender Fehler auf der Konsole ausgegeben.

3.7 Überführung der Eingabedaten ins Datenmodel

Zur Überführung der Daten werden zunächst pro eingelesenem Knoten die Kennwerte Vorgangsnummer, Vorgangsbezeichnung und die Nummern der Vorgänger und Nachfolger des jeweiligen Knoten bestimmt. Beim Überführen der Daten ins Model werden die Knoten anschließend initialisiert, also die Referenzen zwischen den Vorgängern und Nachfolgern erstellt. Die Start- und Endpunkte des Graphen werden im Model je in einer Liste gespeichert. Es wird überprüft, ob die Referenzen gültig sind, also zu jeder Vorgängerreferenz auch eine entsprechende Nachfolgerreferenz (und umgekehrt) existiert.

Die Knoten bilden also anschließend eine doppelt verkettete Liste von Knoten, die vorwärts von den Startpunkten aus, und rückwärts von den Endpunkten aus, durchlaufen werden kann.

3.8 Berechnung im Controller

Der eigentliche Hauptalgorithmus des Controllers besteht aus drei Teilen.

Zunächst wird überprüft, ob der Netzplan (im folgenden Graph genannt) aus zusammenhängenden Knoten besteht und ob er keine Zyklen hat. Dies wird mittels Backtracking überprüft, wo jeweils ein virtueller Graph (Baum) ausgehend von allen Startpunkten durchlaufen wird.

Im Falle der Prüfung, ob die Knoten miteinander direkt oder indirekt verbunden sind, wird jeder Knoten durchlaufen und die einzelnen Knoten in einer Validation-Liste gesammelt, falls diese noch nicht enthalten sind. Falls nach Durchlauf des gesamten Graphen alle Knoten des Graphen in der Validation-Liste enthalten sind, ist der Graph Zusammenhängend.

Im Falle der Prüfung auf Zyklusfreiheit wird ähnlich verfahren. Alle Knoten des Graphen werden durchlaufen. Erreicht die Funktion zum zweiten Mal einen Knoten (Hier ebenfalls durch eine Validation-Liste geregelt), so wird ein Zyklus festgestellt. Falls jeder Knoten nur einmal durchlaufen wird, so wird die Zyklusfreiheit festgestellt.

Beide Methoden verlaufen nach dem Prinzip des Backtrackings, bei dem der Graph bis zu den Blättern durchlaufen wird und im Falle des Erreichens eines Abbruchkriteriums am Blatt das Ergebnis in einem externen Korb gespeichert wird (hier im Model).

In der zweiten Hauptfunktion des Controllers wird das Model initialisiert nach drei Schritten:

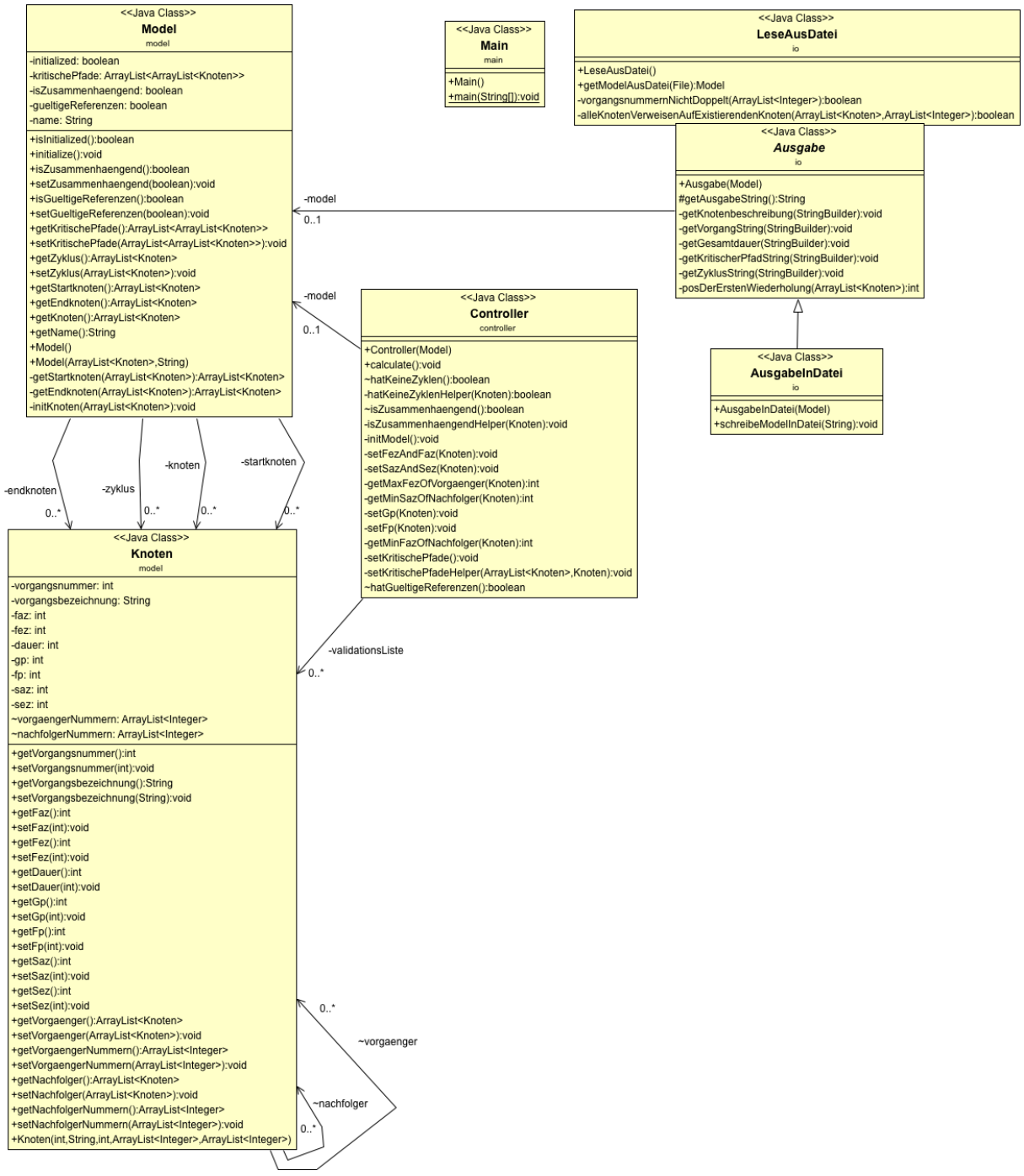
1) Vorwärtsrechnung:
   Bei gegebenem Anfangstermin werden aufgrund der angegebenen Dauer eines Vorganges die frühestmöglichen Anfangs- und Endzeiten eingetragen. Weiterhin lässt sich die Gesamtdauer eines Projekts bestimmen. Dabei wird der Baum von allen Startknoten aus vorwärts durchlaufen:
   Der Startknoten hat als FAZ immer den Wert 0. Für den FEZ gilt: FEZ = FAZ + Dauer. Der FEZ eines Vorgängers ist FAZ aller unmittelbar nachfolgenden Knoten. Münden mehrere Knoten in einen Vorgang, dann ist der FAZ der größte FEZ der unmittelbaren Vorgänger.
2) Rückwärtsrechnung:
   Bei der Rückwärtsrechnung wird ermittelt, wann die einzelnen Vorgänge spätestens begonnen und fertiggestellt sein müssen, damit die Gesamtprojektzeit nicht gefährdet ist. Dazu wird der Graph von allen Endpunkten aus durchlaufen.
   Für die Startpunkte ist der früheste Endzeitpunkt (FEZ) auch der späteste Endzeitpunkt (SEZ), also SEZ = FEZ. Für den spätesten Anfangszeitpunkt gilt: SAZ = SEZ – Dauer. Der SAZ eines Vorgangs wird SEZ aller unmittelbarer Vorgänger. Haben mehrere Vorgänge einen gemeinsamen Vorgänger, so ist dessen SEZ der kleinste SAZ aller Nachfolger.
3) Ermittlung der Zeitreserven:
   Für alle Knoten wird der Gesamtpuffer (GP) sowie der freie Puffer (FP) berechnet. GP = SAZ – FAZ = SEZ – FEZ und FP= (kleinster FAZ der nachfolgenden Knoten) – FEZ

Anschließend werden die kritischen Pfade berechnet, falls diese existieren. Dazu wird erneut Backtracking verwendet: Ausgehend von jedem Startknoten wird eine Hilfsmethode auf jeden Startknoten aufgerufen:

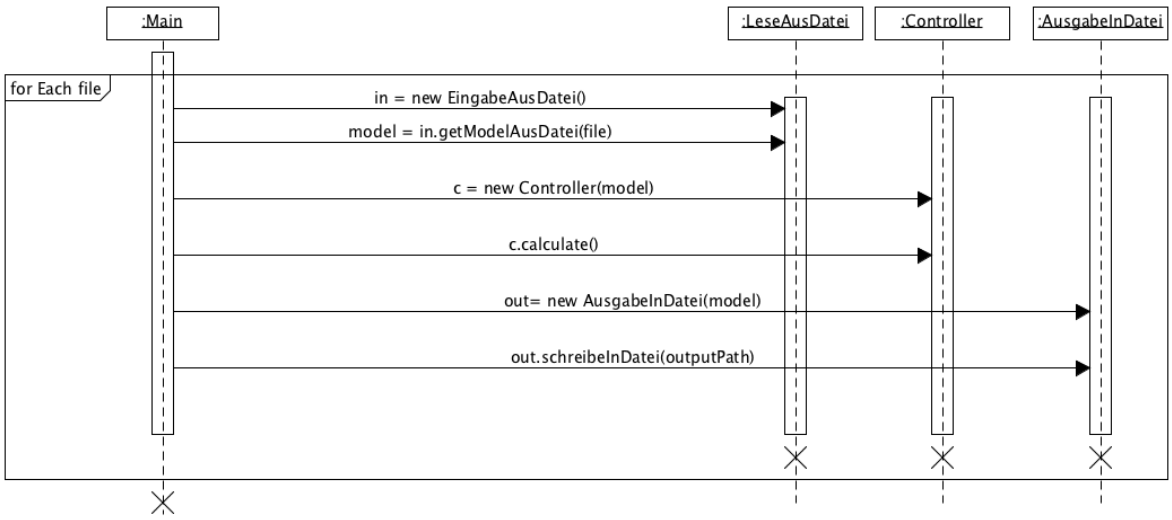
Diese prüft zunächst die Abbruchbedingung, dass der aktuell in der Hilfsmethode betrachtete Knoten ein Endpunkt ist. Ist dies der Fall, wird der berechnete Pfad im externen Model zu einer Liste hinzugefügt und die Methode beendet. Ansonsten wird geprüft, ob der aktuelle Knoten das Kriterium für einen Kritischen Pfad erfüllt (GP = 0 und FP = 0). Ist dies der Fall, so wird der aktuelle Knoten zum Pfadarray hinzugefügt und die Hilfsmethode auf jedem Nachfolger des aktuellen Knotens aufgerufen.

4 Programmkonzeption

4.1 UML Klassendiagramm

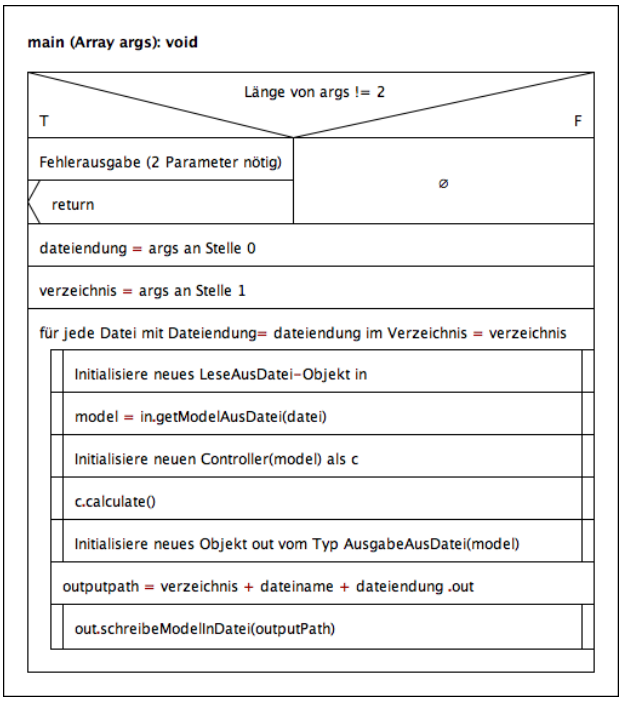


4.2 Programmablauf im Sequenzdiagramm

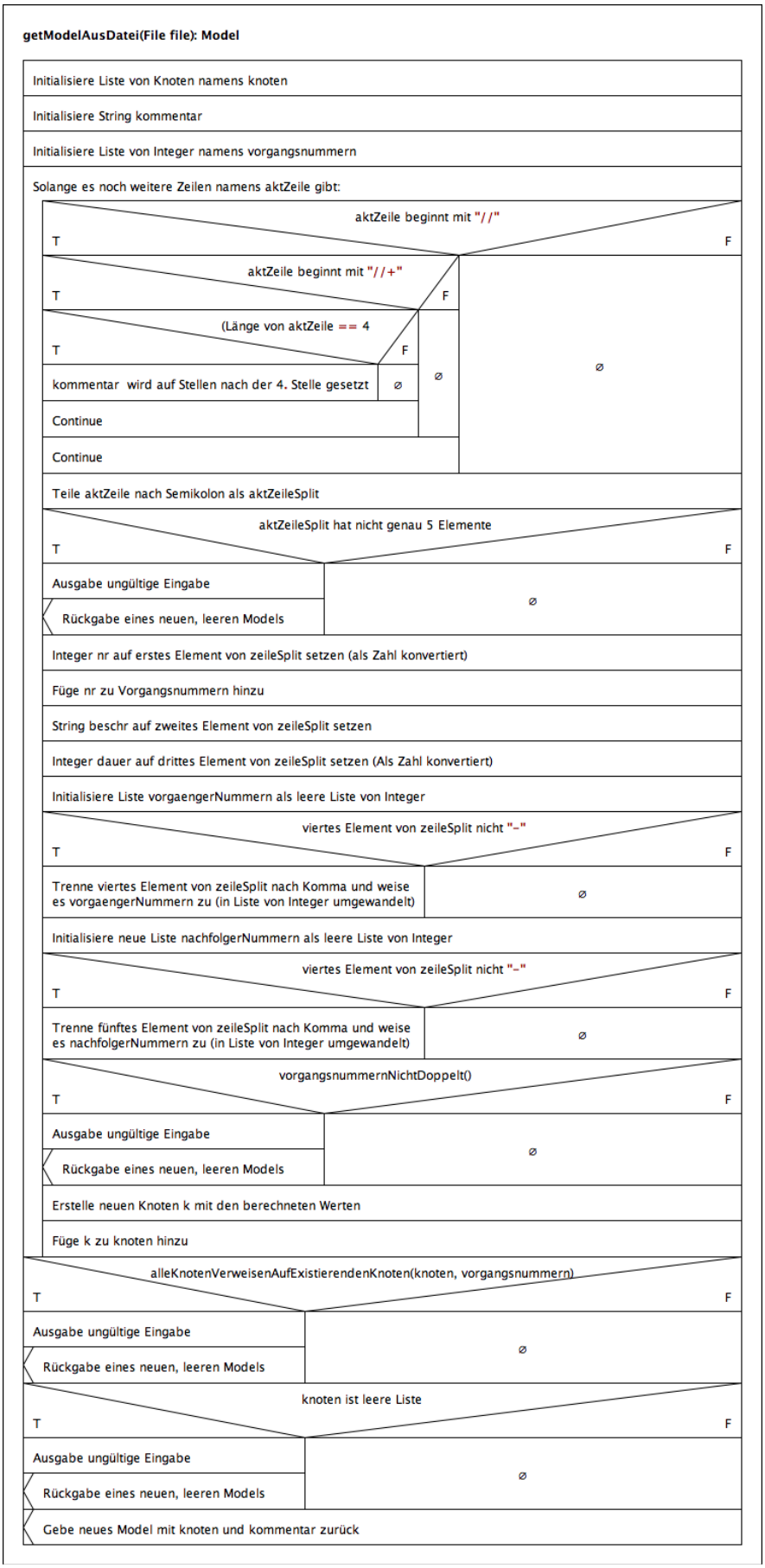


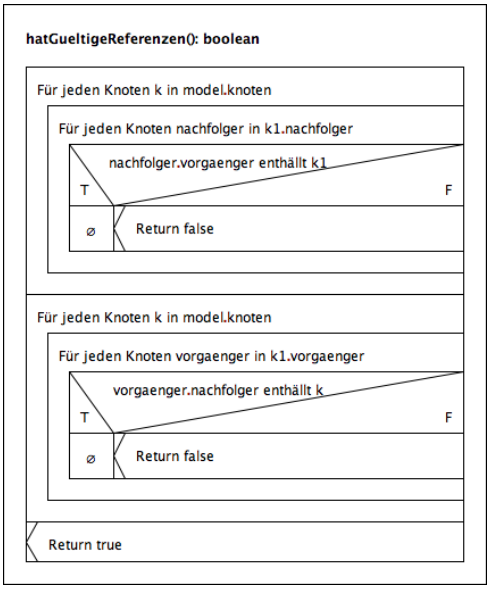
4.3 Nassi-Shneiderman-Diagramme

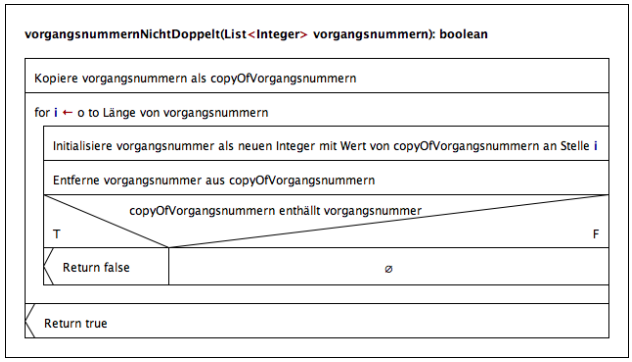
4.3.1 Main

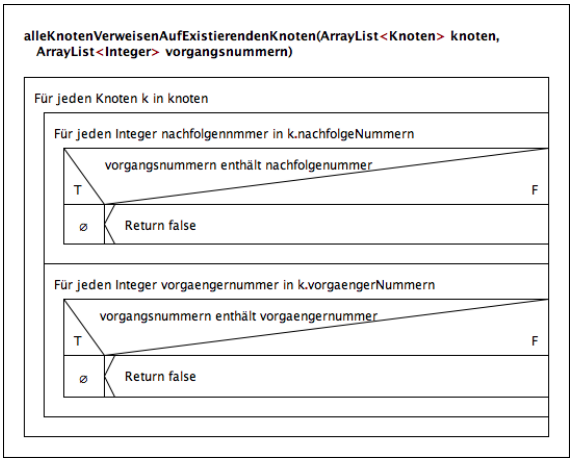


4.3.2 EinlesenAusDatei

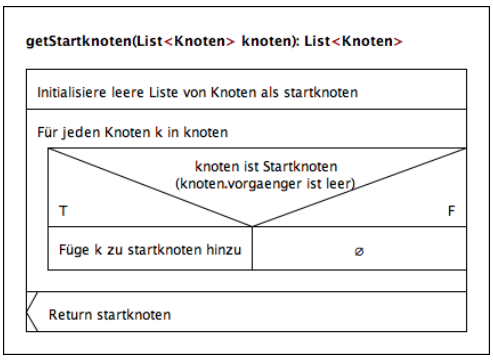


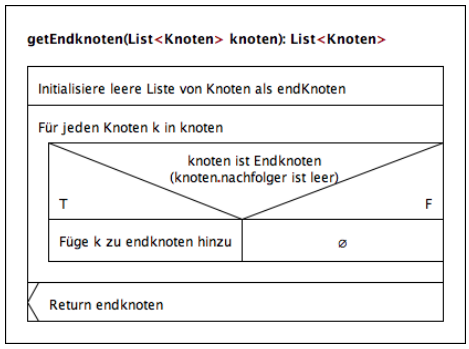


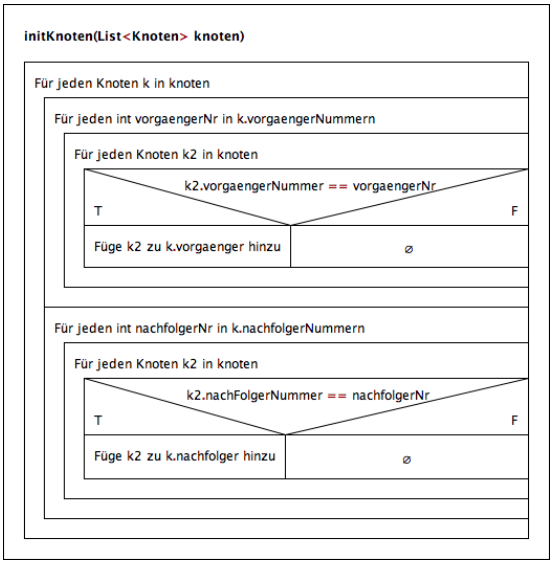




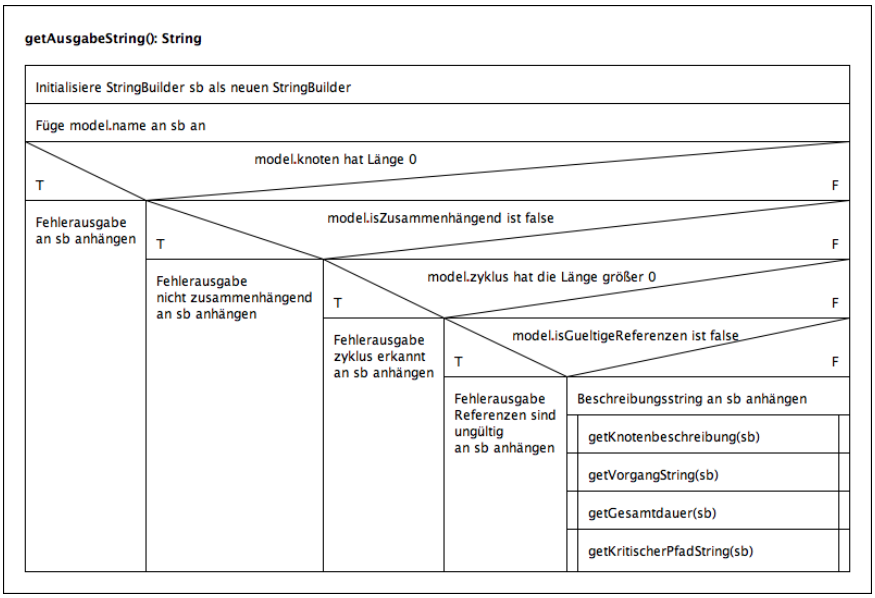
4.3.3 Model - Erzeugung des Models

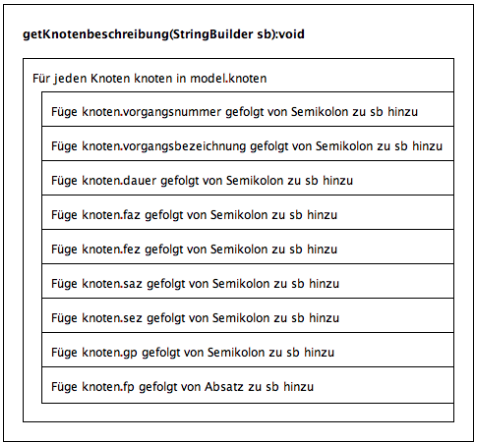


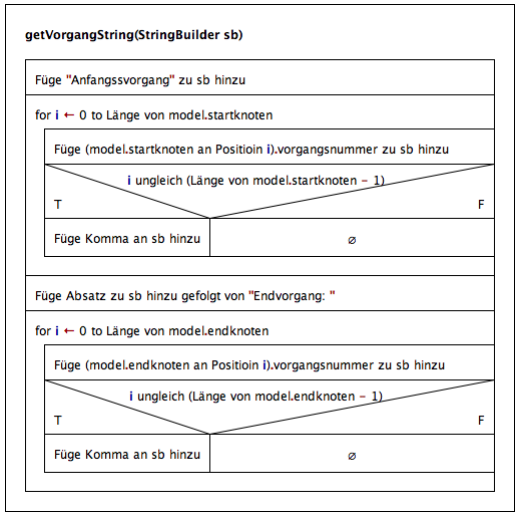


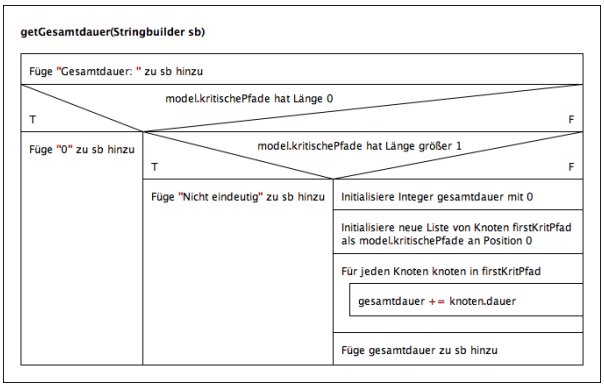

initKnoten(List<Knoten> knoten)
Für jeden Knoten k in knoten
Für jeden int vorgaengerNr in k.vorgaengerNummern
Für jeden Knoten k2 in knoten
k2.vorgaengerNummer == vorgaengerNr
T
F
Füge k2 zu k.vorgaenger hinzu
⌀
Für jeden int nachfolgerNr in k.nachfolgerNummern
Für jeden Knoten k2 in knoten
k2.nachFolgerNummer == nachfolgerNr
T
F
Füge k2 zu k.nachfolger hinzu
⌀

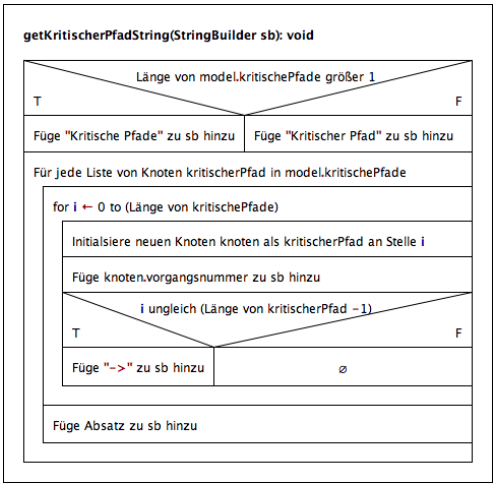
4.3.4 Ausgabe

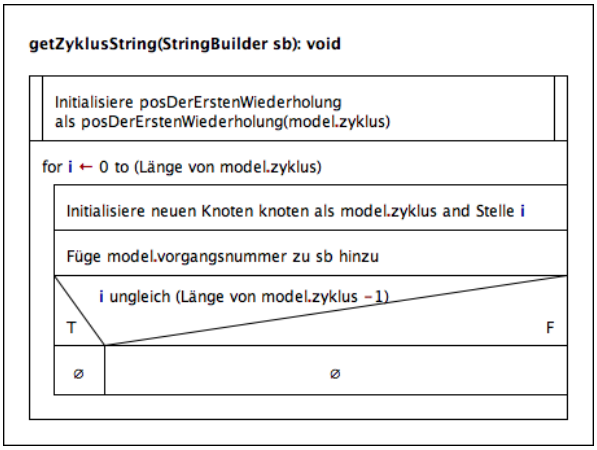

getAusgabeString(): String
Initialisiere StringBuilder sb als neuen StringBuilder
Füge model.name an sb an
model.knoten hat Länge 0
T
F
Fehlerausgabe an sb anhängen
model.isZusammenhängend ist false
T
F
Fehlerausgabe nicht zusammenhängend an sb anhängen
model.zyklus hat die Länge größer 0
T
F
Fehlerausgabe zyklus erkannt an sb anhängen
model.isGueltigeReferenzen ist false
T
F
Fehlerausgabe Referenzen sind ungültig an sb anhängen
Beschreibungsstring an sb anhängen
getKnotenbeschreibung(sb)
getVorgangString(sb)
getGesamtdauer(sb)
getKritischerPfadString(sb)

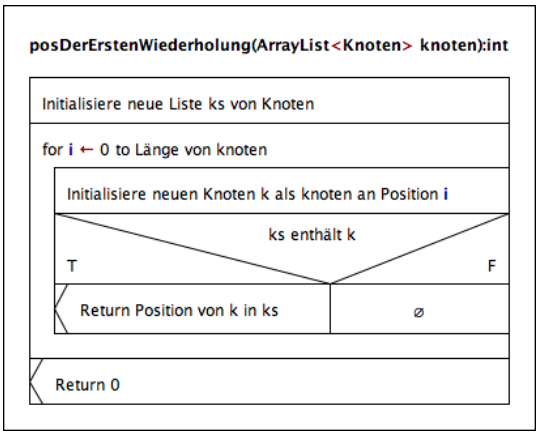

getKnotenbeschreibung(StringBuilder sb):void
Für jeden Knoten knoten in model.knoten
Füge knoten.vorgangsnummer gefolgt von Semikolon zu sb hinzu
Füge knoten.vorgangsbezeichnung gefolgt von Semikolon zu sb hinzu
Füge knoten.dauer gefolgt von Semikolon zu sb hinzu
Füge knoten.faz gefolgt von Semikolon zu sb hinzu
Füge knoten.fez gefolgt von Semikolon zu sb hinzu
Füge knoten.saz gefolgt von Semikolon zu sb hinzu
Füge knoten.sez gefolgt von Semikolon zu sb hinzu
Füge knoten.gp gefolgt von Semikolon zu sb hinzu
Füge knoten.fp gefolgt von Absatz zu sb hinzu


getVorgangString(StringBuilder sb)
Füge "Anfangssvorgang" zu sb hinzu
for i ← 0 to Länge von model.startknoten
Füge (model.startknoten an Positioin i).vorgangsnummer zu sb hinzu
i ungleich (Länge von model.startknoten – 1)
T
F
Füge Komma an sb hinzu
⌀
Füge Absatz zu sb hinzu gefolgt von "Endvorgang: "
for i ← 0 to Länge von model.endknoten
Füge (model.endknoten an Positioin i).vorgangsnummer zu sb hinzu
i ungleich (Länge von model.endknoten – 1)
T
F
Füge Komma an sb hinzu
⌀

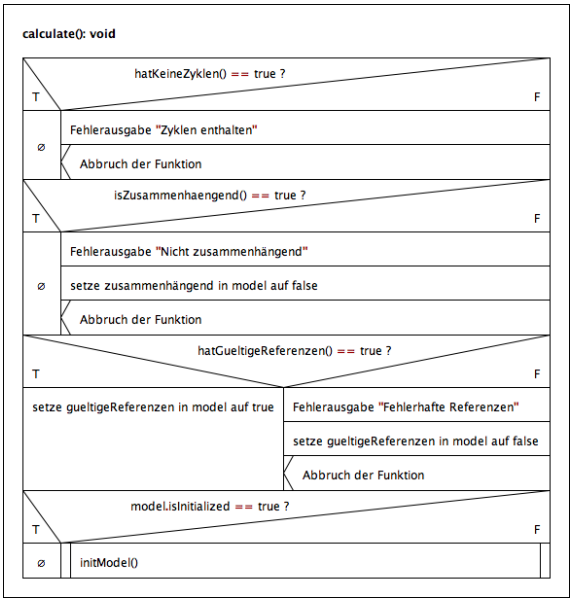


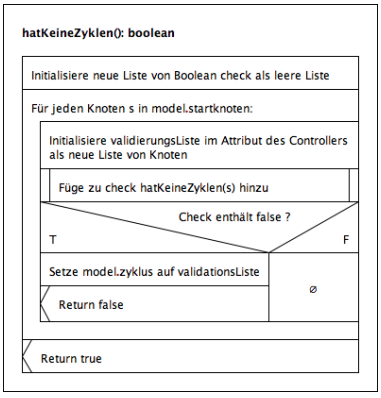


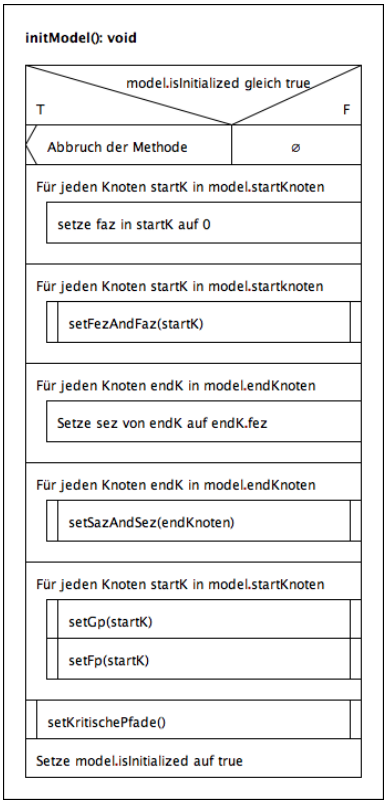


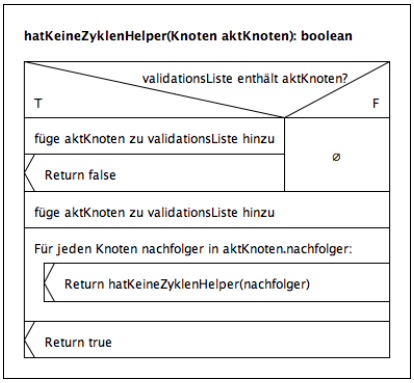


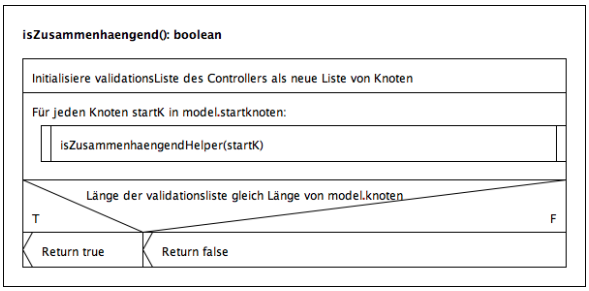
4.3.5 Controllermethoden

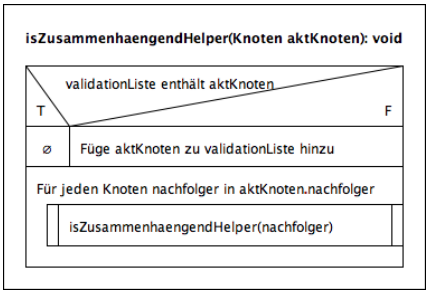


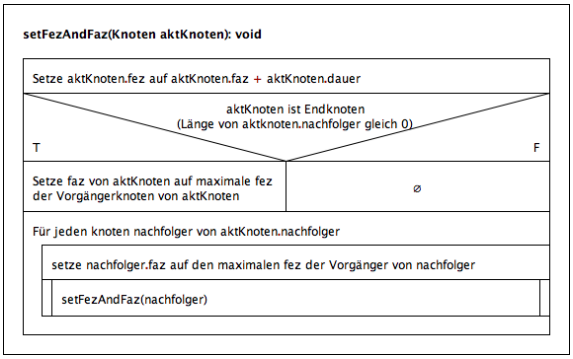


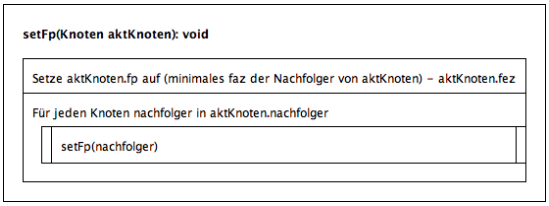

initModel(): void
model.isInitialized gleich true
T
F
Abbruch der Methode
ø
Für jeden Knoten startK in model.startKnoten
setze faz in startK auf 0
Für jeden Knoten startK in model.startknoten
setFezAndFaz(startK)
Für jeden Knoten endK in model.endKnoten
Setze sez von endK auf endK.fez
Für jeden Knoten endK in model.endKnoten
setSazAndSez(endKnoten)
Für jeden Knoten startK in model.startKnoten
setGp(startK)
setFp(startK)
setKritischePfade()
Setze model.isInitialized auf true

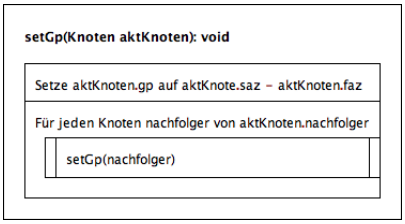

hatKeineZyklenHelper(Knoten aktKnoten): boolean
validationsListe enthält aktKnoten?
T
F
füge aktKnoten zu validationsListe hinzu
Return false
ø
füge aktKnoten zu validationsListe hinzu
Für jeden Knoten nachfolger in aktKnoten.nachfolger:
Return hatKeineZyklenHelper(nachfolger)
Return true

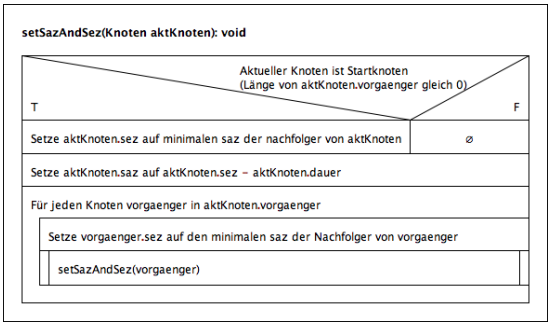

isZusammenhaengend(): boolean
Initialisiere validationsListe des Controllers als neue Liste von Knoten
Für jeden Knoten startK in model.startknoten:
isZusammenhaengendHelper(startK)
Länge der validationsliste gleich Länge von model.knoten
T
F
Return true
Return false

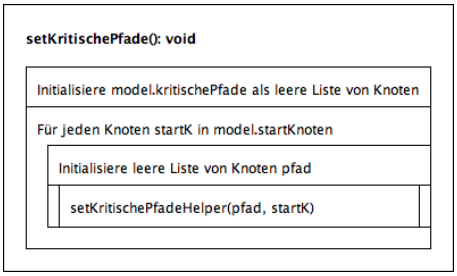

isZusammenhaengendHelper(Knoten aktKnoten): void
validationListe enthält aktKnoten
T
F
ø
Füge aktKnoten zu validationListe hinzu
Für jeden Knoten nachfolger in aktKnoten.nachfolger
isZusammenhaengendHelper(nachfolger)

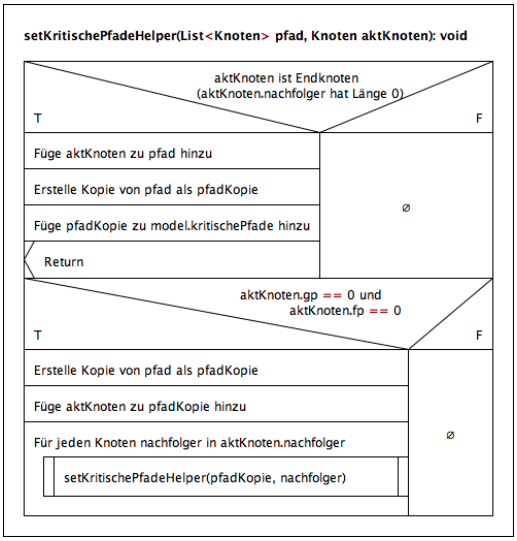

setFezAndFaz(Knoten aktKnoten): void
Setze aktKnoten.fez auf aktKnoten.faz + aktKnoten.dauer
aktKnoten ist Endknoten
(Länge von aktknoten.nachfolger gleich 0)
T
F
Setze faz von aktKnoten auf maximale fez der Vorgängerknoten von aktKnoten
ø
Für jeden knoten nachfolger von aktKnoten.nachfolger
setze nachfolger.faz auf den maximalen fez der Vorgänger von nachfolger
setFezAndFaz(nachfolger)


setFp(Knoten aktKnoten): void
Setze aktKnoten.fp auf (minimales faz der Nachfolger von aktKnoten) – aktKnoten.fez
Für jeden Knoten nachfolger in aktKnoten.nachfolger
setFp(nachfolger)


setGp(Knoten aktKnoten): void
Setze aktKnoten.gp auf aktKnote.saz – aktKnoten.faz
Für jeden Knoten nachfolger von aktKnoten.nachfolger
setGp(nachfolger)


setSazAndSez(Knoten aktKnoten): void
Aktueller Knoten ist Startknoten
(Länge von aktKnoten.vorgaenger gleich 0)
T
F
Setze aktKnoten.sez auf minimalen saz der nachfolger von aktKnoten
ø
Setze aktKnoten.saz auf aktKnoten.sez – aktKnoten.dauer
Für jeden Knoten vorgaenger in aktKnoten.vorgaenger
Setze vorgaenger.sez auf den minimalen saz der Nachfolger von vorgaenger
setSazAndSez(vorgaenger)


setKritischePfade(): void
Initialisiere model.kritischePfade als leere Liste von Knoten
Für jeden Knoten startK in model.startKnoten
Initialisiere leere Liste von Knoten pfad
setKritischePfadeHelper(pfad, startK)


setKritischePfadeHelper(List<Knoten> pfad, Knoten aktKnoten): void
aktKnoten ist Endknoten
(aktKnoten.nachfolger hat Länge 0)
T
F
Füge aktKnoten zu pfad hinzu
Erstelle Kopie von pfad als pfadKopie
Füge pfadKopie zu model.kritischePfade hinzu
Return
ø
aktKnoten.gp == 0 und
aktKnoten.fp == 0
T
F
Erstelle Kopie von pfad als pfadKopie
Füge aktKnoten zu pfadKopie hinzu
Für jeden Knoten nachfolger in aktKnoten.nachfolger
setKritischePfadeHelper(pfadKopie, nachfolger)
ø

5 Abweichung von der handschriftlichen Ausarbeitung

Bei der Erarbeitung des Prüfungsproduktes habe ich einige Änderungen am ursprünglichen Konzept vorgenommen.

5.1 Datenmodell

Einige Methoden- und Klassennamen wurden zur Einheitlichkeit des Codes ins deutsche übersetzt.

5.1.1 Die Sichtbarkeiten der Methoden

Die im Konzept als private gesetzten Hilfsmethoden wurden als im Package sichtbar gesetzt, damit Unittests erstellt werden konnten (vgl. Kapitel 6).

5.1.2 Klasse `Model`

Es wurde ein privates Attribut `initialized` hinzugefügt, um sicherzustellen, dass ein Model nur einmal initialisiert werden kann. Es wurde eine Methode `initialize()` hinzugefügt, um den initialized auf true zu setzen.

Es wurde ein Attribut `isZusammenhaengend` hinzugefügt, welches kapselt, ob der Netzplan zusammenhängend ist.

Es wurde ein Attribut `gueltigeReferenzen` hinzugefügt, welches kapselt, ob der Netzplan gültige Referenzen besitzt, also ob alle Referenzen in den Knoten des Netzplans korrekt sind und somit ob jeder Nachfolger eines Knotens auch in dessen Vorgängern enthalten ist bzw. ob jeder Vorgänger eines Knotens auch in dessen Nachfolgern enthalten ist.

5.1.3 Klasse `Knoten`

Der Konstruktor eines Knoten erwartet als Parameter nun einen Integer `vorgangsnummer`, einen String `vorgangsbezeichnung`, einen Integer `dauer`, eine ArrayList<Integer> `vorgaengerNummern` und eine ArrayList<Integer> `nachfolgerNummern`.

5.1.4 Klasse `Controller`

Der Controller hat eine öffentliche Hauptmethode `calculate` dazu erhalten, über die die gesamte Verarbeitung des Models gelingt. Zudem sind einige nicht öffentliche Hilfsmethoden dazugekommen, um die Verarbeitung des Models zu gewährleisten:

- `hatKeineZyklen():boolean` prüft, ob ein im Model gekapselter Graph zykelfrei ist. Eine weitere Hilfsmethode `hatKeineZyklenHelper(Knoten):boolean` ermöglicht die Überprüfung der Zykelfreiheit mittels Backtracking.
- `istZusammenhaengend():boolean` prüft, ob ein Graph zusammenhängend ist. Hier ermöglicht ebenfalls eine Helper-Methode namens

`istZusammenhaengendHelper(Knoten):boolean` die Überprüfung mittels Backtracking.

- `hatGueltigeReferenzen():boolean` prüft, ob die Referenzen aller Knoten korrekt angegeben sind, also ob alle Referenzen in den Knoten des Netzplans korrekt sind und somit ob jeder Nachfolger eines Knotens auch in dessen Vorgängern enthalten ist bzw. ob jeder Vorgänger eines Knotens auch in dessen Nachfolgern enthalten ist.
- Die Hilfsmethoden `setFez(Knoten): void`, `getFez(Knoten):int`, `setSez(Knoten):void`, `getSez(Knoten):void`, `getFp(Knoten):int`, `getGP(Knoten):int` wurden ersetzt durch geeignetere Methoden, da diese Fehler enthielten:
    - Die neue Methode `setFezAndFaz(Knoten):void` setzt FEZ und FAZ ausgehend von einem aktuellen Knoten für diesen und alle Nachfolger dieses Knotens.
    - Die neue Methode `setSazAndSez(Knoten):void` setzt SAZ für den aktuell betrachteten Knoten sowie alle Vorgängerknoten, ausgehend vom aktuell betrachteten Knoten.
    - Die neue Methode `getMaxFezOfVorgaenger(Knoten):int` berechnet den Maximalen FEZ aller Vorgänger eines Knoten.
    - Die neue Methode `getMinSazOfNachfolger(Knoten):int` berechnet den minimalen SAZ der Nachfolgenden Knoten eines betrachteten Knoten.
    - Die neue Methode `getMinFazOfNachfolger(Knoten)` berechnet den kleinsten FAZ aller Nachfolger eines betrachteten Knoten.

5.1.5 Klasse `LeseAusDatei` (Ursprünglich `InputFromFile`)

Es wurde eine Methode `vorgangsNummernNichtDoppelt(List<Integer>):boolean` hinzugefügt, die prüft, ob Vorgangsnummern nicht mehrfach vorkommen, da dies bei der Initialisierung der Knoten zu schwerwiegenden Fehlern führen würde.

Es wurde eine Methode `alleKnotenVerweisenAufExistierendenKnoten (List<Knoten>, List<Integer>)` hinzugefügt. Die Methode prüft, ob alle Knoten auf einen existierenden Knoten verweisen.

In der `LeseAusDatei()` Methode werden zu Beginn mehrere Fehlerfälle ausgeschlossen. So wird geprüft, ob alle Knoten auf existierende Knoten verweisen und ob Vorgangsnummern nicht mehrfach vorkommen. Treten diese auf, wird jeweils ein entsprechender Fehler auf der Konsole ausgegeben und in der Ausgabe der Datei auf diesen hingewiesen. Werden Strings statt Zahlen eingegeben oder Leerzeichen statt Zahlen, so wird ein entsprechender Fehler auf der Konsole ausgegeben und die Ausgabe entsprechend gestaltet.

5.1.6 Abstrakte Klasse `Ausgabe` (ursprünglich `Output`)

Die abstrakte Klasse Ausgabe wurde mithilfe verschiedener nicht-öffentlicher Hilfsmethoden etwas entzerrt. Die Methode `getAusgabeString():String` sammelt jedoch weiterhin die gesamte Erstellung des Ausgabestrings.

Der Konstruktor der Klasse wird nun mit einem Model aufgerufen, welches als privates Attribut `model` in der Klasse gekapselt wird.

5.1.7 Klasse `AusgabeInDatei` (ursprünglich `OutputToFile`)

Der Konstruktor der Klasse wird ähnlich wie die Klasse `Ausgabe`, von der die Klasse erbt, mit einem Model aufgerufen, welches anschließend an den `super(Model)`-Konstruktor übergeben wird.

6 Unittests

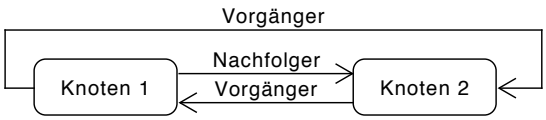
Zum Testen der korrekten Funktionalität des Controllers wurden grundlegende Unittests geschrieben. Sie werden geschrieben, um die funktionalen Einzelteile von Methoden eines Programms zu testen. Unittests gehören zur Gruppe der White-Box-Tests, also zur Gruppe der Tests, die mit Kenntnissen über die innere Funktionsweise des zu testenden Systems ablaufen.

Es wurden Unittests für drei kritische Methoden des Controllers geschrieben (`hatKeineZyklen()`, `isZusammenhaengend()`, `hatGueltigeReferenzen()`)

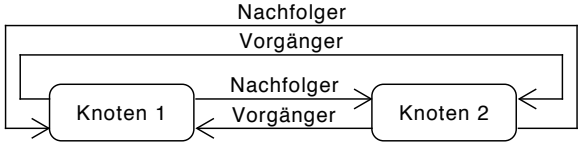
6.1 Prüfung der Methode `hatKeineZyklen()`

Mittels 5 Unittests wird die Funktionalität der Methode `hatKeineZyklen()` auf Korrektheit hin überprüft.

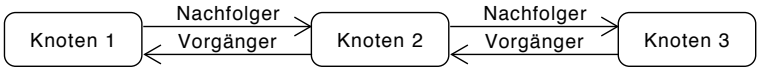
- hatKeineZyklen_ModelOhneZyklen_RueckgabeTrue()
    - Prüft, ob ein Graph aus zwei Knoten ohne Zyklen als zyklenfrei ausgegeben wird.
- hatKeineZyklen_ZweiterKnotenHatErstenKnotenAlsNachfolger_RueckgabeFalse()
    - Prüft, ob ein Graph mit zwei Knoten, bei dem der zweite Knoten den ersten als Nachfolger hat, als zykelbehaftet ausgegeben wird.
- hatKeineZyklen_ErsterKnotenHatZweitenKnotenAlsVorgaengerSowieNachfolgerUndZweiterKnotenHatErstenKnotenAlsVorgaenger_RueckgabeTrueDaKeinExistierenderStartpunkt()
    - Prüft, ob ein Model mit einem Graphen, welcher keinen Startpunkt hat, als zykelfrei ausgegeben wird. Der erste Knoten hat den zweiten Knoten als Vorgänger und als Nachfolger. Der zweite Knoten hat den ersten als Vorgänger. Somit existiert kein Startpunkt:



- hatKeineZyklen_ZweiKnotenHabenSichGegenseitigAlsNachfolgerSowieVorgaenger_RueckgabeTrueDaKeinExistierenderStartpunkt
    - Prüft, ob kein Zykel vorliegt, wenn zwei Knoten sich gegenseitig als Nachfolger und als Vorgänger haben. Somit existiert kein Startpunkt.



- hatKeineZyklen_DritterKnotenHatZweitenKnotenAlsNachfolger_RueckgabeFalse
    - prüft, ob eine einfache Kette von drei Knoten keinen Zykel hat.



6.2 Prüfung der Methode `isZusammenhaengend()`

Mittels zweier Unittests wird die Funktionalität der Methode `isZusammenhaengend()` auf Korrektheit hin überprüft:

- isZusammenhaengend_ZusammenhaengendeKnoten_RueckgabeTrue()
    - Prüft, ob eine einfache Reihe von drei Knoten als zusammenhängend erkannt wird.
- isZusammenhaengend_DritterKnotenHatEinenVorgaengerAberDieserKeinenNachfolger_RueckgabeFalse()
    - Prüft, ob eine Liste von drei Knoten nicht zusammenhängend ist, bei der der erste Knoten den zweiten als Nachfolger hat, der zweiten den ersten als Vorgänger hat, den dritten jedoch nicht als Nachfolger. Der dritte Knoten hat den zweiten Knoten als Vorgänger:

6.3 Prüfung der Methode `hatGueltigeReferenzen()`

- hatGueltigeReferenzen_dreiKnotenMitFehlenderReferenzVomZweitenZumDrittenKnoten_nichtGueltig()
    - Prüft, ob Drei Knoten, bei der keine Referenz vom zweiten zum dritten Knoten existiert, nicht gültig ist:

- hatGueltigeReferenzen_dreiKnotenMitKorrektGesetztenReferenzen_istGueltig()
    - Prüft, ob drei Knoten, bei denen die Referenzen korrekt gesetzt wurden, als gültig akzeptiert wird.

7 Blackbox- Testfälle

Die in diesem Kapitel beschriebenen Testfälle werden nach dem Backbox-Testing-Prinzip durchgeführt. Es wird also nicht die konkrete Implementierung des Programms, sondern lediglich das Verhalten des Programms nach Außen untersucht. Es wird konkret überprüft, ob die Ausgaben des Programms bei entsprechenden Eingaben den erwarteten Ausgaben entsprechen.

Als erstes werden die Testbeispiele aus der Aufgabenstellung untersucht. Anschließend werden weitere Normalfälle, Sonderfälle und mögliche Fehlerfälle untersucht.

Normalfälle sind Fälle, die den definierten Eingabevorgaben entsprechen. Die Gültigkeit einer Eingabe ist im Kapitel Format der Eingabedatei genau erklärt.

Sonderfälle sind Fälle, bei denen die grundlegende Formatierung der Eingabedatei nicht gültig ist, das Programm jedoch dennoch zu einem korrekten Ergebnis kommt. Die fehlerhafte Erfüllung der Fehlerbehafteten Erfüllung der Eingabestruktur wird also bei Sonderfällen ignoriert.

Unter Fehlerfällen werden die Fälle verstanden, die in der Konsolenausgabe als explizite Fehler ausgegeben werden. Sie führen dazu, dass das Programm nicht die gewünschten Ausgaben produziert und daher mit einer entsprechenden Ausgabe in der Ausgabedatei kenntlich gemacht werden.

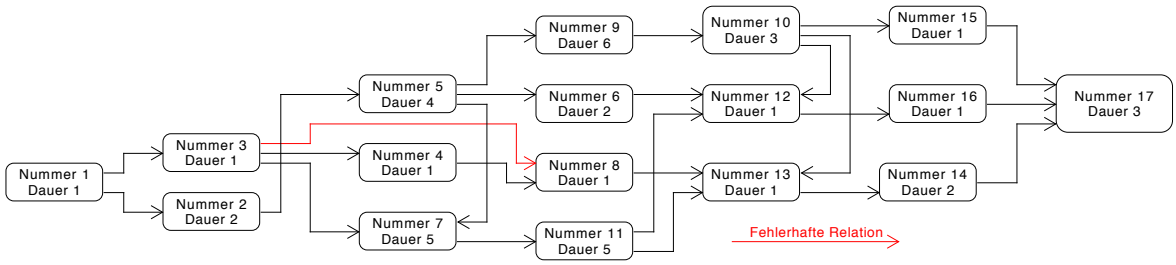
7.1 Besonderheiten der Beispiele 2, 3 und 5 der durch die IHK verbesserten Aufgabenstellung

Das IHK-Beispiel Nummer 2 („*Wasserfallmodell*“) aus der verbesserten Aufgabenstellung war fehlerhaft. Das bedeutet, dass sie nicht das angegebene Ergebnis lieferte, da die Eingabe falsch formuliert war. Die Vorgangsnummer 6 („*Einsatz und Wartung*“) referenziert auf einen nichtexistierenden Knoten mit Nachfolgernummer 7. Es existieren jedoch nur 6 Knoten. Die angegebene Ausgabe ist also falsch, da hier ein Fehlerfall vorliegt. Es muss also wie im unter Kapitel 7.4.2.1 im Kapitel 7.4 (Fehlerfälle) ein entsprechender Fehler auf der Konsole- und eine entsprechende Ausgabe in der Datei erfolgen.

Das IHK-Beispiel Nummer 3 („*Beispiel 3*“) aus der verbesserten Aufgabenstellung war ebenfalls fehlerhaft. Das bedeutet, dass sie nicht das angegebene Ergebnis lieferte, da die Eingabe falsch formuliert war. Die Ausgabe müsste wie unter Normalfall zusammenhängender Graph im Kapitel Eigene Normalfälle für Vorgang Nummer 8 („*Tee trinken*“) ein SEZ- Wert von 12 statt 13 errechnet werden.

Das IHK-Beispiel Nummer 5 („*Beispiel 3 IT-Installation*“) aus der verbesserten Aufgabenstellung war ebenfalls fehlerhaft. Das bedeutet, dass sie nicht das angegebene Ergebnis lieferte, da die Eingabe falsch formuliert war. Vorgang Nummer 4 („*Peripheriebedarf ermitteln*“) hat einen Nachfolger 8, Vorgang Nummer 8 hat jedoch keinen

Vorgänger 3, sondern lediglich den Vorgänger 3 („*Netzwerkplan entwerfen*“). Das Problem ist in der nachfolgenden Abbildung illustriert:



Das Beispiel müsste also statt des angegebenen- eine Eingabestruktur nach dem unter Normalfall Komplexes Beispiel im Kapitel Eigene Normalfälle angegebene Testbeispiel haben. Bei der angegebenen Eingabedatei müsste ähnlich dem Fehlerfall Fehlerhafte Referenz eine Fehlermeldung auf der Konsole und eine entsprechende Ausgabe in der Datei ausgegeben werden.

7.2 Normalfälle

7.2.1 Beispiele aus der durch die IHK verbesserten Aufgabenstellung

7.2.1.1 Beispiel 01 Verzweigter Graph – „Installation von POI Kiosken“

Der Testfall aus der Aufgabenstellung beschreibt einen einfach verzweigten Graphen mit insgesamt 7 Knoten.

Die Eingabe wird erfolgreich eingelesen und korrekt ausgewertet und ausgegeben.

Eingabe

```
//************************************************
//+ Installation von POI Kiosken
//************************************************
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Planung des Projekts; 1; -; 2,4
2; Beschaffung der POI-Kioske; 25; 1; 3
3; Einrichtung der POI-Kioske; 10; 2; 6
4; Netzwerk installieren; 6; 1; 5
5; Netzwerk einrichten; 1; 4; 6
6; Aufbau der POI Kioske; 2; 3,5; 7
7; Tests und Nachbesserung der POI Kioske; 1; 6; -
```

Ausgabe

```
Installation von POI Kiosken


Vorgangsnummer; Vorgangsbeschreibung; D; FAZ; FEZ; SAZ; SEZ; GP; FP
1; Planung des Projekts; 1; 0; 1; 0; 1; 0; 0
2; Beschaffung der POI-Kioske; 25; 1; 26; 1; 26; 0; 0
3; Einrichtung der POI-Kioske; 10; 26; 36; 26; 36; 0; 0
4; Netzwerk installieren; 6; 1; 7; 29; 35; 28; 0
5; Netzwerk einrichten; 1; 7; 8; 35; 36; 28; 28
6; Aufbau der POI Kioske; 2; 36; 38; 36; 38; 0; 0
7; Tests und Nachbesserung der POI Kioske; 1; 38; 39; 38; 39; 0; 0


Anfangsvorgang: 1
Endvorgang: 7
Gesamtdauer: 39


Kritischer Pfad
1->2->3->6->7
```

7.2.2 Eigene Normalfälle

7.2.2.1 Normalfall eines einfachen und linearen Graphen

Dieser Normalfall entspricht einem einfachen linearen Graphen und entspricht einer verbesserten Version des Fehlerhaften Beispiels 2 („Wasserfallmodell“) aus der IHK-Aufgabenstellung.
Dieses Beispiel verdeutlicht den nahezu einfachsten Fall eines Netzplans, da keinerlei Verzweigungen vorliegen.

Die Eingabe wird erfolgreich eingelesen und korrekt ausgewertet und ausgegeben.

Eingabe

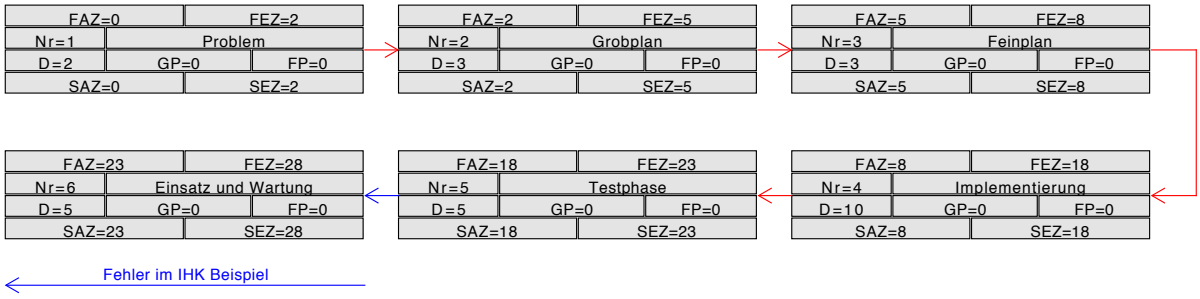
```
//*****************************************************************************
//+ Testfall Linearer Graph - Wasserfallmodell verbessert aus IHK-Aufgabenstellung
//*****************************************************************************
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Problemanalyse; 2; -; 2
2; Grobplanung; 3; 1; 3
3; Feinplanung; 3; 2; 4
4; Implementierung; 10; 3; 5
5; Testphase; 5; 4; 6;
6; Einsatz und Wartung; 5; 5; -
```

Ausgabe

```
Testfall Linearer Graph - Wasserfallmodell verbessert aus IHK-Aufgabenstellung


Vorgangsnummer; Vorgangsbeschreibung; D; FAZ; FEZ; SAZ; SEZ; GP; FP

1; Problemanalyse; 2; 0; 2; 0; 2; 0; 0

2; Grobplanung; 3; 2; 5; 2; 5; 0; 0

3; Feinplanung; 3; 5; 8; 5; 8; 0; 0

4; Implementierung; 10; 8; 18; 8; 18; 0; 0

5; Testphase; 5; 18; 23; 18; 23; 0; 0

6; Einsatz und Wartung; 5; 23; 28; 23; 28; 0; 0


Anfangsvorgang: 1

Endvorgang: 6

Gesamtdauer: 28


Kritischer Pfad

1->2->3->4->5->6
```

Graphische Abbildung des Testfalls

Im Folgenden ist eine grafische Darstellung des Testfalls angegeben, die die Abweichung von der IHK-Beispielaufgabe farblich (blau) hervorhebt:



7.2.2.2 Normalfall Komplexes Beispiel

Das Beispiel stellt einen relativ komplexen Fall eines Netzplans dar. Es ist im Grunde das Beispiel 5 („IT-Installation“) aus der Aufgabenstellung, jedoch wurden zwei Knoten verändert, damit die Referenzen stimmen. Knoten 3 („*Netzplan entwerfen*“) hat nun die Nachfolger 4,7 und 8. Knoten 8 hat die Vorgänger 3 und 4.

Das Beispiel ist zykelfrei und zusammenhängend und besitzt ausgehend von einem Startknoten und einem Endknoten mehrere Parallele Stränge.

Die Eingabe wird erfolgreich eingelesen und korrekt ausgewertet und ausgegeben.

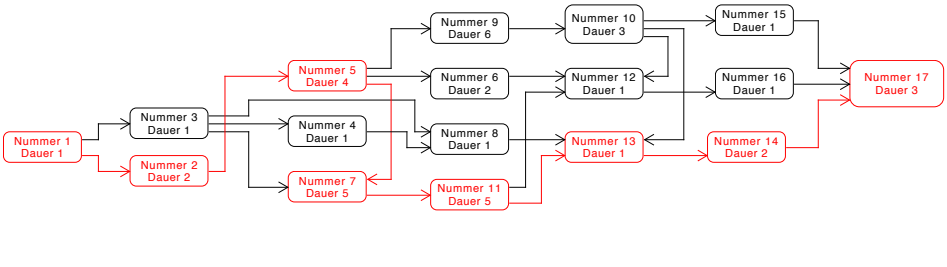
Eingabe

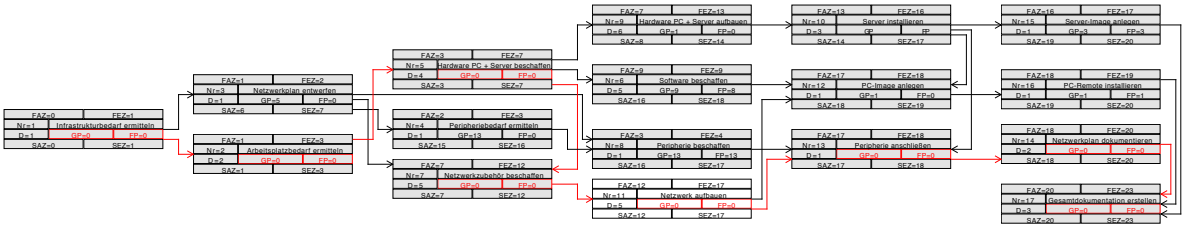
```
//***********************************************
//+ Beispiel 5 IT-Installation
//***********************************************
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Infrastrukturbedarf ermitteln; 1; -; 2,3
2; Arbeitsplatzbedarf ermitteln; 2; 1; 5
3; Netzwerkplan entwerfen; 1;1; 4,7,8
4; Peripheriebedarf ermitteln; 1; 3; 8
5; Hardware PC + Server beschaffen; 4; 2; 6,7,9
6; Software beschaffen; 2; 5; 12
7; Netzwerkzubehör beschaffen; 5; 3,5; 11
8; Peripherie beschaffen; 1; 3,4; 13
9; Hardware PC + Server aufbauen; 6; 5; 10
10; Server installieren; 3; 9; 12,13,15
11; Netzwerk aufbauen; 5; 7; 12,13
12; PC-Image anlegen; 1; 6,10,11; 16
13; Peripherie anschließen; 1; 8,10,11; 14
14; Netzwerkplan dokumentieren; 2; 13; 17
15; Server-Image anlegen; 1; 10; 17
16; PC-Remote installieren; 1; 12; 17
17; Gesamtdokumentation erstellen; 3; 14,15,16; -
```

Ausgabe

```
Beispiel 5 IT-Installation


Vorgangsnummer; Vorgangsbeschreibung; D; FAZ; FEZ; SAZ; SEZ; GP; FP

1; Infrastrukturbedarf ermitteln; 1; 0; 1; 0; 1; 0; 0

2; Arbeitsplatzbedarf ermitteln; 2; 1; 3; 1; 3; 0; 0

3; Netzwerkplan entwerfen; 1; 1; 2; 6; 7; 5; 0

4; Peripheriebedarf ermitteln; 1; 2; 3; 15; 16; 13; 0

5; Hardware PC + Server beschaffen; 4; 3; 7; 3; 7; 0; 0

6; Software beschaffen; 2; 7; 9; 16; 18; 9; 8

7; Netzwerkzubehör beschaffen; 5; 7; 12; 7; 12; 0; 0

8; Peripherie beschaffen; 1; 3; 4; 16; 17; 13; 13

9; Hardware PC + Server aufbauen; 6; 7; 13; 8; 14; 1; 0

10; Server installieren; 3; 13; 16; 14; 17; 1; 0

11; Netzwerk aufbauen; 5; 12; 17; 12; 17; 0; 0

12; PC-Image anlegen; 1; 17; 18; 18; 19; 1; 0

13; Peripherie anschließen; 1; 17; 18; 17; 18; 0; 0

14; Netzwerkplan dokumentieren; 2; 18; 20; 18; 20; 0; 0

15; Server-Image anlegen; 1; 16; 17; 19; 20; 3; 3

16; PC-Remote installieren; 1; 18; 19; 19; 20; 1; 1

17; Gesamtdokumentation erstellen; 3; 20; 23; 20; 23; 0; 0


Anfangsvorgang: 1

Endvorgang: 17

Gesamtdauer: 23


Kritischer Pfad

1->2->5->7->11->13->14->17
```

Grafische Darstellungen des Testfalls





Diskussion

Der Testfall zeigt, dass das Programm in der Lage ist, auch Netzpläne mit einer Knotenanzahl von 17 und mehreren parallelen Ästen in kurzer Zeit auszuwerten.

7.3 Sonderfälle

7.3.1 Eigene Sonderfälle

7.3.1.1 Negative Vorgangsnummern

Es wird eine Testdatei eingelesen, die über eine negative Vorgangsnummer verfügt. Dies stellt zu den bisherigen Testfällen einen Sonderfall dar, da diese stets positive Vorgangsnummern hatten.

Die Eingabe wird erfolgreich eingelesen und korrekt ausgewertet und ausgegeben.

Eingabe

```
//************************************************
//+ Negative Vorgangsnummer
//************************************************
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
-1 ; Knoten A; 1; -; 2,4
2; Knoten B; 25; -1; 3
3; Knoten C; 10; 2; 6
4; Knoten D; 6; -1; 5
5; Knoten E; 1; 4; 6
6; Knoten F; 2; 3,5; 7
7; Knoten G; 1; 6; -
```

Ausgabe

```
Negative Vorgangsnummer


Vorgangsnummer; Vorgangsbeschreibung; D; FAZ; FEZ; SAZ; SEZ; GP; FP
-1; Knoten A; 1; 0; 1; 0; 1; 0; 0
2; Knoten B; 25; 1; 26; 1; 26; 0; 0
3; Knoten C; 10; 26; 36; 26; 36; 0; 0
4; Knoten D; 6; 1; 7; 29; 35; 28; 0
5; Knoten E; 1; 7; 8; 35; 36; 28; 28
6; Knoten F; 2; 36; 38; 36; 38; 0; 0
7; Knoten G; 1; 38; 39; 38; 39; 0; 0


Anfangsvorgang: -1
```

```
Endvorgang: 7
Gesamtdauer: 39

Kritischer Pfad
-1->2->3->6->7
```

Diskussion

Der Testfall zeigt, dass das Programm in der Lage ist auch negative Vorgangsnummern zu verarbeiten und ein gültiges Ergebnis zu liefern.

7.3.1.2 Keine Überschrift

Eingabe

```
//*****************************************************************************
//*****************************************************************************
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Problemanalyse; 2; -; 2
2; Grobplanung; 3; 1; 3
3; Feinplanung; 3; 2; -
```

Ausgabe

```
Vorgangsnummer; Vorgangsbeschreibung; D; FAZ; FEZ; SAZ; SEZ; GP; FP
1; Problemanalyse; 2; 0; 2; 0; 2; 0; 0
2; Grobplanung; 3; 2; 5; 2; 5; 0; 0
3; Feinplanung; 3; 5; 8; 5; 8; 0; 0

Anfangsvorgang: 1
Endvorgang: 3
Gesamtdauer: 8

Kritischer Pfad
1->2->3
```

7.4 Fehlerfälle

7.4.1 Beispiele der IHK

7.4.1.1 Zyklus im Gaphen

Das Beispiel der IHK zeigt einen Graphen mit einem Zyklus. Das Programm gibt einen Fehler auf der Konsole aus:

```
Beispiel 4 mit Zyklus: Zyklen enthalten
```

In der Ausgabedatei wird eine Fehlermeldung angegeben sowie der Zyklus aufgelistet.

Eingabe

```
//***********************************************
//+ Beispiel 4 mit Zyklus
//***********************************************
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Flughafen planen; 24; -; 2
2; Flughafen bauen; 24; 1; 3
3; Baumängel erkennen; 1; 2; 4
4; Baumängel beseitigen; 6; 3; 3,5
5; Flughafenbau abnehmen und genehmigen; 1; 4; 6
6; Flugbetrieb aufnehmen; 1; 5; -
```

Ausgabe

```
Beispiel 4 mit Zyklus


Berechnung nicht möglich.
Zyklus erkannt: 3->4->3
```

7.4.2 Eigene Fehlerfälle

7.4.2.1 Fehlerhafte Referenz

Der Fehlerfall ist ein Graph mit einer ungültigen Referenz auf einen Knoten 7. Es wird folgende Ausgabe auf der Konsole ausgegeben:

```
In Datei F_Fehlerhafte_Referenzen.in: Ungenügende Eingabe: Es existieren ungültige Referenzen,
da mindestens ein Knoten auf einen nicht existenten Knoten referenziert.
```

Eingabe

```
//*****************************************************************************

//+ Fehlerhafte Referenzen Minimal

//*****************************************************************************

//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger

1; Problemanalyse; 2; -; 2

2; Grobplanung; 3; 1; 3

3; Feinplanung; 3; 2; 4

4; Implementierung; 10; 3; 5

5; Testphase; 5; 4; 7;

6; Einsatz und Wartung; 5; 5; -
```

Ausgabe

```
Berechnung nicht möglich.

Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.
```

7.4.2.2 Mehrfache gleiche Vorgangsnummern

Der Fehlerfall beschreibt eine Eingabe, in der mehrfach die gleiche Vorgangsnummer vorkommt. Es wird folgende Ausgabe auf der Konsole ausgegeben:

```
In Datei F_Mehrfache_Vorgangsnummern.in: Ungenügende Eingabe: Es kommt mindestens eine
Vorgangsnummer mehrfach vor.
```

Eingabe

```
//**********************************

//+ Mehrfache gleiche Vorgangsnummern

//**********************************

//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger

1; Knoten A; 2; -; 2

2; Knoten B; 3; 1; 3

3; Knoten C; 3; 2; 4

3; Knoten D; 10; 3; 5

5; Knoten E; 5; 4; 6;

6; Knoten F; 5; 5; -
```

Ausgabe

```
Berechnung nicht möglich.

Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.
```

7.4.2.3 Strings statt Zahlen in der Eingabe

In diesem Testfall wird statt einer Zahl ein String in der Eingabedatei an einer Stelle eingetragen, an der eigentlich eine Zahl erwartet würde. Es wird folgender Fehler auf der Konsole ausgegeben:

In Datei F_StringsStattZahlen.in: Ungenügende Eingabe. Es wurde mindestens eine ungültige Zahl eingeben.

Eingabe

```
//***********************

//+ Strings statt Zahlen

//***********************

//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger

1; Knoten A; "zwei"; -; 2

2; Knoten B; 3; 1; 3

3; Knoten C; 3; 2; 4

4; Knoten D; 10; 3; 5

5; Knoten E; 5; 4; 6;

6; Knoten F; 5; 5; -
```

Ausgabe

```
Berechnung nicht möglich.

Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.
```

7.4.2.4 Leerstellen statt gültigen Zahlen 1

Es wird ein Leerzeichen als Vorgangsnummer eingegeben. Es wird folgender Fehler auf der Konsole ausgegeben:

```
In Datei F_LeerstellenStattErforderlichenWerten01.in: Ungenügende Eingabe. Es wurde mindestens
eine ungültige Zahl eingeben.
```

Eingabe

```
//***************************************

//+ Leerstellen statt erforderlichen Werte

//***************************************

//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger

; Knoten A; 2; -; 2

2; Knoten B; 3; 1; 3

3; Knoten C; 3; 2; 4

4; Knoten D; 10; 3; 5
```

```
5; Knoten E; 5; 4; -;
```

Ausgabe

```
Berechnung nicht möglich.

Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.
```

7.4.2.5 Leerstellen statt gültigen Zahlen 2

Es wird ein Leerzeichen als Dauer eingegeben. Es wird folgender Fehler auf der Konsole ausgegeben:

```
In Datei F_LeerstellenStattErforderlichenWerten02.in: Ungenügende Eingabe. Es wurde mindestens
eine ungültige Zahl eingeben.
```

Eingabe

```
//****************************************
//+ Leerstellen statt erforderlichen Werte
//****************************************
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Knoten A; ; -; 2
2; Knoten B; 3; 1; 3
3; Knoten C; 3; 2; 4
4; Knoten D; 10; 3; 5
5; Knoten E; 5; 4; -;
6; Knoten F; 5; 5; -
```

Ausgabe

```
Berechnung nicht möglich.

Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.
```

7.4.2.6 Nicht zusammenhängend

Es wird ein nicht zusammenhängender Graph eingelesen. Dabei wird folgender Fehler auf der Konsole ausgegeben:

```
Testfall nicht Zusammenhängend: Fehler (Nicht zusammenhängend)
```

Eingabe

```
//************************************************
//+ Testfall nicht Zusammenhängend
//************************************************
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Problemanalyse; 2; -; 2
2; Grobplanung; 3; 1; 3
3; Feinplanung; 3; 2; 5
4; Implementierung; 10; 3; 5
```

```
5; Testphase; 5; 4; 6

6; Einsatz und Wartung; 5; 5; -
```

Ausgabe

```
Testfall nicht Zusammenhängend


Berechnung nicht möglich.

Nicht zusammenhängend.
```

7.4.2.7 Keine Leerzeichen wie erwartet

Die Testdatei hat nicht wie erwartet Leerzeichen nach jedem Semikolon und nach “//+“ in der Überschrift- Kommentarzeile.

```
In Datei F_KeineLeerzeichen.in: Ungenügende Eingabe.
```

Eingabe

```
//*****************************************************************************

//+TestfallkeineLeerzeichen

//*****************************************************************************

//Vorgangsnummer;Vorgangsbezeichnung;Dauer;Vorgänger;Nachfolger

1;Problemanalyse;2;-;2

2;Grobplanung;3;1;3

3;Feinplanung;3;2;4

4;Implementierung;10;3;5

5;Testphase;5;4;6;

6;EinsatzundWartung;5;5;-
```

Ausgabe

```
Berechnung nicht möglich.

Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.
```

7.4.2.8 Minuszeichen vergessen

Es wird ein Minuszeichen beim Startknoten (Vorgänger) nicht angegeben. Stattdessen wird ein leerer String eingefügt.

```
In Datei F_MinuszeichenVergessen.in: Ungenügende Eingabe. Es wurde mindestens eine ungültige Zahl eingeben.
```

Eingabe

```
//*****************************************************************************

//+ Minuszeichen vergessen

//*****************************************************************************

//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger

1; Problemanalyse; 2; ; 2

2; Grobplanung; 3; 1; 3

3; Feinplanung; 3; 2; -
```

Ausgabe

```
Berechnung nicht möglich.

Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.
```

7.4.2.9 Leere Datei

Es wird eine leere Datei eingelesen. Es wird folgender Fehler auf der Konsole ausgegeben:

```
In Datei F_LeereDatei.in: Ungenügende Eingabe: Es wurden keinerlei Vorgänge angegeben.
```

Eingabe

Ausgabe

```
Berechnung nicht möglich.

Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.
```

8 Zusammenfassung und Ausblick

Im Rahmen des Prüfungsprodukts wurde ein Programm entwickelt, welches Netzpläne erstellen und auswerten kann. Es wurde eine Anforderungsanalyse vorgenommen, in der die Anforderungen an das Programm erläutert und spezifiziert wurden. Aus einer verbalen Beschreibung des Ablaufs des Programms wurden UML-Klassendiagramme, ein Sequenzdiagramm und Nassi Shneidermann-Diagramme für die relevanten Algorithmen des Programms entwickelt.

Dem Anwender steht eine ausführliche Anleitung zur Benutzung des Programms zur Verfügung.

Mithilfe von Blackbox-Tests wurden die relevanten Normal-, Sonder- und Fehlerfälle ausgiebig getestet.

8.1 Ausblick

Um die Benutzerfreundlichkeit zu erhöhen könnte eine grafische Eingabemaske für die Knoten des Netzplans erstellt werden.

Es könnte eine ebenfalls grafische Ausgabe erstellt werden, die die Knoten und ihren möglichen Kritischen Pfad darstellt (vgl. Abbildung in Kapitel Testfälle).

9 Anhang: Programmcode

9.1 Package main 39
9.1.1 Klasse Main 39
9.2 Package io 40
9.2.1 Klasse LeseAusDatei 40
9.2.2 Klasse Ausgabe 42
9.2.3 Klasse AusgabeInDatei 45
9.3 Package controller 46
9.3.1 Klasse Controller 46
9.3.2 Unittest Klasse Controller 52
9.4 Package model 56
9.4.1 Klasse Knoten 56
9.4.2 Klasse Model 58

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;

        // Nur zum Testen- wird später aus über die Argumente args der Jar
        // übergeben
        args = new String[2];
        args[0] = ".in";
        args[1] = "/Users/hfs23/Dropbox/MATSE/Programmieraufgaben/2018_GrosseProg/Testfaelle";

        // 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

```java
package io;

import java.util.ArrayList;

import model.Knoten;
import model.Model;

/**
 * Ermöglicht zu einem Model die Ausgabe der kenngrößen und kritischen Pfade
 * auszugeben
 *
 * @author M. Leonard Haufs Prüflingsnummer: 101-20540
 *
 */
public abstract class Ausgabe {
    private Model model;

    /**
     * Konstruktor, der Ausgabe mit einem Model initialisiert
     *
     * @param model
     *            model, welches die auszugebenen Daten enthällt
     */
    public Ausgabe(Model model) {
        super();
        this.model = model;
    }

    /**
     * Gibt den Ausgabestring zurück.
     *
     * Falls nicht zusammenhängend oder falls Zyklen enthalten sind, wird ein
     * entsprechender Fehler ausgegeben.
     *
     * @return Ausgabestring
     */
    protected String getAusgabeString() {
        StringBuilder sb = new StringBuilder();

        if (this.model.getKnoten().size() == 0) {
            sb.append("Berechnung nicht möglich.");
            sb.append("\n");
            sb.append("Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu
                erhalten.");
        } else if (this.model.getZyklus().size() != 0) {
            sb.append(this.model.getName());
            sb.append("\n");
            sb.append("\n");
            sb.append("Berechnung nicht möglich.");
            sb.append("\n");
            sb.append("Zyklus erkannt: ");
            this.getZyklusString(sb);
        } else if (!this.model.isZusammenhaengend()) {
            sb.append(this.model.getName());
            sb.append("\n");
            sb.append("\n");
            sb.append("Berechnung nicht möglich.");
            sb.append("\n");
            sb.append("Nicht zusammenhängend.");
        } else if (!this.model.isGueltigeReferenzen()) {
            sb.append(this.model.getName());
            sb.append("\n");
            sb.append("\n");
            sb.append("Berechnung nicht möglich.");
            sb.append("\n");
            sb.append(
                    "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.");
        } else {
            sb.append("Vorgangsnummer; Vorgangsbeschreibung; D; FAZ; FEZ; SAZ; SEZ; GP; FP");
            sb.append("\n");
            this.getKnotenbeschreibung(sb);
            sb.append("\n");
            this.getVorgangString(sb);
            sb.append("\n");
            this.getGesamtdauer(sb);
            sb.append("\n");
            sb.append("\n");
            this.getKritischerPfadString(sb);
        }

        return sb.toString();
    }

    /**
     * Gibt die Beschreibung eines Knotens im Netzplan. Dabei wird der übergebene
     * StringBuilder verändert.
     *
     * @param sb
     *            Stringbuilder, an den die Beschreibung angehängt werden soll
     */
    private void getKnotenbeschreibung(StringBuilder sb) {
        for (Knoten knoten : model.getKnoten()) {
            sb.append(knoten.getVorgangsnummer());
            sb.append("; ");
            sb.append(knoten.getVorgangsbezeichnung());
            sb.append("; ");
            sb.append(knoten.getDauer());
            sb.append("; ");
            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_ErsterKnotenHatZweitenKnotenAlsVorgaengerSowieNachfolgerUndZweiterKnotenHatErstenKnotenAlsVo
           {
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     }
```