

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 Datenmodell	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 <code>Model</code>	18
5.1.3	Klasse <code>Knoten</code>	18

5.1.4	Klasse <code>Controller</code>	18
5.1.5	Klasse <code>LeseAusDatei</code> (Ursprünglich <code>InputFromFile</code>).....	19
5.1.6	Abstrakte Klasse <code>Ausgabe</code> (ursprünglich <code>Output</code>).....	19
5.1.7	Klasse <code>AusgabeInDatei</code> (ursprünglich <code>OutputToFile</code>).....	20
6	Unit Tests	21
6.1	Prüfung der Methode <code>hatKeineZyklen()</code>	21
6.2	Prüfung der Methode <code>isZusammenhaengend()</code>	22
6.3	Prüfung der Methode <code>hatGuelteReferenzen()</code>	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	31
7.4.1	Beispiele der IHK	31
7.4.2	Eigene Fehlerfälle	32
8	Zusammenfassung und Ausblick	37
8.1	Zusammenfassung	37
8.2	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 Programmcode mittels des Eclipse- Plugins *ObjectAid UML Explorer*² erzeugt.

Für die Erstellung des Sequenzdiagramms und der anderen Abbildungen dieses Dokumentes wurde das Programm Umlet³ 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 Anfangszeitpunkt (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: $SAZ = SEZ - \text{Dauer}$. 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 ganzzahlige 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: $FEZ = FAZ + \text{Dauer}$. 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: $SAZ = SEZ - \text{Dauer}$. Der SAZ eines Vorgangs wird SEZ aller unmittelbarer Vorgänger. Haben

3 Aufgabenanalyse

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 $FP = (\text{kleinster FAZ der nachfolgenden Knoten}) - FEZ$

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: $FEZ = FAZ + \text{Dauer}$. 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: $SAZ = SEZ - \text{Dauer}$. 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 $FP = (\text{kleinster FAZ der nachfolgenden Knoten}) - FEZ$

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:

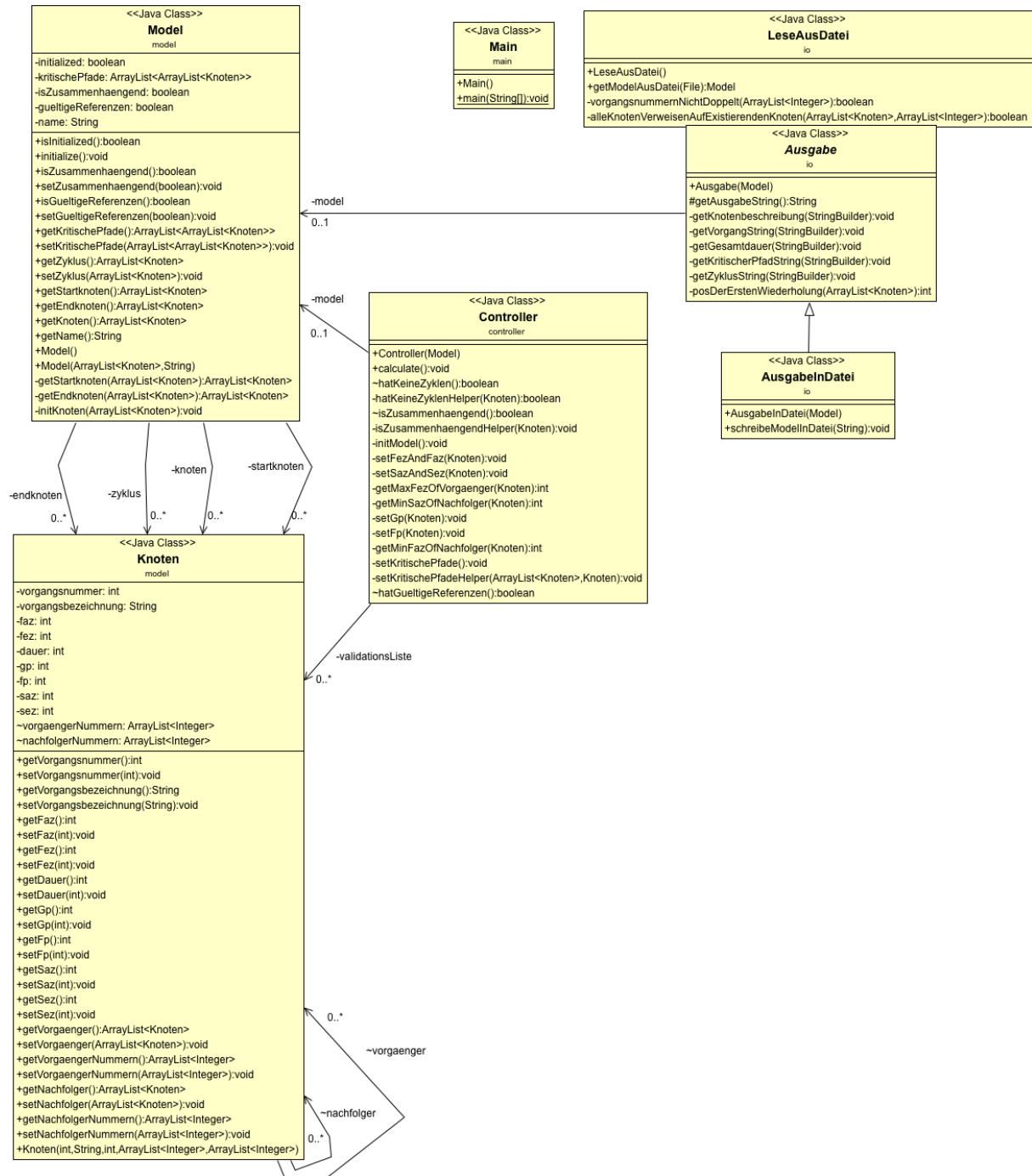
Diese prüft zunächst die Abbruchbedingung, dass der aktuell in der Hilfsmethode betrachtete Knoten ein Endpunkt ist. Ist dies der Fall, wird der berechnete Pfad im externen Model zu

3 Aufgabenanalyse

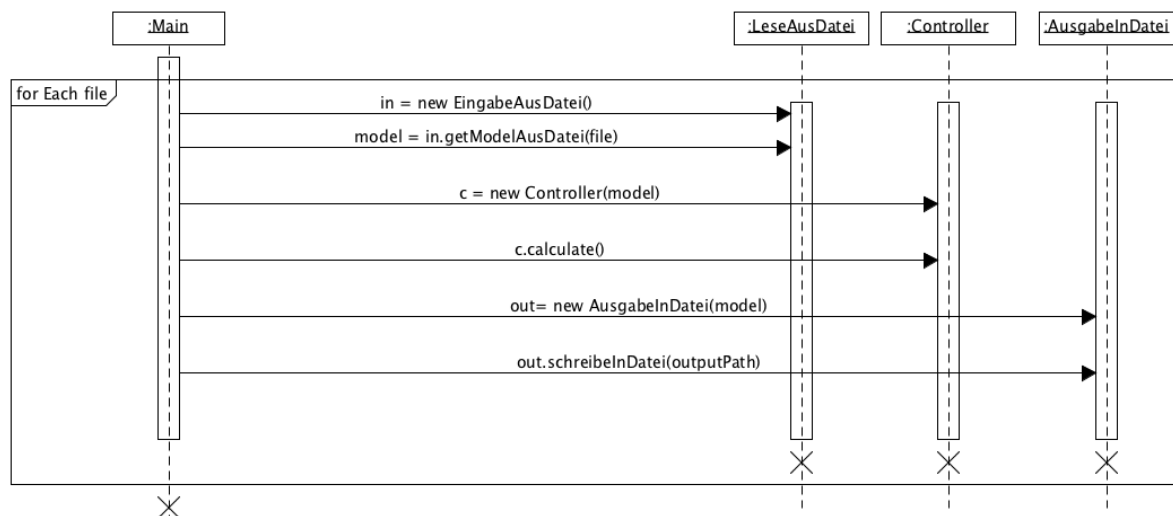
einer Liste hinzugefügt und die Methode beendet. Ansonsten wird geprüft, ob der aktuelle Knoten das Kriterium für einen Kritischen Pfad erfüllt ($GP = 0$ und $FP = 0$). Ist dies der Fall, so wird der aktuelle Knoten zum Pfadarray hinzugefügt und die Hilfsmethode auf jedem Nachfolger des aktuellen Knotens aufgerufen.

4 Programmkonzeption

4.1 UML Klassendiagramm

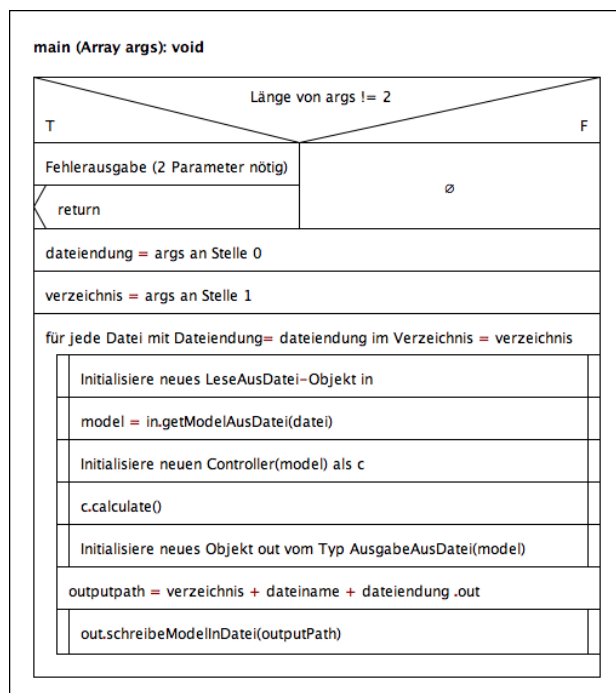


4.2 Programmablauf im Sequenzdiagramm

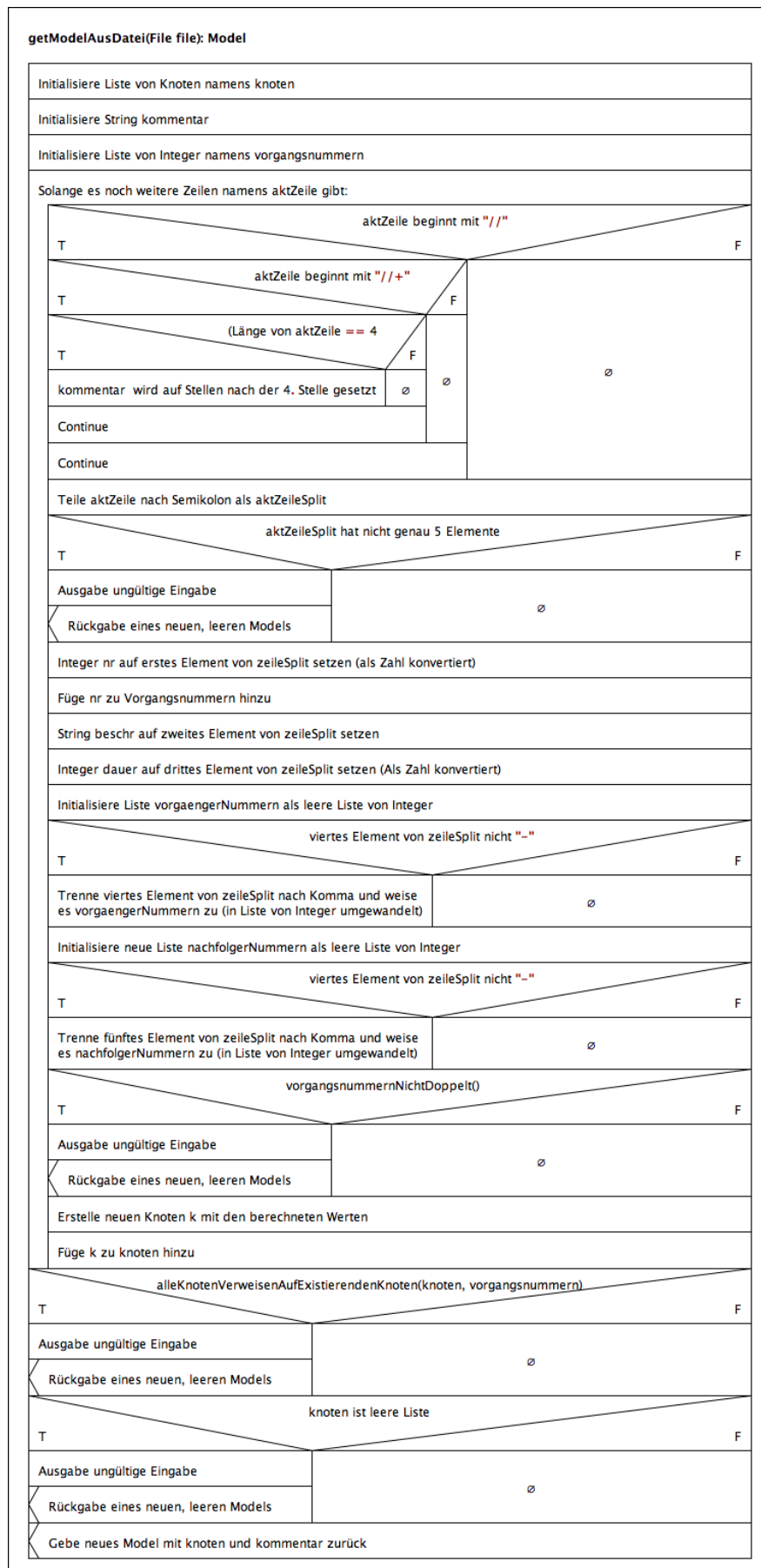


4.3 Nassi-Shneiderman-Diagramme

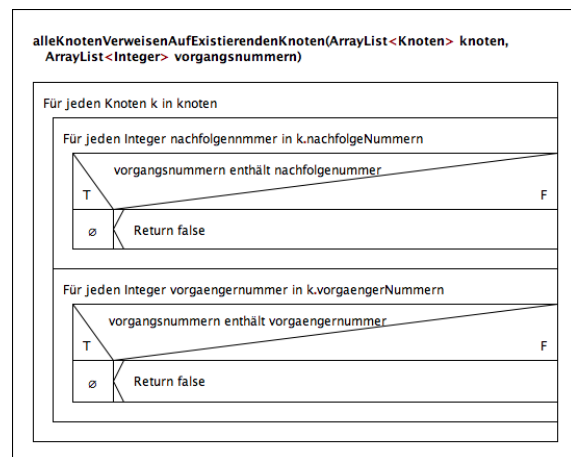
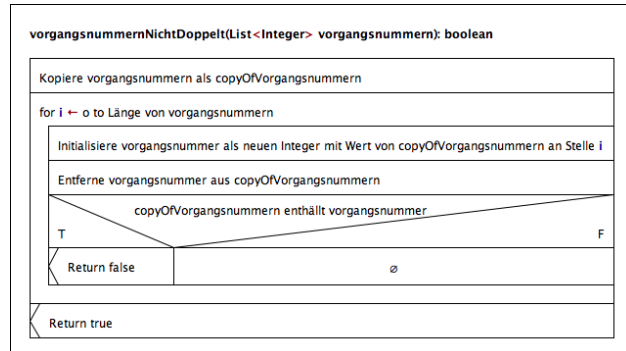
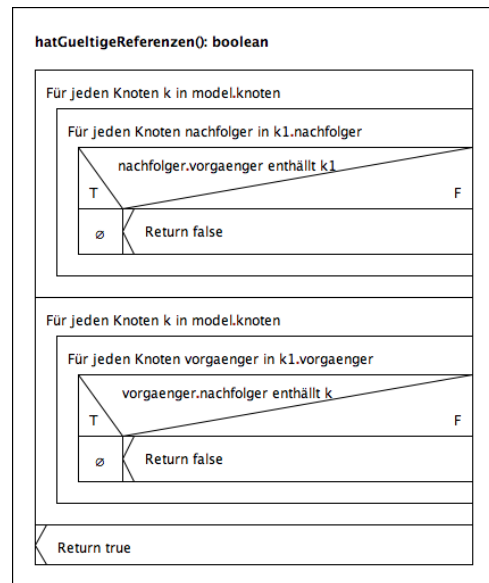
4.3.1 Main



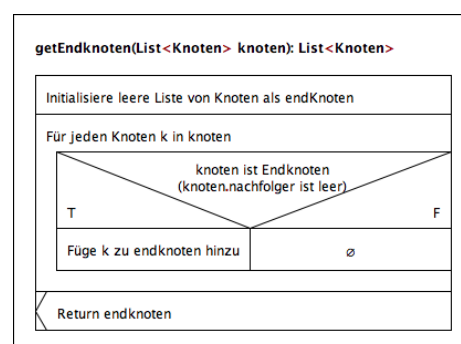
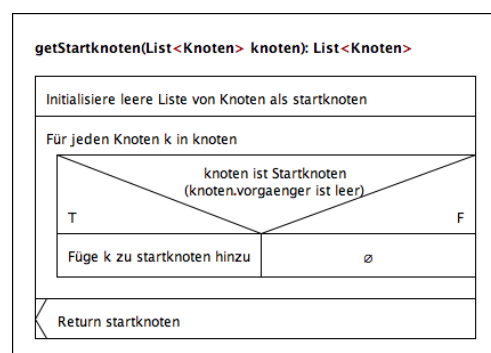
4.3.2 EinlesenAusDatei

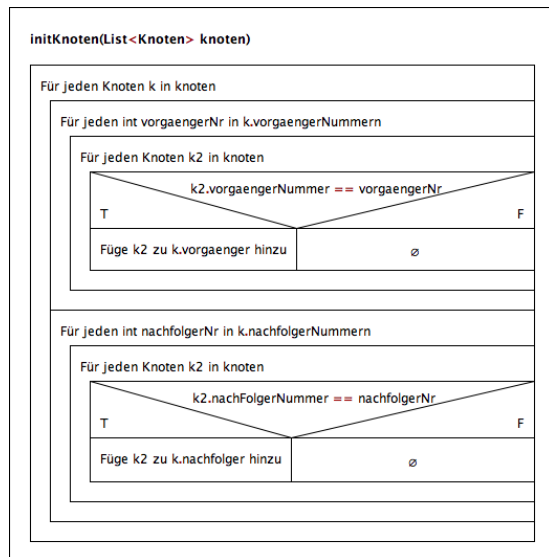


4 Programmkonzeption

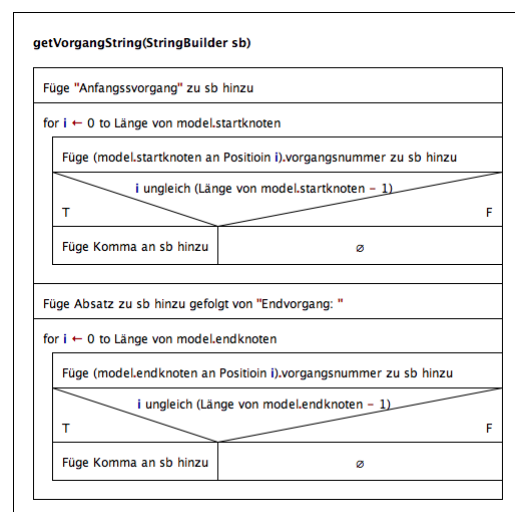
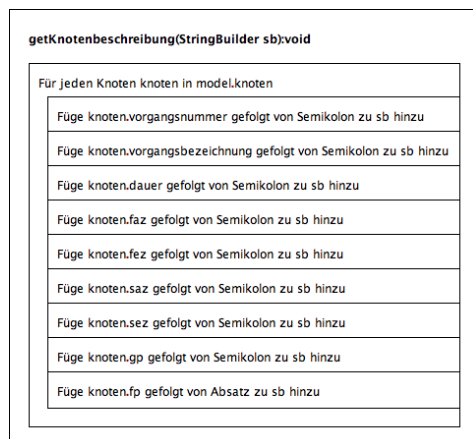
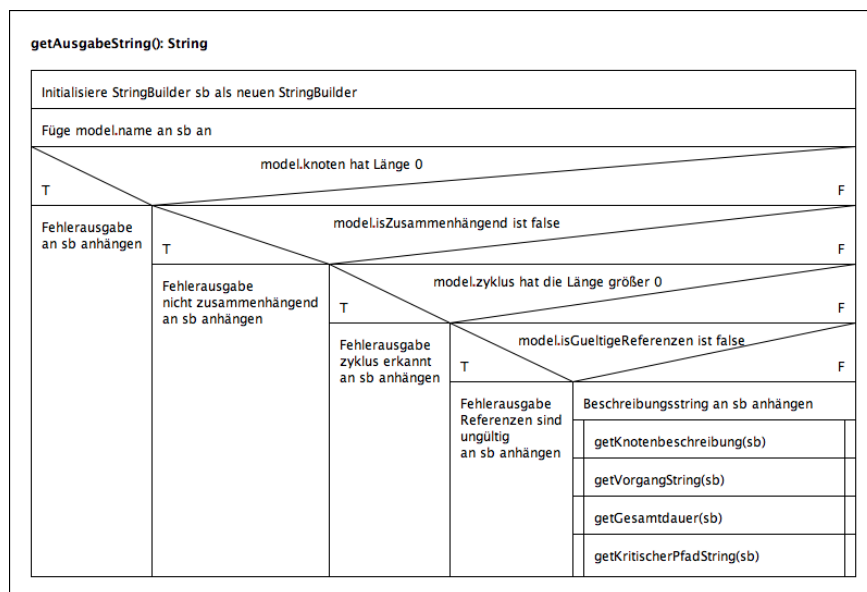


4.3.3 Model - Erzeugung des Models

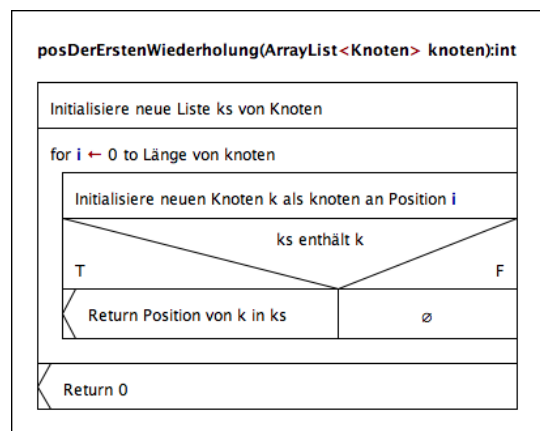
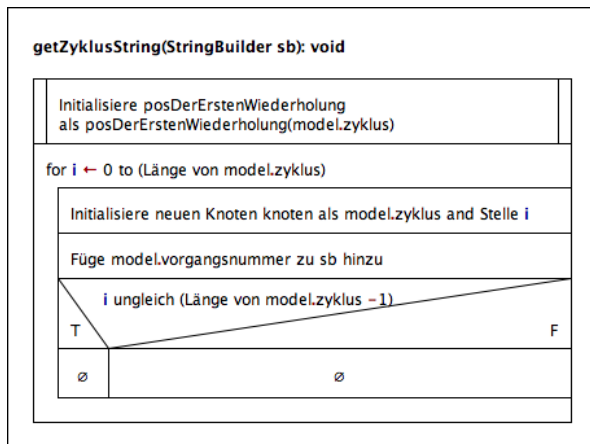
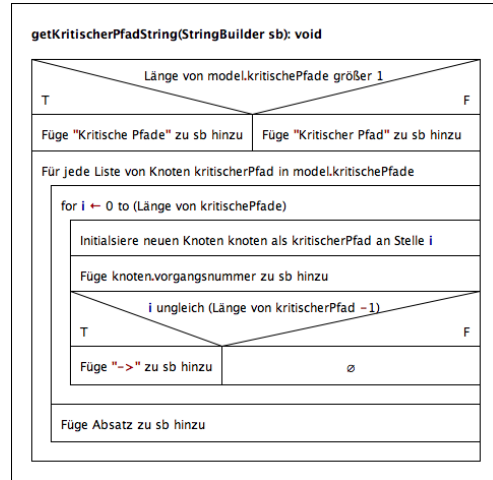
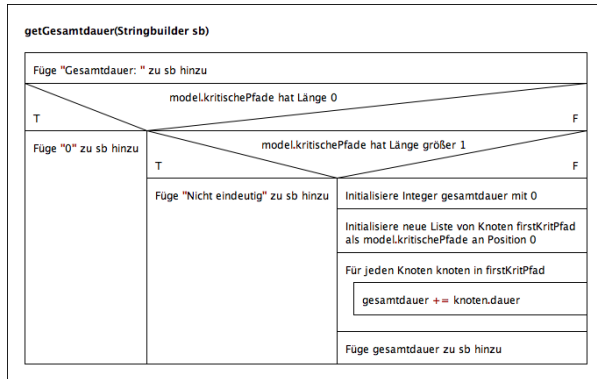




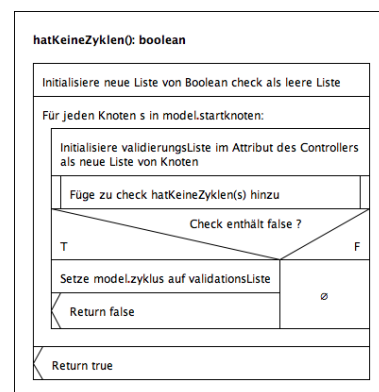
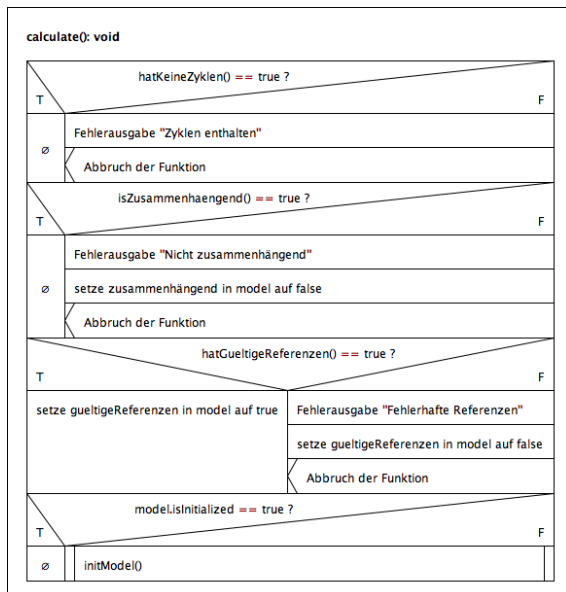
4.3.4 Ausgabe

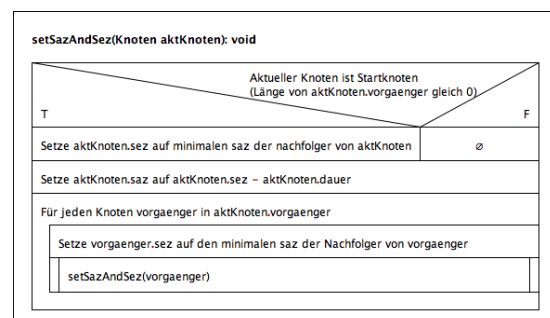
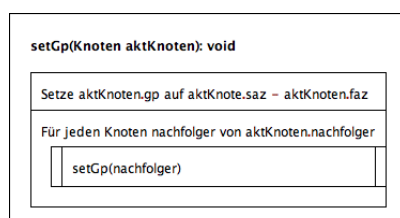
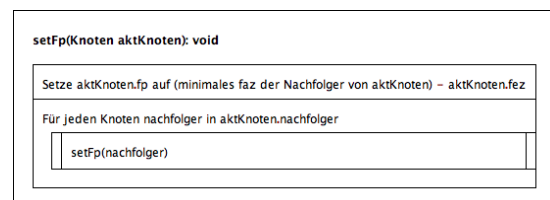
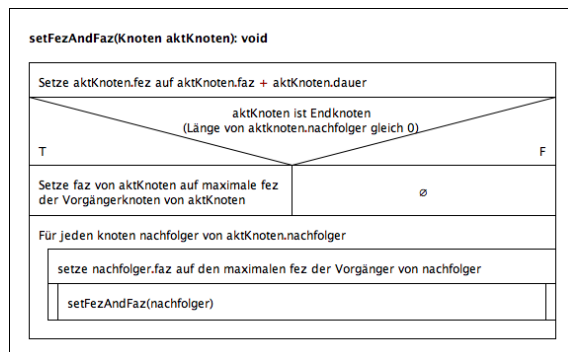
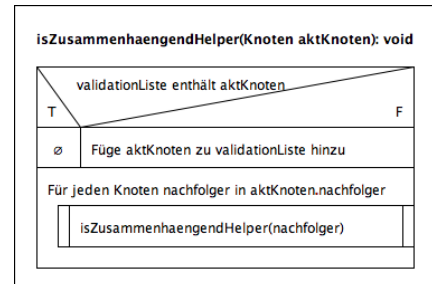
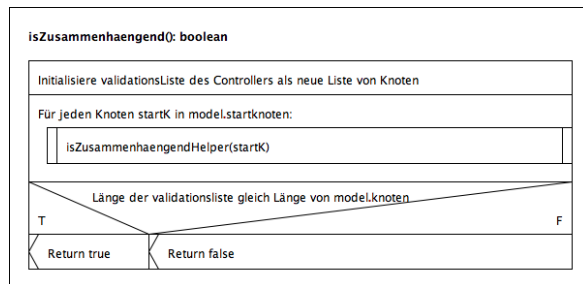
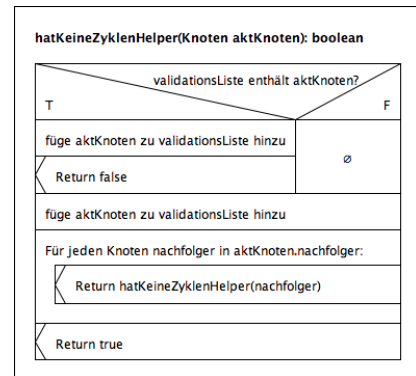
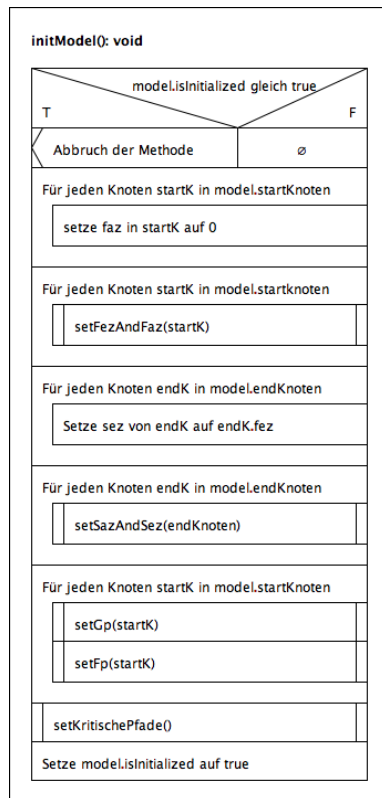


4 Programmkonzeption

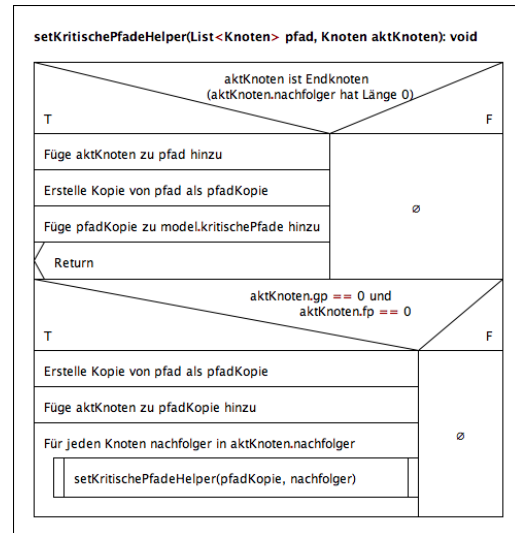
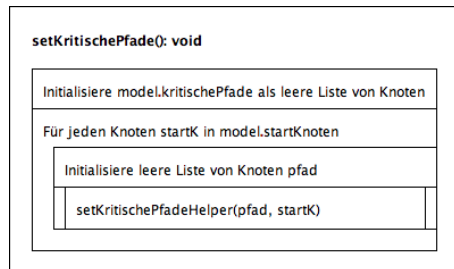


4.3.5 Controllermethoden





4 Programmkonzeption



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 zyklfrei ist. Eine weitere Hilfsmethode `hatKeineZyklenHelper(Knoten):boolean` ermöglicht die Überprüfung der Zyklfreiheit mittels Backtracking.

5 Abweichung von der handschriftlichen Ausarbeitung

- `istZusammenhaengend():boolean` prüft, ob ein Graph zusammenhängend ist. Hier ermöglicht ebenfalls eine Helper-Methode namens `istZusammenhaengend-Helper(Knoten):boolean` die Überprüfung mittels Backtracking.
- `hatGuelteReferenzen():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:
 - o Die neue Methode `setFezAndFaz(Knoten):void` setzt FEZ und FAZ ausgehend von einem aktuellen Knoten für diesen und alle Nachfolger dieses Knotens.
 - o Die neue Methode `setSazAndSez(Knoten):void` setzt SAZ für den aktuell betrachteten Knoten sowie alle Vorgängerknoten, ausgehend vom aktuell betrachteten Knoten.
 - o Die neue Methode `getMaxFezOfVorgaenger(Knoten):int` berechnet den Maximalen FEZ aller Vorgänger eines Knoten.
 - o Die neue Methode `getMinSazOfNachfolger(Knoten):int` berechnet den minimalen SAZ der Nachfolgenden Knoten eines betrachteten Knoten.
 - o 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 Unit Tests

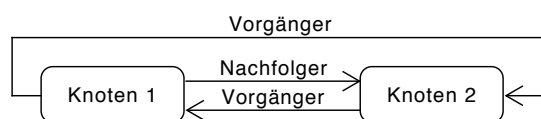
Zum Testen der korrekten Funktionalität des Controllers wurden grundlegende Unittests im Format JUnit 5 geschrieben. Sie werden angefertigt, um die funktionalen Einzelteile von Methoden eines Programms zu testen. Unittests gehören zur Gruppe der White-Box-Tests, also zur Gruppe der Tests, die mit Kenntnissen über die innere Funktionsweise des zu testenden Systems ablaufen.

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

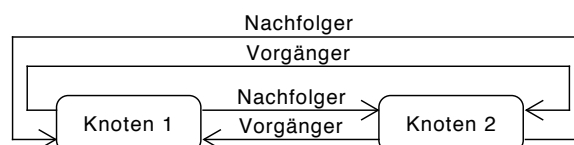
6.1 Prüfung der Methode `hatKeineZyklen()`

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

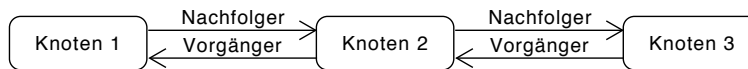
- `hatKeineZyklen_ModelOhneZyklen_RueckgabeTrue()`
 - Prüft, ob ein Graph aus zwei Knoten ohne Zyklen als zyklensfrei ausgegeben wird.
- `hatKeineZyklen_ZweiterKnotenHatErstenKnotenAlsNachfolger_RueckgabeFalse()`
 - Prüft, ob ein Graph mit zwei Knoten, bei dem der zweite Knoten den ersten als Nachfolger hat, als zykelbehaftet ausgegeben wird.
- `hatKeineZyklen_ErsterKnotenHatZweitenKnotenAlsVorgaengerSowieNachfolgerUndZweiterKnotenHatErstenKnotenAlsVorgaenger_RueckgabeTrueDaKeinExistierenderStartpunkt()`
 - Prüft, ob ein Model mit einem Graphen, welcher keinen Startpunkt hat, als zyklensfrei 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, die keinen Zykel hat, als zyklensfrei erkannt wird.



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 Blackbox-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 von der IHK verbesserten 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 trotzdem zu einem korrekten Ergebnis kommt. Die Nicht-Erfüllung der 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.

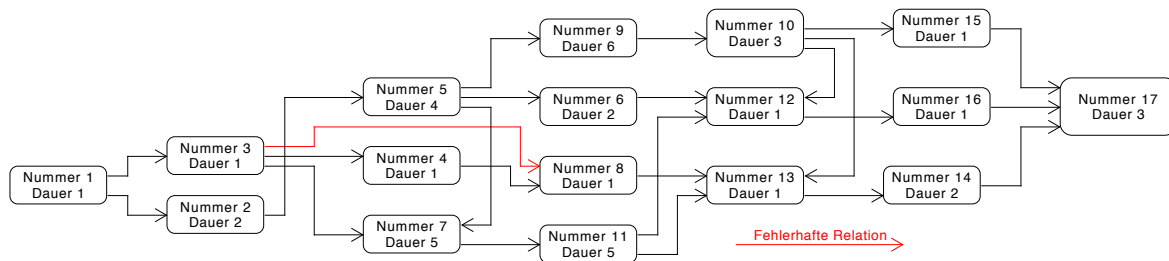
7.1 Besonderheiten der Beispiele 2, 3 und 5 der durch die IHK verbesserten Aufgabenstellung

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 Nachfolgersnummer 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; -
```

7 Blackbox- Testfälle

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

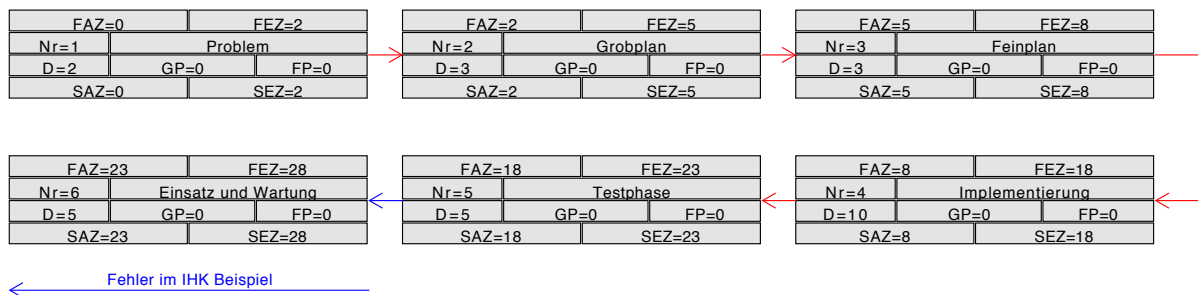
Gesamtdauer: 28

Kritischer Pfad

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

Grafische Darstellung 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 zyklfrei 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

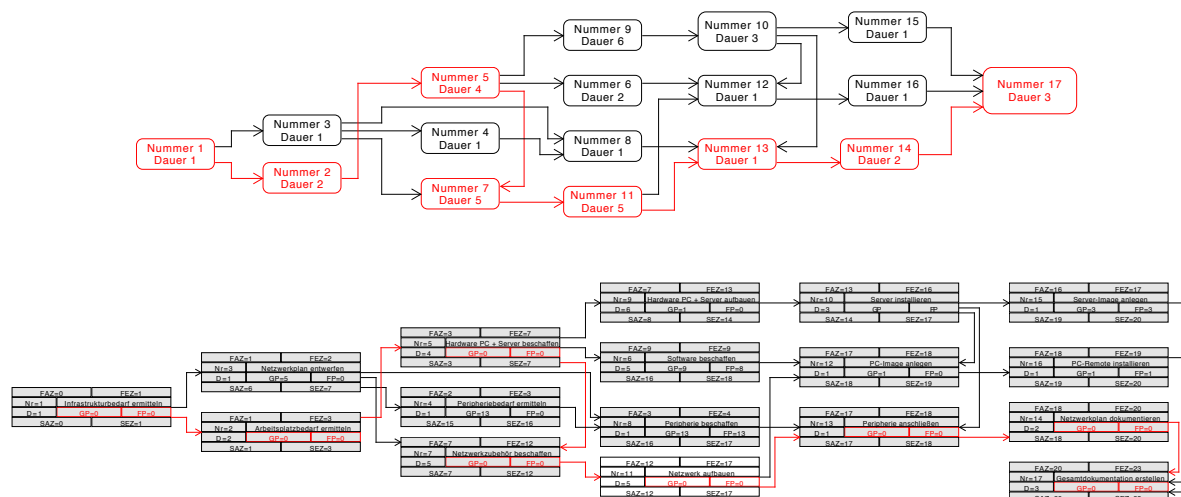
Endvorgang: 17

Gesamtdauer: 23

Kritischer Pfad

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

Grafische Darstellungen des Testfalls



7 Blackbox- Testfälle

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 ohne längere Berechnungszeiten durch den Backtracking-Algorithmus zu befürchten, der schließlich alle möglichen Wege berechnet.

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 Fälle 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

Es wird eine Datei eingelesen, die keine gültige Überschrift enthält.

Dies stellt einen Sonderfall dar, da die Datei korrekt eingelesen wird, jedoch die Eingabeanforderungen nicht erfüllt werden.

Die Eingabe wird erfolgreich eingelesen und korrekt ausgewertet und ausgegeben.

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 Graphen

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 und 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. Der Fehlerfall entspricht dem fehlerhaften Beispiel „Wasserfallmodell“ der Aufgabenstellung. 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 folgender Fehler 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; -
```

7 Blackbox- Testfälle

Ausgabe

Berechnung nicht möglich.

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

Diskussion

Der Fehlerfall hat eine besondere Bedeutung, da im Falle doppelter Indizes keine korrekten Referenzen erzeugt werden können. Die Referenzen werden über die Vorgangsnummern als Indizes bei der Referenzenerzeugung für Vorgänger und Nachfolger verwendet werden.

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 statt einer Zahl als Dauer eingegeben. Es wird folgender Fehler auf der Konsole ausgegeben:

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

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)

7 Blackbox- Testfälle

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

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

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

8.1 Zusammenfassung

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 Unittests (JUnit 5) wurden Whitebox-Tests durchgeführt, um wichtige Funktionen des Controllers zu überprüfen.

Mithilfe von Blackbox-Tests wurden die relevanten Normal-, Sonder- und Fehlerfälle systematisch und ausgiebig getestet. Die Tests liefen alle so ab, wie es nach der Aufgabenstellung zu erwarten ist.

8.2 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). In einer grafischen Darstellung können parallele Prozesse sehr gut dargestellt werden, da eine reine Konsolenausgabe sehr unübersichtlich ist und schnell zu Fehlinterpretationen führen kann. Eine grafische Darstellung kann ebenfalls dienlich sein, um falsche Eingaben mit fehlenden Referenzen zu vermeiden, da eine Eingabe über Text sehr schnell zu Fehlern führen kann.

Das Programm könnte Teil einer Prozessoptimierungssoftware werden, um allgemeine Prozesse mithilfe von kritischen Pfaden zu optimieren und so Schwachstellen zu finden. Das Programm kann ebenfalls in einer Prozessoptimierungssoftware Zyklen finden und so verhindern, dass es Endlosschleifen gibt und so Prozesse nie zu einem Ende kommen. Ebenso ermöglicht der Algorithmus dieses Programms das Aufspüren von nicht zusammenhängenden Knoten in einem Graphen, was ebenfalls in einer Prozessoptimierungssoftware hilfreich sein kann.

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	51
9.4	Package model	55
9.4.1	Klasse Knoten	55
9.4.2	Klasse Model	57

9.1 Package main

9.1.1 Klasse Main

```
1 package main;
2
3 import java.io.File;
4 import java.io.IOException;
5
6 import controller.Controller;
7 import io.AusgabeInDatei;
8 import io.LeseAusDatei;
9 import model.Model;
10
11 /**
12  * @author M. Leonard Haufs Prüfungsnummer: 101-20540
13  *
14  */
15 public class Main {
16     public static void main(String[] args) {
17         String dateiendung;
18         String verzeichnis;
19
20         // Parameterübergabe prüfen
21         if (args.length != 2) {
22             // keine korrekte Parameterübergabe
23             System.out.println(
24                 "Es müssen 2 Parameter übergeben werden. Paramter 1: Endung der Eingabedateien
25                 (z.B.: .in)\nParameter 2: Verzeichnis aus dem die Eingabedateien gelesen
26                 werden soll.");
27             return;
28         }
29         dateiendung = args[0];
30         verzeichnis = args[1];
31
32         File f;
33         try {
34             try {
35                 f = new File(verzeichnis);
36             } catch (Exception ex) {
37                 throw new IOException("Der Angegebene Pfad existiert nicht");
38             }
39
40             if (f.isDirectory() && f.canRead()) {
41                 File[] dateien = f.listFiles();
42                 for (int i = 0; i < dateien.length; i++) {
43                     // Prüfe ob die Datei gelesen werden kann
44                     if (dateien[i].isFile() && dateien[i].canRead()) {
45                         String tempEndung =
46                             dateien[i].getName().substring(dateien[i].getName().lastIndexOf("."),
47                                 dateien[i].getName().length());
48                         // wenn die Dateiendung der gewählten entspricht
49                         // wird die Datei eingelesen
50                         if (dateiendung.equals(tempEndung)) {
51
52                             // Eingabe
53
54                             LeseAusDatei in = new LeseAusDatei();
55                             Model model = in.getModelAusDatei(dateien[i]);
56
57                             // Berechnung
58                             Controller c = new Controller(model);
59                             c.calculate();
60
61                             // Ausgabe
62                             AusgabeInDatei out = new AusgabeInDatei(model);
63
64                             String outputPath = verzeichnis + "/" +
65                                 (dateien[i].getName().replace(dateiendung, ".out"));
66                             out.schreibeModelInDatei(outputPath);
67
68                             // OutputConsole out = new OutputConsole();
69                             // out.printEntireOutputString(model);
70
71                         }
72                     }
73                 }
74                 System.out.println(args[1] + ": Vorgang abgeschlossen.");
75             } else {
76                 throw new IOException("Der Angegebene Pfad ist kein Ordner oder kann nicht geöffnet
77                 werden.");
78             }
79         } catch (IOException ex) {
80             System.out.println(ex.getMessage());
81         }
82     }
83 }
```

9.2 Package io

9.2.1 Klasse LeseAusDatei

```
1 package io;
2
3 import java.io.BufferedReader;
4 import java.io.File;
5 import java.io.FileInputStream;
6 import java.io.FileNotFoundException;
7 import java.io.IOException;
8 import java.io.InputStreamReader;
9 import java.util.ArrayList;
10
11 import model.Knoten;
12 import model.Model;
13
14 /**
15  * Ermöglicht das Einlesen der Daten eines Models aus einer Datei
16  *
17  * @author M. Leonard Haufs Prüfungsnummer: 101-20540
18  *
19  */
20 public class LeseAusDatei {
21
22     /**
23      * Liefert die Daten eines Models, die in einer Datei gespeichert sind.
24      *
25      * @param file
26      *      Datei, aus der gelesen werden soll.
27      * @return Model mit dem gekapselten Daten. Falls eine ungültige Eingabe
28      *      erfolgt, wird ein leeres Model zurückgegeben.
29      */
30     public Model getModelAusDatei(File file) {
31         ArrayList<Knoten> knoten = new ArrayList<>();
32         String kommentar = "Fehler beim Einlesen.";
33         ArrayList<Integer> vorgangsnummern = new ArrayList<>();
34         BufferedReader br;
35         try {
36             br = new BufferedReader(new InputStreamReader(new FileInputStream(file)));
37         } catch (FileNotFoundException ex) {
38             System.out
39                 .println("In Datei " + file.getName() + "Ungenügende Eingabe: Datei konnte
40                     nicht geöffnet werden");
41             return new Model();
42         }
43
44         try {
45             String aktZeile = "";
46             while ((aktZeile = br.readLine()) != null) {
47                 if (aktZeile.startsWith("//")) {
48                     // Zeile beginnt mit "//+ " ?
49                     if (aktZeile.startsWith("//+ ")) {
50                         if (aktZeile.length() > 4) {
51                             kommentar = aktZeile.substring(4, aktZeile.length());
52                         }
53                         continue;
54                     }
55                     continue;
56                 }
57
58                 String aktZeileOhneLeer = aktZeile.replace(" ", "");
59                 String[] zeileSplit = aktZeileOhneLeer.split(";");
60                 if (zeileSplit.length != 5) {
61                     System.out.println("In Datei " + file.getName()
62                         + ": Ungenügende Eingabe. Es müssen je Zeile genau 5 Argumente getrennt
63                         mit einem Semikolon übergeben werden: "
64                         + aktZeile);
65                     br.close();
66                     return new Model();
67                 }
68                 int nr = Integer.parseInt(zeileSplit[0]);
69                 vorgangsnummern.add(nr);
70                 String beschr = aktZeile.split("; ")[1];
71                 int dauer = Integer.parseInt(zeileSplit[2]);
72
73                 ArrayList<Integer> vorgaengerNummern = new ArrayList<>();
74                 if (!zeileSplit[3].equals("-")) {
75                     String[] vorgaengerNummernArr = zeileSplit[3].split(",");
76                     for (int i = 0; i < vorgaengerNummernArr.length; i++) {
77                         String string = vorgaengerNummernArr[i];
78                         int number = Integer.parseInt(string);
79                         vorgaengerNummern.add(number);
80                     }
81                 }
82
83                 ArrayList<Integer> nachfolgerNummern = new ArrayList<>();
84                 if (!zeileSplit[4].equals("-")) {
85                     String[] nachfolgerNummernArr = zeileSplit[4].split(",");
86                     for (int i = 0; i < nachfolgerNummernArr.length; i++) {
87                         String string = nachfolgerNummernArr[i];
88                         int number = Integer.parseInt(string);
89                         nachfolgerNummern.add(number);
90                     }
91                 }
92
93                 // Prüfe, ob vorgangsnummern nicht doppelt vorliegen
94                 if (!vorgangsnummernNichtDoppelt(vorgangsnummern)) {
95                     System.out.println("In Datei " + file.getName()
96                         + ": Ungenügende Eingabe: Es kommt mindestens eine Vorgangsnummer
97                         mehrfach vor.");
98                     br.close();
99                 }
100             }
101         }
102     }
103 }
```

```

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

```

9.2.2 Klasse Ausgabe

```
1 package io;
2
3 import java.util.ArrayList;
4
5 import model.Knoten;
6 import model.Model;
7
8 /**
9  * Ermöglicht zu einem Model die Ausgabe der kenngrößen und kritischen Pfade
10  * auszugeben
11  *
12  * @author M. Leonard Haufs Prüfungsnummer: 101-20540
13  *
14  */
15 public abstract class Ausgabe {
16     private Model model;
17
18     /**
19      * Konstruktor, der Ausgabe mit einem Model initialisiert
20      *
21      * @param model
22      *        model, welches die auszugebenen Daten enthält
23      */
24     public Ausgabe(Model model) {
25         super();
26         this.model = model;
27     }
28
29     /**
30      * Gibt den Ausgabestring zurück.
31      *
32      * Falls nicht zusammenhängend oder falls Zyklen enthalten sind, wird ein
33      * entsprechender Fehler ausgegeben.
34      *
35      * @return Ausgabestring
36      */
37     protected String getAusgabeString() {
38         StringBuilder sb = new StringBuilder();
39
40         if (this.model.getKnoten().size() == 0) {
41             sb.append("Berechnung nicht möglich.");
42             sb.append("\n");
43             sb.append("Bitte sehen Sie sich die Konsolenausgabe an, um weitere Informationen zu erhalten.");
44         } else if (this.model.getZyklus().size() != 0) {
45             sb.append(this.model.getName());
46             sb.append("\n");
47             sb.append("\n");
48             sb.append("Berechnung nicht möglich.");
49             sb.append("\n");
50             sb.append("Zyklus erkannt: ");
51             this.getZyklusString(sb);
52         } else if (!this.model.isZusammenhaengend()) {
53             sb.append(this.model.getName());
54             sb.append("\n");
55             sb.append("\n");
56             sb.append("Berechnung nicht möglich.");
57             sb.append("\n");
58             sb.append("Nicht zusammenhängend.");
59         } else if (!this.model.isGueltigeReferenzen()) {
60             sb.append(this.model.getName());
61             sb.append("\n");
62             sb.append("\n");
63             sb.append("Berechnung nicht möglich.");
64             sb.append("\n");
65             sb.append("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.");
66
67         } else {
68             sb.append("Vorgangsnummer; Vorgangsbeschreibung; D; FAZ; FEZ; SAZ; SEZ; GP; FP");
69             sb.append("\n");
70             this.getKnotenbeschreibung(sb);
71             sb.append("\n");
72             this.getVorgangString(sb);
73             sb.append("\n");
74             this.getGesamtdauer(sb);
75             sb.append("\n");
76             sb.append("\n");
77             this.getKritischerPfadString(sb);
78         }
79
80         return sb.toString();
81     }
82
83     /**
84      * Gibt die Beschreibung eines Knotens im Netzplan. Dabei wird der übergebene
85      * StringBuilder verändert.
86      *
87      * @param sb
88      *        Stringbuilder, an den die Beschreibung angehängt werden soll
89      */
90     private void getKnotenbeschreibung(StringBuilder sb) {
91         for (Knoten knoten : model.getKnoten()) {
92             sb.append(knoten.getVorgangsnummer());
93             sb.append("; ");
94             sb.append(knoten.getVorgangsbezeichnung());
95             sb.append("; ");
96             sb.append(knoten.getDauer());
97             sb.append("; ");
98             sb.append(knoten.getFaz());
99         }
100     }
101 }
```

```

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.getEp());
109         sb.append("\n");
110     }
111 }
112
113 /**
114  * Gibt die Beschreibung von Anfangs- und Endvorgang zurück
115  *
116  * @param sb
117  *         StringBuilder, an den die Beschreibung von Anfangs- und Endvorgang
118  *         angehängt werden soll
119  */
120 private void getVorgangString(StringBuilder sb) {
121     sb.append("Anfangsvorgang: ");
122     for (int i = 0; i < model.getStartknoten().size(); i++) {
123         Knoten startK = model.getStartknoten().get(i);
124
125         sb.append(startK.getVorgangsnummer());
126         if (i != model.getStartknoten().size() - 1) {
127             sb.append(",");
128         }
129     }
130     sb.append("\n");
131     sb.append("Endvorgang: ");
132     for (int i = 0; i < model.getEndknoten().size(); i++) {
133         Knoten endK = model.getEndknoten().get(i);
134
135         sb.append(endK.getVorgangsnummer());
136         if (i != model.getEndknoten().size() - 1) {
137             sb.append(",");
138         }
139     }
140 }
141
142 /**
143  * * Gibt die Gesamtdauer des kritischen Pfades zurück. Sind mehrere Kritische
144  * * Pfade enthalten, so wird "Nicht eindeutig" zurückgegeben
145  *
146  * @param sb
147  *         StringBuilder, an den der Gesamtdauerstring angehängt werden soll.
148  */
149 private void getGesamtdauer(StringBuilder sb) {
150     sb.append("Gesamtdauer: ");
151     if (this.model.getKritischePfade().size() == 0) {
152         sb.append(0);
153     } else if (this.model.getKritischePfade().size() > 1) {
154         sb.append("Nicht eindeutig");
155     } else {
156         int gesamtdauer = 0;
157         ArrayList<Knoten> firstKritPfad = this.model.getKritischePfade().get(0);
158         for (Knoten knoten : firstKritPfad) {
159             gesamtdauer += knoten.getDauer();
160         }
161         sb.append(gesamtdauer);
162     }
163 }
164
165 /**
166  * Hängt die String- Repräsentation des/der Kritischen Pfade(s) an einen
167  * übergebenen StringBuilder an
168  *
169  * @param sb
170  *         StringBuilder, an den die String- Repräsentation des/der
171  *         Kritischen Pfade(s) angehängt werden soll
172  */
173 private void getKritischerPfadString(StringBuilder sb) {
174     if (this.model.getKritischePfade().size() > 1) {
175         sb.append("Kritische Pfade");
176     } else {
177         sb.append("Kritischer Pfad");
178     }
179     sb.append("\n");
180
181     for (ArrayList<Knoten> kritischerPfad : this.model.getKritischePfade()) {
182         for (int i = 0; i < kritischerPfad.size(); i++) {
183             Knoten knoten = kritischerPfad.get(i);
184             sb.append(knoten.getVorgangsnummer());
185             if (i != kritischerPfad.size() - 1) {
186                 sb.append("→");
187             }
188         }
189         sb.append("\n");
190     }
191 }
192
193 /**
194  * Hängt die String- Repräsentation eines Zyklus an einen übergebenen
195  * Stringbuilder an
196  *
197  * @param sb
198  *         StringBuilder, an den die String- Repräsentation des/der Zyklus
199  *         angehängt werden soll
200  */
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 Prüfungsnummer: 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

```
1 package controller;
2
3 import java.util.ArrayList;
4
5 import model.Knoten;
6 import model.Model;
7
8 /**
9  * Hauptberechnungsklasse.
10  *
11  * @author M. Leonard Haufs Prüfungsnummer: 101-20540
12  *
13  */
14 public class Controller {
15     private Model model;
16     private ArrayList<Knoten> validationsListe;
17
18     // Konstruktor
19     public Controller(Model model) {
20         super();
21         this.model = model;
22     }
23
24     /**
25      * Hauptberechnungsmethode des Controllers.
26      *
27      * Falls noch nicht initialisiert wurde, wird auf Zyklen und Zusammenhängigkeit
28      * geprüft. Falls der Netzplan Zyklen enthält, wird im Model in zyklus ein
29      * zyklus gespeichert. Wenn der Netzplan nicht nicht zusammenhängend ist, wird
30      * im Model isZusammenhaengend auf false gesetzt. Sonst auf true.
31      *
32      * Anschließend wird das Model initialisiert, also die kenngrößen berechnet und
33      * anschließend der kritische Pfad, falls er existiert, berechnet
34      */
35     public void calculate() {
36         // Prüfe, ob der im Model gekapselte Netzplan keine Zyklen enthält
37         boolean hatKeineZyklen = this.hatKeineZyklen();
38         if (!hatKeineZyklen) {
39             System.out.println(this.model.getName() + ": Zyklen enthalten");
40             return;
41         }
42
43         // Prüfe, ob der im Model gekapselte Netzplan zusammenhängend ist
44         boolean isZusammenhaengend = this.isZusammenhaengend();
45         if (!isZusammenhaengend) {
46             System.out.println(this.model.getName() + ": Fehler (Nicht zusammenhängend)");
47             model.setZusammenhaengend(false);
48             return;
49         } else {
50             model.setZusammenhaengend(true);
51         }
52
53         // Prüft, ob alle Referenzen in model.knoten korrekt sind, also ob jeder
54         // Nachfolger auch in dessen Vorgaengern enthalten ist bzw. umgekehrt.
55         boolean hatGueltigeReferenzen = this.hatGueltigeReferenzen();
56         if (!hatGueltigeReferenzen) {
57             System.out.println(this.model.getName()
58                 + "Referenzen der Eingabe sind ungenügend. nicht jeder Nachfolger auch in
59                 dessen Vorgaengern enthalten bzw. umgekehrt ist.");
60             model.setGueltigeReferenzen(false);
61         } else {
62             model.setGueltigeReferenzen(true);
63         }
64
65         // Initialisiere das Model
66         if (!this.model.isInitialized()) {
67             initModel();
68         }
69     }
70
71     /**
72      * Prüft, ob der im Model gekapselte Netzplan keine Zyklen enthält
73      *
74      * @return true, falls der Netzplan im Model keine Zyklen enthält, sonst true
75      */
76     boolean hatKeineZyklen() {
77         ArrayList<Boolean> check = new ArrayList<>();
78
79         // Rufe für ausgehend von allen Startknoten die Helpermethode
80         // hatKeineZyklenHelper auf. Falls ein Ergebnis negativ ausfällt wird false
81         // zurückgegeben
82
83         for (Knoten s : this.model.getStartknoten()) {
84             this.validationsListe = new ArrayList<>();
85             check.add(hatKeineZyklenHelper(s));
86             if (check.contains(Boolean.valueOf(false))) {
87                 model.setZyklus(this.validationsListe);
88                 return false;
89             }
90         }
91         return true;
92     }
93
94     /**
95      * Hilfsfunktion zur Überprüfung, ob keine Zyklen existieren
96      */
97 }
```

```

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 enthalten 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 zusammenhängend 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 enthält, 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: Vorwärtsrechnung Bei gegebenem Anfangstermin werden aufgrund der
158      * angegebenen Dauer eines Vorganges die frühestmöglichen Anfangs- und
159      * Endzeiten eingetragen. Weiterhin lässt sich die Gesamtdauer eines Projekts
160      * bestimmen.
161      *
162      * 2. Phase: Rückwärtsrechnung: Bei der Rückwärtsrechnung wird ermittelt,
163      * wann die einzelnen Vorgänge spätestens begonnen und fertiggestellt sein
164      * müssen, damit die Gesamtprojektzeit nicht gefährdet ist.
165      *
166      *
167      * 3. Phase: Ermittlung der Zeitreserven und des kritischen Pfades: In dieser
168      * Phase wird ermittelt, welche Zeitreserven existieren und welche Vorgänge
169      * besonders problematisch sind (kritischer Vorgang), weil es bei diesen keine
170      * Zeitreserven gibt. Dazu wird für alle Knoten der Gesamtpuffer (GP)
171      * berechnet, sowie der freie Puffer (FP).
172      */
173     private void initModel() {
174         // Prüfe, ob das Model bereits initialisiert wurde
175         if (this.model.isInitialized()) {
176             return;
177         }
178
179         //
180         * 1. Phase: Vorwärtsrechnung
181         *
182         * Setze FAZ der Startknoten
183         */
184         for (Knoten startK : this.model.getStartknoten()) {
185             // Der Startknoten hat als FAZ immer den Wert 0
186             startK.setFaz(0);
187         }
188
189         // Setze FEZ aller Knoten als FEZ = FAZ + Dauer
190         for (Knoten startK : this.model.getStartknoten()) {
191             // startK.setFez(startK.getFaz() + startK.getDauer());
192             this.setFezAndFaz(startK);
193         }
194
195         //
196         * 2. Phase: Rückwärtsrechnung
197         *
198         * Bei der Rückwärtsrechnung wird ermittelt, wann die einzelnen Vorgänge
199         * spätestens begonnen und fertiggestellt sein müssen, damit die

```

```

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

```

```

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

```

```

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

```

9.3.2 Unittest Klasse Controller

```
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 /**
13  * Unittest der Klasse Controllers
14  *
15  * @author M. Leonard Haufs Prüfungsnummer: 101-20540
16  *
17  */
18 public class ControllerTest {
19     @Test
20     public void hatKeineZyklen_ModelOhneZyklen_RueckgabeTrue() {
21         // Arrangieren
22         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
23         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
24         ersterKnotenNachfolger.add(2);
25         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, new ArrayList<>(),
26             ersterKnotenNachfolger);
27         knotenliste.add(ersterKnoten);
28         //
29         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
30         zweiterKnotenVorgaenger.add(1);
31         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger, new
32             ArrayList<>());
33         knotenliste.add(zweiterKnoten);
34         Model model = new Model(knotenliste, "Testliste");
35         //
36         Controller controller = new Controller(model);
37
38         // Ausführen
39         boolean keineZyklen = controller.hatKeineZyklen();
40
41         // Auswerten
42         assertEquals(true, keineZyklen);
43     }
44
45     @Test
46     public void hatKeineZyklen_ZweiterKnotenHatErstenKnotenAlsNachfolger_RueckgabeFalse() {
47         // Arrangieren
48         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
49         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
50         ersterKnotenNachfolger.add(2);
51         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, new ArrayList<>(),
52             ersterKnotenNachfolger);
53         knotenliste.add(ersterKnoten);
54         //
55         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
56         zweiterKnotenVorgaenger.add(1);
57         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
58         zweiterKnotenNachfolger.add(1);
59         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
60             zweiterKnotenNachfolger);
61         knotenliste.add(zweiterKnoten);
62         Model model = new Model(knotenliste, "Testliste");
63         //
64         Controller controller = new Controller(model);
65
66         // Ausführen
67         boolean keineZyklen = controller.hatKeineZyklen();
68
69         // Auswerten
70         assertEquals(false, keineZyklen);
71     }
72
73     @Test
74     public void hatKeineZyklen_ErsterKnotenHatZweitenKnotenAlsVorgaengerSowieNachfolgerUndZweiterKnotenHatErstenKnotenAlsVorgaengerUndNachfolger() {
75         // Arrangieren
76         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
77         //
78         ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
79         ersterKnotenVorgaenger.add(2);
80         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
81         ersterKnotenNachfolger.add(2);
82         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
83             ersterKnotenNachfolger);
84         knotenliste.add(ersterKnoten);
85         //
86         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
87         zweiterKnotenVorgaenger.add(1);
88         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
89         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
90             zweiterKnotenNachfolger);
91         knotenliste.add(zweiterKnoten);
92         //
93         Model model = new Model(knotenliste, "Testliste");
94         //
95         Controller controller = new Controller(model);
96
97         // Ausführen
98         boolean keineZyklen = controller.hatKeineZyklen();
99     }
100 }
```

```

94         // Auswerten
95         assertEquals(true, keineZyklen);
96     }
97
98     @Test
99     public void
100         hatKeineZyklen_ZweiKnotenHabenSichGegenseitigAlsNachfolgerSowieVorgaenger_RueckgabeTrueDaKeinExistierenderSt
101     {
102         // Arrangieren
103         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
104         //
105         ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
106         ersterKnotenVorgaenger.add(2);
107         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
108         ersterKnotenNachfolger.add(2);
109         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
110             ersterKnotenNachfolger);
111         knotenliste.add(ersterKnoten);
112         //
113         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
114         zweiterKnotenVorgaenger.add(1);
115         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
116         zweiterKnotenNachfolger.add(1);
117         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
118             zweiterKnotenNachfolger);
119         knotenliste.add(zweiterKnoten);
120         //
121         Model model = new Model(knotenliste, "Testliste");
122         //
123         Controller controller = new Controller(model);
124
125         // Ausführen
126         boolean keineZyklen = controller.hatKeineZyklen();
127
128         // Auswerten
129         assertEquals(true, keineZyklen);
130     }
131
132     @Test
133     public void hatKeineZyklen_DritterKnotenHatZweitenKnotenAlsNachfolger_RueckgabeFalse() {
134         // Arrangieren
135         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
136         //
137         ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
138         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
139         ersterKnotenNachfolger.add(2);
140         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
141             ersterKnotenNachfolger);
142         knotenliste.add(ersterKnoten);
143         //
144         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
145         zweiterKnotenVorgaenger.add(1);
146         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
147         zweiterKnotenNachfolger.add(3);
148         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
149             zweiterKnotenNachfolger);
150         knotenliste.add(zweiterKnoten);
151         //
152         ArrayList<Integer> dritterKnotenVorgaenger = new ArrayList<Integer>();
153         dritterKnotenVorgaenger.add(2);
154         ArrayList<Integer> dritterKnotenNachfolger = new ArrayList<Integer>();
155         dritterKnotenNachfolger.add(2);
156         Knoten dritterKnoten = new Knoten(3, "Dritter Schritt", 10, dritterKnotenVorgaenger,
157             dritterKnotenNachfolger);
158         knotenliste.add(dritterKnoten);
159         //
160         Model model = new Model(knotenliste, "Testliste");
161         //
162         Controller controller = new Controller(model);
163
164         // Ausführen
165         boolean keineZyklen = controller.hatKeineZyklen();
166
167         // Auswerten
168         assertEquals(false, keineZyklen);
169     }
170
171     @Test
172     public void isZusammenhaengend_ZusammenhaengendeKnoten_RueckgabeTrue() {
173         // Arrangieren
174         ArrayList<Knoten> knotenliste = new ArrayList<Knoten>();
175         //
176         ArrayList<Integer> ersterKnotenVorgaenger = new ArrayList<Integer>();
177         ArrayList<Integer> ersterKnotenNachfolger = new ArrayList<Integer>();
178         ersterKnotenNachfolger.add(2);
179         Knoten ersterKnoten = new Knoten(1, "Erster Schritt", 10, ersterKnotenVorgaenger,
180             ersterKnotenNachfolger);
181         knotenliste.add(ersterKnoten);
182         //
183         ArrayList<Integer> zweiterKnotenVorgaenger = new ArrayList<Integer>();
184         zweiterKnotenVorgaenger.add(1);
185         ArrayList<Integer> zweiterKnotenNachfolger = new ArrayList<Integer>();
186         zweiterKnotenNachfolger.add(3);
187         Knoten zweiterKnoten = new Knoten(2, "Zweiter Schritt", 10, zweiterKnotenVorgaenger,
188             zweiterKnotenNachfolger);
189         knotenliste.add(zweiterKnoten);
190         //
191         ArrayList<Integer> dritterKnotenVorgaenger = new ArrayList<Integer>();
192         dritterKnotenVorgaenger.add(2);
193         ArrayList<Integer> dritterKnotenNachfolger = new ArrayList<Integer>();
194         dritterKnotenNachfolger.add(2);
195         Knoten dritterKnoten = new Knoten(3, "Dritter Schritt", 10, dritterKnotenVorgaenger,
196             dritterKnotenNachfolger);
197         knotenliste.add(dritterKnoten);

```



```

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

```

```

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

```

9.4 Package model

9.4.1 Klasse Knoten

```
1 package model;
2
3 import java.util.ArrayList;
4
5 /**
6  *
7  * @author M. Leonard Haufs Prüfungsnummer: 101-20540
8  *
9  */
10 public class Knoten {
11
12     private int vorgangsnummer;
13     private String vorgangsbezeichnung;
14
15     private int faz;
16     private int fez;
17     private int dauer;
18     private int gp;
19     private int fp;
20     private int saz;
21     private int sez;
22
23     ArrayList<Knoten> vorgaenger;
24     ArrayList<Integer> vorgaengerNummern;
25
26     ArrayList<Knoten> nachfolger;
27     ArrayList<Integer> nachfolgerNummern;
28
29     // Getter und Setter
30     public int getVorgangsnummer() {
31         return vorgangsnummer;
32     }
33     public void setVorgangsnummer(int vorgangsnummer) {
34         this.vorgangsnummer = vorgangsnummer;
35     }
36     public String getVorgangsbezeichnung() {
37         return vorgangsbezeichnung;
38     }
39     public void setVorgangsbezeichnung(String vorgangsbezeichnung) {
40         this.vorgangsbezeichnung = vorgangsbezeichnung;
41     }
42     public int getFaz() {
43         return faz;
44     }
45     public void setFaz(int faz) {
46         this.faz = faz;
47     }
48     public int getFez() {
49         return fez;
50     }
51     public void setFez(int fez) {
52         this.fez = fez;
53     }
54     public int getDauer() {
55         return dauer;
56     }
57     public void setDauer(int dauer) {
58         this.dauer = dauer;
59     }
60     public int getGp() {
61         return gp;
62     }
63     public void setGp(int gp) {
64         this.gp = gp;
65     }
66     public int getFp() {
67         return fp;
68     }
69     public void setFp(int fp) {
70         this.fp = fp;
71     }
72     public int getSaz() {
73         return saz;
74     }
75     public void setSaz(int saz) {
76         this.saz = saz;
77     }
78     public int getSez() {
79         return sez;
80     }
81     public void setSez(int sez) {
82         this.sez = sez;
83     }
84     public ArrayList<Knoten> getVorgaenger() {
85         return vorgaenger;
86     }
87     public void setVorgaenger(ArrayList<Knoten> vorgaenger) {
88         this.vorgaenger = vorgaenger;
89     }
90     public ArrayList<Integer> getVorgaengerNummern() {
91         return vorgaengerNummern;
92     }
93     public void setVorgaengerNummern(ArrayList<Integer> vorgaengerNummern) {
94         this.vorgaengerNummern = vorgaengerNummern;
95     }
96     public ArrayList<Knoten> getNachfolger() {
97         return nachfolger;
```

```

98     }
99     public void setNachfolger(ArrayList<Knoten> nachfolger) {
100         this.nachfolger = nachfolger;
101     }
102     public ArrayList<Integer> getNachfolgerNummern() {
103         return nachfolgerNummern;
104     }
105     public void setNachfolgerNummern(ArrayList<Integer> nachfolgerNummern) {
106         this.nachfolgerNummern = nachfolgerNummern;
107     }
108
109     // Konstruktor
110     public Knoten(int vorgangsnummer, String vorgangsbezeichnung, int dauer, ArrayList<Integer>
        vorgaengerNummern, ArrayList<Integer> nachfolgerNummern) {
111         super();
112
113         this.vorgangsnummer = vorgangsnummer;
114         this.vorgangsbezeichnung = vorgangsbezeichnung;
115         this.dauer = dauer;
116         this.vorgaengerNummern = vorgaengerNummern;
117         this.nachfolgerNummern = nachfolgerNummern;
118
119         this.vorgaenger = new ArrayList<>();
120         this.nachfolger = new ArrayList<>();
121     }
122 }

```

9.4.2 Klasse Model

```
1 package model;
2
3 import java.util.ArrayList;
4
5 /**
6  *
7  * @author M. Leonard Haufs Prüfungsnummer: 101-20540
8  *
9  */
10 public class Model {
11     private boolean initialized;
12
13     private ArrayList<Knoten> startknoten;
14     private ArrayList<Knoten> endknoten;
15
16     private ArrayList<Knoten> knoten;
17
18     private ArrayList<ArrayList<Knoten>> kritischePfade;
19     private ArrayList<Knoten> zyklus;
20     private boolean isZusammenhaengend;
21     private boolean gueltigeReferenzen;
22
23     private String name;
24
25     public boolean isInitialized() {
26         return initialized;
27     }
28
29     public void initialize() {
30         this.initialized = true;
31     }
32
33     public boolean isZusammenhaengend() {
34         return isZusammenhaengend;
35     }
36
37     public void setZusammenhaengend(boolean isZusammenhaengend) {
38         this.isZusammenhaengend = isZusammenhaengend;
39     }
40
41     public boolean isGueltigeReferenzen() {
42         return gueltigeReferenzen;
43     }
44
45     public void setGueltigeReferenzen(boolean gueltigeReferenzen) {
46         this.gueltigeReferenzen = gueltigeReferenzen;
47     }
48
49     // Getter und Setter
50     public ArrayList<ArrayList<Knoten>> getKritischePfade() {
51         return kritischePfade;
52     }
53
54     public void setKritischePfade(ArrayList<ArrayList<Knoten>> kritischePfade) {
55         this.kritischePfade = kritischePfade;
56     }
57
58     public ArrayList<Knoten> getZyklus() {
59         return zyklus;
60     }
61
62     public void setZyklus(ArrayList<Knoten> zyklus) {
63         this.zyklus = zyklus;
64     }
65
66     public ArrayList<Knoten> getStartknoten() {
67         return startknoten;
68     }
69
70     public ArrayList<Knoten> getEndknoten() {
71         return endknoten;
72     }
73
74     public ArrayList<Knoten> getKnoten() {
75         return knoten;
76     }
77
78     public String getName() {
79         return name;
80     }
81
82     // Konstruktoren
83
84     public Model() {
85         super();
86         this.knoten = new ArrayList<>();
87         this.name = "not set";
88
89         this.startknoten = new ArrayList<>();
90         this.endknoten = new ArrayList<>();
91
92         this.kritischePfade = new ArrayList<>();
93         this.zyklus = new ArrayList<>();
94         this.gueltigeReferenzen = true;
95     }
96
97     public Model(ArrayList<Knoten> knoten, String name) {
98         this();
99         this.knoten = knoten;
100         this.name = name;
101     }
```

```

102         this.initKnoten(knoten);
103         this.startknoten = this.getStartknoten(knoten);
104         this.endknoten = this.getEndknoten(knoten);
105     }
106
107     /**
108      * Bestimmt die Startknoten einer Liste von Knoten
109      *
110      * @param knoten
111      *         Liste von Knoten
112      * @return Startknoten einer Liste von Knoten
113      */
114     private ArrayList<Knoten> getStartknoten(ArrayList<Knoten> knoten) {
115         ArrayList<Knoten> startknoten = new ArrayList<>();
116         for (Knoten k : knoten) {
117             if (k.getVorgaengerNummern().size() == 0) {
118                 startknoten.add(k);
119             }
120         }
121
122         return startknoten;
123     }
124
125     /**
126      * Bestimmt die Endknoten einer Liste von Knoten
127      *
128      * @param knoten
129      *         Liste von Knoten
130      * @return Endknoten einer Liste von Knoten
131      */
132     private ArrayList<Knoten> getEndknoten(ArrayList<Knoten> knoten) {
133         ArrayList<Knoten> endknoten = new ArrayList<>();
134         for (Knoten k : knoten) {
135             if (k.getNachfolgerNummern().size() == 0) {
136                 endknoten.add(k);
137             }
138         }
139
140         return endknoten;
141     }
142
143     /**
144      * Initialisiert eine liste von Knoten, setzt also die Vorgänger und Nachfolger
145      * der Knoten
146      *
147      * @param knoten
148      *         Liste von Knoten
149      */
150     private void initKnoten(ArrayList<Knoten> knoten) {
151         for (Knoten k : knoten) {
152             for (int vorgaengerNr : k.getVorgaengerNummern()) {
153                 for (Knoten k2 : knoten) {
154                     if (k2.getVorgangsnummer() == vorgaengerNr) {
155                         k.getVorgaenger().add(k2);
156                     }
157                 }
158             }
159
160             for (int nachfolgerNr : k.getNachfolgerNummern()) {
161                 for (Knoten k2 : knoten) {
162                     if (k2.getVorgangsnummer() == nachfolgerNr) {
163                         k.getNachfolger().add(k2);
164                     }
165                 }
166             }
167         }
168     }
169 }
170 }

```