

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	3
2	Benutzeranleitung	4
2.1	Systemvoraussetzungen und Hinweise zum Aufruf	4
2.2	Installation des Programms	4
2.3	Programmstart	4
3	Aufgabenanalyse	6
3.1	Allgemeine Problemstellung	6
3.2	Format der Eingabedatei	6
3.3	Format der Ausgabedatei	7
3.4	Algorithmus	8
3.5	Verbale Beschreibung des Verfahrens	8
3.6	Einlesen der Eingabedatei	8
3.7	Überführung der Eingabedaten ins Datenmodell	9
3.8	Berechnung im Controller	9
4	Programmkonzeption	11
4.1	UML Klassendiagramm	11
4.2	Programmablauf im Sequenzdiagramm	12
4.3	Nassi-Shneiderman-Diagramme	12
4.3.1	Main	12
4.3.2	Einlesen einer Datei	13
4.3.3	Erzeugung des Models - Modelmethoden	14
4.3.4	Ausgabe	15
4.3.5	Controllermethoden	16
5	Abweichung von der handschriftlichen Ausarbeitung	19
5.1	Datenmodell	19
5.1.1	Die Sichtbarkeiten der Methoden	19
5.1.2	Klasse <code>Model</code>	19
5.1.3	Klasse <code>Knoten</code>	19
5.1.4	Klasse <code>Controller</code>	19

5.1.5	Klasse <code>LeseAusDatei</code> (Ursprünglich <code>InputFromFile</code>).....	20
5.1.6	Abstrakte Klasse <code>Ausgabe</code> (ursprünglich <code>Output</code>).....	20
5.1.7	Klasse <code>AusgabeInDatei</code> (ursprünglich <code>OutputToFile</code>).....	21
6	Unittests	22
7	Testfälle	23
7.1	Besonderheiten der Beispiele 2, 3 und 5 der durch die IHK verbesserten Aufgabenstellung	23
7.2	Normalfälle.....	24
7.2.1	Beispiele aus der durch die IHK verbesserten Aufgabenstellung	24
7.2.2	Eigene Normalfälle.....	25
7.3	Sonderfälle.....	29
7.3.1	Eigene Sonderfälle.....	29
7.4	Fehlerfälle	30
7.4.1	Beispiele der IHK	30
7.4.2	Eigene Fehlerfälle	31
8	Zusammenfassung und Ausblick	34
9	Anhang: Programmcode.....	35

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/PATH/FILE.jar ENDUNG VERZEICHNIS
```

Dabei steht der Platzhalter „ENDUNG“ für die Endung der Dateien, die im Verzeichnis „VERZEICHNIS“ durch das Programm eingelesen und verarbeitet werden.

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.

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 [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
...
```

Abbildung 1: Formatierung einer gültigen Eingabe

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->...
...
```

Abbildung 2: Struktur einer gültigen Ausgabe

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 „;“. 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:
- 2) Bei gegebenem Anfangstermin werden aufgrund der angegebenen Dauer eines Vorganges die frühestmöglichen Anfangs- und Endzeiten eingetragen. Weiterhin lässt sich die Gesamtdauer eines Projekts bestimmen.

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.

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.

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.

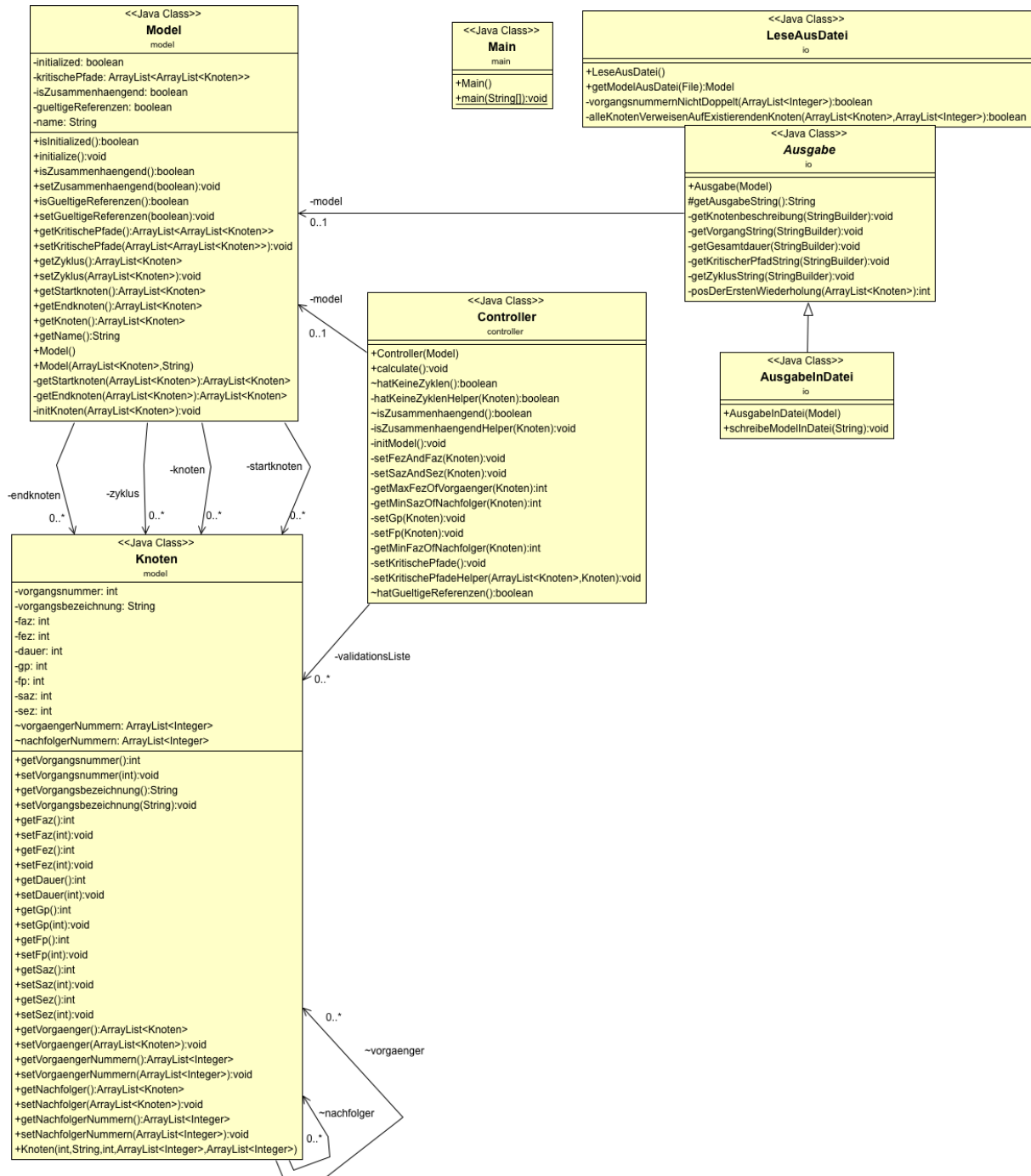
$GP = SAZ - FAZ = SEZ - FEZ$ und $FP = (\text{kleinster FAZ der nachfolgenden Knoten}) - FEZ$

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:

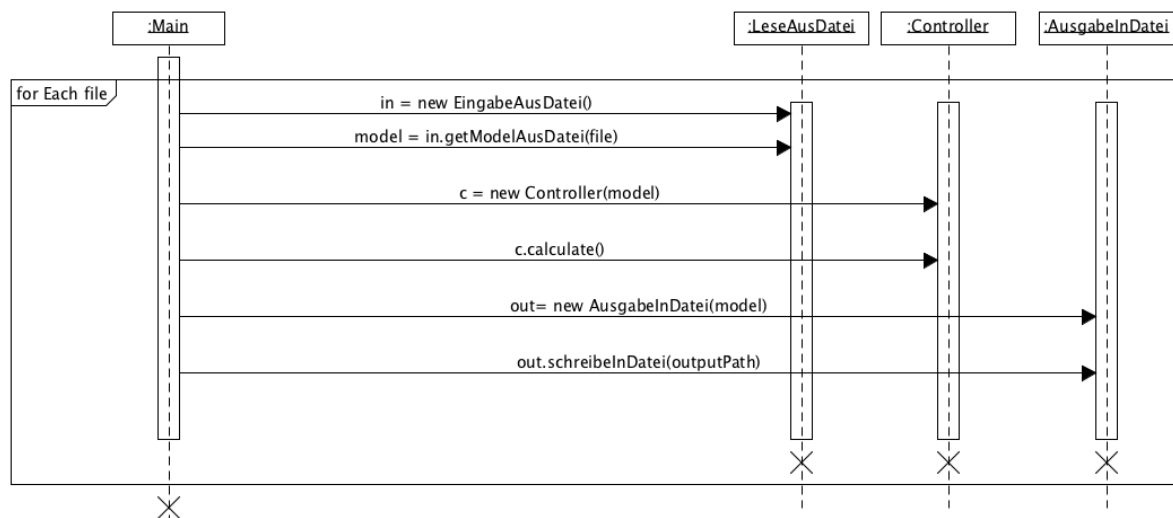
Diese prüft zunächst die Abbruchbedingung, dass der aktuell in der Hilfsmethode betrachtete Knoten ein Endpunkt ist. Ist dies der Fall, wird der berechnete Pfad im externen Model zu einer Liste hinzugefügt und die Methode beendet. Ansonsten wird geprüft, ob der Aktuelle Knoten das Kriterium für einen 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

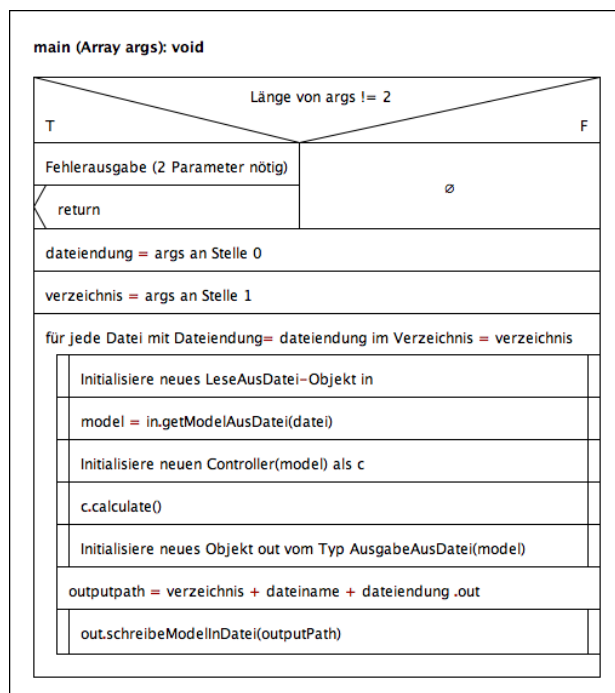


4.2 Programmablauf im Sequenzdiagramm

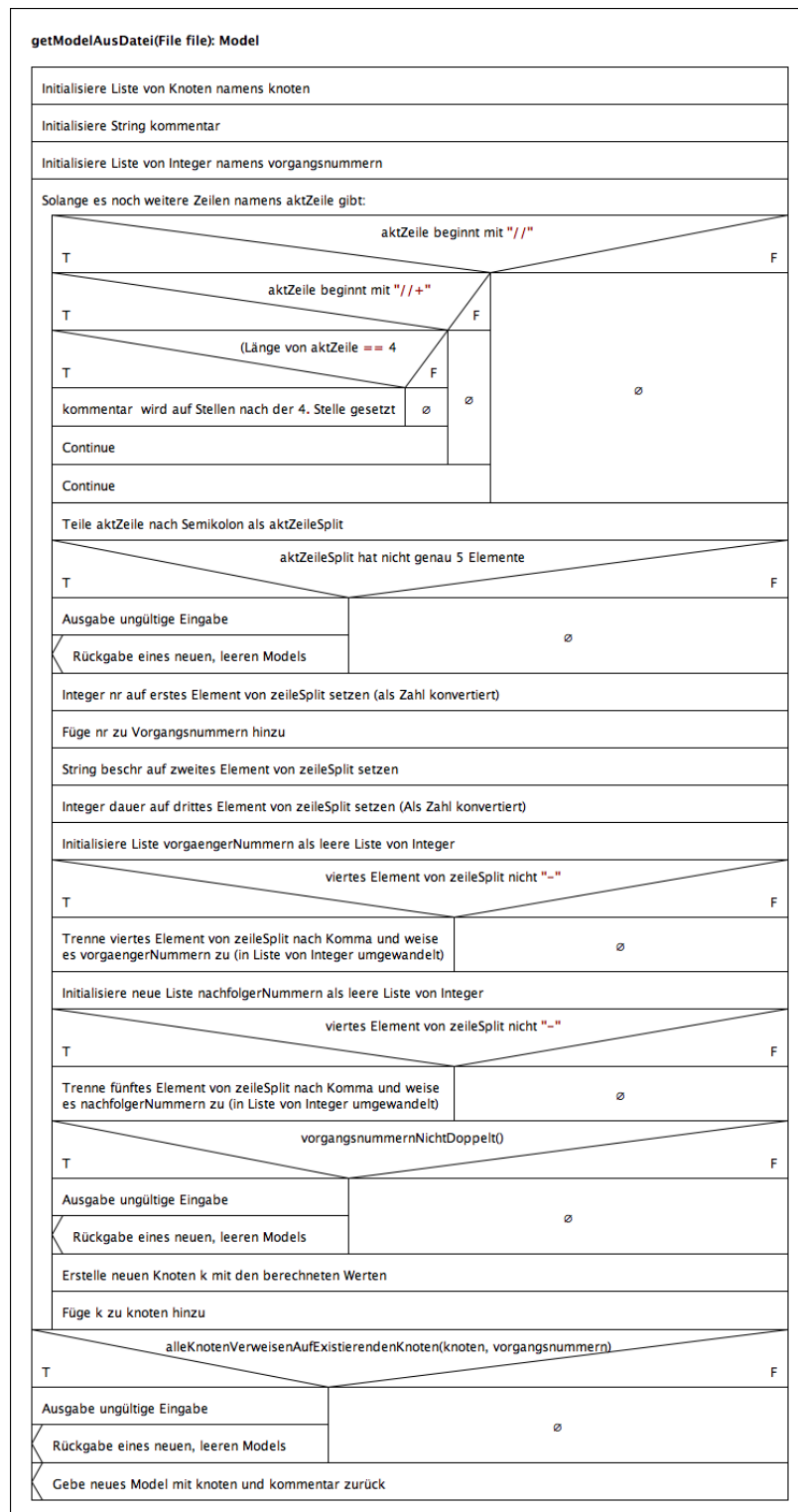


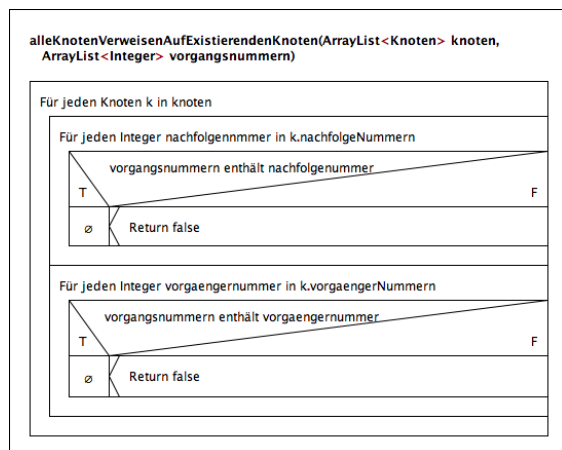
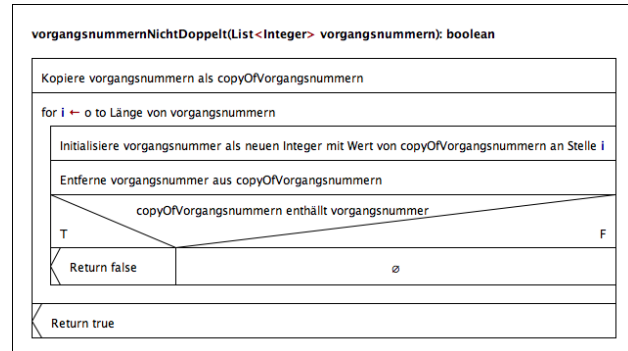
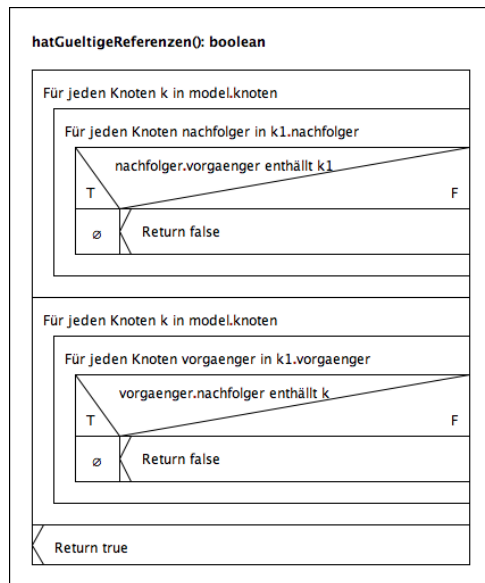
4.3 Nassi-Shneiderman-Diagramme

4.3.1 Main

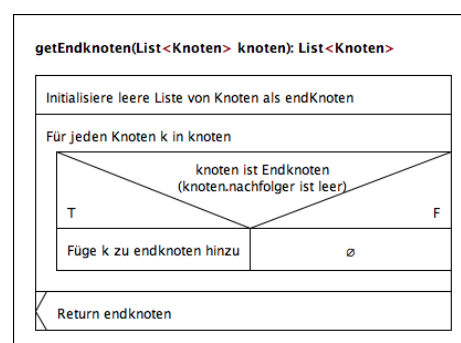
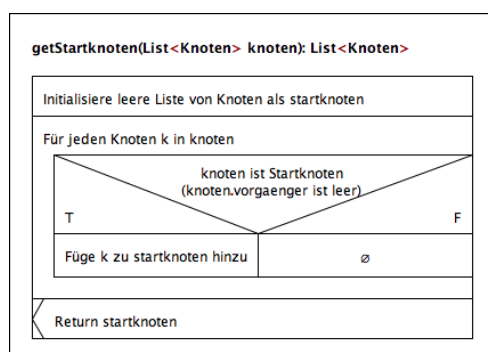


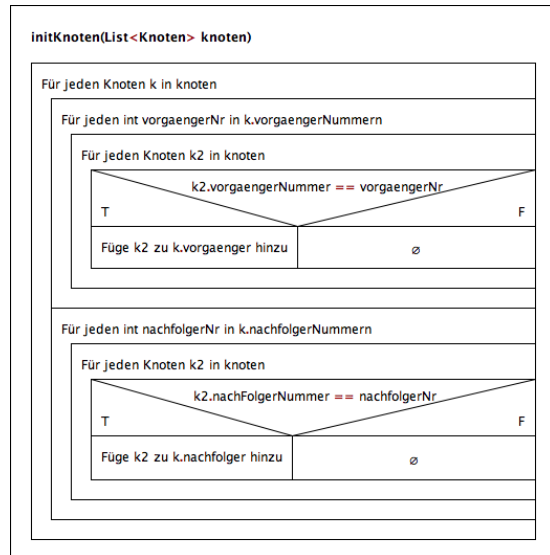
4.3.2 Einlesen einer Datei



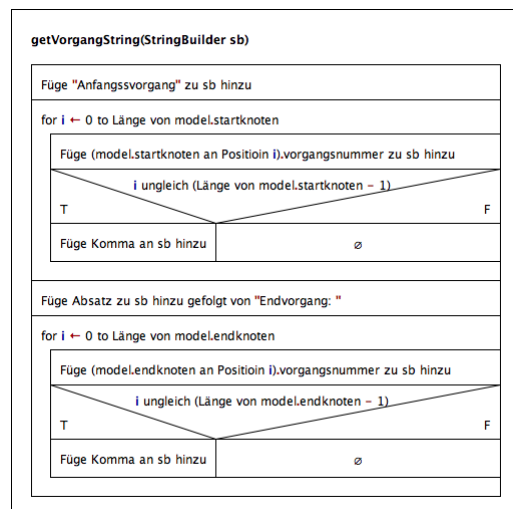
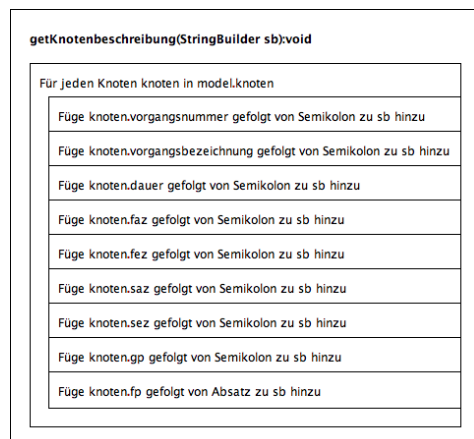
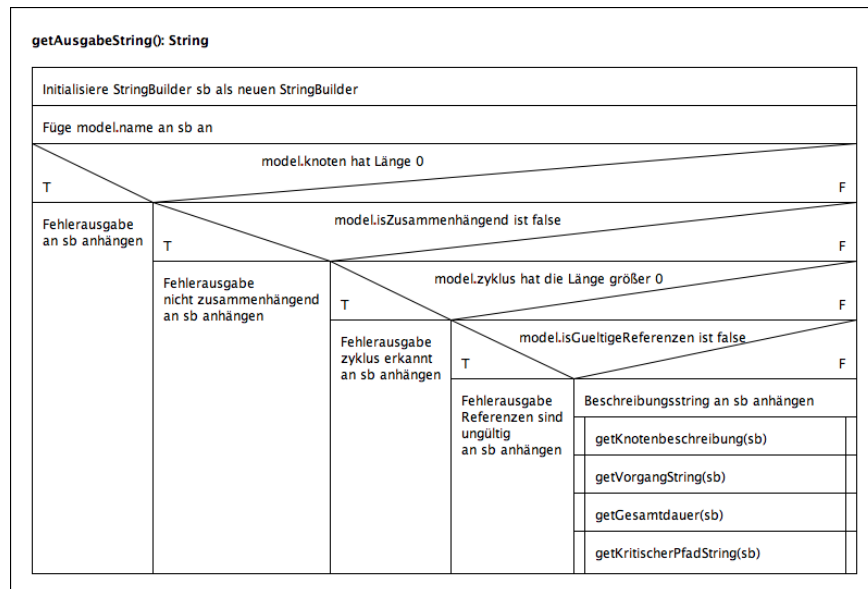


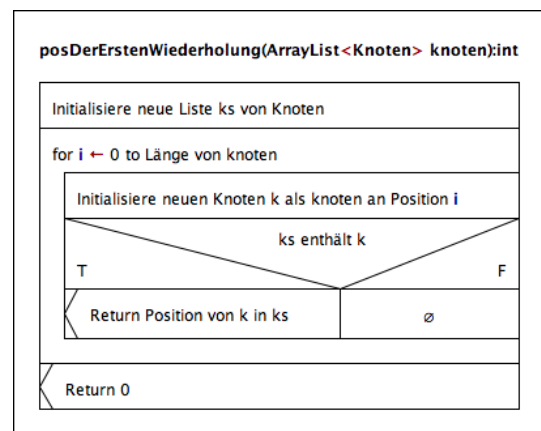
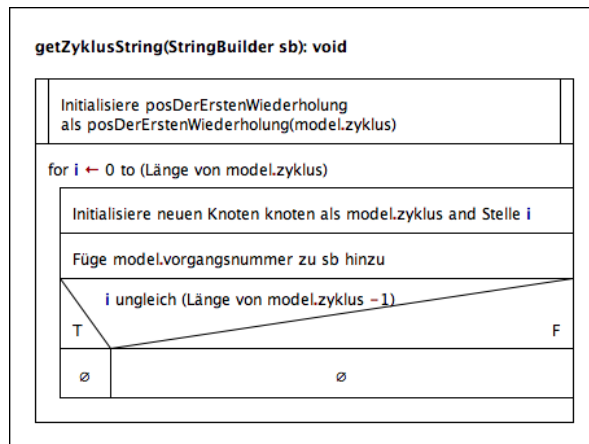
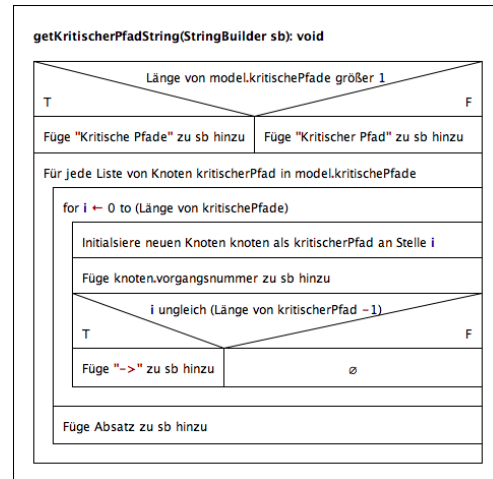
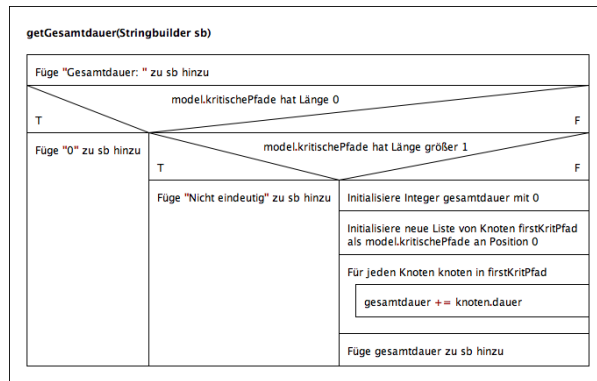
4.3.3 Erzeugung des Models - Modelmethoden



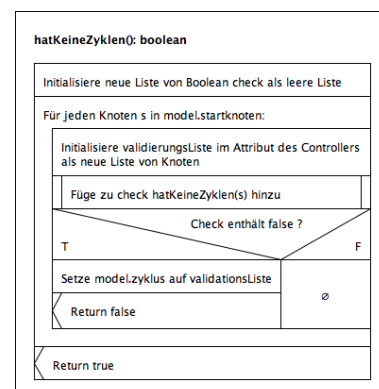
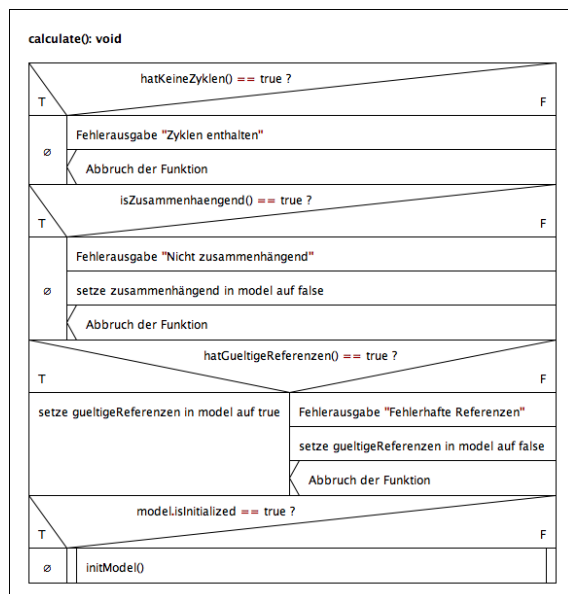


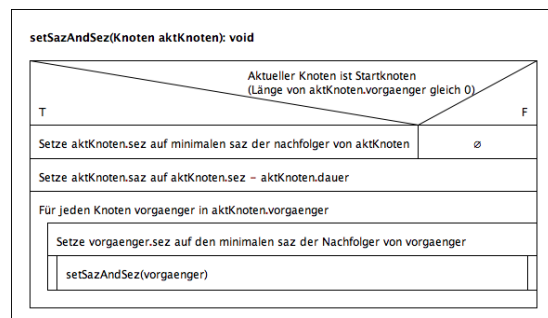
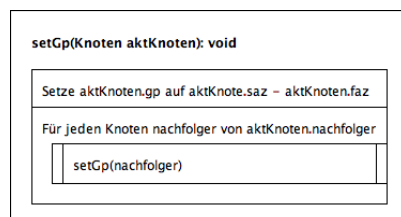
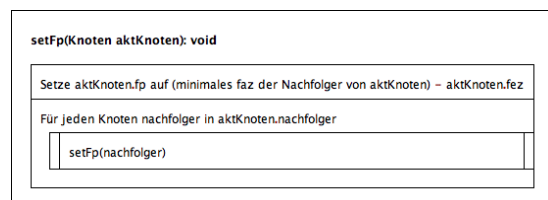
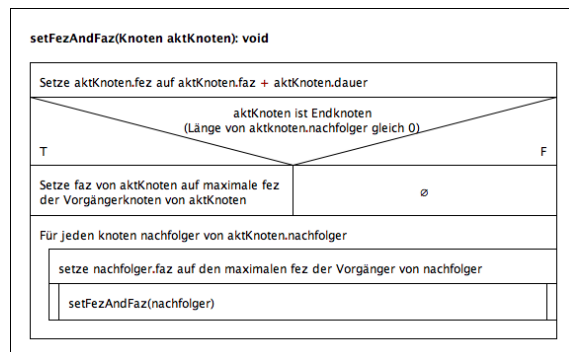
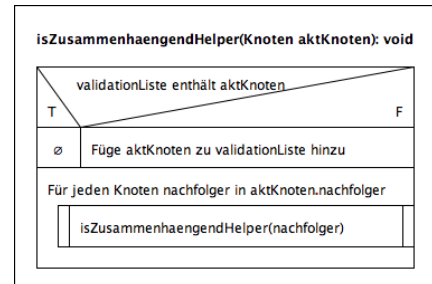
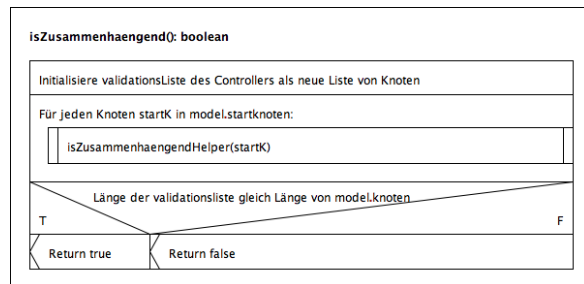
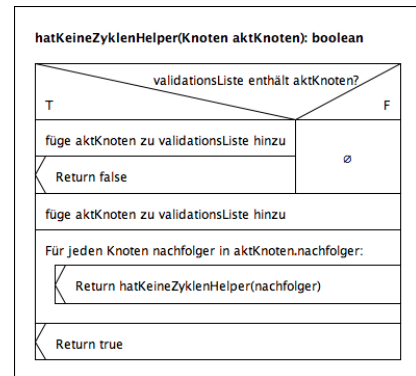
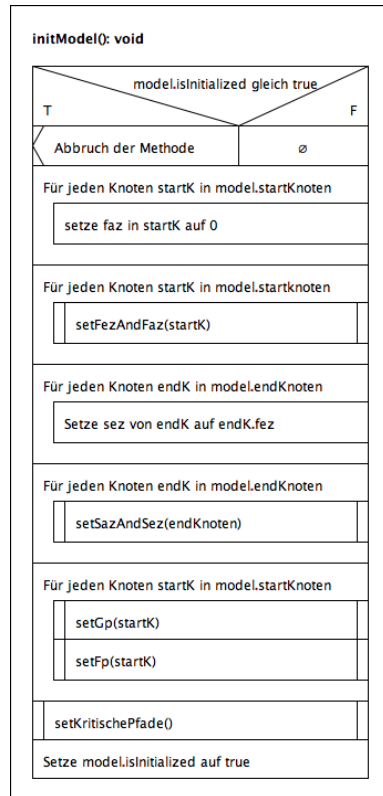
4.3.4 Ausgabe

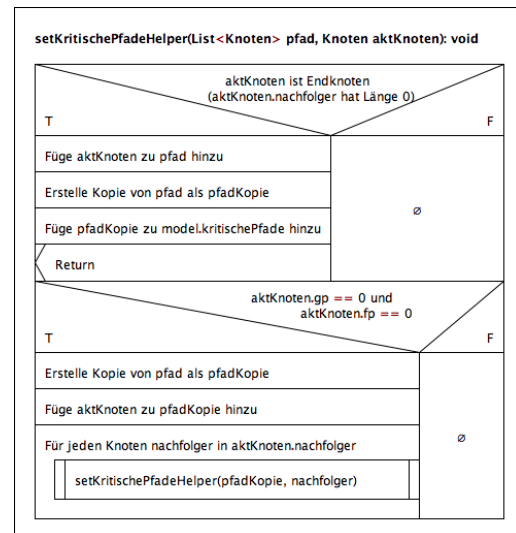
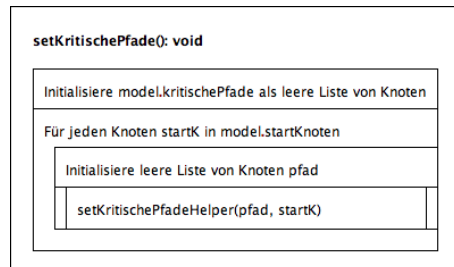




4.3.5 Controllermethoden







5 Abweichung von der handschriftlichen Ausarbeitung

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

5.1 Datenmodell

Einige Methodennamen 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.

5.1.2 Klasse `Model`

Es wurde ein `private` 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, also 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.
- `istZusammenhaengend():boolean` prüft, ob ein Graph zusammenhängend ist. Hier ermöglicht ebenfalls eine Helper-Methode namens

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

- `hatGeltigeReferenzen() : boolean` prüft, ob die Referenzen aller Knoten korrekt angegeben sind, also ob alle Referenzen in den Knoten des Netzplans korrekt sind, also 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.
 - o

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 diese hingewiesen. Werden Strings statt Zahlen eingegeben oder Leerzeichen statt Zahlen, so wird ein entsprechender Fehler auf der Konsole ausgegeben und die Ausgabe entsprechend gestaltet.

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

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

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

5.1.7 Klasse `AusgabeInDatei` (ursprünglich `OutputToFile`)

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

6 Unittests

[...]

7 Testfälle

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

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

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

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

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

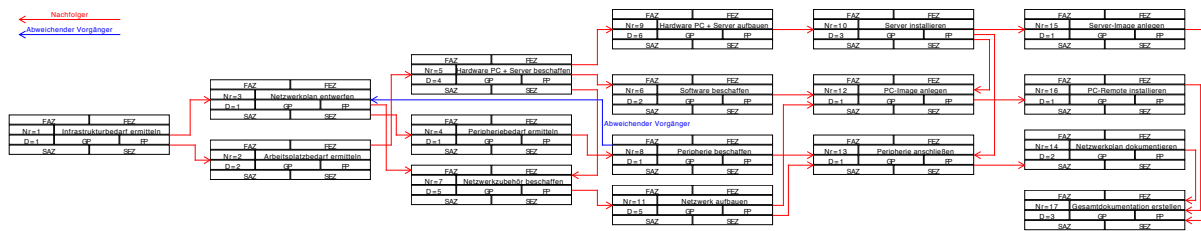
7.1 Besonderheiten der Beispiele 2, 3 und 5 der durch die IHK verbesserten Aufgabenstellung

Das IHK-Beispiel Nummer 2 („Wasserfallmodell“) aus der verbesserten Aufgabenstellung war fehlerhaft. Das bedeutet, dass sie nicht das angegebene Ergebnis lieferte, da die Eingabe falsch formuliert war. Die Vorgangsnummer 6 („Einsatz und Wartung“) referenziert auf einen nichtexistierenden Knoten mit Nachfolgernummer 7. Es existieren jedoch nur 6 Knoten. Die angegebene Ausgabe ist also falsch, da hier ein Fehlerfall vorliegt. Es muss also wie im unter [Fehlerfall 1: Falsche Referenz](#) im Kapitel [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- das unter [Normalfall Komplexes Beispiel](#) im Kapitel [Eigene Normalfälle](#) angegebene Ergebnis liefern.

7.2 Normalfälle

7.2.1 Beispiele aus der durch die IHK verbesserten Aufgabenstellung

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

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

Die Eingabe wird erfolgreich eingelesen und korrekt ausgewertet und ausgegeben.

Eingabe

```
//*****
//+ Installation von POI Kiosken
//*****

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

Ausgabe

Installation von POI Kiosken

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

```
1; Planung des Projekts; 1; 0; 1; 0; 1; 0; 0
2; Beschaffung der POI-Kioske; 25; 1; 26; 1; 26; 0; 0
3; Einrichtung der POI-Kioske; 10; 26; 36; 26; 36; 0; 0
4; Netzwerk installieren; 6; 1; 7; 29; 35; 28; 0
5; Netzwerk einrichten; 1; 7; 8; 35; 36; 28; 28
6; Aufbau der POI Kioske; 2; 36; 38; 36; 38; 0; 0
7; Tests und Nachbesserung der POI Kioske; 1; 38; 39; 38; 39; 0; 0
```

Anfangsvorgang: 1

Endvorgang: 7

Gesamtdauer: 39

Kritischer Pfad

1->2->3->6->7

7.2.2 Eigene Normalfälle

7.2.2.1 Normalfall eines linearen und Graphen

Dieser Normalfall entspricht einem einfachen linearen Graphen und entspricht einer verbesserten Version des Fehlerhaften Beispiels 2 („Wasserfallmodell“) aus der IHK-Aufgabenstellung.

Dieses Beispiel verdeutlicht den nahezu einfachsten Fall eines Netzplans, da keinerlei Verzweigungen vorliegen.

Die Eingabe wird erfolgreich eingelesen und korrekt ausgewertet und ausgegeben.

Eingabe

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

Ausgabe

Testfall Linearer Graph - Wasserfallmodell verbessert aus IHK-Aufgabenstellung

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

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

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

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

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

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

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

Anfangsvorgang: 1

Endvorgang: 6

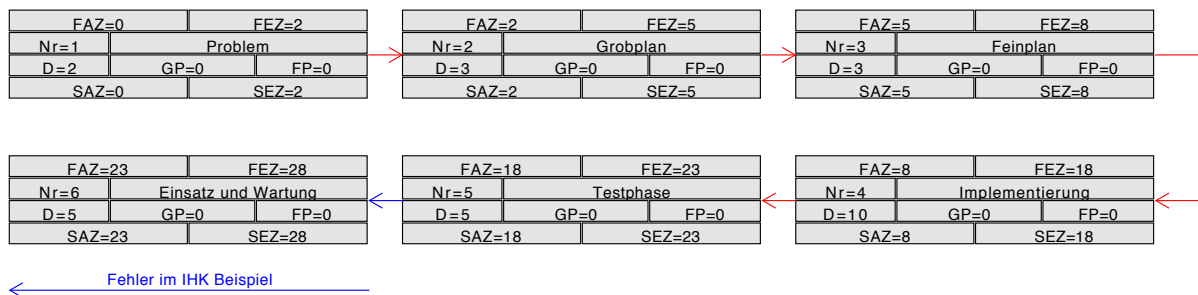
Gesamtdauer: 28

Kritischer Pfad

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

Graphische Abbildung des Testfalls

Im Folgenden ist eine Graphische Darstellung des Testfalls angegeben, die die Abweichung von der IHK-Beispielaufgabe farblich hervorhebt:



7.2.2.2 Normalfall Komplexes Beispiel

Das Beispiel stellt ein sehr 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.

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

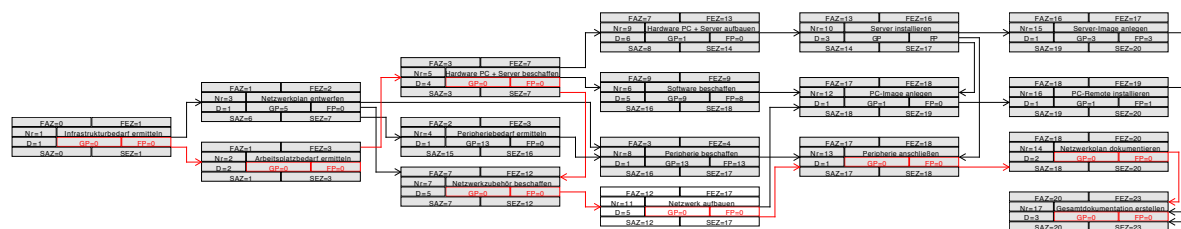
Endvorgang: 17

Gesamtdauer: 23

Kritischer Pfad

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

Grafische Ausgabe des Testfalls



Diskussion

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

7.3 Sonderfälle

7.3.1 Eigene Sonderfälle

7.3.1.1 Negative Vorgangsnummern

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

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.

7.4 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 Ausgabe wird eine Fehlermeldung angegeben sowie der Zyklus aufgelistet.

Eingabe

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

Ausgabe

Beispiel 4 mit Zyklus

Berechnung nicht möglich.

Zyklus erkannt: 3->4->3

7.4.2 Eigene Fehlerfälle

7.4.2.1 Fehlerhafte Referenz

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

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

Eingabe

```
//*****
//+ Fehlerhafte Referenzen Minimal
//*****
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Problemanalyse; 2; -; 2
2; Grobplanung; 3; 1; 3
3; Feinplanung; 3; 2; 4
4; Implementierung; 10; 3; 5
5; Testphase; 5; 4; 7;
6; Einsatz und Wartung; 5; 5; -
```

Ausgabe

Berechnung nicht möglich.

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

7.4.2.2 Mehrfache gleiche Vorgangsnummern

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

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

Eingabe

```
//*****
//+ Mehrfache gleiche Vorgangsnummern
//*****
//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Knoten A; 2; -; 2
2; Knoten B; 3; 1; 3
3; Knoten C; 3; 2; 4
3; Knoten D; 10; 3; 5
5; Knoten E; 5; 4; 6;
6; Knoten F; 5; 5; -
```

Ausgabe

Berechnung nicht möglich.

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

7.4.2.3 Strings statt Zahlen in der Eingabe

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

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

Eingabe

```
//*****
//+ Strings statt Zahlen
//*****

//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
1; Knoten A; "zwei"; -; 2
2; Knoten B; 3; 1; 3
3; Knoten C; 3; 2; 4
4; Knoten D; 10; 3; 5
5; Knoten E; 5; 4; 6;
6; Knoten F; 5; 5; -
```

Ausgabe

Berechnung nicht möglich.

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

7.4.2.4 Leerstellen statt gültigen Zahlen 1

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

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

Eingabe

```
//*****
//+ Leerstellen statt erforderlichen Werte
//*****

//Vorgangsnummer; Vorgangsbezeichnung; Dauer; Vorgänger; Nachfolger
; Knoten A; 2; -; 2
2; Knoten B; 3; 1; 3
3; Knoten C; 3; 2; 4
4; Knoten D; 10; 3; 5
5; Knoten E; 5; 4; -;
```

Ausgabe

Berechnung nicht möglich.

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

7.4.2.5 Leerstellen statt gültigen Zahlen 2

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

In Datei F_LeerstellenStattErforderlichenWerten01.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.

8 Zusammenfassung und Ausblick

9 Anhang: Programmcode

9.1	Package main	36
9.1.1	Klasse Main	36
9.2	Package io	38
9.2.1	Klasse LeseAusDatei	38
9.2.2	Klasse Ausgabe	41
9.2.3	Klasse Ausgabe	44
9.3	Package controller	45
9.3.1	Klasse Ausgabe	45
9.4	Package model	52
9.4.1	Klasse Knoten	52
9.4.2	Klasse Model	54

9.1 Package main

9.1.1 Klasse Main

```
1  package main;
2
3  import java.io.File;
4  import java.io.IOException;
5
6  import controller.Controller;
7  import io.AusgabeInDatei;
8  import io.LeseAusDatei;
9  import model.Model;
10
11 /**
12  * @author M. Leonard Haufs Prüflingsnummer: 101-20540
13  *
14  */
15 public class Main {
16     public static void main(String[] args) {
17         String dateiendung;
18         String verzeichnis;
19
20         // Nur zum Testen- wird später aus über die Argumente args der Jar
21         // übergeben
22         args = new String[2];
23         args[0] = ".in";
24         args[1] = "/Users/hfs23/Dropbox/MATSE/Programmieraufgaben/2018_GrosseProg/Testfaelle";
25
26         // Parameterübergabe prüfen
27         if (args.length != 2) {
28             // keine korrekte Parameterübergabe
29             System.out.println(
30                 "Es müssen 2 Parameter übergeben werden. Paramter 1: Endung der
31                 Eingabedateien (z.B.: .in)\nParameter 2: Verzeichnis aus dem die
32                 Eingabedateien gelesen werden soll.");
33
34             return;
35         }
36         dateiendung = args[0];
37         verzeichnis = args[1];
38
39         File f;
40         try {
41             try {
42                 f = new File(verzeichnis);
43             } catch (Exception ex) {
44                 throw new IOException("Der Angegebene Pfad existiert nicht");
45             }
46
47             if (f.isDirectory() && f.canRead()) {
48                 File[] dateien = f.listFiles();
49                 for (int i = 0; i < dateien.length; i++) {
50                     // Prüfe ob die Datei gelesen werden kann
51                     if (dateien[i].isFile() && dateien[i].canRead()) {
52                         String tempEndung =
53                             dateien[i].getName().substring(dateien[i].getName().lastIndexOf("."),
54                                 dateien[i].getName().length());
55                         // wenn die Dateiendung der gewählten entspricht
56                         // wird die Datei eingelesen
57                         if (dateiendung.equals(tempEndung)) {
58
59                             // Eingabe
60
61                             LeseAusDatei in = new LeseAusDatei();
62                             Model model = in.getModelAusDatei(dateien[i]);
63
64                             // Berechnung
65                             Controller c = new Controller(model);
66                             c.calculate();
67
68                             // Ausgabe
69                             AusgabeInDatei out = new AusgabeInDatei(model);
70
71                             String outputPath = verzeichnis + "/" +
72                                 (dateien[i].getName().replace(dateiendung, ".out"));
73                             out.schreibeModelInDatei(outputPath);
74
75                             // OutputConsole out = new OutputConsole();
76                             // out.printEntireOutputString(model);
77                         }
78                     }
79                 }
80             }
81         }
82     }
83 }
```

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

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 Modells 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 Modells, die in einer Datei gespeichert sind.
24      *
25      * @param file      Datei, aus der gelesen werden soll.
26      * @return Model mit dem gekapselten Daten. Falls eine ungültige Eingabe
27      *         erfolgt, wird ein leeres Model zurückgegeben.
28      */
29     public Model getModelAusDatei(File file) {
30         ArrayList<Knoten> knoten = new ArrayList<>();
31         String kommentar = "Fehler beim Einlesen.";
32         ArrayList<Integer> vorgangsnummern = new ArrayList<>();
33
34         BufferedReader br;
35         try {
36             br = new BufferedReader(new InputStreamReader(new FileInputStream(file)));
37         } catch (FileNotFoundException ex) {
38             System.out.println(ex);
39             return new Model();
40         }
41
42         try {
43             String aktZeile = "";
44             while ((aktZeile = br.readLine()) != null) {
45                 if (aktZeile.startsWith("//")) {
46                     // Zeile beginnt mit "//+ " ?
47                     if (aktZeile.startsWith("//+ ")) {
48                         if (aktZeile.length() > 4) {
49                             kommentar = aktZeile.substring(4, aktZeile.length());
50                         }
51                         continue;
52                     }
53                     continue;
54                 }
55             }
56
57             String aktZeileOhneLeer = aktZeile.replace(" ", "");
58             String[] zeileSplit = aktZeileOhneLeer.split(";");
59             if (zeileSplit.length != 5) {
60                 System.out.println("In Datei " + file.getName()
61                     + ": Ungenügende Eingabe. Es müssen je Zeile genau 5 Argumente
62                     + "getrennt mit einem Semikolon übergeben werden: "
63                     + aktZeile);
64                 br.close();
65                 return new Model();
66             }
67             int nr = Integer.parseInt(zeileSplit[0]);
68             vorgangsnummern.add(nr);
69             String beschr = aktZeile.split("; ")[1];
70             int dauer = Integer.parseInt(zeileSplit[2]);
71
72             ArrayList<Integer> vorgaengerNummern = new ArrayList<>();
73             if (!zeileSplit[3].equals("-")) {
74                 String[] vorgaengerNummernArr = zeileSplit[3].split(",");
75                 for (int i = 0; i < vorgaengerNummernArr.length; i++) {
76                     String string = vorgaengerNummernArr[i];
77                     int number = Integer.parseInt(string);
78                     vorgaengerNummern.add(number);
79                 }
80             }
81         }
82     }
83 }
```



```

78         }
79     }
80
81     ArrayList<Integer> nachfolgerNummern = new ArrayList<>();
82     if (!zeileSplit[4].equals("-")) {
83         String[] nachfolgerNummernArr = zeileSplit[4].split(",");
84         for (int i = 0; i < nachfolgerNummernArr.length; i++) {
85             String string = nachfolgerNummernArr[i];
86             int number = Integer.parseInt(string);
87             nachfolgerNummern.add(number);
88         }
89     }
90     // Prüfe, ob vorgangsnummern nicht doppelt vorliegen
91     if (!vorgangsnummernNichtDoppelt(vorgangsnummern)) {
92         System.out.println("In Datei " + file.getName()
93             + ": Ungenügende Eingabe: Es kommt mindestens eine Vorgangsnummer
94             mehrfach vor.");
95         br.close();
96         return new Model();
97     }
98     Knoten k = new Knoten(nr, beschr, dauer, vorgaengerNummern, nachfolgerNummern);
99     knoten.add(k);
100 }
101 br.close();
102 } catch (IOException ex) {
103     System.out.println(ex);
104     new Model();
105 } catch (NumberFormatException e) {
106     System.out.println("In Datei " + file.getName()
107         + ": Ungenügende Eingabe. Es wurde mindestens eine ungültige Zahl
108         eingeben.");
109     return new Model();
110 }
111 if (!alleKnotenVerweisenAufExistierendenKnoten(knoten, vorgangsnummern)) {
112     System.out.println("In Datei " + file.getName()
113         + ": Ungenügende Eingabe: Es existieren ungültige Referenzen, da
114         mindestens ein Knoten auf einen nicht existenten Knoten
115         referenziert.");
116     return new Model();
117 }
118 Model model = new Model(knoten, kommentar);
119 return model;
120 }
121
122 /**
123  * Prüft, ob die Vorgangsnummern nicht doppelt vorliegen
124  *
125  * @param vorgangsnummern
126  *       die zu Prüfen sind
127  * @return true, falls die Vorgangsnummern nicht doppelt vorliegen
128  */
129 private boolean vorgangsnummernNichtDoppelt(ArrayList<Integer> vorgangsnummern) {
130     @SuppressWarnings("unchecked")
131     ArrayList<Integer> copyOfVorgangsnummern = (ArrayList<Integer>)
132         vorgangsnummern.clone();
133
134     for (int i = 0; i < copyOfVorgangsnummern.size(); i++) {
135         Integer vorgangsnummer = copyOfVorgangsnummern.get(i);
136         copyOfVorgangsnummern.remove(vorgangsnummer);
137         if (copyOfVorgangsnummern.contains(Integer.valueOf(vorgangsnummer))) {
138             return false;
139         }
140     }
141     return true;
142 }
143
144 /**
145  * Prüft, ob alle Knoten auf einen existierenden Knoten verweisen.
146  *
147  * @param knoten
148  *       Knotenliste, der zu prüfenden Knoten
149  * @param vorgangsnummern
150  *       Liste der Vorgangsnummern aller Knoten
151  * @return true, falls alle Knoten auf einen existierenden Knoten verweisen,
152  *         sonst false
153  */
154 private boolean alleKnotenVerweisenAufExistierendenKnoten(ArrayList<Knoten> knoten,
155     ArrayList<Integer> vorgangsnummern) {
156     for (Knoten k : knoten) {
157         for (int nachfolgernummer : k.getNachfolgerNummern()) {
158             if (!vorgangsnummern.contains(Integer.valueOf(nachfolgernummer))) {
159                 return false;
160             }
161         }
162     }
163 }

```

```

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

```

9.2.2 Klasse Ausgabe

```
1 package io;
2
3 import java.util.ArrayList;
4
5 import model.Knoten;
6 import model.Model;
7
8 /**
9  * Ermöglicht zu einem Model die Ausgabe der kenngrößen und kritischen Pfade
10  * auszugeben
11  *
12  * @author M. Leonard Haufs 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             sb.append(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             sb.append(this.getKnotenbeschreibung(sb));
71             sb.append("\n");
72             sb.append(this.getVorgangString(sb));
73             sb.append("\n");
74             sb.append(this.getGesamtdauer(sb));
75             sb.append("\n");
76             sb.append("\n");
77             sb.append(this.getKritischerPfadString(sb));
```

```

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

```

```

162     }
163 }
164
165 /**
166  * 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 private void getZyklusString(StringBuilder sb) {
202     int posDerErstenWiederholung = this.posDerErstenWiederholung(this.model.getZyklus());
203
204     for (int i = posDerErstenWiederholung; i < this.model.getZyklus().size(); i++) {
205         Knoten knoten = this.model.getZyklus().get(i);
206         sb.append(knoten.getVorgangsnummer());
207         if (i != this.model.getZyklus().size() - 1) {
208             sb.append("->");
209         }
210     }
211     sb.append("\n");
212 }
213
214 /**
215  * Gibt die Position des ersten Elementes in einer ArrayList von Knoten zurück,
216  * die doppelt vorkommt
217  *
218  * @param knoten
219  *         ArrayList<Knoten>, die überprüft werden soll
220  * @return Position des ersten Elementes in einer ArrayList von Knoten, die
221  *         doppelt vorkommt
222  */
223 private int posDerErstenWiederholung(ArrayList<Knoten> knoten) {
224     ArrayList<Knoten> ks = new ArrayList<>();
225     for (int i = 0; i < knoten.size(); i++) {
226         Knoten k = knoten.get(i);
227         if (ks.contains(k)) {
228             return ks.indexOf(k);
229         }
230         ks.add(k);
231     }
232     return 0;
233 }
234 }
235
236 }

```

9.2.3 Klasse Ausgabe

```
1  package io;
2
3  import java.io.File;
4  import java.io.FileWriter;
5  import java.io.IOException;
6
7  import model.Model;
8
9  /**
10   *
11   * @author M. Leonard Haufs 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 Ausgabe

```
1 package controller;
2
3 import java.util.ArrayList;
4
5 import model.Knoten;
6 import model.Model;
7
8 /**
9  * Hauptberechnungsklasse.
10  *
11  * @author M. Leonard Haufs 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   * @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     // * Setze FAZ für alle Nachfolger der Startknoten als der größte
197     // * (späteste) FEZ der unmittelbaren Vorgänger.
198     // */
199     // for (Knoten startK : this.model.getStartknoten()) {
200     //     for (Knoten nachfolger : startK.getNachfolger()) {
201     //         setFaz(nachfolger);
202     //     }
203     // }
204
205     /*
206     * 2. Phase: Rückwärtsrechnung
207     *
208     * Bei der Rückwärtsrechnung wird ermittelt, wann die einzelnen Vorgänge
209     * spätestens begonnen und fertiggestellt sein müssen, damit die
210     * Gesamtprojektzeit nicht gefährdet ist.
211     *
212     * Für den letzten Vorgang ist der früheste Endzeitpunkt (FEZ) auch der
213     * späteste Endzeitpunkt (SEZ), also SEZ = FEZ.
214     */
215     for (Knoten endK : this.model.getEndknoten()) {
216         endK.setSez(endK.getFez());
217     }
218
219     /*
220     * Für den spätesten Anfangszeitpunkt gilt: SAZ = SEZ - Dauer.
221     */
222     for (Knoten endKnoten : this.model.getEndknoten()) {
223         this.setSazAndSez(endKnoten);
224     }
225
226     // /*
227     // * Der SAZ eines Vorgangs wird SEZ aller unmittelbarer Vorgänger
228     // *
229     // * Haben mehrere Vorgänge einen gemeinsamen Vorgänger, so ist dessen SEZ der
230     // * früheste (kleinste) SAZ aller Nachfolger.
231     // */
232     // for (Knoten endK : this.model.getEndknoten()) {
233     //     setSez(endK);
234     // }
235
236     // 3. Phase: Ermittlung der Zeitreserven
237     for (Knoten startK : this.model.getStartknoten()) {
238         /*
239         * Berechnung des Gesamtpuffers für jeden Knoten
240         */
241         this.setGp(startK);
242
243         /*
244         * Berechnung des freien Puffers
245         */

```

```

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

```

```

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

```

```

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

```

```

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

```

9.4 Package model

9.4.1 Klasse Knoten

```
1  package model;
2
3  import java.util.ArrayList;
4
5  /**
6   *
7   * @author M. Leonard Haufs 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,
111                  ArrayList<Integer> vorgaengerNummern, ArrayList<Integer> nachfolgerNummern) {
112         super();
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 private ArrayList<Knoten> getStartknoten(ArrayList<Knoten> knoten) {
108     ArrayList<Knoten> startknoten = new ArrayList<>();
109     for (Knoten k : knoten) {
110         if (k.getVorgaengerNummern().size() == 0) {
111             startknoten.add(k);
112         }
113     }
114
115     return startknoten;
116 }
117
118 private ArrayList<Knoten> getEndknoten(ArrayList<Knoten> knoten) {
119     ArrayList<Knoten> endknoten = new ArrayList<>();
120     for (Knoten k : knoten) {
121         if (k.getNachfolgerNummern().size() == 0) {
122             endknoten.add(k);
123         }
124     }
125
126     return endknoten;
127 }
128
129 private void initKnoten(ArrayList<Knoten> knoten) {
130     for (Knoten k : knoten) {
131         for (int vorgaengerNr : k.getVorgaengerNummern()) {
132             for (Knoten k2 : knoten) {
133                 if (k2.getVorgangsnummer() == vorgaengerNr) {
134                     k.getVorgaenger().add(k2);
135                 }
136             }
137         }
138
139         for (int nachfolgerNr : k.getNachfolgerNummern()) {
140             for (Knoten k2 : knoten) {
141                 if (k2.getVorgangsnummer() == nachfolgerNr) {
142                     k.getNachfolger().add(k2);
143                 }
144             }
145         }
146     }
147 }
148
149 }

```