

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



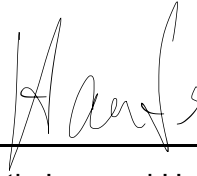
# I Inhaltsverzeichnis

<b>I</b>	<b>Inhaltsverzeichnis .....</b>	<b>i</b>
<b>1</b>	<b>Eigenständigkeitserklärung .....</b>	<b>1</b>
<b>2</b>	<b>Benutzeranleitung .....</b>	<b>2</b>
2.1	Systemvoraussetzungen und Hinweise zum Aufruf .....	2
2.2	Installation des Programms .....	2
2.3	Programmstart .....	2
2.4	Externe Programme zur Betrachtung der Diagramme .....	3
<b>3</b>	<b>Aufgabenanalyse .....</b>	<b>4</b>
3.1	Allgemeine Problemstellung .....	4
3.2	Format der Eingabedatei .....	4
3.3	Format der Ausgabedatei .....	5
3.4	Algorithmus .....	6
3.5	Verbale Beschreibung des Verfahrens .....	6
3.6	Einlesen der Eingabedatei .....	7
3.7	Überführung der Eingabedaten ins Datenmodell .....	7
3.8	Berechnung im Controller .....	7
<b>4</b>	<b>Programmkonzeption .....</b>	<b>9</b>
4.1	UML Klassendiagramm .....	9
4.2	Programmablauf im Sequenzdiagramm .....	10
4.3	Nassi-Shneiderman-Diagramme .....	10
4.3.1	Main .....	10
4.3.2	Einlesen einer Datei .....	11
4.3.3	Erzeugung des Models - Modelmethoden .....	12
4.3.4	Ausgabe .....	13
4.3.5	Controllermethoden .....	14
<b>5</b>	<b>Abweichung von der handschriftlichen Ausarbeitung .....</b>	<b>17</b>
5.1	Datenmodell .....	17
5.1.1	Die Sichtbarkeiten der Methoden .....	17
5.1.2	Klasse <code>Model</code> .....	17
5.1.3	Klasse <code>Knoten</code> .....	17

5.1.4	Klasse <code>Controller</code> .....	17
5.1.5	Klasse <code>LeseAusDatei</code> (Ursprünglich <code>InputFromFile</code> ) .....	18
5.1.6	Abstrakte Klasse <code>Ausgabe</code> (ursprünglich <code>Output</code> ).....	18
5.1.7	Klasse <code>AusgabeInDatei</code> (ursprünglich <code>OutputToFile</code> ).....	19
<b>6</b>	<b>Unittests</b> .....	<b>20</b>
6.1	Prüfung der Methode <code>hatKeineZyklen()</code> .....	20
6.2	Prüfung der Methode <code>isZusammenhaengend()</code> .....	21
6.3	Prüfung der Methode <code>hatGuelteReferenzen()</code> .....	21
<b>7</b>	<b>Blackbox- Testfälle</b> .....	<b>22</b>
7.1	Besonderheiten der Beispiele 2, 3 und 5 der durch die IHK verbesserten Aufgabenstellung.....	22
7.2	Normalfälle .....	23
7.2.1	Beispiele aus der durch die IHK verbesserten Aufgabenstellung.....	23
7.2.2	Eigene Normalfälle .....	24
7.3	Sonderfälle .....	28
7.3.1	Eigene Sonderfälle .....	28
7.4	Fehlerfälle.....	29
7.4.1	Beispiele der IHK.....	29
7.4.2	Eigene Fehlerfälle.....	30
<b>8</b>	<b>Zusammenfassung und Ausblick</b> .....	<b>34</b>
8.1	Ausblick .....	34
<b>9</b>	<b>Anhang: Programmcode</b> .....	<b>35</b>

### 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. | 2

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

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

## 2 Benutzeranleitung

---

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

### 2.4 Externe Programme zur Betrachtung der Diagramme

Für die Nassi Shneidermann- Diagramme wurde das Programm *Structorizer*<sup>1</sup> verwendet.

Für die Erstellung des Sequenzdiagramms und der anderen Abbildungen dieses Dokumentes wurde das Programm Umllet<sup>2</sup> verwendet.

Die im Hauptverzeichnis GrosseProg\_101201540 unter dem Ordner „Diagramme“ gespeicherten Dateien lassen sich mit diesen Programmen öffnen.

Eclipse-Plugin auflisten!

---

<sup>1</sup> <http://structorizer.fisch.lu>

<sup>2</sup> <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.

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



### 3 Aufgabenanalyse

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 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 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.

### 3 Aufgabenanalyse

$GP = SAZ - FAZ = SEZ - FEZ$  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 Dateiergung, 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 Dateiergung der dem Programm übergebenen Endung entspricht. Falls dies nicht der Fall ist, wird die nächste Datei überprüft. Für jede Datei mit entsprechender Dateiergung wird zusätzlich die Lesbarkeit dieser Datei festgestellt. Kann die Datei nicht gelesen werden, wird anschließend die nächste Datei untersucht.

Immer ein Fehler ausgegeben.

Vorgangsnummern nicht doppelt-überprüfung

### 3.7 Überführung der Eingabedaten ins Datenmodell

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 <sup>Zyklus</sup>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 <sup>Zyklus</sup>zyklus festgestellt. Falls jeder Knoten nur einmal durchlaufen wird, so wird die <sup>Zyklus</sup>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:

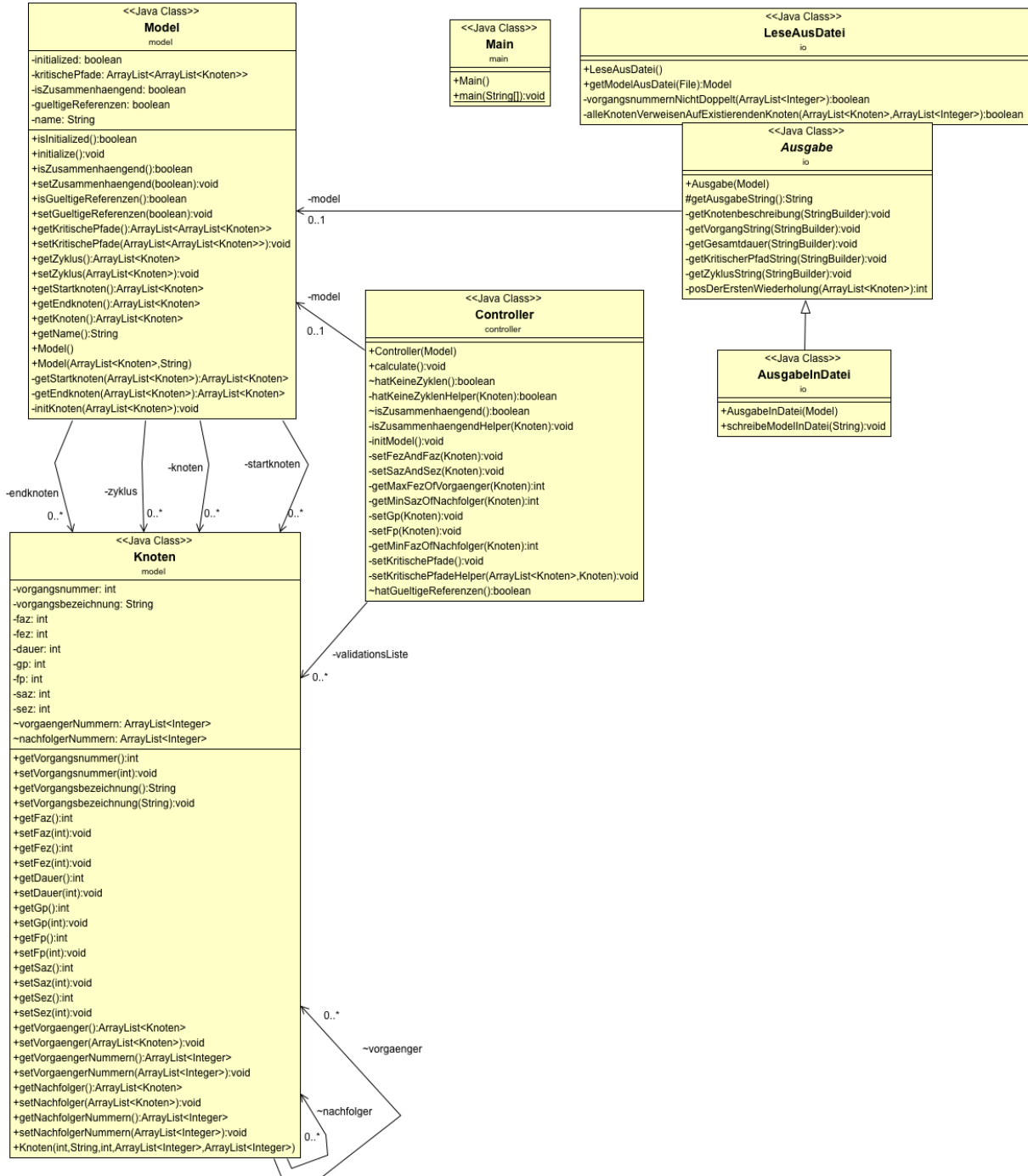
- 1) 4) Vorwärtsrechnung:  
Bei gegebenem Anfangstermin werden aufgrund der angegebenen Dauer eines Vorganges die frühestmöglichen Anfangs- und Endzeiten eingetragen. Weiterhin lässt sich die Gesamtdauer eines Projekts bestimmen. Dabei wird der Baum von allen Startknoten aus vorwärts durchlaufen:  
Der 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.
- 5) 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.
- 6) 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:

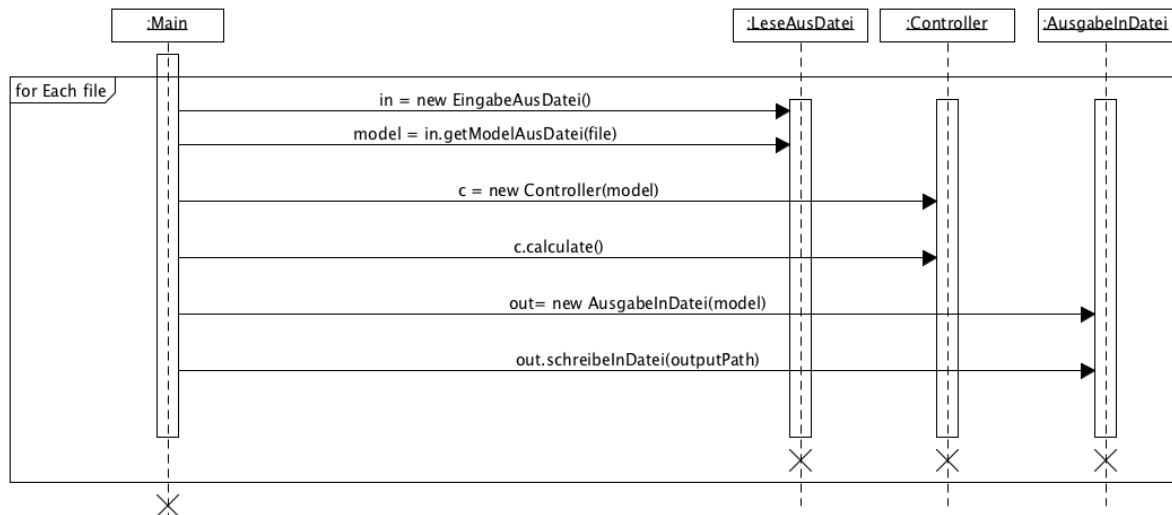
Diese prüft zunächst die Abbruchbedingung, dass der aktuell in der Hilfsmethode betrachtete Knoten ein Endpunkt ist. Ist dies der Fall, wird der berechnete Pfad im externen Model zu einer Liste hinzugefügt und die Methode beendet. Ansonsten wird geprüft, ob der Aktuelle Knoten das Kriterium für einen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

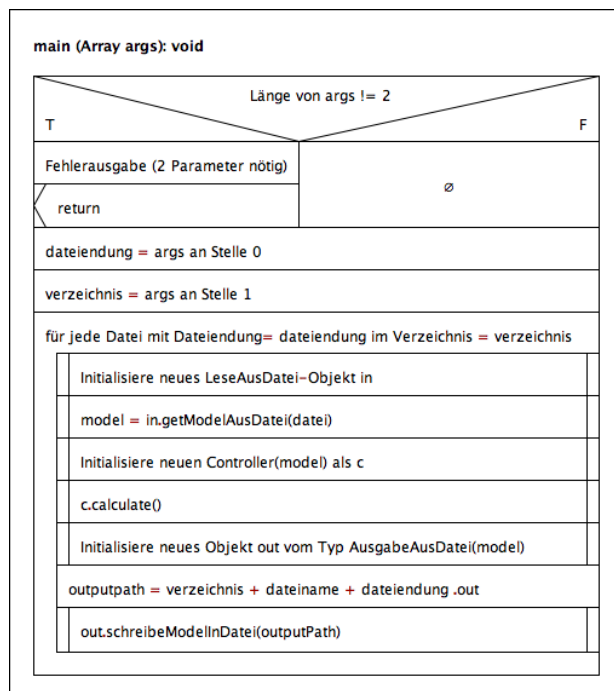


## 4.2 Programmablauf im Sequenzdiagramm



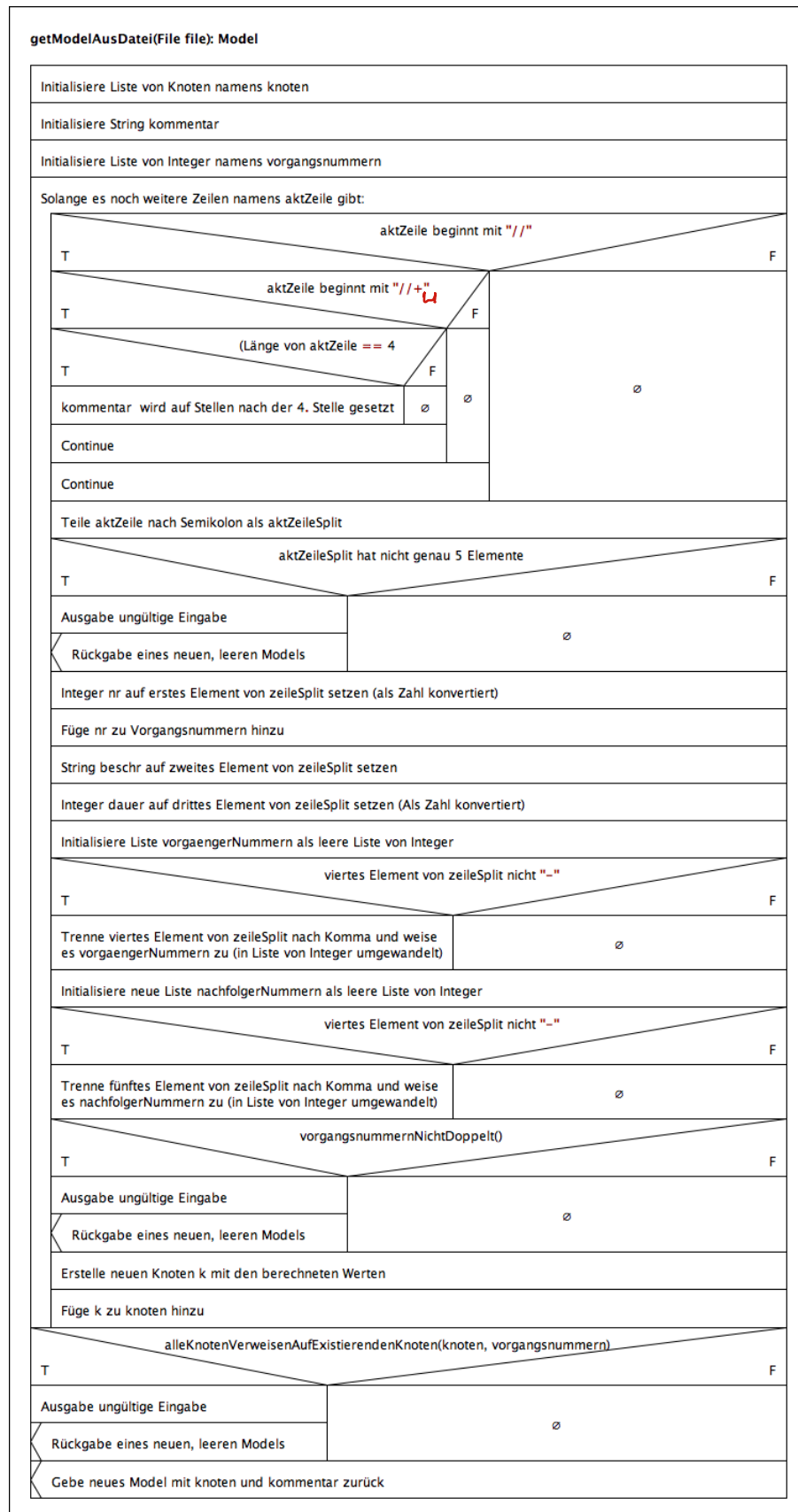
## 4.3 Nassi-Shneiderman-Diagramme

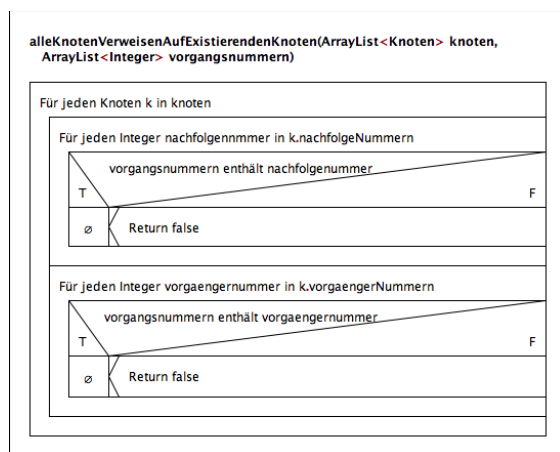
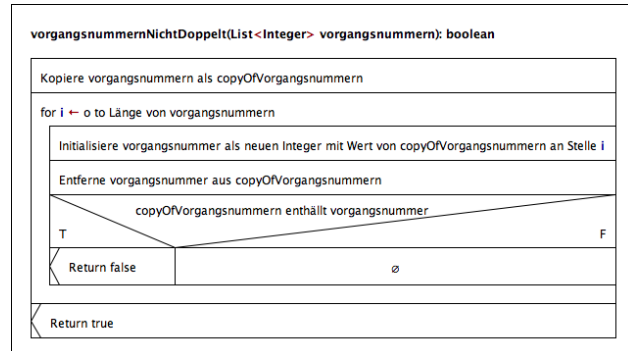
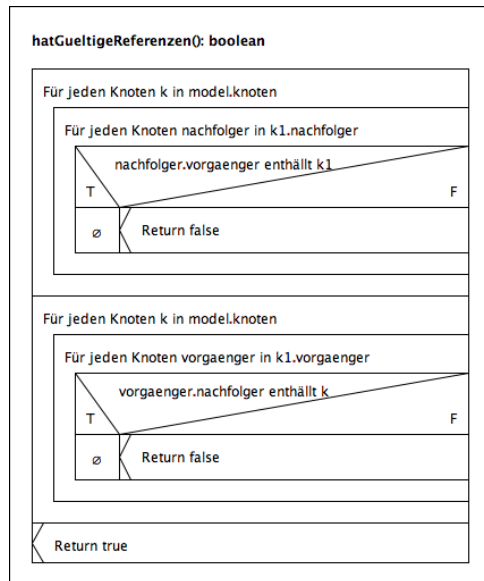
### 4.3.1 Main



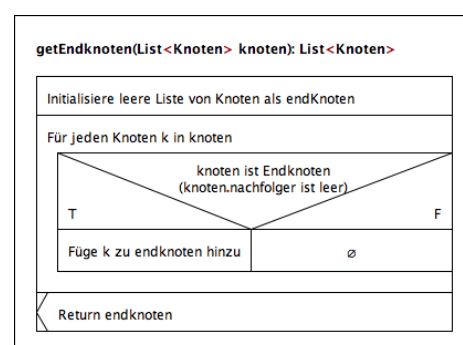
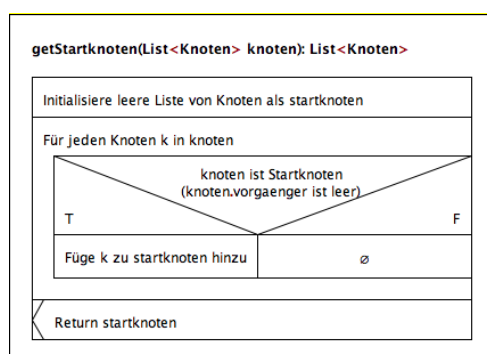
## 4 Programmkonzeption

### 4.3.2 Einlesen einer Datei *Klassenname*



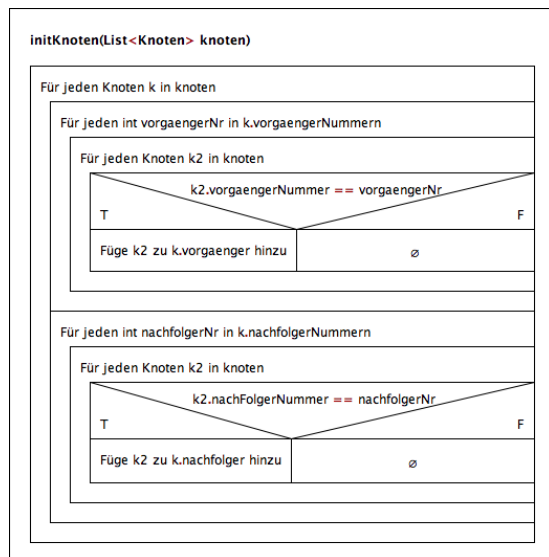


### 4.3.3 Erzeugung des Models - Modelmethoden

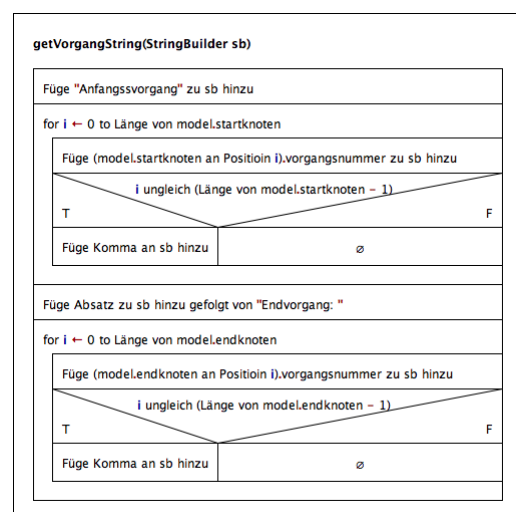
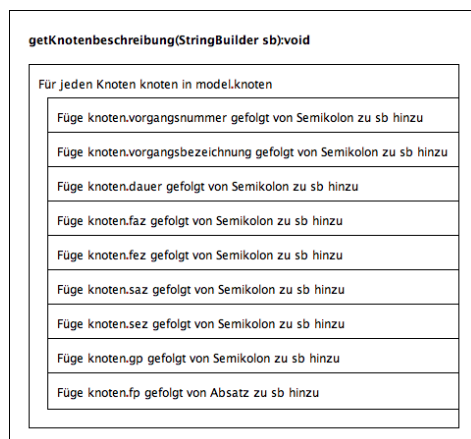
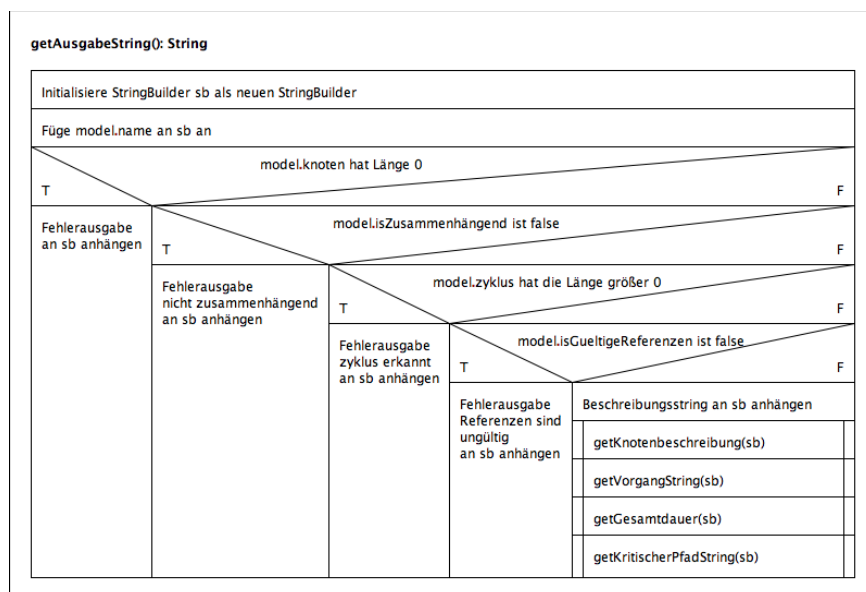


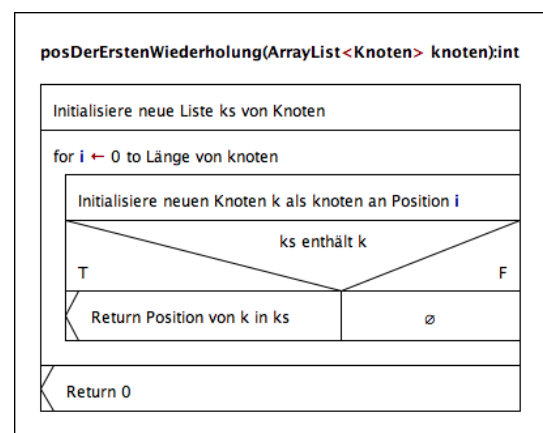
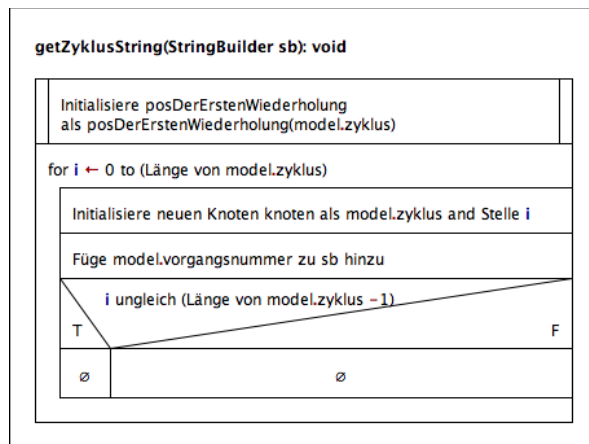
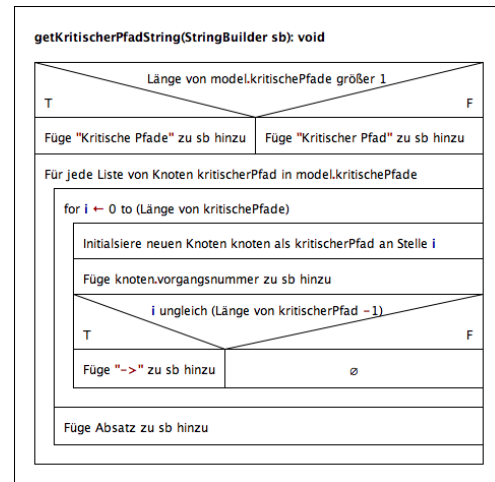
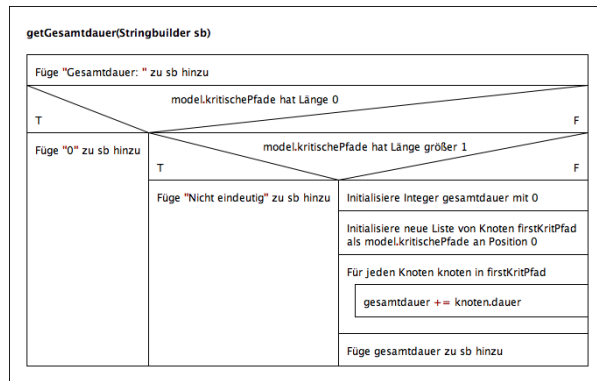


## 4 Programmkonzeption

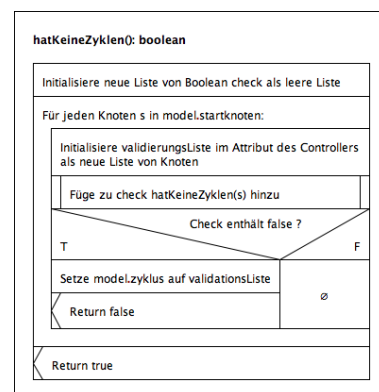
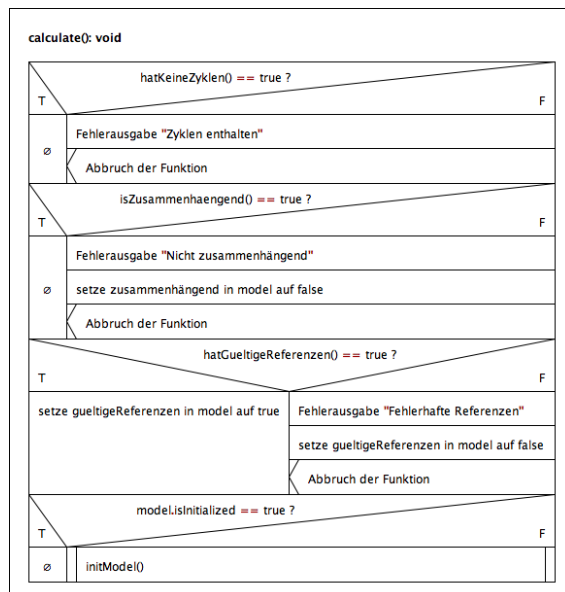


### 4.3.4 Ausgabe

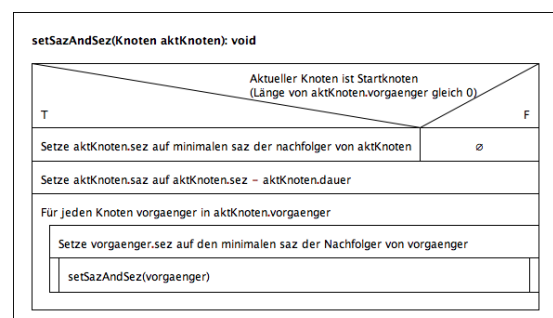
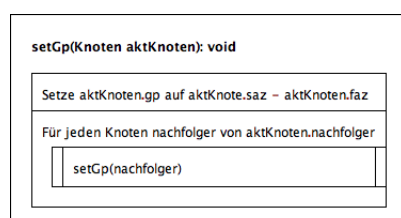
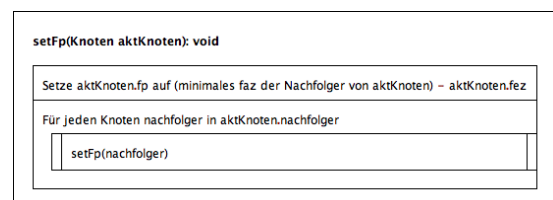
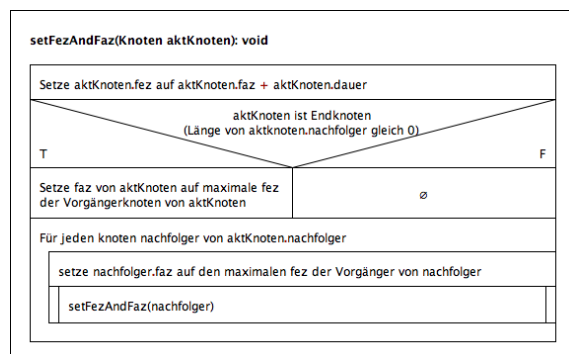
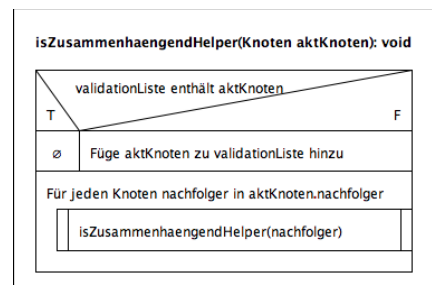
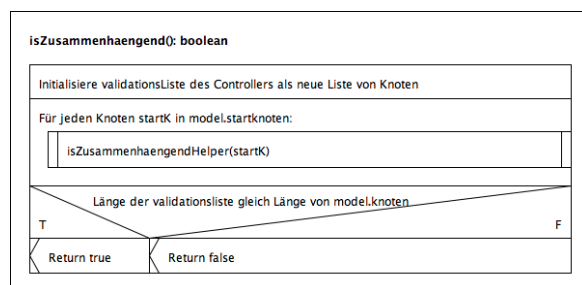
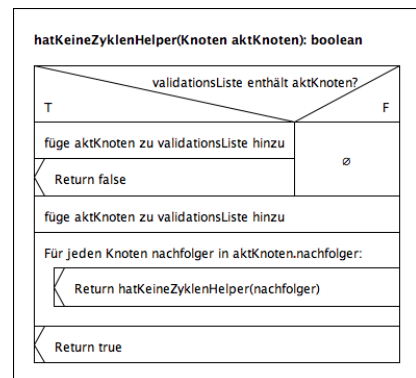
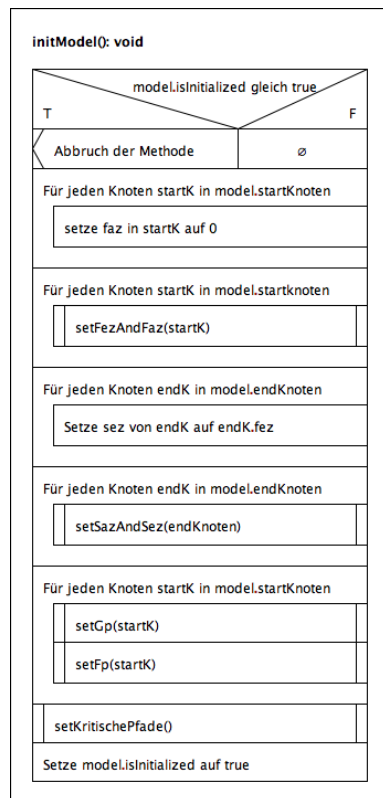


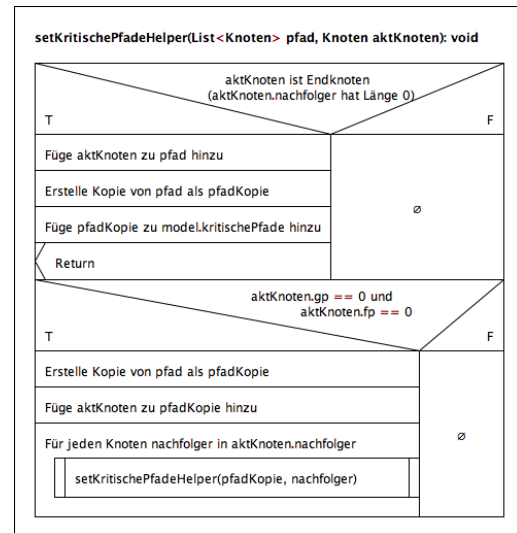
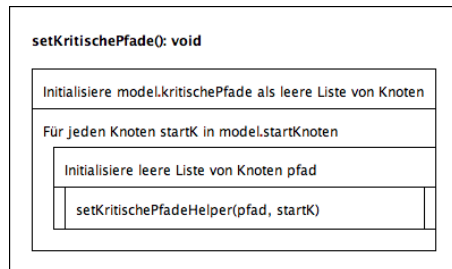


### 4.3.5 Controllermethoden



## 4 Programmkonzeption





## 5 Abweichung von der handschriftlichen Ausarbeitung

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

### 5.1 Datenmodell

*und Klassennamen*

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. *(vgl. Kapitel ...)*

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

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

- und somit...*
- `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.

### 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.

## 5 Abweichung von der handschriftlichen Ausarbeitung

---

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

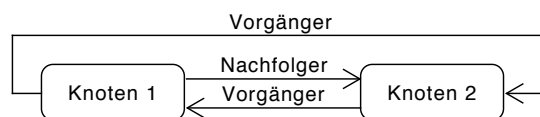
Sie 1  
Zum Testen der korrekten Funktionalität des Controllers wurden grundlegende Unittests geschrieben. Unittests werden geschrieben, um die funktionalen Einzelteile von Methoden eines Programms zu testen. Unittests gehören zur Gruppe der White-Box-Tests, also zur Gruppe der Tests, die mit Kenntnissen über die innere Funktionsweise des zu testenden Systems ablaufen.

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

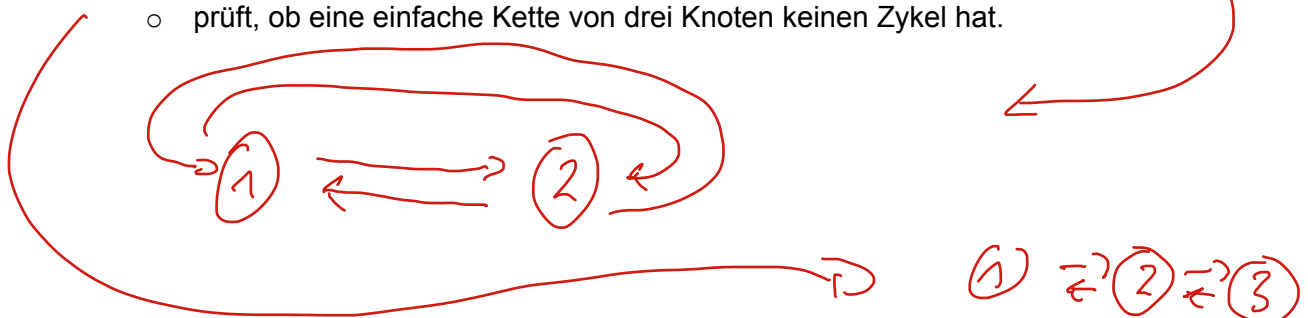
### 6.1 Prüfung der Methode `hatKeineZyklen()`

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 keinen Zykel hat.





### 6.2 Prüfung der Methode `isZusammenhaengend()`

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

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

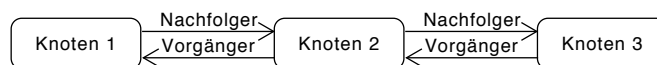


### 6.3 Prüfung der Methode `hatGueltigeReferenzen()`

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



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



## 7 Blackbox- Testfälle

Die in diesem Kapitel beschriebenen Testfälle werden nach dem 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 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

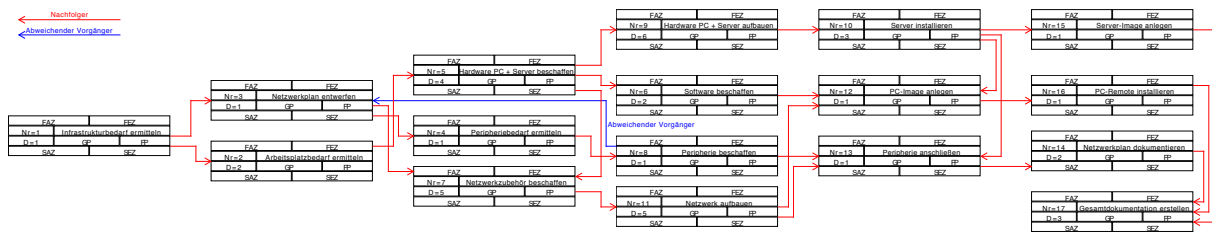
Das IHK-Beispiel Nummer 2 („*Wasserfallmodell*“) aus der verbesserten Aufgabenstellung war fehlerhaft. Das bedeutet, dass sie nicht das angegebene Ergebnis lieferte, da die Eingabe falsch formuliert war. Die Vorgangsnummer 6 („*Einsatz und Wartung*“) referenziert auf einen nichtexistierenden Knoten mit 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

## 7 Blackbox- Testfälle

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

*← Disjunktion*

## 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; -
```

## 7 Blackbox- Testfälle

### 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

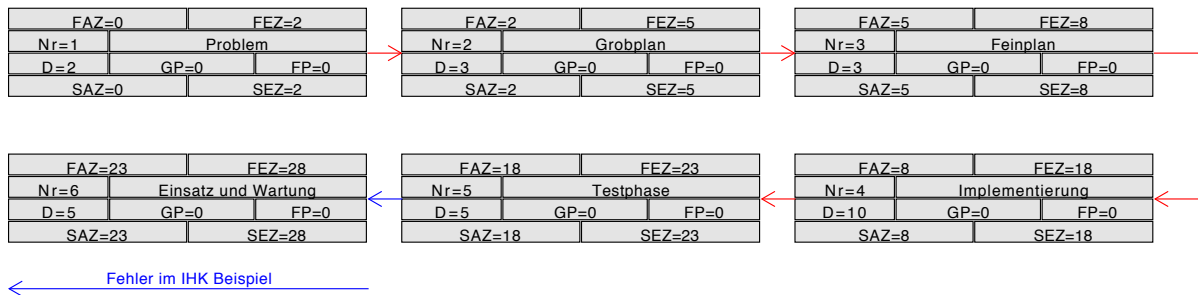
Gesamtdauer: 28

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: <sup>eines</sup> Komplexes Beispiels

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.

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; -
```

## 7 Blackbox- Testfälle

### 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 *Sonderfall* Negative Vorgangsnummern

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

Die Eingabe wird erfolgreich eingelesen und korrekt ausgewertet und ausgegeben.

#### Eingabe

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

#### Ausgabe

```
Negative Vorgangsnummer

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

Anfangsvorgang: -1
Endvorgang: 7
Gesamtdauer: 39

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



## 7 Blackbox- Testfälle

---

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

#### Eingabe

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

#### Ausgabe

Beispiel 4 mit Zyklus

Berechnung nicht möglich.

Zyklus erkannt: 3->4->3

## 7.4.2 Eigene Fehlerfälle

### 7.4.2.1 Fehlerhafte Referenz

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

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

#### Eingabe

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

#### Ausgabe

Berechnung nicht möglich.

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

### 7.4.2.2 Mehrfache gleiche Vorgangsnummern

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

In Datei F\_Mehrfache\_Vorgangsnummern.in: Ungenügende Eingabe: Es kommt mindestens eine Vorgangsnummer mehrfach vor.

#### Eingabe

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

## 7 Blackbox- Testfälle

---

### 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.

#### 7.4.2.6 Nicht zusammenhängend

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

Testfall nicht Zusammenhängend: Fehler (Nicht zusammenhängend)

### Eingabe

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

## 7 Blackbox- Testfälle

---

5; Testphase; 5; 4; 6

6; Einsatz und Wartung; 5; 5; -

### Ausgabe

Testfall nicht Zusammenhängend

Berechnung nicht möglich.

Nicht zusammenhängend.

## 8 Zusammenfassung und Ausblick

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

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

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

### 8.1 Ausblick

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

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



## 9 Anhang: Programmcode