

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>3</b>
<b>2</b>	<b>Benutzeranleitung .....</b>	<b>4</b>
2.1	Systemvoraussetzungen und Hinweise zum Aufruf .....	4
2.2	Installation des Programms .....	4
2.3	Programmstart .....	4
<b>3</b>	<b>Aufgabenanalyse .....</b>	<b>6</b>
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
<b>4</b>	<b>Programmkonzeption .....</b>	<b>11</b>
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 .....	13
4.3.4	Ausgabe .....	14
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 Model .....	17
5.1.3	Klasse Knoten .....	17
5.1.4	Klasse Controller .....	17

5.1.5	Abstrakte Klasse Ausgabe .....	17
5.2	[...] .....	18
5.3	[...] .....	18
<b>6</b>	<b>Testfälle .....</b>	<b>19</b>
6.1	Normalfälle .....	19
6.1.1	IHK_Beispiel.txt .....	19
<b>7</b>	<b>Zusammenfassung und Ausblick .....</b>	<b>22</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.

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 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.

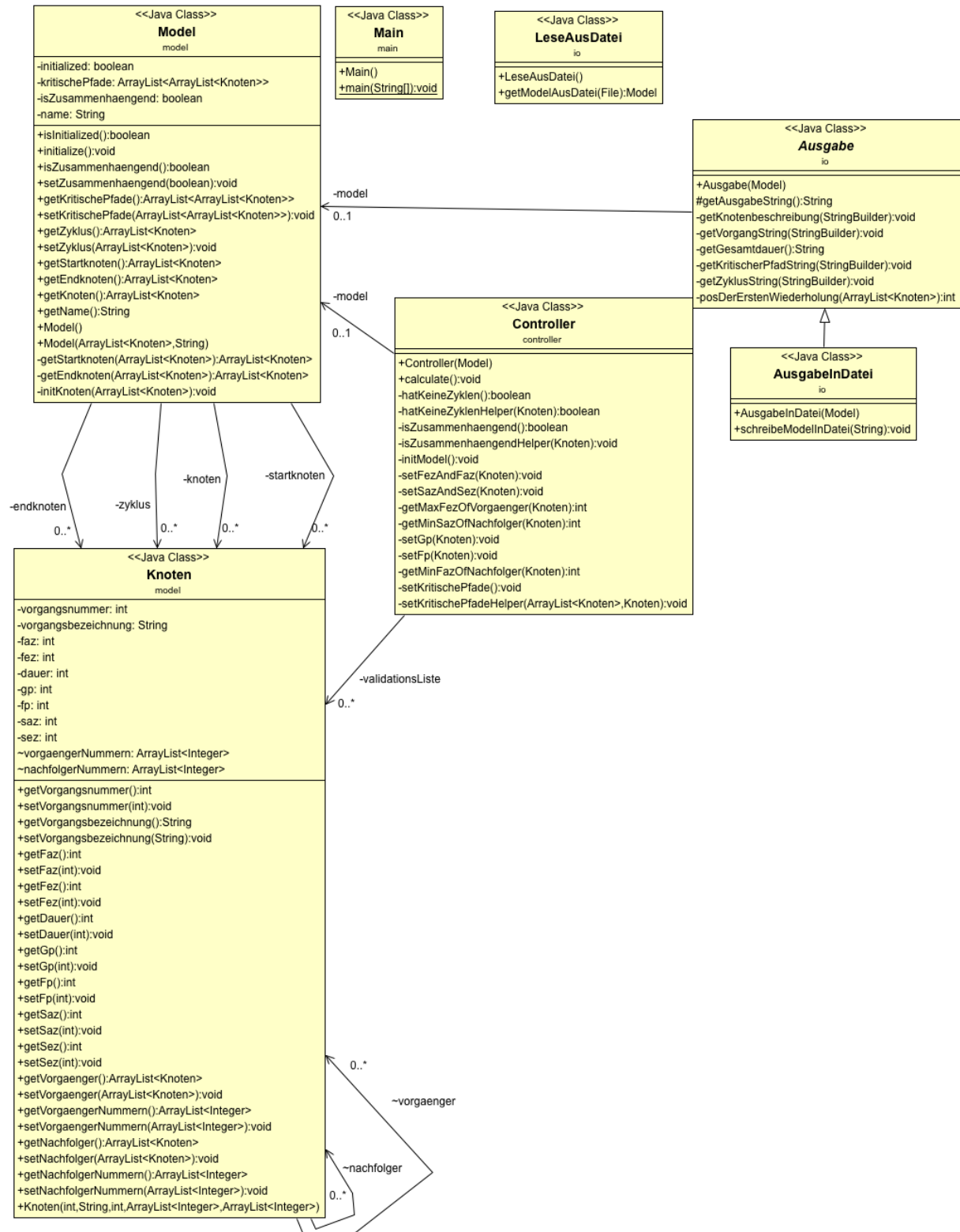
$GP = SAZ - FAZ = SEZ - FEZ$  und  $FP = (\text{kleinster FAZ der nachfolgenden Knoten}) - FEZ$

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:

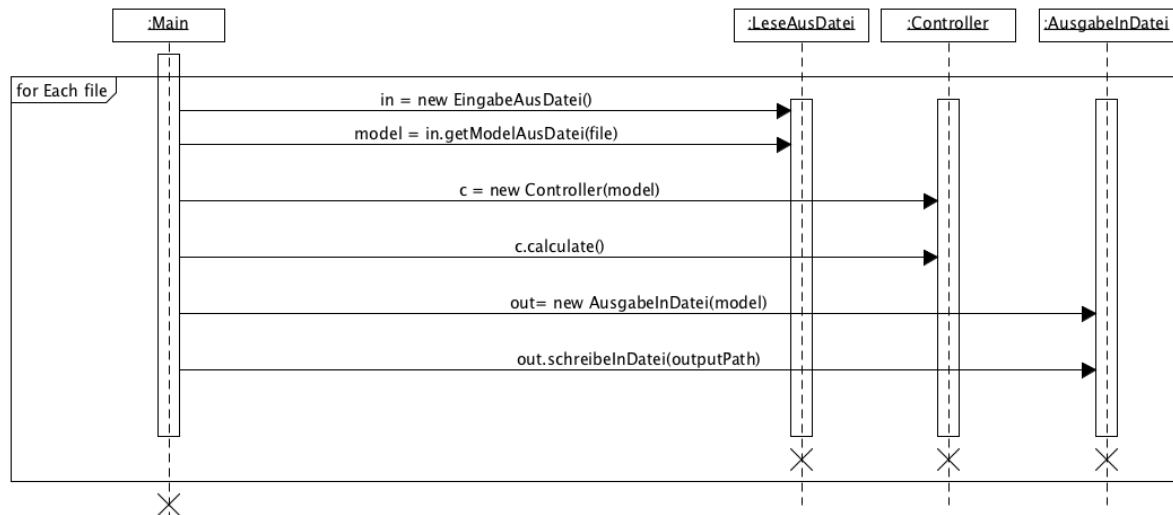
Diese prüft zunächst die Abbruchbedingung, dass der aktuell in der Hilfsmethode betrachtete Knoten ein Endpunkt ist. Ist dies der Fall, wird der berechnete Pfad im externen Modell zu einer Liste hinzugefügt und die Methode beendet. Ansonsten wird geprüft, ob der aktuelle Knoten das Kriterium für einen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

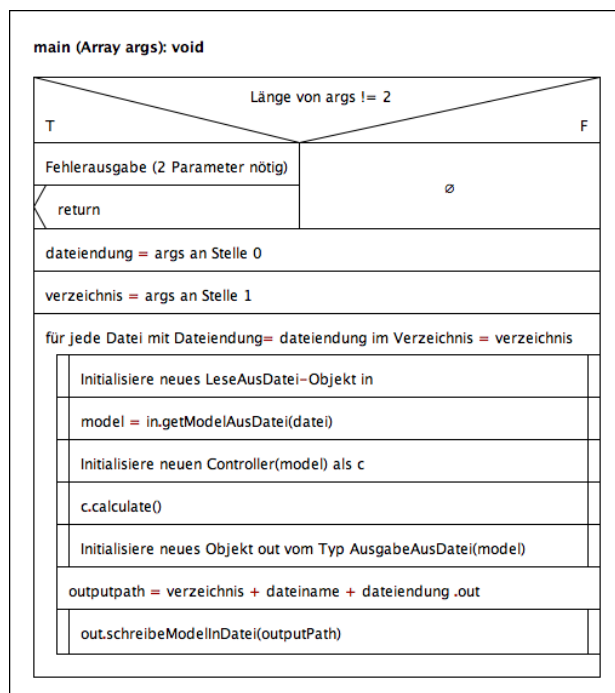


## 4.2 Programmablauf im Sequenzdiagramm

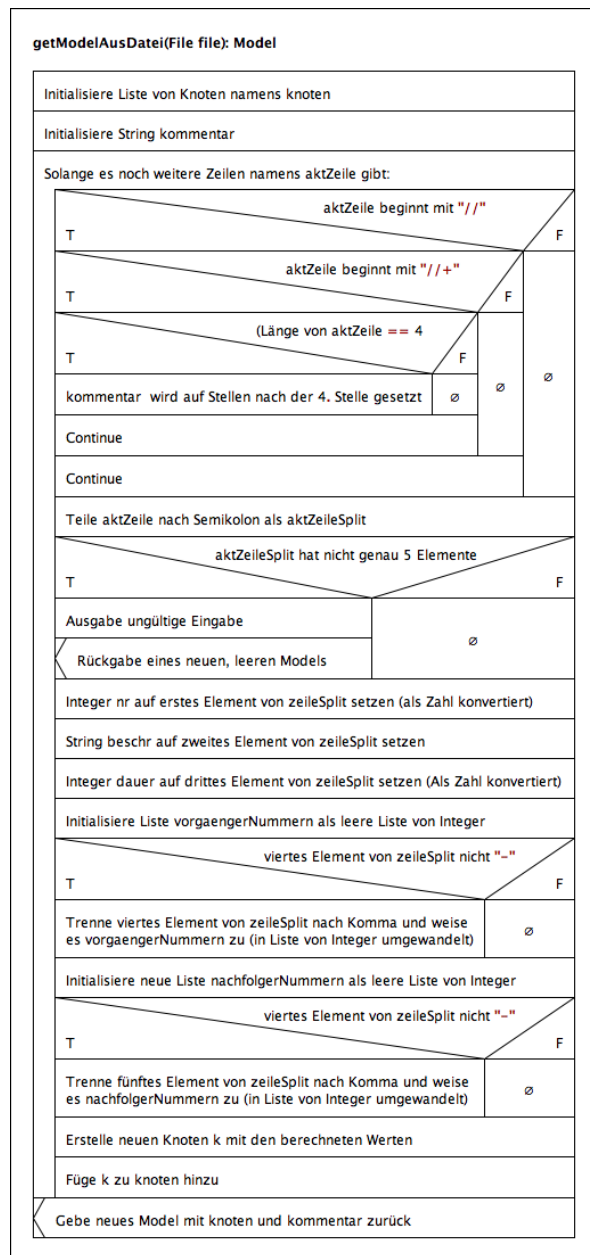


## 4.3 Nassi-Shneiderman-Diagramme

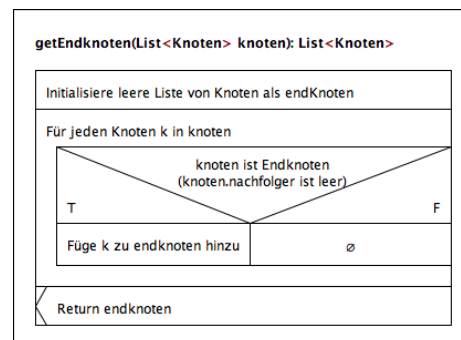
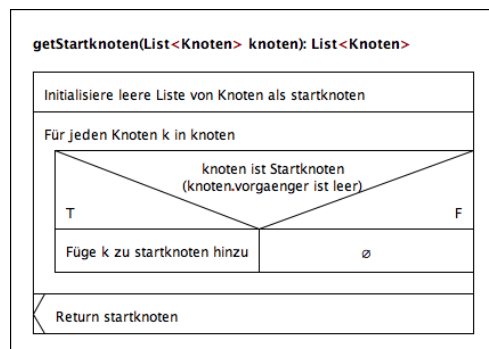
### 4.3.1 Main

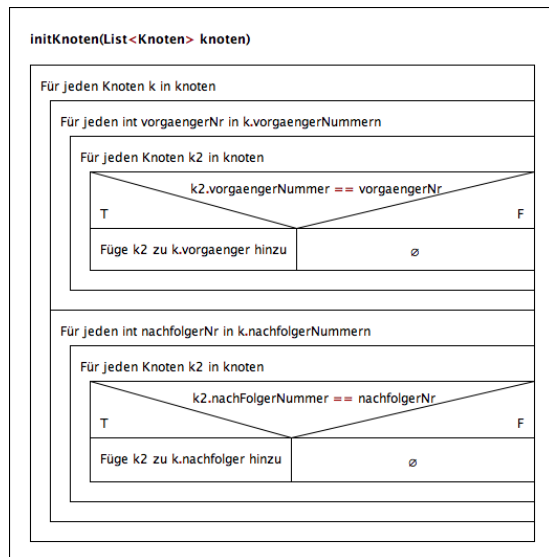


## 4.3.2 Einlesen einer Datei



## 4.3.3 Erzeugung des Models

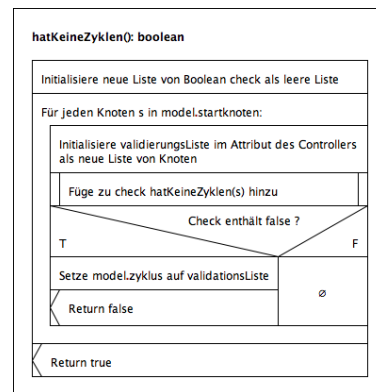
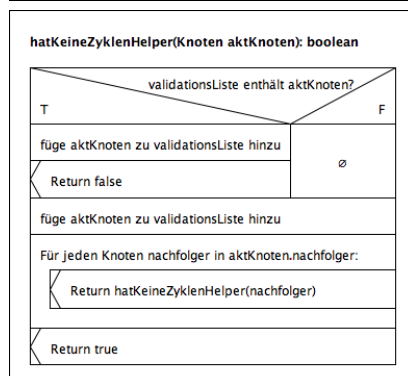
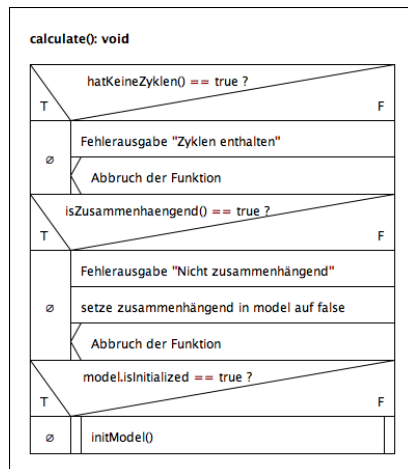




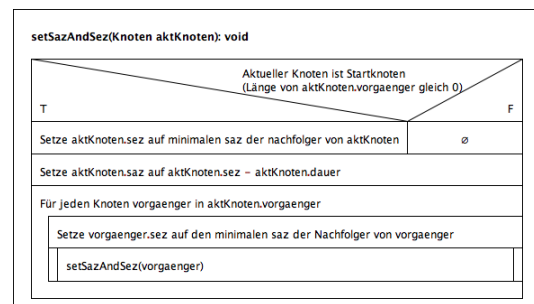
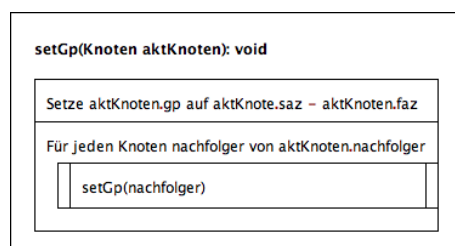
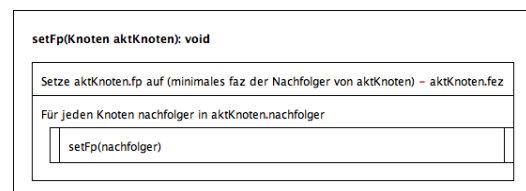
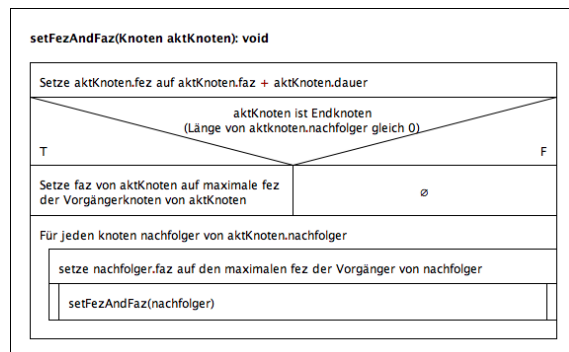
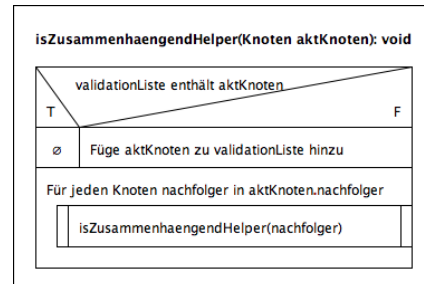
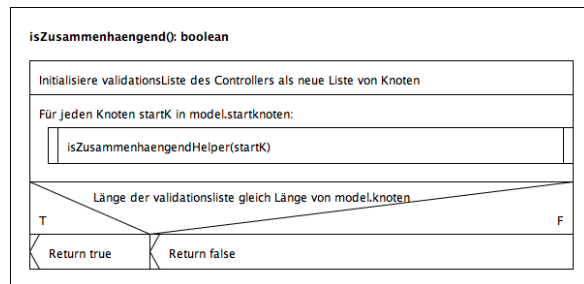
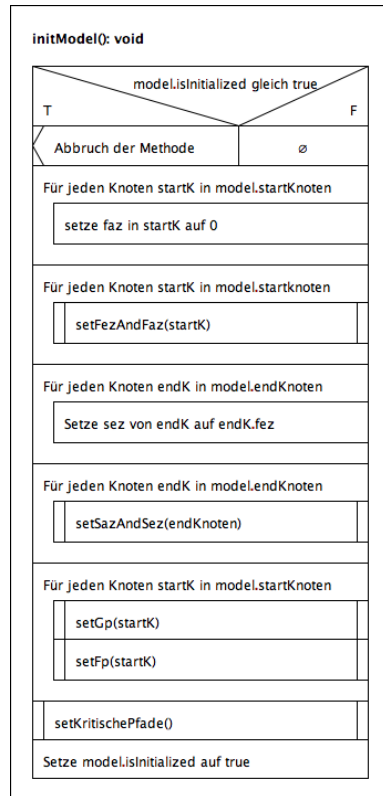
#### 4.3.4 Ausgabe

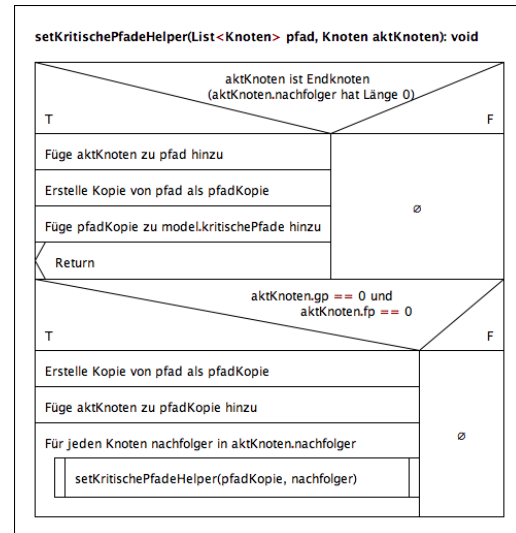
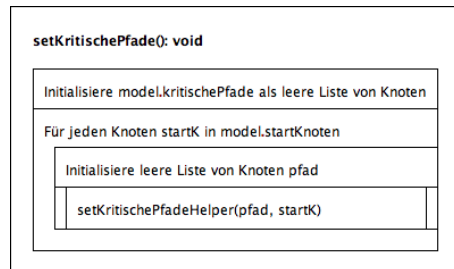
[...]

#### 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.

#### 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` prüft, ob ein im Model gekapselter Graph Zykelfrei ist. Eine weitere Hilfsmethode `hatKeineZyklenHelper` ermöglicht die Überprüfung der Zykelfreiheit mittels Backtracking.
- `istZusammenhaengend` prüft, ob ein Graph zusammenhängend ist. Hier ermöglicht ebenfalls eine Helper-Methode namens `istZusammenhaengendHelper` die Überprüfung mittels Backtracking.

#### 5.1.5 Abstrakte Klasse Ausgabe

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

5.2 [...]

5.3 [...]

## 6 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 Formatierung der Eingabedatei nicht gültig ist, jedoch das Programm dennoch zu einem gültigen Ergebnis kommt und somit bewertet werden kann.

Unter Fehlerfällen werden die Fälle verstanden, die im Fehlerprotokoll abgebildet werden und dazu führen, dass das Programm nicht die gewünschten Ausgaben produziert.

### 6.1 Normalfälle

#### 6.1.1 IHK\_Beispiel.txt

##### 6.1.1.1 Eingabe

[...]

## 6.1.1.2 Ausgabe

A small yellow square icon containing three black dots, positioned in the top-left corner of a large rectangular frame.

#### 6.1.1.3 Diskussion

Durch Vergleich mit dem Beispiel ist festzustellen, dass die Ergebnisse [...]

## 7 Zusammenfassung und Ausblick

[...]