Dokumentation der Praktischen Arbeit zur Prüfung zum Mathematisch-technischen Softwareentwickler

Thema: Automatisierte Erstellung von Netzplänen

17. Mai 2018

Christian Peters

Prüfungs-Nummer:	20613
Programmiersprache:	Java
Ausbildungsort:	Institut für Kernphysik Forschungszentrum Jülich

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Kunst einer erfolgreichen zeitlichen Planung von Projekten ist es, die einzelnen Schritte bis zum Ziel optimal aufeinander abzustimmen. Kommt es in einer Kette von Aufgaben auch nur an einer Stelle zu Verzögerungen, kann dies das gesamte Projekt gefährden. Damit Projektmanager auch in schwierigen Situationen stets den Überblick behalten können, gibt es einige Werkzeuge, die bei der Planung großer Projekte von besonderer Bedeutung sind. Eines dieser Werkzeuge soll nun im Rahmen dieser Arbeit näher untersucht werden: Der Netzplan.

1.1 Was ist ein Netzplan?

Ein Netzplan dient der Darstellung aller Abhängigkeiten zwischen den einzelnen Vorgängen eines Projektes. Ein Vorgang innerhalb eines Projektes lässt sich hierbei als einzelner Knoten in einer verzweigten Kette aus weiteren Vorgängen auffassen, die alle zusammen wiederum das gesamte Projekt repräsentieren. Die Abhängigkeiten unter den Vorgängen werden hierbei durch Pfeile ausgedrückt: Ein Vorgänger besitzt einen Pfeil hin zu seinem Nachfolger. Jeder Vorgang verfügt hierbei über bestimmte Eigenschaften, die ihn eindeutig beschreiben:

- **Vorgangsnummer:** Eine eindeutige Identifikationsnummer des Vorgangs
- **Vorgangsbezeichnung:** Eine lesbare Bezeichnung des Vorgangs
- **Dauer:** Wie lange dauert die Ausführung des Vorgangs?

- **Vorgänger:** Welche anderen Vorgänge müssen zuerst ausgeführt werden, bevor der aktuelle Vorgang ausgeführt werden kann?

- **Nachfolger:** Welche anderen Vorgänge warten auf eine Abarbeitung des aktuellen Vorgangs?

- **Frühester Anfangszeitpunkt (FAZ):** Wann kann frühestens mit der Bearbeitung des Vorgangs begonnen werden?

- **Spätester Anfangszeitpunkt (SAZ):** Wann muss spätestens mit der Bearbeitung begonnen werden, damit der Zeitplan nicht gefährdet wird?

- **Frühester Endzeitpunkt (FEZ):** Wann kann der Vorgang frühestens abgeschlossen werden?

- **Spätester Endzeitpunkt (SEZ):** Wann muss der Vorgang spätestens abgeschlossen sein, damit der Zeitplan nicht gefährdet wird?

- **Gesamtpuffer (GP):** Wie viel Spielraum liegt zwischen dem frühesten und dem spätesten Anfangszeitpunkt?

- **Freier Puffer (FP):** Wie viel Spielraum existiert, wenn kein nachfolgender Vorgang durch den aktuellen Vorgang verzögert werden soll?

Wichtig bei der Angabe der Vorgänger und Nachfolger ist, dass ein Netzplan keine zyklischen Abhängigkeiten enthalten darf (sonst könnte sich die Dauer eines Projektes bis ins Unendliche hinziehen). Zudem muss es stets mindestens einen Vorgang ohne Vorgänger (einen *Startvorgang*) und mindestens einen Vorgang ohne Nachfolger (einen *Endvorgang*) geben.

Da zu Beginn der Planungsphase im Normalfall noch nicht alle aufgelisteten Attribute bekannt sind (i.d.R. nur Vorgangsnummer, Vorgangsbezeichnung, Dauer, sowie Vorgänger und Nachfolger), ist es die Aufgabe des Projektmanagers, die restlichen Eigenschaften so zu bestimmen, dass der Zeitplan eingehalten wird. An dieser Stelle kommt die Rolle des Netzplans zum tragen: Durch die Verkettung der einzelnen Vorgänge mit ihren Vorgängern und Nachfolgern können die fehlenden Attribute (FAZ, SAZ, FEZ, SEZ, GP, FP) systematisch berechnet werden. Hieraus ergibt sich auch unmittelbar die Gesamtdauer des Projektes: Sie entspricht entweder dem FEZ der Endvorgänge oder ist undefiniert, falls dieser nicht eindeutig ist.

1.2 Kritische Pfade

In jedem Projekt gibt es immer auch Vorgänge, deren Abarbeitung an besonders enge Fristen geknüpft ist und daher keinerlei Spielraum erlaubt. Solche Vorgänge, bei denen entsprechend der obigen Notation $GP = 0$ und $FP = 0$ gilt, werden im Folgenden als *kritische Vorgänge* bezeichnet. In jedem Projekt gibt es mindestens eine Kette von Vorgängen beginnend bei einem Startvorgang hin zu einem Endvorgang, die ausschließlich aus kritischen Vorgängen besteht. Eine derartige Konstellation wird auch als *kritischer Pfad* bezeichnet, da eine reibungslose Abarbeitung dieser Vorgänge Grundvoraussetzung für die Einhaltung des Zeitplans ist. Der Projektmanager sollte also schon zu Beginn alle kritischen Pfade identifizieren und besonders auf die termingerechte Abarbeitung der zugehörigen Vorgänge achten. Auch bei dieser Aufgabe hilft ihm der Netzplan, der sich systematisch nach kritischen Pfaden durchsuchen lässt.

1.3 Ziel dieser Arbeit

Das Ziel dieser Arbeit ist es nun, die Erstellung von Netzplänen sowie die Suche nach kritischen Pfaden zu automatisieren. Das zu diesem Zweck zu entwickelnde Programm soll in der Lage sein, die einzelnen Vorgänge eines Projektes aus einer Eingabedatei einzulesen und hieraus einen korrekten Netzplan zu konstruieren. Die fehlenden Eigenschaften der Vorgänge (FAZ, SAZ, FEZ, SEZ, GP, FP) sollen dazu systematisch errechnet werden. Anschließend soll der Netzplan auf kritische Pfade durchsucht werden, die Ergebnisse der Suche, sowie die vollständige Beschreibung aller Vorgänge und ihrer Fristen inklusive der Gesamtdauer des Projektes, sollen abschließend in einer Ausgabedatei gespeichert werden.

Kapitel 2

Verfahrensbeschreibung

2.1 Datenstrukturen

Die Grundlage eines jeden Programms ist durch die Art der verwendeten Datenstrukturen gegeben. Damit im weiteren Verlauf Netzpläne effizient erzeugt werden können, ist eine solide logische Repräsentation unerlässlich. Die grundlegende Darstellung der wichtigsten Elemente dieses Programms soll nun im Folgenden näher beschrieben werden.

2.1.1 Logische Repräsentation eines Vorgangs

Ein einzelner Vorgang kann als Objekt dargestellt werden, welches die in 1.1 beschriebenen Attribute *Vorgangsnummer, Vorgangsbezeichnung, Dauer, Vorgänger, Nachfolger, FAZ, SAZ, FEZ, SEZ, GP, FP* enthält und einen Zugriff auf diese ermöglicht. Hierbei werden jedem Attribut die folgenden Definitionsbereiche zugeordnet, welche später vom Anwender eingehalten werden müssen:

- **Vorgangsnummer:** Element der ganzen Zahlen $\mathbb{Z}$, welches im gesamten Projekt *eindeutig* sein muss.
- **Vorgangsbezeichnung:** Nicht leere Zeichenkette, welche das Semikolon nicht enthalten darf[1].
- **Dauer:** Element der natürlichen Zahlen $\mathbb{N}$. Dies bedeutet, dass keine negativen Dauern und auch keine nicht existenten Dauern zugelassen sind.

[1] Dies hängt mit dem Eingabeformat zusammen, was an späterer Stelle erläutert wird.

- **Vorgänger:** Liste gültiger Vorgangsnummern.

- **Nachfolger:** Liste gültiger Vorgangsnummern.

- **FAZ, SAZ, FEZ, SEZ, GP und FP:** Elemente der natürlichen Zahlen $\mathbb{N}$ *inklusive* der 0.

Mehrere Vorgänge, welche zusammen ein Projekt bilden, können somit in einer Liste aus einzelnen Vorgangsobjekten abgelegt werden.

2.1.2 Logische Repräsentation eines Netzplans

Ein Netzplan kann formal als gerichteter azyklischer Graph beschrieben werden, wobei die Knotenmenge durch die Menge aller Vorgänge des Projektes gegeben ist und die Kantenmenge durch die Vorgänger-Nachfolger-Relation unter den Vorgängen beschrieben werden kann. Ein Hilfsmittel, welches zusätzlich zu der Liste aller Vorgänge des Projekts verwendet wird, um den Netzplan als Graph darzustellen, ist die sogenannte *Adjazenzmatrix*. Diese Matrix A besitzt für jeden Knoten des Graphen sowohl eine Zeile als auch eine Spalte und hat daher bei n Knoten die Dimension $n \times n$. Existiert eine direkte Verbindung von Knoten i nach Knoten j (dies ist anschaulich der Fall, wenn Vorgang j Nachfolger von Vorgang i ist), so ist der jeweilige Eintrag a_{ij} der Adjazenzmatrix 1, ansonsten 0. Anhand der Adjazenzmatrix lässt sich auch leicht erkennen, ob ein Knoten Start- oder Endknoten ist: Da ein Startknoten i keinen Vorgänger hat, gilt für die entsprechenden Einträge der i'ten Spalte: $a_{ki} = 0 \forall 1 \leq k \leq n$. Für jeden Endknoten j gilt analog $a_{jk} = 0 \forall 1 \leq k \leq n$.

Es sei an dieser Stelle schon angemerkt, dass Vorgänge in ihrer Nummerierung nicht zwingend die Zahlen 1 bis n (oder später im Rechner 0 bis $n-1$) belegen müssen. Daher wird zusätzlich eine Abbildung $g : \textit{Vorgangsnummern} \rightarrow \mathbb{N}$ eingeführt, welche die Vorgangsnummern lückenlos auf die ersten n natürlichen Zahlen abbildet. Dementsprechend bezeichnet g^{-1} die zugehörige Umkehrabbildung, welche eine Zahl von 1 bis n auf die entsprechende ursprüngliche Vorgangsnummer abbildet.

2.2 Einlesen der Vorgänge

2.2.1 Format der Eingabedatei

Das Format, in dem die Daten dem Programm zugeführt werden, soll hier anhand des folgenden Beispiels erläutert werden:

```
//*********************************************
//+ 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; -
```

Abbildung 2.1: Beispiel einer Eingabedatei

Eine Datei besteht aus einer ungeordneten Menge an Kommentar- und Datenzeilen. Leerzeilen sind *nicht* zugelassen. Kommentarzeilen lassen den Anwender Anmerkungen hinzufügen, welche vom Programm nicht beachtet werden. Sie beginnen stets mit einem `//`. Eine besondere Art der Kommentarzeile ist die *Überschriftzeile*, da sie im Gegensatz zu den normalen Kommentarzeilen vom Programm besonders berücksichtigt wird. Sie beginnt immer mit einem `//+` und kann an beliebiger Stelle innerhalb der Datei auftauchen. Die Zeichenkette, welche auf das einleitende `//+` folgt, beschreibt das jeweilige Projekt und darf nicht leer sein. Pro Datei muss es *genau eine* Überschriftzeile geben. Im gegebenen Beispiel sind die Zeilen 1, 3 und 4 klassische Kommentarzeilen und werden vom Programm nicht beachtet. Zeile 2 ist die Überschriftzeile.

Datenzeilen liefern dem Programm die notwendigen Informationen zu den einzelnen Vorgängen eines Projekts. Jede Datenzeile beschreibt genau einen Vorgang und besteht aus fünf Elementen, welche jeweils durch ein Semikolon getrennt werden: *Vorgangsnummer, Vorgangsbezeichnung, Dauer, Vorgänger und Nachfolger*[2]. Die Eingaben zu Vorgangsnummer, Vorgangsbezeichnung und Dauer müssen hierbei mit den Definitionsbereichen der in 2.1.1 beschriebenen Attribute eines Vorgangs übereinstimmen. Die Elemente Vorgänger und Nachfolger werden jeweils durch gültige Vorgangsnummern angegeben,

[2]Dies ist auch der Grund, weshalb im Definitionsbereich der Vorgangsbezeichnung keine Semikola zugelassen sind.

welche durch Kommata getrennt aufgezählt werden. Doppelte Vorgänger oder Nachfolger sind nicht erlaubt. Hat ein Vorgang keinen Vorgänger bzw. keinen Nachfolger, so steht an dieser Stelle das Zeichen −. Es handelt sich in diesem Fall um einen Start- oder Endvorgang. Jedes Projekt muss mindestens einen Start- und einen Endvorgang enthalten.

2.2.2 Algorithmus zum Einlesen

Der Algorithmus, der zum Einlesen der Eingabedatei implementiert wurde, iteriert im Wesentlichen über alle Zeilen und prüft, um was für eine Art der Zeile es sich handelt. Je nach Art der Zeile werden die Eingaben validiert, stimmt etwas nicht, so wird ein Fehler geworfen. Während der Iteration wird sukzessive eine Liste mit gelesenen Vorgängen befüllt, die am Ende zurückgegeben wird. Diese Liste wird im folgenden Struktogramm (siehe Abbildung 2.2), welches einen Überblick über die wesentlichen Gesichtspunkte des Algorithmus gibt, als *Resultat* bezeichnet.

2.3 Konstruktion des Netzplans

Um aus der eingelesenen Liste aus Vorgangsobjekten nun den Netzplan konstruieren zu können, sind einige Schritte notwendig, die an dieser Stelle näher beschrieben werden.

2.3.1 Initialisierung

Während der Initialisierungsphase wird zunächst geprüft, ob alle Grundvoraussetzungen erfüllt sind, damit es sich um einen gültigen Netzplan handelt. Hierzu wird zunächst die Adjazenzmatrix erzeugt und währenddessen die Konsistenz der Vorgänger - Nachfolger - Relation sichergestellt.

Erzeugung der Adjazenzmatrix

Um die Adjazenzmatrix zu erzeugen, wird jede Vorgangsnummer einer Spalte in der Matrix zugeordnet. Hierzu wird die Abbildung g verwendet, die bereits bei der Beschreibung des Netzplans erläutert wurde. Man erhält g, indem man ein assoziatives Feld anlegt, wobei jeder Vorgangsnummer die zugehörige Position in der Liste aus Vorgängen zugeordnet wird. Für jeden Vorgang wird nun in der zugehörigen Zeile der Adjazenzmatrix in der Spalte jedem seiner Nachfolger eine 1 in der Matrix eingetragen. Hierbei wird auch

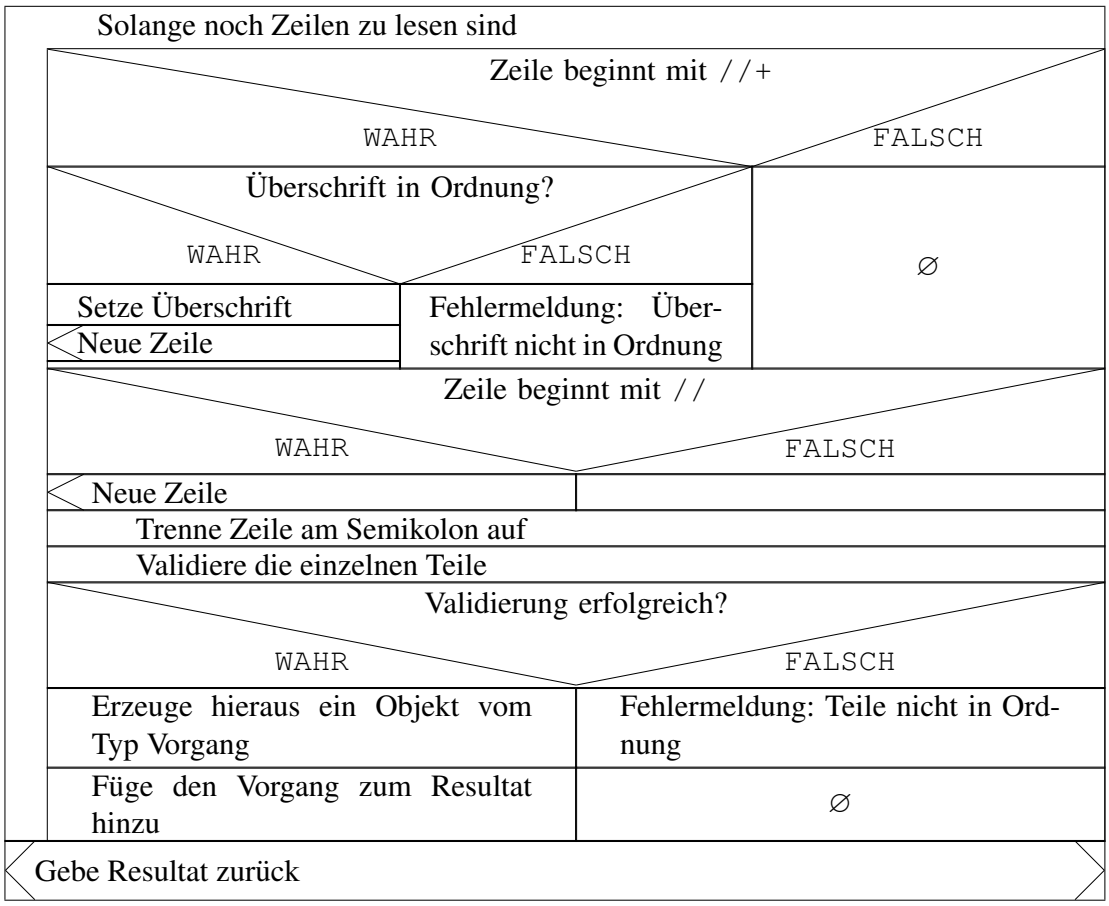

Abbildung 2.2: Algorithmus zum Einlesen der Eingabedatei

gleich abgeprüft, ob die Beziehung eines Vorgangs zu seinen Vorgängern und Nachfolgern konsistent ist: Ist z.B. ein Vorgang *a* als Vorgänger von Vorgang *b* eingetragen, hat diesen aber nicht als Nachfolger, so ist die Beziehung inkonsistent und der Netzplan kann nicht erzeugt werden. Das Programm wird in diesem Fall mit einem Fehler beendet.

Hängt der Graph zusammen?

Nachdem die Adjazenzmatrix erzeugt wurde und die Konsistenz der Beziehungen unter den Vorgängen sichergestellt ist, muss im nächsten Schritt geprüft werden, ob der Graph auch zusammenhängt, da es sich nur in diesem Fall um einen gültigen Netzplan handelt. Um dies zu bewerkstelligen, wird der bis dahin gerichtete Graph vorübergehend als ungerichtet aufgefasst. Beginnend bei einem beliebigen Knoten wird der Graph im nächsten Schritt traversiert. Bleiben nach dem Durchlaufen noch Knoten übrig, welche nicht be-

sucht wurden, war der Graph nicht zusammenhängend.

Um aus dem gerichteten Graph einen ungerichteten Graph zu erzeugen, kann einfach eine symmetrische Version der Adjazenzmatrix erstellt werden. Hierzu wird zu jedem Element $a_{ij} = 1$ das an der Diagonalen gespiegelte Element a_{ji} ebenfalls auf 1 gesetzt. Die Traversierung, welche im nächsten Schritt erfolgt, wird mittels einer Breitensuche durchgeführt.

Überprüfen auf Zyklen

Da ein gültiger Netzplan keine Zyklen enthalten darf, muss nun geprüft werden, ob der vorliegende Graph auch azyklisch ist. Hierzu wird ausgehend von jedem Startknoten eine Expansion des Graphen gebildet (d.h. es werden alle möglichen Pfade durch den Graphen erzeugt). Sobald ein Pfad im nächsten Schritt der Expansion ein Element aufnehmen würde, welches bereits in diesem Pfad enthalten ist, wurde ein Zyklus entdeckt und das Programm kann mit einem Fehler beendet werden. Auch hier wird die zu diesem Zweck benötigte Traversierung mit einer Breitensuche durchgeführt.

2.3.2 Vorwärtsrechnung

Nachdem in den letzten Abschnitten sichergestellt wurde, dass aus der Liste an Vorgängen ein gültiger Netzplan erzeugt werden kann, müssen nun die noch fehlenden Einträge FAZ, SAZ, FEZ, SEZ, GP und FP für jeden Vorgang berechnet werden. In der Phase der Vorwärtsrechnung werden für jeden Vorgang die Werte FAZ und FEZ gesetzt, weiterhin kann nach dieser Phase auch die Dauer des Projektes abgelesen werden. Diese entspricht nämlich wie in Abschnitt 1.1 beschrieben dem FEZ der Endvorgänge, insofern dieser eindeutig ist.

Um die Vorwärtsrechnung durchzuführen, wird eine Warteschlange *(Queue)* mit abzuarbeitenden Knoten erzeugt, welche zunächst nur die Startknoten enthält. Solange diese Queue noch Elemente hat, wird in jedem Schritt das erste Element bearbeitet. Hierbei wird zuerst der FEZ des Elements auf FAZ+Dauer gesetzt. Man beachte hierbei, dass Startknoten von Beginn an einen FAZ von 0 haben. Anschließend werden alle Nachfolger des Elements betrachtet. Der FAZ eines jeden Nachfolgers wird nun auf den FEZ des aktuellen Elements gesetzt, wenn sich der FAZ des Nachfolgers hierdurch vergrößert. Jeder Nachfolger wird dann hinten an die Queue der zu bearbeitenden Knoten angefügt und eine neue Iteration beginnt.

2.3.3 Rückwärtsrechnung

In der Phase der Rückwärtsrechnung werden nun für jeden Knoten die Werte SAZ und SEZ gesetzt. Hierbei wird ebenfalls eine Queue mit abzuarbeitenden Knoten erzeugt, welche diese Mal nur die Endknoten enthält. Bei jedem Endknoten wird nun SEZ=FEZ gesetzt, da Endknoten keinen Puffer haben. Die Queue wird nun wieder beginnend mit dem ersten Element abgearbeitet, bis sie leer ist. Hierbei wird zunächst der SAZ des aktuellen Elementes auf SEZ-Dauer gesetzt. Im Anschluss werden alle Vorgänger des Elementes betrachtet. Wurde der SEZ des Vorgängers noch nicht gesetzt oder würde er sich verringern, so wird dieser auf den SAZ des aktuellen Elements gesetzt. Anschließend wird wieder jeder Vorgänger hinten an die Queue angehängt und die Iteration beginnt von neuem.

2.3.4 Berechnung der Zeitreserven

Die Zeitreserven eines jeden Vorganges werden durch die Attribute GP und FP ausgedrückt. Um diese zu berechnen, werden alle Vorgänge durchlaufen. Hierbei wird für jeden Vorgang GP=SAZ-FAZ=SEZ-FEZ gesetzt. Für den Wert FP gilt folgender Zusammenhang: FP= $\min_{i \in \{Nachfolger\}} FAZ_i - FEZ$. Dieser wird ebenfalls für jeden Vorgang ermittelt und gesetzt.

2.4 Auffinden der kritischen Pfade

Nachdem der Netzplan nun vollständig erzeugt wurde, stellt sich noch die Frage, wie die kritischen Pfade gefunden werden können. Hierzu wird mittels einer iterativen Tiefensuche (Breitensuche ist ebenfalls möglich, im Programmsystem ändert sich hier nur eine einzige Zeile) ausgehend von jedem kritischen Startknoten jeder kritische Pfad erzeugt. Sobald ein Endknoten zu einem bis dahin kritischen Pfad hinzugefügt wird, ist ein kritischer Pfad komplett und kann dem Resultat angefügt werden. Ein Überblick dieses Algorithmus inklusive der Unterscheidung von Tiefen- und Breitensuche findet sich in Form eines Struktogramms in Abbildung 2.3.

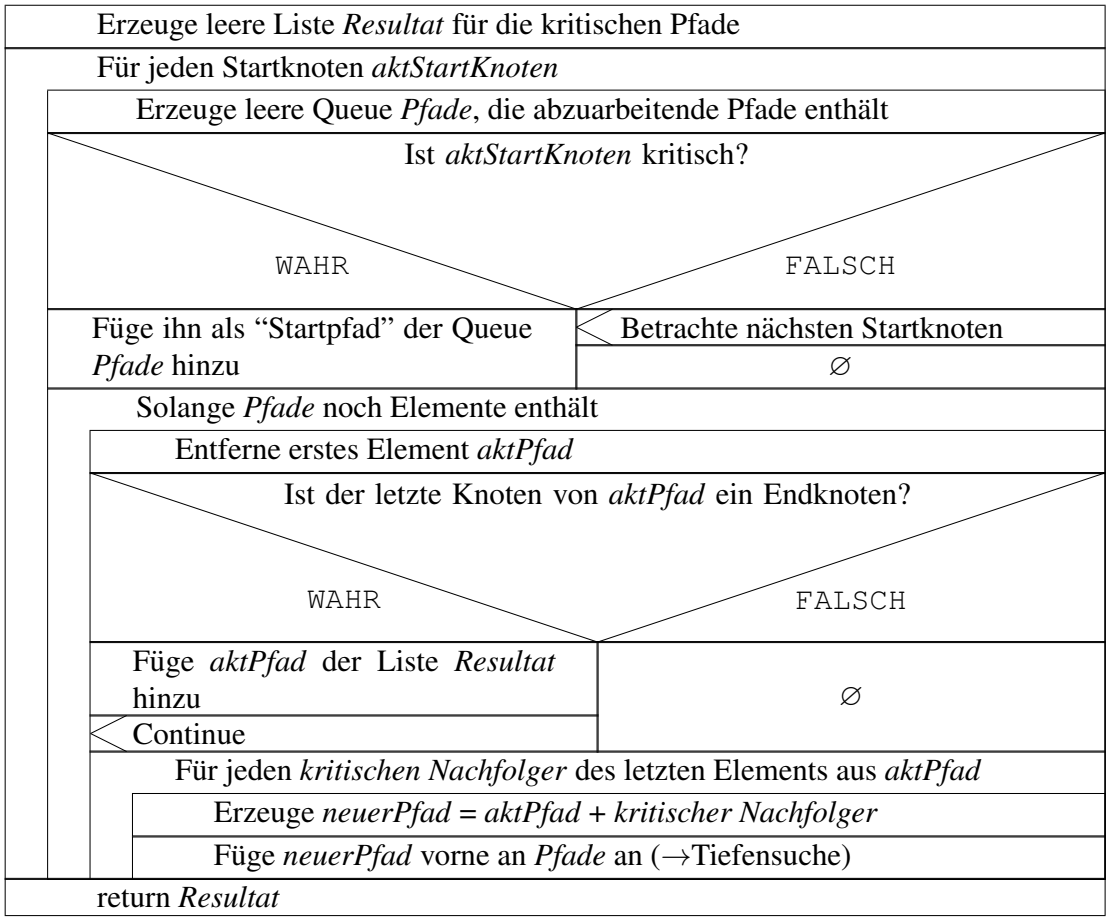

Abbildung 2.3: Algorithmus zum Finden aller kritischen Pfade. Man beachte, dass der Unterschied zur Breitensuche nur in einer Zeile liegt: Bei der Breitensuche würde *neuer-Pfad* hinten an *Pfade* angefügt.

2.5 Ausgabe der Ergebnisse

Das Format, in dem die Ergebnisse des Programms ausgegeben werden, soll hier anhand des folgenden Beispiels erläutert werden, welches gleichzeitig die Ausgabe zu der zuvor vorgestellten Eingabedatei bildet:

```
Installation von POI Kiosken

Vorgangsnummer; Vorgangsbezeichnung; 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
```

Abbildung 2.4: Die resultierende Ausgabedatei

Das Format der Ausgabedatei wird durch die Überschriftzeile eingeleitet, welche nur den Text der anfangs eingelesenen Überschrift enthält. Es folgt eine Leerzeile, woraufhin die Spaltenüberschriften des Ergebnisses ausgegeben werden. Anschließend wird jeder Vorgang mit all seinen Attributen ausgegeben (Vorgänger und Nachfolger ausgenommen). Nach einer weiteren Leerzeile schließen sich die Anfangsvorgänge, die Endvorgänge und die Gesamtdauer des Projekts an. Es folgt wiederum eine Leerzeile, woraufhin unter der entsprechenden Überschrift die kritischen Pfade untereinander ausgegeben werden.

Kapitel 3

Benutzeranleitung

3.1 Lieferumfang

Im Lieferumfang dieser Software sind folgende Elemente als Inhalt der .zip-Datei enthalten:

- **Dokumentation:** Enthält die vollständige Beschreibung des Softwaresystems sowie der unterliegenden Verfahren und ist in der Datei `Dokumentation.pdf` zu finden.
- **Das ausführbare Programm:** Das Programm in Form der ausführbaren .jar-Datei `netzplanerstellung-1.0.0.jar`.
- **Quelltext des Programms:** Der vollständige Quelltext des Programms findet sich im Ordner `src`.
- **Testfälle:** Die vollständige Sammlung aller Testfälle liegt im Ordner `testcases`.
- **Skript zum Ausführen aller Testfälle:** Liegt als Datei `run_testcases.ksh` vor.
- **Ant-Buildfile zum automatisierten Übersetzen des Quellcodes:** `build.xml`-Datei.

3.2 Ausführen des Programms

Die Ausführung des Programms kann nach dem folgenden Schema eines Kommandos erfolgen:

```
java -jar netzplanerstellung-1.0.0.jar <Eingabedatei> <Ausgabedatei>
```

Hierbei gibt `<Eingabedatei>` den Pfad zu einer gültigen Eingabedatei nach dem in Abschnitt 2.2.1 beschriebenen Format an. Wurde das Format nicht eingehalten, so beendet sich das Programm unter Ausgabe einer Fehlermeldung. `<Ausgabedatei>` gibt den Namen der Datei an, in welcher das Programm die Ergebnisse abspeichern soll.

Achtung: Diese Datei wird nur erzeugt, wenn sich das Programm fehlerfrei beendet. Im Falle eines Fehlers wird eine entsprechende Meldung auf der Kommandozeile ausgegeben, die nähere Informationen enthält.

3.3 Ausführen aller Testfälle

Um alle Testfälle hintereinander ausführen zu können, wurde dem Lieferumfang das Skript `run_testcases.ksh` beigelegt, welches mithilfe eines *Kornshell-Interpreters* ausgeführt werden kann. Der Aufruf kann folgendermaßen erfolgen:

```
./run_testcases.ksh <Programm> <Eingabeordner> <Ausgabeordner>
```

Hierbei gibt `<Programm>` den Pfad zur ausführbaren .jar-Datei an, `<Eingabeordner>` beschreibt, in welchem Ordner die Testfälle zu finden sind und `<Ausgabeordner>` gibt den Ordner an, in dem die Ergebnisdateien abgelegt werden. Optional können die Argumente von rechts nach links weggelassen werden. Die Standardwerte sind in diesem Fall:

- `<Programm> = netzplanerstellung-1.0.0.jar`
- `<Eingabeordner> = testcases`
- `<Ausgabeordner> = testcases`

Ein Aufruf von `./run_testcases.ksh` arbeitet also mithilfe dieser Standardwerte.

Die Vorgehensweise dieses Skriptes kann wie folgt beschrieben werden: Es werden zunächst alle Eingabedaten mit der Endung `.in` aus dem Ordner `<Eingabeordner>` erfasst und nacheinander dem Programm zugeführt. Zugehörig zu jeder Eingabedatei, die das Programm fehlerfrei durchlaufen konnte, wird anschließend im Ordner `<Ausgabeordner>` eine Ergebnisdatei mit der Endung `.out` erzeugt. Konnte die Ergebnisdatei nicht erzeugt werden, so wird die Fehlermeldung des Programms auf der Kommandozeile ausgegeben.

Kapitel 4

Entwicklerdokumentation

4.1 Entwicklungsumgebung

Die Entwicklung der Software wurde in folgender Umgebung vorgenommen:

Programmiersprache	:	Java
Compiler	:	javac 1.8.0_161
Rechner	:	Intel Core i7-4790 CPU @ 3.60GHz, 16GB RAM
Betriebssystem	:	Ubuntu 16.04 LTS

Weiterhin wurde zum Übersetzen der Software das Build-Management-Tool *Ant* in der *Version 1.9.6* verwendet. Das Skript `run_testcases.ksh`, welches der Hintereinanderausführung aller Testfälle dient, wurde auf Basis des *Kornshell-Interpreters* verfasst.

4.2 Aufbau der Software

4.2.1 Klassenstruktur

Die Klassenstruktur der Software wird an dieser Stelle durch das in Abbildung 4.1 gezeigte UML-Klassendiagramm beschrieben. Nähere Informationen zur Funktionsweise der einzelnen Methoden können bei Bedarf dem ausführlich kommentierten Quelltext entnommen werden.

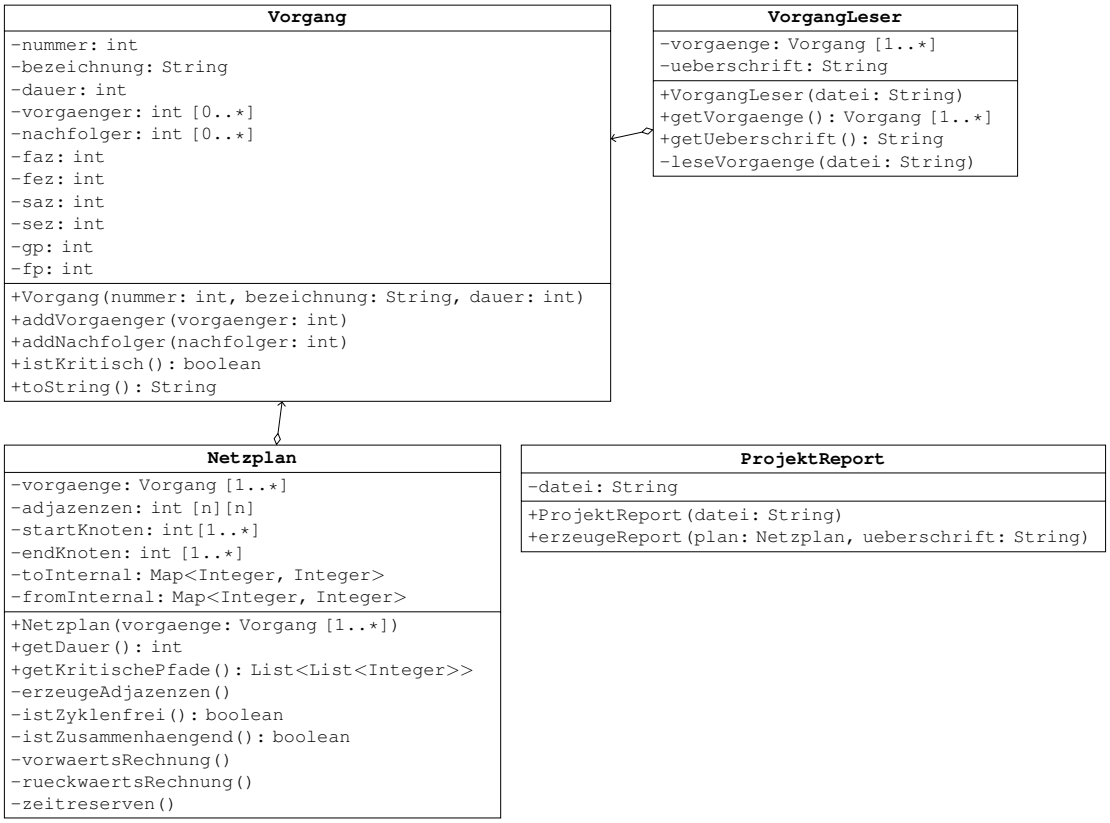

Abbildung 4.1: Der grundlegende Aufbau der verwendeten Klassen.

4.2.2 Softwarearchitektur

Der architektonische Aufbau der Software entspricht in etwa den untersten beiden Schichten einer *Drei-Schichten-Architektur*. Die *Präsentationsschicht* wurde hier bewusst nicht berücksichtigt, da während des eigentlichen Programmablaufs keine Benutzerinteraktion stattfindet. Sie kann allerdings aufbauend auf den existierenden Schichten *Datenhaltung* und *Logik* problemlos aufgesetzt werden. Die Klassen des Programms können den einzelnen Schichten wie folgt zugeordnet werden:

- **Datenhaltungsschicht:** VorgangLeser und ProjektReport sind teile der Datenhaltungsschicht, da sie für die Kommunikation des Programms mit einem persistenten Datenträger verantwortlich sind.
- **Logikschicht:** Hier sind die Klassen Vorgang und Netzplan anzusiedeln, da die tatsächlichen Berechnungen innerhalb dieser Klassen (hauptsächlich in der Klasse Netzplan) vollzogen werden.

4.3 Übersetzen der Software

Um die Software zu übersetzen, empfiehlt es sich, das Build-Management-Tool *Ant* in der genannten Version zu verwenden. Gibt man den Befehlt `ant` im Verzeichnis der `build.xml`-Datei in der Kommandozeile ein, so erzeugt Ant automatisch zwei neue Verzeichnisse: Das Verzeichnis `build`, in welchem die übersetzten Dateien abgelegt werden und das Verzeichnis `dist`, in welchem das ausführbare Programm in Form einer `.jar`-Datei erzeugt wird. Durch einen Aufruf von `ant clean` werden alle im Zuge der Übersetzung erzeugten Dateien und Verzeichnisse wieder entfernt.

4.3.1 Übersetzen ohne den Einsatz von Ant

Möchte man das Programm ohne den Einsatz von Ant übersetzen, so sollte zunächst manuell ein Verzeichnis (z.B. `build`) erzeugt werden, in welchem die übersetzten Dateien vom Compiler abgelegt werden können. Ausgehend vom Hauptverzeichnis kann nun durch den Befehl

```
javac -d build/ -sourcepath src/ src/netzplanerstellung/Main.java
```

das Programm übersetzt werden.

Erzeugen einer ausführbaren .jar-Datei ohne Ant

Um ohne den Einsatz von Ant eine ausführbare .jar-Datei erzeugen zu können, muss zunächst ein Manifest erzeugt werden, welches in jeder ausführbaren .jar-Datei vorhanden sein muss und zumindest Informationen über die Hauptklasse enthält. Es ist hierzu ausreichend, die folgenden Informationen in einer Datei `MANIFEST.MF` abzulegen:

```
1 Main-Class: netzplanerstellung.Main
2 <Leerzeile>
```

Man beachte, dass die `MANIFEST.MF` Datei mit einer Leerzeile enden muss. Soll die .jar-Datei nicht im aktuellen Verzeichnis erzeugt werden, so kann zunächst ein Ordner `dist` angelegt werden. Mit dem folgenden Befehl kann anschließend in diesem Ordner die ausführbare .jar-Datei erzeugt werden:

```
jar cmf MANIFEST.MF dist/netzplanerstellung-1.0.0.jar -C build .
```

Kapitel 5

Testdokumentation

In diesem Kapitel werden nun alle mitgelieferten Testfälle aufgelistet, beschrieben und diskutiert. Die Zusammenstellung dieser Testfälle ist schon während der Implementierung nach dem *Whitebox*-Prinzip erfolgt: Jede neue Verzweigung im Programm wurde sofort mithilfe eines neuen Testfalls festgehalten. Um diesen Vorgang der Entwicklung zu beschreiben, passt ebenfalls der Begriff *Test-Driven-Development*.

Die Gesamtheit der Testfälle wird hier in folgende Kategorieren unterteilt:

- **Normalfälle:** Führen zu einer ordnungsgemäßen Abarbeitung des Programms.
- **Sonderfälle:** Enthalten Spezialfälle, wie z.B. ein Minimalbeispiel, um auch ungewöhnliche Situationen testen zu können.
- **Fehlerfälle:** Führen nicht zu einer ordnungsgemäßen Abarbeitung des Programms und ziehen Fehlermeldungen nach sich.

Sämtliche Testfälle sind im Ordner `testcases` zu finden. Zu jedem Testfall, welcher in einer ordnungsgemäßen Abarbeitung des Programms mündet, findet sich ebenfalls im Ordner `testcases` eine zugehörige Ausgabedatei mit der Endung `.out`.

5.1 Normalfälle

Name:	Installation von POI Kiosken
Dateiname:	`ihk_01.in`
Beschreibung:	Projekt, welches sich mit der Einrichtung eines Kiosk-Terminals beschäftigt. Es enthält eine Verzweigung nach dem Startvorgang, die an Vorgang Nr. 6 wieder zusammengeführt wird. Das Projekt enthält einen einzigen kritischen Pfad.
Diskussion:	Bei diesem Testfall handelt es sich um ein einfaches Normalbeispiel, welches die grundsätzliche Funktionsfähigkeit des Programms unter Beweis stellt.

Name:	Wasserfallmodell
Dateiname:	`ihk_02_korrigiert.in`
Beschreibung:	Dieser Testfall veranschaulicht die Vorgehensweise bei der Softwareentwicklung nach dem Wasserfallmodell. Es gibt keinerlei Verzweigungen und nur einen kritischen Pfad. Dieses Beispiel wurde aus der ursprünglichen Datei `ihk_02.in` erzeugt, indem in Zeile 11 der Vorgänger von Vorgang 6 auf Vorgang 5 korrigiert wurde.
Diskussion:	Der Testfall gibt einen simplen Fall ohne Verzweigungen an und stellt sicher, dass das Programm auch bei listenähnlichen Netzplänen den einzigen kritischen Pfad findet.

Name:	Beispiel 3
Dateiname:	`ihk_03.in`
Beschreibung:	Dieser Testfall stellt ein Projekt mit mehreren Start- und Endvorgängen dar. Er enthält mehrere Verzweigungen und zwei kritische Pfade.
Diskussion:	Dieser Testfall zeigt, dass das Programm auch in der Lage ist, bei mehreren Start- und Endvorgängen korrekt zu arbeiten. Weiterhin wird demonstriert, dass auch mehrere kritische Pfade fehlerfrei gefunden werden.[1]

[1] Die Ausgabe des Programms stimmt an einer Stelle aus gutem Grund nicht mit der aus der Aufgabenstellung gegebenen Kontrolllösung überein: Der SEZ von Vorgang 8 beträgt richtigerweise 12 und nicht 13, wie es in der Beispielausgabe steht.

Name:	Beispiel 5 IT-Installation
Dateiname:	`ihk_05_korrigiert.in`
Beschreibung:	Hier wird ein komplexes Beispiel mit 17 verschiedenen und verzweigten Vorgängen illustriert. Dieses Beispiel wurde aus der ursprünglichen Datei `ihk_05.in` erzeugt, indem in Zeile 13 bei Vorgang 8 der Vorgänger auf Vorgang 4 korrigiert wurde. Bei diesem Testfall gibt es einen kritischen Pfad.
Diskussion:	Dies ist der bis hierhin komplexeste Testfall, den das Programm bisher abarbeiten musste und es zeigt sich, dass es auch hier nicht auf Probleme stößt.[2]

5.2 Sonderfälle

Name:	Nummern ausser der Reihe
Dateiname:	`sonderfall_01.in`
Beschreibung:	Dieser Testfall hat im Wesentlichen den gleichen Inhalt wie die Datei `ihk_01.in`, bis auf einen Unterschied: Die Nummer von ehemals Vorgang 1 wurde auf 700 gesetzt und die Nummer von ehemals Vorgang 4 wurde auf -25 festgelegt.
Diskussion:	Hier soll geprüft werden, ob das Programm auch dann noch korrekt arbeitet, wenn die Vorgangsnummern nicht genau in einer Reihe aufeinander folgen. Dank der Definition der Abbildung g, auch als `toInternal` im Quelltext zu finden (Abbildung der externen auf die internen Vorgangsnummern), sind auch unorthodoxe Vorgangsnummern kein Problem für das Programm.

Name:	Ein einziger Knoten
Dateiname:	`sonderfall_02.in`
Beschreibung:	Dieses Projekt besteht aus nur einem Vorgang, welcher gleichzeitig Start- und Endvorgang ist.
Diskussion:	Dieser Testfall stellt das kleinste noch funktionsfähige Beispiel dar. Auch hier ist das Programm erfolgreich und berechnet den einzigen kritischen Pfad korrekt.

[2] Aufgrund der Korrektur stimmt auch hier die Ausgabe des Programms nicht mit der gegebenen Kontrolllösung überein.

5.3 Fehlerfälle

Name:	Wasserfallmodell
Dateiname:	`ihk_02.in`
Beschreibung:	Diese Eingabedatei enthält einen Tippfehler: Vorgang 6 hat den nicht existenten Vorgang 7 als Vorgänger, obwohl er eigentlich Nachfolger von Vorgang 5 ist.
Diskussion:	Hier wird erstmals überprüft, ob das Programm Inkonsistenzen in der Vorgänger - Nachfolger - Relation korrekt erkennt.
Fehlermeldung:	`Fehler bei der Erstellung des Netzplans: Inkonsistente Beziehung gefunden! Vorgang 5 hat Vorgang 6 als Nachfolger, ist aber selbst nicht Vorgänger von diesem!`

Name:	Beispiel 4 mit Zyklus
Dateiname:	`ihk_04.in`
Beschreibung:	Weiterer Test zur Erkennung von Inkonsistenzen.
Diskussion:	Dieser gegebene Fehlerfall sollte vermutlich einen Zyklus illustrieren, enthält allerdings Inkonsistenzen in den Beziehungen der Vorgänge.
Fehlermeldung:	`Fehler bei der Erstellung des Netzplans: Inkonsistente Beziehung gefunden! Vorgang 4 hat Vorgang 3 als Nachfolger, ist aber selbst nicht Vorgänger von diesem!`

Name:	Beispiel 4 mit Zyklus (korrigiert)
Dateiname:	`ihk_04_korrigiert.in`
Beschreibung:	Das Projekt zu diesem Testfall führt nicht zu einem gültigen Netzplan, da es einen Zyklus enthält. Die Korrektur zum vorigen Beispiel wurde in Zeile 8 durchgeführt: Vorgang 3 hat nun Vorgang 4 auch als Vorgänger, damit ein Zyklus zustande kommt.
Diskussion:	Hier wird erstmals überprüft, ob das Programm Zyklen im Netzplan erkennt.
Fehlermeldung:	`Fehler bei der Erstellung des Netzplans: Es wurde ein Zyklus erkannt! 3->4->3`

Kapitel 6

Ausblick

Anhang A

Abweichungen und Ergänzungen zum Vorentwurf