# Dokumentation der Praktischen Arbeit zur Prüfung zum Mathematisch-technischen Softwareentwickler

3. April 2020

**Lukas Dahlberg**

Prüfungs-Nummer:

Programmiersprache: JAVA 8

# Inhaltsverzeichnis

# 1. Aufgabenanalyse

## 1.1. Analyse

## 1.2. Eingabeformat

## 1.3. Ausgabeformat

## 1.4. Anforderung an das Programm

## 1.5. Sonderfälle

## 1.6. Fehlerfälle

Die auftretenden Fehler kann man in drei verschiedene Fehlerarten aufteilen. Zu diesen gehören technische, syntaktische und semantische Fehlerfälle.
Technische Fehler liegen vor, wenn die angegebene Datei nicht vorhanden ist, durch fehlende Zugriffsrechte nicht gelesen werden kann, oder die Ausgabedatei durch fehlende Schreibrechte nicht erstellt werden kann. Syntaktische Fehler treten auf, wenn die Eingabedatei nicht dem vorgegebenen Format entspricht, bei semantischen Fehlern sind die Daten in der Eingabedatei fehlerhaft.

Durch die Analyse der Aufgabenstellung und des Eingabeformated ergeben sich folgende syntaktische Fehlerfälle:

- Mögliche Fehlerfälle auflisten

Durch die Analyse der Aufgabenstellung und des Eingabeformates ergeben sich folgende semantische Fehlerfälle:

- Mögliche Fehlerfälle auflisten

Die Behandlung der Fehlerfälle wird in der Verfahrensbeschreibung angegeben.

# 2. Verfahrensbeschreibung

## 2.1. Vorgehensweise

### Sonderfälle

Die oben genannten Sonderfälle werden wie folgt behandelt:

**Sonderfall Bezeichnung** Sonderfall Beschreibung und Behandlung

### Fehlerfälle

Die oben genannten Fehlerfälle werden wie folgt behandelt:

**Fehlerfall Bezeichnung** Fehlerfall Beschreibung und Behandlung

In den Fällen, wo nach einem Fehler weitergearbeitet wird, wird auch weiter auf andere Fehler geprüft. Sollten Fehler entdeckt werden und die Verarbeitung fortgesetzt, so wird in der Ausgabe eine Hinweismeldung erscheinen, dass trotz eines Fehlers die Verarbeitung fortgeführt wurde.

## 2.2. Gesamtsystem

### 2.2.1. Main-Funktion

### 2.2.2. Model-Klassen

### 2.2.3. IO-Klassen

#### 2.2.4. Controller-Klassen

## 2.3. Datenfluss

# 3. Programmbeschreibung

## 3.1. Pakete

Das Programms ist in drei Module unterteilt, welche nach dem MVC-Model entwickelt wurden. Für jedes dieser Module wurde zunächst ein Interface deklariert, damit das Programm leicht wart- und erweiterbar bleibt. Die Namen der Interfaces beginnen mit einem „I“, wie es etwaige Namenskonventionen vorgeben.

### 3.1.1. Model

**IModel** Schnittstelle zum Speichern der Daten: Spielsteine, Spielfeld, Matrix der Nachbarschaft für Felder

**Model** Implementierung von IModel.

### 3.1.2. IO

**IDateiBehandlung** Schnittstelle zur Ein- & Ausgabe der Daten.

**DateiBehandlung** Implementierung von IDateiBehandlung. Die Klasse verfügt über eine Methode zum lesen der Eingabedatei, eine zum schreiben der Ausgabedatei und eine zum schreiben eines Fehlers in eine Ausgabedatei.

### 3.1.3. Controller

**PuzzleLoeserFactory** Schnittstelle zum Starten des Controllers, damit die Verarbeitung gesteuert werden kann.

**PuzzleLoeser** Abstrakte Klasse als Schnittstelle zum Starten. PuzzleLoeserEinsetzen. Beinhaltet die Implementierungen und den Lösungsalgorithmus.

## 3.2. Schnittstellen

### 3.2.1. Program

Main

**public static void** main(**string**[] args)

**Eingabeparameter:** Array mit den Kommandozeilen-Argumenten

**Rückgabeparameter:** keine

### 3.2.2. Spielstein

#### drehen

**public** Spielstein drehen();

**Eingabeparameter:** keine vorhanden

**Rückgabeparameter:** der gedrehte Spielstein wird zurückgegeben.

#### getFeld

**public int** getFeld();

**Eingabeparameter:** keine vorhanden

**Rückgabeparameter:** das Feld, auf dem der Spielstein liegt, ist der Spielstein nicht gelegt, wird 0 zurückgegeben.

#### setFeld

**public void** setFeld(**int** feld);

**Eingabeparameter:** Die Feldnummer, des Feldes, in die der Stein gesetzt wurde.

**Rückgabeparameter:** keiner

#### kannKombi

**public** boolean kannKombi(List<Integer> kombination);

**Eingabeparameter:** Eine Liste mit den Kantenziffern, der Nachbarfelder, beginnend mit der Grundkante.

**Rückgabeparameter:** Wenn der Spielstein in der aktuellen Orientierung die Kombination abdecken kann, dann wird **true** zurückgegeben, sonst **false**.

### 3.2.3. Spielfeld

#### GetInstance

**public** Spielfeld getInstance()

**Eingabeparameter:** keiner

**Rückgabeparameter:** eine Spielfeldinstanz.

#### GetKombination

**public** List<Integer> GetKombination(**int** feld)

**Eingabeparameter:** Die Feldnummer, für das Feld, deren Kombination gesucht ist.

**Rückgabeparameter:** Eine Liste mit Integer-Werten für die Kantenziffer, die an Position x vorhanden sein muss, beginnend von der Grundkante. Liegt an einer Kante noch kein weiterer Stein, wird anstelle der Kantenziffer ein null in der Liste zurückgegeben.

### 3.2.4. DateiBehandlung

#### SchreibeDatei

**public void** schreibeDatei(List<Spielstein> loesung)

**Eingabeparameter:** Eine Liste der Spielsteine, aufsteigend nach den Feldern, in die diese gelegt werden sortiert. Ist keine Lösung vorhanden, wird eine leere Liste übergeben.

**Rückgabeparameter:** keiner

#### SchreibeFehler

**public void** schreibeFehler(String text)

**Eingabeparameter:** Die Fehlermeldung die Ausgegeben werden soll, mit vorangestellter Zeile und Datei in der der Fehler festgestellt wurde, getrennt durch #

**Rückgabeparameter:** keiner

#### LeseDatei

**public** List<Spielstein> leseDatei()

**Eingabeparameter:** keine vorhanden

**Rückgabeparameter:** Eine Liste der in der Eingabedatei beschriebenen Spielsteine, sortiert nach der Einlesereihenfolge. Kommt es beim einlesen zu Fehlern, so wird null zurückgegeben.

### 3.2.5. PuzzleLoeser

#### loesePuzzle

**public void** loesePuzzle()

**Eingabeparameter:** keine vorhanden

**Rückgabeparameter:** keiner

#### GetLoesung

**public** List<Spielstein> getLoesung()

**Eingabeparameter:** keine vorhanden

**Rückgabeparameter:** Eine Liste der Spielsteine, aufsteigend nach den Feldern, in die diese gelegt werden sortiert. Ist keine Lösung vorhanden, wird eine leere Liste zurückgegeben.

#### setzeSpielstein

**private** boolean setzeTeilInFeld( Spielstein stein , **int** feld )

**Eingabeparameter:** Der zu setzende Spielstein und die Feldnummer des Feldes, in welcher der Spielstein zu setzen ist.

**Rückgabeparameter:** Wenn der Stein in das Feld gesetzt werden kann, wird **true** zurückgegeben, andernfalls **false**.

### PuzzleLoesbarAnfang

**private** boolean puzzleLoesbarAnfang()

**Eingabeparameter:** keine vorhanden

**Rückgabeparameter:** Wenn das Puzzle lösbar scheint, wird **true** zurückgegeben, andernfalls **false**. D.h., wenn die Anzahl der einzelnen Kantenziffern gerade ist, wird angenommen, dass das Puzzle lösbar ist.

### PuzzleLoesbarAlgorithmus

**private** boolean puzzleLoesbarAlgorithmus()

**Eingabeparameter:** keine vorhanden

**Rückgabeparameter:** Wenn noch setzbare Steine vorhanden sind, wo mindestens einer, noch gesetzt werden kann, wird **true**, andernfalls **false** zurückgegeben.

### PuzzleGeloest

**private** boolean puzzleGeloest()

**Eingabeparameter:** keine vorhanden

**Rückgabeparameter:** gibt **true** zurück, wenn keine ungenutzten Steine mehr vorhanden sind, andernfalls wird **false** zurückgegeben.

## 3.3. Präzisierung

### 3.3.1. Präzisierung SetSpielstein

In der Methode setSpielstein ( Spielstein stein , **int** feld ) wird überprüft, ob der Spielstein in das Feld setzbar ist. Hierfür wird zunächst mit der Methode kannKombi(List<Integer> kombination) geschaut, ob der übergebene Spielstein, die Kombination für das übergebene Feld abdecken kann. Zur Bestimmung der Kombination wird hier die Methode getKombination(**int** feld) aufgerufen. Wenn gewährleistet ist, dass der Spielstein in das Feld setzbar ist, dann wird das Array felder angepasst. Der übergebene Spielstein wird in felder [ feld − 1] gesetzt. In Folge dessen werden die Attribute des Steins angepasst, dass dieser als gelegt aufgefasst wird, in dem diesen die Informationen, des Feldes in dem der Stein liegt übergeben wird und das boolsche Attribut gelegt auf **true** gesetzt wird.

### 3.3.2. Präzisierung GetKombination

### 3.3.3. Präzisierung GetNachbarSteine

### 3.3.4. Präzisierung LoesePuzzle

### 3.3.5. Präzisierung Drehen

## 3.4. Sequenzdiagramm

# 4. Testdokumentation

Parallel zu dem Programm wurden Tests erstellt, um die direkte Funktionalität von hinzugefügtem Code zu überprüfen. Diese wurden als Systemtests ausgeführt und sind nach jeder Erweiterung des Programms komplett ausgeführt worden. Die Tests sind in Form von Eingabedateien in der Abgabe im Ordner Tests, die dazugehörigen Ausgaben im Ordner Output enthalten.

## 4.1. Systemtests

Alle Systemtests wurden durch Dateien getestet und die Ergebnisse manuell überprüft, weil eine automatische Verifizierung über Dateien für den gegebenen Zeitraum zu aufwendig ist und nicht gefordert war. Die Ein- und Ausgabedateien der Systemtests sind in der Abgabe enthalten.

### 4.1.1. Normalfall

Der Normalfall liegt vor, wenn der Inhalt der Eingabedatei keine Sonderfälle aus Abschnitt 1.5 auf Seite 2 beinhaltet und keiner der in Abschnitt 1.6 auf Seite 2 aufgelisteten Fehler eintritt.

Der Normalfall ist anhand des gegebenen IHK-Beispiels getestet worden. Anhand des ersten IHK-Beispiels wird in Abschnitt 4.2 auf Seite 17 der Algorithmus und die einzelnen Schritte genauer erläutert.

Für den Normalfall sind noch vier weitere Testbeispiele entwickelt worden. Das erste Beispiel LOESBAR_Beispiel1.txt soll einen Normalfall mit vielen gleichen Werten testen, das zweite Beispiel LOESBAR_Beispiel2.txt soll die Anforderung testen, dass zwischen den Kantenziffern beliebig viele Leerzeichen sein können. Die dritte Beispieldatei LOESBAR_Beispiel3.txt soll einen Normalfall mit nur unterschiedlichen Teilen testen, die auch bei Verdrehung nicht mit den anderen übereinstimmen. Das vierte Beispiel NICHT_LOESBAR_Beispiel1.txt testet die Anforderungen für den Fall, dass ein Puzzle übergeben wurde, welches nicht dem Sonderfall entspricht, da die Anzahl jeder Kantenziffer gerade ist, jedoch auch zu keiner Lösung führt.

### 4.1.2. Sonderfall

Für den Sonderfall, der in der Aufgabenanalyse im Abschnitt Sonderfälle (Abschnitt 1.5 auf Seite 2) beschrieben wurde, wurden insgesamt drei Testfälle erstellt, wobei die drei Testfälle eine ungerade Anzahl an Kantenziffern testet. Es waren mehr Testfälle als aufgelistete Sonderfälle nötig, weil die Beschreibung eines Sonderfalls allgemein ist, der Sonderfall aber horizontal und vertikal überprüft werden muss.

### 4.1.3. Fehlerfall

Es sind für die sieben Fehlerfälle, die in der Aufgabenanalyse im Abschnitt Fehlerfälle (Abschnitt 1.6 auf Seite 2) aufgelistet sind, sieben Tests entwickelt worden. Wie auch schon bei den Sonderfällen sind einige in der Beschreibung zusammengefasst, aber einzeln getestet worden. In allen Fällen wird eine Fehlermeldung in die Ausgabedatei geschrieben und das Programm korrekt beendet, ohne die die Logik zum Lösen des Puzzles zu starten.

Zusätzlich wurde noch getestet, wie das Programm auf nicht vorhandene Eingabedateien und nicht schreibbare Ausgabedateien reagiert. In beiden Fällen wurde die Fehlermeldung in einer Ausgabedatei im Root-Ordner ausgegeben und das Programm korrekt beendet.

## 4.2. Ausführliches Beispiel

An dieser Stelle wird das erste IHK-Beispiel herangezogen, also ein gültiges und lösbares Puzzle. Da das Beispiel keinerlei Fehler enthält, wird auf die Prüfung dieser in dem Beispiel verzichtet.

Die Eingabedatei von diesem Beispiel sieht wie folgt aus:

Trotz der Null-Indizierung in Java, werden weiterhin die Indizes 1 bis 12 verwendet. Im Programmcode werden auch diese Werte weitergegeben und beim auslesen oder schreiben von Arrays und Lists entsprechend angepasst.

Zunächst wird bestimmt, ob das Puzzle überhaupt lösbar ist mit der Methode puzzleLoesbarAnfang().

Die Rechenvariablen anzKZ0, anzKZ1 und anzKZ2 werden mit 0 initialisiert. In Folge dessen werden alle Kanten jedes Puzzlesteils einmal aufgerufen, und die entsprechende Rechenvariable um 1 erhöht, wenn eine 0, 1 oder 2 eingelesen wurde. Im Anschluss wird das Ergebnis der boolschen Berechnung des Ergebnis, in dem überprüft wird, ob anzKZ0 \% 2 == 0 \&\& anzKZ1 \% 2 == 0 \&\& anzKZ2 \% 2 == 0 ist, zurückgegeben.

Anschließend wird der Backtracking Algorithmus gestartet. Hier wird bis das Puzzle gelöst ist probiert, den ungenutzten Spielstein mit der geringsten Kartennummer (Position in der Eingabedatei) auf das nächste freie Feld zu setzen. Begonnen wird mit dem Feld 1 und Stein 1, welcher aufgrund dessen, dass keinerlei Bedingungen an diesen gestellt werden, gelegt werden kann.

Nun wird die Tiefe des Algorithmus um 1 erhöht, und es wird fortgefahren mit dem einsetzen des 2. Steins in Feld 2. Dieser passt jedoch in der Grundorientierung nicht an das besetzte Feld 1 dran. Aus diesem Grund wird der Stein solange gedreht, bis dieser passt. Nach maximal 5 Drehungen, gilt der Stein als nicht legbar und der Algorithmus würde wieder eine Stufe nach oben gehen. Da der Stein allerdings nach einer Drehung an Feld 1 passt, wird mit dem dritten Spielstein für Feld drei fortgefahren. Nachdem der Algorithmus fertig ist, wird diese Ausgabedatei erstellt:

# 5. Zusammenfassung und Ausblick

## 5.1. Zusammenfassung

Durch die funktionale Trennung nach dem Model-View-Controller-Modell, kann die Ausgabe leicht geändert werden, ohne den Lösungsalgorithmus oder die Datenhaltung zu beeinflussen. Es können allerdings auch andere Lösungsalgorithmen entwickelt werden und der Controller, welcher diesen beinhaltet, einfach ausgetauscht werden. Damit ist das Programm offen für Erweiterung aber geschlossen gegenüber Änderungen. Ein Manko ist, dass die Eingabe nicht in einer separaten Klasse erstellt wurde, sondern gemeinsam mit der Ausgabe. Damit muss bei Änderungen in der Eingabe auch die Ausgabe neu erstellt werden. Da dieses Problem aber durch Vererbung umgangen werden kann, indem die Eingabe ein weiteres Interface implementiert, ist an dieser Stelle nicht von dem Konzept abgewichen wurden.

Das abspeichern der Spielsteine in einer Liste spart Zeit, weil bei der Werteabfrage bzw. bei Verwendung dieser nicht jedes mal der Typ konvertiert werden muss. Aufgrund dessen, dass nur eine mögliche Lösung gesucht werden muss, spart dies Zeit, da der Algortihmus mit finden der ersten möglichen Lösung endet.

Die Schwäche des Algorithmus liegt allerdings in sehr unterschiedlichen Steinen, wo der nächste passende Stein der letzte ungenutzte ist. Dort greifen die vorhandenen Abbruchbedingungen nicht bzw. nicht wirksam genug und die Berechnungsdauer steigt rapide an. An diese Stelle müssten bei weiterer Entwicklung noch Optimierungen vorgenommen werden.

## 5.2. Ausblick

Der Algorithmus kann noch verbessert werden, indem nicht für jeden ungenutzten Stein, jedes Feld ausprobiert wird, sondern nur für die ungenutzten, die die von den Nachbarfeldern geforderte Kombination enthalten. Dass heißt, dass nur die Steine probiert werden, deren Kantenziffern eine Folge beinhaltet, dass diese mit den Kantenziffern an den Nachbarfeldern übereinstimmen. Somit muss nicht mehr für jeden nicht legbaren Stein fünf Legeversuche durchgeführt werden, sondern nur noch für die Steine, die gelegt werden können. Dadurch kann selbst bei großen Abständen der Kartennummern für passende Karten der Algorithmus diese schnelle legen.

Als mögliche Erweiterungen für das Programm ist folgendes denkbar

- Eine grafische Oberfläche
  - Ausgabe des Puzzles in Fünfecken.
  - Ausgabe aller möglichen Lösungen des Puzzles.
  - Eingabe der Puzzleteile über die grafische Oberfläche.
- Verwendung anderer geometrischer Figuren als Puzzleteile, wie z.B. Sechsecke. Hierfür müsste auch das Spielfeld entsprechend angepasst werden.
- Die Anzahl verschiedener Kantenziffern kann erhöht bzw. verringert werden, um das Puzzle entsprechend schwieriger bzw. leichter zu gestalten.
- Andere Algorithmen, mit denen Regeländerungen verbunden sein könnten, wie z.B. dass die Summe der aneinanderliegenden Kanten 3 ergeben muss. (Dieses Beispiel ist nur sinnvoll, wenn weitere Kantenziffern erlaubt sind.)

# A. Abweichungen und Ergänzungen zum Vorentwurf

## A.1. Abweichungen

Auf dem Programmkonzept wurden am Hauptalgorithmus die folgenden Modifikationen vorgenommen:

In der Implementierung sind die folgenden Modifikationen bei Datenstrukturen vorgenommen worden:

Bei der Behandlung der Sonder- und Fehlerfälle wurden keine Modifizierungen vorgenommen.

## A.2. Ergänzungen

# B. Benutzeranleitung

## B.1. Verzeichnisstruktur der Abgabe

Im Prüfungsprodukt sind folgende Dateien und Verzeichnisse vorhanden:

#### Wurzelverzeichnis

- vorkompilierte Version des Programms
- Skript zum automatischen Ausführen des Programms mit mehreren Eingabedateien
- Skript zum Kompilieren des Programms
- Skript zum Löschen aller erstellten Ausgabedateien

**Output** Enthält die Ausgabedateien des Programms.

**src** Enthält den Quellcode des Programms.

**Tests** Enthält beispielhafte Eingabedateien.

**Doku** Enthält die Dokumentation.

## B.2. Vorbereitung des Systems

### B.2.1. Systemvoraussetzungen

Um das Programm verwenden zu können, wird ein Microsoft Betriebssystem in der Version 8 oder höher benötigt. Auf dem Betriebssystem muss die Windows PowerShell sowie eine Java Runtime Environment (JRE) der Version 8 oder höher installiert sein.

Die PowerShell sollte nach Angaben von Microsoft auf jedem Windows Rechner mit einem Betriebssystem von Windows 8 oder höher standardmäßig installiert sein.

Sollte die JRE nicht oder in einer veralteten Version installiert sein, kann dies unter der folgenden Adresse heruntergeladen werden:
https://www.java.com/de/download/

#### PowerShell einrichten

Die PowerShell untersagt das Ausführen von Skripten in der Standardeinstellung. Damit dies geändert werden kann, muss die PowerShell als Administrator gestartet werden.

Um die PowerShell als Administrator zu starten, geht man in den Startmenüeintrag der PowerShell. Diesen finden Sie durch das öffnen des Startmenüs und anschließender Eingabe von „PowerShell“Dort machen Sie einen Rechtsklick auf „Windows PowerShell“, und klicken in dem aufkommendem Menü auf den Punkt „Als Administrator ausführen“. Je nach Sicherheitseinstellungen des Betriebssystems erscheint ein Fenster mit der Nachfrage, ob Änderungen durch das Programm zugelassen werden sollen, dies wird mit „Ja“ bestätigt. Gegebenenfalls müssen Sie auch Ihr Administratorpasswort eingeben um fortfahren zu können. Anschließend öffnet sich dann die Windows PowerShell mit Administratorrechten.

Als erstes sollten die aktuellen Einstellungen mittels Get−ExecutionPolicy ausgelesen und notiert werden, damit die Einstellungen später wieder zurückgesetzt werden können. Im nächsten Schritt werden die Richtlinien geändert. Dies geschieht über den Befehl Set−ExecutionPolicy RemoteSigned. RemoteSigned bedeutet, dass Skripte, die aus dem Internet heruntergeladen wurden, signiert sein müssen um ausgeführt zu werden. Lokal erstellte Skripte werden immer ausgeführt.

Falls die Skripte nicht ausgeführt werden sollten, müssen die Richtlinien weiter herabgesetzt werden, dies geschieht über Set−ExecutionPolicy Unrestricted. Unrestricted bedeutet, dass alle Skripte ausgeführt werden, jedoch wird für unsignierte Skripte, die aus dem Internet stammen, eine Warnung ausgegeben.

Die Richtlinien der PowerShell können in den ursprünglichen Zustand mittels des Befehls Set−ExecutionPolicy <POLICY> zurückgesetzt werden. Dabei ist <POLICY> der im ersten Schritt ausgelesene Wert. Um die Voreinstellungen der PowerShell zu erhalten, wird als Parameter Restricted angegeben.

### B.2.2. Installation

Der gesamte Inhalt der Abgabe ist in ein beliebiges und beschreibbares Verzeichnis zu kopieren. Danach ist das Programm betriebsbereit.

## B.3. Kompilieren

Ein vorkompiliertes und ausführbares Programm liegt in Form von GroProLDahlberg.jar der Abgabe bei. Soll das Programm wegen Änderungen im Quelltext, oder aus anderen Gründen neu kompiliert werden, kann mit dem Skript compile.ps1 das Programm neu kompiliert werden.

Damit dies erfolgreich ausgeführt werden kann, darf die vorliegende Verzeichnisstruktur nicht verändert werden, weil das Skript speziell auf diese Struktur angepasst ist. Weiter ist erforderlich, dass ein Compiler für Java (JDK) der Version 8 in der Umgebungsvariable PATH eingebunden ist. Ist der Compiler nicht in der Umgebungsvariable funktioniert der Compilerbefehl nicht.

Um die PATH-Variable zu erweitern, geht man im Startmenü mit einem Rechtsklick auf „Computer“→ „Eigenschaften“. Dort wird „Erweiterte Systemeinstellungen“ am linken Rand ausgewählt. Auf der Registerkarte „Erweitert“ wird der unterste Knopf „Umgebungsvariablen...“ gewählt. Im nächsten Schritt wählt man die PATH-Variable aus, dabei ist es egal, ob nun die Benutzer- oder die Systemvariable ändert. Jedoch werden zur Änderung der Systemvariable Administratorrechte benötigt. Nach der Auswahl der PATH-Variable wird auf den Knopf „Bearbeiten...“ gedrückt. Im Feld für den Wert der Variable wird am Ende folgendes angehängt:

;<Pfad zum JDK bin Ordner>
Falls ein JDK einer höheren Version vorhanden ist, kann auch dieses verwendet werden. Hier wird allerdings nicht garantiert, dass der Kompiliervorgang erfolgreich ist.

Beim anfügen in die PATH-Variable ist es wichtig auf das Semikolon zu achten, da dieses als Trennzeichen agiert.

Wenn diese Bedingungen erfüllt sind, kann das PowerShell-Skript compile.ps1 ausgeführt werden. Dabei wird automatisch die Datei GroProLDahlberg.jar neu erstellt.

## B.4. Programmaufruf

Um einzelne Dateien zu verarbeiten, müssen diese dem Programm GroProLDahlberg.jar über die Kommandozeile oder PowerShell als Parameter mitgegeben werden. In beiden Fällen erstellt das Programm eine Ausgabedatei, die den ursprünglichen Dateinamen um „.out“ erweitert.

## B.5. Testen der Beispiele

Sollen mehrere Testbeispiele verarbeitet werden, müssen diese in einem Verzeichnis zusammen gefasst werden. Durch das PowerShell-Skript start.ps1 werden diese dann nacheinander verarbeitet. Das Skript hat als optionale Parameter DIR und TYP. Über DIR kann ein Verzeichnis mit den Testfällen angegeben werden und über TYP die Endung der Dateien. Die default-Werte sind für DIR ./Tests und für TYP .txt. Diese werden verwendet, wenn keine anderen Werte übergeben werden. Ein Beispiel für den Aufruf ist:

./start.ps1 TYP=.input

# C. Entwicklungsumgebung

| | | |
|---|---|---|
| Programmiersprache | : | JAVA 8 |
| Compiler | : | Microsoft (R) Visual C# 2010 Compiler<br>Version 4.0.30319.1 |
| Rechner | : | Intel(R) Core(TM) i5-6300U<br>2,50GHz<br>32GB Arbeitsspeicher |
| Betriebssystem | : | Windows 10 Enterprise 1809<br>64 Bit-Betriebssystem |

# D. Verwendete Hilfsmittel

- Eclipse Photon
  Entwicklungsumgebung für Java und andere Programmiersprachen
  https://eclipse.org
- Java Platform, Standard Edition 8
  API Specification
  Internetseite mit Erklärungen und Hilfen rund um die Programmiersprache
  https://https://docs.oracle.com/javase/8/docs/api/
- Notepad++
  Open-Source-Texteditor für die Windows Desktopumgebung
  http://notepad-plus-plus.org/
- Overleaf
  Plattform unabhängiger LaTeX-Editor
  https://overleaf.com
- Papeeria
  Plattform unabhängiger LaTex-Editor
  https://papeeria.com
- Structorizer
  Plattform unabhängiges Programm zur Erzeugung von Nassi-Shneiderman-Diagrammen
  http://structorizer.fisch.lu/
- diagrams.net
  Online UML-Diagramm Editor
  https://draw.io

# E. Erklärung

Erklärung des Prüfungsteilnehmers / der Prüfungsteilnehmerin:

Ich versichere durch meine Unterschrift, dass ich das Prüfungsprodukt selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe. Die Arbeit hat in dieser Form keiner anderen Prüfungsinstitution vorgelegen.

Das abgegebene Prüfungsprodukt entspricht der gedruckten Version.

Köln, den 3. April 2020

Ort und Datum

Unterschrift des Prüfungsteilnehmers

# Anhang F

# Aufgabenstellung

# Anhang G

# Quellcode

1. **Sourcedateien**
   24 Seiten
2. **Skripte**
   3 Seiten
3. **Inline-Dokumentation**
   7 Seiten

# Anhang H

# In- und Output der Testdokumentation

**Normalfall**

**Sonderfall**

**Fehlerfall**