\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dokumentation der praktischen Arbeit  
zur Prüfung zur  
mathematisch-technischen Softwareentwicklerin

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6. Mai 2022

**Johanna Schindler**

Prüfungs-Nummer: 142 - 18741

Programmiersprache: Java

Inhalt

[1 Aufgabenanalyse 1](#_Toc101860230)

[1.1 Allgemeines 1](#_Toc101860231)

[1.2 Eingabe 1](#_Toc101860232)

[1.3 Problemstellung 2](#_Toc101860233)

[1.4 Ausgabe 2](#_Toc101860234)

[1.5 Grenz- und Fehlerfälle 2](#_Toc101860235)

[1.5.1 Grenzfälle 2](#_Toc101860236)

[1.5.2 Fehlerfälle 3](#_Toc101860237)

[1.6 Anforderungsliste 3](#_Toc101860238)

[2 Verfahrensbeschreibung 4](#_Toc101860239)

[2.1 Eingabe und Initialisierung 4](#_Toc101860240)

[2.2 Berechnung der … 4](#_Toc101860241)

[2.3 Ausgabe 5](#_Toc101860242)

[3 Programmbeschreibung 6](#_Toc101860243)

[3.1 Grobentwurf 6](#_Toc101860244)

[3.1.1 Module 6](#_Toc101860245)

[3.1.2 Ein- und Ausgabe 6](#_Toc101860246)

[3.1.3 Model 7](#_Toc101860247)

[3.1.4 Controller// TODO auf Problem anpassen 7](#_Toc101860248)

[3.2 Ablauf 8](#_Toc101860249)

[3.2.1 Gesamtablauf 8](#_Toc101860250)

[3.2.2 Initialisierung des Programms 8](#_Toc101860251)

[3.2.3 Strategie 8](#_Toc101860252)

[3.3 Algorithmen // TODO wenn Plural unterteilen 8](#_Toc101860253)

[3.4 Ein- und Ausgabekonvertierung 8](#_Toc101860254)

[3.4.1 Eingabekonvertierung 8](#_Toc101860255)

[3.4.2 Ausgabekonvertierung 9](#_Toc101860256)

[4 Testdokumentation 10](#_Toc101860257)

[4.1 Begründung und Diskussion der Testfälle 10](#_Toc101860258)

[4.1.1 Grenzfalltests 10](#_Toc101860259)

[4.1.2 Fehlertests 10](#_Toc101860260)

[4.1.3 Technische Fehler 11](#_Toc101860261)

[4.1.4 Normalfälle 11](#_Toc101860262)

[4.2 Ausführliches Beispiel 12](#_Toc101860263)

[4.2.1 Eingabe 12](#_Toc101860264)

[4.2.2 Algorithmen // TODO wenn mehrere unterteilen 12](#_Toc101860265)

[4.2.3 Ausgabe 12](#_Toc101860266)

[5 Zusammenfassung und Ausblick 13](#_Toc101860267)

[5.1 Zusammenfassung 13](#_Toc101860268)

[5.2 Ausblick 13](#_Toc101860269)

[5.2.1 Ein- und Ausgabe 13](#_Toc101860270)

[5.2.2 Strategien TODO 13](#_Toc101860271)

[6 Abbildungsverzeichnis // TODO aktualisieren 13](#_Toc101860272)

[A Abweichungen und Ergänzungen zum Vorentwurf 14](#_Toc101860273)

[B Benutzeranleitung 14](#_Toc101860274)

[a. Verzeichnisstruktur 14](#_Toc101860275)

[b. Systemvorraussetzungen 14](#_Toc101860276)

[c. Installation 15](#_Toc101860277)

[d. Kompilieren 15](#_Toc101860278)

[e. Programmausführung 15](#_Toc101860279)

[f. Testausführung 15](#_Toc101860280)

[C Entwicklerdokumentation 16](#_Toc101860281)

[a. Verfügbare Klassen und Schnittstellen 16](#_Toc101860282)

[b. Nutzungshinweise 16](#_Toc101860283)

[c. Fehlerbehandlung 16](#_Toc101860284)

[D Entwicklungsumgebung 16](#_Toc101860285)

[E Verwendete Hilfsmittel 17](#_Toc101860286)

[F Erklärung 1](#_Toc101860287)

// TODO Erklärung Seitenzahl raus

**// TODO deleteme**

**Legende der Farben:**

// TODO vorher

// TODO in GroPro

# Aufgabenanalyse

## Allgemeines

// TODO

Im Rahmen dieser Arbeit soll ein Software-System entwickelt werden, welches

Das System nimmt… entgegen und bestimmt …

Das Software-System soll nach dieser Eingabe im Folgenden … ermitteln und … zurückgeben.

## Eingabe

// TODO Form und Format

Dazu müssen zunächst die Eingabeparameter eingelesen werden. Dies soll, da in der Aufgabenstellung nicht explizit angegeben, in Textdateien im Textformat stattfinden, welche die Dateiendung „.txt“ aufweisen müssen. Alle Zeilen, welche mit einem Semikolon beginnen, werden als Kommentar betrachtet und bis auf die Beschreibung des Testbeispiels in den ersten drei Zeilen der Datei ignoriert. Auch Leerzeilen werden vom Programm übersprungen.

Eine gültige Eingabedatei („ihk\_1.txt“) könnte wie folgt aussehen: // TODO

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
; Testbeispiel von Abbildung 1  
; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
; Groesse des Gebietes  
5 4  
; Hoehen  
4 5 3 2 1  
3 4 2 0 4  
2 2 4 3 2  
4 2 1 2 1

Abbildung 1: Beispiel einer gültigen Eingabedatei

// TODO

Zum Initialisieren des Programms wird die Eingabedatei zeilenweise eingelesen und die vorher beschriebenen Werte aus den Zeilen ausgelesen, geprüft und in die vorgesehene Datenstruktur überführt. Im Anschluss werden mithilfe eines … Algorithmus … bestimmt.

## Problemstellung

// TODO Was macht der Algorithmus?

## Ausgabe

// TODO

Nach Beendigung des Algorithmus soll eine Ausgabe des Ergebnisses auf zwei verschiedene Weisen erzeugt werden.

1. Ausgabe auf dem Monitor

Die Ausgabe von dem Beispiel in Abbildung 1 könnte dabei so aussehen:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Testbeispiel von Abbildung 1  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Ausdehnung in X: 5  
Ausdehnung in Y: 4  
Eingelesene Hoehen:  
4 5 3 2 1  
3 4 2 0 4  
2 2 4 3 2  
4 2 1 2 1  
Benoetigte Antennen: 1  
Antenne 1: 2 2

Abbildung 2: Beispiel einer Ausgabe in der Konsole

1. Ausgabe in eine Textdatei

Der Inhalt der Ausgabe in eine Textdatei ist identisch mit dem in a) beschriebenen und in Abbildung 2 beispielhaft aufgeführten. Die zu erstellende Ausgabetextdatei erhält den Namen der Eingabedatei mit dem Suffix „\_Ausgabe.txt“.

## Grenz- und Fehlerfälle

Während des Programmablaufs, das heißt beim Einlesen, Initialisieren, Durchführen und Ausgeben können verschiedene Grenz- und Fehlerfälle auftreten, die im Folgenden beschrieben werden.

// TODO

### Grenzfälle

1. Das kleinstmögliche (sinnvolle) Feld der Größe 2x2 wird eingegeben.
2. Das größtmögliche Feld der Größe // TODO wird eingegeben

### Fehlerfälle

Die auftretenden Fehler können üblicherweise in drei Fehlerarten aufgeteilt werden. Man unterscheidet zwischen technischen, syntaktischen und semantischen Fehlern.

#### Technische Fehler

Technische Fehler können beim Zugriff auf die Ein- und Ausgabedateien auftreten. Dabei können folgende Fehler auftreten:

1. Es wurde keine Eingabedatei angegeben.
2. Die Eingabedatei existiert nicht.
3. Die Eingabedatei ist nicht lesbar.
4. Die Ausgabedatei ist nicht beschreibbar.

Da das Anlegen einer Ausgabedatei bei technischen Fehlern unter Umständen nicht möglich ist, werden diese auf der Standardfehlerausgabe des Systems ausgegeben.

#### Syntaktische Fehler

1. Die Beschreibung des Testfalls fehlt oder ist nicht im gewünschten Format angegeben worden.
   1. Ein Kommentar ist nicht durch vorangestelltes „;“ gekennzeichnet.? TODO

Syntaktische Fehler liegen vor, wenn die Eingabedatei nicht das vorgegebene Format und in Abschnitt 1.21.2 beschriebene hat. (Die Angaben der Zeilennummern beziehen sich immer auf eine Betrachtung der Eingabe inklusive der Kommentarzeilen, abgezogen jeglicher auftretender Leerzeilen.)

1. Die Eingabedatei ist leer.

Syntaktische Fehler erzeugen eine Fehlermeldung, welche anstatt der eigentlichen Programmausgabe in die Konsole und in die Ausgabedatei geschrieben wird.

#### Semantische Fehler

1. // TODO

Semantische Fehler erzeugen ebenfalls eine Fehlermeldung, welche anstatt der eigentlichen Programmausgabe in die Konsole und in die Ausgabedatei geschrieben wird.

## Anforderungsliste

Der Übersicht halber wurden die in den vorherigen Abschnitten herausgearbeiteten Anforderungen an das Programm hier noch einmal stichpunktartig aufgeführt, um im Folgenden eine Referenz auf diese zu vereinfachen.

1. Das Programm liest Textdateien im gewünschten Format ein.
2. Bei technischen Fehlern gibt das Programm eine geeignete Fehlermeldung in der Standardfehlerausgabe des Systems aus.
3. Im Falle der genannten semantischen und syntaktischen Fehlerfälle gibt das Programm eine Fehlermeldung in der Standardfehlerausgabe des Systems und in einer Ausgabedatei aus.
4. Das Programm ermittelt // TODO
5. Das Programm gibt das ermittelte Ergebnis im gewünschten Format in der Konsole und in einer Ausgabedatei aus.

# Verfahrensbeschreibung

// TODO Verweis auf Anforderungsliste

Da die Aufgabe nun detailliert beschrieben und analysiert wurde, wird im Folgenden der Ablauf des Programms näher erläutert, wobei technische Details verfeinert und Abläufe konkretisiert werden.

## Eingabe und Initialisierung

// TODO Welche **logischen** Datenstrukturen (sprachenunabh.) und warum? zB Feld fester Größe oder verkettete Liste

Nach dem Start des Programms überprüft dieses das Vorhandensein der angegebenen Eingabedatei. Ist diese vorhanden, wird sie zeilenweise eingelesen. Sollten technische Fehler beim Einlesen auftreten, wird das Programm abgebrochen und eine Fehlermeldung ausgegeben. Die Eingabedatei wird als Parameter dem Programmaufruf mitgegeben.

Die Datei wird dann zeilenweise eingelesen, wobei Kommentar- und Leerzeilen übersprungen werden.

Wurden also keine syntaktischen oder semantischen Fehler wie in Abschnitt 1.5.2 beschrieben festgestellt, so wurden das Feld mit folgenden Werten initialisiert:

## Berechnung der …

// TODO evtl unterteilung in versch Strategien

Nach Beendigung der Eingabe und der Initialisierung der Dateistrukturen wird das erzeugte … an … übergeben, der … übernimmt.

// grober Ablauf -> TODO warum rekursiv/iterativ?

// dann genaue Beschreibung der Rekursion o.Ä.

// mathematische Formeln

// Skizzen

Die genaue Vorgehensweise beim … ist wie folgt:

## Ausgabe

// TODO Konvertierung und Ausgabe in gewünschte Formate

Anschließend an den Algorithmus kann das Ergebnis auf die zwei gewünschten Arten ausgegeben werden. Das Verfahren ist simpel:

# Programmbeschreibung

// TODO Verweis auf Anforderungsliste

// TODO Diagramme soweit es geht vorbereiten

**Anmerkung zu deutschen und englischen Begriffen**

Eigene Bezeichner, insbesondere für inhaltliche Elemente, wurden deutsch benannt. Vorkommende englischsprachige Bezeichner wurden verwendet, wenn dies geläufige Begriffe in der Softwareentwicklung sind, da so die Aufgabe der entsprechenden Komponente allgemein ersichtlich ist, beispielsweise bei Reader, Converter oder get/set/is.

## Grobentwurf

Das Programm soll, nach Ausklammerung von Ein- und Ausgabe durch eine Model-View-Controller Architektur umgesetzt werden, welche im folgenden Paketdiagramm grob skizziert wird.

### Module

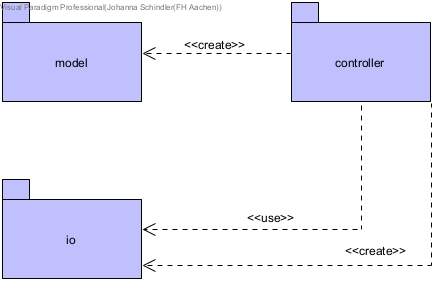


Abbildung 3: Module im Paketdiagramm

Das Model beinhaltet dabei die benötigten Daten in strukturierter Form, also …

Der Controller beinhaltet den Algorithmus inklusive eines Konvertierers für die Ein- und Ausgabe. Der Konvertierer wandelt die eingelesenen Daten in Objekte der Klassen aus dem Model Package um, erstellt diese also. Der Algorithmus operiert dann auf diesen. Eine klassische View existiert in diesem Programm nicht, da keine visuelle Anzeige des Programms in der Aufgabenstellung gefordert war. Stattdessen gibt es das Modul IO, welches Klassen beinhaltet, die für das Ein- und Auslesen zuständig sind.

// TODO Klassendiagramm für alle Klassen

### Ein- und Ausgabe

// TODO Klassendiagramm innerhalb IO (-> Exception) und EingabeAusgabeKonvertierer -> IOException

Die Ein- und Ausgabe werden durch jeweils ein Interface abstrahiert, also mittels des Strategy-Entwicklungsmusters implementiert. Dies erleichtert die Erweiterung des Programms um weitere Formen der Ein- und Ausgabe. (siehe Absatz 5.2.1)

Das erstellte Objekt der Klasse DateiReader liest in der Methode *lies()*, sofern keine technischen Fehler auftreten, beliebige Eingabedateien mit der Endung „.txt“ ein und lässt dabei Leerzeilen aus. Hier findet noch keine Syntax- oder Semantiküberprüfung statt. Die Methode gibt den gesamten Inhalt der Datei als String zurück. Der IOConverter konvertiert diesen Inhalt in ein //TODO Modelobjekt unter Berücksichtigung der Syntax- und Semantikregeln. Da von der Konvertierer - Klasse kein Objekt erstellt werden soll, sondern eine statische Methode die Konvertierung übernimmt, ist der Konstruktor der Klasse privat.

Nachdem ein Ergebnis gefunden wurde, erstellt der IOConverter die Ausgabe des //TODO Modelobjekts (Result) als String. Dieser wird dann je nach gewünschtem Ausgabetyp an ein Objekt der Klasse DateiWriter oder KonsolenWriter gereicht und die Ausgabe umgesetzt.

Da der IOConverter im Gegensatz zum DateiReader, DateiWriter und KonsolenWriter somit Kenntnis über die Bedeutung des Dateiinhaltes hat und Objekte der Klassen im Model Package erstellt, befindet er sich im Package Controller.

Beim reinen Ein- und Auslesen können durch die Entkopplung von dem eigentlichen Dateiinhalt nur Exceptions der Klasse java.io.IOException auftreten. Für die syntaktischen und semantischen Fehler, die danach bei der Validierung des Dateiinhalts auftreten können, (siehe Absatz 1.5.2) wurde die Klasse ValidierungsException implementiert, welche von java.lang.RuntimeException erbt, da sie Fehler behandelt, die zur Laufzeit auftreten können. Es handelt sich demnach nicht um Eingabe oder Ausgabefehler, wie bei der Klasse java.io.IOException, da das Lesen des Inhalts der Textdatei zu diesem Zeitpunkt bereits abgeschlossen ist. Die Klasse ValidierungsException liegt im Package Controller, ebenso wie die erstellende Klasse IOConverter.

Die verschiedenen Ausgabetypen werden durch die Enumeration AusgabeTyp benannt, da es sich hierbei um eine feste Anzahl disjunkter Ausgabetypen handelt. Das Ausgabeformat bleibt durch die gewählte Herangehensweise austauschbar, solange der Ausgabetyp dem System bekannt, also in der Enumeration AusgabeTyp und als entsprechende Implementierung von IWriter vorhanden ist.

### Model

Das Model beinhaltet …

// wofür werden die gebraucht

### Controller// TODO auf Problem anpassen

Das Interface IStrategie abstrahiert die Schnittstelle, welche die Klasse BacktrackingStrategie bereitstellen muss. Die Strategie Controller sind ebenfalls im Strategy- Entwicklungsmuster entworfen worden, damit im Falle einer anderen Lösungsstrategie diese leicht einzufügen ist.

Bei den Methoden handelt es sich um einen Konstruktor, der … erhält und die Ausführ-Methode, die im Falle des Backtrackings eine private, rekursive Methode aufruft und dann das Result zurückgibt. BacktrackingStrategie hat dabei außerdem weitere Hilfsmethoden, die zur Berechnung … benötigt werden. Die Klasse Programm übernimmt den kompletten Programmablauf, das heißt sie steuert nach Aufruf der Methode *starteProgramm()* die Eingabe, das Starten des Algorithmus und die Ausgabe. Kommt der Algorithmus zu keinem Ergebnis, muss ein Fehler vorliegen und eine Exception der Klasse AlgorithmusException wird erstellt. Diese liegt wie die aufrufende Klasse im Package Controller.

## Ablauf

### Gesamtablauf

Der Ablauf des Programms lässt sich wie folgt als UML-Sequenzdiagramm beschreiben. Die Klasse Main übernimmt dabei die Aufgabe, das eigentliche Programm mit dem Eingabedateinamen zu starten und ist an sich trivial, sodass auf ein Klassendiagramm verzichtet wurde.

Zunächst wird der Gesamtablauf der Programmausführung vereinfacht dargestellt. Vor allem die Initialisierung und die Abläufe innerhalb des Backtracking-Algorithmus sind komplexer als hier dargestellt. Diese werden daher anschließend noch mittels Aktivitätsdiagrammen präzisiert, um die Sequenzdiagramme möglichst übersichtlich und verständlich zu halten.

//TODO SD

### Initialisierung des Programms

//TODO SD

### Strategie

//TODO SD

## Algorithmen // TODO wenn Plural unterteilen

//TODO AD

## Ein- und Ausgabekonvertierung

//TODO ADs

### Eingabekonvertierung

### Ausgabekonvertierung

# Testdokumentation

In diesem Kapitel wird das Programm an einem ausgesuchten Beispiel ausführlich getestet, sowie die weiteren Testbeispiele erklärt und begründet.

## Begründung und Diskussion der Testfälle

Die Testfälle lassen sich anhand des Präfix des Dateinamens kategorisieren.

|  |  |
| --- | --- |
| **n\_** | Tests für einfache Normalfälle, bei denen … |
| **g\_** | Tests für in Unterabschnitt 1.5.1 beschriebene Grenzfälle |
| **syn\_** | Tests für in Unterabschnitt 1.5.2.2 beschriebene syntaktische Fehlerfälle |
| **sem\_** | Tests für in Unterabschnitt 1.5.2.3 beschriebene semantische Fehlerfälle |
| **ihk\_** | vorgegebene Testbeispiele der Aufgabenstellung, beinhalten Normal- und Sonderfälle |

Die Nummerierung der Grenz- und Fehlerfälle entspricht der in Abschnitt 1.5 verwendeten.

### Grenzfalltests

Die in Abschnitt 1.5.1 beschriebenen Grenzfälle werden durch die folgenden Tests geprüft.

// TODO Grenzfälle

|  |  |
| --- | --- |
| **Grenzfall 1** | wird durch die Datei „g1\_minimal.txt“ getestet.  Die Eingabe des kleinstmöglichen Feldes soll als gültig erkannt werden und …. (trivial) |
| **Grenzfall 2** | wird durch die Datei „g2\_maximal.txt“ getestet.  Die Eingabe des größtmöglichen Feldes soll als gültig erkannt werden…. |

### Fehlertests

Die in Abschnitt 1.5.2 beschriebenen syntaktischen und semantischen Fehlerfälle werden durch die folgenden Tests geprüft.

// TODO Beschreibung „Der Test soll zeigen dass“

// TODO Fehlermeldungen anpassen

|  |  |
| --- | --- |
| **Syntaktischer Fehler 1** | wird durch die Datei „syn1\_kommentarFehlt.txt“ getestet. Der Test soll zeigen, dass das Programm erkennt, dass die Beschreibung fehlt, wenn keine Zeile im gewünschten Format erkannt werden kann. Dann soll die folgende Fehlermeldung in die gewünschten Ausgabemedien schreibt.  **"Syntaxfehler: Beschreibung fehlt oder ist nicht korrekt als solche gekennzeichnet. Die Beschreibungszeile beginnt mit //+."** |
| **Syntaktischer Fehler 2** |  |

### Technische Fehler

Die in Abschnitt 1.5.2.1 beschriebenen technischen Fehlerfälle lassen sich aus technischen Gründen nicht als Ein- und Ausgabedateien der Abgabe beifügen. Aus diesem Grund werden hier die resultierende Ausgabe in der Konsole und die entsprechenden Aufrufe aufgeführt.

// TODO konkrete Fehlermeldungen benennen

|  |  |
| --- | --- |
| **Technischer Fehler 1** | Der Test soll zeigen, dass das Programm das Fehlen einer Eingabedatei erkennt und den Benutzer mit der folgenden Fehlermeldung in der Konsole informiert.  **"Eingabe/Ausgabe Fehler: Eingabedatei existiert nicht"**  java -jar ./run/Programm.jar |
| **Technischer Fehler 2** | Der Test soll zeigen, dass das Programm erkennt, dass die angegebene Eingabedatei nicht existiert und den Benutzer mit einer passenden Fehlermeldung informiert.  **"Eingabe/Ausgabe Fehler: Eingabedatei existiert nicht"**  java -jar ./run/Programm.jar existiertNicht.txt |
| **Technischer Fehler 3** | Der Test soll zeigen, dass das Programm erkennt, dass die Leserechte für die angegebene Eingabedatei fehlen und den Benutzer mit einer passenden Fehlermeldung informiert.  **"Eingabe/Ausgabe Fehler: beim Lesen der Eingabedatei"**  java -jar ./run/Programm.jar nichtLesbar.txt |
| **Technischer Fehler 4** | Der Test soll zeigen, dass das Programm erkennt, dass die Schreiberechte zur passenden Ausgabedatei zur Eingabedatei fehlen und die Ausgabe nicht geschrieben werden kann. Darüber soll der Benutzer mit einer passenden Fehlermeldung informiert werden.  **"Eingabe/Ausgabe Fehler: keine Schreibrechte auf Ausgabedatei"**  java -jar ./run/Programm.jar nichtSchreibbar.txt |

Bei sonstigen unerwarteten Fehlern beim Lesen oder Schreiben einer Datei wird die folgende Fehlermeldung in die Konsole geschrieben:

**"Eingabe/Ausgabe Fehler: beim Lesen/Schreiben der Eingabedatei"**

### Normalfälle

Normalfälle sind zum einen durch die Beispiele aus der Aufgabenstellung gegeben, welche durch die Eingabedateien „ihk\_1.txt“, „ihk\_2.txt“, „ihk\_3.txt“ getestet wurden. Weitere Normalfälle sind //TODO.

// TODO detaillierte Diskussion

|  |  |
| --- | --- |
| **IHK-Beispiel 1** | bildet dabei den Fall, dass … ab. Der Test soll zeigen, dass der Algorithmus … findet. |
| **IHK-Beispiel 2** | bildet dabei den Fall, dass … ab. Der Test soll zeigen, dass der Algorithmus … findet. |

Die weiteren Beispiele // TODO Hier soll getestet werden, ob eine einfache Lösung auch bei größeren Feldern schnell gefunden werden kann. // TODO Ergebnis?

## Ausführliches Beispiel

Als ausführliches Beispiel wird die Eingabedatei … verwendet, da ...

### Eingabe

// TODO

### Algorithmen // TODO wenn mehrere unterteilen

// TODO

### Ausgabe

// TODO

# Zusammenfassung und Ausblick

## Zusammenfassung

Das Programm erfüllt die in der Aufgabenstellung definierten Anforderungen und wurde in Bezug auf Funktionalität und Fehlererkennung ausführlich getestet. Durch die hohe Laufzeitkomplexität des Backtracking-Algorithmus ist die Bearbeitung großer Felder jedoch sehr rechenaufwendig, wodurch das Testen des Grenzfalls der maximalen Fläche lediglich der Größe // TODO möglich war.

## Ausblick

Für das Programm gibt es diverse Erweiterungsmöglichkeiten in verschiedenen Bereichen.

### Ein- und Ausgabe

Die bisherige Ein- und Ausgabe über Textdateien im vorgegebenen Format lässt sich durch die Verwendung des Strategy-Entwicklungsmusters und die Trennung von IO und Konvertierung leicht um weitere Formate erweitern. Dies ist durch zusätzliche Implementierungen der Interfaces IReader und IWriter möglich.

Denkbar wären

|  |  |
| --- | --- |
| **Ausgabe als XML** | Dazu müsste die Implementierung von IWriter einen String mit XML in eine „.xml“ Datei schreiben. Die Methode zur Erzeugung des Strings muss dafür dem IOConverter hinzugefügt werden. |
| **Ausgabe in eine Datenbank** | Dazu müsste die Implementierung von IWriter einen String einem SQL-Query an eine Datenbank absetzen. Die Methode zur Erzeugung des ein SQL-Queries muss dafür dem IOConverter hinzugefügt werden. |

### Strategien TODO

Neben der verwendeten Backtracking-Strategie wäre noch eine Vielzahl weiterer möglich. Diese müssten ebenfalls das Interface IStrategie implementieren. Diese leichte Erweiterbarkeit wurde auch hier durch das Verwenden des Strategy-Entwicklungsmusters erreicht.

Zwar ermittelt der Backtracking-Algorithmus in jedem Fall das bestmögliche Ergebnis, jedoch ist er sehr rechenaufwendig. Gerade bei größeren Flächen kann es daher sinnvoll sein, auf andere Verfahren auszuweichen, welche zu Lasten der bestmöglichen Lösung schneller ein Ergebnis erzeugen.

# Abbildungsverzeichnis // TODO aktualisieren

[Abbildung 1: Beispiel einer gültigen Eingabedatei 4](file:///C:\Users\Johanna\Documents\FH\6_Semester\IHK\GroPro\GroPro_Dokumentation_Vorlage.docx#_Toc101270722)

[Abbildung 2: Beispiel einer Ausgabe in der Konsole 2](file:///C:\Users\Johanna\Documents\FH\6_Semester\IHK\GroPro\GroPro_Dokumentation_Vorlage.docx#_Toc101270723)

[Abbildung 6: Module im Paketdiagramm 6](#_Toc101270724)

1. Abweichungen und Ergänzungen zum Vorentwurf
2. Benutzeranleitung
3. Verzeichnisstruktur

Im angegebenen Archiv befinden sich folgende Verzeichnisse:

|  |  |
| --- | --- |
| **run** | beinhaltet Skripte zum Kompilieren des Programms und zum Ausführen der Testfälle, sowie das kompilierte Programm |
| **src** | enthält den Quellcode des Programms |
| **doc** | enthält diese Dokumentation, sowie die Java API-Dokumentation, auf die in Anhang C verwiesen wird |
| **testfaelle** | beinhaltet die ausgeführten Testbeispiele als Ein- und Ausgabedateien |

1. Systemvorraussetzungen

Um das Programm verwenden zu können, wird ein Microsoft Betriebssystem in der Version 10 oder höher, sowie eine Java Laufzeitumgebung (JRE) in der Version 16 benötigt. Für das Kompilieren des Programms muss zusätzlich das Java Development Kit (JDK) in der Version 16 oder höher installiert sein.

Sollten diese nicht oder in veralteter Version installiert sein, kann ein Download der JRE und JDK in gewünschter Version unter der folgenden Adresse geschehen:

<https://www.oracle.com/java/technologies/downloads/>

Um das Programm, wie in Abschnitt d beschrieben, zu kompilieren, also für alle „.java“-Dateien zunächst „.class“-Dateien und schließlich die Projekt „.jar“-Datei zu erstellen, muss das Tool Maven installiert sein. Nachdem die Java SDK bereits installiert wurde, kann Maven unter der folgenden Adresse in der Version 3.8.1 als „.zip“ heruntergeladen werden. Dafür müssen 10MB Speicher für die Installation und ungefähr 500MB für das lokale Repository frei sein.

<https://maven.apache.org/download.cgi>

Die „.zip“-Datei muss danach in einen geeigneten Ordner entpackt werden und der Speicherort des Unterordners /bin in den PATH-Variablen des Systems hinterlegt werden.

1. Installation

Der gesamte Inhalt der vorliegenden „.zip“-Datei ist in ein beliebiges, beschreibbares Verzeichnis zu kopieren. Danach ist das Programm betriebsbereit. Das ausführbare Skript, das alle Testfälle nacheinander ausführt, liegt im Verzeichnis run, genauso wie die „.jar“-Datei zur Ausführung einzelner Testbeispiele.

1. Kompilieren

Um den Quellcode des Programms in eine ausführbare Datei zu kompilieren, muss im Terminal auf Wurzelebene des Projektes der Befehl mvn clean install durchgeführt werden. Dadurch wird ein eventuell bestehender „target“-Ordner gelöscht und die benötigten Dateien inklusive der „.jar“ -Datei im Ordner „target“ hinterlegt. Diese muss im Anschluss in das Verzeichnis /run kopiert oder verschoben werden und zu „Programm.jar“ //TODO umbenannt werden. Damit ist das Programm kompiliert.

1. Programmausführung

//TODO

Um einzelne Testdateien zu verarbeiten, muss das Programm über die Kommandozeile (cmd) ausgeführt werden. Der Aufruf aus dem Wurzelverzeichnis des Projektes erfolgt über den Befehl:

java -jar ./run/Programm.jar <pfad/zur/inputdatei.txt>

1. Testausführung

Um alle in testfaelle/input vorliegenden Testdateien nacheinander zu verarbeiten, kann das Skript run/RunProgram.bat genutzt werden. Mittels Doppelklick auf die Datei im Windows Explorer startet die Ausführung.

Zum Öffnen des Ordners /run [hier klicken](../run).

Zum Testen von bestimmten weiteren, bereits im Verzeichnis testfaelle/input hinterlegten Beispielen muss das Programm wie in Absatz B.e beschrieben ausgeführt werden.

// TODO falls sehr lange dauert Hinweis

1. Entwicklerdokumentation
2. Verfügbare Klassen und Schnittstellen

Die Dokumentation der verfügbaren Klassen und Schnittstellen wurde mit Hilfe des Tools „javadoc“ aus dem Quellcode erzeugt und im mitgelieferten Archiv im Verzeichnis doc/javadoc abgelegt. Zur Einsicht muss die Datei doc/javadoc/index.html in einem Webbrowser geöffnet werden.

Zum Öffnen der index.html [hier klicken](javadoc/index.html).

1. Nutzungshinweise

Im Folgenden wird exemplarisch erklärt, wie das Programm angepasst und erweitert werden kann.

* **Strategien**

Die Strategien sind im Paket controller zu finden. Sollen die bisherigen Strategien //TODO durch weitere ergänzt werden, so geschieht das durch eine Änderung der Methode Programm:*laufeProgramm()*, in der die weiteren Strategien auf gleiche Art wie die bisherigen verwendet werden können. Diese weiteren Strategien müssen das Interface IStrategie implementieren.

* **Ein- und Ausgabe**

Die Funktionalität für Ein- und Ausgabe befindet sich im Paket io. Die Programmiererschnittstelle wird durch die Interfaces IReader fürs Lesen und IWriter für das Schreiben des Dateiinhaltes realisiert. Bisherige Implementierungen sind DateiReader, DateiWriter und ConsoleWriter. Die Erweiterung des Programms für andere Ein- und Ausgabeformate wird über eine weitere Implementierung dieser Interfaces realisiert.

**Wichtig: Die Implementierung von IWriter legt nur das Ausgabemedium, also zum Beispiel Datei oder Konsole fest. Die Konvertierung ins Ausgabeformat wird im Folgenden erklärt.**

* **Ausgabeformat**

Das Ausgabeformat wird durch die Methode IOConverter:*convertResultToOutput()* bestimmt. Soll dieses verändert werden oder weitere Formate ergänzt werden, muss das also in der Klasse IOConverter geschehen. Der Aufruf der Konvertierungsmethode geschieht im Programm.

* **Model**

Die Klassen für die in der Strategie benötigten Datenstrukturen wurden im Paket model abgelegt. Diese müssen für eigene Implementierungen von IStrategie, beziehungsweise IReader und IWriter importiert werden.

1. Fehlerbehandlung

Die Fehlerbehandlung beim Initialisieren des Programms ist im Falle von technischen Fehlern Aufgabe der Implementierung von IReader und im Falle von syntaktischen oder semantischen Fehlern von der Klasse IOConverter. Die Klassen werfen im Fehlerfall eine spezielle Exception: java.io.IOException bei klassischen Lese- und Schreibefehlern und ValidierungsException bei syntaktischen oder semantischen Fehlern. Die Klasse ValidierungsException erbt dabei von java.lang.RuntimeException, da es sich um Laufzeitfehler handelt. Sollte der Algorithmus zu keinem Ergebnis kommen, wirft die Implementierung von IStrategie eine Exception der Klasse AlgorithmusException, welche ebenfalls von java.lang.RuntimeException erbt.

Die Fehlermeldung, welche als String im ersten Parameter des Konstruktors in die Exception gegeben wird und über die Methode *getMessage()* ausgelesen werden kann, wird bei ValidierungsException und AlgorithmusException dann in die Konsole und die Ausgabedatei geschrieben, bei einer java.io.IOException nur in die Konsole. Diese Konvention muss von zusätzlichen Implementierungen des Interfaces IStrategie und bei einer Veränderung von der Klasse Programm // TODO beibehalten werden.

1. Entwicklungsumgebung

Programmiersprache : Java

Compiler : OpenJDK javac 16

Rechner : Dell Latitude 5500

Betriebssystem : Windows 11 Pro

1. Verwendete Hilfsmittel

* IntelliJ 2021.1

Entwicklungsumgebung für Java

<https://www.jetbrains.com/de-de/idea/>

* Maven 3.8.1

Projektmanagement-Tool

<https://maven.apache.org/download.cgi>

* Visual Paradigm Professional 16.3

UML-Tool

<https://www.visual-paradigm.com/>

* Microsoft Word

Texteditor

<https://www.microsoft.com/de-de/download/office.aspx>

1. Erklärung

Erklärung des Prüfungsteilnehmers:

Ich erkläre verbindlich, dass das vorliegende Prüfprodukt von mir selbstständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. 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 Prüfprodukts als Prüfungsleistung ausschließt.

// TODO Unterschrift

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ort, Datum Unterschrift des Prüfungsteilnehmers