Abschlussprüfung Sommer 2019 zum
Mathematisch- technische/r Softwareentwickler/in (IHK)

Entwicklung eines Softwaresystems

VEREINFACHTE TEXTEINGABE AM HANDY

17. Juni 2020

Maher Albezem

Prüflingsnummer : XXXXX
Programmiersprache : Java
Ausbildungsbetrieb : SLA RWTH Aachen

Eidesstattliche Erklärung

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 Prüfprodukts als Prüfungsleistung ausschließt.

Aachen, den 17. Juni 2020

Maher Albezem, XXX XXXXX

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Aufgabenanalyse

Im Rahmen der Aufgabenstellung ist ein Programm zur vereinfachten Texteingabe mit Hilfe einer numerischen Tastatur zu entwickeln. Eine Software, die dieses Problem löst, ist unter dem Namen T9 bekannt. Die Tastatur verfügt nur über folgende Tasten: 1234567890*# . Die möglichen Eingaben des Programms "T9ßind 12 unterschiedliche Buchstaben in der Menge $E := \{1, 2, ..., 9, 0, *, \#, Enter\}$. Die Enter-Taste wird in allen Konsolenprogrammen benötigt, um Eingaben zu bestätigen.

Mit der Menge E sollen Wörter bzw. Sätze in Großbuchstaben geschrieben und mit Hilfe eines Wörterbuchs erkannt werden. Die Zuordnung der Buchstabe nzu den numerischen Taste ist wie folgt festgelegt:

␣	ABC	DEF
1	2	3
GHI	JKL	MNO
4	5	6
PQRS	TUV	WXYZ
7	8	9
Ja	.	*NEIN*
*	0	#

Das Programm kann dynamisch erweitert werden, indem man Wörter zum Wörterbuch hinzufügt.

1.1 Algorithmus

Bei der Eingabe wird zwischen zwei Modies unterschieden: Einfachmodus und Explizitmodus. Die beiden Modus unterscheiden sich in der Interpretation der

Eingabemenge E. Das Programm bildet beliebig viele Sätze mit Einschränkung der möglichen Eingaben (s. Restriktionen). Alle Einträge im Wörterbuch, die nur per Explizitmodus gespeichert worden sind, werden Extern in einer Textdatei gespeichert und bei jedem erneuten Ausführen des Programms wieder Aufgerufen und für Textsuche verwendet.

1.1.1 Modus der normalen Texteingabe

Für die Texteingabe wird pro Buchstabe nur eine Zifferntaste gedrückt.

Beispieleingabe:

S	O	F	T	W	A	R	E	␣
7	6	3	8	9	2	7	1	1

Beim Eingeben der Leertaste 1 oder der Punkt 0 wird Ende des Wortes bzw. des Satzes signalisiert. Das Wort wird anhand der Ziffernfolge im Wörterbuch gesucht. Falls mehrere Wörter die gleiche Ziffernfolge haben, wird das Häufigste Wort vorgeschlagen. Falls kein Eintrag im Wörterbuch gefunden wurde, wird der Nuzter zum Explizitmodus weitergeleitet.

1.1.2 Explizitmodus

Im Explizitmodus wird keine Suche im Wörterbuch stattfinden. Der Nuzter wird stattdessen aufgefordert, Wörter zum Speichern im Wörterbuch einzugeben. Die Eingabe wird explizit erfolgen. D.h. die genauen Buchstaben werden eingegeben.

Beispieleingabe:

W	O	R	T	␣
91	63	73	81	1

Hier signalisieren auch Leertaste 1 oder Punkt 0 Ende des Wortes bzw. des Satzes. Für jede Buchstabe werden zwei Ziffern benötigt. Eine für Ort im Ziffernblock und die Zweite für den Ort innerhalb der gewählten Ziffernblock.

1.1.3 Wörter und Sätze

Das Programm unterscheidet zwischen Wörter und Sätze. Falls die Eingabe mit einem Punkt(0) endet wird Ende des Satzes signalisiert und im Anschluss das ganze Satz

gezeigt. Falls die Eingabe mit einer Leertaste(1) endet wird Ende des Wortes signalisiert und im Anschluss weiergeschrieben bis Ende des Satzes eingegben wird.

1.1.4 das Wörterbuch

Alle Einträge des Wörterbuchs werden in einer HashMap sowie einer Baumstruktur gespeichert. Der Schlüssel der jeweiligen Einträge ist der explizite Zifferncode aus der Menge {2, 3, 4, 5, 6, 7, 8, 9} Wobei Leertast (0) oder Punkt(0) nicht gespeichert werden. Somit sind die Einträge im Wörterbuch eineindeutig abgebildet und somit einfach zu finden. Am Anfang bzw. am Ende des Programmablaufs wird das Wörterbuch in eine Textdatei gespeichert bzw. gelesen. Beim speichern werden alle Einträge des Wörterbuchs Zeilenweise in der Textdatei geschrieben. Eine Zeile hat folgendes Muster:
<Code><Wort><Häufigkeit>
Beispile: 91637333 WORF 2

1.1.5 Benutzerdialog im Hauptprogramm

Die Benutzereingabe bzw. Ausgabe erfolgt mit einer Konsole. Die Taste Enter beendet die Eingabe.

Falls ein Satz nur mit einem Punkt eingegeben wurde, wird das Programm beendet.

1.2 Restriktionen

1.2.1 Restriktionen der Eingabedatei

Eine Textdatei enthält Zeilenweise gespeicherte Einträge vom Wörterbuch. Solche Textdatei muss folgende Restriktionen erfüllen:

- Die Eingabedatei muss existieren und lesbar sein, sonst wird kein Wörterbuch geladen.
- Die Eingabedatei soll in Form einer Textdatei mit lebarer ASCII-Text sein.
- Pro Eintrag des Wörterbuchs wird eine Zeile in der Textdatei gespeichert und durch Zeilenvorschub getrennt.
- Das Muster Pro Eintrag muss wie im Abschnitt 1.1.4 (das Wörterbuch) beschrieben übereinstimmen

- Die Eingabedatei darf keine doppelte Einträge enthalten
- Die Textdatei muss mit einem normalen Texteditor bearbeitbar sein.
- Die Ziffernfolge für jeden Eintrag muss mit der Menge der möglichen Eingaben im Benutzerdialog enthalten sein.

Falls die Eingabedatei nicht konform mit der oben genannten Punkten, wird die Datei nicht geladen. Andernfalls wird die im Programm angegebene Textdatei geladen.

1.2.2 Restriktionen der Benutzereingabe

Wie oben in den Abschnitten **Modus der normalen Texteingabe** und **Explizitmodus** angegeben haben die Konsoleneingaben folgende Restriktionen:

- Alle Eingaben sind nur aus der Menge $E := \{1, 2, ..., 9, 0, *, \#, Enter\}$
- Eine Explizite Ziffernfolge muss konform mit der vorgeschriebenen Zuordnung der Tastatur in der Aufgabenanalyse
- Kein Wort sollte aus nur einer Leertaste (1) oder nur einem Punkt (0) bestehen

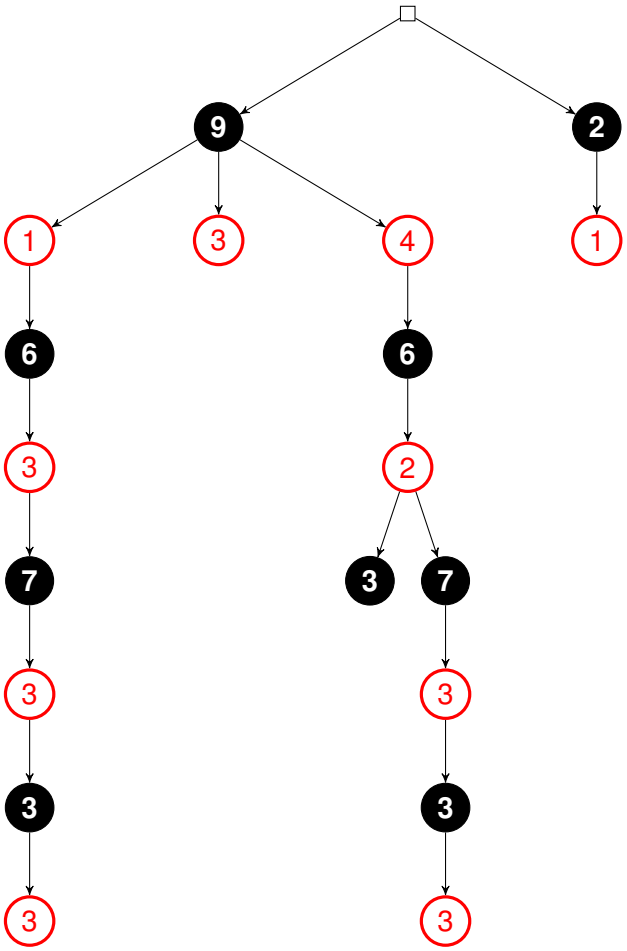
Kapitel 2

Verfahrensbeschreibung

Das Programm hat zwei elementare Funktionalitäten. Die Suche nach Wortvorschläge im Wörterbuch anhand einer einfachen Ziffernfolge sowie die Filterung und die Interpretation von Eingaben von Expliziten Ziffernfolgen.

2.1 Die Suche im normalen Modus der Texteingabe

Für eine effiziente Suche im Wörterbuch wird eine Datenstruktur eines Baumes verwendet. Für jede Ziffer in der expliziten Ziffernfolge eines Wortes wird einen Knotenpunkt im Baum erstellt. Dies vereinfacht die Suche erheblich im Vergleich mit der Datenstruktur einer Tabelle. Ein Beispielbaum mit folgenden gespeicherten Wörtern {ZNRF, WORF, ZO, Z, Y, A}



Die Knotenpunkte im Baum Bilden die Menge der expliziten Ziffernfolgen eineindeutig ab. Alle Knoten mit der Endziffer eines Wortes enthält die Häufigkeit des Wortes als integer Zahl gespeichert. Schwarze und rote Knoten im Baum sind identisch. Bei der Suche im Baum werden alle schwarzen Knoten besucht, die einen identischen Index mit der vorligenden Ziffer haben. Die roten Knoten werden für die Identifikation des Wortes benötigt. Deswegen werden die roten Knoten direkt unterhalb eines besuchten schwarzen Knoten immer besucht. Die roten Knoten sind die Endziffer eines Wortes. Aus diesem Grund sind die Häufigkeiten (n) in diesem Knoten als Integerzahl gespeichert. Alle anderen Knoten haben die Häufigkeit (n=0).

Die Suche im Baum erfolgt nach dem Prinzip von Sammelkorb. Alle Endknoten der gefundenen Wörter werden gesammelt und anschließend ihre Häufigkeiten verglichen. Der Knotenpunkt mit max(n) wird zurückgegben. Um aus diesem Knoten ein Wort zu bilden, durchläuft das Programm die Knoten rückwärts bis zum Wurzel und speichert dabei die Indexen der Knoten. Die sich ergebende Ziffernfolge wird umgekehrt und in einem Wort umgewandelt. Falls alle gefunden Knoten die Häufigkeit (n=0) haben, ist kein Wort gefunden, die die Ziffernfolge entspricht.

2.2 Wortbildung und Interpretation

Das Wörterbuch ist in der Lage, Wörter als explizite Ziffernfolgen als Buchstaben in Großschreibung zu interpretieren. Für diese Umwandlung von Ziffern zu Buchstaben sind if-else-Abzweigung nötig. Diese Unterscheiden dann die Ziffernfolgen voneinander. Um festzustellen, dass die Eingaben der Nutzer auf der Konsole stimmig mit der Anforderungen sind, werden sie in der Konsole zuerst von Punkten oder Leertasten am Ende gefiltert und nach Richtigkeit gecheckt.

Kapitel 3

Programmkonzeption

3.1 Klassen

Das Programm beinhaltet folgende Klassen:

3.1.1 `Main`

Die Hauptklasse für das Programm T9. Diese Klasse beinhaltet die Einsprungfunktion des gesamten Programms. Es Wird ein Wörterbuch sowie eine Konsole erstellt. die Konsole wird gestartet und nach Ende der Nutzerinteraktion wieder beendet.

3.1.2 `Konsole`

Diese Klasse ist die Konsole für die Nutzerinteraktion. Beim Starten der Konsole wird das Wörterbuch aus einer Datei geladen. Bei diesem Vorgang wird der Nutzer benachrichtigt, ob das Wörterbuch geladen ist oder nicht. Die Konsole verwaltet die Eingaben und Ausgaben des Programms. Die Eingaben vom Nutzer werden gefiltert und nach Richtigkeit geprüft. Im Falle einer Falschen Eingabe, wird ein Fehler auf der Konsole mit folgender Form gezeigt:
Falsche Eingabe
<Grund>
Mögliche Ausgaben:

1. Kein Woerterbuch geladen
2. Woerterbuch geladn

3. Kein passendes Wort gespeichert. Eingabe im Explizitmodus:
4. <Wort-Vorschlag>
5. Wort ok?
6. Wort gespeichert
7. Der Satz lautet <gebildeter Satz>
8. Programmende.

Alle Ausgaben mit den <>-Klammern sind dynamisch.

3.1.3 `Woerterbuch`

Diese Klasse repräsentiert das Wörterbuch des Programms T9. Die Klasse beinhaltet eine Tabelle in Form einer HashMap, um die Speicherung des Wörterbuchs sowie das Lesen aus einer Textdatei zu vereinfachen. Die Klasse beinhaltet einen *Baum* für die Textsuche, eine integer Zahl als festgelegte Größe des Wörterbuches und eine Pfadangabe als String für die Textdatei des Wörterbuchs. Die Methode `ladeBib` lädt das Wörterbuch aus der Textdatei. Dabei wird bei Fehler keine Datei geladen. Die Methode `speichereBib` speichert das Woerterbuch in der Textdatei. Bei Fehler wird nicht gespeichert und keine Fehlermeldung zurückgegeben. Für die Suche sowie Hinzufügen eines Wortes wird in der Klasse `Baum` die Suchfunktion sowie die `add`-Funktion aufgerufen. `add` fügt ein Wort sowohl im HashMap als auch im Baum. Beim Erstellen eines Baumes wird die Methode `makeTree` aufgerufen, die ein Baum für die vereinfachte Suche erstellt. Dabei werden alle Einträge im HashMap gelesen und im Baum gespeichert. Die Verwandlung von einer expliziten Ziffernfolge in einem Wort im Alphabet übernimmt die statische Methode `makeWord`. Diese Methode wird auch in der Klasse `Baum` verwendet, um gesuchte Ziffernfolgen umzuwandeln.

3.1.4 `Baum`

Beinhaltet alle Einträge des Wörterbuchs im Form von Baumstruktur. Alle Knoten sind von der Klasse `Node`. Der Wurzelknoten ist eine ArrayList **root**, die alle Knoten enthält. `search` bekommt ein **key**, ein String von Ziffernfolge, und sucht rekursiv nach einem Treffer im Baum. Falls kein Treffer gefunden ist wird der String -1 zurückgegben. Um Wörter aus einem Endknoten zu konstruieren, wird `constructWord` mit einem Knoten aufgerufen. Diese Methode sucht alle Elternknoten der Reihe nach und speichert die Indexen der Knoten in einem String. Der String wird rückwärts umgewandelt und zu einem Wort mit Hilfe von `makeWord` interpretiert. Um ein Wort im Baum hinzufügen wird `add` aufgerufen. Diese Methode Wird rekursiv aufgerufen um Knoten im Baum hinzufügen oder falls schon vorhanden aktualisieren. Die Methode bekommt also eine

explizite Ziffernfolge und ein Häufigkeit des Wortes. Falls keine Häufigkeit von einem Wort existiert wird die Methode 0 bekommen. Falls die Häufigkeit 0 ist, wird jedesmal, wenn ein bereits im Baum gespeichertes Wort erneut ausgewählt wird, wird seine Häufigkeit um eins erhöht.

3.1.5 `Node`

Die Klasse repräsentiert einen Knoten im `Baum`. Sie beinhaltet Integerzahl `n` als Häufigkeit sowie ein Integerzahl als Index, ArrayList aus Node für allen Kindknoten und ein Node für einen Elternknoten. Die Methode `has` prüft nach Existenz eines Kindknotens und die Methode `get` liefert einer der Kindknoten mit einem eingegebenen Index.

3.2 UML-Klassendiagramm

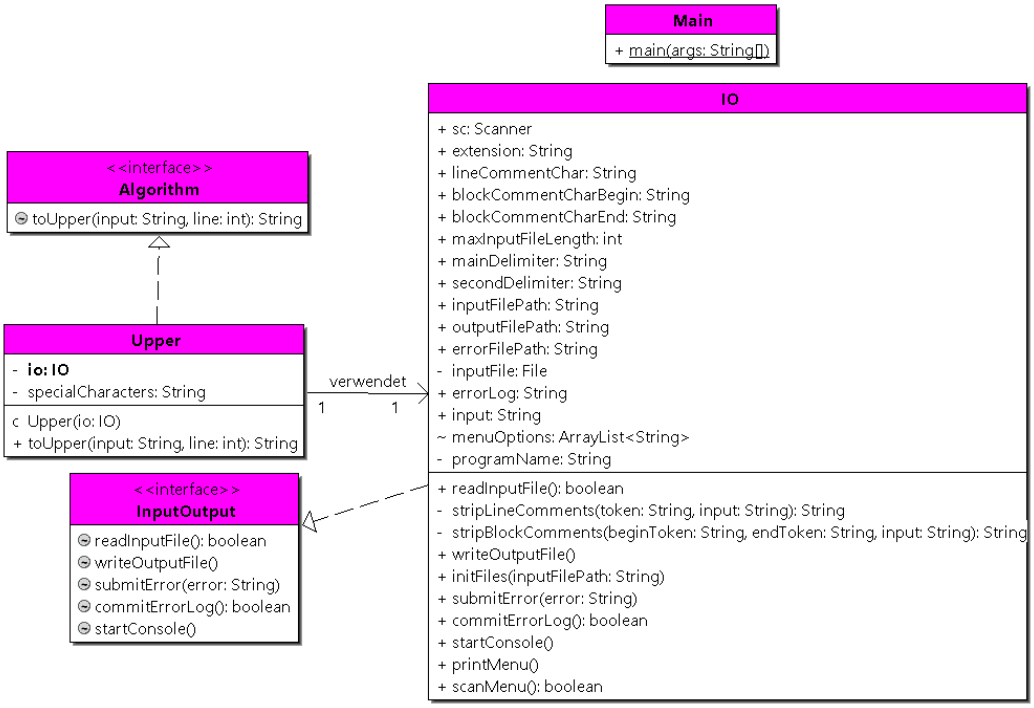


Abbildung 3.1: UML-Klassendiagramm der implementierten Klassen

3.3 Aktivitätsdiagramm zum Programmablauf

Eine detaillierte Beschreibung des Programmablaufs anhand eines Aktivitätsdiagramm. Dies verdeutlich den Lebenszyklus mit den Kontroll-Beziehungen.

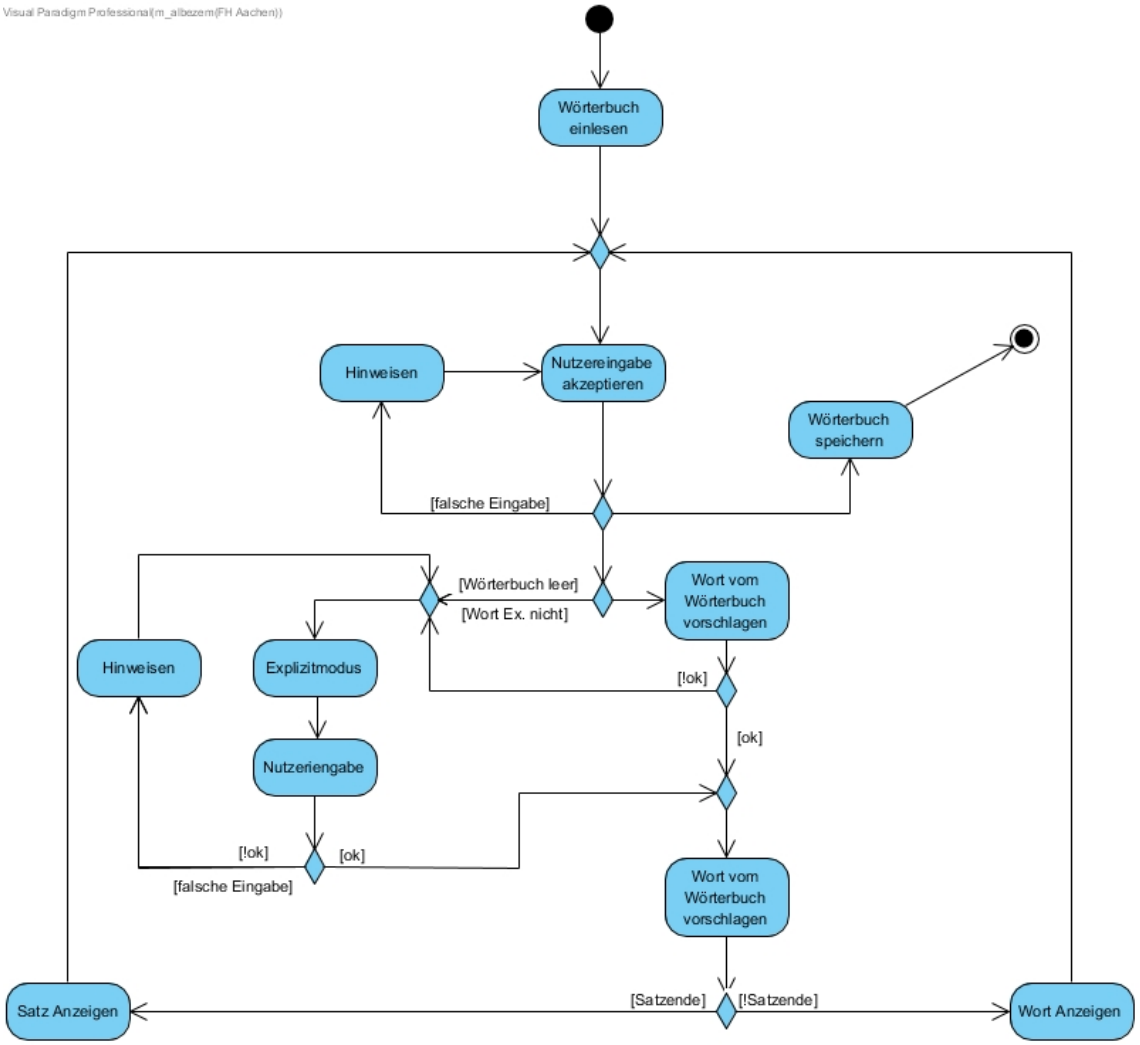


Abbildung 3.2: UML-Aktivitätsdiagramm zum Programmablauf

3.4 Beschreibung der Methoden

Eine detailierte Beschreibung der Methoden folgt in Form von Nassi-Schneiderman-Diagrammen bzw. Struktogrammen.

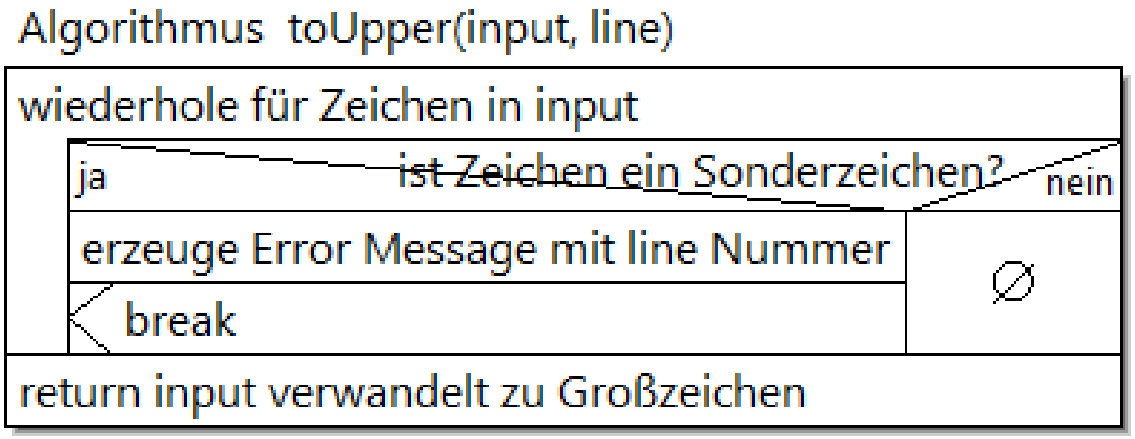


Abbildung 3.3: Algorithmus zur Umwandlung von Buchstaben in einem String zu Großbuchstaben

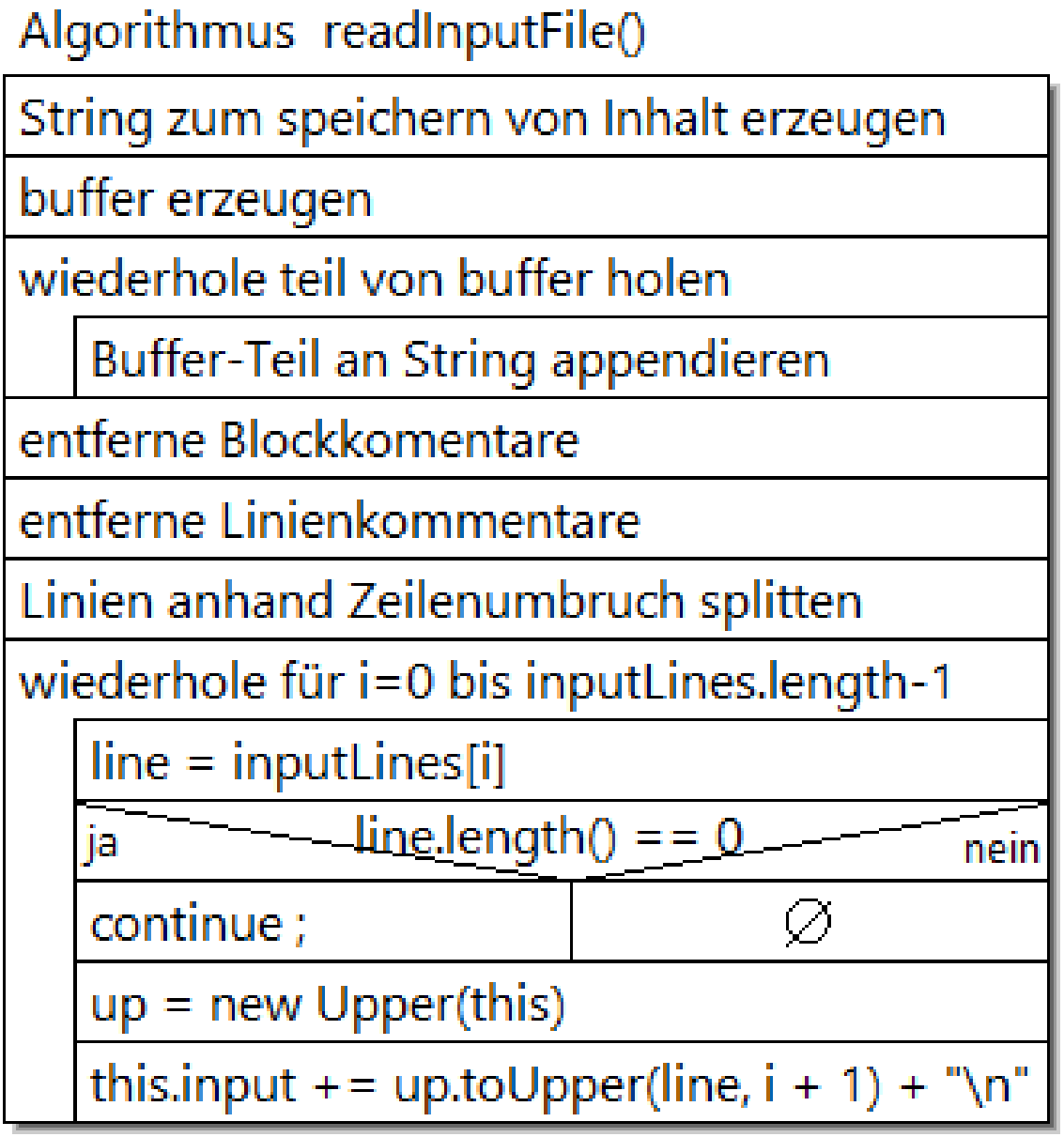


Abbildung 3.4: Algorithmus zum Lesen von Input-Datei

Kapitel 4

Entwicklungsumgebung

Die Entwicklung und Tests der Software wurden auf einem System mit folgender Spezifikation durchgeführt:

- Betriebsystem: Microsoft Windows 10 Enterprise (64 Bit)
- Prozessor: AMD FX(tm)-8350 Eight-Core Processor, 4000 MHz, 4 Kern(e), 8 logische(r) Prozessor(en)
- Installierter physischer Speicher (RAM) 16,0 GB
- Grafikkarte: NVIDIA GeForce 210

Weitere Spezifikationen:

- Die Software wurde mit der Programmiersprache Java in der Version 11.0.2 entwickelt. Verwendet wurde die IDE Intellij IDEA Community 2019.3.2.
- Zur Erstellung von Textdateien wurde der Texteditor Notepad++ benutzt.
- Für die Klassen- und Sequenzdiagramm wurde das Programm Visual Pardigm 16 verwendet.
- Zur Erstellung von Nassi-Schneiderman-Diagrammen wurde Java-Editor18.23 verwendet.
- Zur Erstellung dieser Dokumentation wurde eine Online-Latex-Editor Overleaf verwendet.

Kapitel 5

Benutzeranleitung

5.1 Verzeichnisstuktur

- README.md ist die Readme-Datei für das Projekt
- `src` : In diesem Verzeichnis befinden sich alle Testklassen, sowie der Quellcode des Programms
- `out`: In diesem Verzeichnis befinden sich alle kompilierten Java Dateien
- `doc` : In diesem Verzeichnis befindet sich die Programm-Dokumentation
  - `javadoc`: In diesem Verzeichnis liegt die generierte Entwicklerdokumentation
  - `documentation`: In diesem Verzeichnis liegen die Latex-Dateien zur Erzeugung dieser Dokumentation
  - `Dokumentation.pdf`: Dieses Dokument
- `tests`: In diesem Verzeichnis befinden sich Beispiele für die Eingabe-, Ausgabe und Fehlerdateien
  - `ihk` für Beispiele aus der Aufgabenstellung
  - `itests` für selbst ausgesuchte Beispiele

5.2 Systemanforderungen

Die Software ist unter Java JRE 11 (Java Runtime Environment Version 11) ausführbar. Dafür muss die genannte Java-Version oder höher auf der Maschine installiert

sein. Außerdem sollte beachtet werden, dass Java nicht abwährtskompatibel ist und entsprechend keine ältere Version verwendet werden sollte.
Das Programm wurde wie im Kapitel 4 beschrieben entwickelt.
Die Ausführbarkeit wird unter dem genannten Voraussetzungen garantiert.
Zudem kann das Programm unter Linux/Unix mit oben gennanten Vorausetzungen (Java-Version) ausgeführt werden.

5.3 Benutzung der Kommandozeile unter Windows

Mit Hilfe der Kommandozeile (Eingabeaufforderung unter Windows bzw. Shell unter Linux/Unix) müssen Sie in das Verzeichnis `~\out\artifacts\makeUpper_jar\`. Das Programm wird wie folgt gestartet:

Beispielaufruf: `java -jar T9.jar` falls das Programm in Kommandozeile-Modus aufgerufen werden sollte

Beispielaufruf: `java -jar T9.jar -i testfall.txt` falls das Programm mit einer Eingabedatei testfall.txt ausgeführt werden sollte.

5.4 Die Ausführung der Testfälle

Je nach Betriebssystem liegt ein Bash- oder Shell-Script zur Verfügung, um die Testfälle im Verzeichnis *tests* automatisiert auszuführen.

Die Dateien (run_tests.bat bzw. run_test.sh) führen die oben genannten Befehle automatisiert mit allen Testfällen im Verzeichnis `tests` aus. Dabei muss die Pfadeingabe zur .jar-Datei und der zur Testdatei(en) korrekt sein. Für nähere Informationen zu Testfällen siehe Kapitel 6

Kapitel 6

Testfälle und Diskussion

6.1 Konvention

Die Testfälle sind unterteilt in IHK-Beispliele und selbst erstellte Tests. Die Testdateien haben folgende Benennung: Die Bezeichnung [0-9] steht für eine Zahl aus dem Bereich von 0 bis 9

- E[0-9][0-9][0-9].txt für **Eingabedateien**
- E[0-9][0-9][0-9].output.txt für **Ausgabedateien**
- E[0-9][0-9][0-9].error.txt für **Fehlerdateien**

Es wird grundsätzlich immer zu jeder Eingabedatei eine Ausgabe- und Fehlerdatei erzeugt. Hat das Programm keine Fehler bzw. keine Ausgabe erzeugt, sind die Dateien **leer**. Beispielsweise erzeugt die Eingabedate E001.txt nach meiner Konvention die Ausgabedatei E001.output.txt und die Fehlerdatei E001.error.txt.

6.2 IHK-Beispiele

Nachfolgend sind die Beispieler von der IHK-Aufgabenstellung mit aufgelistet. Aus den Kommentaren in der Eingabedateien ist eine genauere Beschreibung zu sehen.

- **Testfall1**: Kein Randwert und kein Sonderfall. Eingabe:

```
% Test mit kleinem a
a
```

Ausgabe:

```
A
```

Fehler: Leer

- **Testfall2**: einzelne Großbuchstabe Eingabe:

```
b
```

Ausgabe:

```
B
```

Fehler: Leer

- **Testfall 11**: einzelne Sonderzeichen Eingabe:

```
_
```

Ausgabe:

```
_
```

Fehler:

```
2020/05/14 11:19:29: Error at line 1 : contains special character
```

? ?

6.3 Eigene Beispiele

6.3.1 Äquivalenzklassen

Nachfolgend sind Testbeispiele aufgelistet, die das Programm weitestgehend funktional testen. Dabei werden die Äquivalenzklassen behandelt, die ich mir in der Konzeptionsphase ausgedacht habe. In Tabelle 6.1 sind die Äquivalenzklassen in zulässige und unzulässige Mengen unterteilt. Liegt ein Representant in der unzulässigen Menge, sollte das Programm einen entsprechenden Fehler ausgeben.

Name	Zulässige Bereich	Unzulässige Bereich	Repräsentant
Buchstaben	{A-Z,a-z,0-9}	Sonderzeichen	Ä
Sätze			

Tabelle 6.1: Tabelle mit den Äquivalenzklassen

6.3.2 Beispiele

- **Testfall1**: Kein Randwert und kein Sonderfall. Eingabe:

```
% Test mit kleinem a
a
```

Ausgabe:

```
A
```

Fehler: Leer

- **Testfall2**: einzelne Großbuchstabe Eingabe:

```
b
```

Ausgabe:

```
B
```

Fehler: Leer

- **Testfall 11**: einzelne Sonderzeichen Eingabe:

```
_
```

Ausgabe:

```
1 _
```

Fehler:

```
1 2020/05/14 11:19:29: Error at line 1 : contains special character
```

Kapitel 7

Abweichungen des Prüfungsproduktes vom Konzept

Bei der Umsetzung des Konzeptes in das Programmsystem wurden einige kleine Änderungen vorgenommen. Folgende Änderungen am Konzept sind anzuführen:

- Die Baumstruktur in der Klasse `Baum` speichert die Wörter mit ihrem expliziten Ziffernfolge. Dies liegt daran, dass die Reihenfolge der Buchstaben in einem Knoten nicht ausreicht, um Wörter eineindeutig zuzuordnen.
- Die Klassenstruktur hat noch zusätzliche Hilfsfunktionen wie `nutzerinteraktion()` in **Konsole** oder `makeWord()` in **Woerterbuch**
- Die Nassi-Schneidermann-Diagrammen ersetzen den im Prüfungsproduktes aufgeführten Pseudocode.

Kapitel 8

Zusammenfassung und Ausblick

Es kann zahlreiche Erweiterungen zu dieser Software geben. Möglichkeiten wären beispielsweise, eine erweiterte Interpretation der Tasten, um Zahlen darzustellen, Darstellung von Klein und Großschreibung.

Für die Eingabe sowie die Ausgabe könnte eine graphische Oberfläche implementiert werden. Dadurch könnte der Anwender die Ausgabe graphisch besser sehen.

Das Programm speichert die Eingaben von Nutzer im Wörterbuch nur bei einer expliziten Eingabe von Zifferncode. Dies kann einfacher gestaltet werden, indem die Wortvorschläge nicht nur aus dem Wörterbuch vorgeschlagen werden, sondern aus dem Tippverhalten des Nutzers. Ein Wörterbuch bestehend aus Konstellationen von den Häufigsten benutzen Buchstaben können vorgeschlagen werden.

Anhang A

Quellcode

A.1 Main

```
1  public class Main {
2      public static void main(String[] args) {
3          // create IO class for input and output
4          IO io = new IO();
5          if (args.length == 0) {
6              // lunch program in normal mode (console mode)
7              io.startConsole();
8          } else if (args.length == 2) {
9              // lunch in testmode
10             for (String arg : args) {
11                 // first argument must be a parameter "-i" referencing input
12                 if (arg.equals("-i")) {
13                     // go to next argument
14                     continue;
15                 } else if (arg.charAt(0) == '-') {
16                     System.out.println("Input Error: false parameter (" + arg +
       ")");
17                     return;
18                 }
19                 // second argument is input file with correct extension
20                 String[] inputFile = arg.split("\\.");
21                 if (inputFile.length < 2) {
22                     System.out.println("Input Error: input file name is too
       short");
23                     return;
24                 }
25                 // check file extension
26                 String extension = inputFile[inputFile.length - 1];
27                 if (!extension.equals(io.extension)) {
28                     System.out.println("Input Error: wrong input file extension
       ");
29                     return;
30                 }
```

```
31                  // initiate files for input, output and errors
32                  io.initFiles(arg);
33
34                  // start reading the file
35                  boolean ok = io.readInputFile();
36                  if(!ok){
37                      // abort process
38                      return;
39                  }
40
41                  io.writeOutputFile();
42                  ok = io.commitErrorLog();
43                  if(!ok){
44                      System.out.println("Error log could not be committed");
45                  }
46
47
48              }
49          } else {
50              System.out.println("Input Error: false number of arguments");
51              return;
52          }
53      }
54  }
```

Listing A.1: Klasse `Main`

A.2 Input Output

```
1  import java.io.*;
2  import java.nio.ByteBuffer;
3  import java.nio.channels.FileChannel;
4  import java.nio.channels.FileLock;
5  import java.nio.charset.StandardCharsets;
6  import java.nio.file.Files;
7  import java.nio.file.Paths;
8  import java.time.LocalDateTime;
9  import java.time.format.DateTimeFormatter;
10 import java.util.ArrayList;
11 import java.util.Scanner;
12
13 public class IO implements InputOutput {
14     public Scanner sc = new Scanner(System.in); // for user input via console
15     // restrictions
16     public String extension = "txt";
17     public String lineCommentChar = "%";
18     public String blockCommentCharBegin = "%*";
19     public String blockCommentCharEnd = "*%";
20     public int maxInputFileLength = 1000;
21     public String mainDelimiter = ",";  // delimiter for values
22     public String secondDelimiter = "\t";  // delimiter for value pares
23
24
25     // files options
26     public String inputFilePath;
```

```
27      public String outputFilePath;
28      public String errorFilePath;
29      // files
30      private File inputFile;
31
32      // saving errors
33      public String errorLog = "";
34
35      // problem specific values
36      public String input = "";
37
38      // console options
39      private ArrayList<String> menuOptions = new ArrayList<String>() {
40          {
41              add("0");
42          }
43      };
44      // program name to use in console
45      private String programName = "Make Upper v0.1";
46
47
48      /**
49       * Reads input file according to restrictions
50       */
51      public boolean readInputFile() {
52          try {
53              if (!this.inputFile.exists()) {
54                  submitError("Input File don't exist");
55                  return false;
56              } else if (this.inputFile.length() == 0) {
57                  submitError("Input File is empty");
58                  return false;
59              }
60              BufferedReader reader = new BufferedReader(new FileReader(this.
    inputFile));
61              StringBuilder fileContents = new StringBuilder();
62              char[] buffer = new char[4096];
63              while (reader.read(buffer, 0, 4096) > 0) {
64                  fileContents.append(buffer);
65              }
66              String input = "";
67              input = new String(Files.readAllBytes(Paths.get(this.inputFilePath)
    ));
68              input = stripBlockComments(this.blockCommentCharBegin, this.
    blockCommentCharEnd, input);
69              // strip line comment
70              input = stripLineComments(this.lineCommentChar, input);
71
72              String[] inputLines = input.split(System.getProperty("line.
    separator"));
73              for (int i = 0; i < inputLines.length; i++) {
74                  String line = inputLines[i];
75                  // if line is empty
76                  if (line.length() == 0) {
77                      continue;
78                  }
```

```
79                  // save text
80                  Upper up = new Upper(this);
81                  this.input += up.toUpper(line, i + 1);
82              }
83
84          } catch (Exception e) {
85              System.out.println(e.getMessage());
86              submitError(e.getMessage());
87              return false;
88          }
89          return true;
90      }
91
92      private String stripLineComments(String token, String input) {
93          StringBuilder output = new StringBuilder();
94          String[] inputLines = input.split("\n");
95          for (int i = 0; i < inputLines.length; i++) {
96              String line = inputLines[i];
97
98              line = line.split(this.lineCommentChar)[0];
99              if (line.length() != 0) {
100                 output.append(line).append("\n");
101             }
102         }
103         return output.toString();
104     }
105
106     private String stripBlockComments(String beginToken, String endToken,
    String input) {
107         StringBuilder output = new StringBuilder();
108         while (true) {
109             // search one occurrence at a time
110             int begin = input.indexOf(beginToken);
111             int end = input.indexOf(endToken, begin + beginToken.length());
112             if (begin == -1 || end == -1) {  // if no occurrences where found
113                 output.append(input);
114                 return output.toString();
115             }
116             output.append(input.substring(0, begin));
117             input = input.substring(end + endToken.length());
118         }
119     }
120
121     /**
122      * Write ouput file according to predefined rules
123      */
124     public void writeOutputFile() {
125         try {
126             File fout = new File(this.outputFilePath);
127             FileOutputStream fos = new FileOutputStream(fout);
128             BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
    fos));
129
130             writer.write(this.input);
131             writer.close();
132         } catch (IOException e) {
```

```
133             submitError(e.getMessage());
134         }
135     }
136 
137     /**
138      * Initiates the input, output and error file path variables according to
    given input file name
139      *
140      * @param inputFilePath: String
141      */
142     public void initFiles(String inputFilePath) {
143         this.inputFilePath = inputFilePath;
144         this.inputFile = new File(this.inputFilePath);
145         // get error file name
146         String name = inputFile.getName().split("\\.")[0];
147         if (this.inputFilePath.contains(File.separator)) {
148             this.errorFilePath = this.inputFilePath.substring(0, this.
    inputFilePath.lastIndexOf(File.separator) + 1) + name
149                     + ".error." + this.extension;
150         } else {
151             // if same directory
152             this.errorFilePath += ".error." + this.extension;
153         }
154         // get output file name
155         name = inputFile.getName().split("\\.")[0];
156         if (this.inputFilePath.contains(File.separator)) {
157             this.outputFilePath = this.inputFilePath.substring(0, this.
    inputFilePath.lastIndexOf(File.separator) + 1) + name
158                     + ".output." + this.extension;
159         } else {
160             this.outputFilePath += ".output." + this.extension;
161         }
162 
163 
164     }
165 
166     /**
167      * appends the error message to the overall error log of the programm with
    a time stamp at the beginning
168      *
169      * @param error: String
170      */
171     public void submitError(String error) {
172         DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:
    ss");
173         LocalDateTime now = LocalDateTime.now();
174         this.errorLog += dtf.format(now) + ": " + error + "\n";
175     }
176 
177     /**
178      * Write the error log to a file with error log file path.
179      *
180      * @return ture: if successful/false: if unsuccessful
181      */
182     public boolean commitErrorLog() {
183         try {
```

```
184             // Files.write(Paths.get(this.errorFilePath), this.errorLog.
    getBytes());
185             RandomAccessFile stream = new RandomAccessFile(this.errorFilePath,
    "rw");
186             FileChannel channel = stream.getChannel();
187             FileLock lock = null;
188 
189             String value = this.errorLog;
190             byte[] strBytes = value.getBytes();
191             ByteBuffer buffer = ByteBuffer.allocate(strBytes.length);
192             buffer.put(strBytes);
193             buffer.flip();
194             channel.write(buffer);
195             stream.close();
196             channel.close();
197 
198         } catch (IOException e) {
199             e.printStackTrace();
200             return false;
201         }
202         return true;
203     }
204 
205     /**
206      * Starts the console mode of the programm
207      */
208     public void startConsole() {
209         System.out.println("Program" + this.programName + " started...");
210         while (true) {
211             printMenu();
212             boolean exit = scanMenu();
213             if (exit) {
214                 System.out.println("Program " + this.programName + " exited");
215                 break;
216             }
217         }
218     }
219 
220     /**
221      * Print the main menu on the console
222      */
223     public void printMenu() {
224         System.out.println("Main Menu:");
225         System.out.println("(0) exit");
226     }
227 
228     /**
229      * Get user input option for the main menu
230      *
231      * @return true: if program should exit/false: if repeat console input
    procedure
232      */
233     public boolean scanMenu() {
234         String key = sc.nextLine();
235         int option;
236         if (!this.menuOptions.contains(key)) {
```

```
237             option = -1;
238         } else {
239             option = Integer.parseInt(key);
240         }
241         switch (option) {
242             case 0:
243                 // exit program
244                 return true;
245             default:
246                 System.out.println("Wrong input!");
247                 return false;
248         }
249     }
250
251 }
```

Listing A.2: Klasse `Input Output`

A.3 Upper

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class Upper implements Algorithm {
5      private IO io;
6      private String specialCharacters=" !#$%&'()*+,-./:;<=>?@[]^_`{|}";
7      public Upper(IO io) {
8          this.io = io;
9      }
10
11     public String toUpper(String input, int line) {
12         for (int i = 0; i < input.length(); i++) {
13             if (this.specialCharacters.contains(Character.toString(input.charAt
       (i))))
14             {
15                 io.submitError("Error at line " + line + " : contains special
       character");
16                 break;
17             }
18         }
19         return input.toUpperCase();
20     }
21 }
```

Listing A.3: Klasse `Upper`

A.4 Algorithmus Schnittstelle

```
1  public interface Algorithm {
2      String toUpper(String input, int line);
3  }
```

Listing A.4: Klasse `Baum`

A.5 Knoten

```
1 public interface InputOutput {
2     boolean readInputFile();
3     void writeOutputFile();
4     void submitError(String error);
5     boolean commitErrorLog();
6     void startConsole();
7 }
```

Listing A.5: Klasse `Node`