Dokumentation der Praktischen Arbeit zur Prüfung zum Mathematisch-technischen Softwareentwickler

13. Mai 2016

Felix Kibellus

Prüfungs-Nummer:	101-20547
Programmiersprache:	Java
Ausbildungsort:	Institut für Kernphysik Forschungszentrum Jülich

Inhaltsverzeichnis

1. Aufgabenanalyse

1.1. Problembeschreibung

Die Aufgabe besteht darin, die Knobelfreunde AG mit einer geeigneten Software bei der Erstellung von Rätseln zu unterstützen. Diese Software bekommt einige Wörter als Eingabe und soll diese in einem $n \times m$-Rechteckfeld verteilen (Ähnlich wie beim Kreuzworträtsel). Dabei dürfen keine Wörter völlig allein stehen, sondern müssen sich mit anderen Wörtern überschneiden. Dies ist natürlich nur dann möglich, wenn an der Stelle der Überschneidung der gleiche Buchstabe steht. Die nicht durch Wörter der Eingabe belegten Felder des $n \times m$-Rasters sollen dann durch zufällige Buchstaben aufgefüllt werden, sodass ein Rätsel entsteht, in dem die angegebenen Wörter zu suchen sind. Ziel ist die Entwicklung eines Softwaresystems, welches die Wörter so verteilt, dass $n \cdot m$ minimal ist. (Also der Flächeninhalt des $n \times m$-Buchstabenrasters)

1.2. Modularisierung

Das Vorliegende Gesamtproblem lässt sich in verschiedene Teilprobleme zerlegen.

1. Lesen der Eingabedaten aus der Datei
2. Prüfen der Eingabedaten auf syntaktische/semantische Korrektheit
3. Erstellung des Rätsels (minimaler Flächeninhalt)
4. Auffüllen mit zufälligen Buchstaben
5. Ausgabe in eine Datei

1.3. Eingabe

Die Eingabedatei muss das folgende Format haben:
$;< Kommentarzeile >$
$;< Kommentarzeile >$
$\vdots$
$< Wort1 >$
$< Wort2 >$
$\vdots$

EOF
Es dürfen 0 bis n Kommentarzeilen und 2 bis m Wörter existieren.
Syntax: Kommentarzeilen müssen immer mit Semikolon eingeleitet werden(erstes Zeichen) und enden auf \ n. Die Wörter enthalten ausschließlich die Zeichen [A-Z] in Groß- und Kleinschreibung. Umlaute sind in AE,OE,etc. umzuändern. Es dürfen sich keine Leerzeichen zwischen den Buchstaben befinden. Leerzeilen vor oder nach den Wörtern sind erlaubt und werden vom Programm selbstständig entfernt. Es darf nur ein Wort pro Zeile stehen. Eine Zeile endet mit \n.
Beispiel:

```
;Dies ist ein Kommentar
Auto
Oma
```

Dies wäre eine gültige Eingabe.
Die Eingabedatei sollte die Dateiendung ".in "haben. Es muss eine Eingabedatei angegeben werden.

1.4. Ausgabe

Die Ausgabedatei soll das folgende Format haben:

Kommentare aus Eingabedatei
Eingelesene Wörter :
$< Wort1 >, < Wort2 >, \ldots$
LEERZEILE
$**$*Ratsel nicht versteckt*
LEERZEILE

```
Lösung
nicht
versteckt
```

LEERZEILE
$**$*Rätsel versteckt*

```
Lösung
versteckt
```

LEERZEILE
Kompaktheitsmaß :$< Maß >$

Die Lösungen bestehen ausschließlich aus Großbuchstaben [A-Z]. Leerzeichen sind nur für das nicht versteckte Rätsel zugelassen. Die Angabe einer Ausgabedatei ist optional. Wenn die Eingabedatei auf .in endet, soll die Ausgabedatei dem Dateinamen der Eingabedatei entsprechen und .in durch .out ersetzt werden. Andernfalls soll lediglich .out angehängt werden.

1.5. Spezifikationen

- Wenn die Eingabedatei falsch formatiert ist oder nicht existiert wird das Programm mit passender Fehlermeldung abgebrochen.
- Wenn der Pfad zur Ausgabedatei nicht existiert oder nicht beschreibbar ist wird auch so verfahren.
- In der Eingabedatei müssen mindestens zwei Wörter stehen, sonst wird das Programm abgebrochen.
- Im Rätsel dürfen die Wörter nur von oben nach unten bzw. von links nach rechts lesbar sein. Diagonale Wörter oder solche, die von unten nach oben bzw. rechts nach links zu lesen sind, sind nicht zulässig.
- Sollte ein Wort in der Eingabedatei falsch sein, so ist die Zeilennummer auszugeben.
- Wenn ein Wort nicht mit den anderen zu verbinden ist, bricht das Programm mit einer Fehlermeldung ab.

1.6. Programmaufruf

Das Programm ist über die Konsole mit

java -jar raetselLoeser [-o OUT | −− output OUT][-l | −− log LEVEL] [-h | −− help] INFILE

aufzurufen. OUT ist der Dateiname der Ausgabedatei, INFILE der Name der Eingabedatei. Es soll möglich sein, Logging mit einen steuerbaren Level zu aktivieren. Es soll eine Hilfe angezeigt werden (-h). Wie in Linux üblich, soll das Programm nur im Fehlerfall eine Ausgabe nach Standard-Error liefern.

2. Verfahrensbeschreibung

2.1. Logische Datenstrukturen

Die Datenstruktur für die Eingabedaten muss folgendes leisten:

1. Speicherung der Kommentarzeilen zur späteren Ausgabe in die Ausgabedatei
2. Speicherung der zu verwendenden Wörter

Sinnvoll ist also eine Dateistruktur, welche zwei Listen aus Zeichenketten beinhaltet. Die verwendeten Wörter sollen dabei schon in Großbuchstaben vorliegen. Für das Beispiel der Eingabedatei ergibt sich also:

Liste1 = { “;Dies ist ein Kommentar “}
Liste2 = { “AUTO “, “OMA “}

Zur Speicherung von möglichen Lösungen und von Zwischenergebnissen bei der Erstellung der Lösungen benötigt man ein $n \times m$ Zeichenfeld, in dem die bereits eingetragenen Wörter stehen, sowie eine Liste mit den noch nicht eingetragenen Wörtern. Die Laufzeit wird später maßgeblich davon abhängen, wie schnell das Programm bestimmte Buchstaben im Raster finden kann. Deshalb wird zusätzlich eine Map: Buchstabe $\rightarrow$ Menge von (x,y)-Punkten benötigt.
Beispiel:

Noch nicht verwendet: { “AUTO “}
Ratser : OMA
Map : { 0 $\rightarrow$ {(0,0)}, M $\rightarrow$ {(0,1)}, A $\rightarrow$ { (0,2) }

Soll nun AUTO eingefügt werden, muss die Map nur nach den Buchstaben A,U,T und O befragt werden, um mögliche Ansatzkoordinaten zu erhalten. Das Raster muss sich während der Laufzeit vergrößern können. Nach einfügen von Auto würde es beispielsweise so aussehen:

O	*M*	*A*
		U
		T
		O

Dies führt jedoch zu Problemen bei der Benutzung der Map. Wenn das Raster nach links oder nach oben hin erweitert werden muss, ändern sich die Koordinaten der Buchstaben. Also müssen in diesem Fall alle Einträge der Map so verändert werden, dass sie dem Koordinatensystem des neuen Rasters entsprechen. Eine alternative Idee wäre hier das Speichern eines Offsets, welches jedes mal auf die Punkte in der Map addiert wird,

wenn diese abgefragt werden. Es wurde jedoch entschieden, diese Alternative nicht weiter zu verfolgen, da die Erweiterung des Rasters viel seltener als die Abfrage der Punkte ausgeführt wird. Es wurde festgestellt, dass sich die Verwendung der Map wegen der ständigen Aktualisierung nicht positiv auf das Laufzeitverhalten auswirkt. Deshalb wurde die Verwendung als optionale Einstellung deklariert, sodass der Algorithmus sowohl mit, als auch ohne Map verwendet werden kann.

Die Datenstruktur der Ausgabedaten muss folgende Informationen enthalten:

1. Informationen aus der Eingabedatei

2. Rasterdarstellung (optional) ohne zusätzliche Buchstaben

3. Rasterdarstellung versteckt

Es macht also Sinn, die Datenstruktur für die Ausgabedaten so zu wählen, dass sie einen Verweis auf die Eingabedaten und zwei $n \times m$-Zeichenfelder enthält.

2.2. Verfahren

Das Eigentliche Verfahren soll mit einem Suchbaum durchgeführt werden. Dazu sollen die verfügbaren Wörter in jeder möglichen Darstellung kombiniert werden, um dann eine optimale Lösung zu erhalten. Die Idee des Kernalgorithmus ist also folgende:

1. erstelle die Startkonfiguration mit einem beliebigen Wort und speichere es in einer Liste

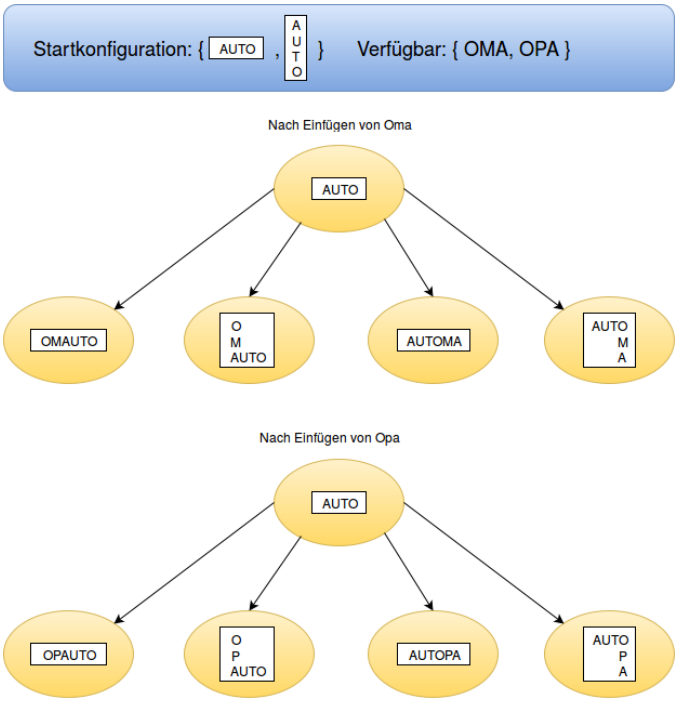
2. Solange die Liste nicht leer ist wird folgendes ausgeführt:

    a) hole Konfiguration aus der Liste

    b) füge jedes noch nicht benutzte Wort in jeder erdenklichen Art ein

    c) speichere die daraus entstehende Konfiguration in der Liste

Beispiel:



In der zweiten Iteration würde nun in jede Konfiguration wo OPA noch verfügbar ist OPA eingefügt und in jede Konfiguration wo OMA noch verfügbar ist OMA eingefügt. Hierbei können sehr schnell doppelte Einträge entstehen .
Beispiel:
OMA in AUTOPA = OMAUTOPA
OPA in OMAUTO = OMAUTOPA
Es macht natürlich keinen Sinn diese doppelten Einträge mitzuführen. Deshalb wird eine Konfiguration nur dann in die Liste eingefügt, wenn sie noch nicht vorhanden ist. Dazu wird vor dem Einfügen geprüft, ob das Wort noch nicht in der Liste enthalten ist. Wenn sich ein Wort gar nicht kombinieren lässt (z.B. EI in OMA,OPA und AUTO), wird EI nicht eingefügt.
Der Fall wird jedoch im eigentlichen Verfahren nicht explizit betrachtet und ist daran zu erkennen, dass alle Kombinationen versucht worden sind, ohne dass eine Lösung produziert wurde. Wenn eine fertige Lösung gefunden wurde (letztes Wort erfolgreich eingefügt), so wird diese als aktuell beste Lösung gespeichert, wenn eine andere Lösung gefunden wird, wird geprüft, ob diese besser ist als die aktuell beste Lösung (dann wird diese ersetzt) oder schlechter (dann wird sie verworfen). Sollte bereits eine beliebige Lösung existieren und das Verfahren soll eine Konfiguration betrachten, welche bereits vor dem Einfügen schlechter ist wird diese verworfen. Dies ist möglich, da das Qualitätskriterium monoton steigend für wachsende Wortzahl ist. Dies ist darauf zurückzuführen, dass die Ausdehnung des Zeichenfeldes durch das Hinzufügen eines Wortes nur anwachsen oder gleich bleiben kann.

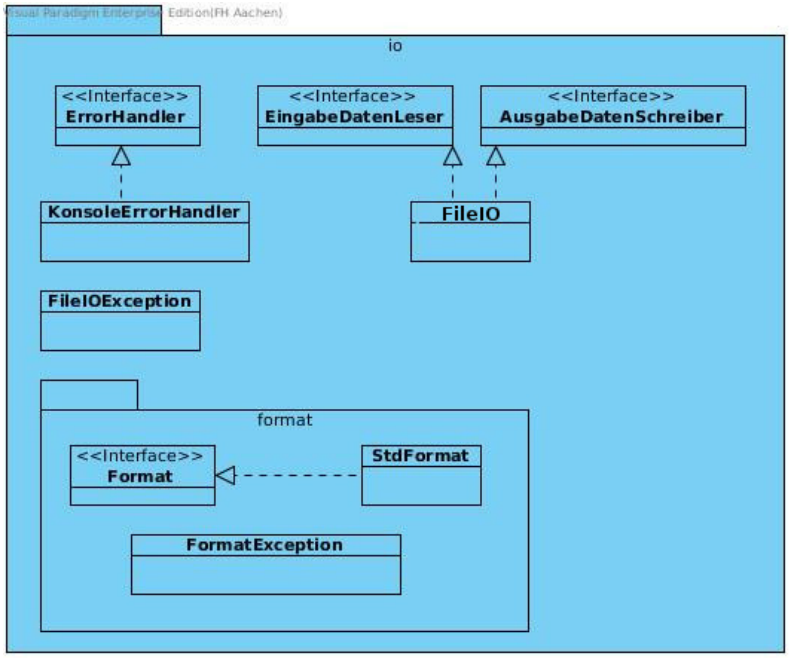
2.3. Einschätzung/Bewertung

Die Verwendung einer Liste implementiert eine Tiefensuche, wenn neue Kombinationen vorne in die Liste eingefügt werden und eine Breitensuche wenn die Kombination hinten an der Liste angefügt werden. Die Tiefensuche liefert früher fertige Ergebnisse. Weitere Zwischenergebnisse/Zweige können dann bereits verworfen werden, wenn ihr Platzbedarf größer als die gefundene Lösung ist. Daher wird die Verwendung der Tiefensuche bevorzugt.

3. Programmbeschreibung

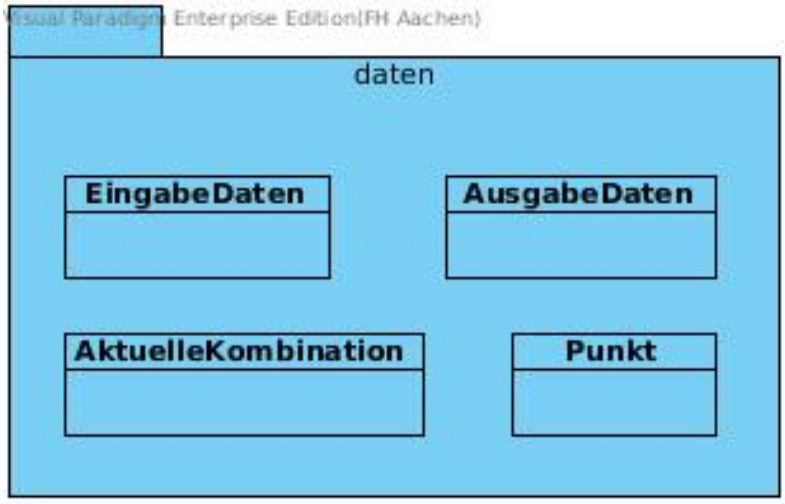
3.1. Architektur

Beim Entwurf dieser Software steht Austauschbarkeit und Erweiterbarkeit der einzelnen Module im Vordergrund. Es soll möglich sein, Ein- und Ausgabeverfahren, Algorithmus, Fehlerbehandlung und Format der Ein/Ausgabedatei einfach auszutauschen. In anbetracht der funktionalen Trennung soll das 3-Schichten Modell verwendet werden. Daten-, Logik- und Präsentationsschicht sollen getrennt voneinander Implementiert werden. Nur durch vorgegebene Schnittstellen darf zwischen den Schichten interagiert werden. Klassen der gleichen Schicht werden zusätzlich in Paketen zusammengefasst.

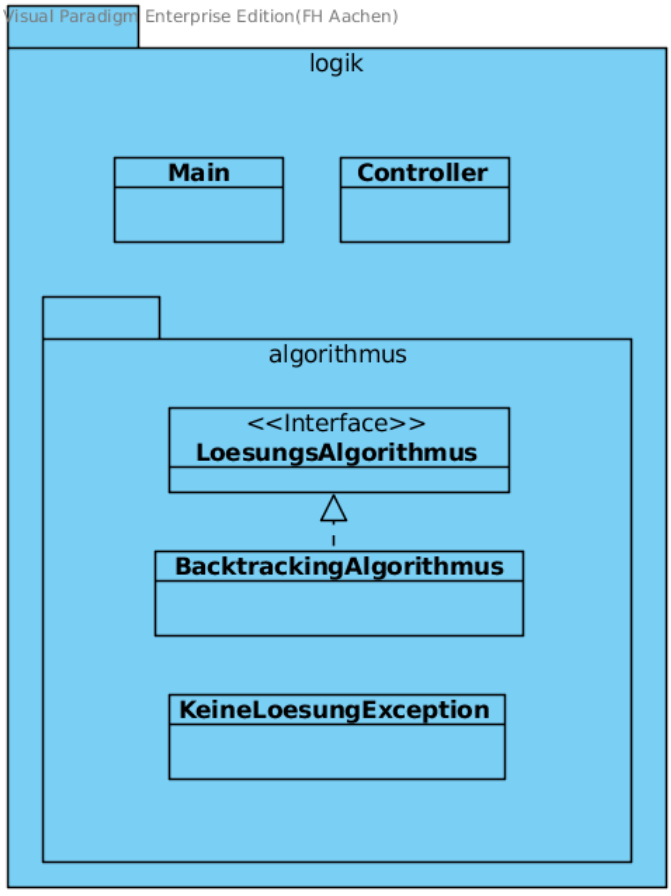


Das Paket **io** beinhaltet alle Klassen der Präsentationsschicht. Die Interfaces EingabeDatenLeser und AusgabeDatenSchreiber dienen zur Abstraktion. Im konkreten Fall soll aus einer Datei gelesen/in eine Datei geschrieben werden. Die Klasse FileIO leiste dies so, wie in Kapitel eins beschrieben, ist jedoch durch die Abstraktion durch Interfaces einfach austauschbar, ohne den restlichen Code verändern zu müssen. Das Softwaresystem arbeitet nur auf dem Interface ErrorHandler, die konkrete Klasse ConsoleErrorHandler implementiert die Aufgabe so, dass die Fehlernachricht auf die Fehlerkonsole geschrieben wird. Darüber hinaus kann durch die Abstraktion des Formatierers das Format der

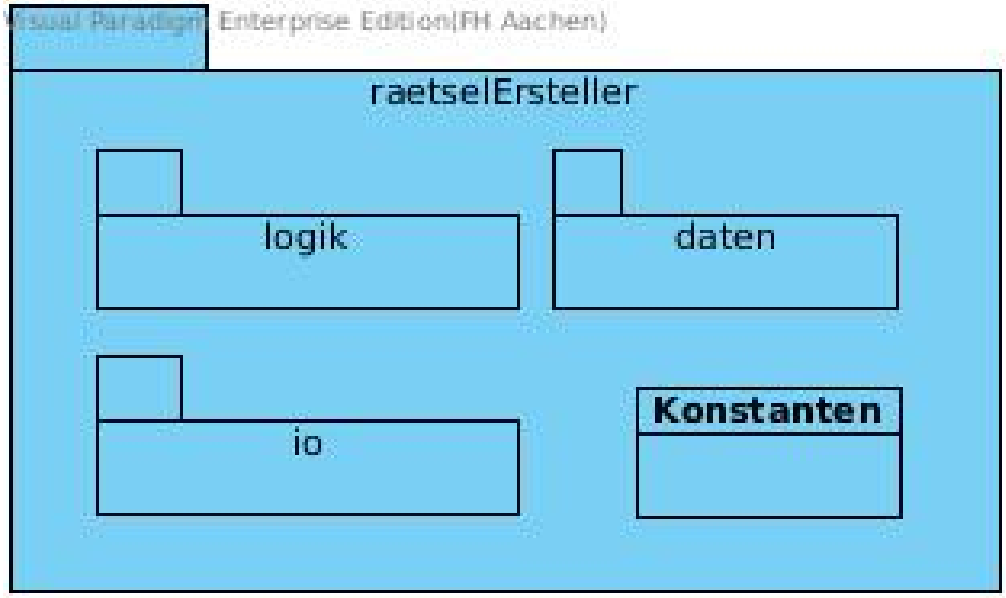
Eingabe-Datei geändert werden ohne die Klasse FileIO zu verändern.



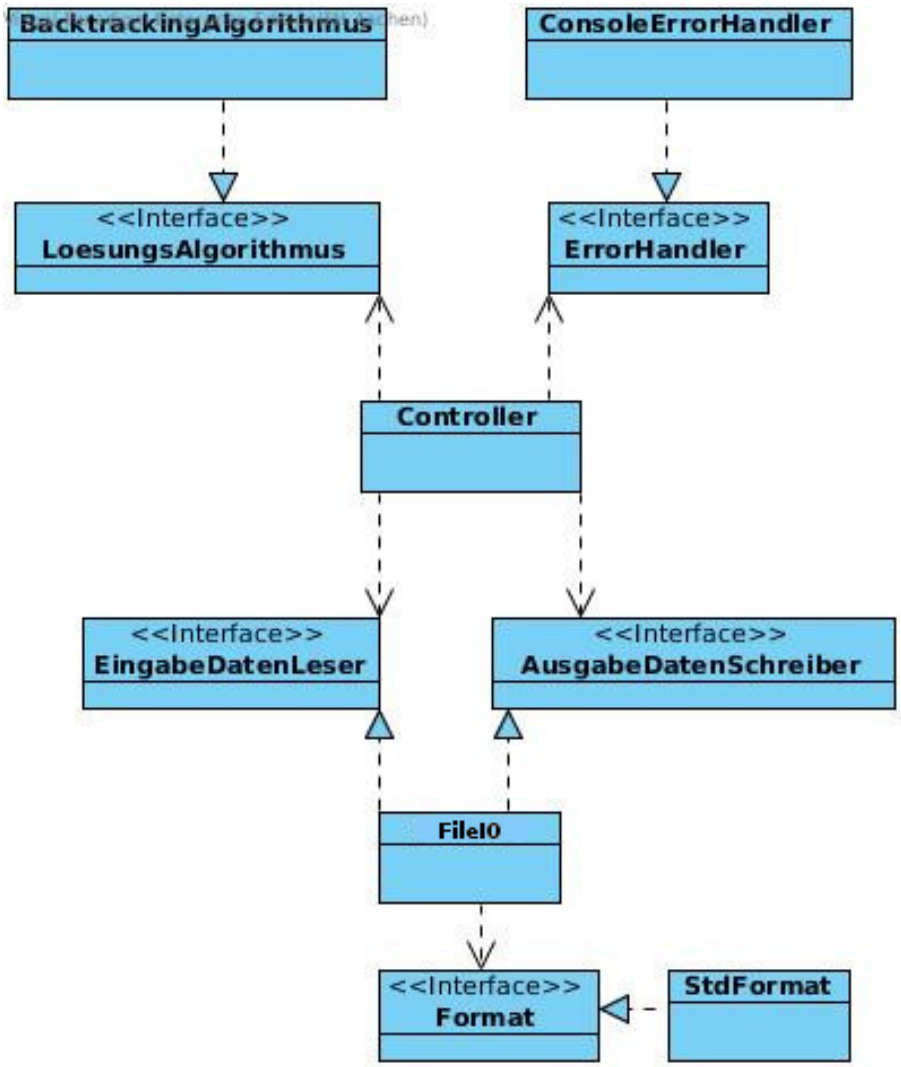
Das Paket **Daten** fasst Datenstrukturen für Eingabe-, Ausgabe- und Laufzeitdaten zusammen. Diese Datenstrukturen sind so wie in Kapitel 2 beschrieben. Zusätzlich wird hier eine Klasse Punkt benutzt, um Koordinaten in der Ebene (x,y) zu beschreiben.



Das Paket **Logik** fasst Klassen der Logikschicht zusammen. Damit der Algorithmus einfach austauschbar ist wird auch hier eine Abstraktion verwendet. Die Klasse Main enthält den Einstiegspunkt des Programms und übernimmt Controlling-Aufgaben.
Die erste Ebene der Quelldateiverzeichnisstruktur sieht so aus:



Alle eben definierten Pakete werden zu einem Pakt raetselErsteller zusammengefasst. Darüber hinaus befindet sich auf dieser Ebene eine Klasse Konstanten. Diese enthält unter anderem alle Ausgabe-Strings, sodass für das Austauschen der Sprache nur diese Datei geändert werden muss. Darüber hinaus enthält die Klasse Konstanten verschiedene Flag-Variablen, sodass nur diese Datei geändert werden muss um das Programmverhalten zu verändern. Beispielsweise existiert in der Klasse eine Variable, über die gesteuert werden kann ob die Map verwendet werden soll oder nicht.

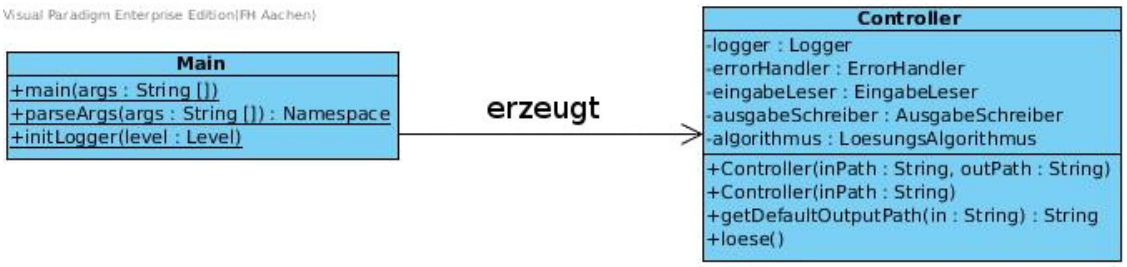


Das obenstehende Diagramm verdeutlicht die Idee der Architektur. Die Klasse Contoller hält Referenzen auf die Interfaces, konkrete Implementierungen sind dem Contoller nicht bekannt.

3.2. Klassen

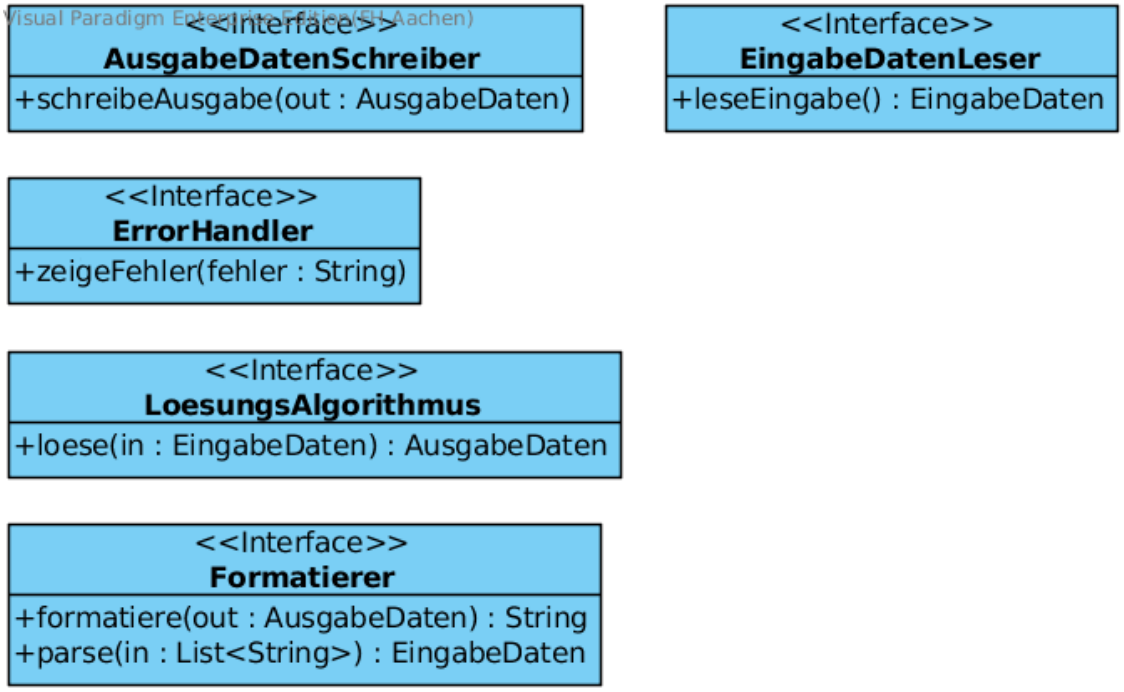
3.2.1. Main

Die Klasse Main definiert den Startpunkt des Programms und erzeugt die Klasse Controller. Der Controller leitet Aufgaben zu den anderen abstrahierten Programmteilen weiter.



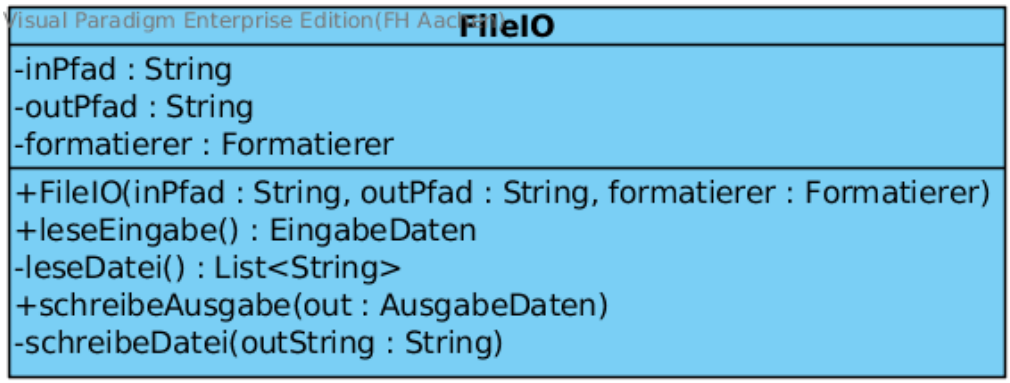
3.2.2. Interfaces

Die folgenden Interfaces dienen zur Abstraktion der konkreten Implementierung von unterschiedlichen Programmteilen.



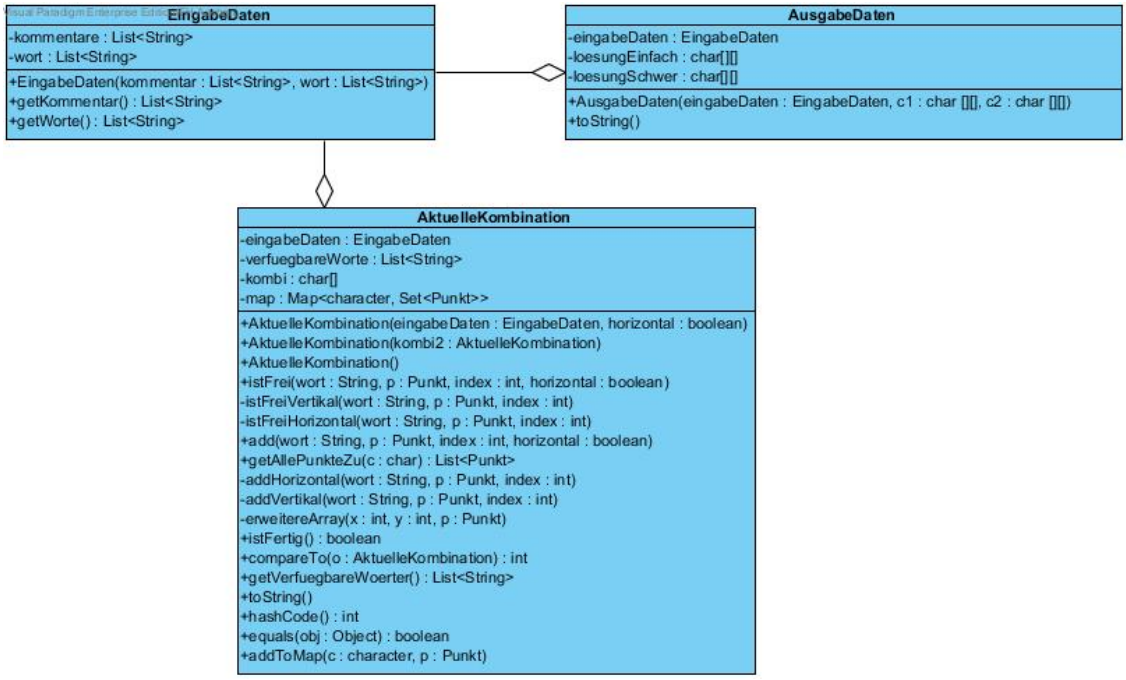
3.2.3. FileIO

Die Klasse FileIO dient zum lesen und schreiben von Dateien.



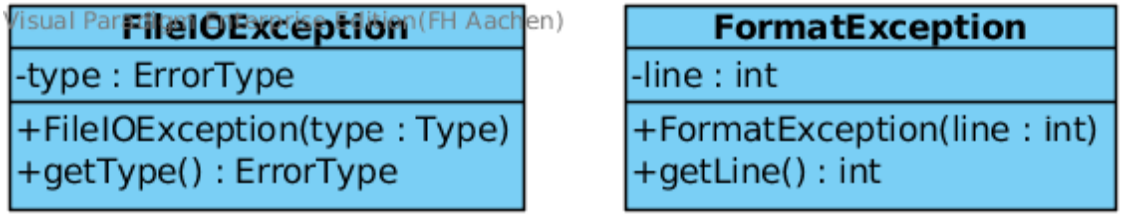
3.2.4. Daten-Klassen

Zur Speicherung von Ein- und Ausgabedaten werden die folgenden Datenstrukturen verwendet.



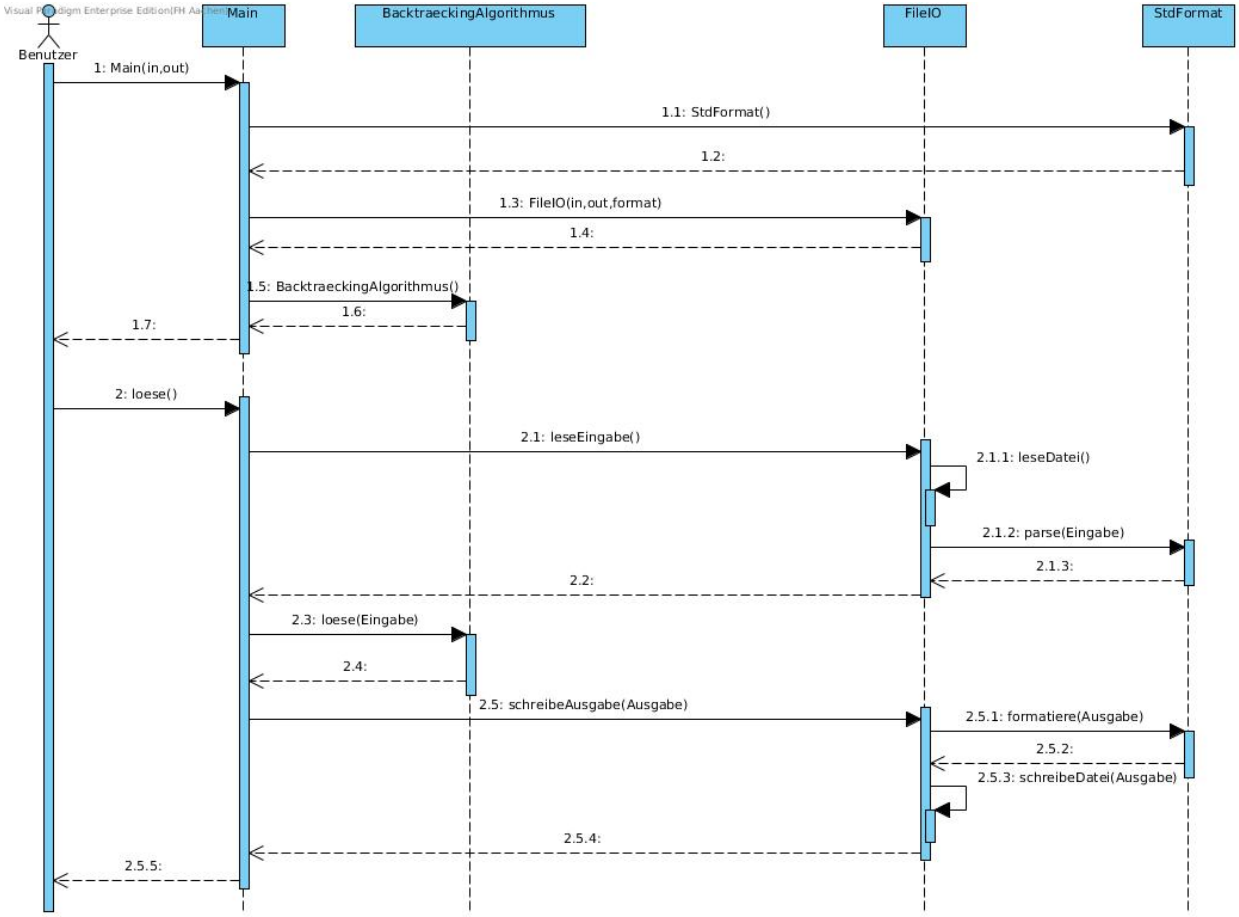
3.2.5. Exceptions

Zur Behandlung von Fehlern dienen die folgenden Klassen.



3.3. Zusammenspiel der Klassen

Der fehlerfreie Fall durch das folgenden Sequenzdiagramm erläutert:

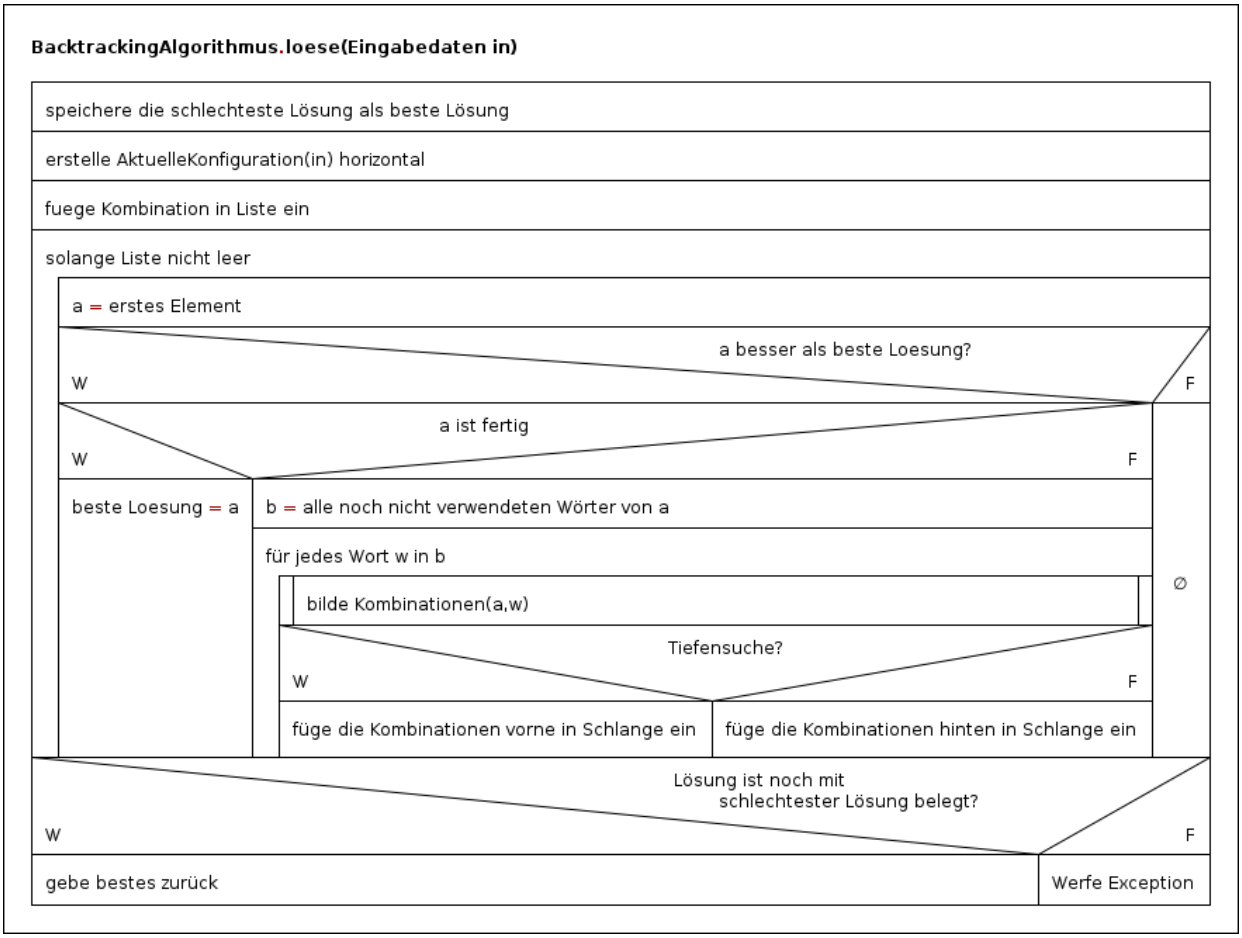


3.4. Beschreibung spezieller Methoden

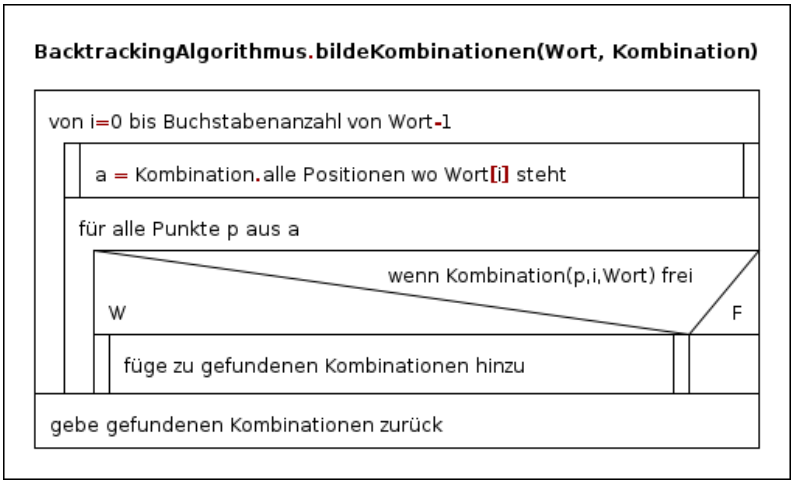
3.4.1. Main.loese()

Loese ließt die Eingabedaten vom EingabeDatenLeser, gibt diese an den LoesungsAlgorithmus weiter und übergibt das erhaltene Objekt der Klasse AusgabeDaten an den AusgabeDatenSchreiber.

3.4.2. Nassi-Schneidermann-Diagramm zu loese

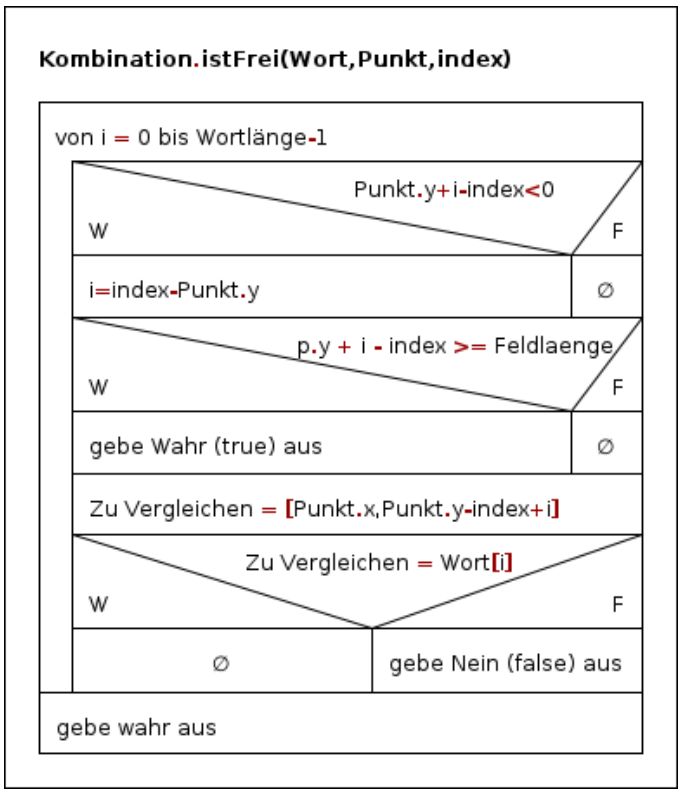


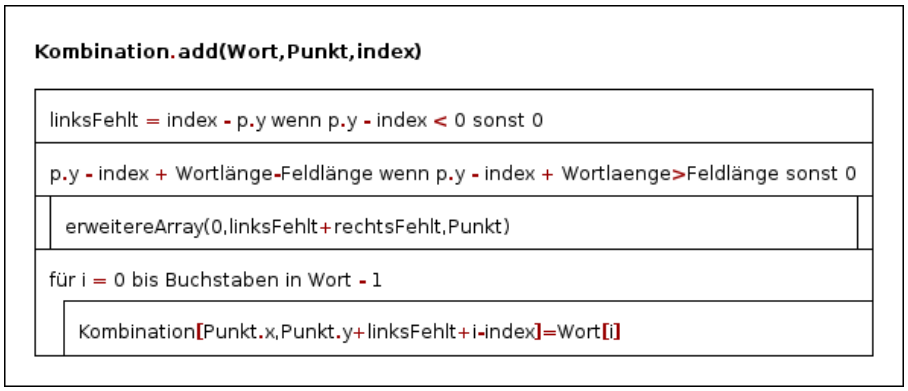
3.4.3. Nassi-Schneidermann-Diagramm zu bildeKombinationen



3.4.4. Kombination.istFrei(Wort, Punkt, index)

Die Variablen punkt und index sind so gewählt, dass Kombination[punkt.x, punkt.y] = Wort[index], also der Buchstabe von Wort an der Stelle Index stimmt mit dem Buchstaben von Kombination an der Stelle Punkt überein. Die folgenden Struktogramme decken nur den Fall horizontal=true ab. Der Fall horizontal=false lässt sich analog mit der x-Koordinate behandeln.





4. Laufzeituntersuchung

In diesem Kapitel soll auf das Laufzeitverhalten des entwickelten Programms eingegangen werden. Dabei wird untersucht, wie stark sich das Laufzeitverhalten ändert, wenn mehr Wörter zur Berechnung verwendet werden.

Das Laufzeitverhalten hängt maßgeblich davon ab, wie viele Kombinationen gebildet werden können. In jeder Iteration des Kernalgorithmus wird die Methode **bildeKombinationen** aufgerufen. Die Anzahl der Kombinationen, die in dieser Methode gebildet werden hängt von verschiedenen Faktoren ab:

1. Anzahl der bisher eingefügten Wörter
2. Anzahl der übereinstimmenden Buchstaben
3. Anordnung der bisher eingefügten Wörter

Wenn also bisher sehr viele Wörter bereits eingefügt worden sind, und viele Buchstaben des einzufügenden Wortes mit den bereits eingefügten Wörtern übereinstimmen liefert **bildeKombinationen** eine große Anzahl neuer Kombinationen. Jede der so gebildeten Kombinationen wird dann in eine Liste eingefügt, und nach dem gleichen Verfahren bearbeitet. Wenn in der ersten Iteration a_1 Kombinationen gebildet worden sind, werden in der nächsten Iteration für jede dieser a_1 Kombinationen wiederum neue Kombinationen erstellt. Dieses Verfahren wird so lange fortgeführt, bis in jeder Kombination alle n verfügbaren Wörter eingefügt worden sind. Da eine weitere Analyse recht schwierig ist, ohne die genaue Anzahl der entstandenen Kombinationen zu kennen, wird für die folgende Analyse $a = E[a_i]$ definiert. Es wird also in der Analyse davon ausgegangen, dass in jeder Iteration genau so viele Kombinationen gebildet werden können, wie im Durchschnitt entstehen. Dies erlaubt natürlich keine genaue Vorhersage des Laufzeitverhaltens. Da jedoch nur ein Interesse an der Kenntnis der Komplexitätsklasse des Algorithmus besteht, ist dies nicht weiter von Belang. Die durchgeführte Vereinfachung erlaubt nun eine einfachere Analyse. In der ersten Iteration werden a Kombinationen erzeugt. Wenn in der nächsten Iteration erneut jede dieser a Kombinationen a neue Kombinationen erzeugt liegen danach $a \cdot a = a^2$ Kombinationen vor. Bei n Wörtern befinden sich also zum Schluss a^n Kombinationen in der Schlange. Daraus geht hervor, dass bei diesem Algorithmus ein asymptotisch exponentielles Laufzeitverhalten vorliegt. Wenn der Algorithmus mit Tiefensuche ausgeführt wird werden schon sehr früh Lösungen gefunden. Diese sind zwar nicht optimal, können aber als Referenzlösung dienen um schlechtere Teillösungen verwerfen zu können. Da bei einer Breitensuche fertige Lösungen erst ganz zum Schluss zur Verfügung stehen ist diese Optimierung nur bei einer Tiefensuche möglich. Deshalb wirkt sich die Verwendung der Tiefensuche positiv auf das Laufzeitverhalten aus.

5. Testdokumentation

Um zu gewährleisten, dass ein robustes Programm entwickelt wurde, welches auch bei unerwarteten oder fehlerhaften Eingaben noch korrekt funktioniert wurden Testfälle entwickelt. Dazu wurden die erarbeiteten Testfälle kategorisiert. Im Unterkapitel IHK-Beispiele finden sich die Normalfälle aus der Aufgabenstellung. Anhand dieser Beispiele wird die Programmfunktionalität genau erklärt. Im Unterkapitel Normalfälle befinden sich Testfälle, an denen genau analysiert wurde, in welcher Konfiguration das Programm ein gutes oder schlechtes Laufzeitverhalten zeigt. Danach werden Sonderfälle betrachtet. Hier wird beispielsweise getestet, wie das Programm auf Eingaben reagiert, die nicht kombiniert werden können. Abschließend werden verschiedene Fehlerfälle betrachtet werden.
Alle Laufzeitmessungen wurden in der empfohlenen Standardkonfiguration, also mit Tiefensuche durchgeführt.
Folgende Testfälle wurden betrachtet:

5.1. IHK-Beispiel

T1: IHK_Testfall1

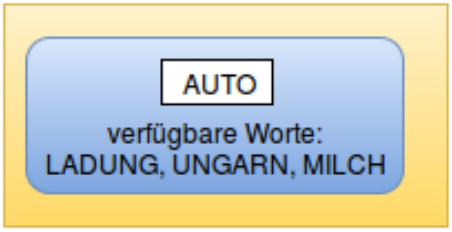
Im folgenden wird das erste IHK-Beispiel ausführlich diskutiert. Die folgenden Erläuterungen können auch am Programm nachvollzogen werden, wenn die Option -l FINEST

angegeben wird. In diesem Fall zeigt das Programm genau an, welche Kombinationen gebildet worden sind, sodass die Funktionalität genau nachvollzogen werden kann. Die Eingabedatei dieses Beispiels sieht wie folgt aus:

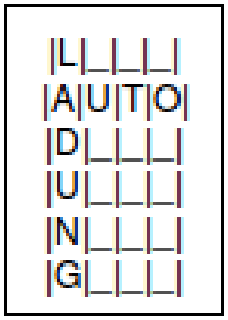
```
;*************************
;       Testfall 1
;*************************
Auto
Ladung
Ungarn
Milch
```

Tests/IHKTestfall1.in

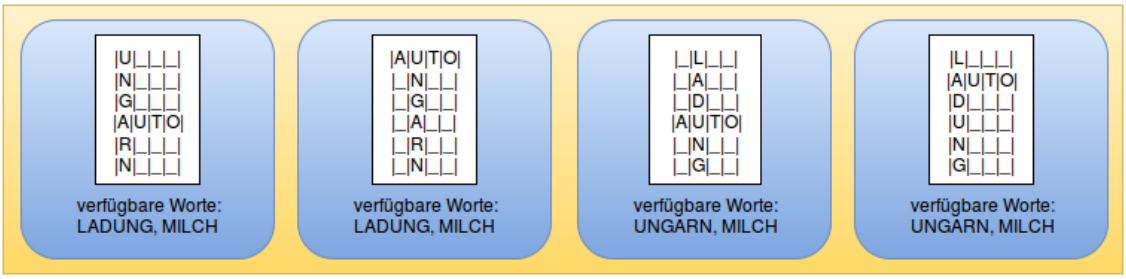
Der implementierte Algorithmus startet immer damit, aus dem ersten Wort der Eingabedaten eine horizontale Kombination zu bilden und diese in die Liste einzufügen. In diesem Fall ist das erste Wort Auto. Die Liste sieht also folgendermaßen aus:



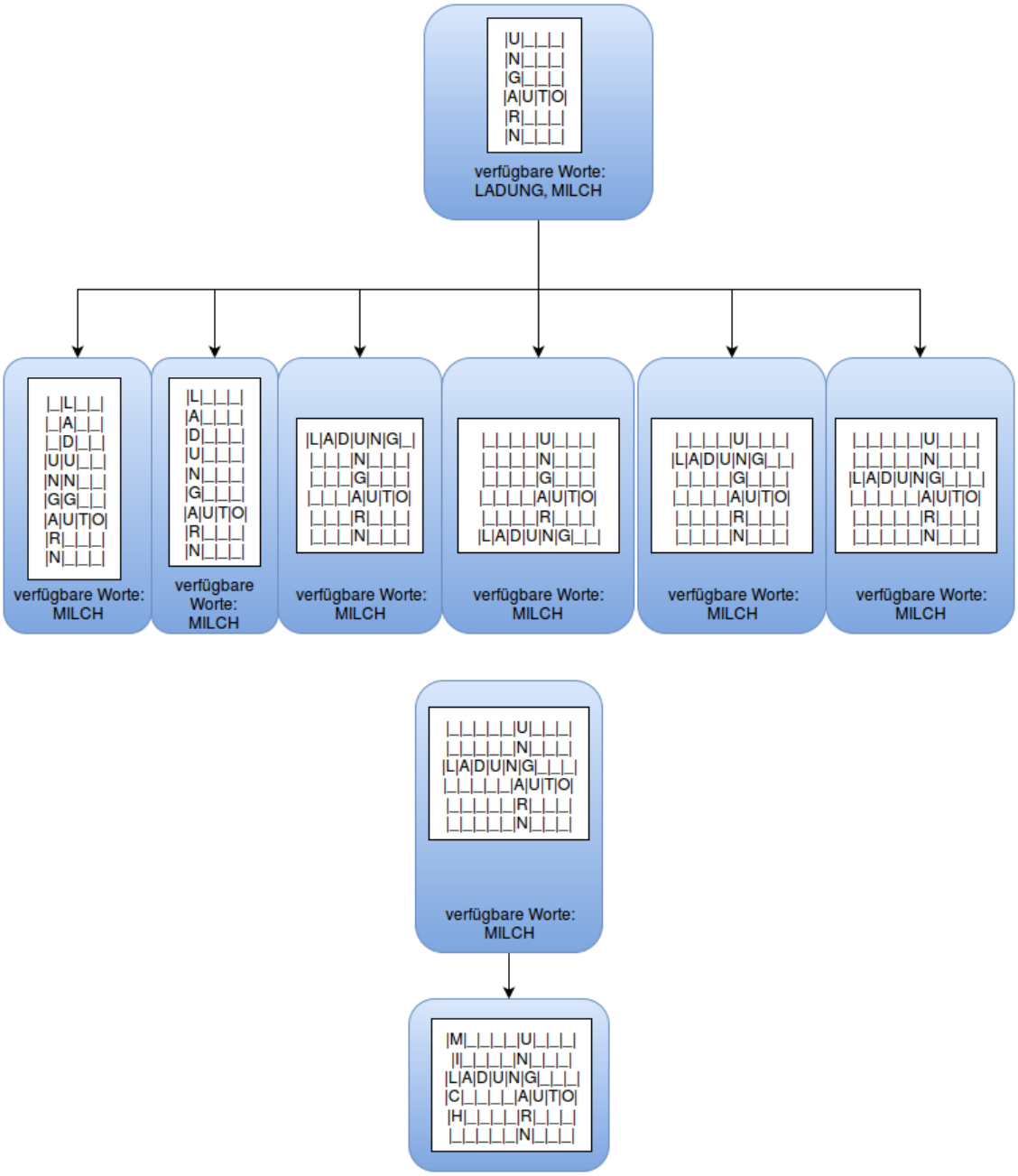
Es wurde eine Kombination mit einem waagerecht angeordneten AUTO eingefügt. Die anderen drei Worte sind bei dieser Kombination noch verfügbar. Nun werden sämtliche Kombinationsmöglichkeiten aus den verfügbaren Worten und der aktuellen Kombination gebildet. Dazu wird mit dem nächsten verfügbaren Wort begonnen. In diesem Fall ist dies LADUNG. Nun wird für jeden Buchstaben aus dem Wort LADUNG geprüft, ob er in der aktuellen Kombination auftaucht. Für das L ist dies nicht der Fall. Das A aus Ladung taucht jedoch als erster Buchstabe von AUTO auf. Als nächstes wird geprüft, ob LADUNG in die aktuelle Kombination sowohl vertikal als auch horizontal eingefügt werden kann. Horizontal ist dies nicht möglich, da das UTO von AUTO nicht mit DUN von LADUNG übereinstimmt. Vertikal kann das Wort jedoch eingefügt werden, da über und unter AUTO keine weiteren Buchstaben stehen. Dazu muss das aktuelle Fehl jedoch erweitert werden. Es wird nun berechnet, dass das Feld nach oben um eins und nach unten um vier erweitert werden muss. Nach der Erweiterung des Feldes kann LADUNG eingefügt werden:



Mit den anderen Buchstaben von LADUNG wird nun analog verfahren. Da das U von LADUNG mit dem U von AUTO übereinstimmt, kann LADUNG noch an einer weiteren Stelle vertikal eingefügt werden. Danach wird versucht das Wort UNGARN einzufügen. Dies ist ebenfalls an zwei Stellen vertikal möglich. Das Wort MILCH kann nicht in die aktuelle Kombination eingefügt werden, da die Buchstaben M, I, L, I, C und H nicht in AUTO enthalten sind. Nun sind alle verfügbaren Worte der aktuellen Kombination ausprobiert worden. Dadurch wurde im Suchbaum eine neue Ebene erstellt. Die Liste mit den aktuellen Kombinationen sieht deshalb nun folgendermaßen aus:

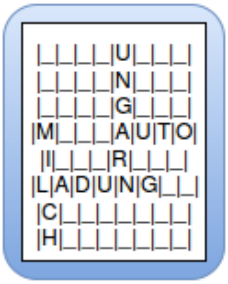


Es fällt auf, dass sich die zuletzt gefundenen Kombinationen vorne in der Liste befinden. Dies liegt daran, dass der Algorithmus eine Tiefensuche durchführt. Wenn eine Breitensuche durchgeführt werden soll, müssen die neuen Kombinationen hinten an die Liste angehängt werden. In der folgenden Iteration wird wieder zunächst das erste Element der Liste betrachtet. Dies ist bei einer Tiefensuche immer die Kombination, welche zuletzt eingefügt wurde. Bei dieser Kombination stehen noch die Worte LADUNG und MILCH zur Verfügung. Da LANDUNG das erste Wort in der Liste ist, wird dieses zuerst in die Kombination mit aufgenommen. Dabei ergeben sie die folgenden Kombinationsmöglichkeiten:

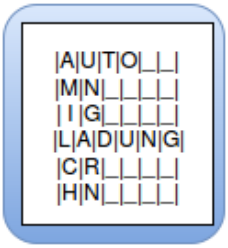


Das Wort Milch kann erneut nicht mit LADUNG und AUTO kombiniert werden. Durch das Einfügen von LADUNG steht nun jedoch ein L in der aktuellen Kombination zur Verfügung. Aus diesem Grund kann in der nächsten Iteration das Wort MILCH zur Kombination hinzugefügt werden. Die erste Lösung ist gefunden. Da bisher noch keine andere Lösung gefunden wurde, wird diese Lösung als aktuell beste Lösung gespeichert. Da das Zeichenfeld dieser Lösung die Dimension 6×9 besitzt hat die gefundene Lösung die Bewertung 52. Sollte eine andere Lösung gefunden werden, wird die Bewertung der neuen Lösung mit 52 verglichen. Wenn die neue Lösung eine kleinere Bewertung erhält wird diese als aktuell beste Lösung gespeichert. Nun werden die anderen

bisher ungelösten Kombinationen in der Liste nach dem gleichen Verfahren betrachtet. Die nächste so gefundene Lösung lautet:



Diese Lösung besitzt die Dimension 8×8, also die Bewertung 64. Damit ist sie schlechter als die bisher gefundene Lösung und wird verworfen. Sollte eine Teillösung bereits schlechter sein als die aktuell beste Lösung wird sie ebenfalls verworfen und der betreffende Zweig nicht mehr weiter verfolgt. Das Verfahren verfolgt nun erneut die noch nicht zu Ende geführten Teillösungen auf die beschriebene Art. Der Algorithmus findet zunächst eine Lösung, die besser als die aktuell beste Lösung ist und speichert diese. In einem anderen Teilbaum wird danach diese Lösung gefunden:



Diese Lösung ist mit einer Bewertung von 36 erneut besser als die aktuell beste Lösung. Im Folgenden findet der Algorithmus keine Lösung, welche eine noch niedrigere Bewertung erzielt. Deshalb handelt es sich bei der so gefundenen Lösung um eine optimale Lösung. Der Algorithmus erzeugt abschließend die folgende Ausgabedatei:

```
;*************************
;       Testfall 1
;*************************
Eingelesene Woerter:
AUTO, LADUNG, UNGARN, MILCH

** Raetsel nicht versteckt

AUTO
MN
IG
LADUNG
CR
HN

** Raetsel versteckt
```

```
17
   AUTOWM
19 MNDBND
   IGAYCN
21 LADUNG
   CRQGTQ
23 HNQRVD

25 Kompaktheitsmass: 36
```

Tests/IHKTestfall1.out

T2: IHK_Testfall2

Beim zweiten Testfall handelt es sich um einen sehr laufzeitkritischen Testfall. Die hohe Wortanzahl führt dazu, dass der Suchbaum sehr hoch ist und somit sehr viele Kombinationen betrachtet werden müssen. Dieser Testfall eignet sich somit gut um die Effizienz der Map-Optimierung zu Testen. Da jedes mal wenn das Feld nach links oder nach oben vergrößert werden muss alle Punkte in der HashMap geändert werden müssen ist hier ein Laufzeitgewinn zwangsläufig der Fall. Bei der Berechnung des zweiten Beispiels ergaben sich die folgenden Laufzeitmessungen:

ohne Map:	19s 785ms
mit Map:	38s 556ms

Der zusätzliche Aufwand beim Erweitern des Arrays wirkt sich also so negativ auf das Laufzeitverhalten aus, dass der schnellere Zugriff auf die verfügbaren Buchstaben nicht ausreicht um das Laufzeitverhalten positiv zu verbessern.

```
1 ;*************************
  ;       Testfall 2
3 ;*************************
  Auto
5 Ladung
  Ungarn
7 Milch
  Essen
9 Versuch
  Mathematik
11 Informatik
  Programmierung
```

Tests/IHKTestfall2.in

```
  ;*************************
2 ;       Testfall 2
  ;*************************
4 Eingelesene Woerter:
AUTO, LADUNG, UNGARN, MILCH, ESSEN, VERSUCH, MATHEMATIK, INFORMATIK,
    PROGRAMMIERUNG
6
```

```
** Raetsel nicht versteckt

     I
ESSEN
 M   F
PAUTO
RTN R
OHGVM
GEAEA
RMRRT
AANSI
MT UK
MILCH
IKAH
E D
R U
U N
N G
G

** Raetsel versteckt

NIELI
ESSEN
TMJOF
PAUTO
RTNMR
OHGVM
GEAEA
RMRRT
AANSI
MTPUK
MILCH
IKAHF
EJDAU
ROUTS
UYNJB
NCGFR
GDLAL

Kompaktheitsmass: 85
```

Tests/IHKTestfall2.out

T3: IHK_Testfall3

```
;*************************
;      Testfall 3
;*************************
Pruefung
Aufgabe
Mathematik
Informatik
```

```
Programmierung
Zeit
```

Tests/IHKTestfall3.in

```
;*************************
;       Testfall 3
;*************************
Eingelesene Woerter:
PRUEFUNG, AUFGABE, MATHEMATIK, INFORMATIK, PROGRAMMIERUNG, ZEIT

** Raetsel nicht versteckt

    MATHEMATIK
            U
    Z       F
PRUEFUNG G
    INFORMATIK
    T       B
PROGRAMMIERUNG

** Raetsel versteckt

KQSMATHEMATIKE
QUGKFBZHWUOWNU
QOHZRAUFVFNVDO
PRUEFUNGDGKIBQ
YPCINFORMATIKY
XEZTTZFSIBEEEG
PROGRAMMIERUNG

Kompaktheitsmass: 98
```

Tests/IHKTestfall3.out

T4: IHK_Testfall4

```
;*************************
;       Testfall 4
;*************************
Programmierung
Zeit
Konzentration
Rechnen
Ergebnis
Integral
Stochastik
Statistik
```

Tests/IHKTestfall4.in

```
;*************************
;       Testfall 4
```

```
;*************************
Eingelesene Woerter:
PROGRAMMIERUNG, ZEIT, KONZENTRATION, RECHNEN, ERGEBNIS, INTEGRAL,
    STOCHASTIK, STATISTIK

** Raetsel nicht versteckt

PROGRAMMIERUNG
     E   Z
     CERGEBNIS
STOCHASTIK
     NSTATISTIK
KONZENTRATION
    INTEGRAL

** Raetsel versteckt

PROGRAMMIERUNG
RHRXEYPJZEFZAM
MVJYCERGEBNISS
STOCHASTIKJFKC
XMETNSTATISTIK
KONZENTRATIONG
RAOINTEGRALFMC

Kompaktheitsmass: 98
```

Tests/IHKTestfall4.out

5.2. Normalfälle

T5: Normalfall1

Der folgende Normalfall prüft, ob Eingabedateien, welche Leerzeichen vor und/oder nach eingegebenen Worten zu Problemen führen, und ob das Problem korrekt bearbeitet wird. Es wurde die folgende Eingabedatei gewählt:

```
;*************************
;       Normalfall 1
;*************************
      blank
baer
     rad
```

Tests/Normalfall1.in

Vor und nach dem Wort blank befinden sich Leerzeichen. Hinter dem Wort baer und vor dem Wort rad ebenfalls. Startet man das Programm mit dieser Eingabedatei kommt es nicht zu einem Fehler und das Programm schließt die Bearbeitung Ordnungsgemäß ab. Die erzeugte Ausgabedatei demonstriert die Korrektheit des Verfahrens im Bezug auf Leerzeichen in der Eingabedatei:

```
;*************************
;       Normalfall 1
;*************************
Eingelesene Woerter:
BLANK, BAER, RAD

** Raetsel nicht versteckt

BAER
 BLANK
   D

** Raetsel versteckt

BAERXT
IBLANK
UHCDWF

Kompaktheitsmass: 18
```

Tests/Normalfall1.out

T6: Normalfall2

Der folgende Test wurde mit sämtlichen Kombinationen an Optimierungsmöglichkeiten ausgeführt.

	Map aktiviert	Map deaktiviert
Tiefensuche	0s 115ms	0s 75ms
Breitensuche	11s 666ms	11s 253ms

Diese Laufzeituntersuchung zeigt deutlich, dass die Tiefensuche aus den in den vorherigen Kapiteln angeführten Gründen die bessere alternative ist. Eine klare Tendenz in eine positive oder negative Richtung kann bei der Beurteilung der Map in diesem Beispiel jedoch nicht gegeben werden. Dazu ist die Problemgröße noch zu klein, die Auswirkungen der Map auf die Laufzeit sind nicht groß genug. In IHKTestfall2 wurde jedoch bereits festgestellt, dass bei großen Problemgrößen von der Verwendung der Map abzuraten ist.

```
Schwanz
Katze
Haustuer
Keller
Teller
Mutter
```

Tests/Normalfall2.in

```
Eingelesene Woerter:
SCHWANZ, KATZE, HAUSTUER, KELLER, TELLER, MUTTER
```

```
** Raetsel nicht versteckt

    K  K
SCHWANZE
 MUTTERL
    Z  L
   TELLER
HAUSTUER

** Raetsel versteckt

HNALKEQKX
SCHWANZEI
SMUTTERLT
KHGTZIZLQ
QADTELLER
HAUSTUERP

Kompaktheitsmass: 54
```

Tests/Normalfall2.out

5.3. Sonderfälle

T7: Sonderfall1

Der folgende Testfall soll prüfen, ob das Programm korrekt mit Wortkombinationen umgeht, bei denen ein Wort Teil eines anderen ist. Die Folgende Eingabedatei stellt einen solchen Fall dar:

```
;*************************
;      Sonderfall 1
;*************************
Ei
Eisen
Eisenerz
```

Tests/Sonderfall1.in

Das Wort Ei liegt vollständig in dem Wort Eisen. Das Wort Eisen wiederum liegt vollständig in dem Wort Eisenerz. Folglich besteht die minimale Lösung lediglich aus dem Wort EISENERZ, da EI und EISEN darin enthalten sind. Die folgende Ausgabedatei demonstriert, dass dieser Sonderfall für den Algorithmus kein Problem darstellt:

```
;*************************
;      Sonderfall 1
;*************************
Eingelesene Woerter:
EI, EISEN, EISENERZ

```

```
** Raetsel nicht versteckt

EISENERZ

** Raetsel versteckt

EISENERZ

Kompaktheitsmass: 8
```

Tests/Sonderfall1.out

T8: Sonderfall2

Der nächste Sonderfall beinhaltet zwei Worte, dessen Buchstaben zwei disjunkte Mengen darstellen. Es darf folglich nicht möglich sein diese Worte zu einem Rätsel zu kombinieren, da nicht mit anderen Worten verbundene Worte nicht gültig sind. Die Eingabedatei sieht wie folgt aus:

```
;*************************
;      Sonderfall 1
;*************************
wort
kaese
```

Tests/Sonderfall2.in

Der Algorithmus erkennt diesen Fall korrekt und gibt einen passende Fehlermeldung auf der Fehlerkonsole aus:

```
java -jar dist/raetselErsteller.jar Tests/Sonderfall2.in
Zu den eingegebenen Worten kann kein Raetsel gebildet werden.
```

Konsolenausgabe:AusgabeSonderfall2

T9: Sonderfall3

Der folgende Sonderfall beinhaltet eine Kombination aus Wörtern, welche bis auf das E vollständig disjunkte Buchstaben aufweisen. In diesem Fall können zwar zwei Wörter miteinander kombiniert werden, das dritte Wort kann jedoch nicht eingefügt werden, da an das E bereits ein horizontales und ein vertikales Wort angrenzen. Auch in diesem Fall erscheint die Fehlermeldung, dass aus den angegebenen Worten kein Raetsel gebildet werden kann.

```
;Sonderfall 3
erdogan
ems
elch
```

Tests/Sonderfall3.in

5.4. Fehlerfälle

T10: Fehlerfall1

In diesem Testfall wird geprüft, wie das Programm darauf reagiert wenn die angegebene Eingabedatei nicht existiert. Die Angabe der Eingabedatei ist nicht optional, ein korrektes Verhalten ist also ein Programmabbruch. Wie erwünscht bricht das Programm ab, diagnostiziert den Fehler korrekt und gibt eine passende Fehlermeldung aus.

```
java -jar dist/raetselErsteller.jar Tests/IHKTestfall5.in
Die Eingabedatei konnte nicht gefunden werden. Das Programm wird
    abgebrochen.
```

Konsolenausgabe:AusgabeFehlerfall1

T11: Fehlerfall2

Die Angabe einer Ausgabedatei ist optional. Wenn ein Pfad angegeben wird muss dieser jedoch auch korrekt sein. In diesem Fall soll die Ausgabe von IHKBeispiel1 nach /dieser/pfad/ist/falsch/IHKTestfall1.out erfolgen. Der angegebene Pfad ist jedoch ungültig. Das Programm erkennt den Fehlerfall wie gewünscht und bricht mit einer passenden Fehlermeldung ab.

```
java -jar dist/raetselErsteller.jar -o dieser/pfad/ist/falsch/
    IHKTestfall1.out Tests/IHKTestfall1.in
Es liegt ein Problem mit dem Pfad der Ausgabedatei vor.
```

Konsolenausgabe:AusgabeFehlerfall2

T12: Fehlerfall3

In diesem Testfall wurde das Programm mit fehlerhaften Argumenten gestartet. Die Argumente -falscher existier nicht und kann vom ArgParser nicht verarbeitet werden. Das Programm zeigt dem Benutzer daraufhin an welche Argumente zur Verfügung stehen und beendet sich ordnungsgemäß. Es kommt nicht zu weiteren Fehlern.

```
java -jar dist/raetselErsteller.jar -falscher Parameter
Benutzung: java -jar raetselErsteller.jar
       [-h] [-l LOG] [-o OUTPUT] [-t] [-a ALGORITHM] EINGABEDATEI
java -jar raetselErsteller.jar: error: unrecognized arguments: '-
    falscher'
```

Konsolenausgabe:AusgabeFehlerfall3

T13: Fehlerfall4

In der Aufgabenanalyse wurde eine gültige Eingabedatei so definiert, dass sie mindestens zwei Wörter enthalten muss. In diesem Fehlerfall wird eine Eingabedatei benutzt, welche nur ein Wort enthält. Die Eingabedatei sieht wie folgt aus:

```
;*************************
;       Fehlerfall 4
;*************************
Auto
```

Tests/Fehlerfall4.in

Auto ist hier das einzige angegebene Wort. Die gewünschte Funktionalität ist also, dass das Programm mit einem Fehler abbricht. Es ist nicht erwünscht, dass dieser Testfall bearbeitet wird und als Lösung ein Rätsel mit nur einem Wort produziert wird. Der Algorithmus verarbeitet die Eingabedatei wie gewünscht und produziert die folgende Fehlermeldung:

```
java -jar dist/raetselErsteller.jar Tests/Fehlerfall4.in
Die Eingabedatei muss mindestens 2 Woerter enthalten, damit das
    Problem sinnvoll zu bearbeiten ist.
```

Konsolenausgabe:AusgabeFehlerfall4

T14: Fehlerfall5

Es wurde ebenfalls definiert, dass die Worte ausschließlich aus Buchstaben bestehen dürfen. Im folgenden Fehlerfall beinhaltet das dritte Wort sowohl Zahlen als auch Sonderzeichen. Es wird also ein Programmabbruch verlangt. Darüber hinaus soll das Programm in diesem Fall einen Hinweis auf die Zeilennummer der fehlerhaften Zeile geben. Wie gewünscht bemerkt das Programm, dass die Eingabedatei falsch formatiert ist. Es weist den Benutzer darauf hin, dass Worte ausschließlich aus Buchstaben bestehen dürfen und gibt die Zeilennummer der fehlerhaften Zeile aus.

```
;*************************
;       Fehlerfall 5
;*************************
Auto
Test
falsches123;?Wort
Haus
```

Tests/Fehlerfall5.in

```
java -jar dist/raetselErsteller.jar Tests/Fehlerfall5.in
Eingabedatei ist falsch formatiert. Ein eingegebenes Wort besteht
    nicht ausschliesslich aus Buchstaben. Der Fehler befindet sich in
    der Naehe von Zeile 6.
```

Konsolenausgabe:AusgabeFehlerfall5

T15: Fehlerfall6

Beim folgenden Testfall befindet sich in der Eingabedatei, zwischen den Wörtern Haus und Hamster eine Leerzeile. Dies ist nicht zulässig. Wie gewünscht bricht das Programm an dieser Stelle ab.

```
;-------------------------------
;     Fehlerfall 6
;-------------------------------
Haus

Hamster
```

Tests/Fehlerfall6.in

```
java -jar dist/raetselErsteller.jar Tests/Fehlerfall6.in
In der Eingabedatei befindet sich eine Leerzeile
```

Konsolenausgabe:AusgabeFehlerfall6

6. Fazit und Ausblick

Es wurde ein Softwaresystem entwickelt, welches in der Lage ist, aus einer gegebenen Menge von Wörtern ein Rätsel zu generieren, welches bezüglich eines Qualitätskriteriums minimal ist. Als Qualitätskriterium wurde der Flächeninhalt des Zeichenfeldes verwendet. Das Softwaresystem ist in der Lage die Eingabedaten aus einer Datei zu lesen und die Ausgabedaten wieder in eine Datei zu schreiben. Die Funktionalitäten zum Einlesen, Ausgeben, Berechnen und Formatieren wurden derart abstrahiert, dass diese leicht auszutauschen sind. Es ist möglich den Algorithmus mit Tiefensuche oder mit Breitensuche zu betreiben. Darüber hinaus wurde eine Optimierung über eine Map vorgenommen, sodass Buchstaben sehr schnell gefunden werden können. Eine Laufzeitanalyse hat hier ergeben das die Verwendung von Map oder Breitensuche keine positiven Auswirkungen auf das Laufzeitverhalten hat und daher nicht empfohlen ist. Das Softwaresystem wurde getestet und diese Tests dokumentiert. Darüber hinaus wurde eine Laufzeitanalyse durchgeführt, welche ergeben hat, dass ein asymptotisch exponentielles Laufzeitverhalten vorliegt.

In einer weiterführenden Entwicklung, sind verschiedene Optimierungen denkbar. Um auch größere Mengen Wörter bearbeiten zu können, ohne lange Laufzeiten in Kauf nehmen zu müssen sind einige Laufzeitoptimierungen möglich:

6.1. Parallelisierung

Das Verfahren lässt sich hervorragend parallelisieren. Da die einzelnen Kombinationen vollständig unabhängig voneinander gebildet werden können, ist kaum Kommunikation zwischen den Threads nötig. Da das Verfahren iterativ implementiert wurde ist es möglich, ohne großen Mehraufwand eine parallelisierte Version zu erzeugen. Dazu müsste lediglich ein Threadpool erzeugt werden, in dem frei werdende Threads Kombinationen aus der Liste erhalten, daraus neue Kombinationen bilden und diese dann wieder in die Liste einfügen.

6.2. Heuristiken

Es ist möglich, durch geschicktes Auswählen von Teilbäumen das Laufzeitverhalten des Algorithmus positiv zu beeinflussen. Wenn beispielsweise sehr schnell eine gute Lösung gefunden wurde, können andere, schlechtere Teilbäume schon sehr früh verworfen werden.

6.2.1. Verfolgen der besten Kombination

Aktuell verwendet die Tiefensuche die zuletzt erstellte Kombination zur weiteren Berechnung in einer neuen Ebene des Suchbaums. Wenn jedoch nicht die letzte erstelle Kombination, sondern die bisher beste erstellte Kombination weiter verfolgt werden würde könnte dies zu einer Verbesserung des Laufzeitverhaltens führen. Dieser Idee liegt die Vermutung zu Grunde, dass gute Lösungen bereits in einem ungelösten Stadium eine kleine Ausdehnung haben.

A. Abweichungen und Ergänzungen zum Vorentwurf

Während der Implementierungsphase musste stellenweise von dem ursprünglichen Entwurf abgewichen werden. Das folgende Kapitel führt diese Abweichungen auf und erläutert die Notwendigkeiten.

A.1. Abweichungen zum Vorentwurf

A.1.1. Einfügen des Startworts in beiden Orientierungen

Im ursprünglichen Entwurf war vorgesehen das Startwort sowohl horizontal als auch vertikal einzufügen. Es wurde festgestellt, dass dies nicht notwendig ist. Dies hängt damit zusammen, dass durch eine Orientierung bereits eine optimale Lösung berechnet werden kann, da die Lösungen durch eine Drehung ineinander überführbar sind. Deshalb wurde auf das vertikale Einfügen des Startwortes verzichtet. Das Laufzeitverhalten konnte somit um den Faktor 2 verbessert werden.

A.1.2. Umstrukturierung der Main-Klasse

Es wurde festgestellt, dass die Konzeption der Main-Klasse Controlling-Aufgaben nicht sauber genug von der eigentlichen Main-Funktion abgegrenzt hat. Durch die Tatsache, dass in der Main-Funktion die Main-Klasse initialisiert wurde litt die flüssige Lesbarkeit des Programmcodes. Deshalb wurde im aktuellen Entwurf die Main-Funktion mitsamt der Funktion für den Argparser und der Funktion zum initialisieren der Logger in eine neue Main-Klasse verschoben. Die Klasse, welche früher den Namen Main hatte wurde in Controller umbenannt. Dieser Entwurf bringt auch den Vorteil mit sich, dass das Softwaresystem einfacher als Bibliothek bereitgestellt werden kann und andere Entwickler das System mit ihren eigenen Main-Funktionen nutzen können.

A.1.3. Änderungen an der Klasse AktuelleKombination

Die Klasse AktuelleKombination benötigte noch einige weitere Methoden, welche im Vorentwurf nicht berücksichtigt worden sind:

1. Kopier-Konstruktor zum Erstellen einer tiefen Kopie
2. Die Methode istFertig zum Abfragen ob alle Wörter verwendet wurden

3. Das Interface Compareable wurde implementiert, um die Kombinationen hinsichtlich ihrer Bewertung vergleichen zu können.

4. Zu Debug-Zwecken wurde eine toString-Methode hinzugefügt

5. Überschreiben von HashCode und Equals

Das Überschreiben von Equals und HashCode ist notwendig, um zu überprüfen ob die aktuelle Kombination bereits in einem anderen Teilbaum existiert und deshalb nicht weiter verfolgt werden muss.

Vereinfachung der Methoden zum Erweitern des Feldes

Wenn ein Wort in eine aktuelle Kombination eingefügt werden soll muss das vorhandene Feld eventuell erweitert werden. Dazu wurden im Vorentwurf vier Methoden benötigt: addLinks, addRechts, addOben und addUnten. Es wurde festgestellt, dass diese Aufteilung nicht zwingend nötig ist. Im aktuellen Entwurf wurden diese vier Methoden auf die Methode **erweitereArray(int x, int y, Punkt p)** zurückgeführt. Die Parameter x und y geben dabei an um wie viel das alte Feld in Richtung x und y erweitert werden muss. Der Punkt p definiert dabei die Position im neuen Feld, an der das alte Feld eingefügt werden soll. Durch diese Optimierung konnte der Methoden- und Codeumfang reduziert werden. Darüber hinaus konnte die Problematik beseitigt werden, dass die alten add-Methoden zum Teil doppelten Code enthielten.

A.1.4. Umstellung der Datenstruktur bei Verwendung der Map

Im Ursprünglichen Entwurf war geplant, dass die Map von char auf eine Liste aus Punkten abbildet. Es hat sich gezeigt, dass an den Punkten wo sich Wörter überschneiden die Punkte doppelt in die Liste eingefügt werden. Um hier performant doppelte Einträge zu verhindern wurde diese Liste durch ein HashSet ersetzt.

A.2. Ergänzungen zum Vorentwurf

A.2.1. Leerzeichen in der Eingabedatei

Im Vorentwurf wurde festgelegt, dass die in der Eingabedatei angegebenen Wörter keine Leerzeichen vor oder nach dem Wort enthalten dürfen. In einer späteren Überlegung ist jedoch aufgefallen, das durch diese Einschränkung die Benutzerfreundlichkeit unnötig leidet. Deshalb wurde diese Einschränkung entfernt. Dazu musste im ursprünglichen Entwurf lediglich die trim-Methode auf den eingelesenen Wörtern aufgerufen werden, um die zusätzliche Leerzeichen zu entfernen.

A.2.2. Zusätzliche Kommandozeilenparameter

Zusätzlich zu den im Vorentwurf geplanten Kommandozeilen-Parametern wurden zwei weitere hinzugefügt. Zur komfortableren Zeitmessung wurde der Parameter -t hinzugefügt. Um einfach zwischen Breiten- und Tiefensuche wechseln zu können wurde darüber hinaus der Parameter -a zum festlegen des Algorithmus eingefügt.

A.2.3. Die Klasse KeineLoesungException

Für den Fall, dass zu den eingelesenen Wörtern keine gültige Kombination gefunden werden kann war im Vorentwurf geplant null zurückzugeben. Damit in der aufrufenden Methode nicht jedes mal geprüft werden muss, ob das zurückgegebene Objekt null ist wurde für diesen Fall eine zusätzliche Exception implementiert. Somit musste in der loese-Methode des Controllers nur ein weiterer catch-Block hinzugefügt werden.

A.3. Nicht benötigte Schnittstellen

Im Vorentwurf waren für die Klasse AusgabeDaten die beiden Methoden getKommentare() und getWorte() vorgesehen, damit der Formatierer diese Daten in die Formatierung mit einbeziehen kann. Da der StdFormatierer jedoch im aktuellen Entwurf die toString-Methode der Klasse AusgabeDaten nutzt sind diese beiden Methoden nicht mehr zwingend notwendig. Es wurde jedoch entschieden diese Methoden auch im aktuellen Entwurf zu übernehmen. Dies hat den Hintergrund, dass die Möglichkeit einer alternativen Formatierung, etwa durch hinzufügen eines neuen Formatierers, für zukünftige Entwicklungen offen gehalten werden soll.

B. Hilfsmittel

Bei der Erstellung des Softwaresystems wurden die folgenden Hilfsmittel verwendet:

Git

Zur Versionskontrolle wurde das Versionskontrollsystem Git verwendet. In regelmäßigen Abständen wurden damit Sicherheitskopien erstellt, sodass jederzeit zu alten Versionen zurückgekehrt werden konnte.

Argparser

Um den Argparser nicht selber implementieren zu müssen wurde hier auf eine fertige Lösung zurückgegriffen und **argparse4j** verwendet.

Ant

Um das Kompilieren und Ausführen des Codes zu erleichtern wurde das Entwicklungswerkzeug Ant verwendet. Hier wurden benötigte Targets definiert, welche aufgerufen werden können um vorher definierte Aufgaben automatisiert zu bearbeiten.

Erstellung von Diagrammen

Für die Erstellung von UML-Diagrammen wurde **Visual Paradigm** verwendet. Die Nassi-Schneiderman-Diagramme wurden mit dem Programm **structorizer** erstellt.

C. Benutzeranleitung

C.1. Systemvoraussetzungen

C.1.1. Installierte Programme

Zum Ausführen des Programms wird eine Java Laufzeitumgebung der Version JavaSE-1.7 oder höher vorausgesetzt. Das mitgelieferte Skript *runall.sh* setzt eine UNIX-Distribution mit installierter Bash voraus. Um das mitgelieferte Ant-Skript nutzen zu können muss zusätzlich Ant installiert sein. Das Programm ist jedoch auch ohne Ant benutzbar. Da das Programm als .zip-Archiv ausgeliefert wird muss ein Programm zum entpacken zur Verfügung stehen.

C.1.2. Hardware

Um Probleme mit einer größeren Anzahl Worte zu bearbeiten ist das folgende Minimum an Hardwarespezifikationen empfohlen:

- **Prozessor:** 2x1.8 GhZ
- **Arbeitsspeicher:** 1 Gib

Denken Sie daran, ihrer Java Laufzeitumgebung zusätzlichen Speicher zuzuweisen, wenn dies benötigt wird. Probleme mit wenigen Wörtern können auch mit niedrigeren Systemvoraussetzungen bearbeitet werden.

C.2. Verzeichnisstruktur

Entpacken Sie die Datei RaetselErsteller.zip. Danach finden Sie die folgende Verzeichnisstruktur vor.

```
RaetselErsteller
├── src
├── dist
├── lib
├── doc
├── Tests
├── build.xml
└── runall.sh
```

```
├── Dokumentation.pdf
└── bin
```

Der Ordner src enthält den Quellcode. Die Datei raetselErsteller.jar befindet sich in dem Ordner dist. Das Verzeichnis lib beinhaltet die verwendeten Bibliotheken. Die Entwicklerdokumentation als Javadoc befindet sich im Ordner doc. Der Ordner Tests beinhaltet alle mitgelieferten Testfälle mitsamt Ausgabedateien. Die Datei build.xml definiert verschiedene Targets, welche von Ant verwendet werden. Darüber hinaus enthält der Ordner das Bash-Skript runall.sh. Dieses Skript führt alle Testfälle im Ordner Tests nacheinander aus. Die Datei Dokumentation.pdf beinhaltet die Dokumentation. Im Ordner bin finden sich die Kompilierten .class Dateien.

Im Ordner Tests befinden sich die folgenden Ein- und Ausgabedateien:

```
RaetselErsteller
├── ...
├── Tests
│   ├── IHKTestfall1.in
│   ├── IHKTestfall1.out
│   ├── IHKTestfall2.in
│   ├── IHKTestfall2.out
│   ├── IHKTestfall3.in
│   ├── IHKTestfall3.out
│   ├── IHKTestfall4.in
│   ├── IHKTestfall4.out
│   └── ...
└── ...
```

Diese Testfälle dienen der Demonstration der Funktionalität und der Überprüfung von Fehler- und Sonderfällen.

C.3. Programmaufruf über die Kommandozeile

Das Programm muss über die Kommandozeile aufgerufen werden. Führen Sie dazu den Befehl

java -jar raetselLoeser.jar [-h] [-l LOG] [-o OUTPUT] [-t] [-a ALGORITHM] EINGABEDATEI

aus um das Programm zu starten. Durch Angabe von -h oder −−hilfe können Sie weitere Hilfen zu den verfügbaren Parametern erhalten. Die Option -l oder –log zeigt Logging-Informationen auf dem Bildschirm an, sodass die Programmausführung besser nachvollzogen werden kann. Wenn Sie den Pfad oder den Namen der Ausgabedatei selbst definieren möchten, so kann dazu die Option -o oder −−output verwendet werden. Wenn diese Option nicht genutzt wird, benutzt das Programm den Pfad der Eingabedatei und ersetzt im Dateinamen das .in durch .out. Über -a oder −−algorithm kann der Lösungsalgorithmus verändert werden. Gültige Argumente sind Tiefensuche und Breitensuche. Die empfohlene Einstellung ist hier Tiefensuche. Abschließend muss eine Eingabedatei angegeben werden. Die Angabe der Eingabedatei ist nicht optional und wird zur Pro-

grammausführung zwingend benötigt.

C.4. Benutzung von Ant

Im Verzeichnis RaetselErsteller befindet sich eine Datei build.xml. Diese Datei wird von Ant verwendet um verschiedene Aufgaben zu automatisieren. Es folgt eine Übersicht der verschiedenen Ant-Befehle.

ant init Erstellt die Ordner bin und dist.

ant clean Löscht die Ordner bin, dist und doc.

ant javadoc Erstellt eine Dokumentation im Ordner doc.

ant create-manifest Erstellt die zum Erstellen eines Archivs benötigte Manifestdatei.

ant compile Kompiliert alle in src befindlichen Dateien und schreibt die .class Dateien nach bin.

and archive Erstellt ein .jar-Archiv.

and execute Führt das Programm aus.

Es ist auch möglich ant ohne Angabe eines Targets auszuführen. In diesem Fall wird execute ausgeführt, da dies als Standardtarget eingestellt ist. Sie müssen bei der Ausführung der Targets nicht darauf achten, alle vorher benötigten Targets auszuführen, da Ant diese Aufgabe selbstständig übernimmt.

C.5. Ausführung der Testbeispiele

Die Datei runall.sh führt alle Testfälle im Ordner Tests aus. Dabei werden alle Dateien als Eingabedateien betrachtet die nicht auf oder .out enden. Die zugehörigen Ausgabedateien enden dann auf .out und befinden sich im selben Ordner. Bereits existierende Ausgabedateien werden überschrieben. Das Skript lässt sich mit *./runall.sh* starten.

D. Entwicklungsumgebung

Programmiersprache	:	Java
Compiler	:	javac Version 1.8
Rechner	:	Intel Core i7-3820 CPU @ 3.60GHz x 8, 16Gib Ram
Betriebssystem	:	Ubuntu 14.4

E. Quellcode

E.1. raetselErsteller

E.1.1. logik

```
package raetselErsteller.logik;

import java.util.logging.ConsoleHandler;
import java.util.logging.Level;

import net.sourceforge.argparse4j.ArgumentParsers;
import net.sourceforge.argparse4j.impl.Arguments;
import net.sourceforge.argparse4j.inf.ArgumentParser;
import net.sourceforge.argparse4j.inf.ArgumentParserException;
import net.sourceforge.argparse4j.inf.Namespace;
import raetselErsteller.Konstanten;
import raetselErsteller.io.FileIO;
import raetselErsteller.io.format.StdFormat;
import raetselErsteller.logik.algorithmus.BacktrackingAlgorithmus;

/**
 * Diese Klasse beinhaltet den Startpunkt des Programms und die Logik
 * zum Parsen von Argumenten. Ausserdem wird in dieser Klasse das
 * Logging konfiguriert und die benoetigten Logger auf das vom
   Benutzer
 * gewuenscht Level eingestellt.
 * @author Felix Kibellus
 * */
public class Main {

  /**
   * Startpunkt des Programms. Zunaechst werden die uebergebenen
   * Argumente analysiert und auf Korrektheit geprueft. Dann wird das
   * Programm, wie in den Argumenten beschrieben, konfiguriert.
   * Anschliessend wird es gestartet.
   * */
  public static void main(String[] args) {
    Namespace ns = parseArgs(args);// Parsen der Argumente
    // initialisieren des Loggers
    if (ns.getString("log") == null)
      initLogger(Level.OFF);
    else
      initLogger(Level.parse(ns.getString("log")));
    // timestamp
```

```
    Controller.timestamp = ns.getBoolean("timestamp");
    // algorithmus
    if (ns.getString("algorithm").equals("Breitensuche"))
      Controller.tiefensuche = false;
    // erstellen der Klasse Controller
    Controller main = null;
    String out = ns.getString("output");
    if (out != null)
      main = new Controller(
          ns.getString(Konstanten.inputData), out);
    else
      main = new Controller(ns.getString(Konstanten.inputData));
    // starten des Loesungsalgorithmus
    main.loese();
  }

  /**
   * Diese Funktion liest die uebergebenen Argumente ein und reagiert
   * darauf. Sollte der Benutzer falsche Argumente angeben oder die
   * Hilfe anfordern, so wird das Programm in dieser Methode beendet.
   *
   * @param args
   *            Die von Benutzer hinter dem Programmnamen
   *            eingegebenen Argumente.
   * @return Eine Map, welche von Argument auf Parameter abbildet.
   * */
  private static Namespace parseArgs(String[] args) {
    String name = Konstanten.projectname;
    ArgumentParser parser = ArgumentParsers
        .newArgumentParser(name).defaultHelp(true)
        .description(Konstanten.beschreibung);
    parser.addArgument("-l", "--log").help(Konstanten.log);
    parser.addArgument("-o", "--output").help(Konstanten.output);
    parser
        .addArgument("-t", "--timestamp")
        .help(Konstanten.timeStamp)
        .action(Arguments.storeTrue());
    parser.addArgument(Konstanten.inputData).help(Konstanten.input);
    parser
        .addArgument("-a", "--algorithm").help(Konstanten.algo)
        .setDefault("Tiefensuche");
    Namespace ns = null;
    try {
      ns = parser.parseArgs(args);
    } catch (ArgumentParserException e) {
      // Der Parser gibt selbst eine Fehlermeldung aus, und
      // erklaert dem Benutzer wie das Programm zu verwenden ist
      //(korrekte Parameter).
      parser.handleError(e);
      System.exit(1);// beendet das Programm
    }
    return ns;
```

```
  }
92
  /**
94  * Diese Funktion konfiguriert die Logger aller Klassen. Der
    * Default-Handler zur Ausgaeb wird entfernt und ein eigener
96  * Handler gesetzt. Bei diesem Handler wird ein Logging-Level
    * gesetzt, sodass er nur Meldungen ausgibt, die mindestens dem
98  * gesetzten Level entsprechen.
    *
100 * @param level
    *            Alle Logging-Meldungen die mindestens dieses Level
102 *            haben werden ausgegeben.
    * */
104 private static void initLogger(Level level) {
      // deaktiviere die default handler
106   Controller.logger.setUseParentHandlers(false);
      BacktrackingAlgorithmus.logger.setUseParentHandlers(false);
108   StdFormat.logger.setUseParentHandlers(false);
      FileIO.logger.setUseParentHandlers(false);
110   // logger level setzen
      Controller.logger.setLevel(Level.ALL);
112   BacktrackingAlgorithmus.logger.setLevel(Level.ALL);
      StdFormat.logger.setLevel(Level.ALL);
114   FileIO.logger.setLevel(Level.ALL);
      // eigenen Handler erstellen
116   ConsoleHandler consoleHandler = new ConsoleHandler();
      consoleHandler.setLevel(level);
118   // eigenen handler setzen
      Controller.logger.addHandler(consoleHandler);
120   BacktrackingAlgorithmus.logger.addHandler(consoleHandler);
      StdFormat.logger.addHandler(consoleHandler);
122   FileIO.logger.addHandler(consoleHandler);
  }
124
}
```

raetselErsteller/logik/Main.java

```
1  package raetselErsteller.logik;

3  import java.util.logging.Level;
   import java.util.logging.Logger;
5
   import raetselErsteller.Konstanten;
7  import raetselErsteller.daten.AusgabeDaten;
   import raetselErsteller.daten.EingabeDaten;
9  import raetselErsteller.io.AusgabeDatenSchreiber;
   import raetselErsteller.io.ConsoleErrorHandler;
11 import raetselErsteller.io.EingabeDatenLeser;
   import raetselErsteller.io.ErrorHandler;
13 import raetselErsteller.io.FileIO;
   import raetselErsteller.io.FileIOException;
15 import raetselErsteller.io.format.FormatException;
```

```
import raetselErsteller.io.format.Formatierer;
import raetselErsteller.io.format.StdFormat;
import raetselErsteller.logik.algorithmus.BacktrackingAlgorithmus;
import raetselErsteller.logik.algorithmus.KeineLoesungException;
import raetselErsteller.logik.algorithmus.LoesungsAlgorithmus;

/**
 * Diese Klasse stellt den zentralen Punkt der Programmarchitektur dar.
 * Sie verwaltet die einzelnen Aufgaben und leitet Funktionsaufrufe an
 * die entsprechenden Schnittstellen von Daten-, Ein/Ausgabeschicht und
 * Algorithmus weiter.
 *
 * @author Felix Kibellus
 * */
public class Controller {

  public static Logger logger = Logger.getLogger(Controller.class
      .getName());

  private EingabeDatenLeser eingabeLeser;
  private AusgabeDatenSchreiber ausgabeSchreiber;
  private LoesungsAlgorithmus algorithmus;
  private ErrorHandler errorHandler;

  protected static boolean timestamp;
  protected static boolean tiefensuche = true;

  /**
   * Initialisiert die Controller-Klasse. In diesem Konstruktor wird
   * festgelegt, wie die Ein- und Ausgabe erfolgen soll. Sollte keine
   * Interaktion ueber Files mehr erwuenscht sein, so muss hier die
   * Klasse FileIO durch eine eigene Klasse, welche die noetigen
   * Interfaces implementiert, ersetzt werden. Ausserdem wird in
   * diesem Konstruktor der ErrorHandler gesetzt. Sollte keine
   * Fehlerausgabe auf StandardError mehr erwuenscht sein, so muss
   * das betreffende Objekt durch einen alternativen ErrorHandler
   * angepasst werden.
   *
   * @param inPath
   *            Der Pfad zur Eingabedatei
   * @param outPath
   *            Der Pfad zur Ausgabedatei
   * */
  public Controller(String inPath, String outPath) {
    logger.log(Level.FINER, "starte Methode");

    Formatierer dataFormat = new StdFormat();
    FileIO io = new FileIO(inPath, outPath, dataFormat);
    eingabeLeser = io;
    ausgabeSchreiber = io;
```

```
    algorithmus = new BacktrackingAlgorithmus(tiefensuche);
67    errorHandler = new ConsoleErrorHandler();
  }
69
  /**
71 * Wenn dieser Konstruktor aufgerufen wird, wird der Ausgabepfad
   * ueber die Funktion getDefaultOutputPath bestimmt.
73 * */
  public Controller(String inPath) {
75  this(inPath, getDefaultOutputPath(inPath));
  }
77
  /**
79 * Diese Funktion liefert den Default-Dateinamen der Ausgabedatei.
   * Endet der Dateiname der Eingabedatei nicht auf .in, so wird nur
81 * .out angehaengt. Endet der Dateiname der Eingabedatei auf .in,
   * so wird das .in durch .out ersetzt.
83 *
   * @param in
85 *            der Pfad zur Eingabedatei
   * */
87 private static String getDefaultOutputPath(String in) {
    logger.log(Level.FINER, "starte Methode");
89
    String inEnd = Konstanten.dateiendungIn;
91  String outEnd = Konstanten.dateiendungOut;
    if (!in.endsWith(inEnd))
93    return in + outEnd;
    String result = in.substring(0, in.length() - 3)
95      + outEnd;
    return result;
97  }

99  /**
   * In dieser Methode werden zunaechst die Eingabedaten aus der
101 * Eingabedatei gelesen. Danach wird der Loesungsalgorithmus mit
   * den Eingabedaten ausgefuehrt. Zum Schluss werden die
103 * Ausgabedaten in eine Ausgabedatei geschrieben. Wenn es bei einem
   * der beschriebenen Schritten zu einem Fehler kommt, wird die
105 * passende Fehlermeldung ueber den errorHandler ausgegeben.
   * */
107 public void loese() {
    logger.log(Level.FINER, "starte Methode");
109
    try {
111   // Lesen der Eingabedaten
      EingabeDaten input = eingabeLeser.leseEingabe();
113   // Aufruf der Loesungsroutine
      AusgabeDaten output = null;
115   if (timestamp) {
        // mit Zeitmessung
117     long t1 = System.currentTimeMillis();
```

```
        output = algorithmus.loese(input);
        long t2 = System.currentTimeMillis();
        long runtime = t2 - t1;
        int seconds = (int) runtime / 1000;
        int millis = (int) runtime % 1000;
        System.out.println(Konstanten.laufzeit + seconds + "s "
            + millis + "ms");
      } else {
        // Ohne Zeitmessung
        output = algorithmus.loese(input);
      }
      // Schreiben der Ausgabedaten
      ausgabeSchreiber.schreibeAusgabe(output);
      // Fehlerbehandlung
    } catch (FileIOException e) {
      switch (e.getType()) {
        case InputFileNotFound :
          errorHandler
              .zeigeFehler(Konstanten.inputFileNotFound);
          break;
        case OutputPathInvalid :
          errorHandler
              .zeigeFehler(Konstanten.outputPathInvalid);
          break;
        default :
          errorHandler.zeigeFehler(Konstanten.unknownIoErr);
      }
    } catch (FormatException e) {
      switch (e.getType()) {
        case KeinBuchstabe :
          int line = e.getLine();
          errorHandler.zeigeFehler(Konstanten
              .buchstabeInZeile(line));
          break;
        case NichtGenugWorte :
          errorHandler
              .zeigeFehler(Konstanten.nichtGenugWorte);
          break;
        case Leerzeile:
          errorHandler.zeigeFehler(Konstanten.leerzeile);
          break;
        default :
          errorHandler
              .zeigeFehler(Konstanten.unknownFormatErr);
          break;
      }
    } catch (KeineLoesungException e) {
      errorHandler.zeigeFehler(Konstanten.keineLoesung);
    } catch(OutOfMemoryError e){
      errorHandler.zeigeFehler(Konstanten.keinSpeicher);
    }
  }
```

```
171 }
```

raetselErsteller/logik/Controller.java

algorithmus

```
1  package raetselErsteller.logik.algorithmus;
2
3  import raetselErsteller.daten.EingabeDaten;
4  import raetselErsteller.daten.AusgabeDaten;
5
6  /**
7   * Dieses Interface dient der abstraktion des Loesealgorithmus. Es
8   * definiert die Schnittstelle fuer einen Algorithmus, welcher aus den
9   * eingelesenen Eingabedaten die benoetigten Ausgabedaten generiert.
10  *
11  * @author Felix Kibellus
12  * */
13 public interface LoesungsAlgorithmus {
14
15   /**
16    * Kombiniert die Worte aus den Eingabedaten zu einem Raetsel mit
17    * minimaler Ausdehnung(Flaecheninhalt).
18    *
19    * @param inputData
20    *            Ein Objekt der Klasse EingabeDaten, welches die Worte
21    *            enthaelt, welche zu einem Raetsel kombiniert werden
22    *            sollen.
23    *
24    * @return Das fertige Raetsel mit minimaler Ausdehnung.
25    * */
26   AusgabeDaten loese(EingabeDaten in)
27       throws KeineLoesungException;
28 }
```

raetselErsteller/logik/algorithmus/LoesungsAlgorithmus.java

```
1  package raetselErsteller.logik.algorithmus;
2
3  import java.util.LinkedList;
4  import java.util.List;
5  import java.util.Random;
6  import java.util.Set;
7  import java.util.logging.Level;
8  import java.util.logging.Logger;
9
10 import raetselErsteller.daten.AktuelleKombination;
11 import raetselErsteller.daten.AusgabeDaten;
12 import raetselErsteller.daten.EingabeDaten;
13 import raetselErsteller.daten.Punkt;
```

```
14 import raetselErsteller.daten.SchlechtesteAktuelleKombination;

16 /**
   * Diese Klasse stellt einen Loesungsalgorithmus zum Erstellen von
18 * wortraetseln dar. Sie ist in der Lage, zu einer gegebenen Menge von
   * Worten ein Raetsel mit minimaler Ausdehnung zu finden. Dazu wird
      ein
20 * Backtracking-Algorithmus verwendet, der saemtliche Kombinations-
   * moeglichkeiten ausprobiert, und somit einen Suchbaum aufbaut. Da in
22 * diesem alle moeglichen Loesungen enthalten sind wird die beste
   * Loesung gefunden. Es besteht die Moeglichkeit diesen Suchbaum in
24 * Breiten- oder Tiefensuche zu durchlaufen. Empfohlen ist die
   * Verwendung einer Tiefensuche.
26 *
   * @author Felix Kibellus
28 * */
public class BacktrackingAlgorithmus implements LoesungsAlgorithmus {
30
  // Wird zum Loggen der Funktionalitaet benoetigt
32 public static Logger logger = Logger
      .getLogger(BacktrackingAlgorithmus.class.getName());
34
  // gibt an ob der Suchbaum in Tiefen- oder Breitensuche durchlaufen
36 // wird
  private boolean tiefensuche;
38
  /**
40  * Erstellt ein Objekt vom Typ BacktrackingAlgorithmus
    *
42  * @param tiefensuche
    *            gibt an ob der Algorithmus Tiefensuche oder
44  *            Breitensuche verwenden soll
    *
46  * */
  public BacktrackingAlgorithmus(boolean tiefensuche) {
48   this.tiefensuche = tiefensuche;
  }
50
  /**
52  * Diese Methode liefert ausgehend von eingelesenen Eingabedaten
    * ein Raetsel mit minimaler Ausdehnung. Dazu wird ein
54  * Backtracking-Ansatz werdende.
    *
56  * @param eingabe
    *            Ein Objekt der Klasse EingabeDaten, welches die Worte
58  *            enthaelt, welche zu einem Raetsel kombiniert werden
    *            sollen.
60  *
    * @return Das fertige Raetsel mit minimaler Ausdehnung.
62  * */
  @Override
64 public AusgabeDaten loese(EingabeDaten eingabe)
```

```
      throws KeineLoesungException {
66    logger.log(Level.FINER, "starte Methode");
      // aktuell beste Kombination merken
68    AktuelleKombination besteLoesung = //schlechtest moegliche
          new SchlechtesteAktuelleKombination();
70
      // Merke aktuelle Blaetter des Suchbaums
72    // Suchbaum wird als Liste der unteren Elemente des Baums
      // gespeichert
74    LinkedList<AktuelleKombination> q = new LinkedList<>();
      // Einfuegen des Wurzelknoten (beliebiges Wort)
76    // horiz0ntale Ausrichtung genuegt hier, da die durch vertikale
      // Ausrichtung gefundene Loesung in diese ueberfuehrt werden
78    // kann
      q.add(new AktuelleKombination(eingabe, true));
80
      while (!q.isEmpty()) {
82      // hole Kombination aus Suchbaum
        AktuelleKombination a = q.poll();
84
        // a besser als aktuell beste?
86      if(besteLoesung.compareTo(a) < 0){
          if(a.istFertig()){//a fertige kombination?
88          besteLoesung = a;//a ist neue beste loesung
            continue;
90        }
        } else {
92        continue;//verwerfe a
        }
94
        // fuer alle verfuegbaren Worte
96      for (String s : a.getVerfuegbareWoerter()) {
          // bilde alle Kombinationsmieglichkeiten
98        List<AktuelleKombination> kombis = bildeKombinationen(
              s, a);
100       // fuer jede Kombinationsmoeglichkeit
          for (AktuelleKombination k : kombis)
102         // fuege in Suchbaum ein, wenn diese Kombination
            // nicht schon
104         // in einem anderen Teilbaum vorliegt.
            if (!q.contains(k)){
106           logger.log(Level.FINEST, "\n"+k.toString());
              // Tiefenschue=> vorne in Liste einfuegen
108           if (tiefensuche)
                q.addFirst(k);
110           else
                q.offer(k);
112         }

114     }
      }
116   if (besteLoesung instanceof SchlechtesteAktuelleKombination) {
```

```
      throw new KeineLoesungException();
    } else {
      char[][] schwer = fuelleAuf(besteLoesung.getKombi());
      return new AusgabeDaten(
          eingabe, besteLoesung.getKombi(), schwer);
    }
  }

  /**
   * Diese Methode bildet alle Kombinationsmoeglichkeiten aus einem
   * Wort und einer AktuellenKombination. Dabei werden nur korrekte
   * Kombinationen erstellt. Das Wort wird also an einer Stelle
   * eingefuegt, es mindestens ein anderes Wort der Kombination
   * schneidet. Ausserdem wird das Wort nur dann eingefuegt, wenn an
   * allen Kreuzungspunkten mit andern Woertern die Buchstaben
   * uebereinstimmen. Die uebergebene Kombination wird nicht
   * veraender, sondern es werden Kopien der Kombination erzeugt. Ist
   * es in keinem Fall moeglich das Wort in die Kombination
   * einzufuegen, so ist die zurueckgegebene Liste leer.
   *
   * @param wort
   *            Das Wort, welches in die Kombination eingefuegt
   *            werden soll.
   * @param kombi
   *            Die Kombination, in welche das Wort eingefuegt werden
   *            soll
   * @return eine Liste aus allen Kombinationsmoeglichkeiten von Wort
   *         und Kombination
   * */
  private List<AktuelleKombination> bildeKombinationen(String wort,
      AktuelleKombination kombi) {
    List<AktuelleKombination> result = new LinkedList<>();
    // iteriere ueber alle Buchstaben des Wortes
    for (int i = 0; i < wort.length(); i++) {
      // Alle Punkte an denen der Buchstabe in der Kombination
      // vorkommt
      Set<Punkt> a = kombi.getAllePunkteZu(wort.charAt(i));
      // fuer jeden gefunden Punkt der Kombination
      for (Punkt p : a) {
        // kann das Wort horizontal eingefuegt werden?
        if (kombi.istFrei(wort, p, i, true)) {
          AktuelleKombination copy = new AktuelleKombination(
              kombi);// Kopiere die Konfiguration
          // fuege Wort horizontal ein
          copy.add(wort, p, i, true);
          // fuege erhaltene Kombination zum Resultat hinzu
          result.add(copy);
        }
        // kann das Wort vertikal eingefuegt werden?
        if (kombi.istFrei(wort, p, i, false)) {
          AktuelleKombination copy = new AktuelleKombination(
              kombi);
```

```
            // fuege Wort vertikal ein
170         copy.add(wort, p, i, false);
            // fuege erhaltene Kombination zum Resultat hinzu
172         result.add(copy);
          }
174     }
      }
176   return result;
  }
178
  /**
180  * Fuellt alle freien Felder des Arrays mit zufaelligen
   * Grossbuchstaben. Ein Feld gilt als frei, wenn es den Wert '\0'
182  * enthaelt. Das uebergebene Array wird nicht veraendert. Fuer die
   * Rueckgabe wird ein neues Array erzeugt. Diese Methode wird
184  * verwendet, um aus dem mit den eingelesenen Worten erzeugten
   * Array ein fertiges Raetsel zu machen.
186  *
   * @param einfach
188  *            Ein zweidimensionales Zeichenarray, welches mit
   *            zufaelligen Buchstaben aufgefuellt werden soll.
190  * @return Ein neues Feld, welches mit zufaelligen Grossbuchstaben
   *         aufgefuellt wurde.
192  * */
  private char[][] fuelleAuf(char[][] einfach) {
194   Random r = new Random(); // fuer Zufallszahlen
    char[][] result = new char[einfach.length][einfach[0].length];
196   for (int i = 0; i < result.length; i++) {
      for (int j = 0; j < result[0].length; j++) {
198     char c = einfach[i][j]; // alter Wert
        if (c == '\0') { // muss aufgefuellt werden?
200       // erstelle zufaelligen Buchstaben
          c = (char) ('A' + r.nextInt(26));
202     }
        result[i][j] = c;
204   }
    }
206   return result;
  }
208
}
```

raetselErsteller/logik/algorithmus/BacktrackingAlgorithmus.java

```
1 package raetselErsteller.logik.algorithmus;

3 /**
 * Diese Exception repraesentiert den Ausnahmefall, dass aus den
5 * angegebenen Worten kein Raetsel gebildet werden kann. Dies ist
 * beispielsweise der Fall, wenn die Worte disjunkte Buchstabenmengen
7 * enthalten.
 *
9 * @author Felix Kibellus
```

```
 * */
public class KeineLoesungException extends Exception {

  private static final long serialVersionUID = 1L;
}
```

raetselErsteller/logik/algorithmus/KeineLoesungException.java

E.1.2. daten

```
package raetselErsteller.daten;

/**
 * Diese Klasse repraesentiert einen Punkt (x,y) im R^2 und wird
 * verwendet um Positionen von Buchstaben in einer Kombination zu
 * beschreiben.
 *
 * @author Felix Kibellus
 * */
public class Punkt {

  public int x; // x-Koordinate des Punktes
  public int y; // y-Koordinate des Punktes

  /**
   * Erstellt ein Objekt der Klasse Punkt
   *
   * @param x
   *            Die x-Koordinate des Punktes
   * @param y
   *            Die y-Koordinate des Punktes
   * */
  public Punkt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  /**
   * Erstellt eine tiefe Kopie eines Punktes
   * */
  public Punkt(Punkt p) {
    this.x = p.x;
    this.y = p.y;
  }

  /**
   * Wandelt des Punkt in einen String um. Format: (x,y)
   * */
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
```

```
42
   @Override
44 public int hashCode() {
     return x+y;
46 }

48 @Override
   public boolean equals(Object obj) {
50   if (this == obj)
       return true;
52   if (obj == null)
       return false;
54   if (getClass() != obj.getClass())
       return false;
56   Punkt other = (Punkt) obj;
     if (x != other.x)
58     return false;
     if (y != other.y)
60     return false;
     return true;
62 }


64


66 }
```

raetselErsteller/daten/Punkt.java

```
package raetselErsteller.daten;
2
import java.util.List;
4
/**
6 * Diese Klasse dient der Repraesentation der Eingabedaten. Ein Objekt
  * dieser Klasse enthaelt:
8 * -Liste der Kommentare ausn der Eingabedatei
  * -List der Worte, welche in einer Kombination verwendet werden
    sollen
10 *
  * @author Felix Kibellus
12 * */
public class EingabeDaten {
14   // liste der Kommentare aus der Eingabedatei
     private List<String> kommentare;
16   // liste der Worte, welche kombiniert werden sollen
     private List<String> woerter;
18
     /**
20    * Erstellt ein Objekt der Klasse EingabeDaten
      *
22    * @param kommentar
      *            Liste der Kommentare aus der Eingabedatei
24    * @param woerter
```

```
     *            Liste der Worte aus der Eingabedatei
     * */
    public EingabeDaten(List<String> kommentar, List<String> woerter) {
      this.kommentare = kommentar;
      this.woerter = woerter;
    }

    /**
     * Liefert die Liste der Kommentare zurueck. Diese Methode wird in
     * der aktuellen implementierung nicht verwendet, wird aber
     * benoetigt, wenn ein alternativer Formatierer die Eingabedaten
     * formatieren soll.
     * */
    public List<String> getKommentare() {
      return kommentare;
    }

    /**
     * Liefert die Liste der Worte zurueck. Diese Methode wird in der
     * aktuellen implementierung nicht verwendet, wird aber benoetigt,
     * wenn ein alternativer Formatierer die Eingabedaten formatieren
     * soll.
     * */
    public List<String> getWorte() {
      return woerter;
    }

    /**
     * Wandelt die Eingabedaten in einen String um. Die Darstellung
     * genuegt dem in der Aufgabenanalyse beschriebenen Format.
     * */
    public String toString() {
      StringBuilder sb = new StringBuilder();
      //schreibe Kommentare:
      for (String s : kommentare) {
        sb.append(s + "\n");
      }
      //schreibe eingelesene Worte:
      sb.append("Eingelesene Woerter:\n");
      for (int i = 0; i < woerter.size(); i++) {
        sb.append(woerter.get(i));
        if (i != woerter.size() - 1)
          sb.append(", ");
      }
      return sb.toString();
    }
}
```

raetselErsteller/daten/EingabeDaten.java

```
package raetselErsteller.daten;

import java.util.HashMap;
```

```
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import raetselErsteller.Konstanten;

/**
 * Diese Klasse definiert eine aktuelle Kombinationsmoeglichkeit von
 * benutzten Worten. Sie enthaelt eine Menge von noch nicht benutzten
 * Worten, und ein char[] der aktuellen Kombination. Ueber die
   Methoden
 * istFrei und add koennen aus dieser Kombination neue Kombinationen
 * entwickelt werden.
 * */
public class AktuelleKombination
  implements Comparable<AktuelleKombination>{

  // Referenz auf die eingelesenen Eingabedaten
  private EingabeDaten eingabeDaten;

  /*
   * Alle Worte in dieser Liste wurden noch nicht benutzt und koennen
   * noch eingefuegt werden. Wird ein Wort ueber add eingefuegt, so
   * wird es automatisch aus dieser Liste geloescht.
   */
  private List<String> verfuegbareWorte;

  // Aktuelle Kombination der bereits eingefuegten Worte
  private char[][] kombi;

  //speichert die Positionen von Buchstaben
  private Map<Character, Set<Punkt>> map;

  /**
   * Erstellt eine neue Kombination mit einem Startwort. Als Startwort
   * wird das erste Wort in den Eingabedaten verwendet.
   *
   * @param eingabe
   *            Die Eingabedatei, welche unter anderen die Worte
   *            enthaelt, die in einem Objekt dieser Klassen
   *            kombiniert werden sollen.
   *
   * @param horizontal
   *            Gibt an ob das Startwort horizontal oder vertikal
   *            eingefuegt werden soll.
   * */
  public AktuelleKombination(EingabeDaten eingabe,
      boolean horizontal) {
    this.eingabeDaten = eingabe;
    verfuegbareWorte = new LinkedList<>(eingabe.getWorte());
```

```
55      if (Konstanten.useMap)
          map = new HashMap<Character, Set<Punkt>>();
57      // benutze erstes verfuegbares Wort
        String startWort = verfuegbareWorte.remove(0);
59      // fuege das Wort horizontal oder vertikal ein
        if (horizontal) {
61        kombi = new char[1][startWort.length()];
          for (int i = 0; i < kombi[0].length; i++) {
63          kombi[0][i] = startWort.charAt(i);
            addToMap(startWort.charAt(i), new Punkt(0, i));
65        }
        } else {
67        kombi = new char[startWort.length()][1];
          for (int i = 0; i < kombi.length; i++) {
69          kombi[i][0] = startWort.charAt(i);
            addToMap(startWort.charAt(i), new Punkt(i, 0));
71        }
        }
73    }

75    /**
       * Kopierkonstruktor zum Erzeugen einer tiefen Kopie.
77     * */
      public AktuelleKombination(AktuelleKombination kombi2) {
79      this.eingabeDaten = kombi2.eingabeDaten;

81      this.verfuegbareWorte = new LinkedList<>(
            kombi2.verfuegbareWorte);
83      this.kombi = new char[kombi2.kombi.length][kombi2.kombi[0].length
       ];
        for (int i = 0; i < kombi.length; i++) {
85        for (int j = 0; j < kombi[0].length; j++) {
            kombi[i][j] = kombi2.kombi[i][j];
87        }
        }
89      if (Konstanten.useMap) {
          map = new HashMap<>();
91        for (char c : kombi2.map.keySet()) {
            Set<Punkt> mapCpy = new HashSet<>();
93          for (Punkt p : kombi2.map.get(c))
              mapCpy.add(new Punkt(p));
95          map.put(c, mapCpy);
          }
97      }
      }
99

      /**
101    * Default Konstruktor.
       * */
103   public AktuelleKombination(){
        eingabeDaten = null;
105     verfuegbareWorte = new LinkedList<>();
```

```
    kombi = new char[0][0];
    map = new HashMap<>();
  }

  /**
   * Diese Methode prueft ob ein Wort in die aktuelle Kombination
   * eingefuegt werden kann, ohne das es eine Kollision mit einem
   * anderen Buchstaben gibt. Diese Methode prueft nicht, ob die
   * Kombination danach noch gueltig ist(etwa weil das eingefuegte
   * Wort keine Ueberschneidungen mit dem Rest der Kombination
   * aufweist.
   *
   * @param wort
   *            Das Wort, welches in die Kombination eingefuegt werden
   *            soll.
   * @param p
   *            Die Position(basierend auf dem Koordinatensystem der
   *            Kombination) an welcher der Buchstabe wort[index]
   *            eingefuegt werden soll.
   * @param index
   *            An dieser Stelle des Wortes soll das Wort an der
   *            Stelle p in die Kombination eingefuegt werden.
   * @param horizontal
   *            Gibt an ob das Wort horizontal oder vertikal
   *            eingefuegt werden soll.
   * */
  public boolean istFrei(String wort, Punkt p, int index,
      boolean horizontal) {
    if (horizontal)
      return istFreiHorizontal(wort, p, index);
    else
      return istFreiVertikal(wort, p, index);
  }

  /**
   * Ueberprueft, ob das Wort horizontal eingefuegt werden kann.
   * Siehe: istFrei
   */
  private boolean istFreiHorizontal(String wort, Punkt p, int index) {
    for (int i = 0; i < wort.length(); i++) {
      // vor linkem Rand => springe nach vorne
      if (p.y + i - index < 0)
        i = index - p.y;
      // hinter rechtem Rand => keine Kollision mehr moeglich
      if (p.y + i - index >= kombi[0].length)
        return true;
      char zuVergleichen = kombi[p.x][p.y + i - index];
      if (zuVergleichen != '\0' // Position ist noch frei
          && zuVergleichen != wort.charAt(i))
        return false; // Kollision mit anderem Wort
    }
    return true; // Wort kann eingefuegt werden
```

```
  }
159
  /**
161  * Ueberprueft, ob das Wort vertikal eingefuegt werden kann. Siehe:
  * istFrei
163  */
  private boolean istFreiVertikal(String wort, Punkt p, int index) {
165   for (int i = 0; i < wort.length(); i++) {
      // vor oberem Rand => springe nach unten
167   if (p.x + i - index < 0)
        i = index - p.x;
169   // hinter unterem Rand => keine Kollision mehr moeglich
      if (p.x + i - index >= kombi.length)
171     return true;
      char zuVergleichen = kombi[p.x + i - index][p.y];
173   if (zuVergleichen != '\0'// Position ist noch frei
          && zuVergleichen != wort.charAt(i))
175     return false;// Kollision mit anderem Wort
    }
177   return true;// Wort kann eingefuegt werden
  }
179
  /**
181  * Diese Methode fuegt ein neues Wort zur Kombination hinzu.
  * Achtung: vorher sollte ueber istFrei geprueft werden, ob das Wort
183  * eingefuegt werden kann.
  *
185  * @param wort
  *            Das Wort, welches in die Kombination eingefuegt werden
187  *            soll.
  * @param p
189  *            Die Position(basierend auf dem Koordinatensystem der
  *            Kombination) an welcher der Buchstabe wort[index]
191  *            eingefuegt werden soll.
  * @param index
193  *            An dieser Stelle des Wortes soll das Wort an der
  *            Stelle p in die Kombination eingefuegt werden.
195  * @param horizontal
  *            Gibt an ob das Wort horizontal oder vertikal
197  *            eingefuegt werden soll.
  * */
199 public void
      add(String wort, Punkt p, int index, boolean horizontal) {
201   if (horizontal)
      addHorizontal(wort, p, index);
203   else
      addVertikal(wort, p, index);
205   // Wort ist nicht mehr verfuegbar, da es eingefuegt wurde
    verfuegbareWorte.remove(wort);
207 }

209 /**
```

```
211     * Fuegt das angegebene Wort horizontal in die Kombination ein.
        * Siehe: add
        * */
213     private void addHorizontal(String wort, Punkt p, int index) {
          // pruefe ob array erweitert werden muss
215       int linksFehlt = p.y - index < 0 ? index - p.y : 0;
          int rechtsFehlt = p.y - index + wort.length() > kombi[0].length
217           ? p.y - index + wort.length() - kombi[0].length
              : 0;
219       // erweitere das Array
          Punkt offset = new Punkt(0, linksFehlt);
221       erweitereArray(0, linksFehlt + rechtsFehlt, offset);
          // fuege Wort in Array ein
223       for (int i = 0; i < wort.length(); i++) {
            kombi[p.x][p.y + linksFehlt + i - index] = wort.charAt(i);
225         // add punkt zur map
            addToMap(wort.charAt(i), new Punkt(p.x, p.y + linksFehlt
227               + i - index));
          }
229     }

231     /**
        * Fuegt das angegebene Wort vertikal in die Kombination ein. Siehe:
233     * add
        * */
235     private void addVertikal(String wort, Punkt p, int index) {
          // pruefe ob array erweitert werden muss
237       int obenFehlt = p.x - index < 0 ? index - p.x : 0;
          int untenFehlt = p.x - index + wort.length() > kombi.length
239           ? p.x - index + wort.length() - kombi.length
              : 0;
241       // erweitere das Array
          Punkt offset = new Punkt(obenFehlt, 0);
243       erweitereArray(obenFehlt + untenFehlt, 0, offset);
          for (int i = 0; i < wort.length(); i++) {
245         kombi[p.x + i - index + obenFehlt][p.y] = wort.charAt(i);
            // add punkt zur map
247         addToMap(wort.charAt(i), new Punkt(p.x + i - index
                + obenFehlt, p.y));
249       }
        }
251
        /**
253     * Erweitert das Array in x und/oder y Richtung und kopiert die
        * Daten des alten Arrays in das neue.
255     *
        * @param x
257     *            Anzahl Elemente, welche in x-Richtung erweitert werden
        *            muessen.
259     * @param y
        *            Anzahl Elemente, welche in x-Richtung erweitert werden
261     *            muessen.
```

```
     *
263  * @paran p Position im neuen Koordinatensystem, an der das alte
     *          Array eingefuegt werden soll (obere linke Ecke des alten
265  *          Arrays).
     * */
267  private void erweitereArray(int x, int y, Punkt p) {
       if (x <= 0 && y <= 0)
269      return;
       char[][] neu = new char[kombi.length + x][kombi[0].length + y];
271    for (int i = 0; i < kombi.length; i++) {
         for (int j = 0; j < kombi[0].length; j++) {
273        neu[i + p.x][j + p.y] = kombi[i][j];
         }
275    }
       kombi = neu;
277    if(!Konstanten.useMap)return;
       //map veraendern
279    for(char c: map.keySet()){
         Set<Punkt> newSet = new HashSet<>();
281      for(Punkt p2 : map.get(c)){
           newSet.add(new Punkt(p2.x+p.x, p2.y+p.y));
283      }
         map.put(c, newSet);
285    }
     }
287
     /**
289   * Gibt zurueck, ob die aktuelle Kombination zu einer vollstaendigen
      * (wenn auch nicht optimalen) Loesung geworden ist.
291   * */
     public boolean istFertig() {
293    return verfuegbareWorte.size() == 0;
     }
295
     /**
297   * Vergleicht die AktuelleKombination mit einem anderen Objekt
      * der selben Klasse. Vergleichskriterium ist die Bewertung.
299   * Diese ist definiert als Flaecheninhalt des kombi-Feldes.
      *
301   * @return 1 wenn other<this, -1 wenn other>this, 0 sonst.
      * */
303  @Override
     public int compareTo(AktuelleKombination o) {
305    if(o instanceof SchlechtesteAktuelleKombination) return 1;
       int bewertungThis = kombi.length*kombi[0].length;
307    int bewertungOther = o.kombi.length*o.kombi[0].length;
       if(bewertungThis<bewertungOther) return 1;
309    if(bewertungThis>bewertungOther) return -1;
       return 0;
311  }

313  /**
```

```
     * Gibt alle Punkte zurueck, an denen in dieser Kombination ein
315  * uebergebenes Zeichen steht.
     *
317  * @param c
     *            Das Zeichen nach dem gesucht werden soll.
319  * */
    public Set<Punkt> getAllePunkteZu(char c) {
321   if (Konstanten.useMap) {
        if (map.get(c) == null)
323       return new HashSet<>();
        return map.get(c);
325   } else {
        Set<Punkt> result = new HashSet<>();
327     for (int i = 0; i < kombi.length; i++) {
          for (int j = 0; j < kombi[0].length; j++) {
329         if(kombi[i][j]==c)
              result.add(new Punkt(i,j));
331       }
        }
333     return result;
      }
335 }

337 /**
     * Liefert alle Worte zurueck, die noch nicht in dieser Kombination
339  * benutzt wurden.
     * */
341 public List<String> getVerfuegbareWoerter() {
      return verfuegbareWorte;
343 }

345 /**
     * Liefert die in der aktuellen Kombination verwendeten Buchstaben
347  * als Zeichenarray zurueck.
     *
349  * @return char[][], welches die Kombination repraesentiert.
     * */
351 public char[][] getKombi() {
      return kombi;
353 }

355 /**
     * Wandelt die aktuelle Kombination in einen String um. Dargestellt
357  * wird zuerst die Anordnung der Buchstaben in Rasterform und dann
     * die Liste der noch verfuegbaren Worte.
359  *
     * @return Repraesentation der Kombination als String.
361  * */
    public String toString() {
363   StringBuilder sb = new StringBuilder();
      // char[][] als Raster darstekken
365   for (int i = 0; i < kombi.length; i++) {
```

```
      for (int j = 0; j < kombi[0].length; j++) {
        char c = kombi[i][j];
        sb.append("|");
        if (c == '\0')
          sb.append("_");
        else
          sb.append(c);
      }
      sb.append("|\n");
    }
    // noch nicht verwendete Worte
    sb.append("Worte:\n");
    for (String s : verfuegbareWorte)
      sb.append(s + "\n");
    return sb.toString();
  }

  /**
   * Fuegt einen neuen Buchstaben zur Map hinzu
   * */
  private void addToMap(char c, Punkt p) {
    if (!Konstanten.useMap)
      return;
    if (map.get(c) == null)
      map.put(c, new HashSet<Punkt>());
    Set<Punkt> s = map.get(c);
    s.add(p);
  }

  /**
   * Erstellt einen Hash-Code zu der aktuellen Kombination.
   **/
  @Override
  public int hashCode() {
    return kombi[0].length * kombi.length
        + verfuegbareWorte.hashCode();
  }

  /**
   * Vergleicht, ob ein objekt dieser Klasse und ein beliebiges
   * anderes Objekt gleich sind. Handelt es sich dabei um ein Objekt
   * der Klasse AktuelleKombination, so gelten sie als gleich, wenn
   * die bereits eingefuegte Kombination und die noch verfuegbaren
   * Worte gleich sind.
   * */
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
```

```
      return false;
    AktuelleKombination other = (AktuelleKombination) obj;
    // pruefe dimensionen
    if (other.kombi.length != kombi.length)
      return false;
    if (other.kombi[0].length != kombi[0].length)
      return false;
    // pruefe Worte
      if (!verfuegbareWorte.equals(other.verfuegbareWorte))
        return false;
    // pruefe felder
    for (int i = 0; i < kombi.length; i++) {
      for (int j = 0; j < kombi[0].length; j++) {
        if (kombi[i][j] != other.kombi[i][j])
          return false;
      }
    }

    return true;
  }

}
```

raetselErsteller/daten/AktuelleKombination.java

```
package raetselErsteller.daten;

/**
 * DIes ist eine aktuelle Kombination, welche immer schlechter als
 * eine andere aktuelle Kombination ist. Sie wird verwendet um
 * die aktuell beste Kombination im Loesungsalgorithmus zu
 * initialisieren.
 * @author Felix Kibellus
 * */
public class SchlechtesteAktuelleKombination
  extends AktuelleKombination{

  @Override
  public int compareTo(AktuelleKombination o) {
    return -1;
  }
}
```

raetselErsteller/daten/SchlechtesteAktuelleKombination.java

```
package raetselErsteller.daten;

import raetselErsteller.Konstanten;

/**
 * Diese Klasse dient zur Repraesentation der Ausgabedaten. Ein Objekt
 * dieser Klasse enthaelt eine optimale Kombination der Worte in der
```

```
 * Eingabedatei. Diese Kombination wird in dieser Klasse einmal mit
   und
 * einmal ohne zusaetzliche Auffuellzeichen gespeichert. Darueber
 * hinaus haelt diese Klasse eine Referenz auf die Eingabedaten aus
 * welchen die Loesung erstellt wurde.
 *
 * @author Felix Kibellus
 * */
public class AusgabeDaten {

  // Eingabedaten, welche zu diesen Ausgabedaten gehoeren
  private EingabeDaten eingabeDaten;
  private char[][] loesungEinfach; // ohne zusaetzliche zeichen
  private char[][] loesungSchwer; // mit zusaetzlichen zeichen

  /**
   * Erzeugt ein Objekt der Klasse AusgabeDaten.
   *
   * @param eingabeDaten
   *            die zu den Ausgabedaten gehoerenden Eingabedaten
   * @param c1
   *            die Loesung ohne Auffuellzeichen
   * @param c2
   *            die Loesung mit Auffuellzeichen
   * */
  public AusgabeDaten(EingabeDaten eingabeDaten,
      char[][] c1, char[][] c2) {
    this.eingabeDaten = eingabeDaten;
    this.loesungEinfach = c1;
    this.loesungSchwer = c2;
  }

  /**
   * Wandelt die Ausgabedaten in Textform um. Die Darstellung genuegt
   * dem in der Aufgabenanalyse definierten Format, und kann vom
   * Formatierer direkt benutzt werden. Die Ausgabe der Ausgabedaten
   * schiesst die Ausgabe der Eingabedaten mit ein. Deshalb wird in
   * dieser Methode an die toString-Methode der Klasse EingabeDaten
   * deligiert.
   * */
  public String toString() {
    StringBuilder sb = new StringBuilder();
    // Eingabedaten
    sb.append(eingabeDaten.toString() + "\n");
    sb.append("\n");

    // Raetsel nicht verseteckt
    sb.append("** "+Konstanten.raetselNichtVersteckt+"\n");
    sb.append("\n");
    for (int i = 0; i < loesungEinfach.length; i++) {
      for (int j = 0; j < loesungEinfach[0].length; j++) {
        if (loesungEinfach[i][j] == '\0')
```

```
59            sb.append(" ");
            else
61            sb.append(loesungEinfach[i][j]);
        }
63      sb.append("\n");
    }
65
    // Raetsel versteckt
67  sb.append("\n");
    sb.append("** "+Konstanten.raetselVersteckt+"\n");
69  sb.append("\n");
    for (int i = 0; i < loesungSchwer.length; i++) {
71    for (int j = 0; j < loesungSchwer[0].length; j++) {
        sb.append(loesungSchwer[i][j]);
73    }
      sb.append("\n");
75  }

77  // Kompaktheitsmass
    sb.append("\n");
79  int mass = loesungSchwer.length * loesungSchwer[0].length;
    sb.append("Kompaktheitsmass: " + mass);
81
    return sb.toString();
83  }
}
```

raetselErsteller/daten/AusgabeDaten.java

E.1.3. io

```
package raetselErsteller.io;
2
import raetselErsteller.daten.EingabeDaten;
4import raetselErsteller.io.format.FormatException;

6/**
 * Dieses Interface definiert die Schnittstelle zum Lesen der
8 * Eingabedaten. Durch die Abstraktion dieser Schnittstelle muss der
 * Controller nicht wesentlich veraendert werden, um die
10 * Eingabefunktionalitaet zu aendern.
 *
12 * @author Felix Kibellus
 * */
14public interface EingabeDatenLeser {

16  /**
    * Liesst die Eingabedaten ein. Wie die Eingabe genau funktioniert
18  * (Datei, Konsole, Netzwerk,...) muss durch eine konkrete Klasse
    * entschieden werden, welche dieses Interface implementiert.
20  *
```

```
    * @return Die Eingabedaten, welche die Worte enthalten, die zu
    *         einem fertigen Raetsel kombiniert werden sollen
    * */
  EingabeDaten leseEingabe() throws FileIOException, FormatException;
}
```

raetselErsteller/io/EingabeDatenLeser.java

```
package raetselErsteller.io;

import raetselErsteller.daten.AusgabeDaten;

/**
 * Dieses Interface definiert die Schnittstelle zum Schreiben der
 * Ausgabedaten. Durch die Abstraktion dieser Schnittstelle muss der
 * Controller nicht wesentlich veraendert werden, um die
 * Ausgabefunktionalitaet zu aendern.
 *
 * @author Felix Kibellus
 * */
public interface AusgabeDatenSchreiber {

  /**
   * Gibt die Ausgabedaten aus. Wie die Ausgabe genau funktioniert
   * (Datei, Konsole, Netzwerk,...) muss durch eine konkrete Klasse
   * entschieden werden, welche dieses Interface implementiert.
   *
   * @param outputData
   *            Die Ausgabedaten, welche die fertigen Raetsel
   *            beinhalten.
   * */
  void schreibeAusgabe(AusgabeDaten out)
      throws FileIOException;
}
```

raetselErsteller/io/AusgabeDatenSchreiber.java

```
package raetselErsteller.io;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

import raetselErsteller.daten.EingabeDaten;
import raetselErsteller.daten.AusgabeDaten;
import raetselErsteller.io.FileIOException.ErrorType;
import raetselErsteller.io.format.Formatierer;
```

```
import raetselErsteller.io.format.FormatException;
18
/**
20  * Diese Klasse kann zum Lesen und Schreiben von Dateien verwendet
    * werden. Sie liest die Eingabedaten aus einer Eingabedatei und
22  * schreibt die Ausgabedaten in eine Ausgabedatei. Das Format von Ein-
    * und Ausgabedatei kann ueber einen Formatierer angepasst werden.
24  *
    * @author Felix Kibellus
26  * */
public class FileIO implements EingabeDatenLeser,
    AusgabeDatenSchreiber {
28
  public static Logger logger = Logger.getLogger(FileIO.class
30      .getName());

32  // pfad zur Eingabedatei
  private String inPfad;
34  // pfad zur Ausgabedatei
  private String outPfad;
36  // Formatierer fuer ein und Ausgabe
  private Formatierer formatierer;
38
  /**
40   * Erstellt ein Objekt der Klasse FileIO.
     *
42   * @param inPfad
     *            der Pfad zur Eingabedatei
44   * @param outPfad
     *            der Pfad zur Ausgabedatei
46   * @param formatierer
     *            ein Formatierer, der regelt wie Ein- und Ausgabe
48   *            formatiert werden soll
     * */
50  public FileIO(String inPfad, String outPfad,
      Formatierer formatierer) {
52    logger.log(Level.FINER, "starte Methode");

54    this.inPfad = inPfad;
    this.outPfad = outPfad;
56    this.formatierer = formatierer;
  }
58
  /**
60   * Schreibt die uebergebenen Ausgabedaten in eine Datei
     *
62   * @param out
     *            Ausgabedaten, welche in eine Datei geschrieben werden
64   *            sollen
     * */
66  @Override
  public void schreibeAusgabe(AusgabeDaten out)
```

```
68      throws FileIOException {
    logger.log(Level.FINER, "starte Methode");
70
    String outputString = formatierer.formatiere(out);
72  schreibeDatei(outputString);
  }
74
  /**
76 * Liest die Eingabedaten aus einer Datei und gibt diese zurueck.
   * Dazu werden zuerst die Zeilen aus der Datei eingelesen, und dann
78 * vom Formatierer in eine Eingabedatei umgewandelt.
   *
80 * @return Aus der Datei gelesene Eingabedaten
   * */
82 @Override
  public EingabeDaten leseEingabe() throws FileIOException,
84     FormatException {
    logger.log(Level.FINER, "starte Methode");
86
    List<String> inputStrings = leseDatei();
88  EingabeDaten inputData = formatierer.parse(inputStrings);
    return inputData;
90 }

92 /**
   * Schreibt einen String in die im Konstruktor angegebene Datei.
94 *
   * @param outString
96 *            Text, der in die Ausgabedatei geschrieben werden soll
   * */
98 private void schreibeDatei(String outString)
     throws FileIOException {
100   logger.log(Level.FINER, "starte Methode");

102   File file = new File(outPfad);
    try (FileWriter fw = new FileWriter(file)) {
104   fw.write(outString);
    } catch (IOException e) {
106   throw new FileIOException(ErrorType.OutputPathInvalid);
    }
108 }

110 /**
   * Liest die im Konstruktor angegebene Datei zeilenweilse aus.
112 *
   * @return Liste aus den eingelesenen Zeilen
114 * */
  private List<String> leseDatei() throws FileIOException {
116   logger.log(Level.FINER, "starte Methode");

118   File file = new File(inPfad);
    List<String> inputStrings = new LinkedList<>();
```

```
120     try (Scanner sc = new Scanner(file)) {
          while (sc.hasNext()) {
122         String line = sc.nextLine();
            inputStrings.add(line);
124       }
        } catch (FileNotFoundException e) {
126       throw new FileIOException(ErrorType.InputFileNotFound);
        }
128     return inputStrings;
      }
130
}
```

raetselErsteller/io/FileIO.java

```
1  package raetselErsteller.io;

3  /**
    * Dieses Interface definiert die Schnittstelle zum Ausgeben von
5   * Fehlern. Wenn ein Fehler auftritt, leitet der Controller die
    * zugehoerige Fehlernachricht an diese Schnittstelle weiter. Eine
7   * Klasse, welche dieses Interface implementiert muss die Nachricht
    * dann geeignet verarbeiten. (StandardErr, Datei, ...)
9   *
    * @author Felix Kibellus
11  * */
public interface ErrorHandler {
13
  /**
15 * Diese Methode stellt die Schnittstelle zur Fehlerausgabe dar.
   *
17 * @param msg
   *            Eine Nachricht, welche den aufgetretenden Fehler
19 *            beschreibt.
   * */
21 void zeigeFehler(String fehler);
}
```

raetselErsteller/io/ErrorHandler.java

```
package raetselErsteller.io;
2
/**
4  * Diese Klasse kann zur Fehlerausgabe benutzt werden. Sie schreibt
   * auftretende Fehler auf StandardErr, also auf die Fehleraushabe.
6  *
   * @author Felix Kibellus
8  * */
public class ConsoleErrorHandler implements ErrorHandler {
10
  /**
12 * Schreibt einen Fehler auf die Fehleraushabe.
   *
```

```
14     * @param msg
       *             Die Fehlernachricht, welche ausgegben werden soll.
16     * */
     @Override
18   public void zeigeFehler(String msg) {
       System.err.println(msg);
20   }

22 }
```

raetselErsteller/io/ConsoleErrorHandler.java

```
  package raetselErsteller.io;
2
  import java.io.IOException;
4
  /**
6  * Diese Klasse wird verwendet um Aussnahmen beim Lesen und Schreiben
   * von Dateien zu behandeln. Sie ist in der Lage, die beiden Faelle
8  * Eingabedatei existiert nicht und Ausgabepfad ist ungueltig zu
   * repraesentieren. Ueber die Methode getType() kann abgefragt werden,
10 * welcher von beiden Faellen eingetreten ist.
   *
12 * @author Felix Kibellus
   * */
14 public class FileIOException extends IOException {

16   private static final long serialVersionUID = 1L;

18   // Speichert welcher Fehlertyp genau eingetreten ist
     private ErrorType type;
20
     /**
22    * Erzeugt ein Objekt vom Typ FileIOException
      *
24    * @param type
      *             Hier muss angegeben werden, ob die Eingabedatei nicht
26    *             existiert oder ob der Pfad zur Ausgabedatei ungueltig
      *             ist
28    * */
     public FileIOException(ErrorType type) {
30     this.type = type;
     }
32
     /**
34    * Liefert zurueck, welcher Fehlerfall bei der Ein- oder Ausgabe
      * aufgetreten ist.
36    * */
     public ErrorType getType() {
38     return type;
     }
40
     //Enum, welches die verschiedenen Ein- und Ausgabefaelle definiert
```

```
42    public enum ErrorType {
        InputFileNotFound, OutputPathInvalid
44    }
}
```

raetselErsteller/io/FileIOException.java

format

```
1  package raetselErsteller.io.format;

3  import java.util.List;

5  import raetselErsteller.daten.AusgabeDaten;
   import raetselErsteller.daten.EingabeDaten;
7
   /**
9   * Diese Schnittstelle dient der Abstraktion des Formates von
    * Ein- und Ausgabedatei.
11  * @author Felix Kibellus
    * */
13 public interface Formatierer {

15   /**
      * Formatiert die Ausgabedaten zu einem Text, der in die
17    * Ausgabedatei geschrieben werden kann.
      * */
19   String formatiere(AusgabeDaten out);

21   /**
      * Erzeugt aus den aus der Eingabedatei eingelesenen Zeilen
23    * ein Objekt vom Typ Eingabedaten.
      * */
25   EingabeDaten parse(List<String> in) throws FormatException;
}
```

raetselErsteller/io/format/Formatierer.java

```
   package raetselErsteller.io.format;
2
   import java.util.LinkedList;
4  import java.util.List;
   import java.util.logging.Level;
6  import java.util.logging.Logger;

8  import raetselErsteller.daten.AusgabeDaten;
   import raetselErsteller.daten.EingabeDaten;
10 import raetselErsteller.io.format.FormatException.GrundType;

12 /**
    * Diese Klasse kann zum formatieren von Ausgabedaten und zum parsen
```

```
14  * von Eingabedaten verwendet werden. Das verwendete Format genuegt
        den
    * in der Aufgabenanalyse definierten Kriterien.
16  *
    * @author Felix Kibellus
18  * */
    public class StdFormat implements Formatierer {
20
      //zum Loggen benoetigt
22    public static Logger logger = Logger.getLogger(StdFormat.class
          .getName());
24
      /**
26     * Formatiert die Ausgabedaten. Dazu wird das in der
       * toString-Methode definierte format gewaehlt.
28     * @param data Die Ausgabedaten, welche formatiert werden sollen.
       * */
30    @Override
      public String formatiere(AusgabeDaten data) {
32      logger.log(Level.FINER, "starte Methode");

34      return data.toString();
      }
36
      /**
38     * Diese Methode erzeugt aus den aus der Eingabedatei gelesenen
       * Zeilen ein Objekt der Klasse EingabeDaten. Dabei werden die
40     * Kommentare gespeichert, sodass diese spaeter ausgegeben werden
       * koennen. Ausserdem werden Fehler in den Eingabedaten erkannt.
42     * Beispielsweise koennen weniger als 2 Worte angegeben worden sein,
       * oder ein Wort nicht ausschliesslich aus Buchstaben bestehen.
44     * */
      @Override
46    public EingabeDaten parse(List<String> inputString)
          throws FormatException {
48      // logger.log(Level.FINER, "starte Methode");

50      // lese kommentare
        List<String> kommentare = new LinkedList<>();
52      int i;
        for (i = 0; i < inputString.size(); i++) {
54        String line = inputString.get(i);
          if (line.startsWith(";"))
56          kommentare.add(line);
          else
58          break; // nun kommen die worte
        }
60      List<String> worte = new LinkedList<>();
        for (int j = i; j < inputString.size(); j++) {
62        String line = inputString.get(j);
          line = line.trim();
64        if(line.equals(""))
```

```
            throw new FormatException(j+1, GrundType.Leerzeile);
66        if (!istKorrekt(line))
            throw new FormatException(j+1, GrundType.KeinBuchstabe);
68        worte.add(line.toUpperCase());
      }
70      if (worte.size() <= 1) {
        throw new FormatException(i+1, GrundType.NichtGenugWorte);
72      }
      return new EingabeDaten(kommentare, worte);
74    }

76    /**
       * Diese Funktion prueft, ob es sich bei der uebergebenen Zeile
78     * ausschliesslich um Buchstaben handelt.
       * */
80    private boolean istKorrekt(String line) {
        for (int i = 0; i < line.length(); i++) {
82        if (!Character.isLetter(line.charAt(i)))
            return false;
84      }
        return true;
86    }

88  }
```

raetselErsteller/io/format/StdFormat.java

```
  package raetselErsteller.io.format;
2
  /**
4  * Diese Klasse wird zur Repraesentation von Aussnahmen verwendet,
   * welche beim Parsen der Eingabedaten entstehen koennen.
6  * Ueber getLine() kann abgefragt werden, in welcher Zeile der Fehler
   * aufgetreten ist. Ueber getType() kann abgefragt werden, welcher
8  * Fehler genau aufgetreten ist.
   * @author Felix Kibellus
10 * */
  public class FormatException extends Exception{
12
    private static final long serialVersionUID = 1L;
14  private int line;//Zeile, in der der Fehler aufgetreten ist
    private GrundType type; //genauer Grund des Fehlers
16
    /**
18   * Erzeugt ein Objekt der Klasse FormatException
     * @param line Zeile in der der Fehler aufgetreten ist
20   * @param type genauer Grund fuer den Fehlers.
     * */
22  public FormatException(int line, GrundType type){
      this.line = line;
24    this.type = type;
    }
26
```

```
  /**
28  * Gibt die Zeile an, in der der Fehler aufgetreten ist.
   * */
30 public int getLine(){
    return line;
32 }

34 /**
   * Gibt den genauen Fehlergrund an.
36 * */
   public GrundType getType() {
38   return type;
   }
40

   //Dient zur Unterscheidung, welcher Fehler genau aufgetreten ist
42 public enum GrundType{
     //eingelesenes Wort bestand nicht nur aus Buchstaben
44   KeinBuchstabe,
     //es wurden weniger als 2 Worte eingelesen
46   NichtGenugWorte,
     //in der Eingabedatei befand sich eine Leerzeile
48   Leerzeile
   }
50 }
```

raetselErsteller/io/format/FormatException.java

E.2. Konstanten

```
package raetselErsteller;
2

/**
4 * In dieser Klasse sind Konstanten definiert. Diese lassen sich in 4
  * Kategorien unterteilen: -Programmeinstellungen -Hiletext fuer den
6 * Argparser -Fehlermeldungen -Ausgabe Die Programmeinstellungen
  * koennen geaendert werden, um das Verhalten des Programms zu
8 * beeinflussen. Sollte das Programm in einer anderen Sprache
    benuetigt
  * werden, genuegt es die Texte in dieser Datei auszutauschen.
10 *
  * @author Felix Kibellus
12 * */
public class Konstanten {
14

  // ##################################################################
16 // ################## Programmeinstellungen #######################
  // ##################################################################
18

  public static final boolean useMap = false;
20
```

```
/*
 * Hier koennen die gewuenschten Dateiendungen fuer ein und
 * Ausgabedatei angegeben werden.
 */
public static final String dateiendungIn = ".in";
public static final String dateiendungOut = ".out";

// ##############################################################
// ################## Hilfe fuer Argparser ######################
// ##############################################################

// So wird der Platzhalter fuer die Eingabedatei in der
// Hilfeansicht
// des Argparsers dargestellt
public static final String inputData = "EINGABEDATEI";

// So wird in der Hilfeansicht des Argparsers der
// Programmaufruf erklaert
public static final String projectname = "java -jar"
    + " raetselErsteller.jar";

// Kurze Beschreibung der Programmfunktionalitaet.
public static final String beschreibung = "Dieses Programm kann "
    + "zum Erstellen von Raetseln genutzt werden.";

// Erklaerung der Kommandozeilen-Option -l bzw --log
public static final String log = " ber diesen Parameter kann das "
    + "Logging aktiviert werden. Moegliche Parameter sind ALL, "
    + "CONFIG, FINE, FINER, FINEST, INFO, OFF, SEVERE und "
    + "WARNING. Default ist OFF. FINER zeigt alle "
    + "Methodenaufrufe an, mit FINEST kann die Funktionalit t "
    + "des Algorithmus nachvollzogen werde,";

// Erklaerung der Kommandozeilenoption zum Angeben
// einer Ausgabedatei
public static final String output = "Dieser Parameter legt den "
    + "Pfad fuer die Ausgabedatei fest. Default ist Pfad und "
    + "Name der Eingabedatei. Endet die Eingabedatei auf .in"
    + " wird .in durch .out ersetzt. Andernfalls wird nur .out "
    + "an den Namen der " + "Eingabedatei angehaengt.";

// Beschreibung zur Angabe der Eingabedatei
public static final String input = "Hier muss der Pfad zur "
    + "Eingabedatei angegeben werden. Hier gibt es keinen "
    + "Default-Wert. Die Eingabedatei muss angegeben werden.";

// Erklaerung zum Kommandozeilen-Option -t bzw. --timestamp
public static final String timeStamp = "Gibt die benoetigte "
    + "Laufzeit des Algorithmus aus. Die zeit zum Einlesen und "
    + "Ausgeben ist nicht " + "mit eingeschlossen";

// Beschreibung der Kommandozeilen-Option -a bzw. --algorithm
```

```
  public static final String algo = "Legt fest welcher Algoritmus "
74      + "verwendet werden soll. Zur Auswahl stehen Tiefensuche "
        + "und Breitensuche. Es wird empfohlen Tiefensuche zu "
76      + "benutzen.";

78  // ##############################################################
    // ################### Fehlermeldungen ##########################
80  // ##############################################################

82  //Diese Fehlermeldung wird ausgegeben, wenn bei der Berechnung
    //mehr Speicher benoetigt wird als der Java VM zugeteilt ist,
84  public static String keinSpeicher = "Es steht nicht genug Speicher"
        + " zur Verfuegung. Das Programm wird abgebrochen.";
86
    public static String leerzeile = "In der Eingabedatei befindet" +
88      " sich eine Leerzeile.";

90  // Diese Fehlermeldung wird ausgegeben, wenn die angegebene
    // Eingabedatei nicht gefunden wurde.
92  public static final String inputFileNotFound = "Die Eingabedatei "
        + "konnte nicht gefunden werden. Das Programm wird "
94      + "abgebrochen.";

96  // Diese Fehlermeldung wird ausgegeben, wenn in der Eingabedatei
    // ein ungueltiges Wort gefunden wurde und der Formatierer eine
98  // Fehlermeldung in einer bestimmten Zeile produziert. Ueber den
    // Parameter line kann die betroffene Zeile direkt in die
100 // Fehlermeldung integriert werden.
    public static String buchstabeInZeile(int line) {
102   return "Eingabedatei ist falsch formatiert. Ein eingegebenes "
          + "Wort besteht nicht ausschliesslich aus Buchstaben. "
104       + "Der Fehler " + "befindet sich in der Naehe von "
          + "Zeile " + line + ".";
106 }

108 // Diese Fehlermeldung wird ausgegeben, wenn weniger als zwei worte
    // in der Eingabedatei enthalten sind.
110 public static final String nichtGenugWorte = "Die Eingabedatei "
        + "muss mindestens 2 Woerter enthalten, damit das Problem "
112     + "sinnvoll zu bearbeiten ist.";

114 // Sollte ein Pfad fuer die Ausgabedatei angegeben worden sein,
    // dieser aber ungueltig ist, so wird diese Fehlermeldung
116 // ausgegeben.
    public static final String outputPathInvalid = "Es liegt ein "
118     + "Problem mit " + "dem Pfad der Ausgabedatei vor.";

120 // Wenn beim Lesen oder Schreiben von Dateien ein nicht naeher
    // definierbarer Fehler auftritt, wird dieser Fehler ausgegeben.
122 public static final String unknownIoErr = "Unbekannter Fehler beim "
        + "Lesen oder Schreiben von Dateien.";
124
```

```
    // Wenn beim beim Formatieren der Daten ein nicht naeher
126 // definierbarer Fehler auftritt, wird dieser Fehler ausgegeben.
    public static final String unknownFormatErr = "Unbekannter Fehler"
128     + " formatieren der Ein- oder Ausgabe.";

130 // Sollte keine Loesung gefunden werden, etwa weil die
    // Buchstabenmengen der Worte disjunkt sind, wird diese
132 // Fehlermeldung angezeigt.
    public static final String keineLoesung = "Zu den eingegebenen "
134     + "Worten " + "kann kein Raetsel gebildet werden.";

136 // ##############################################################
    // ######################## Ausgabe #############################
138 // ##############################################################

140 // Diese Anzeige wird verwendet, wenn der Benutzer ueber -t eine
    // Zeitmessung verlangt hat.
142 public static final String laufzeit = "Laufzeit: ";

144 // Wird beim Ausgeben der Ausgabedatei verwendet, um das versteckte
    // Raetsel zu beschreiben
146 public static final String raetselVersteckt = "Raetsel versteckt";

148 // Wird beim Ausgeben der Ausgabedatei verwendet, um das noch nicht
    // versteckte Raetsel zu beschreiben
150 public static final String raetselNichtVersteckt = "Raetsel "
        + "nicht versteckt";
152 }
```

raetselErsteller/Konstanten.java