Aufgabe 1: Störung

Inhaltsverzeichnis

Begriffsklärungen:

- Suchtext = Die zu durchsuchende Textdatei
- Lückentext = Lückentext-Datei

Wichtig: Zum Ausführen der JAR-Datei „Aufgabe1_GUI“ ist die Installation des neuesten JDK (Java Development Kit) erforderlich, bzw. mindestens Version 19.0.1:

https://www.oracle.com/de/java/technologies/downloads/#java19

Lösungsidee

Zu Beginn treffe ich die Annahme, dass die Lücken der Lückentexte nur mit Wörtern ausgefüllt werden können, wie auch in der Aufgabenstellung impliziert, und nicht mit Satzzeichen. Auch Ziffern sehe ich auf dieselbe Weise wie Buchstaben als Wortbestandteile an.

Weil der Buchtext erst einmal nur eine zusammenhängende Datei ist, ist es sinnvoll, ihn in viele kleine Einheiten aufzuteilen. Jede Einheit ist entweder ein Wort, eine Zahl (auch mehrstellig), ein Satzzeichen oder eine Reihe von gleichen Satzzeichen. Ein Beispiel für Letzteres sind die doppelten Bindestriche „--“ in der Geschichte. Diese sollten im Lückentext nicht mit einem Leerzeichen separiert sein, weil sie eine zusammenhängende Einheit bilden.

Wenn im Suchtext direkt vor einem Zeilenumbruch ein Bindestrich steht und die vorangehende Einheit ein Wort oder eine Zahl ist, wird sie mit dem Bindestrich und der Einheit nach dem Zeilenumbruch als eine Einheit betrachtet.

Die Einheiten müssen im Lückentext, wie in der Aufgabenstellung erläutert, mit Leerzeichen voneinander getrennt, aber nicht unbedingt klein geschrieben sein, da sie sowieso passend formatiert werden können und es eine Option geben soll, die Groß-Klein-Schreibung zu beachten.

Die einzigen Zeichen, die weder Buchstabe noch Ziffer sind und **nicht** von Buchstaben und Ziffern im Lückentext mit einem Leerzeichen getrennt sein müssen, sind:

- Der Apostroph (‘). Das liegt daran, dass es normalerweise zu einem Wort dazugehört, wie z.B. zum Wort „Alice‘s“ ganz am Anfang der Geschichte.
- Der Bindestrich (-) aus demselben Grund. Ein Beispiel ist das Wort „Eisenbahn-Station“ in Z. 480 der Buchtextdatei.

Nachdem der Suchtext und auch der Lückentext nach diesen Regeln in ihre Einheiten zerlegt worden sind, werden die Einheiten des Suchtextes durchgegangen. Von den bisher gespeicherten, unfertigen Vervollständigungen wird die nächste unausgefüllte Stelle mit der aktuellen Einheit verglichen und wenn diese passt, wird sie zur jeweiligen Vervollständigung hinzugefügt, ansonsten wird die Vervollständigung gelöscht.

Falls alle Plätze einer Vervollständigung belegt sind, wird sie zu den fertigen Vervollständigungen hinzugefügt und aus den unfertigen gelöscht.

Wenn die aktuelle Suchtexteinheit zur ersten Einheit des Lückentextes passt, also entweder dieselbe ist oder der Lückentext an erster Stelle eine Lücke hat, wird eine neue, unfertige Vervollständigung mit dieser Einheit am Anfang erstellt.

Auf diese Weise werden im Verlauf des Textes alle möglichen Vervollständigungen geprüft und, wenn sie passen, bis zum Ende gespeichert.

Umsetzung

Ich habe meine Lösungsidee in Java, Version 19.0.1, implementiert und die Community Edition der IDE „IntelliJ IDEA“ von JetBrains verwendet.

Die GUI-Anwendung (Graphische Benutzeroberfläche) habe ich mit JavaFX entwickelt, wobei ich zur Gestaltung der Oberfläche eine FXML-Datei erstellt und durch ein weiteres Programm (SceneBuilder) indirekt bearbeitet habe. Zum Erstellen der Themes (Darkmode und Lightmode) habe ich CSS verwendet.

Zur eigentlichen Erfüllung der gestellten Aufgabe dienen drei Klassen:

- Die Klasse ***BaseText***, die den Suchtext einliest und in seine Einheiten aufteilt.
- Die Klasse ***GapText***, die eine Lückentext-Datei einliest, aufteilt und nach Vervollständigungen in den Daten des Suchtextes sucht.
- Die Klasse ***Completion***, welche eine (evtl. unfertige) Vervollständigung eines Lückentextes modelliert und von ***GapText*** in mehreren ***ArrayList***-Objekten nach dem Prinzip der Klassenaggregation instanziiert wird.

Die Klasse „BaseText“:

Das sind die Attribute und erklärungsbedürftigen Methoden der Klasse ***BaseText***.

Attribute:

BaseText hat 5 Attribute. Es gibt eine ArrayList ***textUnits***, in der die Texteinheiten gespeichert werden und parallel dazu eine ArrayList ***unitStartEndIndices*** aus Integer-Arrays, in denen später jeweils der Start-Zeichen-Index und der End-Index einer Einheit gespeichert werden. In ***textString*** wird der Text als String gespeichert.

In der ArrayList ***newLineIndices*** werden die Zeichen-Indizes angegeben, an denen eine neue Zeile beginnt und ***nLines*** gibt die Anzahl der Zeilen an.

Konstruktoren:

```
public BaseText(String text) throws IOException {
public BaseText(File file) throws IOException {
```

Diese Klasse hat zwei verschiedene eigenständige Konstruktoren, die sich beide sehr ähneln. Beim einen wird ein String übergeben, der den Suchtext enthält, beim anderen ein ***File***-Objekt aus dem ***java.io***-package. Nach dem Initialisieren der Attribute wird mithilfe der Methode ***assembleTextUnitsFromString()*** bzw. ***assembleTextUnitsFromFile()*** der Text mit einem ***BufferedReader*** auf einem ***FileReader*** mit UTF-8-Codierung bzw. auf einem ***StringReader*** nach dem im Lösungsweg beschriebenen Prinzip in seine Einheiten aufgeteilt.

Die Zeichen werden einzeln eingelesen, mithilfe einer weiteren Methode ***charType()*** einem Typen zugewiesen und entsprechend interpretiert, sodass am Ende die einzelnen Einheiten, deren Start- und End-Indizes und die Neuzeilen-Indizes bestimmt sind.

linesByCharIndices und lineByCharIndex:

```
public int[] linesByCharIndices(int index0, int index1) {
public int lineByCharIndex(int index, boolean newlinesIgnored) {
```

Diese beiden Methoden geben anhand von Zeichenindexen im String die Indexe der Zeilen zurück, in denen diese stehen.

Dafür werden die Neuzeilen-Indexe aus newLineIndices iteriert, bis der aktuelle Index größer/gleich dem jeweils angegebenen Index ist. Der vorige Zeilenindex wird dann dem jeweiligen Zeichenindex zugeordnet.

Mit dem Parameter ***newlinesIgnored*** in ***lineByCharIndex()*** wird angegeben, ob der übergebene Index Neuzeilenzeichen beachtet (***false***) oder nicht (***true***). Ist letzteres der Fall, wird der übergebene Index für jeden neu iterierten Zeilenindex zusätzlich um 1 erhöht, damit die Neuzeilenzeichen mitgezählt werden.

Die Klasse „Completion“:

Repräsentiert eine Vervollständigung mit ihren wichtigen Daten. Im Folgenden die Attribute und Methoden außer den Gettern und Settern.

Attribute:

Dies sind die Attribute von ***Completion***:

- Ein String-Array ***units*** zur Speicherung der bisher eingesetzten Einheiten. An den Indexen, in denen noch keine Einheit eingetragen wurde, steht ein ***null***-Wert
- Die Anzahl an insgesamt auszufüllenden Einheiten ***length*** als Integer-Wert
- Die Anzahl an ausgefüllten Einheiten ***nUnits*** als Integer-Wert
- Der Integer-Wert ***startUnitIndex***, der den Einheitenindex im Suchtext angibt, an dem die Vervollständigung startet
- Die Integer-Arrays ***charIndices*** und ***lineIndices*** zur Speicherung der Start- und End-Zeichen- und Zeilen-Indexe

Konstruktor:

```
public Completion(int startUnitIndex, int length) {
```

Die Attribute ***startUnitIndex*** und ***length*** des Objekts werden mit den Werten der gleichnamigen Parameter initialisiert.

charIndices und ***lineIndices*** werden jeweils als neues Integer-Array der Länge 2 initialisiert, ***units*** als neues String-Array der Länge ***length*** und ***nUnits*** mit 0.

append und isComplete:

```
public void append(String unit) {
public boolean isComplete() {
```

Mit der ***append***-Methode wird im Array ***units*** am Index ***nUnits***, also an der nächsten unausgefüllten Stelle, die übergebene Einheit eingetragen.

Die Methode ***isComplete*** gibt zurück, ob die Anzahl der ausgefüllten Einheiten ***nUnits*** mindestens gleich der Länge ***length*** ist, also ob die Vervollständigung komplett ist.

Die Methode info:

```
public String info() {
```

Gibt die Informationen einer fertigen Vervollständigung als String zurück. Dieser besteht aus dem Text der Vervollständigung und der zugehörigen Zeilenangabe.

Der Vervollständigungstext wird ausgegeben, indem zuerst die Einheiten der Vervollständigung mit Leerzeichen voneinander getrennt in einem String gespeichert werden und dann vor allen Interpunktionszeichen (.,:;) die Leerzeichen entfernt werden.

Die anschließende Zeilenangabe wird aus den Zeilenindexen in ***lineIndices*** ermittelt. Wenn die Endzeile auch die Startzeile ist, ist das Format „Zeile x“, ansonsten „Zeile x - y“.

Die Klasse „GapText“:

Hier die Attribute und relevanten Methoden der Klasse ***GapText***.

Attribute:

GapText verfügt über folgende Attribute:

- Ein String-Array ***gapUnits*** zur Speicherung der mit Leerzeichen getrennten Texteinheiten. Unterstriche werden als ***null***-Wert dargestellt.
- Die Anzahl an Einheiten ***nUnits*** als Integer-Wert
- Der String ***textString*** zur Speicherung des Lückentextes ohne Zeilenumbrüche und sonstige unsichtbare Zeichen an den Zeilen-Anfängen und -Enden
-

Konstruktoren:

```
public GapText(String path, String filename) throws IOException {
public GapText(String gapTextString) {
public GapText(File file) throws IOException {
```

Es gibt einen Konstruktor mit Angabe von einem Pfad und Dateinamen und, wie in ***BaseText***, einen mit String- und einen mit ***File***-Parameter. In allen Konstruktoren wird ein Scanner mit dem entsprechenden String oder ***File*** als Eingabequelle erstellt und an die Methode ***createUnits()*** übergeben.

Ähnlich wie bei ***BaseText*** teilt diese Methode den Text in seine Einheiten auf. In diesem Fall wird die Quelle zeilenweise eingelesen und alle nicht leeren Zeilen werden mit der ***strip()***-Methode formatiert und mit einem Leerzeichen am Ende zu einem ***StringBuilder*** hinzugefügt. An den Leerzeichen wird der resultierende String in die Einheiten aufgeteilt und die Attribute entsprechend initialisiert.

Die Methode match:

```
public boolean match(String unit, int index, boolean matchCase) {
```

Prüft, ob die Einheit ***unit*** am angegebenen Einheitenindex in den Lückentext passt, mit ***matchCase*** als Angabe, ob die Groß-Klein-Schreibung beachtet werden muss.

Wenn an dieser Stelle ein Unterstrich bzw. ein ***null***-Wert steht und die Einheit mindestens einen Buchstaben oder eine Ziffer enthält, wird ***true*** zurückgegeben.

Wenn im Lückentext eine Vergleichseinheit steht, wird sie je nach Wert von ***matchCase*** durch ***equals()*** aus der String-Klasse bzw. durch ***equalsIgnoreCase()*** mit der Einheit ***unit*** verglichen und das Ergebnis zurückgegeben.

Gibt ***false*** zurück, wenn die Lückentexteinheit ***null*** ist, aber ***unit*** keine Buchstaben oder Zahlen enthält.

Die Methode findCompletions:

```
public ArrayList<Completion> findCompletions(BaseText baseText, boolean
matchCase) {
```

Sucht alle Vervollständigungen und gibt sie in Form einer ArrayList mit ***Completion***-Objekten zurück.

Zuerst werden einige Variablen deklariert. Darunter eine mit ***getTextUnits()*** erhaltene ***ArrayList*** der Suchtext-Einheiten, deren Anzahl und drei ***Completion***-ArrayLists:

- ***searches*** mit den unfertigen Vervollständigungen
- ***complete*** mit den fertigen Vervollständigungen
- ***remove*** mit den Vervollständigungen, die wegen einer ***false***-Ausgabe von ***match()*** gescheitert oder bereits fertig sind. Diese werden anschließend aus ***searches*** entfernt.

In einer for-Schleife werden alle Einheiten von ***baseText*** iteriert:

```
for (int i = 0; i < textUnits.size(); i++) {
```

Nach dem Zuweisen der Einheit am aktuellen Index zur Variable ***unit*** kann der Inhalt dieser Schleife in drei Schritte aufgeteilt werden:

1.: Verarbeitung der unfertigen Vervollständigungen

searches wird durchlaufen. Wenn eine Vervollständigung fertig ist, wird sie zu ***complete*** und zu ***remove*** hinzugefügt.

```
    for (Completion c: searches) {
        if (c.isComplete()) {
            complete.add(c);
            remove.add(c);
        }
```

Wenn ansonsten ***unit*** zu der nächsten unvervollständigten Einheit von ***c*** passt, wird ***unit*** an ***c*** angefügt.

```
        else if (match(unit, c.getnUnits(), matchCase)) {
            c.append(unit);
```

Trifft beides nicht zu, wird ***c*** zu ***remove*** hinzugefügt.

```
        } else {
             remove.add(c);
        }
```

*2.: Entfernen der Vervollständigungen in **remove***

Alle Elemente aus ***remove*** werden aus ***searches*** entfernt. Schließlich wird ***remove*** geleert.

```
    for (Completion r: remove) {
        searches.remove(r);
    }
    remove.clear();
```

3.: Evtl. Starten einer neuen Vervollständigung

Nur wenn die Anzahl der übrigen, zu durchsuchenden Texteinheiten mindestens genauso groß ist wie die Anzahl der Lückentext-Einheiten, lohnt es sich, evtl. eine neue Vervollständigung zu beginnen. Ansonsten würde sie garantiert nicht fertiggestellt werden.

```
    if (nSearchUnits - i >= nUnits) {
```

Wenn letzteres der Fall ist und ***unit*** zur ersten Lückentext-Einheit passt, wird eine neue Vervollständigung erstellt. Zu dieser wird ***unit*** angefügt und die Vervollständigung wird zu ***searches*** hinzugefügt.

```
        if (match(unit, 0, matchCase)) {
            Completion c = new Completion(i, nUnits);
            c.append(unit);
            searches.add(c);
        }
```

Wenn die Anzahl der übrigen Suchtexteinheiten kleiner ist als die Einheitenzahl des Lückentextes und ***searches*** leer ist, kann die Schleife abgebrochen werden, weil weder Vervollständigungen ausgefüllt noch neu erstellt werden.

```
    } else if (searches.isEmpty()) {
        break;
    }
}
```

Nach Beenden der Schleife ist eventuell noch eine fertige Completion in ***searches***, weil sie erst beim letzten Durchlauf vervollständigt wurde. Diese wird zu ***complete*** hinzugefügt.

```
for (Completion c: searches) {
    if (c.isComplete()) {
        complete.add(c);
    }
}
```

Zuletzt müssen noch für jede fertige Vervollständigung der Start- und End-Zeichenindex als auch der Start- und End-Zeilenindex bestimmt werden.

```
for (Completion c: complete) {
    c.setCharIndices(baseText.charByUnitIndices
            (c.getStartUnitIndex(), c.getEndUnitIndex()));
    c.setLineIndices(baseText.linesByCharIndices
            (c.getStartIndex(), c.getEndIndex()));
}
return complete;
```

Die Methode completionsInfo:

```
public String completionsInfo(ArrayList<Completion> completions) {
```

Gibt Anhand der übergebenen Vervollständigungen in der ***ArrayList*** „completions“ deren Informationen als String zurück. Zuerst wird der Lückentext ausgegeben. Wenn es keine Vervollständigungen gibt, wird der Benutzer darüber informiert.

Ansonsten werden mithilfe von ***Completion.info()*** die einzelnen Vervollständigungen und ihre Zeilenindexe ausgegeben. Zwischen den Daten für jede ***Completion*** steht jeweils eine Leerzeile.

Die Benutzeroberfläche:

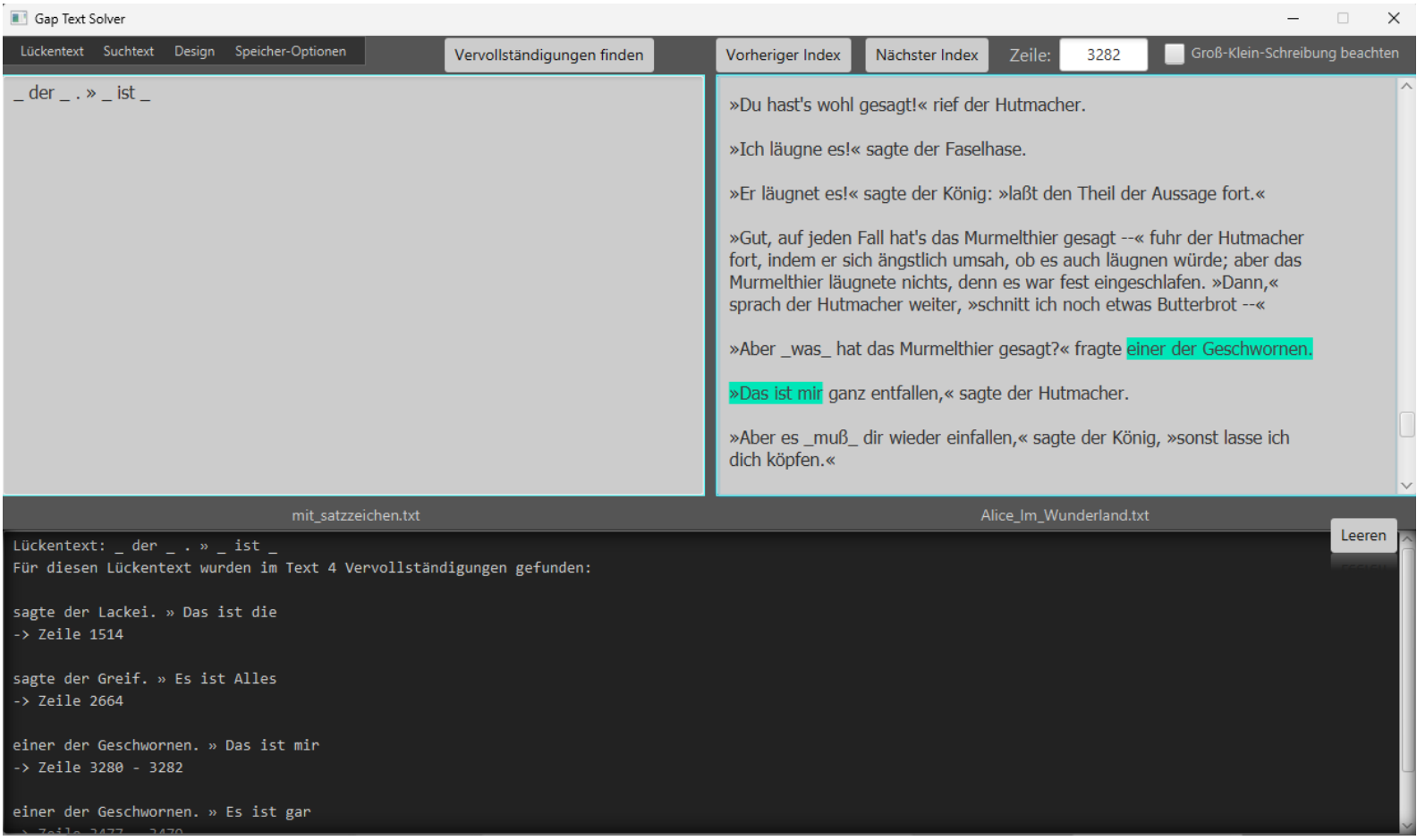
Die Benutzeroberfläche für meine Anwendung beinhaltet eine Main- und eine Controller-Klasse. Die graphische Gestaltung wie auch die Methodenbindungen werden fast ausschließlich in der FXML-Datei und den CSS-Dateien festgelegt.

Neben dem Finden und Ausgeben von Lückentext-Vervollständigungen hat die GUI noch viele andere nützliche Features, zum Beispiel:

- Ein Textfeld zur Bearbeitung des Lückentextes und eines zur Bearbeitung des Suchtextes
- Ein Ausgabefeld, in dem nach dem Suchen die Ergebnisse ausgegeben werden und das mit einem Knopf geleert werden kann
- Ein Menü zum Öffnen, Speichern, Speichern an einem bestimmten Ort und Schließen des Lückentextes und dasselbe für den Suchtext
- Ein Feld, das die aktuelle Zeile im Suchtext-Feld anzeigt und bei Eingabe zur angegebenen Zeile springt
- Eine Auswahl für die nächste/vorherige gefundene Vervollständigung, die im Suchtextfeld markiert wird

- Ein Kästchen zur Auswahl, ob Groß-Klein-Schreibung beim Suchen beachtet werden soll
- Ein Menü zum Wechseln zwischen Dark- und Light Mode
- Ein Menü zum Ändern der Speicheroption zwischen „Nachfragen“, „Immer speichern“ und „Nie speichern“

Ein Screenshot der GUI im Dark Mode



Beispiele

Hier die Ausgaben im Ausgabefeld für alle Beispiel-Lückentexte und einige eigens erstellte Lückentexte.

stoerung0.txt:

Lückentext: das _ mir _ _ _ vor

Für diesen Lückentext wurde im Text eine Vervollständigung gefunden:

Das kommt mir gar nicht richtig vor

-> Zeile 440

stoerung1.txt:

Lückentext: ich muß _ clara _
Für diesen Lückentext wurden im Text 2 Vervollständigungen gefunden:

Ich muß in Clara verwandelt
-> Zeile 425

Ich muß doch Clara sein
-> Zeile 441

stoerung2.txt:

Lückentext: fressen _ gern _
Für diesen Lückentext wurden im Text 3 Vervollständigungen gefunden:

Fressen Katzen gern Spatzen
-> Zeile 213 - 214

Fressen Katzen gern Spatzen
-> Zeile 214

Fressen Spatzen gern Katzen
-> Zeile 214

stoerung3.txt:

Lückentext: das _ fing _
Für diesen Lückentext wurde im Text eine Vervollständigung gefunden:

das Spiel fing an
-> Zeile 2319

stoerung4.txt:

Lückentext: ein _ _ tag
Für diesen Lückentext wurde im Text eine Vervollständigung gefunden:

ein sehr schöner Tag

-> Zeile 2293

stoerung5.txt:

Lückentext: wollen _ so _ sein
Für diesen Lückentext wurde im Text eine Vervollständigung gefunden:

Wollen Sie so gut sein
-> Zeile 2185

nicht_vorhanden.txt (Lückentext, der im Standard-Buchtext keine Lösung hat):

Lückentext: ist _ nicht _ da .
Für diesen Lückensatz gibt es im Text keine Vervollständigungen.

mit_satzzeichen.txt (Enthält zu beachtende Satzzeichen):

Lückentext: _ der _ . » _ ist _
Für diesen Lückentext wurden im Text 4 Vervollständigungen gefunden:

sagte der Lackei. » Das ist die
-> Zeile 1514

sagte der Greif. » Es ist Alles
-> Zeile 2664

einer der Geschwornen. » Das ist mir
-> Zeile 3280 - 3282

einer der Geschwornen. » Es ist gar
-> Zeile 3477 – 3479

mehrere_zeilen.txt (Textdatei mit mehreren Zeilen):

Lückentext: _ zweiundvierzig _ von
Für diesen Lückentext wurde im Text eine Vervollständigung gefunden:

Mit zweiundvierzig Illustrationen von
-> Zeile 14 - 16

groß_klein_chaos.txt (Willkürliche Groß-Klein-Schreibung, Ausgabe ohne „Groß-Klein-Schreibung beachten“ ausgewählt):

Lückentext: ALs eS _ üBEr _ _ , _ sIe

Für diesen Lückentext wurde im Text eine Vervollständigung gefunden:

als es dicht über ihr war, sprach sie

-> Zeile 1021 - 1022

längster_ununterbrochener_satz.txt (Anzahl an Stellen entspricht der Länge des längsten Satzes im Buch, der nicht von Satzzeichen unterbrochen ist):

Lückentext: _

Für diesen Lückentext wurde im Text eine Vervollständigung gefunden:

Bilde dir nie ein verschieden von dem zu sein was Anderen erscheint daß was du warest oder gewesen sein möchtest nicht verschieden von dem war daß was du gewesen warest ihnen erschienen wäre als wäre es verschieden

-> Zeile 2565 - 2568

nur_lücke.txt (Ein Lückentext, der aus nichts als einer Lücke besteht, hier nur die ersten 5 Suchergebnisse):

Lückentext: _

Für diesen Lückentext wurden im Text 25456 Vervollständigungen gefunden:

Alice's

-> Zeile 1

Abenteuer

-> Zeile 1

im

-> Zeile 3

Wunderland

-> Zeile 3

von

-> Zeile 5

Quellcode

Das ist der Quellcode der Klassen ***BaseText***, ***GapText***, ***Completion*** und der Methode ***findGtCompletions()*** aus der ***Controller***-Klasse.

BaseText.java:

```
package application;

import java.io.File;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.StringReader;
import java.io.IOException;

import java.nio.charset.StandardCharsets;
import java.lang.Character;

import java.util.ArrayList;

/**
 * Klasse für die Verarbeitung des zu durchsuchenden Textes
 */
public class BaseText {
    private enum C_TYPES {
        LETTER, WHITESPACE, NEWLINE, BOM, MISC
    }
    private final ArrayList<String> textUnits;
    private final ArrayList<int[]> unitStartEndIndices;
    private String textString;

    private final ArrayList<Integer> newLineIndices;
    private final int nLines;

    public BaseText(String text) throws IOException {
        textString = text;
        textUnits = new ArrayList<String>();
        unitStartEndIndices = new ArrayList<int[]>();
        newLineIndices = new ArrayList<Integer>();
        assembleTextUnitsFromString(text);
        nLines = newLineIndices.size() + 1;
    }
    public BaseText(File file) throws IOException {
        textUnits = new ArrayList<String>();
        unitStartEndIndices = new ArrayList<int[]>();
        newLineIndices = new ArrayList<Integer>();
        assembleTextUnitsFromFile(file);
        nLines = newLineIndices.size() + 1;
    }
    public BaseText() throws IOException {
        this(new File("Alice_im_Wunderland.txt"));
    }
    public BaseText fromPath (String path) throws IOException {
        return new BaseText(new File(path));
    }

    private void assembleTextUnitsFromString(String text) throws IOException {
```

```
        BufferedReader reader = new BufferedReader(new StringReader(text));
        String unit = "";
        int charIndex = -1;
        int[] currentUnitIndices = new int[2];
        int cInt;
        while ((cInt = reader.read()) != -1) {
            char c = (char) cInt;
            C_TYPES type = charType(c);
            if (type == C_TYPES.BOM) {
                continue;
            }
            charIndex ++;
            switch (type) {
                case MISC, LETTER:
                    if (! unit.isEmpty()) {
                        boolean b;
                        if (type == C_TYPES.LETTER) {
                            b = charType(unit.charAt(0)) == C_TYPES.LETTER;
                        } else {
                            b = unit.charAt(0) == c;
                        }
                        if (b) {
                            unit += c;
                            continue;
                        }
                        textUnits.add(unit);
                        currentUnitIndices[1] = charIndex - 1;
                        unitStartEndIndices.add(currentUnitIndices);
                        currentUnitIndices = new int[2];
                    }
                    currentUnitIndices[0] = charIndex;
                    unit = String.valueOf(c);
                    break;
                case WHITESPACE, NEWLINE:
                    if (! unit.isEmpty()) {
                        if (! (type == C_TYPES.NEWLINE && charType(unit.charAt(0)) ==
C_TYPES.LETTER
                                && unit.endsWith("-"))) {
                            textUnits.add(unit);
                            currentUnitIndices[1] = charIndex - 1;
                            unitStartEndIndices.add(currentUnitIndices);
                            currentUnitIndices = new int[2];
                            unit = "";
                        }
                    }
                    if (type == C_TYPES.NEWLINE) {
                        newLineIndices.add(charIndex);
                    }
            }
        }
        textUnits.add(unit);
        currentUnitIndices[1] = charIndex;
        unitStartEndIndices.add(currentUnitIndices);
    }
    private void assembleTextUnitsFromFile(File file) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(file,
StandardCharsets.UTF_8));
        String unit = "";
        StringBuilder textStringBuilder = new StringBuilder();
        int charIndex = -1;
        int[] currentUnitIndices = new int[2];
        int cInt;
        while ((cInt = reader.read()) != -1) {
            char c = (char) cInt;
            C_TYPES type = charType(c);
```

```
            if (type == C_TYPES.BOM) {
                continue;
            }
            textStringBuilder.append(c);
            charIndex ++;
            switch (type) {
                case MISC, LETTER:
                    if (! unit.isEmpty()) {
                        boolean b;
                        if (type == C_TYPES.LETTER) {
                            b = charType(unit.charAt(0)) == C_TYPES.LETTER;
                        } else {
                            b = unit.charAt(0) == c;
                        }
                        if (b) {
                            unit += c;
                            continue;
                        }
                        textUnits.add(unit);
                        currentUnitIndices[1] = charIndex - 1;
                        unitStartEndIndices.add(currentUnitIndices);
                        currentUnitIndices = new int[2];
                    }
                    currentUnitIndices[0] = charIndex;
                    unit = String.valueOf(c);
                    break;
                case WHITESPACE, NEWLINE:
                    if (! unit.isEmpty()) {
                        if (! (type == C_TYPES.NEWLINE && charType(unit.charAt(0)) ==
C_TYPES.LETTER
                                && unit.endsWith("-"))) {
                            textUnits.add(unit);
                            currentUnitIndices[1] = charIndex - 1;
                            unitStartEndIndices.add(currentUnitIndices);
                            currentUnitIndices = new int[2];
                            unit = "";
                        }
                    }
                    if (type == C_TYPES.NEWLINE) {
                        newLineIndices.add(charIndex);
                    }
            }
        }
        textString = textStringBuilder.toString();
        textUnits.add(unit);
        currentUnitIndices[1] = charIndex;
        unitStartEndIndices.add(currentUnitIndices);
    }

    private static C_TYPES charType(char c) {
        if (Character.isAlphabetic(c) || Character.isDigit(c) ||
                c == '\'' || c == '-') {
            return C_TYPES.LETTER;
        }
        if (Character.isWhitespace(c)) {
            if (c == '\n') {
                return C_TYPES.NEWLINE;
            }
            return C_TYPES.WHITESPACE;
        }
        if (c == '\ufeff') {
            return C_TYPES.BOM;
        }
        return C_TYPES.MISC;
    }
```

```
    public int[] linesByCharIndices(int index0, int index1) {
        if (index0 > index1) {
            int t = index0;
            index0 = index1;
            index1 = t;
        }
        int[] lines = {-1, -1};
        for (int i = 1; i < nLines + 1; i++) {
            int nlIndex;
            if (i < nLines) {
                nlIndex = newLineIndices.get(i - 1);
            } else {
                nlIndex = textString.length();
            }

            if (nlIndex >= index0 && lines[0] == -1) {
                lines[0] = i - 1;
            }
            if (nlIndex >= index1) {
                lines[1] = i - 1;
                return lines;
            }
        }
        if (lines[0] > -1) {
            lines[1] = nLines - 1;
            return lines;
        }
        return null;
    }
    public int lineByCharIndex(int index, boolean newlinesIgnored) {
        for (int i = 1; i < nLines + 1; i++) {
            int nlIndex;
            if (i < nLines) {
                nlIndex = newLineIndices.get(i - 1);
            } else {
                nlIndex = textString.length();
            }
            if (newlinesIgnored) {
                index += 1;
            }

            if (nlIndex >= index) {
                return i - 1;
            }
        }
        return nLines - 1;
    }

    public int[] charByUnitIndices(int index0, int index1) {
        int[] indices = new int[2];
        indices[0] = unitStartIndex(index0);
        indices[1] = unitEndIndex(index1);
        return indices;
    }

    public ArrayList<String> getTextUnits() {
        return textUnits;
    }
    public int unitStartIndex(int unitIndex) {
        return unitStartEndIndices.get(unitIndex)[0];
    }
    public int unitEndIndex(int unitIndex) {
        return unitStartEndIndices.get(unitIndex)[1];
    }
```

```
    public String getTextString() {
        return textString;
    }
    public int getNLines() {
        return nLines;
    }
    public int getLineStartIndex(int lineIndex) {
        if (lineIndex > 0) {
            return newLineIndices.get(lineIndex - 1);
        }
        return 0;
    }
}
```

GapText.java:

```
package application;

import java.util.Scanner;
import java.io.IOException;
import java.io.File;
import java.nio.charset.StandardCharsets;

import java.util.ArrayList;

/**
 * Klasse für die Verarbeitung eines Lückentextes
 */
public class GapText {
    private String[] gapUnits;
    private int nUnits;
    private String textString;

    public GapText(String path, String filename) throws IOException {
        Scanner scan = new Scanner(new File(path + "\\" + filename + ".txt"),
                StandardCharsets.UTF_8);
        createUnits(scan);
    }
    public GapText(String gapTextString) {
        Scanner scan = new Scanner(gapTextString);
        createUnits(scan);
    }
    public GapText(File file) throws IOException {
        Scanner scan = new Scanner(file);
        createUnits(scan);
    }

    private void createUnits(Scanner scan) {
        StringBuilder string = new StringBuilder();
        while (scan.hasNextLine()) {
            String line = scan.nextLine();
            if (! line.isBlank()) {
                string.append(line.strip()).append(" ");
            }
        }
        String res = string.toString();
        if (! res.isEmpty()) {
            if (res.charAt(0) == '\ufeff') {
                res = res.substring(1);
            }
        }
        textString = res.strip();
        gapUnits = textString.split(" ");
```

```
        nUnits = gapUnits.length;

        for (int i = 0; i < nUnits; i++) {
            if (gapUnits[i].equals("_")) {
                gapUnits[i] = null;
            }
        }
    }

    /** Gibt ein GapText-Objekt mit der Konsoleneingabe
     * als Datei zurück, falls vorhanden.
     */
    public static GapText fromFilenameInput() {
        Scanner scan = new Scanner(System.in);
        while (true) {
            System.out.print("Name der Lückentext-Datei (ohne \".txt\"): ");
            String input = scan.nextLine();
            try {
                return new GapText("Beispieleingaben", input);
            } catch (IOException e) {
                System.out.println("Die Datei \"" + input +
                        ".txt\" existiert nicht unter \"Beispieleingaben\".\n");
            }
        }
    }

    /** Gibt true zurück, wenn die Einheit unit an der Stelle
     * index in den Lückentext passt.
     */
    public boolean match(String unit, int index, boolean matchCase) {
        if (gapUnits[index] == null) {
            for (char c: unit.toCharArray()) {
                if (Character.isAlphabetic(c) || Character.isDigit(c)) {
                    return true;
                }
            }
        } else if (matchCase) return unit.equals(gapUnits[index]);
        else return unit.equalsIgnoreCase(gapUnits[index]);

        return false;
    }

    /** Findet alle Lösungen für den Lückentext im Suchtext baseText
     * und gibt sie in einer ArrayList zurück.
     */
    public ArrayList<Completion> findCompletions(BaseText baseText, boolean matchCase)
{
        ArrayList<String> textUnits = baseText.getTextUnits();
        int nSearchUnits = textUnits.size();
        ArrayList<Completion> searches  = new ArrayList<Completion>();
        ArrayList<Completion> complete  = new ArrayList<Completion>();
        ArrayList<Completion> remove  = new ArrayList<Completion>();

        for (int i = 0; i < nSearchUnits; i++) {
            String unit = textUnits.get(i);

            for (Completion c: searches) {
                if (c.isComplete()) {
                    complete.add(c);
                    remove.add(c);
                }
                else if (match(unit, c.getnUnits(), matchCase)) {
                    c.append(unit);
                } else {
                    remove.add(c);
```

```
                }
            }

            for (Completion r: remove) {
                searches.remove(r);
            }
            remove.clear();

            if (nSearchUnits - i >= nUnits) {
                if (match(unit, 0, matchCase)) {
                    Completion c = new Completion(i, nUnits);
                    c.append(unit);
                    searches.add(c);
                }
            } else if (searches.isEmpty()) {
                break;
            }
        }
        for (Completion c: searches) {
            if (c.isComplete()) {
                complete.add(c);
            }
        }
        for (Completion c: complete) {
            c.setCharIndices(baseText.charByUnitIndices
                    (c.getStartUnitIndex(), c.getEndUnitIndex()));
            c.setLineIndices(baseText.linesByCharIndices
                    (c.getStartIndex(), c.getEndIndex()));
        }
        return complete;
    }

    /** Gibt die Informationen der Vervollständigungen in completions
     * mit jeweils einer Leerzeile Abstand als String zurück.
     */
    public String completionsInfo(ArrayList<Completion> completions) {
        String info = "Lückentext: " + textString + "\n";
        if (completions.isEmpty()) {
            return info + "Für diesen Lückensatz gibt es im Text keine
Vervollständigungen.";
        }
        info += "Für diesen Lückentext ";
        if (completions.size() > 1) {
            info += "wurden im Text " + completions.size() + " Vervollständigungen
gefunden:\n\n";
        } else {
            info += "wurde im Text eine Vervollständigung gefunden:\n\n";
        }

        int nComp = completions.size();
        String[] cInfos = new String[nComp];
        for (int i = 0; i < nComp; i++) {
            cInfos[i] = completions.get(i).info();
        }
        return info + String.join("\n\n", cInfos);
    }

    public String getTextString() {
        return textString;
    }
}
```

Completion.java:

```
package application;
```

```java
/**
 * Klasse zur Speicherung und Verwaltung einer
 * Lückentext-Vervollständigung
 */
public class Completion {
    private String[] units;
    private int length;
    private int nUnits;

    // Einheiten-Index im Suchtext, an dem die Vervollständigung startet
    private int startUnitIndex;
    private int[] charIndices;
    private int[] lineIndices;

    public Completion(int startUnitIndex, int length) {
        this.startUnitIndex = startUnitIndex;
        this.length = length;
        charIndices = new int[2];
        lineIndices = new int[2];
        units = new String[length];
        nUnits = 0;
    }

    public void append(String unit) {
        units[nUnits] = unit;
        nUnits ++;
    }

    public boolean isComplete() {
        return nUnits >= length;
    }

    public String info() {
        String info = String.join(" ", units);
        char[] punctuation = {'.', ',', ':', ';'};
        for (char c: punctuation) {
            info = info.replace(" " + c, String.valueOf(c));
        }

        info += "\n-> Zeile " + (lineIndices[0] + 1);
        if (lineIndices[1] > lineIndices[0]) {
            info += " - " + (lineIndices[1] + 1);
        }
        return info;
    }

    public void setLineIndices(int[] indices) {
        lineIndices = indices;
    }
    public void setCharIndices(int[] charIndices) {
        this.charIndices = charIndices;
    }

    public int getnUnits() {
        return nUnits;
    }
    public int getStartUnitIndex() {
        return startUnitIndex;
    }
    public int getEndUnitIndex() {
        return startUnitIndex + length - 1;
    }

    public int getStartIndex() {
        return charIndices[0];
```

```
    }
    public int getEndIndex() {
        return charIndices[1];
    }

    public int getStartLineIndex() {
        return lineIndices[0];
    }
    public int getEndLineIndex() {
        return lineIndices[1];
    }
}
```

findGtCompletions() aus Controller.java:

```
public void findGtCompletions() throws IOException {
    if (! btIsCached) {
        baseText = new BaseText(baseTextArea.getText());
        btIsCached = true;
    }
    if (! gtIsCached) {
        gapText = new GapText(gapTextArea.getText());
        gtIsCached = true;
    }
    if (! completionsSearched) {
        markedIndices.clear();
        currentMarkedIndex = -1;
        completions = gapText.findCompletions(baseText, matchSearchCase);
        completionsSearched = true;

        if (! completions.isEmpty()) {
            for (Completion c: completions) {
                int[] indices = new int[2];
                indices[0] = c.getStartIndex();
                indices[1] = c.getEndIndex();
                if (ignoreBtLineBreaks) {
                    indices[0] -= c.getStartLineIndex();
                    indices[1] -= c.getEndLineIndex();
                }
                markedIndices.add(indices);
            }
            markIndex(currentMarkedIndex = 0);
            bPreviousIndex.setDisable(false);
            bNextIndex.setDisable(false);
        } else {
            bPreviousIndex.setDisable(true);
            bNextIndex.setDisable(true);
        }
        output.appendText(gapText.completionsInfo(completions) + "\n\n");
    } else if (currentMarkedIndex == -1 && ! markedIndices.isEmpty()) {
        onNextIndex();
    }
}
```