Jabberwocky - Schlussbericht

Ein statistisches Lernsystem für natürliche Sprachen

Autoren:

Mohamed Maxamed

Muhammet Yilmaz

Dozent:

Prof. Dr. Bradley Richards

Ort, Datum:

Olten, 02.12.2022

Inhaltsverzeichnis

Einleitung

Ein Programm zu entwickeln welches fähig ist, Sprachen zu lernen. wie soll das gehen? Eine komplexe Aufgabe, welche eine strukturierte und organisierte Vorgehensweise vorrausetzt. Die Hauptaufgabe dieses Programmes ist, durch einen gegebenen Algorithmus einen Text zu verarbeiten und mit vorgegebenen Regeln einen neuen Text zu generieren.

Der neu generierte Text wird in den meisten Fällen keinen Sinn ergeben, allerdings wird er syntaktisch vollkommen korrekt sein. Sprich, wenn der Eingabetext beispielsweise in der spanischen Sprache ist, wird der neu generierte Text auch die Regeln der spanischen Syntax befolgen. -> siehe Bild (Ausschnitt aus dem Programm)

Das Projekt Jabberwocky ist in 3 Phasen gegliedert. Bei der ersten Phase, welche auch das Grundgerüst und die Logik abbildet, geht es darum den gegebenen Algorithmus mit Hilfe von geeigneten Klassen und Methoden entsprechend in Java zu implementieren. Die grösste Schwierigkeit besteht hierbei die Textverarbeitung so zu implementieren, dass die gegebenen Regeln eingehalten werden. Ausserdem muss man sich klare Gedanken über die Datenstruktur machen welche man wählt. Immerhin hat jede Datenstruktur seine eigenen Vorteile wie Nachteile. Datenstrukturen werden benötigt um Daten (in diesem Fall ein Text) so abzuspeichern, dass diese zu einem gegebenen Zeitpunkt in einer gewünschten Form verfügbar sind.

Ist das Grundgerüst einmal erstellt kann eine dazu passende Benutzeroberfläche gebaut werden welches die Hauptaufgabe in Phase 2 ist. In der letzten Phase ist das Hauptziel, das bereits vollkommen funktionierende Programm zu optimieren. Dazu gehört beispielsweise das Austauschen der Datenstruktur.

Theoretische Grundlagen

Um ein solches Projekt in einer gewissen Qualität fertigzustellen, benötigt man eine strukturierte Vorgehensweise. Für die erste Phase haben wir uns grundlegende Fragen gestellt wie beispielsweise welche Klassen und Methoden benötigen wir? Bis wann wollen wir welche Phase abgeschlossen haben? Oder welche Zusatz-Funktionalitäten möchten wir einbauen?

Damit wir auch einen gewissen Zeitplan haben, haben wir uns gewisse Deadlines gesetzt, wie beispielsweise, dass die Phase 2 spätestens am 11. November fertig gestellt sein muss, da in dieser Woche das aktuelle Thema in den Vorlesungen für die Phase 3 relevant ist.

Das Model bildet die Logik hinter dem Programm ab. Damit wir uns vollkommen auf die Implementation der Logik konzentrieren konnten, haben wir zu Beginn den zu bearbeitenden Text und die Fenstergrösse festgelegt. Dies ermöglichte uns auch sicherzustellen, dass der Code funktioniert und der generierte Text dem Algorithmus, welcher bereits festgelegt wurde entspricht.

Gewählte Datenstruktur: Array List

Für die Speicherung der Texte haben wir uns für die ArrayList entschieden. Der Hauptgrund dafür war, dass uns diese Datenstruktur am vertrautesten ist und wir die Möglichkeiten dieser Datenstruktur bestens kennen. Ein Text ist in der Regel linear aufgebaut. Die Buchstaben müssen auf einer bestimmten Position stehen damit der Text einen nachvollziehbaren Sinn ergibt.

Wenn wir die ArrayList Klasse betrachten, sehen wir auch da eine gewisse Art von Linearität. Wenn wir beispielsweise ein Objekt der Liste hinzufügen, wird es üblicherweise ans Ende der Liste hinzugefügt ausser wir geben im Paramater vor, auf das wir eine andere Position wünschen.

Diese Datenstruktur ist auch sehr hilfreich, da Sie eine sehr hilfreiche Methode anbietet, die **Sort-Methode**. Diese erlaubt uns einen effizienteren Such-Algorithmus wie beispielsweise eine Binäre Suche.

Praktisches Implementationskonzept

Vom Einlesen einer Textdatei bis zur Generierung eines neuen Textes ist unser Modell sehr funktional aufgebaut. Für jede Teilbearbeitung des Textes haben wir eine Methode geschrieben.

Model

GenerateTextTeile():

```
To -- d
od -- a
da -- y
ay -- ■

Today■
```

Gibt eine Liste im folgenden Format zurück: Fenstergrösse +Trennzeichen+ nächstes Zeichen

Diese Methode erfüllt eine einfache Aufgabe. Sie erstellt eine Liste welche die Fenstergrösse (in diesem Beispiel 2) und das folgende Zeichen abspeichert. Anhand an einer Schleife durchwandert die Methode den Text so lange bis das nächste Zeichen das Schlusszeichen ist welches wir als «■» definiert haben. Wir haben uns für dieses Zeichen entschieden, da es ein sehr unübliches Zeichen in Texten ist. damit wir allerdings beide Teile des Wortpaares (Fenstergrösse + nächstes Zeichen) sauber voneinander mithilfe der Split-Methode trennen können benötigen wir ein Zeichen zwischen den Wortteilen damit kann Buchstabe des Textes verloren geht dieses ist üblicherweise auch ein unübliches Zeichen damit Fehler vermieden werden können.

GetNextChar(String input):

Diese Methode ist simpel gehalten. Sie erhält mittels der FindpossibleCharacters eine Liste mit den möglichen nächsten Zeichen für eine bestimmte Fenstergrösse (Beispielsweise «th»),wählt mittels Zufallsgenerator eine Position in der Liste aus und gibt das Zeichen auf der zufällig gewählten Position zurück.

FindpossibleCharacters():

Diese Methode gibt eine Liste zurück mit möglichen nächsten Zeichen. Hierbei durchwandert Sie die Liste, welche zu Beginn die GenerateTextTeile-Methode erstellt und die Textpaare in einem gewünschten Format gespeichert hat. Mit der folgenden Schleife wird sichergestellt, dass wenn bestimmte Wortkombinationen wie «th» öfters vorkommen, dass Ihre nächsten Zeichen ebenfalls der Liste hinzugefügt werden. -> Siehe Screenshot

```
92          // Schleife durch jeden String der Liste "TextTeile"
93          for(int i = 0; i<TextTeile.size();i++) {
94
95              String [] TextandnextChar =this.TextTeile.get(i).split(this.splitChar.toString());
96
97          // Wenn der Text gleich ist wie der input
98          if(TextandnextChar[0].equals(input)) {
99              // füge das dazugehörige nächste Zeichen dem Array hinzu
100         possibleCharacters.add(TextandnextChar[1].charAt(0));
101             }
102         }
103     return possibleCharacters;
```

GenerateText():

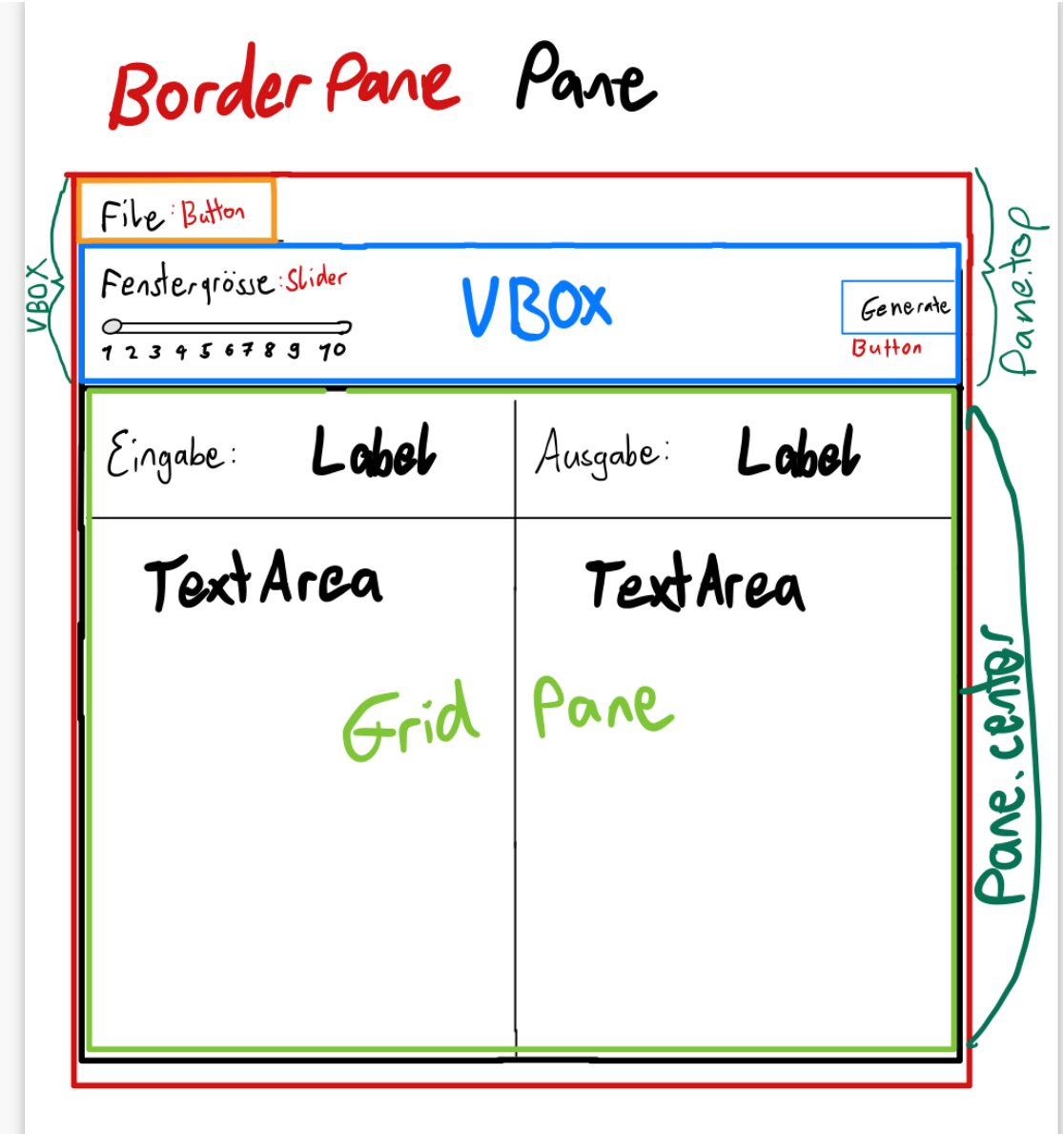
Diese Methode nutzt die oberen Methoden um einen neuen Text zu generieren. Hierbei wird anhand einer Schleife der neue Text aufgebaut. Das zufällige nächste Zeichen wird mit der getNextChar-Methode dem aufbauenden String angehängt.

FileToString():

Die FileToString-Methode liest eine TextDatei und gibt einen String zurück welcher als Eingabe für die Textverarbeitung gilt. Das auslesen der Datei wird durch einen Scanner durchgeführt. -> siehe Screenshot

```
Scanner scan = new Scanner(file);
while(scan.hasNextLine()) {
    text += scan.nextLine();
}
scan.close();
```

View



Controller

Die Hauptaufgabe des Controllers ist das Ausführen von Methoden, welche im Model definiert sind, sowie der Aktualisierung der View. Somit hat der Controller als einzige Komponente den Zugriff auf das Model sowie der Benutzeroberfläche. Im Controller werden alle Interaktionen des Benutzers mit der Oberfläche koordiniert. Es kann beispielsweise definiert werden welche Methoden ausgeführt werden sollen, wenn man einen bestimmten Button klickt.

In unserem Projekt wären es jetzt der «Datei» Button, um Textdateien einzulesen und der «Generieren» Button, um einen neuen Text zu generieren. Durch einen Klick des «Generieren» Buttons wird im Controller ein Ereignis ausgelöst. Dazu nutzt er die Methoden, welche im Model definiert sind. Anschliessend wird der neue Text auf der Benutzeroberfläche (View) angezeigt.

Ergebnisse

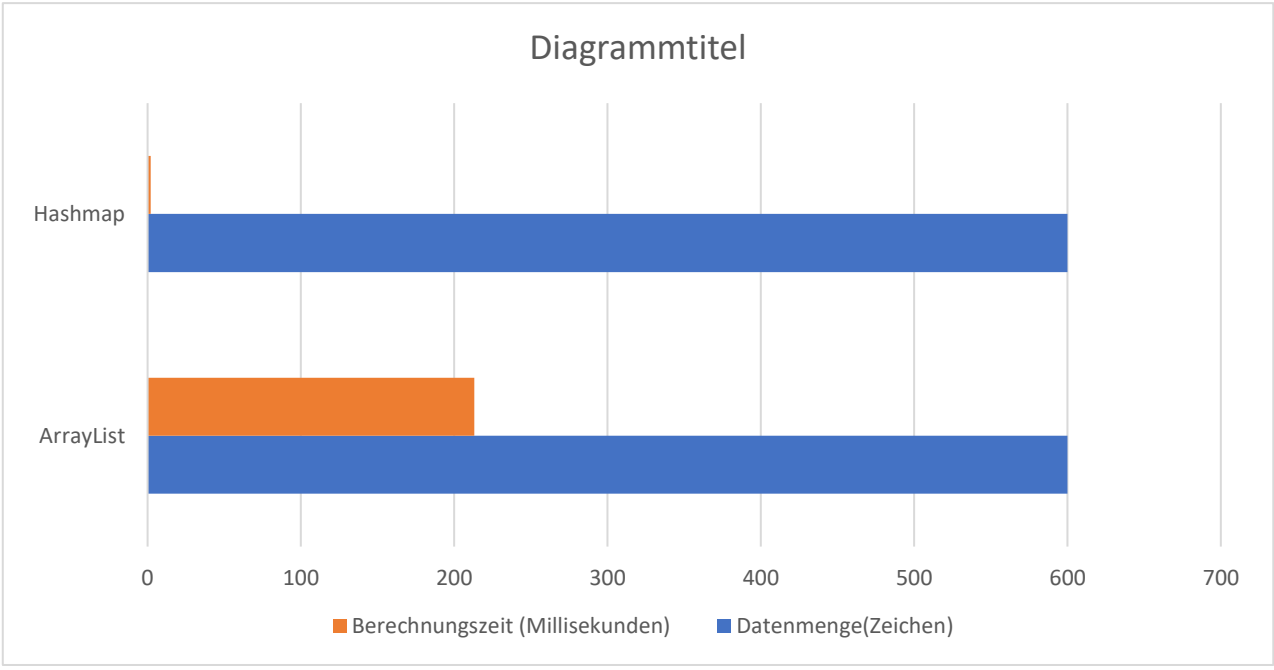
Der Hauptzweck dieses Projektes ist unserer Meinung nach das Erhalten eines besseren Verständnisses der Grundlegendsten Strukturen in Java. Dieses Projekt hat uns einen tieferen Einblick hinter die Kulissen gegeben.

Wechsel der Datenstruktur

Bevor wir mit dem Wechsel der Datenstruktur von einer Arraylist zu einer Hashmap gewechselt sind. Mussten wir zuerst sicherstellen, dass die Methoden richtig funktionieren beziehungsweise die Regeln zum generieren eines neuen Textes einhalten. Der Wechsel selbst ist uns allerdings einfacher gelungen als wir zu beginn gedacht hatten. Der Hauptgrund dafür wahr, dass wir diese Datenstruktur erste in Woche 8 detailliert betrachtet haben. Da wir bereits das Grundgerüst mit der ArrayList erarbeitet hatten, fiel es uns ziemlich leicht die Umstellung zu bewerkstelligen.

Schnelligkeit der verschiedenen Datenstrukturen:

Wie erwartet ist die Hashmap deutlich schneller als die ArrayList. Bei einer Datenmenge von 600 Zeichen benötigte die ArrayList durchschnittlich 200 Millisekunden. Bei der Hashmap sind es bei der gleichen Datenmenge hingegen nur 2 Millisekunden!.



Das Ersetzen der Arraylist durch eine Hashmap hat uns besonders bei der Methode der Methode findPossibleCharacters ziemlich viele Zeilen Code erspart. Die Hashmap ermöglicht das Durchlaufen des ganzen Textes bei der Ermittlung der möglichen nächsten Zeichen zu umgehen. Durch den Key (Die Fenstergrösse) ist es möglich direkt mit der Getter Methode die möglichen nächsten Zeichen aus der Hashmap zu holen. -> siehe Screenshot auf der nächsten Seite

```
134     // Methode gibt eine Liste mit allen möglichen nächsten Zeichen zurück
135
136         private List <Character> findPossibleCharacters(String input){
137     /*
138             List <Character> possibleCharacters = new ArrayList<>();
139                 // Schleife durch jeden String der Liste "TextTeile"
140             for(int i = 0; i<TextTeile.size();i++) {
141
142                 String [] TextandnextChar = this.TextTeile.get(i).split(splitChar.toString());
143
144                     // Wenn der Text gleich ist wie der input
145                     if(TextandnextChar[0].equals(input)) {
146                     // füge das dazugehörige nächste Zeichen dem Array hinzu
147                 possibleCharacters.add(TextandnextChar[1].charAt(0));
148             }
149         }
150         return possibleCharacters;
151     */
152             // wir holen die Elemente direkt von der Map
153             return Teile.get(input);
154         }
```

Schlussfolgernd können wir sagen, dass die gewählte Datenstruktur sehr stark davon abhängt für welche Zwecke man Sie nutzen möchte. Dieses Projekt hat uns gezeigt, dass jede Datenstruktur Ihre Vor- und Nachteile hat.

Zusammenfassung

Die Erarbeitung mit der ArrayList hat unser analytisches Denken so verbessert, dass wir es geschafft haben die Textverarbeitung mit der Hashmap mit nur kleinen Veränderungen zu erledigen Durch die Anpassungen, welche wir durchführen mussten, konnten wir nicht nur sehen wissen, dass eine Hashmap schneller ist sondern vielwichtiger auch an welchen Methoden die jeweilige Datenstruktur bei der Textverarbeitung effizienter arbeitet. Leider hat es zeitlich nicht mehr für die Implementierung einer binären Suche gereicht.

Die grösste Schwierigkeit bei diesem Projekt war die Erarbeitung der Methoden für die Textverarbeitung. Obwohl der Algorithmus vorgegeben wurde, war es schwierig die Logik in geeignete Methoden umzusetzen. Was uns bei der Implementierung ziemlich geholfen hat, war, die API's der Klassen welche wir benötigt haben durchzulesen. Dadurch erhielten wir einen guten Überblick über die Methoden, welche diese Klassen anbieten. Folglich fiel es uns wesentlich leichter diese bereits gegebenen Methoden für unsere geplante Textbearbeitung zu nutzen.

Im Großen und Ganzen sind wir sehr zufrieden über dieses Projekt. Es hat uns die Möglichkeit gegeben unser erlerntes Wissen anzuwenden und zu vertiefen.

Quellen und Literatur

Generate Text Methode für Hashmap: Java StringBuilder class- javatpoint

Einlesen einer TextDatei: (145) Java: Read Text File Easily - YouTube

JavaFX Button gestaltung: Master CSS in 5 Easy Steps | JavaFX GUI Tutorial for Beginners - YouTube

Zeitberechnung für Statistik: (145) Calculating elapsed time with System.currentTimeMillis() and System.nanoTime() | JAVA - YouTube

Erstellung einer Hashmap: Create a HashMap in Java (tutorialspoint.com)