T12 – Texterkennung

SpinPhone

Aufgabenstellung zur Hausarbeit

Softwaretechnologie: Java

im Fach Informationsverarbeitung

Sommersemester 2017

Mihail Atanassov
Börge Kiss

Inhaltsverzeichnis

1 Allgemeine Informationen

Die Hausarbeit besteht aus 3 Teilbereichen:

1) Implementation des Programms gemäß den Anforderungen dieser Beschreibung, Abgabe auf CD.
2) Interne Dokumentation aller Klassen, Attribute und Methoden mit JavaDoc - Evtl. zusätzlich notwendige Dokumentation innerhalb von Methodenkörpern (bspw. komplexere Algorithmen) ist sinnvoll (siehe auch Kapitel 4.4).
3) Externe Dokumentation – ausgedruckt, der Form einer wissenschaftlichen Arbeit entsprechend (siehe Hinweise zur Form der wissenschaftlichen Arbeit[1]). Der Inhalt der externen Dokumentation wird in Kapitel 7.3 kurz beschrieben.

Kennzeichnen Sie CD **und** externe Dokumentation mit Ihrem **Namen**, Ihrer **Matrikelnummer** sowie Ihrer **Email-Adresse** (wichtig! siehe Kapitel 7.4).

Das Programm kann sich an den Vorschlägen in diesem Dokument orientieren, Sie können jedoch auch eine völlig unabhängige, eigene Lösung programmieren. In diesem Fall sollten sie jedoch bei der Dokumentation besonders sorgfältig sein und Abweichungen zu den Vorschlägen begründen, zudem müssen die Anforderungen, die an das Programm gestellt werden, erfüllt werden.

Gruppenarbeit ist möglich, allerdings mit maximal 2 Teilnehmern und lediglich bei der Programmierung, nicht bei der Erstellung der internen und externen Dokumentation. Sie sollten mit dieser Arbeit beweisen, dass Sie der Sprache Java mächtig sind und in der Lage sind, Anforderungen wie die hier gestellten zu erfüllen. Sollte dies nicht möglich sein, weil Ihre Arbeit der eines/r Kommilitonen/in zu sehr ähnelt, werden Sie (beide) zu einer mündlichen Prüfung geladen, in der Sie Ihr Programm erläutern müssen.

2 Programmbeschreibung

Aufgabe ist es, ein Programm zu entwickeln und zu implementieren, das die auf Mobiltelefonen übliche Texterkennung T9 simuliert. T9 steht für „Text on 9 keys“ und bezeichnet ein Verfahren, das zur schnelleren Eingabe von Texten über die Tastatur von Mobiltelefonen gedacht ist.

Über die „echte“ Implementierung von T9 ist nicht viel bekannt – es kann jedoch angenommen werden, dass sie nicht der hier vorgeschlagenen Lösung entspricht, da diese den Nachteil hat, relativ viel Speicher zu benötigen, der in Mobiltelefonen äußerst knapp bemessen ist. Sollten Sie den Anspruch haben, eine speicherschonendere Möglichkeit der Texterkennung zu entwickeln,

[1] http://www.spinfo.phil-fak.uni-koeln.de/sites/spinfo/Studium/hinweise-wissenschaftliche-arbeit.pdf

so dürfen Sie dies gerne tun!

3 Anforderungen an die Datenstruktur

Die Datenstruktur, in der das Lexikon gespeichert werden soll, muss u.a. folgende Anforderungen erfüllen:

1. Sie sollte relativ leicht zu erzeugen sein
2. Sie muss effizient durchsucht werden können
3. Sie muss eine Konvertierung von Zahlenfolgen in Wörter ermöglichen
4. Sie muss Wahrscheinlichkeiten/Häufigkeiten von Wörtern speichern können
5. Sie muss in der Lage sein, effizient Alternativen zu einem Wort finden zu können
6. Sie muss erweiterbar bleiben, damit neue Wörter gelernt werden können

Im Folgenden soll eine Möglichkeit vorgestellt werden, wie eine solche Datenstruktur modelliert werden könnte. Dies ist jedoch nicht der einzig denkbare Weg, Sie können selbstverständlich auch eine eigene Lösung entwerfen und implementieren, oder die hier vorgestellte nur teilweise implementieren und einige Bereiche mit eigenen Ideen realisieren.

3.1 T9-Baum

Eine Möglichkeit, eine solche Datenstruktur zu implementieren, wäre ein „T9-Baum“. In diesem Baum entspricht jeder innere Knoten (d.h. alle Knoten außer den Blättern und der Wurzel) einer der Zahlen von 2 bis 9, also einer der Buchstabentasten eines Mobiltelefons. Jeder innere Knoten kann bis zu 8 Kinder haben, die erneut den Tasten 2 bis 9 entsprechen. Die Blätter im Baum (sowie einige innere Knoten, siehe Abbildung 4) sind spezielle Knoten, die jeweils ein Wort repräsentieren (d.h. als String-Objekt speichern). Abbildung 1 soll diese Idee verdeutlichen:

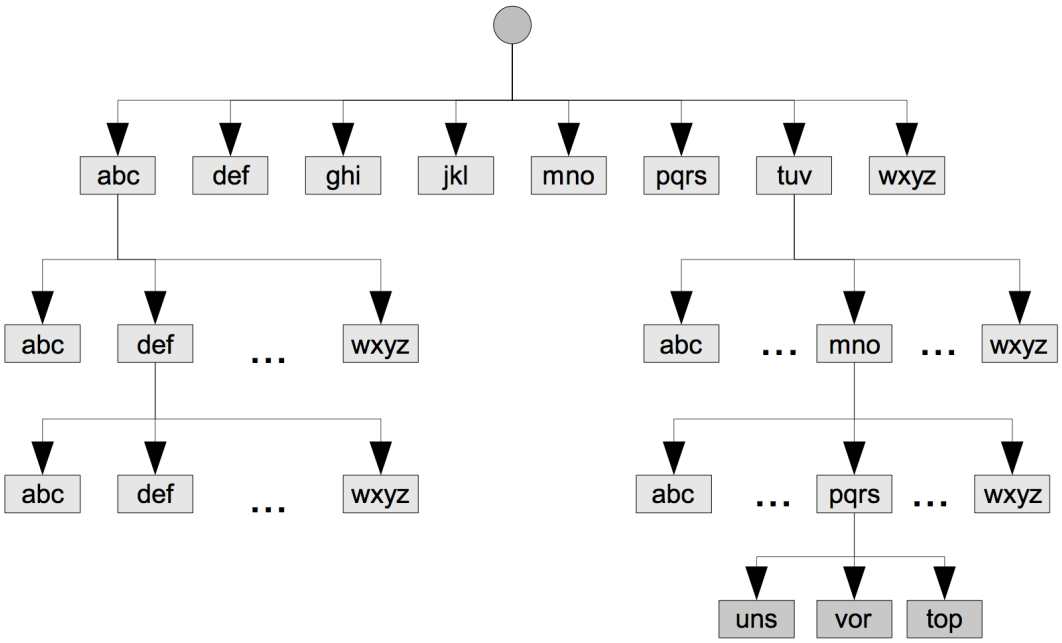

Abbildung 1: *Ausschnitt eines T9-Baums. Zur Veranschaulichung sind die Zahlen-Knoten hier mit den Buchstaben der jeweiligen Taste gekennzeichnet (s.a. Abbildung 2).*

Anmerkung: Der T9-Baum wird sehr groß: Während auf der ersten Ebene (Kinder der Wurzel) nur 8 Knoten vorhanden sind, enthält die 2. Ebene bereits 64 (8^2) Knoten und die dritte Ebene schon 512 (8^3) Knoten (unter der Annahme, dass jeder Knoten immer 8 Kinder hat, was tatsächlich jedoch nicht der Fall ist – so existiert bspw. kein Wort im Deutschen, dass mit den Buchstaben „yy“ beginnt).

Mit Hilfe dieser Datenstruktur lässt sich leicht nach einer Zeichenkette zu einer Folge von Zahlen suchen: Angenommen, der Benutzer möchte das Wort „vor“ schreiben, so drückt er nacheinander die Tasten 8, 6 und 7. Ausgehend von der Wurzel des Baumes würde nun ein Weg zu einem Blatt gesucht, wobei jede Zahl dem jeweiligen Kindknoten entspricht: Bei Eingabe der 8 würde der 8. Kindknoten des Wurzelknotens gewählt, bei anschließender Eingabe der 6 der 6. Kindknoten des 8. Kindknotens der Wurzel, und bei der Eingabe der 7 der 7. Kindknoten des 6. Kindknotens des 8. Kindknotens der Wurzel, wie Abbildung 2 zeigt:

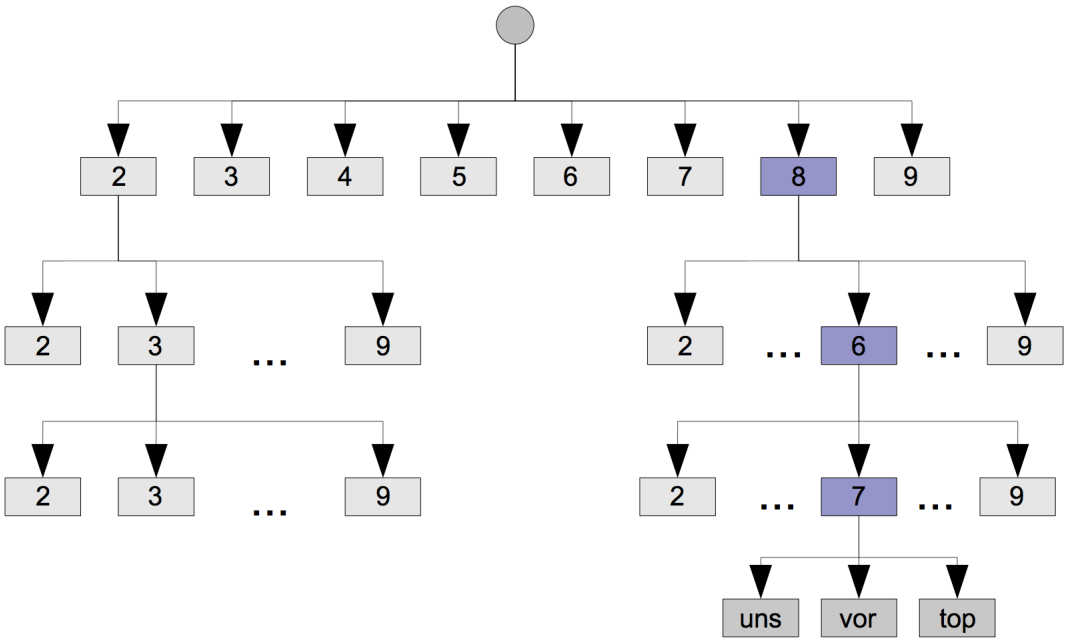

Abbildung 2: *Pfad für die Zahlenfolge 867, die u.a. den Wörtern "vor" und "uns" entspricht.*

Jedes Wort entspricht somit einem eindeutigen Weg von der Wurzel des Baums zu einem Knoten, während umgekehrt dieser Weg nicht eindeutig einem Wort entspricht. So ist 867 beispielsweise auch der Weg für das Wort „uns“.

Nach jedem Tastendruck muss das Programm die häufigste Zeichenrepräsentation der bisher betätigten Tasten finden und darstellen. So könnte beispielsweise „8“ als „u“ interpretiert werden, „86“ hingegen als „vo“ und „867“ schließlich als „vor“. Wie sich die Häufigkeit für eine Zeichenkette berechnen lässt, wird in Kapitel 3.2 erklärt, hier wird zunächst einfach angenommen, dass sie bereits berechnet wurde. Jeder Knoten k im Baum besitzt dafür ein zusätzliches Attribut, in dem gespeichert wird, wie häufig der Pfad von der Wurzel bis zum Knoten k in den verarbeiteten Texten gefunden wurde. Taucht bspw. die Zeichenfolge „uns“ 100-mal im Korpus auf, „vor“ 30-mal und „top“ zehnmal, so ist dies in den entsprechenden Knoten markiert wie in Abbildung 3 zu sehen ist. In den Elternknoten sind die Häufigkeiten der darunterliegenden Kindknoten gespeichert.

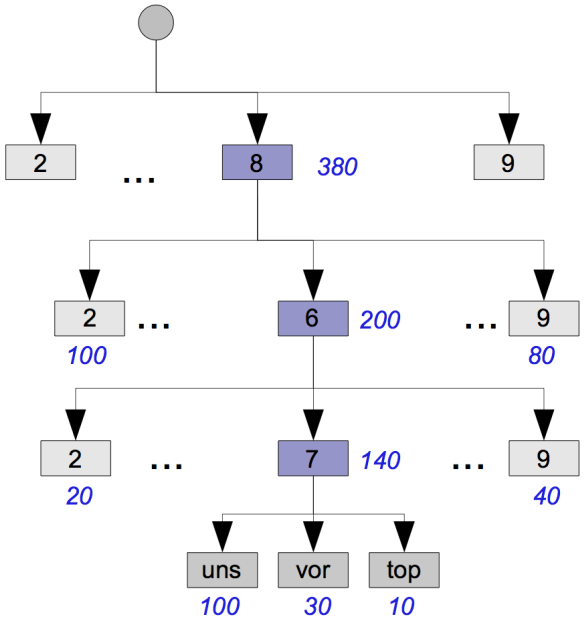

Abbildung 3: *Die Häufigkeit einer Zeichenfolge ist als Attribut im jeweiligen Blatt angegeben. Die Häufigkeit eines inneren Knotens entspricht der Summe der Häufigkeiten aller Kinder.*

Um nun aus einer eingegebenen Zahlenfolge zur darzustellenden Zeichenkette zu gelangen, müsste vom aktuellen Knoten aus der häufigste Weg zu einem Blattknoten gefunden werden. Für Knoten 8 in obigem Beispiel wäre Knoten 6 das „Kind mit größter Häufigkeit“, für Knoten 6 wäre es Knoten 7, und für diesen schließlich „uns“.

Werden beispielsweise die Tasten „86“ gedrückt, so ist der Weg bis zu Knoten 6 in Abbildung 3 festgelegt. Die zurückzugebende Zeichenfolge ergibt sich dann aus der Suche nach dem häufigsten Blatt, ausgehend von Knoten 6.

In der Praxis wird die Datenstruktur leider etwas komplizierter wenn auch das Prinzip gleich bleibt: nicht nur Blätter, sondern auch innere Knoten können „Zeichenknoten“ wie „uns“ als Kinder haben. Beispielsweise beginnen auch die Wörter „unser“, „unsere“ und „unselig“ mit „uns“, in diesem Fall müsste der T9-Baum aussehen wie hier:

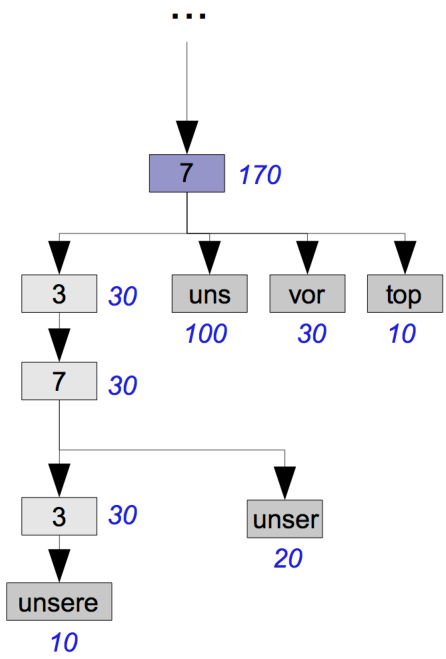

Abbildung 4: *Ausschnitt eines T9-Baums mit gleich beginnenden Wörtern unterschiedlicher Länge.*

Dies macht die Implementation jedoch nicht schwieriger, sondern eigentlich sogar leichter, da die Datenstrukturen für innere Knoten und Blätter sich nun nicht mehr unterscheiden müssen: Jeder Knoten kann sowohl Strings als auch weitere Knoten als Kinder haben.

3.2 Konstruktion der Datenstruktur

Nachdem der T9-Baum beschrieben wurde, soll nun eine Methode vorgeschlagen werden, mit der dieser konstruiert werden kann. Der Einfachheit halber soll angenommen werden, dass die Wörter, aus denen das Lexikon bestehen soll, bereits in einer Liste vorliegen.

Der Aufbau des Baums funktioniert „umgekehrt“ zur Suche in ihm: Ein einzufügendes Wort wird in seine Zahlenrepräsentation zerlegt – aus „uns“ wird (wieder einmal) 867. Ausgehend von der Wurzel wird nun dieser Pfad „abgegangen“, d.h. es wird zunächst zu Kind 8 der Wurzel gegangen, anschließend zu Kind 6 von Knoten 8 usw. Knoten, die noch nicht existieren (zu Beginn ist der Baum natürlich leer bzw. enthält nur die Wurzel), müssen dabei angelegt werden.

Bei jedem besuchten Knoten wird der Häufigkeitszähler um 1 erhöht – so werden sich die Häufigkeiten der Knoten ohne weiteres Zutun ergeben. Ist der Weg, der sich aus dem einzufügenden Wort ergeben hat abgegangen, so wird das Wort selber als Kindknoten des zuletzt besuchten Knotens eingefügt. Abbildung 5 zeigt, wie das Wort „uns“ in den leeren Baum eingefügt wird:

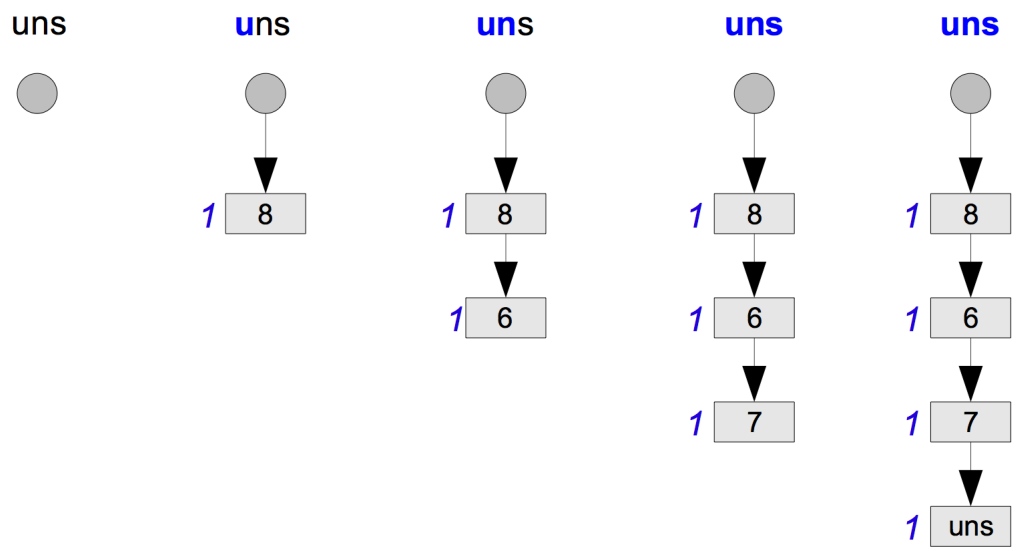

Abbildung 5: *Einfügen eines Wortes in einen leeren T9-Baum (von links nach rechts)*

Wird ein Wort ein weiteres Mal eingefügt, oder wird ein Wort eingefügt, dessen Zahlenrepräsentation (teilweise) bereits vorhanden ist, so werden die vorhandenen Knoten benutzt und hochgezählt:

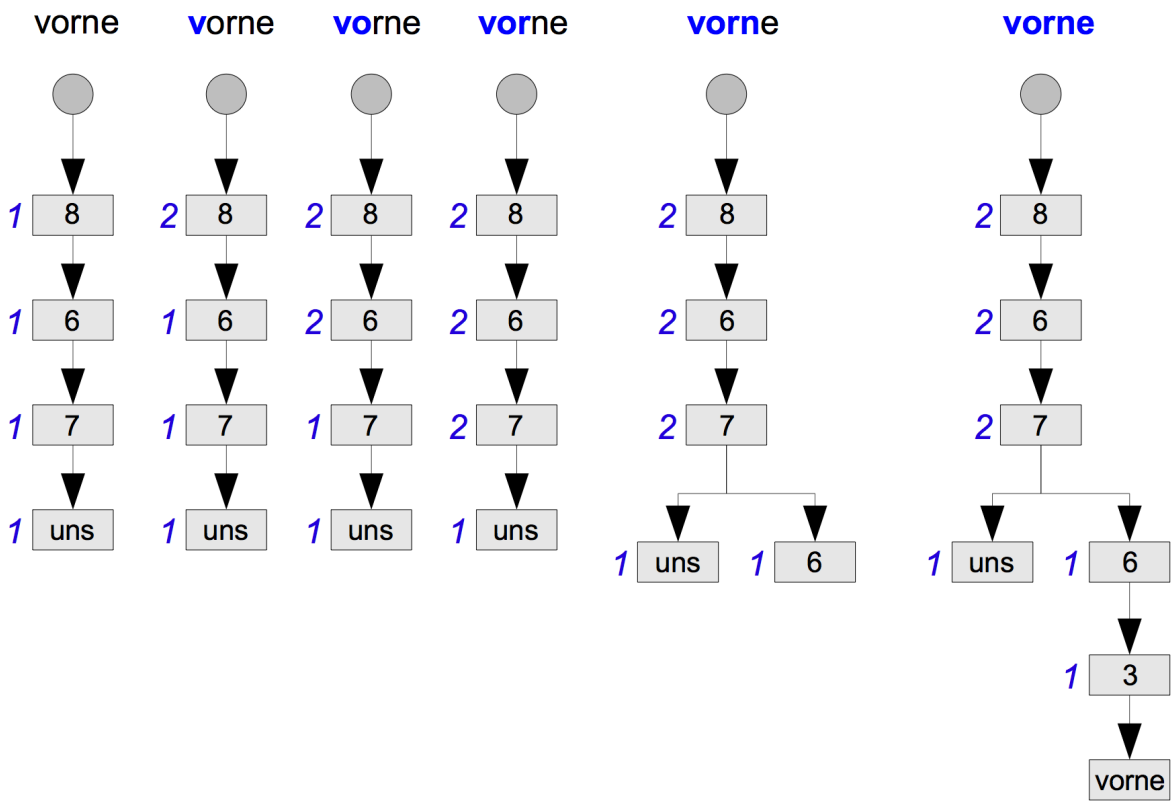

Abbildung 6: *Einfügen eines Wortes in einen teilweise erzeugten T9-Baum*

Dieses Verfahren lässt sich optimieren: Liegt vorher bereits die Information über die Häufigkeit eines Wortes vor (die vermutlich benötigt wird, siehe Kapitel 5), so lässt sich der Aufbau des Baums beschleunigen, indem ein Wort nur einmal eingefügt wird und die Häufigkeit dabei nicht um 1, sondern um die Anzahl seiner Vorkommen hochgezählt wird.

3.3 Steuerung des Programms

Neben den 10 Zifferntasten des Telefons, die wohl nicht weiter erläutert werden müssen, gibt es mehrere weitere Tasten, von denen die ebenfalls durch Ihr Programm interpretiert werden müssen. Zum einen die Taste „löschen“, die den zuletzt eingegebenen Buchstaben entfernen soll, zum anderen die Tasten „*“ und „#“. Diese beiden Tasten sollen dem Programm folgende Funktionen hinzufügen:

- Umschalten zwischen Buchstaben (d.h. T9) oder Zahlenmodus. Im Zahlenmodus soll durch Betätigung der Tasten 0 – 9 lediglich die entsprechenden numerischen Werte eingegeben werden können, während im T9-Modus die „normale“ T9-Funktionalität unterstützt werden soll.
- Umschalten zwischen Groß/Kleinschreibung: Während normalerweise jedes eingegebene Wort kleingeschrieben wird, soll die Betätigung dieser Taste dazu führen, dass der nächste Buchstabe, der eingegeben wird, großgeschrieben wird. Selbstverständlich können Sie dieses Verhalten weiter verfeinern, und bspw. zwischen

unterschiedlichen Schreibmodi (nur Kleinbuchstaben, nächster Buchstabe groß, jeder Anfangsbuchstabe eines Wortes großgeschrieben, jeder Buchstabe großgeschrieben...) unterscheiden. Die Implementation der „löschen"-Funktion ist insofern schwierig, als dass theoretisch auch mehrere Wörter gelöscht werden können. In diesem Fall müsste die T9-Erkennung mit den zu löschenden Wörtern neu initialisiert werden – daher können Sie diesen Punkt ausfallen lassen und sich auf das Löschen von Buchstaben innerhalb eines Wortes beschränken.

3.4 Suche nach Alternativen

Es bleibt die Frage, wie sich bspw. aus dem Pfad 867 die möglichen Wörter extrahieren, und wie sich diese nach der Häufigkeit sortieren ließen, so dass als erste Variante das wahrscheinlichste Wort vorgeschlagen wird, anschließend das mit der zweitgrößten Frequenz usw. Es wäre hilfreich, wenn ein Knoten *k* am Ende eines jeden Zahlen-Pfades „wissen" würde, welches der gespeicherten Worte am häufigsten vorkommt. Wie ließe sich dies ermöglichen?

(Mindestens) drei Varianten sind denkbar:

(1) Jeder Knoten verwaltet zusätzlich eine Liste, in der die möglichen Wörter nach Häufigkeit sortiert enthalten sind. Als erste Möglichkeit wird das erste Wort vorgeschlagen, als 2. Möglichkeit das 2. usw. **Vorteil**: Die Suche nach einer Alternative ist ohne Probleme möglich. **Nachteil**: Sehr viele Informationen werden mehrfach gespeichert, der Speicherbedarf nimmt stark zu.
(2) Ausgehend vom aktuellen Knoten wird der Weg zu dem Blatt gesucht, der das häufigste Wort repräsentiert. Dieses Blatt entspricht dem wahrscheinlichsten Wort. Wird nach einer Alternative gesucht, so muss ein Weg zum nächst häufigen Blatt gesucht werden. **Vorteil**: Der Speicherbedarf bleibt in etwa gleich. **Nachteil**: Die Suche nach den Alternativen ist aufwändiger.
(3) Eine Kombination der Verfahren aus (1) und (2) wird genutzt: In Ebenen, die nahe zur Wurzel sind (und in denen es aufgrund der Eigenschaften des Baumes nicht viele Knoten gibt), werden wie in (1) verwaltet, während Knoten, die weiter unten im Baum liegen, mit der Methode aus (2) durchsucht werden.

Entscheiden Sie sich für eine Variante (oder überlegen Sie sich eine eigene), und begründen Sie Ihre Entscheidung.

Eine sehr elegante Lösung wäre die Implementation eines eigenen Iterators, um Alternativen zurückgeben zu können: Wird die Methode `getAlternative()` des T12Interpreters aufgerufen, so könnte in dieser die Rückgabe eines Iterators weitergeleitet werden. Die Kopplung zwischen der (relativ komplizierten) Funktionalität der Methode wäre so hinter einer sehr einfach zu benutzenden Schnittstelle verborgen.

4 Informationen zur Vorlage

In diesem Kapitel wird der Inhalt des Vorlageprojekts „T12“ vorgestellt. Das Projekt enthält folgende Verzeichnisse:

`src/`	Enthält den Quellcode der Vorlage. Fügen Sie hier eigene Klassen hinzu.
`lib/`	JavaDoc zur Vorlage. Das beiliegende Ant-Skript javadoc.xml speichert JavaDoc immer in diesem Verzeichnis ab, auch die Kommentare zu Ihren Klassen.
`doc/`	Die Bibliotheken, die von der Vorlage verwendet werden.
`Data/`	Die Texte, die von Ihrem Programm verarbeitet werden sollen.
`.settings/`	Verschiedene Einstellungen von Eclipse.

sowie die Dateien

`javadoc.xml`	Das Ant-Skript, das JavaDoc zu den Klassen im src- generiert.
`phone-settings.conf`	Einstellungen des „Telefons“.
`.project`	Vorgefertigte Einstellungen des T12-Projekts.
`.classpath`	Vorgefertigte Abhängigkeiten zu den Bibliotheken in „lib/“

4.1 Inversion of Control und die grafische Benutzeroberfläche

In diesem Kapitel soll erläutert werden, wie die Benutzeroberfläche des Telefons mit Ihrem Programm interagieren kann. Dies geschieht letztlich über ein Interface namens `T12Interpreter`, aber da die Reihenfolge der Methodenaufrufe in diesem Interface nicht

ersichtlich ist (und auch nicht sein kann), soll hier das Konzept zunächst einmal vorgestellt werden.

Bisher haben Sie meist Programme geschrieben, in denen von Ihrer main-Methode ausgehend ein Programm abgearbeitet wurde. Dies ist jedoch für Programme, die mit dem Benutzer interagieren, nicht immer möglich, da unklar ist, was der Benutzer als nächstes machen wird, und es ineffizient wäre, bspw. in einer Schleife auf Benutzereingaben zu warten, da dadurch das komplette Programm blockiert würde. Um dieses Dilemma zu lösen, werden Benutzereingaben in Java, wie bspw. die Betätigung eines Buttons, als „Nachrichten“ verschickt, die von speziellen „Empfängern“ interpretiert werden. In der Hausarbeit wird dieses Konzept benutzt, jedoch müssen Sie sich um die Details des Versendens und Empfangens nicht weiter kümmern, da dies bereits implementiert ist. Wird bspw. eine Zahlentaste (0 bis 9) gedrückt, so wird dieses „Ereignis“ innerhalb der GUI abgefangen und durch den Aufruf einer Methode aus Ihrem Programm weitergeleitet - Ihrem Programm wird sozusagen in einigen Bereichen die Kontrolle entzogen, und es muss nicht mehr selbst aktiv werden, sondern „auf Anregung von außen“ nur noch reagieren. Dieses Prinzip, auch als „Hollywood-Prinzip“ bezeichnet („don't call us, we call you“), nennt sich *Inversion of Control*.

Letztlich ist es einfacher, als es sich vielleicht anhören mag: Sie müssen das Interface `T12Interpreter` implementieren und mit der GUI des Handys „bekannt machen“, und schon funktioniert Ihr Programm mit dem *Inversion of Control*-Prinzip: Wird anschließend z.B. die Taste „3“ gedrückt, so wird in dem `T12Interpreter` die Methode `public void buttonPressed(int number)` aufgerufen, und Sie können in dieser Methode nun das wahrscheinlichste Wort suchen lassen. Wird hingegen die Taste „alternative“ gedrückt, um eine Alternative zum aktuellen Wort zu finden, so ruft die GUI die Methode `getAlternative()` des Interfaces `T12Interpreter` auf, und Sie können die Alternative zurückgeben. Abbildung 7 soll veranschaulichen, welche Aktion des Benutzers mit welcher Methode des Interfaces verbunden ist. Zudem können Sie mit der vorliegenden Klasse `ExampleT12Interpreter` selbst testen, welche Methoden wann aufgerufen werden.

Ein Nachteil des *Inversion of Control*-Prinzips sind die Stacktrace-Ausgaben falls Fehler auftriten, denn diese „beginnen“ nun in einer Ihnen unbekannten Klasse. Sie können jedoch die Methoden, die nicht von Ihnen stammen, ignorieren.

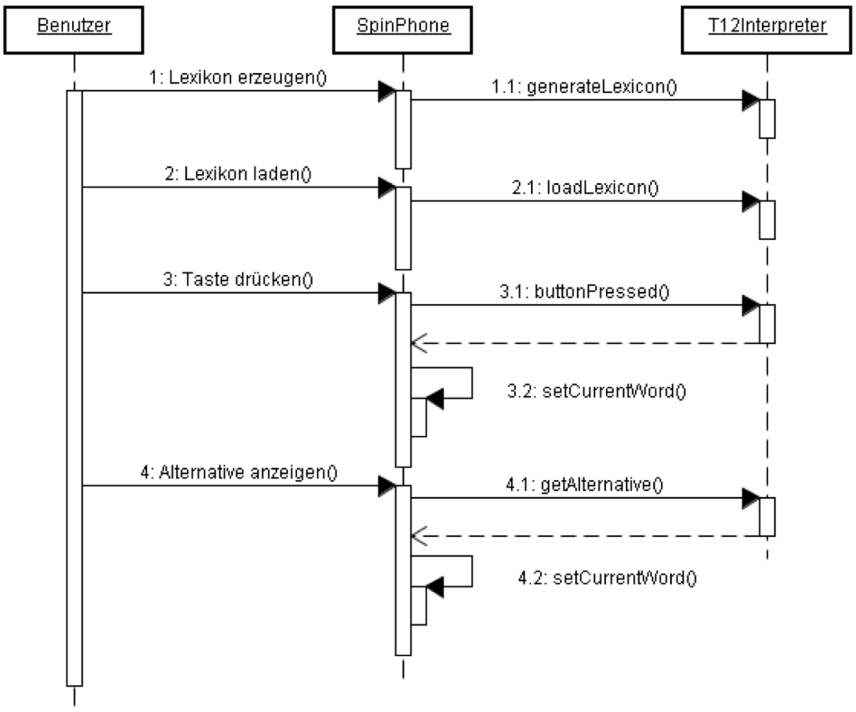

Abbildung 7: *Sequenzdiagramm, das die Interaktion von Benutzer und T12Interpreter veranschaulicht. Bei Betätigung von Buttons werden von der SpinPhone-Instanz die dazugehörigen Methoden des T12-Interpreters aufgerufen und ggf. die GUI aktualisiert.*

4.2 GUI vs. Console

Ein Nachteil an der Benutzeroberfläche ist das Debuggen des Programms. Angenommen, Ihr Programm ist bereits in der Lage dazu aus Zahlenfolgen Wörter zu generieren, hat aber Probleme mit dem „ß“ und stürzt aus unerklärlichen Gründen ab. Um solche Probleme aufzuspüren, müssten Sie jedes Mal das Programm starten, das Lexikon laden und ein Wort eintippen, was nicht besonders komfortabel ist. Ähnlich verhält es sich bei Problemen beim Lernen von neuen Wörtern usw.

Aus diesem Grund gibt es eine zweite Möglichkeit, das Programm laufen zu lassen: Mit Hilfe der Klasse `ConsolePhone`. Dies ist – wie der Name vermuten lässt – eine reine Konsolenanwendung, die Methoden anbietet, mit denen Benutzereingaben simuliert werden können. So könnten Sie beispielsweise durch die Zeilen

```
phone.createLexicon("data", "SpinPhone.lex");
phone.loadLexicon("SpinPhone.lex");
phone.typeAsNumbers("vor");
phone.displayAlternative();
```

ein Lexikon anlegen und laden, die Zahlenkodierung von „vor“ (867) eintippen und anschließend eine Alternative dazu suchen lassen (z.B. „uns“), ohne dass weitere Eingaben von Ihnen erforderlich sind.

Um zwischen den beiden Telefon-Varianten zu wechseln, genügt es, in der main-Methode der Klasse `Application` die jeweilige Methode ein- bzw. auszukommentieren. In der Methode `useConsolePhone()` können Sie dann die Befehle an das Telefon senden, die Sie testen möchten.

4.3 JUnit-Tests

Zusätzlich zur „normalen" Anwendung liegt die Klasse `PhoneTests` mit vorbereiteten JUnit-Tests im Package `t12.tests`. Die JUnit-Tests müssen bestehen. Die Tests lassen die Funktionalität Ihres Programms besser überprüfen und können Fehlfunktionen, die sich evtl. „eingeschlichen" haben, schnell aufdecken.

4.4 JavaDoc: Vorlage und Anforderung

Die Klassen der Vorlage sind vollständig (mit Ausnahme der GUI-Klasse `SettingsFrame`, die den Einstellungsdialog des Programms implementiert) mit JavaDoc kommentiert. Öffnen Sie die Datei „index.html" im Verzeichnis „doc" (ziehen Sie sie in einen Browser oder wählen Sie den Punkt *„Open With > System Editor*" im Kontextmenu), um zur Übersicht über die Klassen im Projekt zu gelangen.

Sie müssen die von Ihnen angelegten Klassen ebenfalls dokumentieren. Evtl. ist es jedoch nicht notwendig, wirklich jede einzelne Methode zu kommentieren – beachten Sie jedoch folgende Anforderungen:

- jede Methode und jede Variable, die nicht als „`private`" markiert ist, **muss** dokumentiert werden, inkl. der Parameter (@param), Rückgabewerte (@returns) oder Exceptions (`@throws`).
- Methoden mit geringerer Sichtbarkeit **müssen** dokumentiert werden, wenn sich die Funktionalität nicht ohne weiteres erschließt, oder wenn bestimmte Anforderungen an die Parameter gestellt werden (X darf nicht null sein, String darf keine Leerzeichen enthalten o.ä.). Dies bedeutet umgekehrt, dass Sie umso weniger kommentieren müssen, je mehr Methoden und Variablen Sie als „`private`" deklarieren, und je besser Sie die Namen der Methoden/Variablen wählen. „`String s`" muss in jedem Fall dokumentiert werden „`String currentWord`" hingegen nicht unbedingt.
- Methoden **sollten** das JavaDoc-Attribut `@see` verwenden, wenn sich der Zweck einer Methode nur in einem „größeren Kontext" ergibt, bzw. wenn Ihnen dieses Attribut zweckmäßig erscheint. So wird in der Dokumentation der Vorlage bspw. aus der JavaDoc der Klasse `ConsolePhone` in die JavaDoc des Interfaces `T12Interpreter` verwiesen, wenn `ConsolePhone` Methoden aus `T12Interpreter` aufruft. Mehr als 2 `@see`-Attribute pro Kommentar sollten Sie jedoch nicht benötigen, entscheiden Sie sich im Zweifelsfall für die Ihrer Meinung nach wichtigsten Referenzen.

Im Projektverzeichnis befindet sich eine Datei namens „javadoc.xml". Dies ist eine „Bauanleitung" für das Java-Werkzeug „ant", das die JavaDocs des Projekts automatisch neu erzeugen kann. Wählen Sie die Datei im Package Explorer aus und rufen Sie im Kontextmenu

den Punkt *„Run As... > Ant Build“* aus. In der Console von Eclipse sollte eine Ausgabe wie diese erscheinen:

```
Buildfile: E:\eclipse\workspace\T12\javadoc.xml
javadoc:
  [javadoc] Generating Javadoc
  [javadoc] Javadoc execution
  [javadoc] Loading source files for package t12...
...

  [javadoc] E:\eclipse\workspace\T12\src\t12\tests\PhoneTests.java:10:
                        package junit.framework does not exist
  [javadoc] import junit.framework.TestCase;
  [javadoc] ^
...

  [javadoc] 2 warnings
BUILD SUCCESSFUL
Total time: 10 seconds
```

Die Warnungen bzgl. JUnit können Sie dabei ignorieren. Anschließend ist die Dokumentation im Verzeichnis „doc“ aktualisiert, neue JavaDoc-Kommentare sollten nun ebenso wie neue von Ihnen erzeugte und kommentierte Klassen auch in den HTML-Seiten erscheinen.

5 Korpus

Die Textdateien, die im Projektverzeichnis „data/“ zu finden sind, sollen benutzt werden, um das T12-Lexikon aufzubauen. Bei diesen Texten handelt es sich um

1. ein „SMS-Korpus“, genauer um eine Sammlung von ca. 1.500 SMS[2], die lediglich manuell etwas aufbereitet wurden
2. ein„Chat-Korpus“, welches Protokolle verschiedener Internet-Chats enthält
3. mehrere „Stoppwort“-Listen[3], d.h. Listen mit den häufigsten Wörtern des Deutschen (die i.d.R. von Suchmaschinen nicht indiziert werden, weil sie in fast jedem Text vorkommen und daher kein gutes Suchkriterium sind).

Dieses Korpus wurde unter der Annahme zusammengestellt, dass die häufigsten Begriffe, die in SMS gebraucht werden, dort ebenfalls häufig vorkommen, so dass das erzeugte Lexikon realistischer ist als bspw. ein Lexikon auf Basis frei verfügbarer Texte des Projekts Gutenberg[4]. Das Korpus hat jedoch auch den Nachteil, dass es relativ „unsauber“ ist: Oftmals werden Abkürzungen verwendet, teilweise fehlen Leerzeichen, wie folgende Beispiele aus dem SMS-Korpus zeigen:

---

[2] Das Originalkorpus findet sich unter http://www.mediensprache.net/archiv/corpora/sms_os_h.pdf, das Copyright des Korpus liegt bei www.mediensprache.net.

[3] Für weitere Informationen (für die Hausarbeit irrelevant) siehe auch: http://de.wikipedia.org/wiki/Stoppwort

[4] Link: http://gutenberg.spiegel.de

(1) Telefon **kaput** nur per Handy zu erreichen
(2) Hast du meinen Brief schon bekommen? Also bis heute Abend, **HDGGGGGGGGGGGGGGGGGGGGGGGGGGGGDL**! * Jojo *
(3) Hey*Wo+seid*ihr*grad*was*macht*ihr*heute*abend*hab*kein*Bock*zum*Strand*zu *gehen*H*E*L
(4) **RosenFürEinenLiebenMenschenDenIchNieMehrVerlierenMag! Riesenknutsch** vom Knüti
(5) **IP383!T H)IP TT!M H)!P 6VW H)! -->** stell dein kaputtes Handy auf den Kopf und lies mal!
(6) HI NADINE!HAST DU **EIGENTL**:FRAU HÖRMANNS **HANDYNR**?**KANNSTE MJIR** MAL **SCHICKN**?VIELE GRÜßE VON MAIKE

Nicht jedes „Wort“ des Korpus sollte daher in das Lexikon eingefügt werden, vielmehr muss jedes Wort zunächst untersucht werden. Grundsätzlich gilt:

- Zahlen und Satzzeichen sollten nie in das Lexikon aufgenommen werden, nur Buchstaben, die sich über die Tastatur der Handy-GUI eingeben lassen, sollten übernommen werden. Alle anderen Wörter können ignoriert werden.
- Satz- und Sonderzeichen sollten als Leerzeichen interpretiert werden, um Sätze wie (3) korrekt analysieren zu können.
- Wörter sollten eine „geeignete“ Länge haben: Mindestens X Buchstaben, höchstens Y Buchstaben. So kann verhindert werden, Wörter wie in (4) zu generieren. Überlegen Sie sich sinnvolle Werte für X und Y, und begründen Sie Ihre Entscheidung.
- Wörter sollten grundsätzlich nur in Kleinschreibung gespeichert werden, d.h. spätestens vor dem Einfügen eines `Strings s` in das Lexikon muss `s = s.toLowerCase()` aufgerufen werden.
- Wörter, die „zu selten“ vorkommen, sollten ignoriert werden: So kann der Speicherbedarf des Programms reduziert werden, zudem besteht die Hoffnung, dass Rechtschreibfehler wie in (1) nur selten gemacht werden und somit nicht ins Lexikon aufgenommen werden. Entscheiden Sie selbst, was „zu selten“ ist, und begründen Sie Ihre Entscheidung. Weitere Ausschlusskriterien sind denkbar, beispielsweise existieren in der deutschen Sprache keine Wörter, in denen ein Buchstabe mehr als drei Mal hintereinander auftritt. So könnten „Wörter“ wie in (2) erkannt werden. Tipp: Es empfiehlt sich, das Korpus zunächst zu analysieren, um obige Entscheidungen möglichst gut treffen zu können. Wenn Sie den Crawler programmiert haben (siehe Kapitel 5.1), sollte es nicht schwer sein, das Programm Wortzähler so anzupassen (oder analog zu dem Programm ein Eigenes zu schreiben), dass Sie sich alle Wörter des Korpus anzeigen lassen und diese nach Länge oder Häufigkeit sortieren lassen. Damit diese Analyse korrekt ist, müssen jedoch auch hier alle Wörter in Kleinschreibung und ohne Satz- und Sonderzeichen betrachtet werden!

5.1 Einlesen des Korpus

Die Texte des Korpus sind im Verzeichnis „data“ innerhalb des Eclipse-Projekts in 16 verschiedenen Verzeichnissen und Unterverzeichnissen abgelegt. Ihr Programm muss dazu in

der Lage sein, alle .txt-Dateien im Verzeichnis „data“ und in allen Unterverzeichnissen zu finden und in das Lexikon einzufügen – dafür ist es vermutlich notwendig, dass Sie einen „Crawler“ programmieren, der rekursiv durch alle Verzeichnisse läuft und dabei die Texte einsammelt und einliest, ähnlich, wie Sie es in einer Hausaufgabe bereits einmal gemacht haben.

6 Speicherung des T9-Baums

Sie dürfen bzw. müssen den Baum mit Hilfe eines `ObjectOutputStream` speichern und mittels `ObjectInputStream` laden. Dafür müssen jedoch alle Klassen, auf die innerhalb des Baums verwiesen wird, das Marker-Interface `java.io.Serializable` implementieren. Falls Sie hier unsicher sind, sehen Sie sich noch einmal das Beispiel aus dem Kurs an.

7 Sonstiges

7.1 „Problemlösungs-Strategie“

Auch wenn es vielleicht etwas trivial klingt: Überlegen Sie, wie Sie ein Problem lösen können, ruhig auch ohne Computer, sondern mit Zettel und Stift. Versuchen Sie, ein Problem zunächst grob zu umschreiben und versuchen Sie dann, einzelne Teilprobleme darin ausfindig zu machen. Dann können Sie diese Strategie rekursiv auf die Teilprobleme anwenden, bis Sie schließlich feststellen, dass Sie nur eine relativ große Menge von einfachen Problemen lösen müssen.

Die Aufgabe „Texte einlesen“ stellt ein recht abgeschlossenes Teilproblem dar - es bietet sich also an, dass Sie mit der Implementation des „Crawlers“ beginnen. Vor der Implementation des T9-Baums, also der „eigentlichen“ bzw. vermutlich aufwändigsten Aufgabe, sollten Sie sich Gedanken über die Art der zu nutzenden Datenstrukturen machen. Bedenken Sie bei der Konstruktion der Datenstruktur, welche Anforderungen diese erfüllen muss, damit sich nicht plötzlich feststellen, dass manche Funktionen nur äußerst schwierig zu implementieren sind.

7.2 Design

Es ist kein guter Stil, alle Funktionalität in einer Klasse zu implementieren, auch wenn das Interface `T12Interpreter` dies suggerieren mag. Klassen, die mehr als 300 Zeilen Code enthalten, sollten vermieden werden – stattdessen bietet es sich an, Aufgaben aufzuteilen und an Hilfsobjekte zu delegieren.

Nutzen Sie außerdem die Möglichkeit, eigene Pakete anzulegen und so Ihr Programm weiter zu strukturieren. Beispielsweise bietet sich vermutlich ein eigenes Paket für die Datenstruktur an, in dem Klassen für Baum und Knoten (und evtl. weitere Hilfsklassen?) abgelegt werden.

7.3 Externe Dokumentation

Die externe Dokumentation soll dazu dienen, Ihre Lösungsstrategie zu erläutern und das

Zusammenspiel der von Ihnen implementierten Klassen zu erklären sowie Probleme, Alternativen und Verbesserungsmöglichkeiten zu diskutieren. UML-Klassen- oder Sequenzdiagramme sind dabei hilfreich, jedoch nicht erforderlich. Die in diesem Dokument hervorgehobenen Punkte („begründen Sie Ihre Entscheidung“) müssen in der externen Dokumentation erwähnt werden. Der Umfang der Dokumentation soll in etwa 10 Seiten betragen, bei deren Formatierung die Anleitung unter Hinweise zur Form der wissenschaftlichen Arbeit verbindlich ist. Dokumentationen die es nicht ermöglichen Kommentare einzufügen (Ränder zu schmal, kein 1,5-facher Zeilenabstand) werden **nicht akzeptiert**! Dies ist jedoch vermutlich auch in Ihrem Interesse, denn so kommen Sie schneller auf die gewünschte Seitenzahl.

7.4 Abgabe

Die Arbeit muss bis spätestens **Donnerstag, 14.09.2017, 12 Uhr** in der Sprachlichen Informationsverarbeitung (Philosophikum, **Raum 3.206 im 3. OG**) abgegeben werden.

Achtung: Um sichergehen zu können, dass Sie nicht vor verschlossenen Türen stehen, sollten Sie zur Abgabe nur zu den Geschäftszeiten kommen (10-12 Uhr)!

Das Programm muss in einem lauffähigen Zustand vorliegen, d.h. es muss (a) ausführbar sein und (b) die gestellten Anforderungen „weitestgehend“ erfüllen, andernfalls gilt die Arbeit als nicht bestanden. Sollten einige der weniger relevanten Anforderungen nicht erfüllt sein, so werden Sie (per Email! Bitte auf dem Deckblatt der Dokumentation angeben) aufgefordert, die entsprechenden Punkte nachzubessern.

7.5 Noch Fragen?

Falls sich noch Fragen ergeben sollten können Sie diese gerne per E-Mail an Mihail Atanassov (matanass@uni-koeln.de) oder an Börge Kiss (bkiss0@uni-koeln.de) zusenden.

VIEL ERFOLG ☺ !!!