Abgabetermin: SA in der KW 12

☐ **Gr. 1, E. Pitzer** **Name** _____ **Aufwand in h** _____

☐ **Gr. 2, F. Gruber-Leitner**

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

T9 (2 + 4 + 4 + 4 + 4 + 6 Punkte)

Der T9-Code (*Text on 9 keys*, siehe *http://de.wikipedia.org/wiki/Text_on_9_keys*) bildet das Leerzeichen und die 26 Buchstaben des Alphabets auf die neun Dezimalziffern 0 sowie 2 bis 9 (z. B. auf den Tasten eines Mobiltelefons) ab, und zwar gemäß folgender Tabelle:

Buchstaben	*Ziffer*
a, b, c	2
d, e, f	3
g, h, i	4
j, k, l	5
m, n, o	6
p, q, r, s	7
t, u, v	8
w, x, y, z	9

(aus *http://de.wikipedia.org/)*

Damit ist eine eindeutige Abbildung von Wörtern auf ganze Zahlen möglich (z. B. wird das Wort "kiss" auf die Zahl 5477 abgebildet).

Nutzbringend (als Eingabehilfe) verwendet wird dieser Code gerade für die Abbildung in der entgegengesetzten Richtung: Eine ganze Zahl (also eine Ziffernfolge, die z. B. über die Handy-Tastatur eingegeben wird) entspricht nun allerdings mehreren Zeichenketten, wobei aber nur wenige davon tatsächlich Wörter sind (z. B. kann die Zahl 5477 nicht nur auf das Wort "kiss" abgebildet werden, sondern sinnigerweise auch auf das Wort "lips", aber auch auf das Wort "lisp" und viele weitere, leider unsinnige Zeichenketten wie z. B. "lhqr").

Für eine Ziffernfolge der Länge n gibt es mindestens 3^n und höchstens 4^n Zeichenketten, in welche diese Ziffernfolge abgebildet werden kann, je nachdem, welche Ziffern darin vorkommen – nur sehr wenige dieser Zeichenketten sind Wörter.

a) Entwickeln Sie eine Funktion `word2number()`, die ein Wort mittels iTap/T9 auf eine Ziffernfolge abbildet. Um längere Wörter und die sich daraus ergebenden sehr großen Zahlen darstellen zu können, sollen auch die Ziffernfolgen in Form von Zeichenketten dargestellt werden (z. B. soll `word2number("kiss")` die Zeichenkette `"5477"` liefern).

b) Entwickeln Sie eine Funktion `number2strings()`, die für eine Ziffernfolge (in Form einer Zeichenkette) als Funktionsergebnis in einem passenden Behälter alle Zeichenketten liefert, die sich aus der Ziffernfolge bilden lassen. Z. B sollte `number2strings("5477")` insgesamt 144 Zeichenketten von `"jgpp"` bis `"liss"` liefern.

Die Funktion aus b) ist als Eingabehilfe noch nicht geeignet, da sie – wie schon erwähnt – auch viele unsinnige Zeichenketten liefert. Hat man allerdings ein Wörterbuch zur Verfügung, z. B. jenes, das Sie unter http://tug.ctan.org/tex-archive/systems/win32/winedt/dict/de_neu.zip herunterladen können, das über 500.000 deutsche enthält, kann man mit dessen Hilfe "die Spreu vom Weizen trennen".

c) Entwickeln Sie eine Funktion `number2words()`, die für eine Ziffernfolge nur mehr solche Zeichenketten liefert, die sich auch im Wörterbuch befinden, bei denen es sich also um "echte Wörter" handelt. Gehen Sie dabei so vor, dass Sie die Wörterbuchdatei beim Programmstart in einen geeigneten Behälter einlesen, und dass Sie dann jede Zeichenkette, die Sie erzeugen im Wörterbuch suchen: wird sie gefunden, handelt es sich um ein Wort und kann in den Ergebnisbehälter aufgenommen werden. Wie beurteilen Sie die Effizienz dieses Verfahrens?

d) Eine weitere Möglichkeit dieses Problem zu lösen, besteht darin, die Wörter im Wörterbuch primär nach ihrer Länge zu betrachten: Wählen Sie einen Behälter, der es gestattet, für alle Wortlängen *n* die Wörter dieser Länge so zu speichern, dass man in einer neuen Version Ihrer Funktion `number2wordsByLength()` aufgrund der Länge der übergebenen Ziffernfolge die Wortsuche auf Wörter mit der passenden Länge einschränken kann. Die Ergebnisse sollten gegenüber c) unverändert bleiben, aber wie beurteilen Sie die Effizienz dieses neuen Verfahrens?

e) Eine noch viel nützlichere Eingabehilfe bestünde darin, dass für eine Ziffernfolge (die noch kein vollständiges Wort darstellen muss) nicht nur jene Wörter berechnet werden, die in der Länge genau dazu passen, sondern auch all jene Wörter, die sich zukünftig (durch Verlängerung der aktuellen Ziffernfolge) daraus ergeben könnten, für welche die Ziffernfolge also ein gültiges Präfix darstellt. Entwickeln Sie eine neue Funktion `numberPrefix2word()`, die für eine Ziffernfolge alle Wörter mit passendem Präfix liefert. So soll z. B. für die Ziffernfolge `"4355"` nicht nur (wie schon bisher) das in der Länge genau passende Wort "hell" sondern auch die längeren Wörter wie "hellen" und "hello" geliefert werden. Überlegen Sie gut, ob der/die bisherige/n Behälter für das Wörterbuch auch für diese Aufgabe (noch) geeignet ist/sind und verwenden Sie ev. einen anderen. Wie beurteilen Sie die Effizienz dieses letzten Verfahrens?

f) Die Benutzerfreundlichkeit lässt sich noch weiter steigern, indem Sie die zu einem Präfix gehörende Wortmenge nach der Verwendungshäufigkeit sortieren. Erweitern Sie ihr Wörterbuch um eine Möglichkeit, die Häufigkeit der einzelnen Wörter zu speichern und initialisieren Sie das Wörterbuch einerseits aus der angegeben Quelle und andererseits, indem Sie aus einem repräsentativen Text – der mindestens 20.000 Wörter umfassen sollte und idealerweise in der gleichen Sprache wie das Wörterbuch verfasst wurde – die Worthäufigkeiten ermitteln. Entwickeln Sie dazu eine weiter Funktion `numberPrefix2sortedWords()`, das für einen Präfix dieselbe Wortmenge wie `numberPrefix2word()` ermittelt, die Wortmenge aber nach der Worthäufigkeit absteigend sortiert zurückgibt.

Ein Tipp, um Ihre Nerven (und Ihren Rechner) zu schonen:

Nutzen Sie die STL möglichst gut aus und testen Sie Ihr Programm zu Beginn eventuell mit einer selbst kleinen Wörterbuchdatei (z.B. 100 Wörter), so dass Sie Fehler leichter lokalisieren können. Verwenden Sie die Wörterbuchdatei mit den über 500.000 Einträgen erst wenn Ihr Programm "halbwegs" funktioniert.

1 T9 2
1.1 Lösungsidee 2
1.1.1 word2number 2
1.1.2 number2strings 2
1.1.3 number2words 3
1.1.4 number2wordsByLength 3
1.1.5 number2PrefixWord 3
1.1.6 numberPrefix2sortedWords 4
1.2 Implementierung 5
1.2.1 T9Converter.h 5
1.2.2 T9Converter.cpp 8
1.2.3 T9Tests.h 14
1.2.4 T9Tests.ccpp 15
1.2.5 testSuiteRunner.cpp 17
1.3 Tests 18
1.3.1 word2number 18
1.3.2 number2strings 18
1.3.3 number2words 20
1.3.4 number2wordsByLength 20
1.3.5 numberPrefix2Words 21
1.3.6 numberPrefix2sortedWords 22

1 T9

Folgend ist die Dokumentation der Aufgabe T9 Converter angeführt, welcher Strings in T9 Codes und visa versa konvertiert.

1.1 Lösungsidee

Folgend ist die Lösungsidee für die Aufgabenstellungen des T9 Konverters angeführt.
Alle Methoden sollen in einer Klasse gekapselt werden (T9Converter). Wiederverwertbare Logik soll in eigene private Methoden ausgelagert werden. Die Klasse soll Konstruktoren zur Verfügung stellen die eine Pfadangabe für das Wörterbuch und den Beispieltext verlangen.
Im Konstruktor soll das Wörterbuch und der Beispieltext einmalig verarbeitet werden.

Da die Aufgabenstellungen mehrerer Indexe verlangen, sollen diese als private Member der Klasse T9Converter definiert werden.

Die Tests sollen wieder mit CUTE und BOOST erfolgen.

1.1.1 word2number

Bei diesem Algorithmus soll ein String in seine T9 Code Darstellung konvertiert werden.
Dies sollte einfach sein, da der Source auch ein Mapping für die T9 Codes haben soll, wo die einzelnen Ziffern auf ihre T9 Codes abgebildet sind.
Ungültige Zeichen sollen erkannt, eine Meldung auf die Konsole ausgegeben und in dem weiteren Verlauf ignoriert werden.

1.1.2 number2strings

Bei dieser Implementierung sollen alle möglichen Kombinationen eines T9 Codes ermittelt und ausgeben werden. Hierbei soll wie folgt vorgegangen werden.

1. Ermittle alle Zeichen die verwendet werden (einzelne T9 Code <-> Zeichen Mappings)
2. Initialisiere ein temporäres Set mit einem Element (Lerrstring)
3. Iteriere über das ermittelten Zeichen Set und concatiniere alle Strings des temporären Sets mit den Zeichen des Sets
4. Ermittle ob sich bereits Strings mit adäquater Länge im temporären Set befinden
    a. JA: Füge sie in das Result Set hinzu
    b. NEIN: Ignoriere diese Strings
5. Fahre damit fort solange es noch Zeichen Sets gibt.

Mit diesem Algorithmus werden alle Kombinationen erfasst auch diese, die die nicht Strings mit adäquater Länge produzieren. Dies ist allerdings erforderlich um alle benötigten Permutationen zu erfassen.
Daher ist anzumerken dass dieser Algorithmus extrem ineffizient ist da er eine Laufzeitkomplexität von ungefähr O(n^4) besitzt. Da sich hier vier verschachtelte Schleifen befinden.

1.1.3 number2words

In dieser Methode soll der gegebene T9 Code in seine Worte übersetzt werden, die sich in einem Index (gebildet aus Wörterbuch) befinden. Daher soll ein Index in folgender Datenstruktur aufgebaut werden.

```
unordered_map<long long, unordered_set<string>>
```

Dieser Index wird die Zugriffszeiten erheblich verringern da keine Permutationen mehr berechnet werden müssen. Ein Nachteil ist aber das nur Worte unterstützt werden, die im Index bzw. Wörterbuch vorhanden sind, daher wird man auf den Umfang des Wörterbuches begrenzt sein.
Bei dem Aufbau des Index soll der T9-Code auf seine möglichen Wörter gemappt werden, wobei der Index nur T9-Codes unterstützen wird, die aus Wörtern des Wörterbuches gebildet werden können.

1.1.4 number2wordsByLength

In dieser Methode soll ein T9-Code in seine möglichen, vom Index unterstützten Worte übersetzt werden. Hierbei soll jedoch die Länge des T9 Codes berücksichtigt werden, die ebenfalls auch die Länge des Wortes vorgibt. Daher soll eine Datenstruktur aufgebaut werden, die die Längen des T9 Code (=Wortlänge) auf die möglichen T9 Codes mit den dazugehörigen Wörtern mapped.

```
unordered_map<int, unordered_map<long long, unordered_set<string>>>
```

Bei dieser Datenstruktur wird das Laufzeitverhalten erneut verbessert weil hier bereits die Länge des T9-Codes berücksichtigt wird und daher die Zugriffe beschleunigt werden, da eine gemappter Länge nur die T9 Codes und natürlich auch die dazugehörigen Wörter beinhaltet, die auf diese Länge passen. Dadurch wird ein Baum aufgebaut dessen Knoten die Längen sind, die wiederum einen Baum darstellen.
Der Aufbau des Index wird hierbei länger dauern aber wird das weitere Laufzeitverhalten verbessern.

1.1.5 number2PrefixWord

In dieser Methode soll ein T9-Code auf all seine möglichen Kombinationen die noch folgen könnten abgebildet werden. Also alle Wörter, die sich bei weiterer Eingabe von T9-Codes bilden lassen könnten.

Da bereits eine Datenstruktur verlangt wird, die die Längen auf die T9-Codes abbildet, kann diese Struktur in dieser Methode wiederverwendet werden um die möglichen Kombinationen von Wörter zu ermitteln.
Es soll wie folgt vorgegangen werden:

1. Iteriere über den Längen Index und suche alle T9 Codes, die länger sind als der gegebene
    a. NEIN: Fahre mit den Iterieren fort
    b. JA: Iteriere über diese T9 Codes und ermittle die Codes die den gegebenen Code als Prefix haben
        i. NEIN: Fahre mit den Iterieren fort
        ii. JA: Speichere diese in einer Collection

Es sind zwar mehrere Iterationen notwendig um zu den Wörtern zu kommen, jedoch muss nicht über die Wörter selbst iteriert werden. Da es sich um einen Index handelt ist auch hier eine performante Laufzeit zu erwarten.

1.1.6 numberPrefix2sortedWords

In dieser Methode sollen die Wörter für einen T9-Code ermittelt werden, die sich in Wörterbuch und einen Beispieltext befinden. Diese sollen nach der Häufigkeit im Beispieltext sortiert ausgegeben werden.
Es soll ein Index aufgebaut werden, der die Häufigkeiten im Beispieltext auf die Wörter des Wörterbuches abbildet.

```
unordered_map<string, int>>
```

Ebenso soll die vorherige Implementierung wieder verwendet werden, da das Resultat dasselbe sein muss. Als Resultat wird ein Vector geliefert, da dieser die Sortierung der eingefügten Elemente beibehält und zwar nach der Reihenfolge in der sie dem Container hinzufügt wurde.

Die Sortierung soll wie folgt erfolgen:

1. Initialisiere eine Map mit folgender Signatur
   `unordered_map<int, unordered_set<string>>`
2. Iteriere über die resultierenden Wörter und ermittle deren Häufigkeit und speichere sie in der Map
    a. Häufigkeit vorhanden: Füge das Wort dem Set hinzu
    b. Nicht vorhanden: Erstelle einen neuen Set, füge das Wort hinzu und Speichere das Set in der Map
3. Fahre fort solange noch Wörter vorhanden sind

1.2 Implementierung

Folgend ist der C Source des Programms T9Converter angeführt.

1.2.1 T9Converter.h

Folgend ist die Spezifikation des T9 Konverters angeführt.

```
/*
 * T9.h
 * This header file specifies the T9 converter and its needed resources
 *
 *  Created on: Mar 6, 2015
 *      Author: Thomas Herzog
 */

#ifndef T9_H_
#define T9_H_

#include <map>
#include <tr1/unordered map>
#include <tr1/unordered_set>
#include <set>
#include <string>
#include <sstream>
#include <exception>
#include <fstream>
#include <utility>
#include <vector>

/**
 * The sorter class which gets used for sorting strings by their occurrence.
 * This sorter makes only sense when used on a std::map<int, std::unordered set<std::string>>
where the key
 * represents the occurrences and the set the string which have the same occurrence
 */
class StringWordFrequencySorter {
    public:
        inline StringWordFrequencySorter() {
        }

        inline bool operator()(const int& left, const int& right) {
            return left > right;
        }
};

/**
 * This class handles the conversion from and to the T9 specification.
 */
class T9Converter {
    private:
        /* The references to the path of the dictonary */
        std::string& dictonaryPath;
        /* The path to the sample file which provides the occurrence count for the dictonary
entries */
        std::string& samplePath;
        /* Map which contains the character mappings to their corresponding T9 value */
        std::tr1::unordered_map<int, std::string> mapping;
        /* Map which contains the character mappings their special characters */
        std::tr1::unordered_map<std::string, std::string> specialCharacterMapping;
        /* The map which represents the index of the dictonary entries (T9Value -> string) */
        std::tr1::unordered map<long long, std::tr1::unordered set<std::string>>
indexDictonaryT9;
        /* Map which represents the index of the dictonary entries mapped to their length
(T9Value length -> map<int, string>)*/
        std::tr1::unordered_map<int,
                std::tr1::unordered_map<long long,
                        std::tr1::unordered_set<std::string>>> indexDictonaryT9Length;
        /* The map which represents the index of the dictonary entries mapped to the frequency
in the sample file */
        std::tr1::unordered_map<std::string, int> indexDictonaryFrequency;
        /* Set which holds the invalid digits which are not supported by T9 */
        std::tr1::unordered_set<int> unsupportedDigits;
        /* Flag which indicates if a dictonary has been provided */
        bool dictonaryProvided = false;
```

```
        /**
         * Initializes the backed map with the T9 specified character number mapping.
         */
        void initT9Map();

        /**
         * Inits the set with the invalid digits which are not supported by T9.
         */
        void initInvalidDigitsSet();

        /**
         * Handles the current line of the provided dictonary.
         *
         * @param:
         *      std::string line: the line to be handled and added to the index(s) if valid.
         */
        void handleDictonaryEntry(std::string line);

        /**
         * Converts the given string (single character) to the corresponding digit.
         *
         * @param:
         *      const std::string c: the string which is allowed to contain only one
character.
         * @return:
         *      the T9 representation of the char
         * @throws:
         *      InvalidCharException if the char is not supported by the T9 specification.
         */
        int string2Digit(const std::string c) const;

        /**
         * Converts the given single digit to the corresponding character;
         *
         * @param:
         *      int digit: the digit to be converted to the corresponding character
         * @return
         *      the corresponding character
         * @throws:
         *      InvalidCharException if the given digit is one of the unsupported digits.
         */
        std::string digit2String(const int digit);

        /**
         * Loads the sample text provided by the string argument
         * @param:
         *      const:std:string sampleTextPath: the path to the sample file
         */
        void loadSampleText(const std::string sampleTextPath);

        /**
         * Normalizes the string by removing all unsupported characters.
         * @param:
         *      const std:string& value: the value to be normalized
         */
        void normalizeString(std::string& value) const;

        /**
         * Reloads the dictonary if there is a dictonary path provided.
         * Does nothing if no dictonary path is present
         *
         * @param:
         *      std::String dictonaryPath: the path to the dictonary to be loaded.
         */
        void reloadDictonary(std::string& dictonaryPath);
    public:

        /**
         * Empty constructor which initializes this handler by initializing the backed T9 map.
         */
        T9Converter();

        /**
         * Constructor which gets a dictonary path provided.
         * This dictonary will be used in stead of the permutation algorithm.
```

```
         *
         * @param:
         *      std::string dictonaryPath: the dictonary path which points
         *                  to a line separated list of words which can be mapped to the T9
specification.
         * @param:
         *      std::string samplePath: the path to the sample file
         *
         */
        T9Converter(std::string dictonaryPath, std::string samplePath);

        /**
         * For now nothing to do
         */
        ~T9Converter();

        /**
         * Converts the word to the corresponding T9 representation.
         *
         * @param:
         *      const string word: the word to be converted
         * @return:
         *      the T9 representation of the word, an thrown exception otherwise
         * @throws:
         *      InvalidCharException if the word contains chars which are supported in the T9
specification.
         */
        long long word2Number(std::string word) const;

        /**
         * Converts a T9 number to all of the possible string which are calculated
permutations.
         *
         * @param:
         *      const int number: the T9 number to be converted to the mapped strings
         * @return:
         *      the strings represented by a set which contains all of the mapped strings for
the T9 number.
         */
        std::tr1::unordered_set<std::string> number2Strings(
                const long long value);

        /**
         * Converts a T9 number to all of the possible string which are provided in the backed
directory. (T9 index)
         *
         * @param:
         *      const int number: the T9 number to be converted to the mapped strings
         * @return:
         *      the strings represented by a set which contains all of the mapped strings for
the T9 number.
         */
        std::tr1::unordered_set<std::string> number2Words(
                const long long value);

        /**
         * Converts a T9 number to all of the possible string which are provided in the backed
directory. (Length index)
         *
         * @param:
         *      const int number: the T9 number to be converted to the mapped strings
         * @return:
         *      the strings represented by a set which contains all of the mapped strings for
the T9 number.
         */
        std::tr1::unordered_set<std::string> number2WordsByLength(
                const long long value);

        /**
         * Converts a T9 number to all of the possible string which have length +1 which are
provided in the backed directory. (length index)
         *
         * @param:
         *      const int number: the T9 number to be converted to the mapped strings
         * @return:
```

```
         *      the strings represented by a set which contains all of the mapped strings for
the T9 number.
         */
        std::tr1::unordered_set<std::string> numberPrefix2Words(
                const long long value);

        /**
         * Converts a T9 number to all of the possible string which have length +1 which are
provided in the backed directory. (sample index)
         *
         * @param:
         *      const int number: the T9 number to be converted to the mapped strings
         * @return:
         *      the strings represented by a set which contains all of the mapped strings for
the T9 number.
         */
        std::vector<std::string> numberPrefix2sortedWords(
                const long long value);

        /**
         * @see T9Converter::reloadDictonary(std:string dictonaryPath)
         */
        void reloadDictonary();

};

#endif /* T9_H_ */
```

1.2.2 T9Converter.cpp

Folgend ist die Implementierung der Spezifikation T9Converter.h angeführt.

```
/*
 * T9.cpp
 *
 *  Created on: Mar 6, 2015
 *      Author: cchet
 */
#include "T9Converter.h"
#include <algorithm>
#include <list>
#include <fstream>
#include <iostream>
#include <iterator>

using namespace std;
using namespace std::tr1;

// ############################################################
// Constructor - Destructor section
// ############################################################
T9Converter::T9Converter() :
        T9Converter("", "") {
} /* T9Converter::T9Converter */

T9Converter::T9Converter(string dictonaryPath, string samplePath) :
        dictonaryPath(dictonaryPath), samplePath(samplePath) {
    initT9Map();
    initInvalidDigitsSet();
    reloadDictonary(dictonaryPath);
    loadSampleText(samplePath);
} /* T9Converter::T9Converter */

T9Converter::~T9Converter() {
} /* T9Converter::~T9Converter */

// ############################################################
// PRIVATE section
// ############################################################
void T9Converter::initT9Map() {
    mapping[0] = "'/"; /* Ignored charactes */
    mapping[2] = "abc";
    mapping[3] = "def";
    mapping[4] = "ghi";
```

```
    mapping[5] = "jkl";
    mapping[6] = "mno";
    mapping[7] = "pqrs";
    mapping[8] = "tuv";
    mapping[9] = "wxyz";

    specialCharacterMapping.insert(pair<string, string>("ÀÄäàáâå", "a"));
    specialCharacterMapping.insert(pair<string, string>("ç", "c"));
    specialCharacterMapping.insert(pair<string, string>("Ëëéêèœ", "e"));
    specialCharacterMapping.insert(pair<string, string>("íî", "i"));
    specialCharacterMapping.insert(pair<string, string>("ñ", "n"));
    specialCharacterMapping.insert(pair<string, string>("öôóø", "o"));
    specialCharacterMapping.insert(pair<string, string>("Üüûú", "u"));
    specialCharacterMapping.insert(pair<string, string>("ß", "s"));
}

void T9Converter::initInvalidDigitsSet() {
    unsupportedDigits.insert(0);
    unsupportedDigits.insert(1);
} /* T9Converter::initInvalidDigitsSet */

int T9Converter::string2Digit(const string c) const {
    int t9Value(-1);
    string tmp = c;
    bool replaced = false;

    for_each(specialCharacterMapping.begin(), specialCharacterMapping.end(),
            ([&tmp, &replaced](pair<string, string> p) {
                if((!replaced) && (p.first.find(tmp) == 0)) {
                    tmp = p.second;
                    replaced = true;
                } /* if */
            }));
    for_each(mapping.begin(), mapping.end(),
            ([&c, &t9Value](pair<int, string> mapPair) {
                if(t9Value == -1) {
                    int idx = mapPair.second.find(c);
                    t9Value = (idx >= 0) ? mapPair.first : -1;
                }/* if */
            }));
    return t9Value;
} /* T9Converter::string2Digit */

string T9Converter::digit2String(const int digit) {
    stringstream ss;
    ss << digit;
    // Validate if given parameter is single digit
    if ((digit < 0) || (digit > 9)) {
        cout << "Parameter must represent a single digit" << endl << flush;
    }/* if */
    // Validate if parameter is supported digit
    auto it = find(unsupportedDigits.begin(), unsupportedDigits.end(), digit);
    if (it != unsupportedDigits.end()) {
        cout << "Invalid digit detected" << endl << flush;
    }/* if */

    return mapping[digit];
} /* T9Converter::digit2String */

void T9Converter::reloadDictonary() {
    if (dictonaryProvided) {
        reloadDictonary(dictonaryPath);
    }/* if */
} /* T9Converter::reloadDictonary */

void T9Converter::reloadDictonary(string& dictonaryPath) {
    if (!dictonaryPath.empty()) {
        ifstream fis;
        fis.open(dictonaryPath.c_str());
        if (fis.fail()) {
            this->dictonaryPath = "";
            dictonaryProvided = false;
            cout << "Failed to open the dictonary file" << endl << flush;
        }/* if */

        // Handle line
```

```
        string line;
        while ((fis.good()) && (!fis.eof())) {
            getline(fis, line);
            normalizeString(line);
            handleDictonaryEntry(line);
        }/* while */

        fis.close();
        this->dictonaryProvided = true;
        this->dictonaryPath = string(dictonaryPath);
    }/* if */
} /* T9Converter::reloadDictonary */

void T9Converter::handleDictonaryEntry(string line) {
    stringstream ss;
    long long idx = word2Number(line);
    unordered_map<long long, unordered_set<string>>::iterator itIndex;
    unordered_map<int, unordered_map<long long, unordered_set<string>>> ::iterator
itLengthIndex;
    unordered_set<string> values;

    /* build index T9Value to unordered set of strings */
    if ((itIndex = indexDictonaryT9.find(idx)) != indexDictonaryT9.end()) {
        (*itIndex).second.insert(line);
    } else {
        values.insert(line);
        indexDictonaryT9.insert(
                pair<long long, unordered_set<string>>(idx, values));
    }/* if */

    /* build index length of t9Value to collection of corresponding t9Values to the
unordered set of strings */
    ss << idx;
    int length = ss.str().length();
    itLengthIndex = indexDictonaryT9Length.find(length);
    if ((itLengthIndex != indexDictonaryT9Length.end())
            && (((*itLengthIndex).second.find(idx))
                    != ((*itLengthIndex).second.end()))) {
        (*itLengthIndex).second[idx].insert(line);
    } else {
        unordered_map<long long, unordered_set<string>> indexMap;
        unordered_set<string> lengthMappedString;
        lengthMappedString.insert(line);
        pair<long long, unordered_set<string>> indexPair(idx,
                lengthMappedString);

        // When length entry present but no t9Value mapped to any string
        if (itLengthIndex == indexDictonaryT9Length.end()) {
            indexMap.insert(indexPair);
            indexDictonaryT9Length.insert(
                    pair<int, unordered_map<long long, unordered_set<string>>>(length,
indexMap));
        } else {
            (*itLengthIndex).second.insert(indexPair);
        }/* if */
    }/* if */

    /* Prepare word frequency unordered_map of the dictonary */
    unordered_map<string, int>::iterator itWordFrequency =
            indexDictonaryFrequency.find(line);
    if (itWordFrequency == indexDictonaryFrequency.end()) {
        indexDictonaryFrequency.insert(pair<string, int>(line, 0));
    }/* if */
} /* T9Converter::handleDictonaryEntry */

void T9Converter::loadSampleText(const string sampleTextPath) {
    if (!sampleTextPath.empty()) {
        vector<string> words;
        ifstream fis;
        fis.open(sampleTextPath.c_str());
        if (fis.fail()) {
            cout << "Failed to open the sampel file" << endl << flush;
        }/* if */

        // Handle line
        for_each(istream_iterator<string>(fis), istream_iterator<string>(),
```

```
                ([this](string word) {
                    normalizeString(word);
                    unordered_map<string, int>::iterator it =
indexDictonaryFrequency.find(word);
                    if(it != indexDictonaryFrequency.end()) {
                        (*it).second++;
                    }/* if */
                }));

        fis.close();
    }
} /* T9Converter::loadSampleText */

void T9Converter::normalizeString(std::string& word) const {
    string::iterator newEnd;
    newEnd = remove_if(word.begin(), word.end(), ([](const char ch) {
        return static_cast<bool>(ispunct(ch));
    }));
    newEnd = remove_if(word.begin(), newEnd, ([](const char ch) {
        return static_cast<bool>(isdigit(ch));
    }));
    word.erase(newEnd, word.end());
    transform(word.begin(), word.end(), word.begin(), ([](const char ch) {
        return static_cast<char>(tolower(ch));
    }));
} /* T9Converter::normalizeString */

// ############################################################
// PUBLIC section
// ############################################################
long long T9Converter::word2Number(string word) const {
    stringstream ss("");
    normalizeString(word);
    for_each(word.begin(), word.end(), [&ss, this](char c) {
        stringstream tmp;
        tmp << c;
        long long t9Value = this->string2Digit(tmp.str());
        if((t9Value >= 2) && (t9Value <= 9)) {
            ss << t9Value;
        }/* if */
    });
    string res = ss.str();
    if (res.empty()) {
        return -1;
    } else {
        return atol(ss.str().c_str());
    }/* if */
} /* T9Converter::word2Number */

unordered_set<string> T9Converter::number2Strings(const long long t9Value) {

    unordered_set<string> result;
    unordered_set<string> resultTmp;
    list<string> usedCharacterList;
    stringstream ss;
    ss << t9Value;
    string t9ValueString = ss.str();

// Get character sets for given number
    for_each(t9ValueString.begin(), t9ValueString.end(),
            ([&usedCharacterList, this](char c) {
                if(c != '\0') {
                    usedCharacterList.push_back(digit2String(((int)c) - 48));
                }}));
// Iterate over the resulting character sets
    for_each(usedCharacterList.begin(), usedCharacterList.end(),
            ([&result, &resultTmp, &t9ValueString](const string characters) {
                unordered_set<string> tmp;
                // Iterate over each character of the character unordered_set
                for_each(characters.begin(), characters.end(), ([&resultTmp, &tmp](const char
c) {
                                    stringstream ss;
                                    // First time add characters directly to result
unordered set
                                    if(resultTmp.empty()) {
                                        ss << c;
```

```
                                        tmp.insert(ss.str());
                                    }
                                    // Each other iteration append each character to each
string contained in the result unordered set
                                    else {
                                        for_each(resultTmp.begin(), resultTmp.end(), ([&tmp,
&c](const string exisitng) {
                                                                stringstream ss;
                                                                ss << exisitng << c;
                                                                tmp.insert(ss.str());
                                                            }));
                                    }/* if */
                                }));
                // Append newly created strings to result unordered_set
                for_each(tmp.begin(), tmp.end(), ([&tmp, &result, &resultTmp,
&t9ValueString](const string s) {
                                    if(s.length() == t9ValueString.length()) {
                                        result.insert(s);
                                    }
                                    resultTmp.insert(s);
                                }));
            }));

    return result;
} /* T9Converter::number2Strings */

unordered_set<string> T9Converter::number2Words(const long long t9Value) {
    unordered_map<long long, unordered_set<string>>::iterator it =
            indexDictonaryT9.find(t9Value);
    if (it != indexDictonaryT9.end()) {
        return (*it).second;
    } else {
        return unordered_set<string>();
    }/* if */
} /* T9Converter::number2Words */

unordered_set<string> T9Converter::number2WordsByLength(
        const long long t9Value) {
    stringstream ss;
    ss << t9Value;
    int length = ss.str().length();
    unordered_map<int, unordered_map<long long, unordered_set<string>>> ::iterator
lengthIndexIterator = indexDictonaryT9Length.find(length);
    if (lengthIndexIterator != indexDictonaryT9Length.end()) {
        unordered_map<long long, unordered_set<string>>::iterator t9ValueIdnexIterator =
                (*lengthIndexIterator).second.find(t9Value);
        if (t9ValueIdnexIterator != indexDictonaryT9Length[length].end()) {
            return (*t9ValueIdnexIterator).second;
        }/* if */
    }/* if */

    return unordered_set<string>();
} /* T9Converter::number2WordsByLength */

unordered_set<string> T9Converter::numberPrefix2Words(const long long value) {
    stringstream ss;
    ss << value;
    unordered_set<string> values;

    string valueString = ss.str();
    int length = valueString.length();

    for_each(indexDictonaryT9Length.begin(), indexDictonaryT9Length.end(),
            ([&values, &valueString, &length](const pair<int, unordered_map<long long,
unordered_set<string>>> indexEntry) {
                if(indexEntry.first >= length) {
                    for_each(indexEntry.second.begin(), indexEntry.second.end(), ([&length,
&values, &valueString](const pair<long long, unordered_set<string>> entry) {
                                        stringstream ss;
                                        ss << entry.first;
                                        int result = ss.str().find(valueString);
                                        if(result == 0) {
                                            values.insert(entry.second.begin(),
entry.second.end());
                                        }/* if */
                                    }));
```

```
                }
            }));

    return values;
} /* T9Converter::numberPrefix2Words */

vector<string> T9Converter::numberPrefix2sortedWords(const long long value) {
    stringstream ss;
    ss << value;
    string valueString = ss.str();
    vector<string> result;

    /* get dictonary result */
    unordered_set<string> values = numberPrefix2Words(value);
    map<int, vector<string>, StringWordFrequencySorter> tmpMap;

    for_each(values.begin(), values.end(),
            ([this, &tmpMap](const string stringValue) {
                unordered_map<string, int>::iterator it =
indexDictonaryFrequency.find(stringValue);
                if(it != indexDictonaryFrequency.end()) {
                    map<int, vector<string>, StringWordFrequencySorter>::iterator itTmpMap =
tmpMap.find((*it).second);
                    if(itTmpMap != tmpMap.end()) {
                        (*itTmpMap).second.push_back(stringValue);
                    } else {
                        vector<string> tmpSet;
                        tmpSet.push_back(stringValue);
                        tmpMap.insert(pair<int, vector<string>>((*it).second, tmpSet));
                    }/* if */
                }/* if */
            }));

    for_each(tmpMap.begin(), tmpMap.end(),
            ([&result](pair<int, vector<string>> mapPair) {
                for_each(mapPair.second.begin(), mapPair.second.end(), ([&result](const string
val) {
                                    result.push_back(val);
                                }));
            }));

    return result;

} /* T9Converter::numberPrefix2sortedWords */
```

1.2.3 T9Tests.h

Folgend ist die Spezifikation der Tests für den T9 Konverter angeführt.

```
/*
 * T9Tests.h
 * This header file specifies the tests for the T9Converter
 *
 *  Created on: Mar 15, 2015
 *      Author: Thomas Herzog
 */

#ifndef T9TESTS H
#define T9TESTS H

#include "cute.h"
#include "../T9Converter.h"

/**
 * prepares the suite by collection the test functions
 */
cute::suite buildSuite();

/**
 * Inits the tests by initializing the T9converter with the
 * dictonary, which takes some time.
 */
void initTests();

/**
 * Cleans the tests up
 */
void cleanupTests();

// ##############################################################
// word2Number tests
// ##############################################################
void test_w2n_invalid_char();

void test_w2n();

// ##############################################################
// number2strings tests
// ##############################################################
void test_n2s_invalid_digit();

void test_n2s();

// ##############################################################
// number2words tests
// ##############################################################
void test_n2w_invalid_digit();

void test_n2w();

// ##############################################################
// number2wordsByLength tests
// ##############################################################
void test_n2wl_invalid_digit();

void test_n2wl();

// ##############################################################
// numberPrefix2Words tests
// ##############################################################
void test_n2pw_invalid_digit();

void test_n2pw();

// ##############################################################
// numberPrefix2sortedWords tests
// ##############################################################
void test_n2psw_invalid_digit();

void test_n2psw();
```

```
#endif /* T9TESTS_H_ */
```

1.2.4 T9Tests.ccpp

Folgend ist die Implementierung der Spezifikation T9Tests.h angeführt.

```
/*
 * T9Tests.cpp
 * This source file contains the tests for the T9Converter.
 *
 *  Created on: Mar 15, 2015
 *      Author: Thomas Herzog
 */
#include <iostream>
#include <stdlib.h>
#include <cute.h>
#include "T9Tests.h"

using namespace std;
using namespace cute;
using namespace std::tr1;

/**
 * Global variable because no state is hold by the instance
 * and the dictonary load takes some time
 */
static T9Converter* t9;
static auto printStrings = ([](const string& s) {
    cout << ">"<< s << endl << flush;
});

suite buildSuite() {
    suite testSuite;
    testSuite.push_back(CUTE(initTests));

    testSuite.push_back(CUTE(test_w2n_invalid_char));
    testSuite.push_back(CUTE(test_w2n));

    testSuite.push_back(CUTE(test_n2s_invalid_digit));
    testSuite.push_back(CUTE(test_n2s));

    testSuite.push_back(CUTE(test_n2w_invalid_digit));
    testSuite.push_back(CUTE(test_n2w));

    testSuite.push_back(CUTE(test_n2wl_invalid_digit));
    testSuite.push_back(CUTE(test_n2wl));

    testSuite.push back(CUTE(test n2pw invalid digit));
    testSuite.push_back(CUTE(test_n2pw));

    testSuite.push_back(CUTE(test_n2psw_invalid_digit));
    testSuite.push_back(CUTE(test_n2psw));

    testSuite.push_back(CUTE(cleanupTests));

    return testSuite;
}

void initTests() {
    cout << "Initializing T9Converter instance..." << endl << flush;
    t9 = new T9Converter("de_neu.dic", "wiki.txt");
    cout << "finished" << endl << flush;
}

void cleanupTests() {
    delete t9;
}

// ############################################################
// word2Number tests
// ############################################################
void test_w2n_invalid_char() {
    cout << endl << "word2Number(\"?\")" << endl << flush;
    ASSERT_EQUAL(-1, t9->word2Number("?"));
```

```
    cout << "word2Number(\"#\")" << endl << flush;
    ASSERT_EQUAL(-1, (long )t9->word2Number("#"));

    cout << "word2Number(\".\")" << endl << flush;
    ASSERT_EQUAL(-1, (long )t9->word2Number("."));
}

void test_w2n() {
    long long result;
    cout << endl << "word2Number(\"kiss\")" << endl << flush;
    result = t9->word2Number("kiss");
    cout << "result: " << result << endl << endl << flush;
    ASSERT_EQUAL(5477, result);
}

// ############################################################
// number2strings tests
// ############################################################
void test_n2s_invalid_digit() {
    cout << endl << "number2Strings(0)" << endl << flush;
    t9->number2Strings(0);
    cout << endl << "number2Strings(1)" << endl << flush;
    t9->number2Strings(1);
}

void test_n2s() {
    unordered_set<string> result;
    cout << endl << "number2Strings(\"5477\")" << endl << flush;
    result = t9->number2Strings(5477);
    cout << "Resulting words: " << result.size() << endl << flush;
    for_each(result.begin(), result.end(), printStrings);
}
// ############################################################
// number2words tests
// ############################################################
void test_n2w_invalid_digit() {
    cout << endl << "number2Words(54771)" << endl << flush;
    t9->number2Words(54770);
    cout << endl << "number2Words(54770)" << endl << flush;
    t9->number2Words(54771);
}

void test_n2w() {
    unordered_set<string> result;
    cout << endl << "number2Words(\"42556\")" << endl << flush;
    result = t9->number2Words(42556);
    cout << "Resulting words: " << result.size() << endl << flush;
    for_each(result.begin(), result.end(), printStrings);
}

// ############################################################
// number2wordsByLength tests
// ############################################################
void test_n2wl_invalid_digit() {
    cout << endl << "number2WordsByLength(54771)" << endl << flush;
    t9->number2WordsByLength(54770);
    cout << endl << "number2WordsByLength(54770)" << endl << flush;
    t9->number2WordsByLength(54771);
}

void test_n2wl() {
    unordered_set<string> result;
    cout << endl << "number2WordsByLength(\"42556\")" << endl << flush;
    result = t9->number2WordsByLength(42556);
    cout << "Resulting words: " << result.size() << endl << flush;
    for_each(result.begin(), result.end(), printStrings);
}

// ############################################################
// numberPrefix2Words tests
// ############################################################
void test_n2pw_invalid_digit() {
    cout << endl << "numberPrefix2Words(54771)" << endl << flush;
    t9->numberPrefix2Words(54770);
    cout << endl << "numberPrefix2Words(54770)" << endl << flush;
```

```
    t9->numberPrefix2Words(54771);
}

void test_n2pw() {
    unordered_set<string> result;
    cout << endl << "numberPrefix2Words(\"42556\")" << endl << flush;
    result = t9->numberPrefix2Words(42556);
    cout << "Resulting words: " << result.size() << endl << flush;
    for_each(result.begin(), result.end(), printStrings);
}

// ############################################################
// numberPrefix2sortedWords tests
// ############################################################
void test_n2psw_invalid_digit() {
    cout << endl << "numberPrefix2sortedWords(54771)" << endl << flush;
    t9->numberPrefix2sortedWords(54770);
    cout << endl << "numberPrefix2sortedWords(54770)" << endl << flush;
    t9->numberPrefix2sortedWords(54771);
}

void test_n2psw() {
    vector<string> result;
    cout << endl << "numberPrefix2sortedWords(\"42556\")" << endl << flush;
    result = t9->numberPrefix2sortedWords(42556);
    cout << "Resulting words: " << result.size() << endl << flush;
    for_each(result.begin(), result.end(), printStrings);
}
```

1.2.5 testSuiteRunner.cpp

Folgend ist die Implementierung des Test Suite Runners angeführt, welcher das Hauptprogramm darstellt.

```
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"
#include "tests/T9Tests.h"

using namespace std;
using namespace cute;

static void runSuite(int argc, char const *argv[]) {
    suite setTestSuite, bagTestSuite;
    xml_file_opener xmlfile(argc, argv);
    xml_listener<ide_listener<> > lis(xmlfile.out);

    setTestSuite = buildSuite();
    makeRunner(lis, argc, argv)(setTestSuite, "AllTests");
} /* runSuite */

int main(int argc, char const *argv[]) {
    runSuite(argc, argv);
} /* main */
```

1.3 Tests

Folgend sind die Tests für den T9Converter angeführt. An dabei zu dieser Dokumentation finden Sie eine Textdatei mit dem vollem Testoutput. Es wird hier lediglich ein Auszug aus dem Konsolenoutput angezeigt, da es den Umfang dieses Dokuments unnötig aufblasen würde.

1.3.1 word2number

Folgend sind die Tests für die Methode word2number angeführt.

```
#beginning AllTests 14

#starting initTests
Initializing T9Converter instance...
finished

#success initTests OK

#starting test_w2n_invalid_char

word2Number("?")
word2Number("#")
word2Number(".")

#success test_w2n_invalid_char OK

#starting test_w2n

word2Number("kiss")
result: 5477


#success test_w2n OK
```

1.3.2 number2strings

Folgend sind die Tests der Methode number2strings angeführt, welche die möglichen Permutationen der Wörter zugehörig zu einem T9 Code ermittelt. Aufgrund des langen Outputs nur ein Ausschnitt angezeigt.

```
#starting test_n2s_invalid_digit

number2Strings(0)
Invalid digit detected

number2Strings(1)
Invalid digit detected

#success test_n2s_invalid_digit OK

#starting test_n2s

number2Strings("5477")
Resulting words: 144
>khrq
>jips
>khpp
>kipp
```

```
#beginning AllTests 4

#starting initTests
Initializing T9Converter instance...
finished

#success initTests OK

#starting test_n2s_invalid_digit

number2Strings(0)
Invalid digit detected

number2Strings(1)
Invalid digit detected

#success test_n2s_invalid_digit OK

#starting test_n2s

number2Strings(786583)
Resulting words: 972
>svnjtf
>suokue
>ptmltf
>pvojve
>pvmktd
>qvnjvd
>stmjte
>pvojud
>rtnktd
>qvojtd
>rtojvd
>stnltf
>svnktf
>stmjvd
>pvmkud
>sumjuf
>qvmjve
>qtmkte
>qumkud
>rtnjtd
>sunltd
>pumjud
>rvmkue

number2Strings(428787266537)
Resulting words: 0

#success test_n2s OK

#starting cleanupTests

#success cleanupTests OK

#ending AllTests
```

1.3.3 number2words

Folgend sind die Tests der Methode number2words angeführt, die Wörter aus einem Wörterbuch für einen T9-Code liefert. Aufgrund des langen Outputs wird hier nur ein Ausschnitt des Outputs gezeigt.

```
#starting test_n2w_invalid_digit

number2Words(54771)

number2Words(54770)

#success test_n2w_invalid_digit OK

#starting test_n2w

number2Words("42556")
Resulting words: 2
>hallo
>gallo

#success test_n2w OK
```

1.3.4 number2wordsByLength

Folgend sind die Tests der Methode number2wordsByLength angeführt, die die möglichen Wörter eines T9-Codes ermittelt und diese in einem Index ermittelt.

```
#starting test_n2wl_invalid_digit

number2WordsByLength(54771)

number2WordsByLength(54770)

#success test_n2wl_invalid_digit OK

#starting test_n2wl

number2WordsByLength("42556")
Resulting words: 2
>hallo
>gallo

#success test_n2wl OK
```

1.3.5 numberPrefix2Words

Folgend sind die Tests der Methode number2prefixWords angeführt, welche alle möglichen folgenden Wörter ermittelt.

```
#starting test_n2pw_invalid_digit

numberPrefix2Words(54771)

numberPrefix2Words(54770)

#success test_n2pw_invalid_digit OK

#starting test_n2pw

numberPrefix2Words("42556")
Resulting words: 25
>gallmannsweil
>gallmücke
>gallomanie
>gallmersgarten
>gallonenweise
>galloway
>gallophob
>gallophil
>halloween
>gallone
>hallore
>gallowitz
>gallophobie
>gallneukirchen
>gallopamil
>galloromanisch
>gallmannsegg
>gallophilie
>gallomane
>gallon
>gallmilbe
>hallo
>hallodri
>gallonen
>gallo

#success test_n2pw OK
```

1.3.6 numberPrefix2sortedWords

Folgend sind die Tests der Methode numberPrefix2sortedWords angeführt, welche die Wörter die sich folgend bilden lassen würden ermittelt und sie nach Worthäufigkeit in einem Sampletext sortiert.

```
#starting test_n2psw_invalid_digit

numberPrefix2sortedWords(54771)

numberPrefix2sortedWords(54770)

#success test_n2psw_invalid_digit OK

#starting test_n2psw

numberPrefix2sortedWords("42556")
Resulting words: 25
>gallmannsweil
>gallmücke
>gallomanie
>gallmersgarten
>gallonenweise
>galloway
>gallophob
>gallophil
>halloween
>gallone
>hallore
>gallowitz
>gallophobie
>gallneukirchen
>gallopamil
>galloromanisch
>gallmannsegg
>gallophilie
>gallomane
>gallon
>gallmilbe
>hallo
>hallodri
>gallonen
>gallo

#success test_n2psw OK

#starting cleanupTests
```