ASSIGNMENT 6
Machine Translation

COMP-202, Summer 2010

Due: Wednesday, June 23rd, 2010

1 Introduction

In this assignment we will write a tool that can translate texts from French to English. We will do it in a fairly simple way: we merely have a table of French words with the corresponding translation in English. When translating, we will read through texts, and attempt to find, for every word encountered, the closest word in our dictionary, and output its English translation. We can use our edit distance implementation to find close matches, but there are also some things to be careful about: When translating, we want to keep punctuation and white space. We also want to keep capitalization, approximately.

This project will be a deeper venture into object oriented programming, and we will write classes for different kinds of objects. We will also get used to `Java HashMap`s, and `Java` collections in general. We will also use some inheritance.

2 Specification

We have done the work for you to split up the problem into a couple of classes, to simplify the problem. You should follow the specification closely, as if the different parts were programmed by different people.

2.1 Word

We will start with a simple class that can represent words. They basically store a word internally as a String. They are immutable. Words help us to manage the notion of capitalization. We identify three different types of capitalization. A word is `ALL_CAPS` (= 0) if it is all capitalized (i.e. the String method `toUpperCase` does not yield a different String). It is `FIRST_CAPS` (= 1) if the first letter is capitalized, and it is `FIRST_LOWER` (= 2) in any other case. You should put these as `public static` constants into the class. Note that in case 1 or 2, it doesn't matter what the remaining characters are like. We also want the following methods:

method name	**parameters**	**function and return**
<constructor>	A `string` denoting the word	
`toString`	-	returns the word as a `String`
`getCapitalization`	-	returns the capitalization of the word, as defined by the above constants
`applyCapitalization`	another `Word`	Returns a new `Word` with the same characters as `this`, but with the same capitalization as the given argument. If the given word is not all caps, all characters after the first should be lower case.
`getDistance`	another `Word`	Returns the edit distance to the given argument. The comparison should be case-insensitive.
`equals`	an `Object`	returns true if the given `Object` is of type `Word` and stores the same text
`hashCode`	-	returns the hash code of the String as an `int`
`length`	-	returns the length of the String as an `int`

You should also include a static method `min`, taking two `Word`s, and returning the one that comes lexicographically before the other. You may assume that a word has always at least one character in it. Hint: You should use the methods that the `String` and `Character` classes provide for you.

Note: you should include the `StringUtils` class from the previous assignment - we have found another use for it here, because our edit distance implementation is sufficiently general.

2.2 Dictionary

A `Dictionary` is just a map from a `Word` to another `Word`. Thus Dictionary should extend `HashMap<Word,Word>`. This allows us to reuse all its methods, and makes them public as well. The dictionary provides a method that takes in a word, and returns its translation. A Dictionary should have the following, additional methods:

method name	**parameters**	**function and return**
<constructor>	-	creates an empty dictionary
<constructor>	A `String` denoting a file name	reads the dictionary from the filename
`getClosestKey`	a Word	Returns the key (`Word` that most closely matches the given `Word` (computed via the edit distance). If multiple keys have the same edit distance to the argument, the 'minimal' key should get returned.
`translate`	a Word	Returns the translation of the key (`Word`) that most closely matches the given word, computed via `getClosestKey`. It should return the original `Word` if the edit distance divided by the argument's `Word` length is more than `TRANSLATE_RATIO_THRESHOLD`. This should ensure that numbers or names do not get translated into random words. The capitalization of the translation should be the same as on the argument.

The constant `TRANSLATE_RATIO_THRESHOLD` should be set to 0.25. Hint: To get the closest key, you have to compare the given `Word` with all keys. You can get all keys via the method `keySet`, defined for `HashMap`s. This will give you a `Set`, which is another `Java` collection. You can access all its elements either by using its iterator, or by crating an `ArrayList` from it (`ArrayList`s can take a `Set` of elements in its constructor).

2.3 Translator

A translator is an object, which stores a dictionary internally. It uses it to perform translations on arbitrary text, which is given as a String. Implement the following interface:

method name	**parameters**	**function and return**
<constructor>	a `Dictionary`	
`translate`	a `String`	returns a translation of the given `String` as a new `String`. All white space and punctuation should be the same as in the input.

Because we want to keep the white space and punctuation, we cannot use a `Scanner`, because it will discard all delimiters. To make your job easier, we are providing you with a class called `TranslationScanner`, which works similar to a Scanner, except that it uses everything that is not a letter or number (or an apostrophe between those) as a delimiter. It will also allow you to find what the exact delimiter are, so that you can build your translation using the original delimiters. It's up to you to figure out how this class works, it is well documented.

2.4 TranslatorTool

This class merely provides a main method. As a command line argument it should take the location of a dictionary text file. It should read whatever comes in at the terminal (without prompt), and output its translation. If no command line argument is given, the program should print out a description of how it works.