Speed-Typer

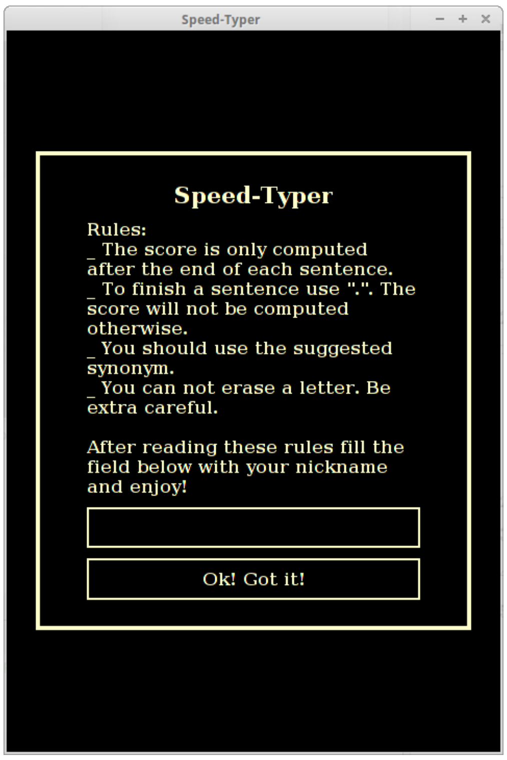
Alexandre Ribeiro João Macedo

1 Introduction

The main goal of this project was to create a simple speed-typer game in Java that could update the graphical user interface (GUI) in real time by using multi-thread programming. Two threads were used. The first to update the GUI and get the user input and the second to check if the words were correctly spelled and look for synonyms. For the last part, we used api's available for free online.

2 Rules

The game have four rules that are displayed in the first screen where the player is asked to enter his nickname as show in figure 1.



Speed-Typer

Rules:
_ The score is only computed after the end of each sentence.
_ To finish a sentence use ".". The score will not be computed otherwise.
_ You should use the suggested synonym.
_ You can not erase a letter. Be extra careful.

After reading these rules fill the field below with your nickname and enjoy!

Figure 1: Welcome Screen

- The score is only computed after the end of each sentence.
- To finish a sentence press ".". The score will not be computed otherwise.
- You should use the suggested synonym. Otherwise it will have a huge negative score. If there is no synonym, you can use any word that you want.
- You can not erase a letter.

The player can type any sentence consisting of any word he wants as long as the rules are followed. He will be awarded by the corrected typed words and punished by the misspelled words or if he did not used the suggested synonym.

2.1 Score

The score is calculated for each sentence. It is the sum of each correctly spelled word length to the power of two minus the sum of each misspelled word length to the power of three with a extra negative bonus of the length of the suggested synonym to the power of four if it was not used. In other terms: let w_i and e_j be the group of correctly spelled and misspelled words, respectively, then

$$\text{Score} = \sum_i (w_i.\text{length})^2 - \sum_j (e_j.\text{length})^3 - (\text{synonym} \notin w_i\ ?) * (\text{synonym.length})^4$$

3 GUI

In order to create the GUI, we used the Swing toolkit for Java. It is important to say that Swing is not thread-safe and only the Event Dispatch Thread (EDT) can make modifications in the GUI when the program is running. To solve this problem, we used the class SwingWorker that allows to launch a thread to do computations in the background and that can send data that will be executed in the EDT. This allow us to a big load of work in a thread without freezing the GUI and when we have the results, we can update the GUI by sending some of the results to be executed at the EDT.

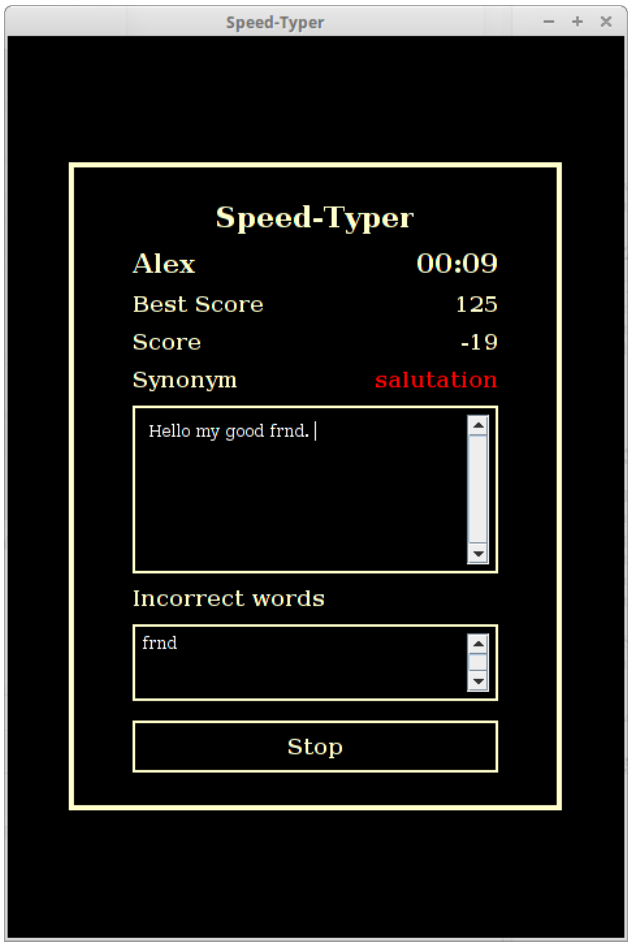


Figure 2: Game Screen

By implementation choice, the SwingWorker was used to read from three different blocking queues of Strings that were filled by the other threads. And when this information was available it would, in the EDT, update the fields of Score, Synonym, and WrongWords. The figure 2 shows an example of the game being played.

4 Implementation

In this section, the focus will be implementation of the WordHandler, that is the part of the code that get the typed words, check if they are correct, calculate the score, get the synonyms and send these informations to the queues mentioned above to update the GUI. The WordHandler takes a BlockingQueue of LinkedList<String>. The LinkedList<String> is filled by the GUI thread that handles the typed keys. Once a word is formed, it is sent to this list. If the character "." is pressed and there are available words. These words, in the form of the after-mentioned list is sent to the BlockingQueue.

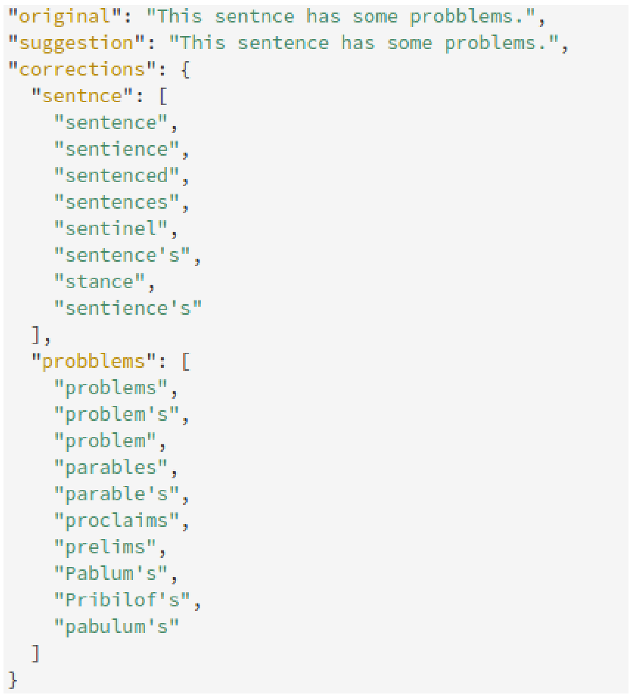
In the WordHandler thread, once a list is received it take all words and check if they are correctly spelled by using an API that will be explained in the next section. Using this result, we calculate the score as explained above and we get the first correct word to create a synonym using another API that will also be discussed in the next session. The relevant informations is sent to the propers queues, where the SwingWorkers will make the GUI update.

5 API's

All the API's calls were made using the Unirest library.

5.1 Spell check API

For the spell check, we used the Montana Flynn Spell Checker, that is available for free. It works in a way that, given an sentence, it returns the misspelled words with suggestions for its correction. The response is given by a JSON file. All we had to do was extract the misspelled words from this file.

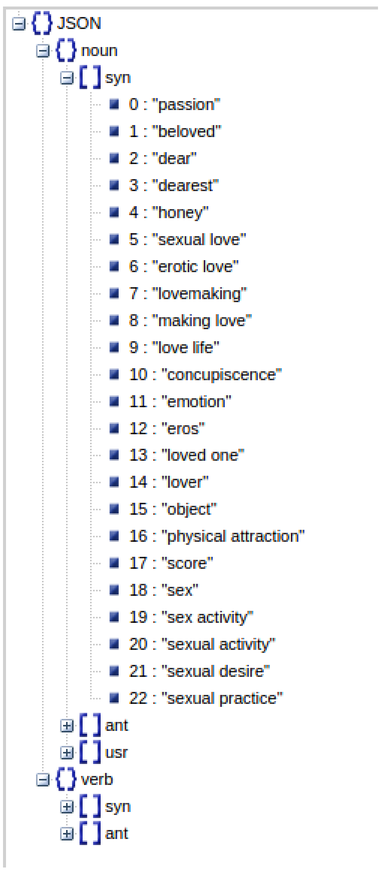


```
"original": "This sentnce has some probblems.",
"suggestion": "This sentence has some problems.",
"corrections": {
  "sentnce": [
    "sentence",
    "sentience",
    "sentenced",
    "sentences",
    "sentinel",
    "sentence's",
    "stance",
    "sentience's"
  ],
  "probblems": [
    "problems",
    "problem's",
    "problem",
    "parables",
    "parable's",
    "proclaims",
    "prelims",
    "Pablum's",
    "Pribilof's",
    "pabulum's"
  ]
}
```

Figure 3: Example output

5.2 Synonym API

For the synonym we used the Words API by BigHugeLabs. This is a partially free API that allows up to 1000 free request for month. The response is also given by a JSON file and in addition to the synonym it also gives if the word can be used as noun, verb or both. This information could be useful to force more elaborate sentences from the user but, as we did not decided how it would impact the score, it was never implemented.



```
⊟ { } JSON
  ⊟ { } noun
    ⊟ [ ] syn
        ■ 0 : "passion"
        ■ 1 : "beloved"
        ■ 2 : "dear"
        ■ 3 : "dearest"
        ■ 4 : "honey"
        ■ 5 : "sexual love"
        ■ 6 : "erotic love"
        ■ 7 : "lovemaking"
        ■ 8 : "making love"
        ■ 9 : "love life"
        ■ 10 : "concupiscence"
        ■ 11 : "emotion"
        ■ 12 : "eros"
        ■ 13 : "loved one"
        ■ 14 : "lover"
        ■ 15 : "object"
        ■ 16 : "physical attraction"
        ■ 17 : "score"
        ■ 18 : "sex"
        ■ 19 : "sex activity"
        ■ 20 : "sexual activity"
        ■ 21 : "sexual desire"
        ■ 22 : "sexual practice"
    ⊞ [ ] ant
    ⊞ [ ] usr
  ⊟ { } verb
    ⊞ [ ] syn
    ⊞ [ ] ant
```

Figure 4: Example output

6 Conclusions

This project ended up with a functional Speed-Typer game but a few poor choices were made. The use of LinkedList<String> is not justified once the API takes the whole sentence as an input and not each word individually. This choice was made before the choice of API and it should be changed to a simple String. The way the SwingWorker is being used is not the way it was intended to be. The WordHandler should be running from a SwingWorker and not just publishing simple tasks to be handled by a few SwingWorkers.

In conclusion, if started from scratch, a few of the choices would have been different based on how the API's and the libraries work. Therefore, when doing a project the technologies should be chose before the data structures and not the other way around. Otherwise, even if the project works, some of the solutions might not be the best.