CS2212 Group Project Specification

Version 1.0

Fall Session 2023

---

1. Overview

Nothing makes a document look less professional than having it riddled with piles of spelling mistakes. To save us from the drudgery of learning to spell everything correctly ourselves, spell checkers have become ubiquitous. Originally standalone tools, they have become integrated with word processors, office packages, web browsers, and operating systems. Nowadays, most things we type are automatically spell checked for us and this document was no exception. (So hopefully there aren't any typos in it!)

Building a spell checker is a classic introductory software project. (In fact, it was mine back in the day.) They are conceptually simple in functioning, so they are straightforward to understand in general, but they are complex enough with sufficient moving parts to make them a sizeable project suitable for a team of developers.

The main purpose of this project is to create a desktop application that spell checks text documents. It will be a standalone spell checker, so you are building a special-purpose application that checks the spelling of documents and nothing else. In essence, it will check given documents word-by-word, looking them up against a dictionary of known-to-be-correct words. On finding a mistake, your application will provide a variety of correction options, including manual corrections, and suggestions for auto-corrections. By the time a user is done, they will have a correctly spelled document that is ready to go!

This document outlines the general specifications for this project. Separate documents within OWL will provide particular specifications for each stage and milestone for the project.

2. Objectives

This project is designed to give you experience in:

- applying the principles of software engineering towards a real-world problem
- working with, interpreting, and following a detailed specification provided to you
- creating models of requirements and design from such a specification
- implementing your design in Java and having to deal with decisions made earlier in the design process
- creating graphical, user-facing content and applications
- writing robust and efficient code
- write good, clean, well-documented Java code that adheres to best practices
- reflecting on good/bad design decisions made over the course of the project

The project is intended to give you some freedom in design and programming to explore the subject matter, while still providing solid direction towards reaching a specified goal.

3. Requirements

Your project will need to adhere to a collection of functional and non-functional requirements. In essence, the functional requirements outline what your application will need to do, while the non-functional requirements specify how you're supposed to go about doing things.

3.1 Functional Requirements

Functional requirements include required functionality, as discussed in the sections below. While this functionality is required, you still have a fair number of design choices to make along the way, as well as opportunities to exercise creativity.

3.1.1 Required Functionality

You must implement all of the required functionality for your project to be considered complete. How the below functionality is delivered is for the most part up to you. While we will not be grading visual appeal or aesthetics directly, if things slide to the point where your application is unintuitive, difficult to use, or unreadable, then this could impact your overall grade.

General Workflow

The general workflow for using your application is as follows: user opens a file, user spell checks the file, user saves the results. This process can be repeated multiple times within a session, with the user spell checking multiple files in this fashion. Technically, the saving part is optional and the user may choose to discard their changes on loading a new file or exiting the program. (They should of course be warned before doing so, however.) The results could either overwrite the loaded file, or they could save a copy into a new location. It is up to the user.

Documents

Your application primarily needs to support spell checking plain text files. That is, files containing nothing but ASCII text. You do not need to worry about spell checking Word, PDF, or other file formats. (Though, if you figure that out, feel free to include it and make it a thing!) You should avoid making other assumptions about the documents, as there is no limit on line length, file length, etc.

To allow spell checking of HTML/XML files, the user should optionally be able to filter or skip tags in the document so that things like <html> or <li> would be bypassed during the check. In essence, text enclosed within <> would be skipped during a check of a file containing such markup, provided that the right option or setting was in use.

Dictionaries

A key part of any spell checker is its dictionary. This is the source truth for all correctly spelled words that the spell checker knows about. It is up to you to select a dictionary for inclusion with your application, though you must use an English-language dictionary. There are many sources online; https://github.com/dwyl/english-words appears to be a good source, with its words_alpha.txt file containing a sorted list of words containing only letters. As noted, the choice is up to you.

How you want to work with the dictionary is up to you. You can either process and load the dictionary into memory when your spell checker runs, or you can try to use it on disk. Your spell checker does need to perform decently (as noted under below under Non-Functional Requirements), so be sure to keep that in mind when making choosing your approach and any data structures used to house the dictionary. Hashing is commonly used, but again it is up to you to decide what's best for your application.

In addition to the built-in "system dictionary" that is immutable and part of your application, you must also support a user dictionary. This contains additional words not in your system dictionary that the user deems to be correct. Often this will contain names, but may contain other things that the user wants to be treated as correctly spelled. Generally speaking, words will get added to the user dictionary as the user is spell checking documents. (Adding a word to the user dictionary is one correction option that is to be presented to the user, as discussed below.) Options should also be presented to the user to allow them to edit or remove words from the user dictionary, or to reset it back to an empty state, which is the default when a user first uses your application.

Words

Before going on, we should have a working definition of what exactly constitutes a word for us here. In general, it's likely best to view a word as a set of characters separated by some sort of white space (space, tab, newline, etc.) or punctuation (period, comma, etc.) . Ultimately, there is some wiggle room on things though, so in the end, it somewhat depends on where you source your dictionary. (After all, if that's the way your application tells if something is a valid word or not, there's going to be some dependency there.)

For example, what should you do with hyphenated words? Take "co-worker" for instance. Should you spell check it as "co-worker", "coworker" (removing the hyphen), or "co" and "worker" (treating things as two separate words). This is a good example of one of those "it depends on your dictionary" sort of things. I'm guessing that splitting the words and treating the hyphen as a separator is the best way of handling things for most dictionaries, but you will have to decide what's best for you, your app, and your dictionary.

Prefixing and suffixing words might also be a consideration, depending on your dictionary. The English language contains a number of base words that can be modified by adding a variety of prefixes to the beginning of the word and suffixes to the end of the word. Take "configure" as an example. A valid prefix is "re" giving us "reconfigure" and a valid suffix is "ed" giving us "reconfigured" (removing the extra "e" at the end of the word as you would do in adding a suffix here). There are a pile of rules governing which prefixes and suffixes are valid for which base words, as well as how to properly apply them. So you have a couple of choices. If you choose well, your dictionary simply contains all the possible words, with prefixes and suffixes already included appropriately. (The sample dictionary above seems to do a good job of this.) Alternatively, you can use a more compact dictionary, and then apply prefix and suffix rules to base words as you go. It's more complex language-wise, but can have big savings on the dictionary size, which can make things more efficient. Which approach you use is up to you. (Though I'd probably recommend the first approach, using a dictionary with all of this sorted out for you already.)

Another question is how to handle words with embedded numbers. If a "word" is nothing but numbers like "007" or "9999", it's likely best to just ignore it entirely and not check its spelling. If it is a mixture of things like "l00t", it is likely best to treat things as a word and check it for correct spelling, as someone might have accidentally inserted the numbers there by mistake. As long as you handle such things consistently, that's the main thing.

A final thought would be what to do with things like filenames (such as "document.txt") or websites (like "google.com"). In practice, it's difficult to tell when the user is intending to use words in this fashion, so it's likely better to use the "punctuation as separator" rule and treat these as separate words ("document" and "txt", and "google" and "com" respectively). You might get extra errors that way, but the user can always accept or correct things as need be.

In the end, again, how you treat words depends somewhat on your dictionary, so if you need to adjust rules somewhat as a result for this, feel free to check with your TA and proceed accordingly.

Error Detection

Your application must check for the following types of errors:

- Misspellings. These are words that cannot be found in either the system dictionary or the user dictionary and so are considered to be spelling mistakes.
- Miscapitalizations. These are words that are incorrectly capitalized in some fashion. Capitalization issues include:
  - Not capitalizing the first word of a new sentence. For example "the dog is big." will throw a capitalization error on "the" because it is the first word of the sentence.
  - Mixed capitalization within a word. Spelling dog as "doG" would throw a capitalization error as the g is apparently capitalized for no reason.
- Double words. This is the repetition of words such as "this this" or "pizza pizza". Such an occurrence might be fine, but it is often a mistake in the documentation, and one that is easy to correct. A double word error should be flagged even if the capitalization of the words is different, so "This this" is still a double word issue.

Please note that you may choose to add additional rules for flagging issues if you would like to do so. For example, spelling dog as "Dog" mid-sentence is unusual, but could be a proper name and so capitalization wouldn't be problematic. Similarly, spelling it in all caps (like "DOG") should also be fine. Some spell checkers will allow anything to appear in all caps, as it might be used as an acronym. For example, "asdf" would be considered a spelling mistake, but "ASDF" might not, as it could be considered to be an acronym. Whether you would want to flag that as an error or not is up to you. (If you chose to consider it as an error, the user could always add it to their user dictionary to bypass the problem that way.) Some spell checkers will consider things like "dr." or "mr." as errors without the capitalization, whereas "Dr." and "Mr." would be considered correct. How you would like to handle these words is also up to you. As long as you catch the three types of errors as noted above, the rest is up to you and your group.

When an error is detected in checking a document, it must be somehow presented to the user. You should do so in context, showing a reasonable amount of surrounding text, with the error highlighted relative to the rest of the text. (Showing the user just the error with no surrounding text might not give them enough information to correct things, so the more context you can provide, the better.) How you choose to do this exactly is up to you. For instance, you could load the document into a larger text panel, and focus the panel on the appropriate line and word when an error is found. The word(s) in error could then be bolded, underlined, shown in reverse text, shown in red, etc. to indicate that they are the issue in the text being shown. Of course the nature of the error should also be reported, whether it's a misspelling, miscapitalization, double word, and so on.

Error Correction

When an error has been detected in the supplied document, a few options should be provided to the user to help correct the problem:

- Manually type a correction.
- Select a correction from a list of possible suggestions. The possible suggestions generated depend on the type of issue:
  - For misspellings, possible replacement words will be suggested by determining possible replacement candidates and checking to ensure that the candidates exist in either the system or user dictionary before being suggested to the user. Candidates are generated by:
    - Removing each letter from the word to see if that creates a valid word. For example, the word "golod" would have "gold" and "good" as suggestions by removing an "o" or the "l" respectively.
    - Inserting each possible letter into each possible position to see if that creates a word. For example, the word "ther" would have "their" and there" as suggestions, made by inserting an "i" or "e" at the appropriate locations. (The words "her" and "the" would also be suggested through removal of letters too.)
    - Swapping each pair of consecutive letters around to see if that creates a word. For example, the word "theer" would have "there" suggested by swapping the last "e" and "r" in the word.
    - Inserting a space or hyphen at each possible interior position of the word to create two new valid words or a valid hyphenated word. For example, the word "notthe" would have "not the" suggested as a correction by inserting a space appropriately.
  - For miscapitalizations, possible suggestions depend on the issue:
    - For missing a capital letter at the beginning of a sentence, the capitalized word is suggested.
    - For mixed captializations, you can suggest the word with no capitalization, all letters capitalized, or just the first letter capitalized.
  - For double words, no suggestion needs to be given; the user can simple opt to remove the offending second instance using the deletion option given below.
- Delete the word containing the issue.
- Ignore the issue this one time. (If it is encountered again in the same file, the error will be raised again.)
- Ignore the issue for the rest of this application session. (The issue will not be raised again until the application is restarted. As long as the application continues to run, however, the issue will not be raised again in the same file or in other files checked.)
- Add the offending word to the user dictionary so that the issue is never raised again.

Additional correction options can be provided if you would like to do so. Similarly, other methods for formulating suggested corrections can be used as well if you have additional ideas. As long as the above is supported, that is the main thing.

Metrics

Your application should compute a few metrics as part of its operation and report them to the user when the spell checking process is complete. These metrics include:

- Number of characters, lines, and words in the document.
- Number of each type of error detected in the document. (E.g. number of misspellings, miscapitalizations, and double words encountered.)
- Number of each type of correction used in fixing errors found in the document. (E.g. number of manual corrections, accepted suggestions, word deletions, etc.)

Housekeeping

Your application also needs to do basic housekeeping things that are part of any decently put together application. For example, the user must be able to exit your application cleanly and must be able to access help of some kind if they get stuck. You may add other housekeeping functions as you see fit to help support your application. How you choose to make these available to the user is up to you.

Persistence

Your application needs to store some data persistently. This includes the user dictionary that is built during a spell check, as well as any other application settings that should persist between sessions. In the case of settings, a configuration file should be stored somewhere appropriately (off of the user's home directory, for instance) so that it can be found readily on subsequent launches of your application.

3.2 Non-Functional Requirements

Your application will need to adhere to the following requirements, and these requirements will be taken into consideration in the assessment of your project.

- The application must be developed in Java, designed to run on a user's desktop as a standalone application.
- The application will need to use a Java graphical user interface of some kind, with Swing and JavaFX being the leading options; the choice of framework is up to you, though you will need to standardize across your group.
- The application should perform well, without an excessive load time (to set up the dictionary for checking) or excessive run time (to find words in the dictionary during a check).
- The application should not need to use any other libraries. If you do wish to use another library, you must obtain written consent (in e-mail form) from the instructor prior to its use.
- All code in the application must be commented using Javadoc.
- You may choose as a team the coding conventions and styles you wish to adopt in your code (for naming things, indentation, etc.). However, you must remain consistent in applying those conventions and styles across all files in the application. Good rules to follow for this will be discussed in class.
- The application should be executable on systems with a standard Java installation, and each team member must be able to compile it and run it from a development environment they have ready access to. The choice of development environment is up to you.
- The application must be well self-contained and not create, modify, or delete files outside of the directory in which the application is installed, and subdirectories of this directory. The user dictionary and a configuration file as necessary should be stored somewhere appropriate, such as off of the user's home directory.
- The application must present a visible response to every user action. Erroneous actions or actions that could not succeed for some reason must be met with a useful, professional error message.
- The application must be designed with sound software engineering principles in mind.
- Project code must be checked into the designated repository assigned to your team and members must commit and push code to the repository regularly. Once groups have been set, we will be posting more information on the project management software to be used for the course, and this document will be updated accordingly.