

Evolutionary and Adaptive Systems MSc.

Programming Techniques Project Report

Spell Checker

By Mike Blow Candidate 80761

16.12.2003

Specification

The basic specification of the application is to:

- Implement a spell checker that takes an input file, identifies the words not in its dictionary and allows the users to change them in an output file.

- Search the dictionary using an efficient algorithm.
- Be insensitive to the case of the letters in the word, unless they are listed in the dictionary as requiring an initial capital.

In addition, this application will:

- Allow the user to extend the dictionary file with any words that they have indicated are acceptable. An accepted word should not be flagged as incorrect for the remainder of the run. A replaced word should match the capitalisation of the original word.
- Learn common misspellings and use these to make appropriate suggestions for corrections.

I decided on this specification as it gives useful functionality without being overly complicated. It also has a degree of flexibility in that it can learn new words and misspellings meaning the more it is used the more useful it will become.

I decided not to implement the other two extensions (plural rules and spelling rules), as my application will cope in a way with plurals and misspelled words. Also implementing lists of rules is a difficult thing, as there will always be the case you had not thought of. My code sidesteps this problem by not using rule-based algorithms.

User Guide

(Note: I am assuming here that the code would be distributed in some sort of executable such as a JAR file).

Introduction

This spell checker is fast and easy to use. It will expand with use, learning new words (especially useful for phrases specific to your occupation) and common misspellings. It can also suggest correct words if a misspelling is encountered.

Getting Started

Double click on the 'SpellChecker.jar' file to begin. You will see the main SpellChecker screen (*fig. 1*).

Press the 'Load text file' button to open the file you wish to spell check (*fig. 2*).

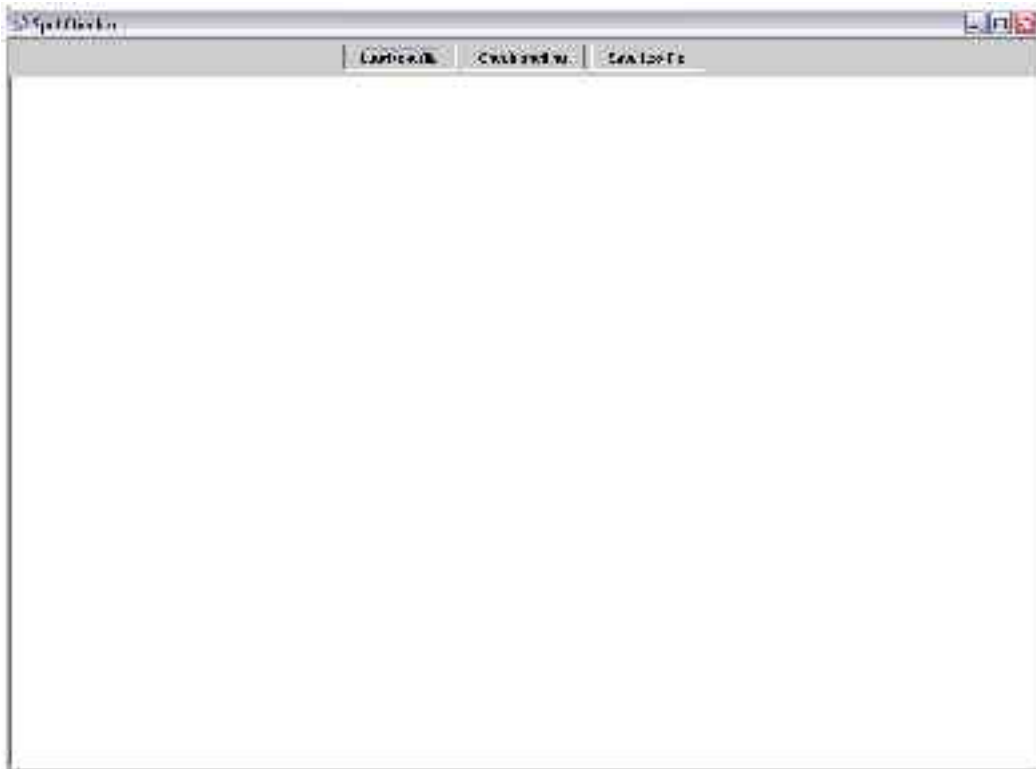


Fig. 1 the main screen

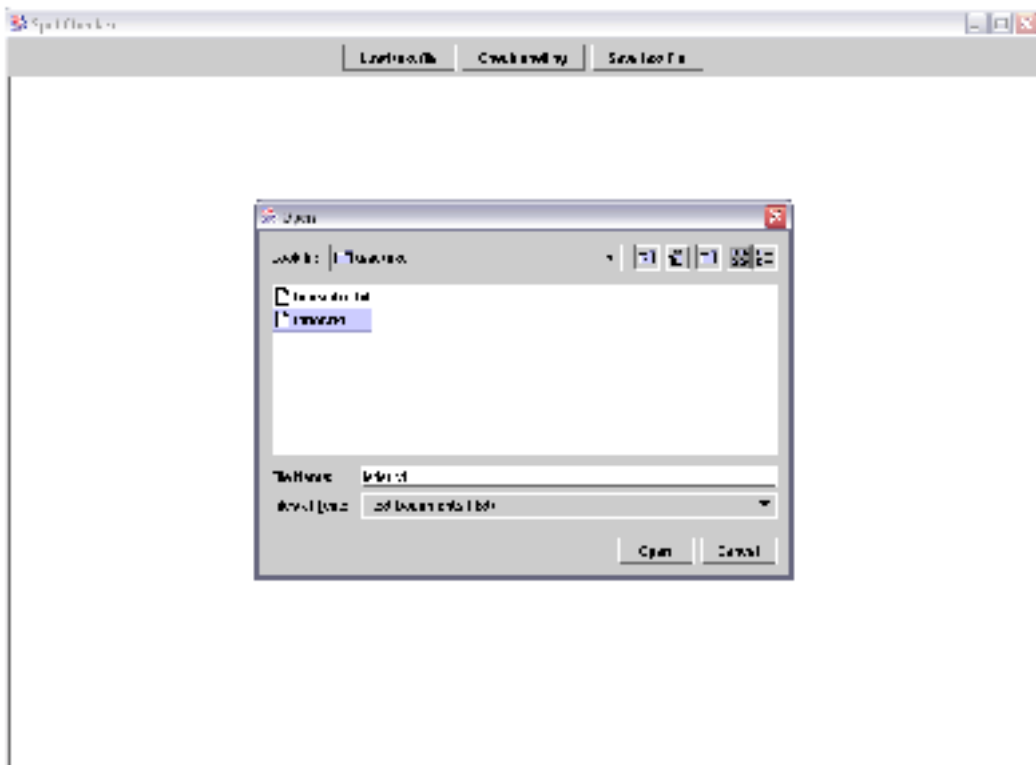


Fig. 2 opening a file

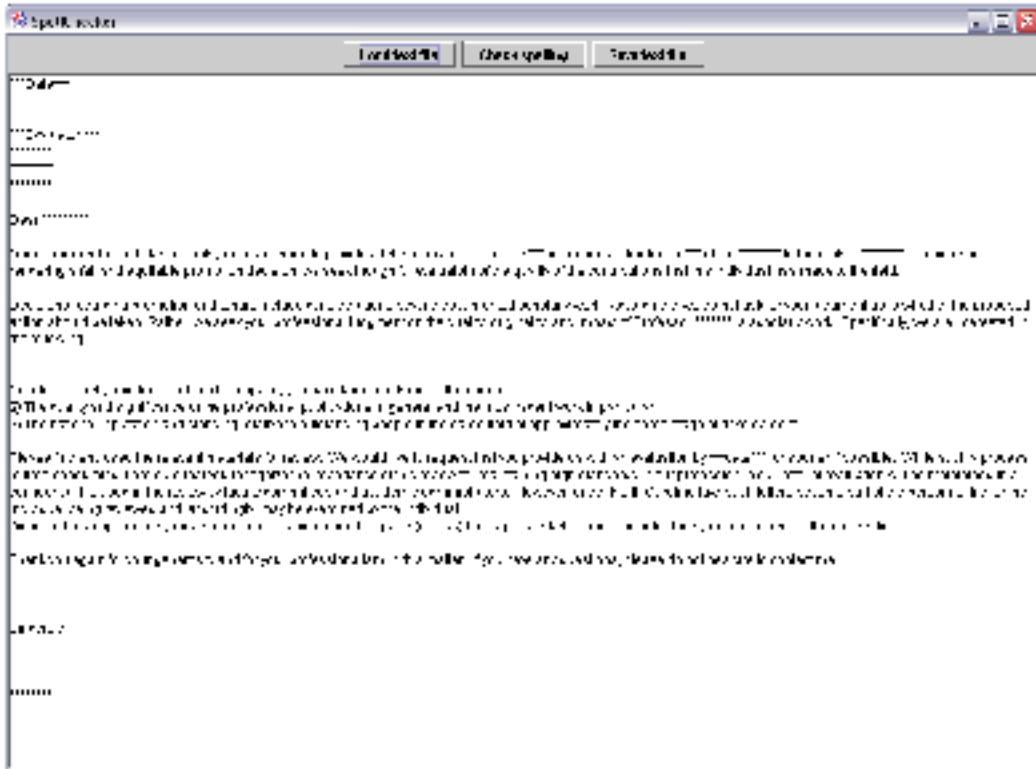


Fig. 3 the text file loaded into the spell checker

Once the document is displayed (fig. 3) you can start to spell check it. Pressing the 'Check spelling' button will start the spell check (fig. 4).

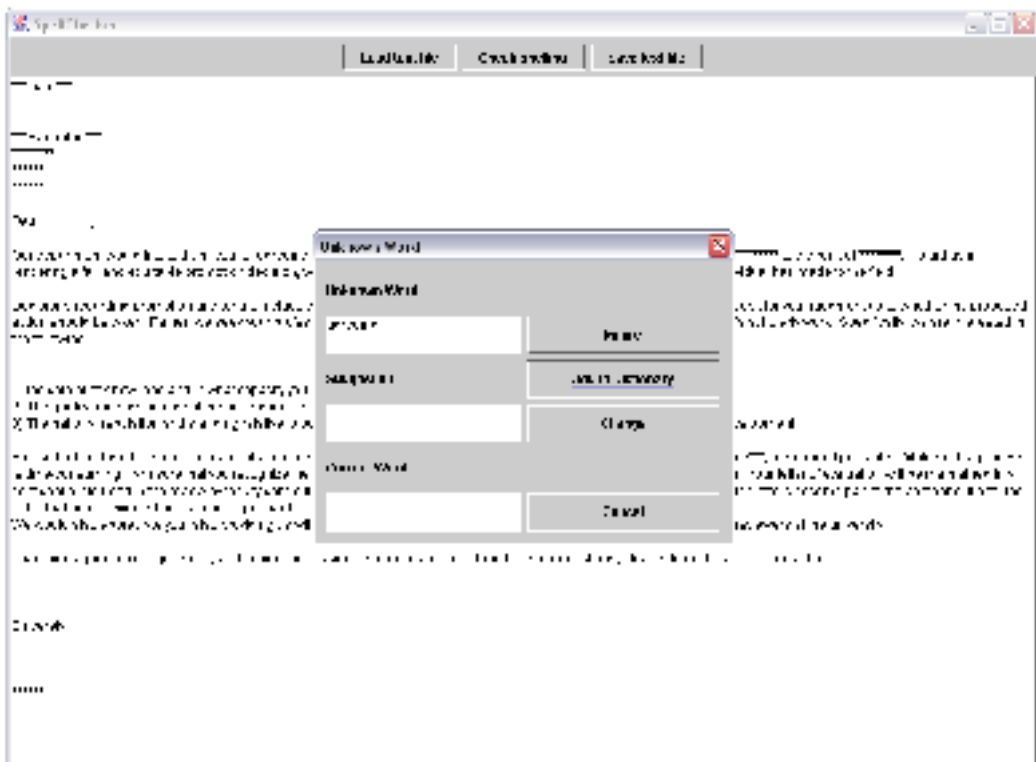


Fig. 4 the spell check in progress

Spell Checking the Document

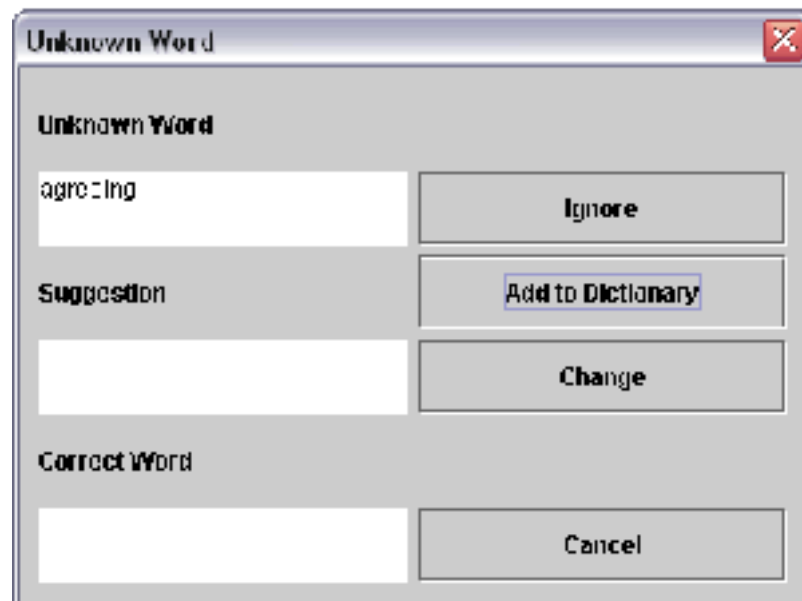


Fig. 5 an unknown word

The spell checker will work through the document checking each word but ignoring whitespace and non-word characters (for instance the ‘***’ strings in the example letter). When it finds a word that is not in the dictionary, it displays it in the ‘Unknown Word’ dialog (*figs. 4 and 5*). You now have three options:

1. If the word is correct but is particular to this document and does not need to be added to the dictionary press ‘Ignore’.
2. If the word is correct and may occur again in the future press ‘Add to Dictionary’. The word will be added to the dictionary immediately, so if the same word occurs later in the document it will no longer be flagged as unknown.
3. If the word is incorrect (*fig. 6*), type the correct word in the ‘Correct Word’ text box and press ‘Change’ (*fig. 7*). The document will be updated with the correct word, and its capitalisation will match that of the original word.

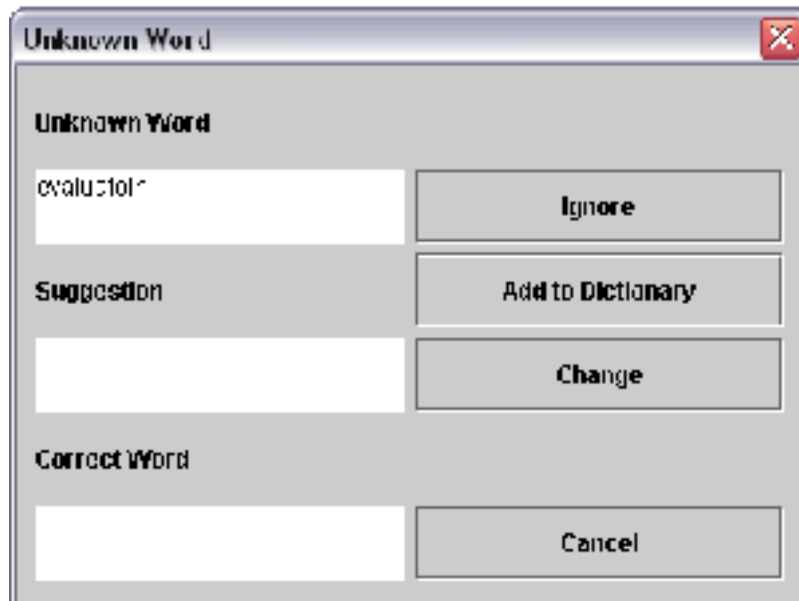


Fig. 6 an incorrect word

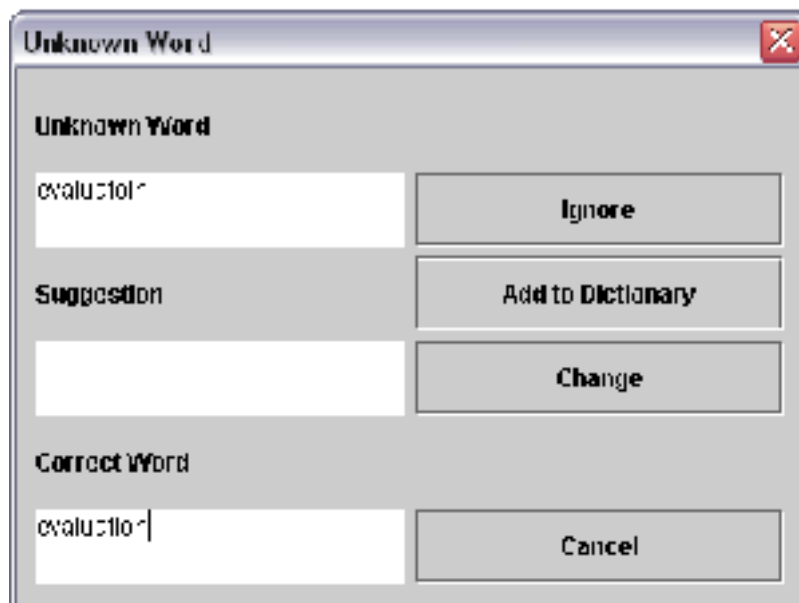


Fig. 7 correcting a word

Finally if the correct word you have typed is not already in the dictionary you are asked if you would like to add it (*fig. 8*).

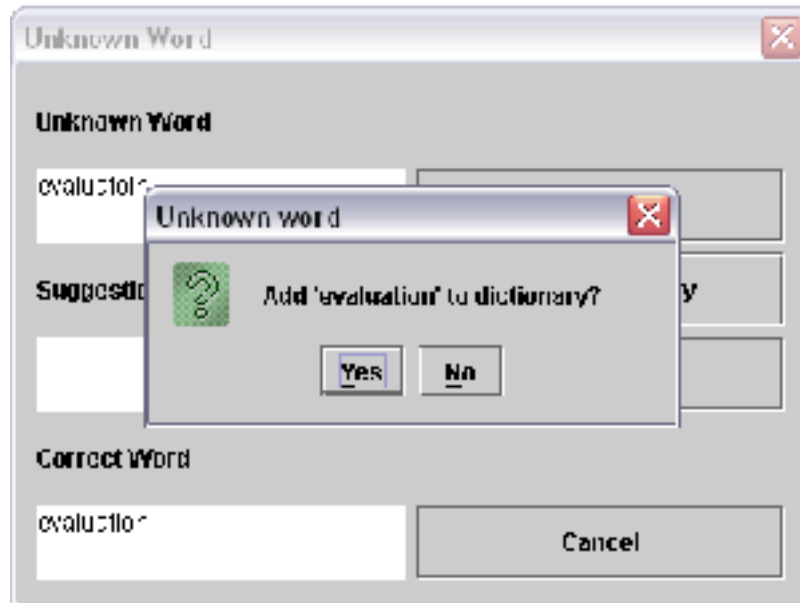


Fig. 8 adding a corrected word

Suggested Words

The spell checker will build up a list of commonly misspelled words as they are corrected. In future, if the same misspelling is encountered, the program will offer the correct word as a suggestion (*fig. 9*). You can copy and paste the suggested word into the correct word text box.

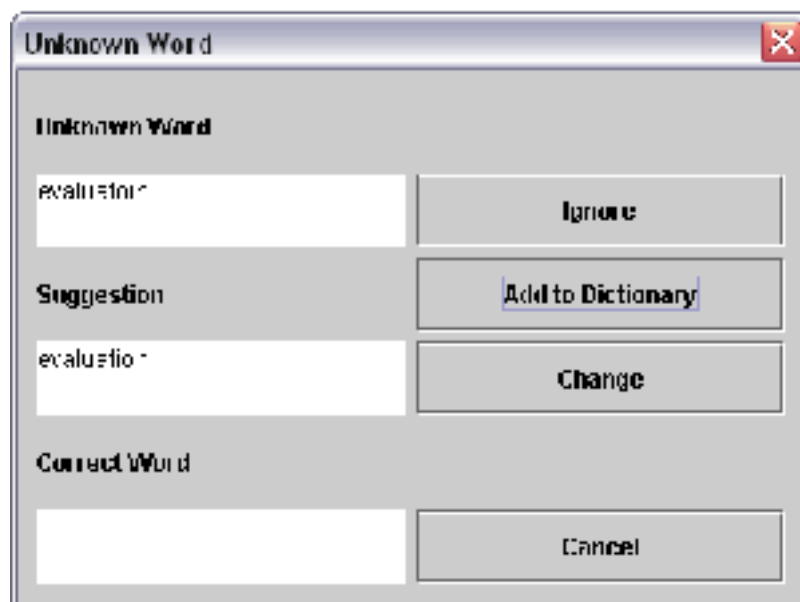


Fig. 9 a suggested correction

The spell checker will notify you when it has finished checking the document (*fig. 10*). Please note that any words and misspellings that have been added to the dictionary during the check are saved and will be used next time you run the program.

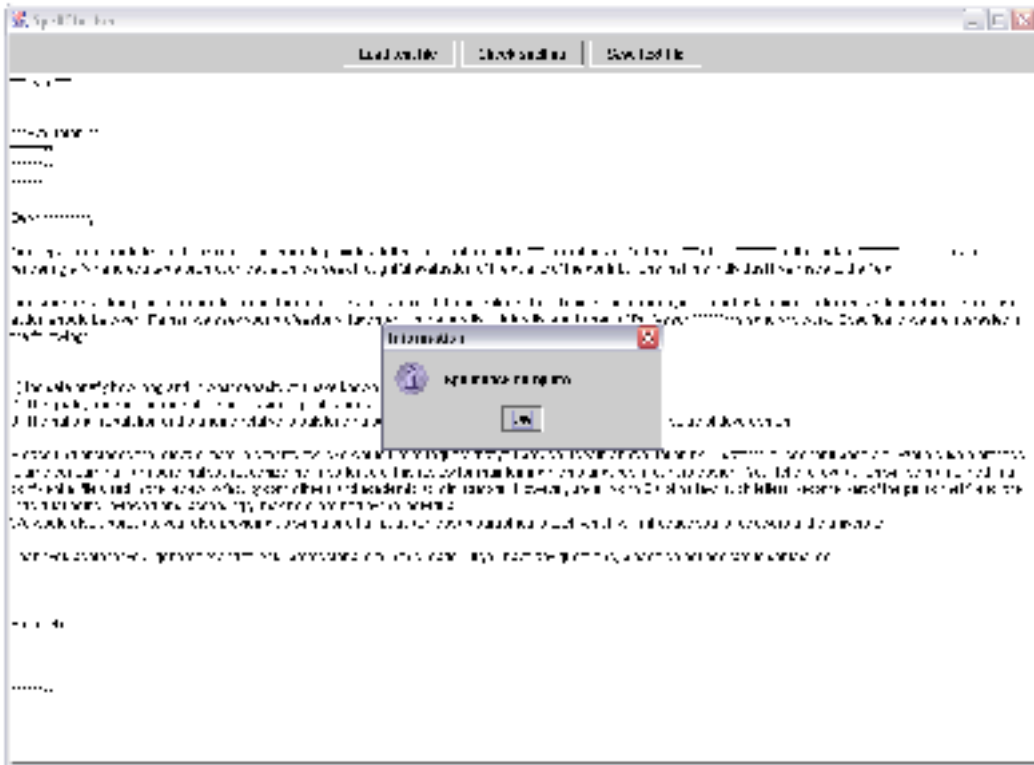


Fig. 10 spell check complete

Once you have spell checked your file the corrected version can be saved to disk by pressing the ‘Save text file’ button (fig. 11).

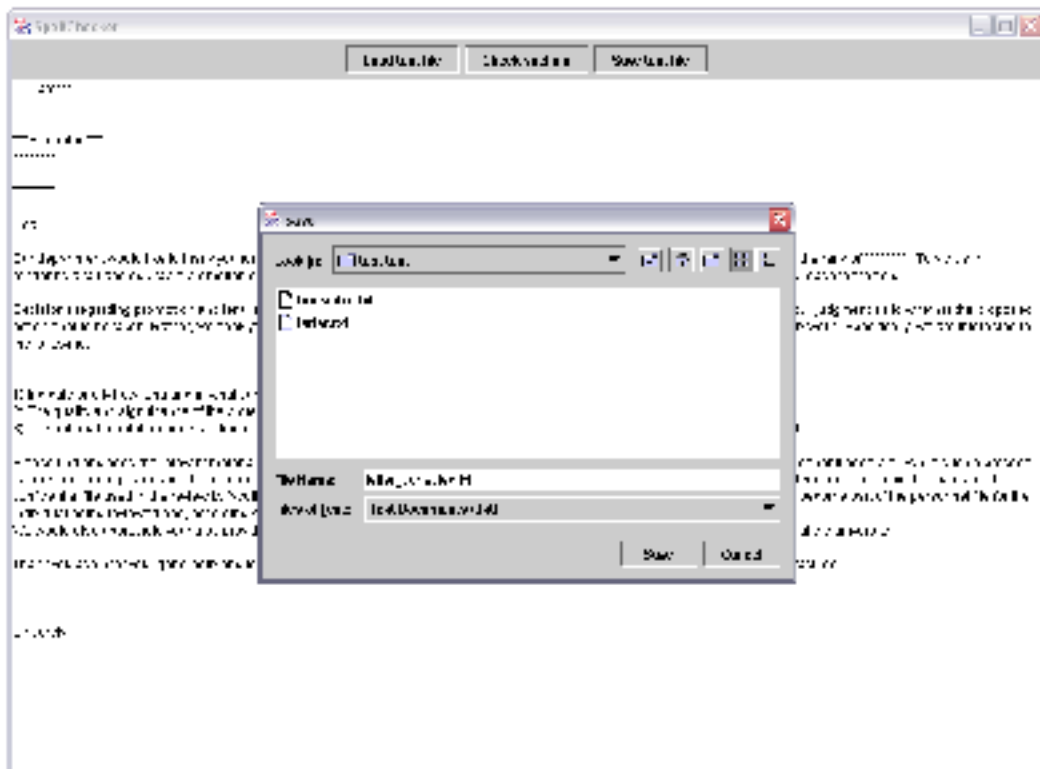


Fig. 11 saving the checked document

Finally load another file to check using the ‘Load text file’ button or close the program by pressing the red cross in the top right hand corner of the frame. The program will warn you if you have changed your document and not saved it.

Design

Implementation

The system is implemented in Java. I decided to use Java for several reasons:

1. Java would be a good choice if this were a commercial product as it is cross-platform and the code would, in theory, require no rewriting to be deployed on any platform that supports a Java Virtual Machine.
2. Java is also very well suited to this sort of application having extensive in-built data structures, GUI facilities and documentation, facilitating fast development and small class size.
3. The performance issues that are cited as the main reason not to use Java are, I believe, not too relevant in this situation. The time the user will take deciding what action to take over a certain word will far outweigh the time taken to process the text.
4. Object oriented programming is a definite bonus in this sort of application, aiding readability and maintenance and allowing code to be re-used as required.

Program Structure

The system is implemented as six classes, three main functional classes and three utility classes. Fig. 12 shows the relationship between the objects and the public methods in each class.

The class functionality is as follows:

SpellCheckerGUI

The main, runnable class, SpellCheckerGUI creates the main user interface frame. It allows the user to load, check, correct and save text files.

Dictionary

Dictionary holds two hashtables of words; the first is simply a list of known correct spellings that are used to check the words in the document. The second is a hashtable of common misspellings. It saves and loads both from disc when the program is started or closed.

CheckWordDialog

This dialog flags any unknown words, and allows the user to do one of three things: ignore the word, add it to the dictionary or change it.

JTextAreaPlus

This class is an extension of JTextArea that adds String array capabilities to the parent class.

Spacer

Spacer is a GUI component of fixed, user-definable size. It is useful for aligning components in a GUI layout.

External Code

ExampleFileFilter is a convenience implementation of the FileFilter interface written by Jeff Dinkins and distributed with Sun's Java SDK*. It allows the author to create file chooser dialogs that will filter out any unspecified file types.

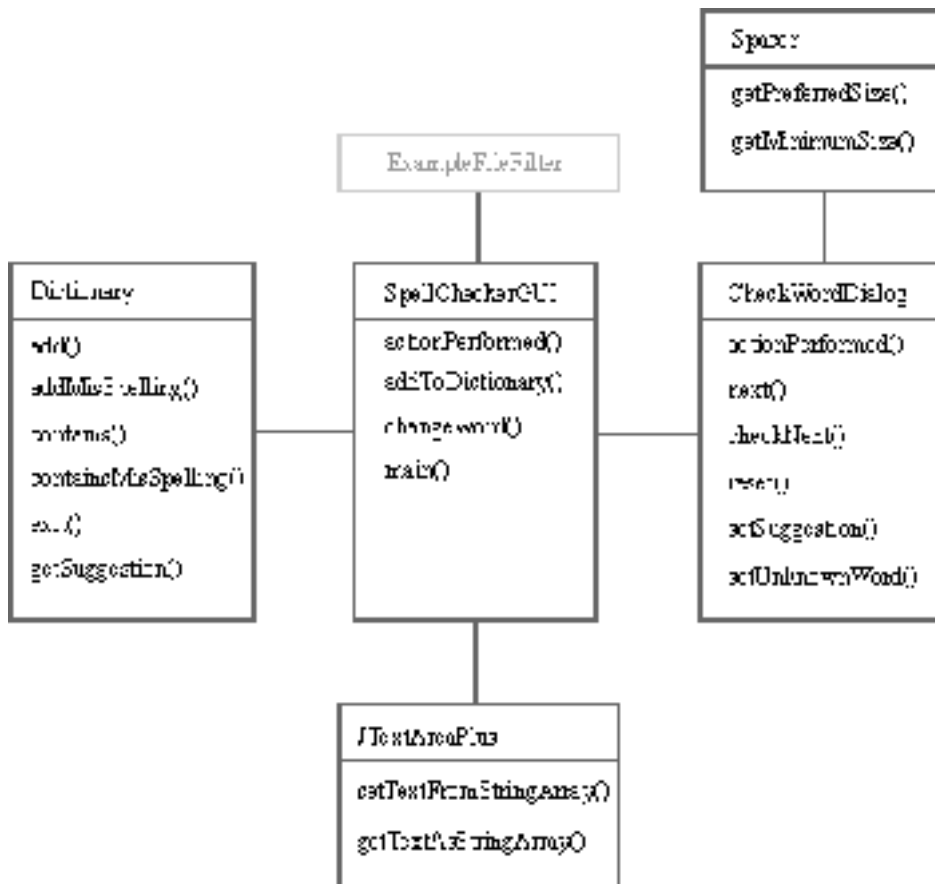


Fig. 12 the program structure, classes and public methods

The Main Functions of the Program

Below there is a list of the main functions of the code with an explanation of each. Rather than just list each method I have given a more detailed explanation of how the code works with references to the relevant methods.

* <http://java.sun.com/j2se/>

Upon startup the program checks whether there are existing Dictionary and MisSpellings hashtable objects on disc from previous runs and if so loads them using a private method Dictionary.loadSerialized(), which uses an ObjectInputStream. If there are no existing objects the dictionary is loaded from a text file and a fresh misSpellings hashtable created.

To load a text file, SpellCheckerGUI.loadText() is called. This method creates a file chooser that filters out any files without a '.txt' suffix using the ExampleFileFilter and reads in the selected file as a string using a RandomAccessFile object. The string is then split into a string array using SpellCheckerGUI.convertTextToStringArray(), which is necessary in order to manipulate the text, e.g. replace words that are misspelled. This method splits the string into words and non-words. Initially I was using a StringTokenizer to split the string at various punctuation tokens (space, period, comma etc.) but soon realised that once again there would always be the unanticipated situation, and so simply changed to the word/non-word strategy. This takes a little more space to code but is far more robust. Once in a String array, the text is passed to the JTextAreaPlus.setTextFromStringArray() method to be displayed in the GUI. The JTextAreaPlus is an extended JTextArea that can accept String arrays as input and can return the displayed text as a String array (this latter facility is not used in this product but I wrote the method in the object for completeness. It uses the same string splitting code as SpellCheckerGUI and I did consider passing the text to the JTextAreaPlus to be split; but decided that the code would be more strongly encapsulated if the two objects each had its own string splitting routine).

To check the spelling SpellCheckerGUI sets a global word index to zero and calls a recursive method, SpellCheckerGUI.checkText(). This method firstly checks whether the string at the index is in fact a word, and then whether it is in the dictionary by calling Dictionary.contains(). If it is, the method increments the index and calls itself to check the next word. If it is not, the method constructs a CheckWordDialog (or sets the existing one blank, if it has already been called) and displays in it the unknown word using CheckWordDialog.setUnknownWord() and a spelling suggestion if one is found using CheckWordDialog.setSuggestion(). The spelling suggestion is gathered from a separate hashtable of common misspellings by calling Dictionary.containsMisSpelling() and Dictionary.getSuggestion(). The misspellings hashtable is updated using Dictionary.addMisSpelling() whenever the user corrects a misspelled word. There are 4 options in the CheckWordDialog:

1. The user can press '**Ignore**'.
2. The '**Add to dictionary**' button causes the unknown word to be added immediately by calling SpellCheckGUI.addToDictionary() which in turn calls Dictionary.add(). Henceforth the word will not be flagged as unknown. The next variable is set true and the dialog set invisible.
3. The '**Change**' button causes the word in the text file to be replaced with the correct one from the dialog. SpellCheckGUI.changeWord() is called which calls Dictionary.addMisSpelling() and updates the text in the text area. Dictionary.addMisSpelling() puts the correct word in the misspellings hashtable with the incorrect word as the key. If a word is not known in the dictionary the misspellings hashtable keys are checked and if the misspelling is there the correct

word can be retrieved and offered as a spelling suggestion. In all these cases, the dialog sets a boolean next variable to true and sets itself invisible. This boolean is checked by the SpellCheckerGUI.checkText() method to see whether to display the next unknown word.

4. If 'Cancel' is pressed however, the 'next' boolean is set false and the next word is not displayed.

To save the updated document, the SpellCheckerGUI.saveText() method again uses ExampleFileFilter to select a '.txt' file. If the file already exists the program checks that it is OK to overwrite, and then writes the text in the JTextAreaPlus to disc in the form of a String of bytes, using the RandomAccessFile.writeBytes() method.

The code includes listeners for Window closing events, and if the close button is pressed the SpellCheckerGUI.exit() routine is called. Firstly this sets the window invisible to give the impression of immediate response, and then calls Dictionary.exit() which writes the dictionary and misSpellings hashtables to disc by calling the Dictionary.serializeWordFile() method on each. This method uses an ObjectOutputStream.

The Main Algorithms and Data Structures

The spell check algorithm can be represented thus:

Check if word is in dictionary

If it is not:

- display unknown word in dialog**
- check common misspellings and display suggestion if found**
- update misspellings when a word is corrected**
- update dictionary as required**

Move onto the next word

The idea behind using this algorithm is that it provides a lot of functionality for relatively small code size. This is a good thing for several reasons; speed of execution and ease of maintenance being the main ones, crucial factors in a commercial product.

The data structures used are two hashtables. These were chosen for ease of use and speed. The main dictionary hashtable simply uses the word as the key and object, but the misspellings hashtable uses the incorrect word as a key to retrieve the correct word. In this way if the misspelling occurs again it can be recognised and the correct word offered as a spelling suggestion.

A binary tree would arguably provide more functionality in that you have access to all words without needing a key, but the checking of these words adds complexity and is still not foolproof. For instance you could return suggestions based on similar spelling but what if the first letter is incorrect? I feel this is where my system scores highly – minimum code for good and useful results.

Expandability

Firstly, the fundamental fact that the code is object-oriented allows for expansion in the future as classes can be reused and altered easily.

The system is to an extent self-expanding in that it allows new words to be added to the dictionary and constantly updates its misspellings list. When the application is closed these data structures are saved out as objects, and then loaded at the start of the next run. In this way any additions made are retained and added to with subsequent use.

A useful addition would be a dictionary editor, allowing the user to maintain the dictionary, for instance to remove incorrect words that had been added by mistake. This could quite easily be designed to load/save the appropriate hashtable and display it in some kind of editor.

Limitations

The main strength of this system is also its weakness - its reliance on the user to add unknown words. Of course if this is used properly it will result in a useful and expanding dictionary but it is open to abuse and mistakes. For this reason a dictionary editor (see 'Expandability') would be useful.

Testing

The program was tested regularly throughout development with a variety of different text files. It was during this testing that it was decided to rewrite the text parsing routine as differentiating simply between word and non-word characters rather than checking an ever-growing list of punctuation marks.

It has been tested loading in a new text file with one already open, and will cope with missing dictionary files (it tells the user it cannot find the dictionary, and then simply starts with a blank dictionary and adds words to it).

As with all code the program would benefit from beta testing by a group of independent testers, who always manage to find the things the author did not consider!

Some notes about the code

OO principles

The code has been written with Object Oriented principles in mind, using inheritance to keep the class sizes small and making all methods private unless they need to be accessed from outside the class.

Imports

It may be noticed that I generally import individual classes as much as possible rather than a whole package – this helps with debugging and readability as it is easy to see what classes are used. Usually when I get to 5 or 6 imports from the same package I will import the package as a whole.

ActionPerformed

The actionPerformed method can be a very good place to hide important code. I prefer to keep it minimal and call other methods to do the work; these separate methods are more easily debugged and make it simple to work with the code as they show up in a development environment's method viewer. Of course it also means these methods can be called from elsewhere in the code.

Error Handling

Java provides extensive error handling capabilities with the *try* and *catch* statements and a list of specific error messages, which I have used in the code. In a similar way to imports, it is possible to merely catch the superclass 'Exception' on every occasion but I prefer to catch the specific errors where possible, a strategy which leads to clearer and more precise code.

Conclusion

I hope I have shown this program to be a useful spell checking tool with several good attributes; it is fast, cross-platform, expandable and has a small code footprint. It is also robust enough to be used as the basis for a more powerful spell checking system.

References

These references were useful in the creation of this program:

Horton, Ivor. (1997). *“Beginning Java”*. Wrox Press.

Walnum, Clayton. (1996). *“Java by Example”*. Que Corporation.

Java on-line documentation from Sun Microsystems, available on the web at: <http://java.sun.com/j2se/1.4.2/docs/api/index.html>

Appendix A

The Code

Java 1.4.2