Spell Checker for EiffelStudio

Software Engineering Laboratory: Open-Source EiffelStudio

Semester project by Benjamin Fischer
Supervised by Julian Tschannen
Prof. Dr. Bertrand Meyer

Student number: 10-916-971
ETH Computer Science
17 February 2013

1 Requirements engineering

1.1 Introduction: Purpose and scope of the system

EiffelStudio is a development environment for the Eiffel programming language. It currently does not have a spell checker. Thus, the idea of this project is to integrate a spell checker into EiffelStudio in order to check the spelling of class and feature comments. The first part of the project is about developing a spell checker library in Eiffel, which is used in the second part to integrate a spell checker tool into EiffelStudio.

1.2 Proposed system

The first part with the spell checker library is the core, while the possible additional features of the second part are only nice to have.

1.2.1 Functional requirements

Part 1: Eiffel library to check the spelling of strings

- Select a spell checking back end
- Select the language
- Check the correctness of a word (ASCII or UTF32)
- Check the correctness of a sentence (ASCII or UTF32)
- Get suggestions for an incorrect word
- Get suggestions for incorrect words in a sentence

- Support a user dictionary and an ignore list, both of which can be created, added words to, loaded and stored as well as queried for list of words
- The spell checker library should make multiple back ends possible. Support for at least one back end should be implemented already, for example GNU Aspell using its command-line interface.

Part 2: Spell checker tool for EiffelStudio

- Minimum requirements
  - Push a button to launch the spell checker
  - Check class and feature comments
  - Show spelling suggestions as warnings in EiffelStudio's error list
- Possible additional features (in no particular order)
  - Optional requirements
    * Check a class name (ignoring a possible class prefix)
    * Check feature names, argument names and names of local variables
    * Check the content of manifest strings
    * Check tag names of assertion and note clauses
  - Fancy requirements
    * Ability to add words to a user dictionary
    * Ability to ignore words for a specific project
    * Automatic spell checking after saving a file
    * Show spelling suggestions and spell checker options in its own tool
    * Correct class text automatically when spelling suggestion is selected

1.2.2 Nonfunctional requirements

- The source code meets the style guidelines for Eiffel as summarised by Marcel Kessler in 'DO IT WITH STYLE – A Guide to the Eiffel Style' at `http://se.inf.ethz.ch/courses/2012b_fall/eprog/additional_materials/style_guideline_summary.pdf`.
- A student in the second year of computer science at ETH should be able to understand and use the spell checker library by the end of one working day.
- Most of the source code should document itself in form of a good programming style with contracts, comments, tests and usage examples. The documentation of the project with requirements engineering, design and testing is summarised and delivered with this final report.

- The Eiffel source code should be void-safe.

2 Design

2.1 Design goals

The main objectives of the software in order of decreasing priority are as follows.

1. Maintainability. Both the spell checker library and its integration into EiffelStudio will be developed from scratch. The library might be used by other developers who directly use the source code. For this reasons, it is particularly important to prepare the software for changes. The most important objective is to make the software maintainable and hence easy to understand as well as improve. It should be possible to add new back ends without significantly changing the other parts of the system.

2. Usability. The library should have an intuitive interface for other developers using it. There should be clear documentation in both formal language like good contracts and informal language in comments. What is more, usage examples for the library should be provided.

2.2 Detailed design

In the following, the spell checker library as seen in the BON class diagram of figure 1 is presented. We give an overview of the structure.

Making use of a facade pattern, the central functionality comes together at the class `SC_SPELL_CHECKER`. It has an associated language represented by `SC_LANGUAGE` in which the current spell checking is happening. The results of checking the spelling of words are given as objects of the type `SC_CORRECTION`, which encapsulates a correction with suggestions for replacements if any. The service of a raw spell checker is abstracted into the deferred class `SC_BACK_END`. Its only effective descendant at the moment is `SC_GNU_ASPELL` for the GNU Aspell spell checker, following an adapter pattern.

For the need of user and ignore dictionaries, the deferred class `SC_WORD_SET` provides an interface for a persistent set of words, realised by an effective heir `SC_WORD_SET_XML` for an XML file format. With reusability in mind, general text processing queries are kept in the class `SC_LANGUAGE_UTILITY`, which frequently serves as an ascendant for implementation inheritance. The raw data of the required Unicode properties is found in its parent `SC_LANGUAGE_DATA`. The class `SC_USAGE_EXAMPLE` gives source code examples of how to use the spell checker library. At the same time, it serves as a demonstration and tests the whole system. Finally, tests of the more hidden parts of the library are collected in the class `SC_TEST_SUITE`.

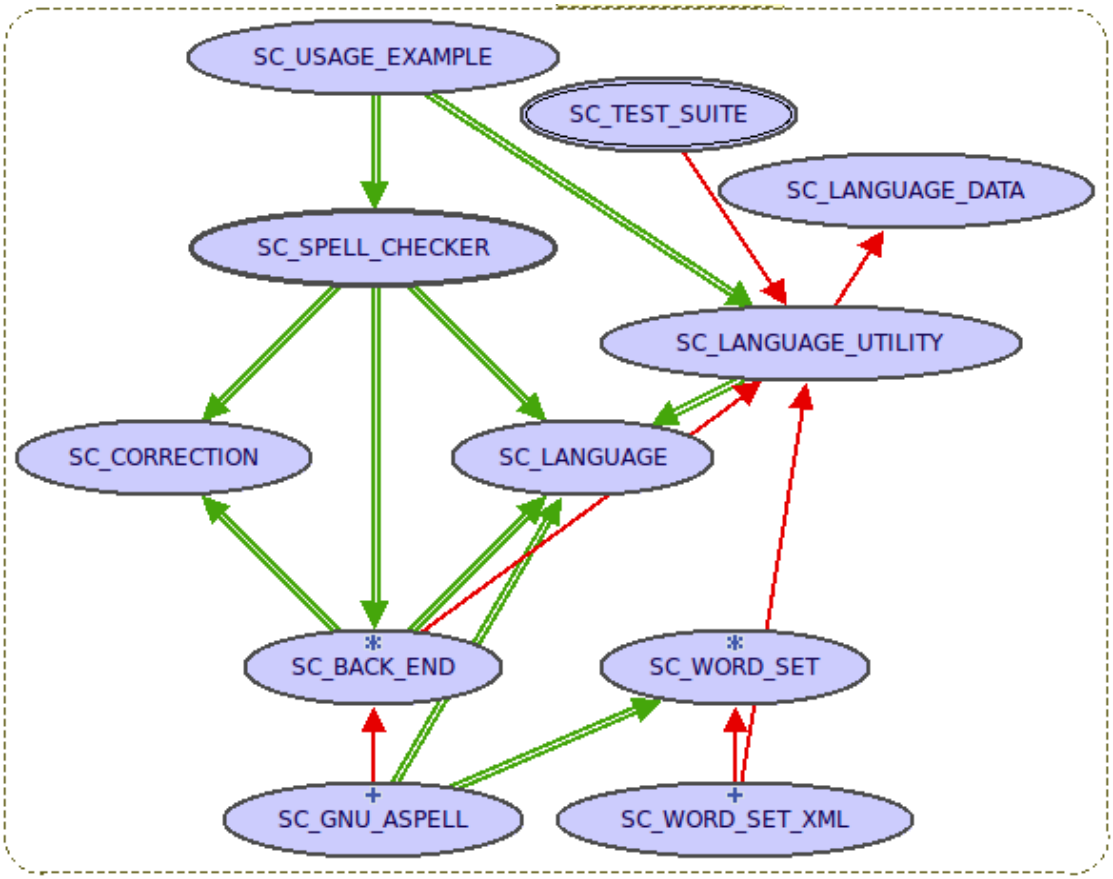

Figure 1: BON class diagram of the spell checker library

3 Library guide

This short user guide explains how to use the spell checker library and develop further back ends.

3.1 Basic usage

We introduce the usage of the spell checker library with an basic example taken from the class SC_USAGE_EXAMPLE, namely the feature use_basics as reproduced in listing 1. First of all, we create a default spell checker object. This instantiates a default back end and uses its default language. Next, we make sure the language for the spelling is British English and check a word. If this succeeds, we would like to know whether the spelling is correct. If the word is spelled incorrectly, then there are possibly suggestions to replace the word with. Note that the word can be misspelt and no suggestions are found. Therefore, one normally wishes to distinguish three cases: correct, incorrect with suggestions and incorrect without suggestions. Please see the class SC_USAGE_EXAMPLE for more examples.

Listing 1: Basic usage of the spell checker taken from the examples in SC_USAGE_EXAMPLE

```
use_basics
        -- Independent example for basic usage of spell checker.
    local
        spell_checker: SC_SPELL_CHECKER
        language: SC_LANGUAGE
        correction: SC_CORRECTION
    do
        create spell_checker
        create language.make_with_region ("en", "GB")
        spell_checker.set_language (language)
        spell_checker.check_word ("inexistent")
        if spell_checker.is_word_checked then
            correction := spell_checker.last_word_correction
            if correction.is_correct then
                print ("Spelling is correct.%N")
            else
                print ("Spelling is incorrect")
                across
                    correction.suggestions as suggestion
                loop
                    if suggestion.is_first then
                        print (". What about: ")
                    else
```

```
                    print (", ")
                end
                print (suggestion.item)
            end
            print ("?%N")
        end
    else
        print ("Spell checker failed: ")
        print (spell_checker.failure_message)
        print ('%N')
    end
end
```

The class SC_SPELL_CHECKER provides the commands

- check_word (word: READABLE_STRING_32)
- check_words (words: LIST [READABLE_STRING_32])
- check_text (text: READABLE_STRING_32)

to check the spelling of a single word, all words in a list or a whole text, respectively. In order to check the kind and success of the last such check, there exist the corresponding predicates is_word_checked, are_words_checked and is_text_checked. The results are offered by the queries

- last_word_correction: SC_CORRECTION
- last_words_corrections: LIST [SC_CORRECTION]
- last_text_corrections: LIST [SC_CORRECTION]

on success of the respective check.

Moreover, the class SC_SPELL_CHECKER provides operations on user and ignore dictionaries. Words in both of these kinds of dictionaries are always treated as correct. The difference is that the words in a user dictionary can be used by the spell checker as suggestions, while those in an ignore dictionary are never suggested as replacements. Both kinds of dictionaries can be extended with a word. Only when the corresponding store feature is called, such changes are made persistent. Finally, the words can be accessed through the type SET [READABLE_STRING_32].

3.2 Extension and reuse

The feature set_back_end of SC_SPELL_CHECKER can change the back end. The class SC_BACK_END is deferred and has many features from SC_SPELL_CHECKER mirrored with default implementations. Developers should note that any effective descendant of the

class SC_BACK_END must redefine one of the features check_word or check_words, since the default behaviour of both is to call the other.

The feature check_text by default breaks the text into words and checks each word. The information about the word boundaries is preserved in the SC_CORRECTION objects, so in particular their function substring can be useful to extract the word. Depending on the language and back end used, this text segmentation may not be able to work as expected. In this case, the functions words_of_text and is_word should be redefined to adapt the text segmentation and word predicate to specific needs.

If a back end to be adapted does not support one of the kinds of dictionaries, the class SC_WORD_SET and its heir SC_WORD_SET_XML can be used. The interface guarantees the procedures extend and prune operating on the word set and load and store for the synchronisation with a persistent storage.

The class SC_BACK_END has a query failure_message in case a subsystem not in control of the program fails. The philosophy is that this message is empty if and only if there is no failure. For this reason, there should always be a nonempty message on failure.

The class SC_LANGUAGE_UTILITY might prove useful in other contexts. It has simple facilities for text processing. In particular, the function words_of_text_with_punctuation finds word limits in a Unicode text. It makes use of the given punctuation to recognise words like "can't" as a whole. To determine whether an arbitrary Unicode character belongs to the major general category of letters, it applies the function is_letter in turn. Furthermore, the function first_newline to find the first newline in a string takes various kinds of newlines into account.

4 Validation

As already mentioned, the spell checker library is tested by the classes SC_TEST_SUITE and SC_USAGE_EXAMPLE. They have already been developed from an early stage of the project to facilitate a test-driven development.

The spell checker tool for EiffelStudio is tested with the help of the class SC_EVOLVABLE. This serves as an example of a piece of Eiffel source code to cover almost all the desired node types of an abstract syntax tree to be checked for spelling. The checked node types are the following. Note that only the names introduced by the code itself should be checked, but not others.

- Comments
- String literals
- Verbatim strings
- Class names
- Names of formal generic types

- New name of renamed features
- Feature names and possible synonyms
- Names of formal parameters, local variables and components of tuples
- Tags of indexing clauses
- Tags of contracts of all kinds
- Identifiers introduced by across loops
- Identifiers introduced by tests for attached variables

5 Deployment

The spell checker library with its tool is now available as a part of EVE, a development environment built on top of EiffelStudio. It is thus distributed under the terms of the GNU General Public License. The library can be found in the folder `library/spelling`, while all the required test data is located at `library/spelling/tests`. The tool integrated into EVE is in the folder `Eiffel/interface/graphical/tools/spelling`.