Checking Spelling in Source Code

Elliott Hughes
enh@acm.org

Abstract

Word processors have long been able to check spelling as a special pass over the whole document, but the feature really came into its own with "check as you type". So useful is this feature (and so well within the capabilities of modern computers) that we now find it in other text-handling applications such as mailers and web browsers. Mac OS offers it on all text components for no extra programming effort. Programs that don't check what you're typing as you type it are becoming increasingly annoying and the surprising number of programs that won't check your spelling at all are even worse.

Despite this, the editors and other tools used by programmers are least likely to offer spelling checking. This article looks at why this is, and describes the addition of spelling checking to a programmer's editor.

1 What's the problem?

If you load source code in a word processor, you'll likely find most tokens adorned with a broken red underline, highlighting supposed spelling errors. A word processor expects word boundaries to be demarcated by whitespace, something far from typical of source code.

Programs often need to name things that are complicated enough to require more than one word (such as the concept "line width"). At the same time, a common prohibition against spaces in identifiers gives rise to a variety of alternative representations for multi-word identifiers.

Different languages have different idioms for this, some of the more common ones being:

- `get_font_list` (C/C++)
- `get-font-list` (Lisp)
- `getFontList` (Java, Smalltalk)

The so-called CamelCase[3] pioneered by Smalltalk is increasingly popular, but C's style is still common and Java actually uses both for different parts of speech: most Java identifiers use CamelCase, but constants are written in all capitals with underscores separating words, `LIKE_THIS`.

(When I was at university and writing Ada, I was told a story about a program containing the identifier `PenIsUp`. After an 'Ada beautifier' converted identifiers to all capitals, this looked like something very different. Changes of case or the use of punctuation are *both* fine, but make sure you use one or the other!)

Despite the popularity of CamelCase with programmers and people dreaming up company and product names, CamelCase hasn't caught on in the real world, and so spelling checkers don't support it.

What about "run together" words?

The Unix program `aspell` has an option to allow "run together" words. This covers cases such as Java's Throwable class, which isn't a word known to many spelling checkers, but can be considered as the words 'throw' and 'able' run together. Similarly, `getFontList` would count as the three words 'get', 'font' and 'list'. This kind of running together is particularly popular in agglutinative natural languages such as Hungarian and Japanese, and perhaps best known in the English-speaking world from Mark Twain's complaints about German[2], so it is supported by spelling checkers, but it doesn't help us.

There are two problems with `aspell`'s running-together, from our point of view. Firstly, `aspell` makes no distinction between `getFontList` and `getfontlist`, but a compiler for a case-sensitive language will.

Secondly, if we want to get sensible suggested corrections for misspellings, allowing words to run together makes the spelling checker's life difficult. If we give it a run-together word, we're likely to get single-word suggestions back. We might think that 'gentlest' and 'cottontails' are unlikely (`echo getGontList | aspell -a`): they're so obviously wrong they're almost funny, but how should the computer know? We told it it was looking at a single word, after all.

If we break identifiers into words before giving them to the spelling checker, we can ensure that case changes are treated as significant, and thus ensure that we get decent suggestions back from the spelling checker. To return to our example, the first suggestion for 'Gont' is 'Font'.

2 Breaking the Camel's Back

The solution to dealing with real-world identifiers is every bit as obvious as it seems: when breaking the text into words, in addition to the usual loop condition that takes non-word characters as word boundaries, we keep track of the case of the previous character and add a few extra conditions:

- A change from lower-case to upper-case means we're at the end of a subword in a CamelCase word.
- A change from upper-case to lower-case in a word longer than 1 character shows we've gone past something like `URL` in `URLConnection`, and should back up and terminate.
- An underscore is no longer a word character; treat it as whitespace.

```
# MAX_VALUE => MAX VALUE
tr!("_", " ")

# getFontList => get Font List
gsub!(/([a-z])([A-Z])/,
      "\\1 \\2")

# URLConnection => URL Connection
gsub!(/([A-Z]+)([A-Z][a-z]+)/,
      "\\1 \\2")
```

Figure 1: Ruby to insert word breaks implied by CamelCasing if you're in a situation where it's easier to pre-process the input.

It's convenient for the implementation to pretend there are three cases, adding 'unknown' to lower-case and upper-case.

Coping with apostrophes

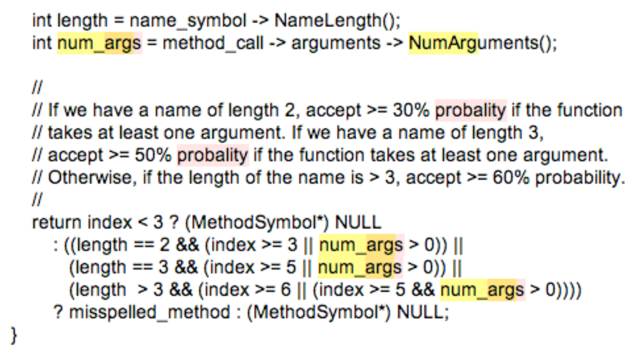
The apostrophe presents a slight problem. In order to handle string literals and comments, "isn't" should be treated as a single word, which suggests apostrophe should be accepted as part of a word. That said, the quoting apostrophes in a comment such as "between 'alpha' to 'beta' inclusive" shouldn't be treated as parts of the words "alpha" and "beta". Ada and VHDL's `name'attribute` syntax adds an additional complication.

All of this is easily handled by considering apostrophe to be a valid character within (but not starting) an identifier and then explicitly recognizing the special cases of trailing apostrophes and apostrophes between words.

3 Displaying Misspellings

The now-traditional way to mark a misspelled word is to draw a red line of some style under the word. In Java, it's easier to make use of Java's ready-made DefaultHighlightPainter class.

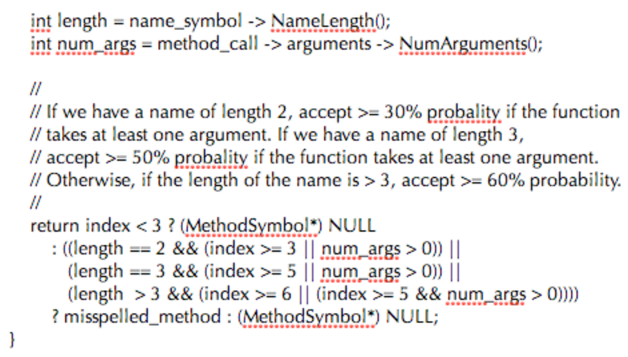
The editor we've worked on and with for the past five years – imaginatively named Edit – also uses



```
int length = name_symbol -> NameLength();
int num_args = method_call -> arguments -> NumArguments();

//
// If we have a name of length 2, accept >= 30% probality if the function
// takes at least one argument. If we have a name of length 3,
// accept >= 50% probality if the function takes at least one argument.
// Otherwise, if the length of the name is > 3, accept >= 60% probability.
//
return index < 3 ? (MethodSymbol*) NULL
    : ((length == 2 && (index >= 3 || num_args > 0)) ||
      (length == 3 && (index >= 5 || num_args > 0)) ||
      (length  > 3 && (index >= 6 || (index >= 5 && num_args > 0))))
    ? misspelled_method : (MethodSymbol*) NULL;
}
```

Figure 2: A snippet of Jikes' source in Edit. Misspellings are in red, matches of the regular expression "`(?i)num_?arg`" are yellow. Note that in addition to the misspellings of 'probability', the abbreviation 'args' is highlighted, but that 'num' is not, because it's short enough to be ignored.



```
int length = name_symbol -> NameLength();
int num_args = method_call -> arguments -> NumArguments();

//
// If we have a name of length 2, accept >= 30% probality if the function
// takes at least one argument. If we have a name of length 3,
// accept >= 50% probality if the function takes at least one argument.
// Otherwise, if the length of the name is > 3, accept >= 60% probability.
//
return index < 3 ? (MethodSymbol*) NULL
    : ((length == 2 && (index >= 3 || num_args > 0)) ||
      (length == 3 && (index >= 5 || num_args > 0)) ||
      (length  > 3 && (index >= 6 || (index >= 5 && num_args > 0))))
    ? misspelled_method : (MethodSymbol*) NULL;
}
```

Figure 3: The same snippet when checked by a word processor. Even without highlighting the regular expression matches, the real misspellings are much harder to see, and get lost in the noise.

this class to highlight all matches of a regular expression search (in the same way that Unix's `less` uses reverse video). Using the same kind of highlighting presents a problem: one kind (a regular expression match, say) might obscure another (a misspelling, say) or even coincide with the selection highlight. Just using a different color isn't good enough if it's possible for one highlight to be drawn underneath another.

A simple and effective solution is to use colors with alpha. Edit uses a solid (alpha 255) color for the selection, a semi-transparent color for highlighting regular expression matches (alpha 128) and a highly transparent color for misspellings (alpha 32).

The graceful handling of overlapping highlights provided by alpha-blending is reason enough not to revert to our original plan of implementing red underlining.

4 Making Corrections

When the user brings up Edit's context menu over a misspelling, options to correct the mistake are appended to the menu. This doesn't require any extra state to be stored: we can use the list of highlights kept by the Highlighter to find the extent of the misspelled word, and we can pass its text to `ispell` for suggestions. These are then converted into instances of an Action subclass that replaces the highlighted range with the suggestion.

One problem with `ispell`'s suggestions is that it doesn't understand that we're writing code, so it has a habit of suggesting we add hyphens or spaces, which is rarely an option. A simple expedient is to replace hyphens in suggestions with underscores, and multi-word suggestions with CamelCase suggestions.

The suggestions aren't as good as those from the Jikes Java compiler which takes into account type and accessibility information. If you're just having trouble spelling gauge, though, they're fine.

5 Performance

The first implementation relied on Apple sample code using JNI to provide a wrapper around Cocoa's NSSpellServer, but the sample code had bad habits from a performance point of view. It could have been improved, but it wasn't portable even to other Unixes, and one of Edit's goals is portability.

When the prototype code for handling CamelCase was working, the Apple code was replaced with a couple of classes. One class deals with the JTextComponent side of things: listening for Document updates, extracting words to send to the checker, and highlighting misspellings. The other class provides methods to ask whether a word is a misspelling and to ask for suggested correction. It works by creating a Process running `ispell` with the `-a` option. Candidate words are written to the pipe, and `ispell`'s responses read back.

The new code provided the portability but didn't provide the necessary performance. Using the newer `aspell` turned out to be even slower, though Edit still falls back to this if `ispell` isn't available.

As-you-type checking was fine, but there was a noticeable slowdown when opening a file with spelling checking enabled. It took too long to open JTextComponent.java (one of the largest source files I ever open). A certain widespread word-processor is happy to wallow in its sloth and check asynchronously when opening a file, but that behavior is distracting and it's possible to do much better.

Communication with `ispell` is expensive, so it's avoided where possible by keeping two HashSets; one of words known from previous communication with `ispell` to be correctly spelled, the other of words known to be misspelled. Using this optimization but with a cold cache, spelling checking adds 385ms to the time taken to open JTextComponent.java (10,257 words). Loading the file again when the HashSets contain all the words takes just 22ms.

(Although I only chose this file because it was the largest in my Edit log, the spelling checker did point out the incorrectly-spelled private method 'shouldSynthensizeKeyEvents'. Oops.)

What is 'As you type'?

The current back-end implementation was developed at a time when Edit checked surrounding words every time a key was pressed, and is more than fast enough for this behavior. We were concerned that it would be distracting to see a word changing back and forth between correct and incorrect as you type it, though in practice this is outweighed by the sense of completion when a word changes from 'incorrect' to 'correct', and immediate feedback when you start to go wrong. (One user stresses the fact that guessing-with-feedback is as good a way to find the correct spelling as any, or at least seems that way because it keeps the brain pleasantly occupied.)

The eagerness to check unfinished words does make it harder for Edit to suggest words to add to a user dictionary, because its list of misspellings includes so many half-finished words.

The less eager alternatives we've considered would also have their annoyances. Checking only after the user stops typing can force a user to stop and wait for the computer to notice. Checking only when the user moves off a word means that the computer makes no use of natural pauses in typing. Edit's solution seems no worse than the others we've used, and its implementation is probably the simplest, even if it does demand better performance from the spelling checker than most other systems seem to manage.

6 Accuracy

Programming vocabulary isn't identical to general vocabulary, so it's only to be expected that we suffer from false positives: words highlighted as misspellings that are acceptable. Examples include:

- Words invented by regularizing natural language (such as 'tokenize' and 'tokenizer' from 'token').
- Words that do exist but are uncommon enough not to be in spelling checker dictionaries (such as 'Highlighter').
- Well-known technical terms (such as 'inode').
- Abbreviations so common that they no longer seem like abbreviations (such as 'args').

There are also a large number of false negatives, words that – while correctly spelled and not highlighted – are *highly* unlikely to be intentional. My favorite example is typing `pubic` instead of `public` as an access modifier.

The Java naming culture, where abbreviations are frowned upon, and where descriptive names are the norm, is ideal. Not only do such conventions make life easier for humans trying to find their way around a large API, they make the computer's task of checking spelling easier. C++ works less well, because the culture there is more ready to accept names like `creat` and `malloc` and `sysctl`.

Even Java has ground-in dirt such as `System.arraycopy` (which at least should be written `arrayCopy`), but such examples are rare. The most problematic Java names are package names, which don't indicate word breaks (as in `java.imageio`), and are one of the few places in the JDK where abbreviations are used (as in `java.lang`). How ironic that the identifiers most frequently abbreviated are at the same time the identifiers least frequently keyed in by humans.

Coping with keywords

As mentioned above, for performance reasons words are checked against HashSets before falling back to asking `ispell`. This provides an opportunity to ensure that the keywords for a source file's language are all accepted as correct: a property can be set on the Document that, if present, provides a HashSet of file-specific correct spellings. This makes it easy to have `struct` correct in C++ but not in Java, and vice versa for `instanceof`, for example. It also means that the spelling checking code doesn't have to know anything about languages or source files.

There could also be a primed HashSet of known-bad words, to cope with false negatives such as the aforementioned 'pubic', but this hasn't yet been implemented.

Coping with case

Although the technique used to cope with keywords extends to identifiers in the standard library, identifiers that contain acronyms in a form other than all-capitals cause trouble. An example is the use of `HttpConnection` in place of `HTTPConnection`. Spelling checkers usually know when a word should have a specific capitalization, and know to prefer 'ASCII' over 'Ascii' and 'British' over 'british', for example.

There doesn't seem to be a case-insensitive switch to `ispell`, so to work around this problem, if one of the suggested corrections is the same as the original word when compared case-insensitively, the original word is accepted as correct. This only works for words known to `ispell`. HashSets don't help because there would have to have an entry for each acceptable form. Three simple solutions spring to mind: entering multiple variants into the HashSet, forcing all words to the same case, or using a case-insensitive hashCode and equals.

Perhaps better would be to start a separate instance of `ispell` for each language, and temporarily add appropriate words to each `ispell`'s accept list. (Multiple instances are needed to avoid mistakenly allowing, for example, `instanceof` in C++ and `struct` in Java.)

7 Is it Useful?

Programmer's editors commonly color text according to what kind of lexical token it represents (so-called "syntax coloring"), which is similar to highlighting correctly-spelled keywords.

Editors that mark misspelled words are far less common. This presumably means one of two things: either it's believed to be difficult, or it isn't believed to be useful. Given that it *is* pretty easy, is there good reason to believe it isn't useful?

Finding errors

Spelling checking can catch an otherwise hard-to-detect class of error: an attempted override that fails because you've misspelled the method name. In most languages this is a quiet failure, because you can't announce your intention to override. The compiler doesn't understand English or spelling and can only assume you're writing a new method. (Some languages force you to say what you think you're doing, but they're sadly rare.)

An error caused by accidentally implementing a `ComponentResized` method instead of `componentResized` in one program went unfound for a year. Sadly, this is a violation of naming conventions, and something simple spelling checking doesn't spot[1].

Most frequently, spelling checking saves you the trouble of asking the compiler to find your typos. It couldn't have stopped `getFontDecent` from making it into the JDK, and it can't warn me that `pubic` isn't an access modifier, but it would stop misspellings of 'allocator' and 'gauge', both of which have been misspelled in codebases I've seen.

Finding questionable code

The C++ dictionary doesn't contain all the functions from the C library. Common C problems such as buffer overflows and incorrect manual memory allocation are often caused by evil code using `malloc`, `sprintf`, and `strlen`. It can be a timesaver when looking for problems if this kind of code stands out like a sore thumb.

(There's never any need to write this kind of code in C++, but it does still get written by unreformed C programmers.)

Improving documentation

Spelling checking also checks string literals and comments "for free", both of which are all too often neglected by busy programmers. (It can't, of course, force programmers to write comments, nor can it force them to write *useful* comments, nor can it ensure that their comments are kept up-to-date. But every little helps, and making comments a better approximation of English is a start.)

[1]A reviewer writes: "funny that one of your examples is typing 'ComponentResized' instead of 'componentResized'; I once did much the same thing, but putting 'componentResixed'. The spelling checker would have spotted it right away, but it took me a while."

Reducing degrees of freedom

One of the difficulties of programming is that there are so many degrees of freedom; so many things to choose between; so many trade-offs to consider. This is as true on the small scale of formatting and naming conventions as it is on the large scale of architecture. And although we've long had patterns and experience to help with the macro problems, there's been nothing nearly as convincing at the small end of the spectrum.

Java is a notable exception in that it comes with an attached set of coding conventions, and a large body of code that follows these conventions. I remember when I first started to use Java, I followed the conventions for no other reason than because they were there, and I wanted my code to look like all the rest. I soon found, though, that there are a couple of quiet advantages to this way of working: you have to think less when you need to name something, and you have to search less when you want to find something. When everyone uses the same vocabulary and even the same syntax when gluing words together, it's easier to just guess the name of the method you're looking for. It doesn't always work – is it `length` or `length()` or `size()` or `getSize()`? – but it works often enough that you come to rely on it.

While spelling checking's contribution is far more modest than any shared literature, it does also seem to be helpful. If nothing else, it's firmed my conviction that Bloch is right when he writes[1]:

> There is little consensus as to whether acronyms should be uppercase or have only their first letter capitalized. While uppercase is more common, a strong argument can be made in favor of capitalizing only the first letter. Even if multiple acronyms occur back-to-back, you can still tell where one word starts and the next words ends. Which class name would you rather see, `HTTPURL` or `HttpUrl`?

The fact that cooperating slightly with the editor allows it to help you acts as an encouragement to adopt one of the conventions it supports (it can cope with URLConnection, but wouldn't be able to cope with HTTPURL if it existed). The more of these useless freedoms that can be taken away by our tools, the more of our brain we have left to consider the important ones.

Might ubiquitous spelling checking encourage hubris? It's possible that the fear of misspelling a difficult word helps programmers stick with familiar vocabulary. It would be unfortunate to see the `--verbose` option become `--magniloquent`, say. Far too many programmers have that kind of sense of humor, seemingly unaware of the detrimental effect it has on the ability of others to use guesswork or `grep` to find things, both important factors in making people feel at home in a codebase and preventing duplication caused by ignorance.

The larger libraries become, the less practical it is to offer a manual in the traditional linear sense because they don't scale; instead we have a tutorial showing the general lay of the land, and JavaDoc-like complete API reference. To make effective use of this kind of documentation, you need to be able to guess what things are called, or at least guess what to search for, and that's hindered by unusual vocabulary, puns, and 'cleverness' in general.

Realistically speaking, there probably isn't anything we can do to prevent thoughtlessness. All we can do is ensure that everything is correctly spelled.

8 Further Work

Can errors like `getFontDecent` be avoided by having special limited dictionaries for code? How would we come up with appropriate dictionaries?

Should code be distinguished from comments and string literals?

Is the best way to deal with errors like `pubic void main` to actually parse the program text as-you-type, and have the lexer/parser errors fed back into the editor more directly than it is currently? (That is, in the same way that spelling checking works, and not in the way editors usually provide hyperlinks to errors somewhere other than directly in the source?)

Could compiler diagnostics be improved using spelling checking? It would be good to warn about misspellings in a class' interface. Would using spelling checker suggestions to guide the choice of suggested repair when faced with an unresolved identifier give results any better than Jikes' already near-perfect guesses?

Where else might this be useful? We've already added the same functionality to the text components in our revision control tools, at the cost of just one import statement and a single extra line of code. The mailers we use as programmers, and the document-writing systems (in this case, TeXShop) would also benefit. If they were written in Java, I'd have done this already.

In fact, is there any reason why *all* general-purpose spelling checkers couldn't or shouldn't support CamelCase?

9 Conclusion

Having source code checked for spelling errors is doable, is neither useless nor actively harmful, and it is quicker than resorting to asking Google when I think I might be misspelling a word. Most useful of all, I'm watched over even when I *don't* realize that I'm misspelling a word!

Acknowledgements

Thanks to Martin Dorey and Ed Porter for their comments on earlier drafts, and for having used and contributed to many versions of Edit over the years.

References

[1] Joshua Bloch. Effective Java Programming Language Guide, Addison-Wesley 2001.

[2] Mark Twain, Appendix: The Awful German Language. "A Tramp Abroad", 1880.

[3] WikiPedia, Entry: "CamelCase". `http://en.wikipedia.org/wiki/CamelCase`