Refactoring with Regular Expressions

Spinellis, Diomidis

DOI
10.1109/MS.2024.3439028

Publication date
2024

Document Version
Final published version

Published in
IEEE Software

Citation (APA)
Spinellis, D. (2024). Refactoring with Regular Expressions. *IEEE Software*, *41*(6), 29-33.
https://doi.org/10.1109/MS.2024.3439028

Editor: Diomidis Spinellis
dds@aueb.gr

Refactoring With Regular Expressions

Diomidis Spinellis

CODE REFACTORING[1] IS a key practice of software development. It entails improving the code's internal quality without altering its functionality. Regularly applied, it reduces accumulated technical debt, contributing to the code's long-term sustainability. Refactoring can also be employed proactively to enable or simplify a subsequent implementation task. Many integrated development environments (IDEs) can automate common refactoring operations, such as renaming an identifier or extracting a method or constant. Yet there are cases where the IDE does not support the required refactoring operation or even a project's programming language. Then, rather than laboriously changing each element by hand, it can be profitable to automate the process by employing the power of regular expressions. This method saves work and time and reduces errors and keyboard fatigue, while also making the task more interesting.

I faced such a case when I attempted to extend the CScout refactoring browser for code that is written in C[2] to make it collect metrics regarding the C preprocessor's use. (The C preprocessor offers features such as file expansion and textual macro replacements. I feel that its overuse accumulates technical debt by making the code difficult to understand, debug, and reason about. The CScout extension would allow me to quantify its use across files and along time.)

The CScout code (written in C++) stems from a quarter-century ago and shows its age: formatting and identifier names are inconsistent; it lacks unit tests; many of its units, such as files, classes, and methods, are overly large; and it does not employ several modern C++ features. Admittedly, it also shows the immaturity of its (and this column's) author at that time; improving current and legacy code quality is a lifelong pursuit for dedicated software developers. Yet, adding the new metrics started smoothly. I located the classes where file and function metrics were stored, changed them into a pair representing values before and after the preprocessor (223dbd0) (the shown Git hashes refer to openly available commits in the task's work branch: https://github.com/dspinellis/cscout/commits/pre-post-metrics/), and started updating each metric operation to tally accordingly the value before or after the C preprocessor (08dd981), while also noting where I could add new metric collection taps (eb7021c) for the newly required counterpart metrics.

Then, unexpectedly, hell broke loose. All I had done was add a single line of code: a newly required header file, *token.h*, in the *metrics.h* file. (A C/C++ header file, typically suffixed with *.h*, defines the interface to some functionality, while a corresponding code file, *.c* or *.cpp*, contains the implementation.) However, the newly included header introduced a circular include dependency: *metrics.h* → (new) *token.h* → *tokid.h* → *fileid.h* → *filemetrics.h* → *metrics.h*.

Circular dependencies are always bad news. At best, they introduce tight undesirable coupling, making code difficult to understand, modularize, and refactor. At worst, they can lead to broken builds and undefined build time and runtime behavior. In my case, because the dependencies were required by diverse templated (generic) C++ functions, necessarily residing in the interdependent header files, the cycle prevented CScout from compiling.

After long head-scratching and diagram sketching, first to isolate the circular dependency, then to understand its cause, and finally to design a

Digital Object Identifier 10.1109/MS.2024.3439028
Date of current version: 10 October 2024

fix, I knew how to address the problem. CScout represents all elements of the program it analyzes through a unique (*file-identifier*, *file-offset*) pair. It abstracts the file identifiers (small integers) as tiny objects of the *Fileid* class. As *Fileids* get embedded into billions of objects, CScout stores additional details about files in a *Filedetails* class. A vector, named *i2d*, indexed by file identifier, efficiently maps file identifiers into file details. To improve the code's readability, *Fileid* had several methods, such as *set_required*, shown in Figure 1, that operated on *Filedetails* through an indirection via the file identifier-to-details mapping vector.

```
void set_required(bool v) {
    i2d[id].set_required(v);
}
```

This entanglement between the ubiquitous *Fileid* class and the newly wider-spread *Filedetails* class was the circular dependency's root cause.

The solution involved isolating the *Filedetails* class from *Fileid* (a00700b; see Figure 2) and rewriting more than one hundred calls to 30 different *Fileid* instance methods to instead call corresponding *Filedetails* static (class) methods with the *Fileid* passed as an argument to them (e306417). For example, the method call on to *set_required* the *Fileid* object *fi*,

```
f.set_required(true);
```

would need to become

```
Filedetails::set_required(f, true);
```

git-subst to the Rescue

When programming, I often encounter cases where I want to replace some text through all files of a project. This can be to clarify an identifier's name, to correct an inconsistently spelled term, to spell out a cryptic file name, to replace a number with a symbolic constant, or, as in what I will describe, to refactor some code. IDEs, code editors, and command line tools offer some such functionality, but they often have trouble identifying the files on which to apply the changes, performing too few or too many of them. Years ago, I decided that this operation should be a plug-in for the Git version control system that would replace a specified string with another in all files managed by Git. I implemented it as a simple shell script named *git-subst* and never looked back; until today, I have used the corresponding command more than 500 times.

The *git-subst* command offers several benefits. Because it is a Unix shell command, it can be easily automated, as I did in this case, to be applied multiple times with different arguments. It is recorded in the shell's command line history, and it can therefore be recalled, edited, and reapplied until the intended result is obtained. Also, its use can be easily and clearly documented in commit messages so that others can readily review or replicate its invocation. Compared with IDE facilities, such as the Visual Studio Code "Search & Replace" command, *git-subst* works independently of the editor or IDE being used. Therefore, it can be used quickly without launching an IDE, and it can be also applied in situations where an IDE with such functionality is not available. Compared with a Unix command line invocation of a *sed* in-place (-i) replacement command, *git-subst* automatically applies the change only to source code files and does so in all underlying directories.

The *git-subst* command is installed by copying it from its Git repository (https://github.com/dspinellis/git-subst/) to a directory in the executable files' path and giving it execute permission. Once there, Git will automatically locate it and allow the execution of commands such as

```
git subst 3.1415927 Math.PI
```

which will replace all instances of the number "3.1415927" with the symbolic constant Math.PI. The

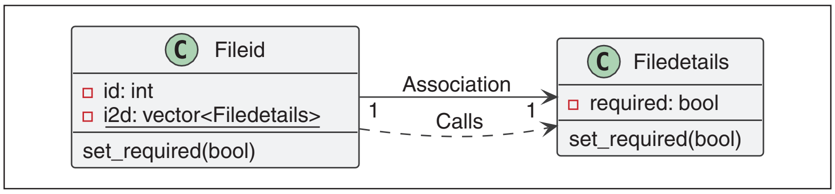

FIGURE 1. Class details before the refactoring.

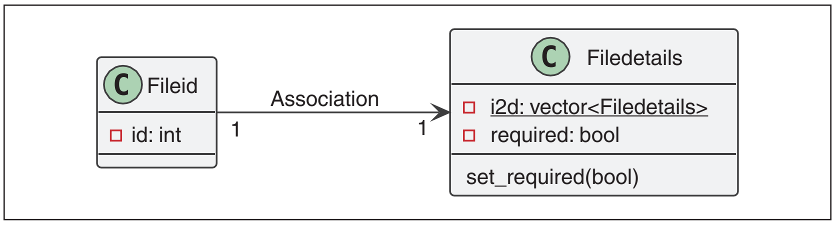

FIGURE 2. Class details after the refactoring.

string to replace is specified as a *regular expression*: a versatile recipe for specifying diverse classes of different strings. Regular expressions are supported by many programming languages, libraries, command line tools, and editors. When programming, I use regular expressions several times per hour. If this does not match your own experience, you may be wasting keystrokes and energy. Consider investing a couple of hours in how they are put together and then applying them in your work.

A few *git-subst* command line options provide finer control of its operation. The `-c` option allows the specification of a positive context for the lines where the replacement will take place, again as a regular expression. For example, `git subst -c ^//colour color` will Americanize the spelling of color only in comment lines [those starting (^) with //]. Other options can specify a negative context (`-C`, lines where the replacement will not be made), the files where the change will take place (e.g., rather than the default, which will apply the change to all files that are under version control, all JavaScript files can be specified with "*.js"), or to perform a trial run without actually making the replacement (`-n`).

The element to be replaced is specified as a regular expression, which allows, e.g., the specification that only whole words rather than parts be replaced: `git subst '\<statuscode\>' statusCode`. (The `\<` sequence matches a beginning of word boundary.) A more advanced feature is the ability of git subst to capture (remember) parts of a regular expression by placing them in brackets and then "replay" that part by writing its original number preceded by a backslash in the replacement string. For example,

```
git subst '\.custom\(([^)]*)\)' '.\1'
```

will change `.custom(name)` into `.name`, `.custom(phone)` into `.phone`, and so on. The preceding regular expression reads as follows: match a literal dot `\.`, followed by `custom`, followed by a literal bracket `\(`, followed by anything but a bracket `[^)]` any number of times*, capture that `()`, followed by a literal closed bracket `\)`. Then the replacement string specifies to write a . followed by the captured part `\1`. Capturing parts of a regular expression and reusing those in the replacement is a powerful method for performing sophisticated ad hoc refactoring changes.

Capturing parts of a regular expression and reusing those in the replacement is a powerful method for performing sophisticated ad hoc refactoring changes.

At 130 lines of code, including license, comments, and usage information, the implementation of *git-subst* is probably one of the most leveraged pieces of code I have written. It obtains the files containing the pattern that needs replacement and processes only these, by using the blindingly fast *git-grep* command. It then uses the Unix stream editor *sed* to perform the replacement. It also relies on *git-stash* to implement the replacement trial run option. Most of the remaining code deals with option processing and autodetecting and adjusting internally used command interfaces according to the flavor of Unix that *git-subst* runs on: Linux, macOS, Cygwin, or a BSD variant.

Getting It All Together

Armed with *git-subst*, my plan for refactoring the code involved obtaining a clean list of methods that needed adjustment and then dynamically generating *git-subst* invocations to fix each one of them. The method definitions appeared in the *fileid.h* header file in single lines, such as the following:

```
void set_required(bool v) {[…]}
```

I converted these into a list of method names for which I needed to adjust the corresponding calls, with a series of editor commands, involving simple regular expressions. In interactive settings, often, rather than writing a single complex regular expression, it is easier to split the task into smaller simple steps. It my case, I instructed the *vim* editor to perform the following changes:

- `g/^\/\//d`: Globally (`g`) delete (`d`) all comment (`//`) lines.
- `%s/ {.*`: Throughout the file (`%`), remove a brace followed by any character (.) repeated any number of times (*).

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology, Athens University of Economics and Business, 104 34 Athens, Greece, and a professor of software analytics in the Department of Software Technology, Delft University of Technology, 2600 AA Delft, The Netherlands. He is a Senior Member of IEEE. Contact him at dds@aueb.gr.

- `%s/ const//`: Remove a space followed by `const`.
- `%s/const //`: Remove `const` followed by a space.
- `%s/^[^ ]* //`: Remove the method's return type by removing from the beginning of the line (`^`) anything but a space (`[^ ]`) repeated any number of times (`*`), followed by a space.
- `%s/& //`: Remove the reference sign (`&`) followed by a space.
- `%s/^&//`: Remove the reference sign appearing at the beginning of the line (`^`).
- `g/^$/d`: Globally (`g`) delete (`d`) all empty lines, i.e., lines whose beginning (`^`) is immediately followed by their end (`$`).

This left me with a list of 29 function names followed by their arguments, such as the following:

```
garbage_collected()
set_required(bool v)
required()
set_compilation_unit(bool v)
compilation_unit()
```

I then separated methods that took no argument from those with arguments because the two would require slightly different handling: the former would need to have the *Fileid* passed as a single new argument, while the latter would need to have the *Fileid* passed as the first argument followed by a comma.

I separated the methods into the two categories by piping the list through the Unix *sort* command, specifying the open bracket as the field separator and the second field (i.e., the argument, if any) as the key [`sort -t\( -k2`].

The final step involved massaging the list of method names to create for each one a *git-subst* invocation that would change the existing *Fileid* instance method calls into *Filedetails* static method calls with the *Fileid* object on which the method was originally called now passed as a parameter. For example, the *git-subst* command for the `set_required` method calls was

```
git subst '([A-Za-z][A-Za-z0-9_]*)\.(set_required)\('
'Filedetails::\2(\1, '
```

This reads as follows: find (while capturing, as denoted by the brackets) a variable name, which starts with an alphabetic character (`[A-Za-z]`), followed by letters, digits, and underlines (`[A-Za-z0-9_]`), any number of times (`*`), followed by a dot (`\.`; the unescaped dot character matches any character), followed by `set_required` (again captured), followed by an open bracket. This matched regular expression is replaced with a static method call to the method of the same name (`\2`; in this case `set_required`), with the object on which the method was called (the first captured identifier; `\1`) as an argument after the open bracket, followed by a comma. For methods lacking arguments, I used a similar *git-subst* command but without a trailing comma.

The risk of automatically performing automatically generated global substitutions with approximately matching regular expressions can be mitigated by using Git as a safety harness. I followed each *git-subst* invocation with *git-commit* (and a suitable one-line subject; see, e.g., b9e24e9) and then *git-grep* to see all instances of the identifier and the changes performed on it. If a specific replacement went awry, it was thus easy to undo the change and adjust it as needed.

Did I enter these almost one hundred commands by hand? Of course not. I used another (surprise!) regular expression replacement in my editor to convert the method name into the three commands. For the methods taking arguments this was

```
%s/.*/git subst '([A-Za-z][A-Za-z0-9_]*)\\.(&)\\('
'Filedetails::\\2(\\1, '^Mgit grep &^Mgit commit -am
'WIP &'
```

This reads as follows: globally (`%`) substitute (`s`), any character (`.`) repeated any number of times (`*`) (i.e., the method name) with the *git-subst* invocation we saw using the replaced method name (`&`) where needed, followed by a new line (`^M`), followed by *git-grep* and the identifier name (`&`), followed by another new line and then a *git-commit* command with an appropriate commit message.

Retrospective

So what happened here? First, I used several simple regular expression replacements to convert the original *Fileid* method declarations into a list of method names. I then employed a somewhat hairy regular expression replacement command to convert these automatically into several *git-subst* invocations for refactoring the method calls. The automation's enabler was the implementation of *git-subst* as a Unix shell command. The *git-subst* invocations also relied on a regular expression to convert the *Fileid* object instance method call into a *Filedetails* class static method call with the original object passed as an argument. Furthermore, I used Git commits after each replacement to provide me with a clean slate on which to see (and undo, if needed) the effects of the subsequent one. After the code compiled, I invoked the *git-rebase* command to squash the 30 temporary commits (2078973…e7cbff2) into a single one (e306417), which would make the project's history easier to read.

Regular expressions are not a panacea for refactoring code. They are difficult to apply on statements that span multiple lines, and there are classes of patterns (such as balancing brackets) that regular expressions are not powerful enough to match. But overall, leveraging regular expressions for refactoring is a vital proficiency that should be part of every developer's skill set.

References

1. M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 2000.
2. D. Spinellis, "CScout: A refactoring browser for C," *Sci. Comput. Program.*, vol. 75, no. 4, pp. 216–231, Apr. 2010, doi:10.1016/j.scico.2009.09.003.

Digital Object Identifier 10.1109/MS.2024.3459129