Editor: **Diomidis Spinellis**
Athens University of Economics and Business, dds@aueb.gr

Refactoring on the Cheap

Diomidis Spinellis

THE REFACTORINGS THAT a good integrated development environment (IDE) can perform are impressive. Yet, there are many reasons to master some cheap-and-cheerful alternative approaches. First, there will always be refactorings that your IDE won't support. Also, although your IDE might offer excellent refactoring support for some programming languages, it could fall short on others. Modern projects increasingly mix and match implementation languages, and switching to a specialized IDE for each language is burdensome and inefficient. Finally, IDE-provided refactorings resemble an intellectual straightjacket. If you only know how to use the ready-made refactorings, you'll miss out on opportunities for other code improvements.

In this column, I describe how you can harness the sophistication of your editor and the power of command-line tools to perform many simple refactorings on your own. As a bonus, you'll see that you can write and run most of them in less time than is required to fire up your favorite IDE.

Post your comments online by visiting the column's blog:
www.spinellis.gr/tools

Within Files

The basic tool for performing a refactoring within a file is the editor's substitution command used in conjunction with regular expressions (see "Dear Editor," *IEEE Software*, vol. 22, no. 2, 2005, pp. 14–15). In this section's examples, I'm using the common Unix substitution syntax `s/old/new/` and the *vi* editor's regular expression syntax; other editors offer similar functionality. Splitting a simple method call into two can be as easy as typing

```
s/getX()/getX().getY()/g
```

However, if the original method has arguments that must now be passed by the second method, you need to get creative and include the bracket in the substitution pattern in order to move the arguments to `getY`:

```
s/getX(/getX().getY(/g
```

Regular expressions aren't powerful enough to parse a typical programming language, so you'll often have to resort to tricks like this one to handle brackets and braces. Another useful regular expression substitution technique involves capturing part of the old pattern and reusing it in the new string. For instance, if you want to change calls to the function `raiseRating`, where its first argument is a variable representing an object, into method calls, you might give a command like

```
s/raiseRating(\([^,]*\),/\1.raiseRating(/g
```

Here, the text within the first `\(\)` pair will get stored in a regular expression variable, which you can then employ in the substitution pattern using `\1`. The `[^,]*` idiom inside the brackets matches everything up to the first comma. Again, this isn't bullet-proof—some legitimate expressions might contain a comma—but it works 95 percent of the time (or does 95 percent of the job).

Capitalizing on the regularity of the text you process can save a lot of pain. The following will transform named HTML headers into list-item hyperlinks:

```
s/<h2><a name="\([^>]*\)>\(.*\)<\/a><\/
h2>/<li> <a href="\#\1>\2<\/a><\/li>/
```

I use it regularly to keep the questions and answers in my FAQ documents in sync. It works only because I write the headings on a single line, but the nearest robust alternative (XSLT) would require prohibitively more work.

continued on p. 94

0740-7459/12/$31.00 © 2012 IEEE

continued from p. 96

Across Files

Quite often, you'll want to apply substitution commands not to a single file but to all files in a directory or throughout your project. The canonical way to do this under Unix (and also Mac OS X and Microsoft Windows with an installation of Cygwin) is to use the stream editor *sed*. This can take as an argument some editing commands and apply them to the files you specify. Modern versions of *sed* can perform a substitution in place. Thus, with a command like

```
sed –i –e 's/Employee/Person/g' proc*.scala
```

you can change `Employee` into `Person` on all Scala files in the current directory whose name starts with `proc`. You can precede the substitution command `s` by one or two regular expressions to specify the lines or a range of lines on which the command will apply. If you wanted to change `getWidget` into `getWidgetReference` in all C++ lines containing `const_iterator`, you'd run

```
sed –i -e '/const_iterator/s/getWidget/
getWidgetReference/g' *.cpp
```

Moreover, if you wanted to change `bProxy` to `buildProxy` only within *javadoc* block comments in all Java files in the current directory, you'd run

```
sed –i -e '/\/\*\*$/,/\*\/$/s/bProxy/
buildProxy/g' *.java
```

The Perl and Ruby scripting languages also offer in-place substitution functionality through command-line invocation options, and their expression evaluators allow you to perform more sophisticated processing. The following command changes the first argument of `setFlags` from decimal to hexadecimal in all the C and C++ files in the current directory:

```
perl –pi.bak –e 's/setFlags\((\d+)/
sprintf("setFlags(0x%x", $1)/ge' *.c *.cpp
```

Large projects typically reside in multiple directories. The Unix *find* and *xargs* commands are the building blocks for applying commands to large hierarchies. *Find* will print a list of files that match the criteria you specify. You then pass those through a pipeline to *xargs*, which will invoke the command you specify in batches of as many files as the operating system allows. The typical invocation for a global substitution through all the project's files is something like

```
find project_directory -type f –print0 |
xargs -0 sed –i -e 's/old/new/g'
```

(The `–print0` and `-0` options allow you to process file names with embedded spaces.) You can restrict the files onto which you apply the substitution by passing a `-name` argument to *find*; here is how you would change `intr_handle_t` into `pci_intr_handle_t` only in files with names starting with `pci_`:

```
find /usr/src/sys -type f –name pci_*.c |
xargs sed -i –e 's/intr_handle_t/pci_intr_
handle_t/g'
```

On File Paths

Some refactorings involve changing file names or moving files around. You can easily accomplish this by using *find* to list the corresponding files and *sed* to craft the text of a command that will accomplish the action you want. You then pipe the generated commands into the shell (*sh*), which will execute them as if you typed them interactively. Depending on your setup, you'll want the commands you create to either manipulate the files directly or to call up your version control system to perform the corresponding action. The following command will add an `fs` prefix to all file names residing in directories ending in `fs`:

```
find . -type f |
sed -n –e 's/\(.*\)fs\/\(.*\)/mv "\1fs\/\2"
"\1fs\/fs_\2"/p' |
sh
```

It acts by generating an *mv* (rename) command for all matching path names. (As invoked, *sed* will read lines from its standard input and only print the lines for which the substitution succeeds.) If you wanted to issue commands to the Subversion version control system, you'd specify `svn rename` instead of *mv*. With similar commands, you can move files around the project's directory hierarchy or remove files that are no longer needed.

Profiting

There are many habits that can increase your effectiveness in the tasks I've outlined here. First, as befits someone who works on the cheap, you must be stingy and lazy. With the ability to easily undo changes either within your editor or through your version control system, don't sweat coming up with the perfect regular expression that will succeed in every imaginable case. Try out simple commands that could conceivably work and see if they're good enough. If not, the compiler or a visual inspection should catch the errors, and you can try a slightly more sophisticated version of the command. In particular, the regular expression repetition operators * and + aren't matching as greedily as you might fear. If you specify a well-chosen string after them, the regular expression engine will backtrack and have the repetition operator match only the part needed for the whole expression to succeed. Many commands in this column utilize this property.

Furthermore, be ready to tolerate a few missed or extraneous changes. If you can easily locate them, correcting them by hand is often faster than crafting the exact command that will work on all cases. Remember, your goal is to be productive; you're writing throw-

away code that no one else will ever see again, so you might as well enjoy employing a few shortcuts.

Be opportunistic in the commands you craft, taking advantage of style, coding, and API conventions to achieve your results. Thus, if you can narrow down on the correct method to change because it happens to always appear as an argument to another method, don't be shy about specifying that in your command.

Regular expressions are nifty, but they aren't sufficiently powerful for all the tasks you'll face. Also, the commands can quickly become overwhelmingly complex. You can solve both problems by running the substitutions step by step. Dodge a sequence that confuses the regular expression matcher by converting it into something innocuous—say, @v@—running the substitution, and then converting it back to its original form. Break complex substitutions into small steps, verifying the results in each step.

Finally, you can simplify your life if you write code in a way that can aid the refactorings I've described here. Be consistent in naming your identifiers, formatting your code, and the way you split it across lines and files. There are many good reasons for writing good code: being able to refactor it on the cheap is just the icing on the cake.

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720-1314; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscription rates: IEEE Computer Society members get the lowest rate of US$54 per year, which includes printed issues plus online access to all issues published since 1984. Go to www.computer.org/subscribe to order and for more information on other subscription prices. Back issues: $20 for members, $163 for nonmembers (plus shipping and handling).

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.

IEEE Software

STAFF

Lead Editor
Dale C. Strok
dstrok@computer.org

Content Editor
Brian Brannon

Manager, Editorial Services
Jenny Stout

Editors
Camber Agrelius, Dennis Taylor, and Linda World

Publications Coordinator
software@computer.org

Production Editor/Webmaster
Jennie Zhu

Contributors
Alex Torres

Cover Artist
Eero Johannes

Director, Products & Services
Evan Butterfield

Senior Manager, Editorial Services
Lars Jentsch

Senior Business Development Manager
Sandra Brown

Membership Development Manager
Cecelia Huffman

Senior Advertising Coordinator
Marian Anderson
manderson@computer.org

Editorial: All submissions are subject to editing for clarity, style, and space. Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Software* does not necessarily constitute endorsement by IEEE or the IEEE Computer Society.

To Submit: Access the IEEE Computer Society's Web-based system, ScholarOne, at http://mc.manuscriptcentral.com/sw-cs. Be sure to select the right manuscript type when submitting. Articles must be original and not exceed 4,700 words including figures and tables, which count for 200 words each.

IEEE prohibits discrimination, harassment and bullying: For more information, visit www.ieee.org/web/aboutus/whatis/policies/p9-26.html.