Editor: Gerard J. Holzmann
NASA/JPL
gholzmann@acm.org

Hi Maintenance

Gerard J. Holzmann

IT HAS OFTEN been pointed out that measuring programmer productivity by the number of lines of code produced per unit of time is dubious. Measuring code quality by the comment-to-code ratio is similarly unhelpful. So, why are these metrics so bad?

Clearly, it's easy for programmers to increase their performance on these metrics by producing unnecessarily bloated code littered with uninformative comments. This effect is known as Goodhart's law, which says that "When a measure becomes a target, it ceases to be a good measure." Debugging or maintaining bloated code can be a nightmare. The unfortunate souls asked to fix it years later will have a hard time reconstructing what pattern of thought led to its creation.

Curiously, bloated and badly written code tends to live longer than well-written code. If you can't understand how or why some code works, you're much less likely to change it. After all, the golden rule of code maintenance is, if it ain't broke, don't fix it.

Code Bloat

Another reason why simplistic code metrics are so unhelpful is that really good programmers tend to write very concise code that doesn't need many explanatory comments. So, it's mostly the bad code that will score well on these metrics.

As is often the case, it's easier to spot the absence of code quality than its presence. As part of my job, I have to review a lot of code. In doing so, I try to leverage the use of automatic code analysis tools as much as possible. But even the best analyzers are of little help when you want to find code that's likely to incur the highest maintenance costs.

High-maintenance code not only is verbose but also tends to rely on unstated, poorly stated, or incompletely stated assumptions. If you want to understand that type of code, you need long chains of reasoning to figure out how and why it works, and under which conditions it could start failing when other parts of the system are updated. The reliance on hidden assumptions is probably the most telling feature of high-maintenance code. So let's look at this in a little more detail.

Hidden Assumptions

As a very simple example, consider the following declaration of an array I came across when reviewing a critical piece of embedded C code (though with the names changed):

```
#define MAX_BUF 28
char buf[MAX_BUF*2 + 1];
```

0740-7459/16/$33.00 © 2016 IEEE

A comment at the declaration explained helpfully that the array was fixed to the given size "because of display limitations." Presumably, the text stored in this buffer was going to be displayed at some point or retrieved from a display entry box. The comment didn't explain if the limitation would prevent more than the given number of characters from ever being stored into that array. It also didn't explain if the display couldn't render more than the given number of characters anyway when the text was going to be written to that display. To determine these things first meant hunting down all uses of the array and all the possible sources that could produce the input. Next, it meant checking whether safeguards were in place to prevent that any changes made elsewhere later in the code's evolution would be consistent with the assumptions implicit in this part of the code.

The macro MAX_BUF introduces a number that seems to depend critically on some other quantity related to the display width that might be defined in some other module. You can avoid the dependency, and thus reduce the risk of mistakes, by using that original limit, and not a derived value, directly in the array declaration.

Later in the function in which this declaration was placed, the library function strcpy was used to fill the array, using a character pointer passed into the function as an argument. I'll call that argument param here:

```
if (strlen(param) > 0) strcpy(buf, param);
```

It would be fair to complain about the poor formatting and the lack of curly braces around the body of the if statement, which could have helped make the code a little easier to read, but we have bigger fish to fry here. The developer tried to ensure that a zero-length string wouldn't be copied. That's very considerate, of course, but what if the param pointer is null or points to a longer string that the target buffer can accommodate? To check that this can't happen again sends us hunting through the surrounding code. We can avoid all this by adding an assertion explicitly stating that these conditions can never happen:

```
assert( param && strlen(param) < sizeof(buf) );
```

I use sizeof here instead of comparing against the size from the array declaration, to protect this piece of new code from future changes in the declaration of the buf array. Who knows, that could happen if the system this code is part of ends up being so profitable that the company can afford to switch to larger displays. For every design parameter like this, we want to ensure that the code contains a "single point of truth." That is, there's only one point in the code at which you can make a change without having to chase down all its hidden dependencies. Of course, you also should never use unsafe functions such as strcpy but switch to the safer strncpy, or strlcpy if it's available.

But we're not done. A few lines later in the function, the contents of buf were updated, independently of the length of param, with a call to the library routine sprintf, again ignoring its more well-behaved sibling snprintf. The call looked like this:

```
sprintf( buf, "%s%c", buf, ch );
```

Your alarm bells should now be ringing loud and clear. First, the developer made an unjustified assumption about how sprintf is implemented, passing it the same array as both a source and a destination of the operation. This is a roll of the dice because the C standard explicitly states that the result of such a call is formally undefined. Another concern is that, because the code didn't check whether the array was updated or left unchanged in the earlier conditional call to strcpy, even a correct execution of sprintf to append a single character to the end of buf now risks overflowing the array when it repeats enough times. Cybercriminals know how to exploit this type of vulnerability. But even if this code wasn't a security vulnerability, it's just plain bad, unnecessarily high-maintenance, code.

Token Patterns

If you have to review this code and you see a careless statement such as the sprintf call I just mentioned, you'll immediately want to check for other update operations in which the source and destination might overlap. Here, it's useful to have some tools at your disposal. The Unix tool grep can easily collect all calls to routines such as sprintf, strcpy, and memcpy. But that will likely give you much more output than you need, requiring far too much additional work to separate the wheat from the chaff.

In cases such as these, a simple tokenizer, such as the ctok tool I discussed in an earlier column,[1] with a small back end, can prove invaluable. For instance, here's how we can do a more appropriate pattern search with the Cobra tool, which is based on the same principles:[2]

```
$ cobra –re 'sprintf\( x:@ident , .* :x .*\)' *.c
```

I used the Cobra tool here to match a *token expression* in the input. A token expression is a regular expression that's defined not over

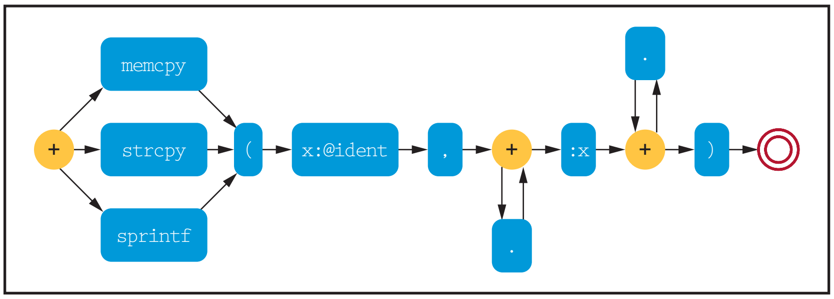

FIGURE 1. A nondeterministic finite-state automaton for the Cobra token expression [memcpy strcpy sprintf] \(x:@ident , .* :x .* \). This automaton performs matching searches for strcpy, memcpy, and sprintf.

strings of characters, like most regular expressions, but over a sequence of lexical tokens. By default, the tool will try to match the literal text of a token, but you can also ask it to match a token type by preceding the text with the @ symbol.

In the previous expression, the identifier name sprintf must be matched exactly in the source code. It is to be followed by an opening parenthesis, which has an escape character in front of it to distinguish it from the corresponding regular expression metacharacter for grouping. Next, the token expression is asked to match any identifier name and store its text in a variable I named x (the name is, of course, arbitrary). This variable-binding operation lets us refer back to that same bound variable later in the expression. The next lexical token to be matched is a comma, which is followed by a sequence of tokens we don't care about unless it includes a second instance of the bound variable anywhere before the matching closing parenthesis. Cobra ensures that parentheses, braces, and brackets are always matched correctly in token expressions. So, we're guaranteed to be checking precisely (and only) the full parameter list of calls to sprintf with this expression.

The token expression isn't sensitive to white space, so it doesn't matter whether the calls to sprintf that we're trying to find span multiple lines of text in the source code.

In this case, we're looking just for uses of sprintf that violate the rules; we can, of course, do matching searches for calls to strcpy, memcpy, and so on. We can also specify all these candidate function names in a single token range in brackets, and use that in the expression. Figure 1 shows the nondeterministic finite-state automaton that's generated from an expression that performs that search.

Writing Low-Maintenance Code

To write concise, readable, and low-maintenance code requires practice, but it helps to look at examples. This is similar to learning to write good prose. For good reason, Steven Pinker titled a chapter "Reverse-Engineering Good Prose as the Key to Developing a Writerly Ear" in *The Sense of Style*, his recent book on writing.[3]

When I implemented the code for processing token expressions in the Cobra tool, I first looked at existing algorithms for converting regular expressions to automata. This class of algorithms is so fundamental that trying to invent a new version from scratch would be foolish. I also expected that the most recent versions would be the best. Surprisingly, that wasn't the case.

A few years ago, Russ Cox wrote an excellent blog entry on existing implementations of regular expression conversion algorithms.[4] He showed a significant difference in performance between recent implementations in Java, Perl, PHP, Python, and Ruby and the by-now-ancient Unix code, still available in tools such as grep and awk. The older code turns out to be much faster. Cox explained how the difference in performance can grow to orders of magnitude for longer expressions. The older code is based on an algorithm that Ken Thompson invented when he implemented regular expressions for the line editor ed. As you probably know, the command name grep is an abbreviation of the ed command g/re/p for globally finding and printing all matches for a given regular expression re.

Dreaming in Code

The paper describing Thompson's algorithm appeared in 1968,[5] shortly after Ken joined Bell Labs. I decided to use that algorithm as the foundation for the Cobra implementation, if only just to learn from how it was designed. The 1968 paper turns out to be an absolute gem. It manages to describe the algorithm in crystal-clear prose in just four pages, including five figures illustrating the main steps.

Thompson's algorithm simulates the execution of a nondeterministic finite-state automaton generated from a postfix version of the regular expression. The conversion is decomposed into a small number of steps that can each be implemented in a straightforward way that requires almost no additional explanation.

Perhaps this is the way we can understand how good code is born. It starts not with the code itself but with developing a really good understanding of the problem to be solved. I suspect that it also depends on the ability to visualize a problem and its possible solutions, before you start writing code. Who hasn't had the experience of suddenly "seeing" the solution to a difficult coding problem as you're about to fall asleep at night? Is that a skill that could be developed and taught? I think I'll have to sleep on that.

References

1. G.J. Holzmann, "Tiny Tools," *IEEE Software*, vol. 33, no. 1, 2016, pp. 24–28.
2. G.J. Holzmann, "Cobra: A Light-Weight Tool for Static and Dynamic Program Analysis," *Innovations in Systems and Software Eng.*, 1 June 2016, pp. 1–15; http://link.springer.com/article/10.1007/s11334-016-0282-x.
3. S. Pinker, *The Sense of Style*, Viking, 2014.
4. R. Cox, "Regular Expression Matching Can Be Simple and Fast," Jan. 2007; https://swtch.com/~rsc/regexp/regexp1.html.
5. K. Thompson, "Regular Expression Search Algorithm," *Comm. ACM*, vol. 11, no. 6, 1968, pp. 419–422.

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.