OPINION

On Teaching Style and Maintainability

GEOFF KUENNING

Geoff Kuenning spent 15 years working as a programmer before changing directions and joining academia. Today he teaches computer science at Harvey Mudd College in Claremont, California, where he has developed a reputation for insisting that students must write readable software, not just working code. When not teaching or improving his own programs, he can often be found trying—and usually failing—to conquer nearby Mount Baldy on a bicycle. geoff@cs.hmc.edu.

Computer science has existed as a separate discipline for more than 50 years, and in that time we have learned a lot about what is important to the field and how to teach it to new entrants. We have long agreed that every self-respecting computer scientist should have a solid grounding in fundamental areas such as algorithms, discrete mathematics, programming languages, data structures, operating systems, software engineering, etc. But in this article, I will argue that there is a major missing component: style and readability. I'll try to convince you that style matters, and I will provide suggestions for how we might encourage better style from both new and experienced software developers.

The list of what we teach incoming students is long, and there are many critical concepts that they need to absorb if they are to be effective in our field. Real programmers use data structures every week, and if they don't have a strong grounding in algorithms, they'll make a major blunder every month. But the essence of software engineering is in the code, and too often we find ourselves wading through the software equivalent of this famous gem:

"In the Nuts (unground), (other than ground nuts) Order, the expression nuts shall have reference to such nuts, other than ground nuts, as would but for this amending Order not qualify as nuts (unground) (other than ground nuts) by reason of their being nuts (unground)."

(If you know what that sentence means, please write me. I've been trying to figure it out for years.)

The issue of comprehensibility is a huge hole in our current education program. Although the 2013 draft ACM curriculum mentions "documentation and style" as a required component of any CS education, the phrase is buried on page 147 as almost an afterthought, given no more attention than power sets and HTTP. (Is HTTP really so fundamental that it even deserves mention?) I claim that this neglect of style is akin to teaching English by concentrating on the common plot devices used in Hollywood thrillers—useful to those working in that specific area, but not to students who need to learn the fundamentals before attempting advanced work.

Think about it for a minute. How much of your programming time is spent writing new code, from scratch? Be honest. Is it ten percent? Five? Less? And how much time is spent working on existing code—trying to understand it, debugging it, adding shiny new features? (Most of us love adding features, because that's one of the rare times we get to write substantial new stuff.)

The reality is that we read code every day: sometimes our own, sometimes written by somebody else, and frequently a blend of the two. Reading code dominates our lives, and it only makes sense that we should try to help ourselves out by making our code easy to read. Even so, too many of us forget that fact and fall into the lazy trap of writing something that we understand at the moment but that won't make sense when we return to it in a year or two.

For example, I found the following snippet (slightly shortened for this article) in a program I use on a daily basis:

```
if (fw < 1)
    fw = 1;
if (fh < 1)
    fh = 1;
if (x + ww - fw > sw)
    x -= ww - fw;
else
    x -= fw;
if (x < 0)
    x = 0;
if (y + wh - fh > sh)
    y -= wh - fh;
else
    y -= fh;
if (y < 0)
    y = 0;
```

Wow. To be fair, this is windowing code, so we can assume the meanings of the suffixes "w" and "h". And the programmers at least had the sense to indent properly (and in the original they used curly braces consistently). But was it really necessary to limit the variable names to single characters, so that the reader must guess their purpose? Why not use max for all the limit-setting? Why are x and y limited to 0, but fw and fh to 1? And perhaps it would be helpful to add a comment explaining why, if x + ww - fw exceeds sw, we *subtract* that quantity (effectively adding fw), but otherwise we ignore ww and subtract fw! There's nothing nearby that gives a hint as to what's going on here.

The Problem

The programmers in the above case were far from incompetent. And they were clearly *trying* to write maintainable code; there are signs of care throughout the program. But in the end they produced something almost incomprehensible. What went wrong?

I believe that the fundamental difficulty is that they weren't taught how to understand what a programmer unfamiliar with the code needs. *They* knew what the variables were for, so single-letter reminders were sufficient. *They* knew why they were adjusting x and y in such an odd fashion, and it never occurred to them that an explanation might be useful. So somebody else who is trying to fix a bug in this program is left to spend hours tracing calls and analyzing the logic, or to step through with a debugger, or (all too often) to guess "Maybe if I change the -= to a +=, things will start working, and it's quicker to recompile and test than to figure out what's going on." But of course that hasty approach often introduces subtle bugs elsewhere.

And why don't programmers understand the needs of readers? There can be many causes, including inexperience, poor skills at explaining things, and even arrogance ("If you don't understand my code, you must just be stupid"). Some of these causes are difficult to address (although the world would probably be a better place if we could ship all the arrogant programmers to a desert island to argue amongst themselves about who is the smartest). But we can begin by doing a better job of teaching style.

Unfortunately, there's a chicken-and-egg problem involved: Relatively few academics have the background necessary to understand how to write maintainable code. The typical career path for a university professor is to go directly from an undergraduate degree to graduate school, and from there straight into a tenure-track position. Undergraduate students usually work only on their own code, and normally only on small programs. Graduates may work a little bit on someone else's code, but eventually they have to develop their own as part of a dissertation, and although that code may be massive (especially in systems-related specialties), it doesn't have to work particularly well and rarely has a lifetime beyond the awarding of a PhD. Because grad students spend 99% of their time working on their own code, which they necessarily understand intimately, they can get away with leaving things uncommented, choosing cryptic variable names, creating disastrously tangled logic, and even worse coding practices.

The result is that many new professors have only a vague idea of what good, maintainable code should look like. Even if they are committed to the concept of good style (and many are), their inexperience makes them poor judges of quality. It is as if we asked a literature professor to teach novel-writing when they had written only one unpublished, un-critiqued book in their lives; no matter how good their intentions, we would get a few great teachers and a plethora of extremely bad ones.

In the end, students who graduate with a bachelor's degree have spotty educations. They may be fantastic at algorithm analysis, but many write code so bad that their new employers must spend months or even years retraining them before they can be trusted to work alone. And in many cases, their bad habits lead to flawed designs, bugs, and security holes in shipped software.

A Solution?

What can be done to improve the situation? Although it's a tough nut to crack, I believe there are several approaches we can take. Much of the solution falls in the laps of colleges and universities, which, after all, have a primary mission of teaching young people how to succeed in our field.

First, we should make maintainability and coding style an important part of the grade on tests and especially on homework. Grading style is labor-intensive, so it's easy to fall into the trap of only grading functionality (often with automated tools). But as I tell my own students, a perfectly working spaghetti program is worthless because it can't be enhanced, whereas a slightly broken but wonderfully readable program is priceless because any half-decent programmer can fix the bugs and the result will be useful for years to come. So it's worth hiring extra TAs and training them to recognize good coding practices. (You *will* have to train them at first, because they've come up through the same style-doesn't-matter ranks.)

Second, find ways to encourage students to read code. One of the best ways to learn English writing is to read the great authors, and the same holds true for software. Professors should provide their students with large sample programs and require them to be read and understood. Reading good code has a double benefit: the code provides examples of how things should be done, and it develops a skill that is essential for anyone embarking on a career in computing. (Exceptionally demanding—or downright mean—professors might also assign students to work with some astoundingly bad code, which in my experience quickly convinces students that readability matters. The Daily WTF (http://thedailywtf.com/Series/CodeSOD.aspx) is a good source of brief examples of bad programming, although many of the articles are more concerned with weak logic than unreadability.)

Third, we need to recognize the importance of industrial experience for our faculty. When universities hire professors, they should give preference to people who have worked on real production code rather than to those who charged straight through to a PhD without ever taking their eye off the ball. It doesn't take much; even a year or two in the trenches will do wonders to open a young student's eyes. (And the wise researcher will choose students who have a bit of industrial background; not only will they eventually become better faculty candidates, their own research projects will go more smoothly.)

Fourth, encourage pair programming in school settings. Working with a partner is a great way to learn how to explain your code to others and how to write in a more readable fashion. Many colleges and universities have already introduced pair programming in CS courses, so this recommendation is easy to implement.

Fifth, when bringing new grad students onto a project, assign them to maintain and enhance existing code. For example, when I joined a research group as a new grad student, we were just starting a push to turn our researchware into a robust system that we could use internally without endless crashes. In addition to working on my own research, I spent most of a year fixing bugs, which gave me an education in the system that couldn't have been duplicated any other way. The end result was that we had working software and all of the students involved had a practical understanding of maintainable code. Additionally, the original author got useful feedback on the quality of what he or she had written.

Sixth, we should make it clear to our students that "functionality first" is not an acceptable design paradigm. As part of that, we should discourage participation in functionality-only programming competitions and work to develop maintainability-focused ones. (See below for how industry can help with this goal.)

Finally, I believe that all schools should require a software engineering course as part of the standard curriculum, and that the course should teach style and maintainability.

Industry's Contribution

Although our post-secondary educational system carries the primary burden of developing new computer scientists, industry can do some things to help change the current situation.

First, when interviewing job candidates, especially new graduates, find ways to discover whether they can write good code. Famous puzzles may tell you how someone approaches a tricky problem, but they will do little to reveal whether their solution will be something your company can live with for the next decade. How much of your code base was written by a whiz kid who left an unmaintainable mess behind? Wouldn't it have been better to hire someone who worked slightly slower, but produced solid, readable code with a simple API? If you test on style, you might just find that jewel of an employee. And I can promise you that if you regularly test new graduates on the quality of their code, word will get back to their younger peers, who will then develop a strong interest in learning how to pass those tests.

Second, encourage professors to get more industry experience, ideally experience working on existing code. One way to do this is to reach out to faculty—especially young faculty—to offer them sabbatical positions or consulting opportunities. Many professors enjoy coding, are unable to do it on a daily basis, and would welcome the chance to get their hands dirty from time to time. There is nothing like experience with existing code—especially poor code—to teach the importance of style.

Third, think about ways to promote style as a first-order factor. Academia and industry sponsor lots of exciting programs for young students, such as the Google Summer of Code, the ACM Programming Competition, and the Netflix Prize. Unfortunately, the usual emphasis is on "Does it work?" rather than "Can we make this work for a decade?" A contest that required maintainability as well as innovation would be harder to judge, but it would do wonders to make students think about the long-term characteristics of their work, especially if a monetary reward were involved.

Fourth, if you don't already do code reviews, institute them. Programmers hate code reviews because they're embarrassing—which is precisely why they're needed. Even the best coders can benefit from an outside eye, and if the code is good, we can all learn from it. This is one of the reasons pair programming has become so popular; it offers an instant, built-in code review process. But even in a pair-programming shop, separate code reviews can further improve quality.

What Can You Do?

Not all of us are in a position to make the changes suggested above. But we can still change ourselves and try to produce better code. First, read a good book on style. I'm fond of Kernighan and Plauger's dated but still relevant classic, *The Elements of Programming Style,* but there are many alternatives.

Second, learn from the programs you work with. Has the author made your life easy or difficult? Can you figure out what a function does without digging deep into the call tree? Is the information you want buried in a maze of macros, function pointers and virtual function calls, global variables, and messy data structures? Or is everything laid out so elegantly that you wish you could take credit?

Third, when you return to one of your own programs from several years ago, do the same analysis, and be ruthless. Can you figure out what you did, and why you did it? Is there a simpler and clearer way to do things? Has your code grown and changed over time, so that some code paths are obsolete?

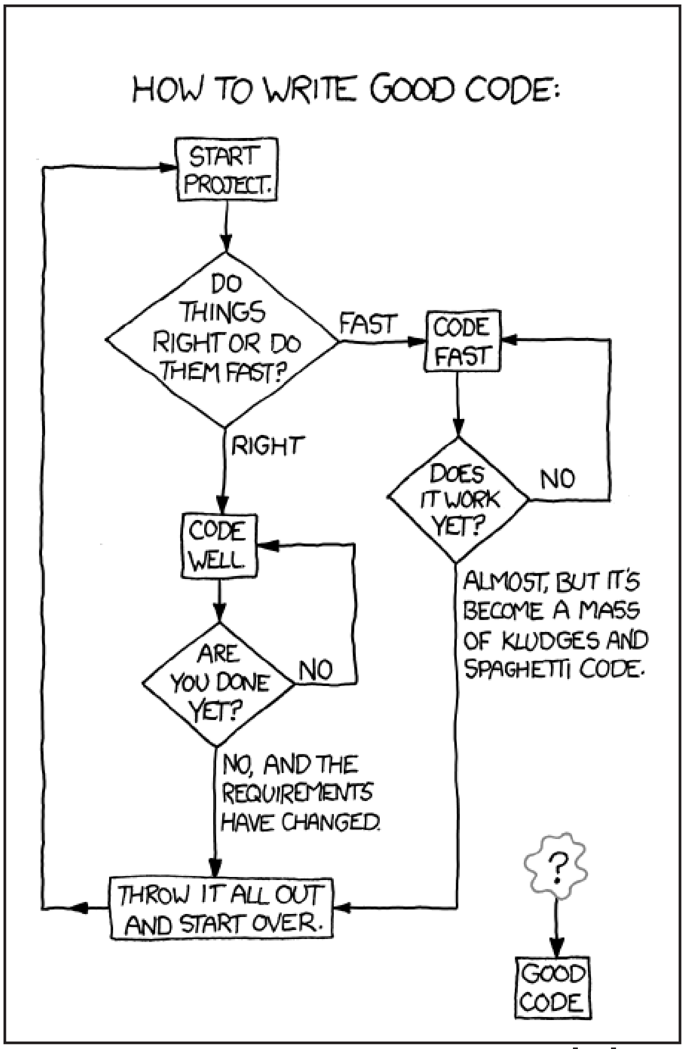
Fourth, show some of your code to a colleague and ask for honest feedback. Do you have enough comments? Are your variable names open to misinterpretation? Does it take ten minutes to figure out that clever for loop you were so proud of, the one with the null body and the tricky use of the side effects of ++? I got slapped down for that last one just a couple of weeks ago, and justifiably so. There's always room for learning.

Is It Hopeless?

As I said above, I don't think we are facing an easy task. When the ACM contest was first announced, I wrote a letter (I believe to the ACM Transactions on Computer Systems; unfortunately the ACM Digital Library doesn't seem to archive letters) suggesting that encouraging students to write hacked-up throwaway code was unwise, and perhaps the contest should instead reward what real programmers do. The suggestion was disdainfully dismissed, and 35 years later we are still lionizing undergraduates for solving toy puzzles with incomprehensible code that will be discarded the next day, never to be seen again. Is this really what we want to encourage? Are these the people you want to hire?

Nevertheless, I think progress can be made. Some of my suggestions above are easy to implement; none are impossible. We should start with baby steps, changing the world one discarded goto at a time. In fact, we have already started; the worst ideas of my youth are long gone, and no modern programmer would dare write unindented code (though, sadly, inconsistency is still rampant). So let us go forth from here and set an example by insisting that *our* students will learn to code well, our own code will be exemplary, and our new hires will earn their jobs by showing that what they write will outlast their careers.

xkcd


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

xkcd.com