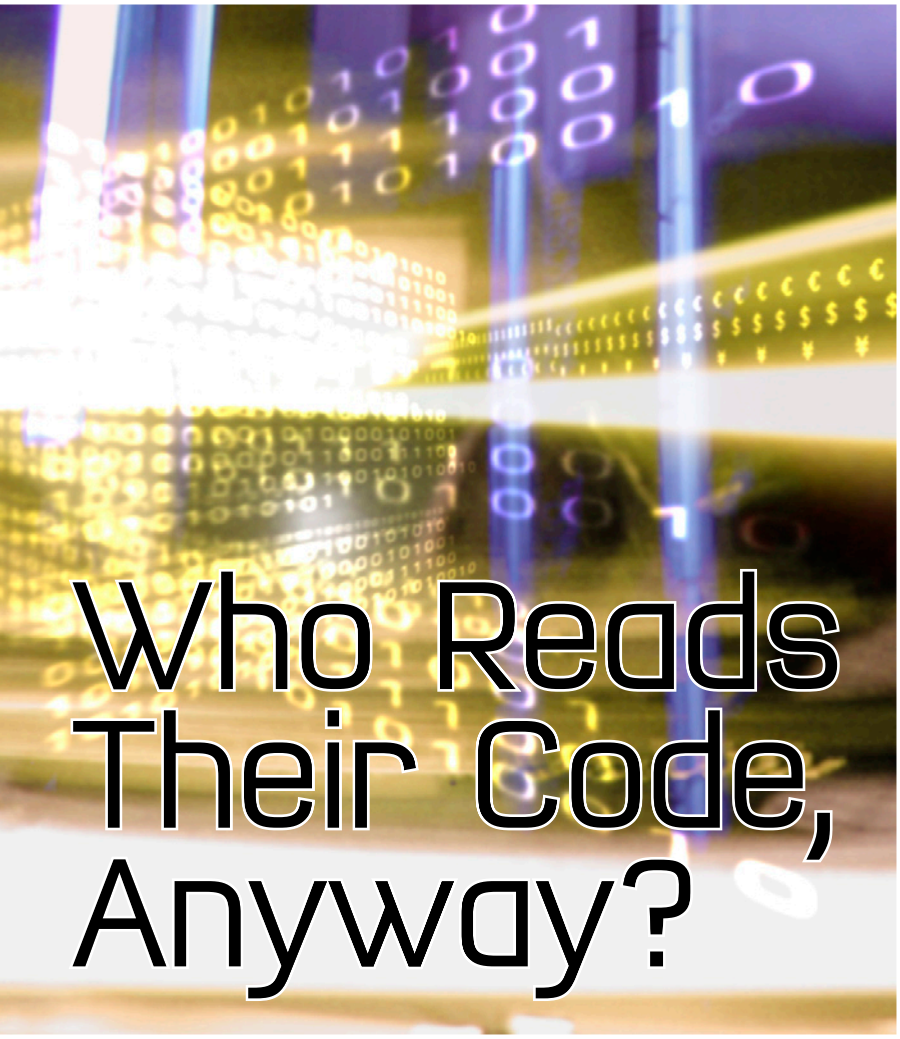

Who Reads Their Code, Anyway?

My very first programming class was back in 1969. The junior high school I went to was blessed to have a Wang Programmable Calculator. It had 256 bytes of memory, a single card reader (as in "single-card reader", not "single card-reader"–80 commands could fit onto a single IBM standard 12-row 40 column card) and a paper tape reader. Programming was pretty basic, as it was meant to be used as a calculator, not a computer. Because of the limited memory and commands, you had to really work to get your programs wedged into the scarce memory. You thought in terms of efficient (albeit hard to read) code.

I was lucky, as my high school had a time-sharing terminal that allowed us to dial in to a GE computer running BASIC and FORTRAN. Yes, Virginia, GE made computers. Back in the 60s, there were eight major computer companies. IBM, the largest, was called "Snow White," followed by the "Seven Dwarfs" (Burroughs, NCR, Control Data Corporation, Honeywell, RCA, UNIVAC and GE). GE eventually sold their computer business to Honeywell. While the programs were bigger and memory not as scarce (the machine we connected to had 96 KILOBYTES of memory!!!), the slow 300-baud modems made efficient coding and debugging critical. My high school had to pay for the long-distance phone calls, so great emphasis was placed on locally desk checking before the costly phone call to upload and download the execution results. It wasn't enough to have efficient code–it had to be easy to read, and easy to debug.

Over the years, there were always tight constraints affecting how I wrote code. At one time, I was working on Contingency Operation/Mobility Planning and Execution System, Logistics Module B. The area I worked on involved calculating the efficient loading of pallets to fit onto an aircraft. This type of problem is referred to as a "bin-packing problem" and is frequently solved by using recursion–a common technique used in programming, allowing you to write programs, procedures, methods or functions that call themselves. While still a widely used and useful language, COBOL had some limitations. At the time, COBOL did not have methods nor did it allow recursion, even at the program level (both of these limitations are now allowed in modern COBOL). The problem was that the code was (and, for all I know, still is) coded using COBOL. Our solution was to simulate recursion using COBOL data structures. For a bunch of young programmers, we quickly realized that you couldn't just hack code. We spent a lot of time designing the code–creating architectural, interface, data, and modular design documents.

I could go on and on–but I know that each and every one of you who has written (or managed) coding projects have learned your own lessons. Although you might not have known it at the time, you almost certainly learned the hard way the "four pillars of software engineering" (reliability, understandability, modifiability and efficiency) as discussed in the book, *Software Engineering with Ada* by Grady Booch.

Reliable programs are important. So are efficient ones. And, given changing requirements, modifiable programs are critical. But, without understandable programs, the other three pillars are pretty much impossible. You might possibly write a reliable and efficient program that works initially, but if you can't understand the code, then attempts to modify it will certainly cause it to fail.

The best ophthalmologist I ever had was an Air Force physician who had a business card that read, "If you can't see them, you can't shoot them." Clarity is important in vision, ***and*** in software. I once took the Personal Software Process^{SM} class, and then taught it for many years. I learned the hard way that the best thing I could do to write code quickly was to write it clearly–because debugging was a very time-consuming activity. I learned the hard way that clear, easy-to-understand code is much quicker to debug and modify.

People read the code we write. The code will be around for years and years, and will be read and modified and re-read, and modified again. When it gets down to the basics, coding is a very people-based activity. To quote from the BackTalk column in the 2010 November/December issue of *CrossTalk*, "Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live" (attributed to Martin Golding).

After all, if you can't read it, you can't fix it.

David A. Cook
Stephen F. Austin State University
cookda@sfasu.edu

Disclaimer: