Software Engineering Advice

Truman Collins September 20th 2014

I have been employed for 26 years as a Software Developer working on various products in the Electronic Design Automation industry. The problems were intellectually challenging, and coding for efficiency and clarity were critical for long-term success. I would like to offer some advice that should be helpful to someone beginning a career in software. I did most of my work in C++, but this advice should be applicable across all languages.

Clean Up Your Code After It Works: This is my most important advice. It will significantly reduce the number of bugs in your code, increase efficiency, and make your code much easier to understand, both for you and anyone else maintaining your code. I list this first because it is so important, but it involves the other recommendations listed here.

Typically, if you are writing a non-trivial piece of code, there are some hoops you go through before you actually get it working. Maybe you have in mind how to do it, and as you write the code, you realize that you have to do a couple of things you didn't realize. Maybe once you get it written, you test it with some simple input and it doesn't do what you expected and you do some debugging. Whatever your path to your code working, it probably doesn't look quite like you thought it would when you started, and it probably isn't organized very well. It works, however, and you just want to be done with it, so you check the code in or turn it in or stop working on it. This is a mistake that will cost you time, effort, and reputation.

This code could easily have bugs that you haven't tested for or uses you haven't considered. It probably wouldn't be very understandable to someone (likely you) some months or years down the road when a bug is discovered or enhancements are needed. The comments and meaningful names of objects, if you went to the trouble to add them, may not be quite right anymore since the code evolved from what you thought would work to what really worked.

At this moment, you understand what you were trying to accomplish, the constraints on the problem, and how the code works better than you ever have or ever will again. This is the moment to spend the time to walk over the code line-by-line, think deeply about what it's doing and what cases you might not have considered, add or modify comments to clearly explain what the code is doing, name variables and functions to improve understanding, and re-factor the code to simplify.

Doing this takes time and may feel wasteful when you are in a hurry, but when this is made a habit, it will transform your productivity and significantly reduce the bugs in your code. Doing this consistently will put you a level above, in terms of quality, from where you would have been otherwise.

Comment Code Clearly: I have found that it pays to comment blocks of code as to generally what they are doing and algorithms for the ideas behind how they solve problems. The comments should include boundary conditions and how special cases are handled. This is valuable not only for someone else who has to maintain your code in the future, which might well be you, but also because the process of describing in English how code works can help to identify problems or cases that you didn't consider before.

Excessive commenting can be counter productive because the more extensive comments are, the harder they are to update when things change.

It Pays To Re-Factor Code: Putting common code that can be used by different parts of a program into a single place, simplifying interfaces, and designing code so that it's harder to accidentally mis-use, are well known recommendations. It is almost always good to strive to achieve these practices from the beginning or spend the time to make these improvements after code is added. It's tempting to just get something to work, but in the long run, good design pays in fewer bugs, more maintainable code, and a smaller volume of it.

Use Meaningful Names: Functions and variables should be given names that are meaningful and add to the clarity of your code. Excessively long names are a hindrance to understanding, but when your code is read later, good names improve understanding.

Memory and Speed: In school I was often told that there is a trade-off between space and speed, and that generally to make a program faster, you have to use more memory, and vice versa. My experience has been that while this is sometimes true with highly optimized code, it is not a general rule at all. Frequently, I have found that by spending some time cleaning up code—mine or someone else's—I can improve memory use, performance, and clarity all at the same time. Also, note that reducing memory use will increase the amount of your data that can live in memory cache, which has much faster access.

Efficiency and Memory Use Matter: As compared with twenty or thirty years ago, computers are incredibly fast and memory is cheap. There is much less emphasis on writing efficient programs that use memory sparingly. I think there is still real value to these, however. Programs are often run under much more demanding input than expected. Time and time again, I have seen customers run into performance problems where the designers reaction was to say "I never expected that they would use it that way!"

Sometimes I think it would be good if software were designed using a tiny machine over a slow remote connection. Then, if you accidentally allocate too much memory it will show up right away. I was once using a GUI where after a particular operation, the screen would flash and show the new state of things. I ran it remotely over a slow connection and discovered that it was redrawing the entire screen three times in that one flash, but no one had noticed.

Also, keep in mind that even if your program doesn't run out of memory or run too slow, it may be running concurrently with other programs that are using significant resources that crowd yours out.