The Meaning of Maintenance

Software maintenance is more than just bug fixes.

Dear KV,

Isn't *software maintenance* a misnomer? I've never heard of anyone reviewing a piece of code every year, just to make sure it was still in good shape. It seems like software maintenance is really just a cover for bug fixing. When I think of maintenance I think of taking my car in for an oil change, not fixing a piece of code. Are there any people who actually review code after it has been running in a production environment?

Still Under Warranty

Dear Still,

The short answer is that, yes, the term *software maintenance* is yet another computer industry bit of Newspeak, probably invented so that people wouldn't have to put *bug fixer* on their resumes. Since there is strength in ignorance, I should probably not answer your question any further, but I find it impossible now to follow the party line.

Although the term *software maintenance* has little meaning other than bug fixing, the question you ask has some deeper implications. The reason to do maintenance on a car, or any other machine, is that it has moving parts that wear out over time. Parts wear out because they are subjected to physical stresses, such as two gears, one of which drives the other in order to transfer energy from a motor to a wheel. Software isn't subjected to physical stresses, though there are pieces of code I've read, only just recently, that rightly deserve to be put under physical stress—or, at least, their authors do. Even though software itself does not wear out after it is executed repeatedly, there *is* a place for maintenance in the software industry.

The more modern term for what really is software maintenance is another bit of Newspeak: *refactoring*. It is unfortunate that every term we come up with to describe something simple and direct in our industry is almost instantly debased so that it loses all meaning. While I'm not generally against debasement, in this case it does make our lives a bit more difficult. Many people wrongly use the term *refactoring* to stand in for: "Oh, we really screwed up the entire system we were writing over the past six months, so we're going to have to write it again from scratch, but we'll keep the name of the program the same, as well as the names of many of the classes. We will, of course, have to change all the insides of the classes, their method signatures, and about 90 percent of the rest of the text that was the program, but from the outside it will look the same, except for all the features; those will change too." When you replace most of your API, the innards of the code, and the features, that is not refactoring; nor is it maintenance. That's a rewrite.

Refactoring actually means that you have some functions or classes that, with a small number of changes, can be used in another program. It is this more honest type of refactoring that begins to look like software maintenance. The reason that refactoring is more like maintenance is because you're looking at several moving parts that no longer mesh well with each other. In the previous program or system in which they were used, they meshed well—otherwise, that program would

not have worked correctly—but they do not mesh correctly with the new design; therefore, you’re required to add a metaphorical bit of grease, or break off a few teeth from the gears to make them work well in their new application.

When you’re refactoring a piece of code is also the perfect time to think about the original design: if it made sense originally, if it makes sense now, and if you want to be stuck with this same design in the future. I am NOT saying that you should rework every single piece of code you see just to make it a bit cleaner, nicer, more generic, etc. You can’t actually even imagine that I would give people license just to diddle with all the code in a system. That kind of navel-gazing really ought to result in whippings, but I hear that those are not allowed in most workplaces at this point.

While you’re maintaining—oh, I mean refactoring—please remember that whatever you change must be tested. Just because you “changed the API only a little bit by adding one teensy-weensy little bit field” does not excuse you from testing your change. If you don’t, I can guarantee that you’ll be back doing the old kind of software maintenance (i.e., bug fixing).

KV

Dear KV,

A co-worker recently chewed me out for including one C file within another C file. Though this may not be the way many people build software, it doesn’t seem wrong to do so, and the compiler doesn’t complain. Is this really wrong, or just unorthodox?

All in One and One in All

Dear All in One,

There are many things that a compiler will not complain about. Compilers, unlike people, have no moral sense and therefore have only a very limited idea of right and wrong. Most people have a better sense of right and wrong than a compiler—most, but not all. Clearly, you are one of those people who does not have a highly developed moral sense, for if you did, you would have realized, even before e-mailing me, that what you were doing was wrong. Why was it wrong?

The simplest reason not to include one C file within another is that it obfuscates the relationships between pieces of code. The few times I’ve seen this kind of thing in practice, it has been done at the end of a C file—sort of like tacking on one file at the end of another. If, for example, you don’t wind up reading the very end of the file, you’ll never know that there is one file (or more) built when the code you’re looking at is compiled. Such surprises are an unwelcome part of a programmer’s day. It’s like grabbing a bag that you thought contained a few loaves of bread only to find out that there is an anvil at the bottom.

If you need to use a piece of code in two places, you don’t `#include` it in both; you build it as a separate module and use the linker to put the pieces together at the end when you build the final executable.

We are no longer programming in the 1950s, and we are not building programs out of paper-tape libraries where you splice together a program out of smaller segments of paper tape. In an environment with linkers, loaders, and compilers capable of generating and working with separate modules, there is no excuse for `#including` one compilable file within another.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

© 2009 ACM 1542-7730/09/0800 \$10.00