Wanton Acts of Debugger

Keep your debug messages clear, useful, and not annoying.

Dear KV,

Why is it that people who add logging to their programs lack the creativity to differentiate their log messages? If they all say the same thing—for example, DEBUG—it's hard to tell what is going on, or even why the previous programmer added these statements in the first place.

Suffering from Similarity

Dear Suffering,

Undifferentiated noise coming from a program is like the sound of fingernails on a blackboard: it is both annoying and useless and generates in the listener an instinctive need to make it stop, violently if necessary. Of course, no sane programmer adds identical debug statements in one instance. All software projects last longer than their original designers think they should. As with many of the problems that plague long-lasting software projects, debug statements accrete like barnacles on the bottom of a boat. And just like barnacles, they have to be scraped off from time to time. People with a modern, Orwellian bent call this “refactoring”; I prefer to call it “cleaning up the crap you should have cleaned up long ago.” If you're very lucky, then all of these statements are contained under some sort of conditional compilation directive or `if (debug)` statement. If they're simply littered throughout the code, you already have a bigger problem, as the original coder was either insane or incompetent.

Good debugging statements have a few things in common. The first requirement is that they can be TURNED OFF. Many parents say children should be seen and not heard, and the same goes for software. If you're not actively debugging a system, then you shouldn't see debugging output flying by or being dumped unceremoniously into a log file. Second, like any other code you're going to put into a program, debug statements need to have meaning.

In a previous column (“File-system Litter,” July 2011; <http://queue.acm.org/detail.cfm?id=2003323>) I showed a simple Emacs macro to insert debugging statements that in C and C++ would show you at which line in a file the program had been executing. Knowing which program statement you've just passed is the kind of debug logging that should be strictly temporary and, I believe, should never be checked into a source repository. It's the kind of thing that will be forgotten once the original programmer goes on to another task, and once forgotten will cause great annoyance when found again. If you're really going to keep a debug statement around, then it ought to be more descriptive than a simple line and file. It needs to say why it's there: Is this a critical error? Does it indicate a problem that requires human intervention?

These two questions bring me to the next requirement, which is that of levels. If you don't break your debug messages down by level, then all of them will seem—to the people who later run your software—to be critical and thus cause great consternation. Most programmers now differentiate their debug messages so that they fit into a small number of categories: info, warning, and error. An *info* message is informational; it is simply telling someone something about the state of the program

but does not mean the program made a mistake or that a human needs to become involved. A *warning* is more severe than an info message and usually indicates that some person needs to know about the status of the program. Good targets for warnings are early indications that resources—such as disk space or memory—are running low and that some sort of human intervention—such as moving or removing old log files—can clear away the problem. An *error* is just that—an error in the program. The error clearly didn’t make the program crash, but something went wrong in such a way that a person needs to go in and clean up whatever mess the program made. In some systems, such as realtime control systems, an error will stop the program, but as users of modern desktop and server software are well aware, many systems log errors and then happily go back to doing whatever it was they were doing before the error occurred.

Now that we have a reasonable set of possible debug messages, let me say a bit about involving people when problems do occur. I have known many good systems administrators in my time, and the thing they like least is to have some program wake them up at 3 a.m. for no good reason. In fact, they will curse you and your software if it cries wolf too often. *Often* is usually measured in single digits of 3 a.m. wake-up calls. If you’re going to insert a debug message that has even the slightest potential of waking up someone in the middle of the night, you had better be absolutely sure that it’s going to be important, and that if it’s a clearable error, that the person who gets woken up can clear it and go back to sleep. Have you ever seen a red-eyed systems administrator at a 9 a.m. triage meeting? If you have, then you know that you don’t want to see that again. If you’re going to log problems in such a way that they demand a person’s attention, then you had better be absolutely sure that the person they annoy can turn them off.

KV

Dear KV,

Why are companies so cheap with test hardware? I just spent two weeks debugging a program in the middle of the night because a new piece of hardware we bought was available only in our live systems. If I’d had this hardware to test with before it was deployed, the problem would have been solved in less than half the time.

All Nighter

Dear All,

It probably won’t make you feel better if I tell you that back in the day, long before I was a programmer, almost all debugging was done in the middle of the night, because the hardware was both expensive and rare. You couldn’t stop running the payroll program just to let someone try to fix a bug in the system, unless that bug stopped the payroll program itself from running.

Of course, in our present circumstances of cheap and plentiful hardware, there are fewer arguments against buying test hardware. If I’m dealing with a particularly cheap client, I’ll simply mark the test hardware as “spares,” a simple ruse to make the accountants happy. Everyone knows that things break and that you would hate to have to wait a week for a spare part to show up, so you must keep them on hand—in a test system near you.

There are still times, though, when some esoteric piece of hardware is going to be too expensive to order as a spare. Here is my way of estimating how expensive is “too expensive.” If the piece of hardware costs more than 50 percent of your yearly wage, then it’s probably too expensive to buy a

test version; any less, and it's time to negotiate with management, because your time is also worth money. The problem is that many companies don't recognize that fact because the accounting department doesn't track hours, but they do notice how much each piece of hardware costs. If you make \$100,000 per year, a not-unlikely figure, then your hourly wage is about \$50. If it takes two weeks to fix a problem, then that's 80 hours, or \$4,000. If the problem could have been fixed in half the time, or 40 hours, then the company could save \$2,000. Sitting around filling out time sheets is something most engineers, thankfully, no longer have to do, but it still makes some sense to pay attention to what your time costs—in particular, if you're trying to convince management to buy hardware that it considers to be an extra cost.

When I work on software for a new piece of hardware, I always include test hardware in the project budget, because, as you have already seen, it's better than staying up nights to fix problems in production. Oh, and don't believe hardware vendors when they say a new piece of hardware is just like an old piece of hardware, only better, stronger, or faster. Their claim might be true, but it's also just as likely to expose a bug in your deployed code. Make sure to get test hardware for the new revision as well.

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.

© 2011 ACM 1542-7730/11/1000 \$10.00