acmqueue

Code Abuse

One programmer's extension is another programmer's abuse.

Dear KV,

During some recent downtime at work, I've been cleaning up a set of libraries, removing dead code, updating documentation blocks, and fixing minor bugs that have been annoying but not critical. This bit of code spelunking has revealed how some of the libraries have been not only used, but also abused. The fact that everyone and their sister use the timing library for just about any event they can think of isn't so bad, as it is a library that's meant to call out to code periodically (although some of the events seem as if they don't need to be events at all). It was when I realized that some programmers were using our socket classes to store strings—just because the classes happen to have a bit of variable storage attached, and some of them are globally visible throughout the system—that I nearly lost my lunch. We do have string classes that could easily be used, but instead these programmers just abused whatever was at hand. Why?

Abused API

Dear Abused,

One of the ways in which software is not part of the real world is that it is far more malleable—as you've just discovered. Although you can use a screw as a nail by driving it with a hammer, you would be hard pressed to use a plate as a fork. Our ability to take software and transmogrify it into shapes that were definitely not intended by the original author is both a blessing and a curse.

Now I know you said you clearly documented the proper use of the API you wrote, but documentation warnings are like yellow caution tape to New York jaywalkers. Unless there is an actual flaming moat between them and where they want to go, they're going to walk there, with barely a pause to duck under the tape.

Give programmers a hook or an API, and you know they're going to abuse it. They're clever folks and have a fairly positive opinion of themselves, deservedly or not. The APIs that get abused the most are the ones that are most general, such as those used to allocate and free memory or objects, and, in particular, APIs that allow for the arbitrary pipelining of data through chunks of code.

Systems that are meant to transform data in a pipeline are simply begging to be abused, because they are so often written in incredibly generic ways that present themselves to the programmer as a simple set of building blocks. Now, you may say these were written as building blocks for networking code, or terminal I/O, or disk transactions; but no matter what you meant when you wrote them, if they're general enough, and you leave them in a dark place where other coders can find them, then the next time you look at them, they may have been used in ways unrecognizable to you. What's even better is when people abuse your code and then demand that you make it work the way they want. I love that, I really do—no, I don't!!!

One example is the handling of hardware terminal I/O in various Unix systems. Terminal I/O systems handle the complexities inherent in various hardware terminals. For those too young to have ever used a physical terminal, it was a single-purpose device hooked to a mainframe or

minicomputer that allowed you to access the system. It was often just a 12-inch-diagonal screen, with 80 characters by 24 lines, and a keyboard. There was no windowing interface. Terminal programs such as `xterm`, `kterm`, and `Terminal` are simply a software implementation of a hardware terminal, usually patterned on the Digital Equipment Corporation's VT100.

Back when hardware terminals were common, each manufacturer would add its own special—sometimes very special—control sequences that could be used to get at features such as cursor control, inverse video, and other modes that existed on only one specific model. To make some sense of all the chaos wrought by the various terminal vendors, the major variants of Unix such as BSD and System V created terminal-handling subsystems. These subsystems could take raw input from the terminal and, by introducing layers of software that understood the vagaries of the terminal implementations, transmogrify the I/O data such that programs could be written generically to, say, move the cursor to the upper left of the screen. The operation would be carried out faithfully on whatever hardware the user happened to be using at that moment.

In the case of System V, though, the same system wound up being used to implement the TCP/IP protocol stack. At first glance this makes some sense, since, after all, networking can easily be understood as a set of modules that take data in, modify it in some way, and then pass it to another layer to be changed again. You wind up with a module for the Ethernet, then one for IP, and then one for TCP, and then you hand the data to the user. The problem is that terminals are slow and networks are fast. The overhead of passing messages between modules isn't significant when the data rate is 9600 bps; but when it's 10 Mbps or higher, suddenly that overhead matters a great deal. The overhead involved in passing data between modules in this way is one of the reasons that System V STREAMS is little known or used today.

When the time finally came to rip out all these terminal I/O processing frameworks (few, if any, hardware terminals remain in service), the number of things they had been extended to do became fully apparent. There were things that were implemented using the terminal I/O systems pretty much as a way to get data into and out of the operating-system kernel, completely unrelated to any form of actual terminal connection.

The reason these systems could be so easily abused was that they were written to be easily extended, and one programmer's extension is another programmer's abuse.

KV

Dear KV,

My company has been working for several weeks on upgrading libraries on our hosted systems. The problem is that we have to stop all our users from running on these systems during the upgrade, and this pisses them off. It's nearly impossible to explain that the upgrades need to be atomic. In fact, they don't seem to understand the original meaning of *atomic*.

About to Go Nuclear

Dear Nuke,

Ah yes, ask any programmer about "atomic operations," and if they have any clue at all, they'll go on about test and set instructions and maybe build you a mutex or a semaphore. Unfortunately, this micro-level understanding of how to protect data structures or sections of code from simultaneous access does not always translate to the macro world where a whole set of operations must be

completed in order to get the job done. For some reason the complex connections between bits of software and how modules of code can relate to each other escapes many people—users and programmers alike.

You and I both know that an atomic operation is simply some job that must be completed, without interference, in one transaction. An atomic operation is one that simply cannot be broken down any further. The case of updating libraries of code is actually a good example.

Bits of code all have interdependencies. When a library is changed, all the code that depends on that library must change to remain compatible with the library in question. Modern programs link against tens, hundreds, or sometimes thousands of libraries. If the linkage were all one way—that is, the program connected only to each of the libraries—that would be complex enough. In reality, however, many libraries require other libraries, and so on, until the combinatorial explosion makes my head hurt—though, actually, I think that’s the hangover from having drunk too much after trying to understand the aforementioned interconnections.

To update a single library in the system, you would need to understand which other libraries depended on it, and how their APIs changed, and if those libraries were also up to date. This is the point where everyone throws in the towel and just upgrades everything in sight, making the size of the atomic operation quite large. Perhaps the easiest way to explain this to your users is to make them graph the connections between the various bits of code, as can be done with systems such as Doxygen. Then, while they’re off scratching their heads, you can bring down the system, upgrade it, and restart it, long before they have figured out the graph.

KV

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

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.

© 2012 ACM 1542-7730/11/1100 \$10.00