Outsourcing Responsibility

What do you do when your debugger fails you?

Dear KV,

I've been assigned to help with a new project and have been looking over the admittedly skimpy documentation the team has placed on the internal wiki. I spent a day or so staring at what seemed to be a long list of open-source projects that they intend to integrate into the system they have been building, but I couldn't find where their original work was described. I asked one of the project team members where I might find that documentation and was told that there really isn't much that they need to document, because all the features they need are available in various projects on github.

I really don't get why people do not understand that outsourcing *work* also means outsourcing *responsibility*, and that in a software project, responsibility and accountability are paramount.

Feeling a Sense of Responsibility

Dear Responsible,

While it might seem that the advent of the “fork me on github” style of system design is a new thing, I, unfortunately, have to assure you it is not. Since the invention of the software library, sometime before I was born, and probably before you were as well, the idea that one could build a system by just grabbing the bits one needed has been the way software has been built. We all depend on bits of code that we didn't write, and often on code we cannot even read, as it arrives in binary form. Even if we could read it, would we? The code to OpenSSL was open source and readable by anyone who cared or dared, yet the Heartbleed bug sat around for two years undiscovered. The problem isn't just about being able to see the code; it has a lot more to do with the complexity inherent in what you might be dragging in to get the job done.

You are correct to quiz the other team members as to why there isn't any documentation on how they intend to stitch together the various bits they download. Even if many parts are made up of preexisting software, there must be an architecture to how they are integrated. In the absence of architecture, all is chaos, and systems that are built in that organic mold work for a while, but eventually they rot, and the stench they give off is the stench of impending doom.

A software system is always built from other components, and the questions that you need to ask are:

- How trustworthy is the component I'm using?
- How stable is the API?
- Do I understand how to use this component?

Let me break those down for you.

Trustworthiness of software isn't simply a matter of knowing whether someone wrote it for the purpose of stealing information, though if you're taking factors for your elliptic curve code from a three-letter agency, you might want to think really hard about that. To say that software is trustworthy is to know that it has a track record—hopefully, measured in years—of being well tested and stable in the face of abuse. People find bugs in software all the time, but we all know a trustworthy piece of software when we use it, because it rarely fails in operation.

Stability of APIs is something I have alluded to in other responses, but it bears, or should I say seems to require, frequent repetition. If you were driving a car and the person giving you directions revised them every block, you would think that person had no idea where the hell he or she was going, and you would probably be right. Similarly, a piece of software where the APIs have the stability of Jell-O indicates that the people who built those APIs didn't really know what they were doing at the start of the project, and probably still don't know now that the software has a user base. I frequently come across systems that seem to have been written to solve a problem quickly—and in a way that gets Google or Facebook to fork over a lot of cash for whatever dubious service has been created with it. An API need not be written in stone, but it should be stable enough that you can depend on it for more than a point release.

Understanding the use of a component is where the github generation seems to fall on its face most often. Some programmers will do a search based on a problem they're trying to solve; find a Web page or entry in Stack Overflow that points to a solution to their problem; and then, without doing any due diligence, pull that component into their system, regardless of the component's size, complexity, or original intended purpose. To take a trivial example, I typed "red black tree" into github's search box. It then spat out, "We've found 259 repository results." That means there are 259 different implementations of a red black tree present. Of course, they span various languages.

Repositories	Language
56	C
43	Java
41	C++
17	JavaScript
13	Python
9	Ruby
8	Go
8	C#
4	Haskell
3	Common Lisp

How are we to evaluate all (any?) of these implementations? We can sort them by user ratings (aka "stars"), as well as forks, which is how many times someone has tried to extend the code. Neither of these measurements is objective in any way. We still don't know about code size, API stability, performance, or the code's intended purpose, and this is for a relatively simple data structure, not for some huge chunk of code such as a Web server.

To know if a piece of code is appropriate for your use, you have to read about how the author used it. If the author produced documentation (and, yes, I'll wait until you stop laughing), then that might give an indication of his or her goal, and you can then see if that matches up with yours. All of this is the due diligence required to navigate the sea of software that is churned out by little typing fingers every day.

Lastly, you are quite right about one thing: you can outsource work, but at the end of the day it's far harder to outsource responsibility.

KV

Dear KV,

What do you do when your debugger fails you? You have talked in the past about tools that you use to find bugs without resorting to print statements, such as `printf()` in C, and their cousins in other languages, but there comes a time when tools fail, and I find I must use some form of brute force to find the problem and solve it.

I'm working with a program where, when we dump the state of the system for an operation that is supposed to have no side effects, the state clearly changes; but, of course, when the debugger is attached to the program, the state remains unchanged. Before we resort to print statements, maybe you could make another suggestion.

Brute Forced

Dear Brute,

Tools, like the people who write them, are not perfect, and I have had to resort to various forms of brute-force debugging, forsaking my debugger for the lowly likes of the humble print statement.

From what you have written, though, it sounds like another form of brute force might be more suitable: binary search. If you have a long-running operation that causes a side effect, the easiest way to find the part of the operation that's causing you trouble is to break down the operation into parts. Can you trigger the error with only half the output? If so, which half? Once you identify the half that has the bug, divide that section in half again. Continue the halving process until you have narrowed down the location of the problem and, well, not quite voila, but you'll definitely have made more progress than you would by cursing your debugger—and it will take less time than adding a ton of print statements if the segment of the system you're debugging is truly large.

Often print statements will mask timing bugs, so if the bug is timing related, adding a print statement may mislead you into thinking the bug is gone. I have seen far too many programmers ship software with debug and print statements enabled, although the messages go into `/dev/null`, simply because “it works with debug turned on.” No, it doesn't “work with debug turned on”; the debug is masking the bug and you're getting lucky. The user of the software is going to be unlucky when the right moment comes along and, irrespective of the print statements, has a timing error. I hope you're not working on braking systems or avionics, because, well, boom.

If your goal is to find the bug and fix it, then I can recommend divide and conquer as a debugging approach when your finer tools fail you.

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.

© 2014 ACM 1542-7730/14/0600 \$10.00