Understanding the Problem

For those of you new to *Queue*, Kode Vicious is our monthly column devoted to the practice of programming. This is where our resident code maven responds to your questions about everything from debugging to denial-of-service attacks. Whatever your concern, Kode Vicious will break it down, sort it out, and, we hope, set you straight. Have a question for KV? E-mail him at kv@acmqueue.com and let him know what's bugging you. If we print your question, we'll send you a special piece of *Queue* memorabilia.

Dear KV,

I've done a one-day intro class and read a book on Java but never had to write any serious code in it. As an admin, however, I've been up close and personal with a number of Java server projects, which seem to share a number of problems:

- Performance—about 10 times slower than C.
- Complexity—programs seem very large and obtuse.
- Slow to code—projects always running behind.

Is there any data showing that Java projects are any more or less successful than those using older languages? Java does have heavy commercial support, as well as the noble aim of helping programmers reduce certain types of errors. But as professional programmers, we use sharp tools, and they are dangerous for exactly the reasons they are useful. Trying to protect everyone from “level 1” programmer errors seems very limiting to me.

I keep seeing projects to replace legacy apps start amid fanfare and hoopla—and with significant budgets—using the most “modern” techniques, only to end up being cancelled or only partially implemented.

Am I missing something?

Run Down With Java

Got a question for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!

A koder with attitude, KV ANSWERS

YOUR QUESTIONS.

MISS MANNERS HE AIN'T.

Dear Run Down,

Having taken a course on Java and read a book on it, you're actually ahead of old KV on the Java wave.

I'm still hacking C, Python, and bits of PHP for the most part. Given your comments, perhaps I'm lucky, but somehow I doubt that. I'm rarely lucky.

I could almost reprint your letter without comment, but I think there are larger issues that you raise, and I really can't let these things go without commenting or, perhaps, screaming and tearing my hair out. It turns out that shaving my head has helped with all those bald patches I got from tearing my hair out.

As a reader of KV, you've probably already realized that I rarely bash languages or make comparisons among them, and I'm going to stick to my guns on that, even in this response. I don't believe the majority of the problems you're seeing come from Java itself, but from how it is used, as well as the way in which the software industry works at this point in time.

The closest I've come to Java was to work on a project to build some lower-level code in C that would be managed by a Java application. There were two teams: one that wrote the systems in C, which could operate independently of the Java management application; and one that wrote in Java. Now, you would expect that the Java team and the C team would have met on a regular basis, and that they would have exchanged data and designed documents so that the most effective set of APIs could be built to manage the lower-level code efficiently. Well, you would be wrong. The teams worked nearly independently, and most of the interactions were disastrous. There were many reasons for this, some of which were traditional management problems; but the real reason for this “failure to communicate” was that the two teams were on two different worlds and no one wanted to string a phone line between them.

The Java team members were all into abstraction. Their APIs were beautiful creations of sugar and syntax that scintillated in the sunshine, moving everyone to gaze in wonder. The problem was that they didn't under-

stand the underlying code they were interacting with, other than to know what the data types and structure layouts were. They did not have a deep appreciation of what their management application (so-called) was supposed to manage. They made grand assumptions, often wrong, and when they ran their code it was slow, buggy, and crashed a lot.

The C team wasn't perfect either. There was a certain level of arrogance, shocking I know, toward the Java team—and although information wasn't hidden, it was certainly the case that if the C engineers thought the Java engineers didn't "get it," they would just throw up their hands and walk away. The C team did produce code that shipped and worked well. The problem was that the goal of the company was to build an integrated set of products that could be managed by a single application. Although the C team won the battle, the company lost the war.

I don't believe the majority of the problems you're seeing come from Java itself, but from how it is used.

Someone looking at the code as it was delivered might have thought, "Well, the Java programmers just weren't up to the task; next time hire better programmers, or get better tools or..." The fact is, that's not the real problem here. The problem isn't Java; it was the fact that the people building the system could produce a lot of lines of code but didn't understand what they were building.

I have seen this problem many times. It often seems that projects are planned like some line from an old Judy Garland/Mickey Rooney musical. One character says to the other, "Hey kids, let's put on a show!" It always works in the movies, but as a project plan it rarely leads to people living happily ever after.

To build something complex, you have to understand what you're building. The legacy applications you mention are another great example. Ever seen a company

convert a legacy app? I hope not; it's not very fun. Here's the way legacy conversion goes: You have a program that works. It does something. You may have the source code, or you may not. No laughing now, I've seen this. When the legacy program runs, it does what it should, most of the time. Next the team comes in and tries to dissect what the program does and then reproduce it, with bug-for-bug compatibility, and they find that their modern techniques don't reproduce the same bugs in the right way. So they get to a point where the program sort of works, or sort of doesn't, and then they usually give up and reimplement whatever it was, from scratch.

One of the reasons such travesties can continue to occur is that unlike in any engineering discipline in the real world (think aeronautics or civil engineering), failure just means a loss of money.

Now, when I say "just," that can be a big just. The overhaul of the IRS computer systems cost millions in overruns, as did the system developed for the Department of Motor Vehicles in California. There is a laundry list of such failed projects to choose from. These may make headlines for a while, but they're not quite on the level of a bridge failing, like the Tacoma Narrows, or the space shuttle exploding, twice. People generally remember where they were when the space shuttle *Challenger* blew up, but they don't remember where they were when they heard about an IRS computer cost overrun.

With more and more computers and software being put into mission-critical systems, perhaps this attitude will change with time.

Unfortunately, we're going to need a few more spectacular failures, likely with a human instead of monetary cost attached, before people put more time into planning what they do and figuring out what their code is actually meant to be doing. Once we do that, the fact that we're using Java or Perl or the language du jour will have a lot less effect and will probably be discussed a lot less 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 has made San Francisco his home since 1990.

© 2006 ACM 1542-7730/06/1100 \$5.00