The Next Big Thing

A koder with attitude, KV ANSWERS

YOUR QUESTIONS.

MISS MANNERS HE AIN'T.

To put it another way, there is no Holy Grail.

Your question is really a more general version of questions that have been

asked before about whether language X is better or worse than language Y. I've gone over that territory in this space before (November 2006, February 2006, September 2005), and the answer is still the same: it depends.

Whether you use functional programming depends on what kind of problem you're trying to solve, in what environment you're solving it, and with whom you're solving it. Some problems decompose well into programs written in a functional language, others do not. Many problems *can* be solved with functional programming, but sometimes they shouldn't.

So, if you have a problem that looks like it will be easy to solve with functional techniques, and that means that the solution will be elegant, easy to understand, and will perform well under the expected workload, then you might want to solve it in a functional language. I say *might* because the conditions I just gave are only the first barrier to overcome. If, for example, you work in a place where C or C++ is the overwhelmingly popular language and most problems are solved using those tools, and most of your co-workers are using those tools, then you would truly be a fool to use anything else. As with many things, set and setting are important; otherwise, you're in for a bad trip.

Now, if you're working in a place with a diversity of styles and tools and the people you are working with are amenable to using functional programming, then by all means you should use it. Lots of great software has been written using functional programming, and I expect that to continue. Of course, lots of crap software has also been written using functional programming and I also expect that to continue. Software, in the end, is usually less about the tools than it is about the people and the thought processes that go into it. So, by all means, keep up on the functional programming, but I wouldn't throw out your copy of Kernighan and Ritchie just yet.

KV

This month Kode Vicious considers the future of functional programming and then offers up his top five protocol-design tips. Enjoy! And then send your own code-related questions to kv@acmqueue.com. We'll thank you with an authentic *Queue* tchotchke.

Dear KV,

I know you did a previous article where you listed some books to read ("Kode Vicious Bugs Out," April 2006). I would also consider adding *How to Design Programs*, available free on the Web, <http://www.htdp.org/>.

This book is great for explaining the process of writing a program. It uses the Scheme language and introduces FP (functional programming). I think FP could be the future of programming—John Backus of the IBM Research Laboratory suggested this in 1977 (<http://www.stanford.edu/class/cs242/readings/backus.pdf>). Even Microsoft has yielded to FP by introducing FP concepts in C# with LINQ (Language Integrated Query). Do you feel FP has a future in software development, or are we stuck with our current model of languages with increasing features?

Fully Functional

Dear Functional,

I have not read the book you mention but I am familiar with functional programming, its advantages, and its shortcomings. Do I think that FP is the future of software development? In a word, no.

What you propose here is the classic silver bullet. If only we all programmed in, well, take your pick: Lisp, Basic, Fortran, Assembler, PHP, Cobol, Prolog, Perl, Python, C, C++, C#, Java, or whatever the flavor of the week is, then the world would be a wonderful shiny place, just like Tomorrowland, only with better food.

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!

Dear KV,

I got into a funny argument at work on a distributed systems project. One of the people on my project is insisting that all the data we send in our new, lightweight update system be encoded as type/length/value instead of just type/value. Since we have a type, I don't see why length matters. We can always infer one from the other.

The protocol needs to be as light as possible because it may be transported over SMS to cellphones, as well as going over a more robust network such as the Internet.

Do you think that the length must really be included?

Wondering about the Value of Length

Dear Wondering,

It's funny how so many funny arguments have unfunny consequences. Actually, the funniest thing is that you refer to the Internet as "robust," but I think I'll just leave that aside for now.

Network protocols and protocol design are near and dear to KV's heart, and, yes, KV does have a heart. Instead of talking only about the importance of length, I'm going to broaden the scope of my answer and provide you with KV's top five network protocol design recommendations.

1. Always encode a length. Since you asked about type/value vs. type/length/value encoding, I'll start there. Leaving out the length in a packet or other type of encoding is a recipe for buffer overflow disaster. If the program reading the packet or data from the wire can't figure out, in an easy way, how much data to read, this will invariably lead to someone writing code wherein he or she tries to guess just how much to read—and having to guess is bad, very bad. A poorly chosen guess usually leads to a buffer overflow, and since we're discussing a network protocol, that buffer overflow can be remotely exploited. Congratulations, your new protocol is now the Internet's new whipping boy, as script kiddo after script kiddo is able to write trivial security exploits that destroy what you built.

2. Protocols need version numbers. People who think, "Oh this is a one-off," need to be offed, or perhaps politely removed from the project. There is rarely any such thing as a one-off in software. You need to design and build protocols, just like software, so that they can be updated. Without a version number encoded somewhere in the protocol, it is impossible to upgrade it without guessing—and guessing, again, is bad.

3. Align your fields. Back in the old days, when, as an older friend of mine says, "Dinosaurs roamed the earth,"

Leaving out
the length in a packet
is a recipe for buffer
overflow disaster.

it was very important to squeeze every last bit that you possibly could into a packet; therefore, protocols were often designed without data alignment in mind. A program receiving a packet would have to read an entire 8-, 16-, or 32-bit entity and then play bit-twiddling games with the retrieved value to

get at the one or two bits it really cared about. Although the current trend in the opposite direction—wherein people design protocols that seem intentionally to waste bandwidth—is a problem, a balance needs to be struck, and that balance ought to be struck in favor of byte-aligned fields. The alignment issue most often comes up when protocol designers decide to add flags to their packets, so that brings us to...

4. Group your flags. Almost all network protocols that I know have some number of flags (i.e., single bits of information) that they need to communicate. Group the flags together so that they can more easily be byte-aligned (see item 3).

5. Leave room to grow. Designing a protocol to be used once is a recipe for more work—work that you wouldn't have to do if you left just a little bit of room for extensibility when you came up with the protocol in the first place. A few extra flag bits and the ability to have packet extensions at a later time are two things every protocol designer should be thinking about before releasing version 1.0. This last point is not an excuse to waste space; it is a plea to think about what happens in the future, which may be only as far away as next week.

Obviously the list could be longer, and really could be the subject of an entire book, but for now those are my top five. If you can get those right, then perhaps your protocol will become wildly popular and everyone will ask you to "do it again!" Won't that be fun?

KV

CODE 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.

© 2007 ACM 1542-7730/07/0900 \$5.00