Kode Vicious Unleashed

Koding konundrums driving you nuts? Ko-workers making you crazy? Not to worry, Kode Vicious has you covered—answering your questions, solving your problems, and just making the world a better place.

Dear KV,

My officemate writes methods that are 1,000 lines long and claims they are easier to understand than if they were broken down into a smaller set of methods. How can we convince him his code is a maintenance nightmare?

Fond-of-Abstractions

Dear FoA,

The short answer to your question is to make your officemate maintain his own code forever, as that should be punishment enough. Unfortunately, people with annoying habits, such as talking loudly on cellphones, driving poorly, or giving unwanted advice, rarely see the errors of their ways. This is why there must always be crusaders for the good and right, such as ourselves.

I note that you used the word *methods*, rather than *functions*, which indicates to me that you are using an object-oriented language. Before we go further, let me point out that everything I say here applies both to methods in object-oriented languages and functions in non-object-oriented languages. The base problem is that there is too much functionality crammed into one place. Several arguments can be made to explain why such overly long methods are problematic.

The first argument that comes to mind is code reuse. One of the reasons to have methods or functions in a program is to capture the essence of a single idea or algorithm so that others can easily reuse it. When a method grows to 1,000 lines, it usually becomes highly specialized to one job, and a job that is probably not needed that often. It is far better to break down the larger problem

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.

into smaller ones, some of which may be reused by other parts of the software. Another advantage of

smaller, reusable methods is that you can use them in the next project you work on. Reusable methods are a benefit to the koders, who now do not have to write as much code, and to the company the koders work for because they can now finish a project faster. This kind of work avoidance is one of my favorite reasons for anything I do.

Another argument is that overly long methods are just plain hard to read and understand. A method that is 1,000 lines long, when viewed in a window that displays 50 lines at a time, works out to 20 pages of code to work through. Now, I don't know about anyone else, but 20 pages of anything—a book, magazine, or code—is hard to digest and keep all in my brain at any one time. Understanding anything requires context, and that context ought to be local. Jumping from page 18 back to page 2 because that's where the variable *fibble* was last modified often causes me to lose my place. I wind up staring at page 2 thinking, "Why was I here?" I get all glassy eyed, stare into space, and occasionally begin to drool, which makes my co-workers very nervous.

Finally, we come to your excellent point about code maintenance. Clearly, if something is hard to understand, as we just established in the last paragraph, then it will also be hard to maintain. A 1,000-line method is doing too many things at once. How do you find the bug in a method when it is doing the equivalent of balancing your checkbook, whistling the *Ode to Joy*, and juggling chain saws, all at once? Compounding the problem is that the number of possible side effects in a piece of code goes up with every line you add—perhaps not exponentially, but certainly more than linearly.

There are a few ways to set your officemate on the righteous path to klean kode, if not clean living. One way is to present this set of arguments and see how your officemate responds. Using a neutral third party's code as an example is a good way to avoid the, "I'm a better programmer than you" pissing match, which rarely wins anyone over to your side. If rational argument fails, you can try to use the software specification as a way to

get reasonably sized chunks of functionality out of this person. You do have a specification for your software, right? If it clearly states the amount of work that is to be done for each method, then it will be pretty clear when your officemate violates the spec, and you can then come down on him as hard as you like.

Of course, sometimes reasonable argumentation (i.e., the carrot) and direct control (i.e., the stick) fail. At that point I recommend something lingering, with boiling oil or melted lead. Just don't tell Amnesty International—though if they had to maintain your officemate's code, they might understand.

KV

Dear Kode Vicious,

Regarding your coding peeve 2, code dingleberries ("Kode Vicious Strikes Again," November 2004), I agree it's better to clean out unused code. I often comment out areas replaced by revised code because I might want to remember how I did something later. Putting it in a source-code control system is a good idea, but how can you remember which version it was in, years later?

Short-Term Memory

Dear STM,

I'm not sure what source-code control system you're using, but all the ones I've used in the past 10 years have the ability to store a comment with each check-in. This check-in comment, once stored, can also be retrieved. When I am looking at a piece of code that has faded from my memory—for example, one that was checked in on a Friday before a particularly good party—I will get the entire history of that file and scan it for telltale signs of what I or others were thinking when various changes were made. I'm sure, STM, that you put in proper comments when you check in your code, but I'm also sure that there are plenty of readers who could benefit from a short rant on another of my pet peeves.

Lame check-in comments are a pet peeve that didn't make my top five, but they are certainly in my top 10. Many years ago I actually worked with an engineer who refused to put in any comments when he checked in his code. It was completely in keeping with his manner, as he also rarely spoke more than 10 words in a day at work, and answered all questions with a "yes" or a "no." Eventually the team installed a trigger that required at least three words to be put into any comment before a file could be checked in. If the comment was empty, then the check-in was rejected. To be honest, this didn't help us very much, as we then got comments such as: **Added new feature** or **Fixed bug 511**.

Although I considered applying thumbscrews to this particular engineer, I was told that that was not a modern business practice, at least in the United States. After a couple of years of this madness, the engineer left the company. Although he was a pretty good koder, his software had bugs, just like any of us have in our software. The major difference was that fixing anything that this guy had worked on was a nightmare because none of the code had any context. There were incredibly lame comments in the code, an already addressed peeve, and the check-in trail told you nothing about the code's evolution.

Check-in comments come under the same restrictions as comments in code. In the first place, there should actually be a comment. As far as I'm concerned, all source-code control systems should force an engineer to include a comment in a check-in. If it were possible to deliver an electric shock to the keyboard of those who check in code without comments, I would install that, too. Secondly, the comments have to be useful. The minimum three-word stricture placed on the engineer I worked with clearly was insufficient. I figure if you fix a trivial bug, then you should include a full sentence about what the bug number was so it can be looked up later, what the problem turned out to be, and how you fixed it. Brevity may be the soul of wit, but unless you are Oscar Wilde (and he's dead, so you're clearly not), you will need at least three full sentences to meet my criteria. Of course, when you add something more significant, such as a new feature, then you need an entire paragraph to describe what the feature is, how it works, and how it is used.

I'm sure that I will get complaints that there are better places to store such information. The bug information should be in the bug database, and the feature information should be in the specification. True, but when I open a piece of code I want the greatest amount of information available to me in one place. I do not want to have to switch between the bug database window, the electronic version of the spec, and my code. So, please, next time you check in something to your source-code control system, pretend that I'm standing behind you with a cattle prod, and carefully consider what you're about to write.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He is an avid bicyclist and traveler who has made San Francisco his home since 1990.

© 2005 ACM 1542-7730/05/0200