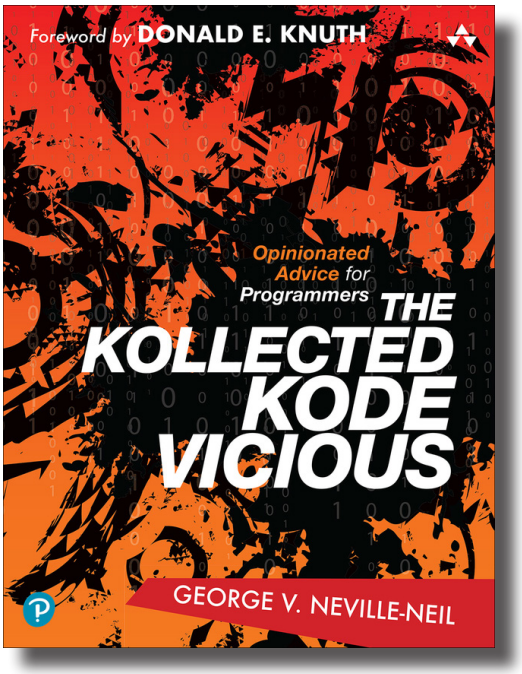
Pragmatic, Bite-Sized Programming Advice from Koder-with-Attitude, Kode Vicious



ORDER & SAVE

Save 35% When You Order

from **informit.com/infoq/kode**
and enter the code **KODEVICIOUS**
during checkout

FREE US SHIPPING on print books

Major eBook Formats
Only InformIT offers PDF, EPUB, & MOBI together for one price

OTHER AVAILABILITY

Through O'Reilly Online Learning **(Safari)** subscription service

Booksellers and online retailers including Amazon/Kindle store and Barnes & Noble | bn.com

"For many years I have been a fan of the regular columns by Kode Vicious in Communications of the ACM. The topics are not only timely, theyre explained with wit and elegance.
—From the Foreword by Donald E. Knuth

Writing as Kode Vicious (KV), George V. Neville-Neil has spent more than 15 years sharing incisive advice and fierce insights for everyone who codes, works with code, or works with coders. Now, in The Kollected Kode Vicious, he's brought together his best essays and Socratic dialogues on the topic of building more effective computer systems. These columns have been among the most popular items published in ACMs Queue magazine, as well as Communications of the ACM, and KVs entertaining and perceptive explorations are supplemented here with new material that illuminates broader themes and addresses issues relevant to every software professional.

Neville-Neil cuts to the heart of the matter and offers practical takeaways for newcomers and veterans alike on the following topics:

- The Kode at Hand: What to do (or not to do) with a specific piece of code
- Koding Konundrums: Issues that surround code, such as testing and documentation
- Systems Design: Overall systems design topics, from abstraction and threads to security
- Machine to Machine: Distributed systems and computer networking
- Human to Human: Dealing with developers, managers, and other people

Each chapter brings together letters, responses, and advice that apply directly to day-to-day problems faced by those who work in or with computing systems. While the answers to the questions posed are always written with an eye towards humor, the advice given is deadly serious.

Pearson Addison-Wesley

**Discount code KODEVICIOUS confers a 35% discount off the list price on informit.com only. Discount may not be combined with any other offer, including the book + eBook "Best Value" bundle, and is subject to change.*

informIT® the trusted technology learning source

Foreword by Donald E. Knuth (*DK*)

Dear DK,

My job keeps me too busy to read real books about computer science. And I don't seem have a great attention span. But I know that everything about computers keeps changing rapidly, and I'm afraid that I'll soon be obsolete if I don't keep up with the field.

Can you suggest any reliable source of current information, by which I might painlessly improve the quality of my work?

Harried Information Hider

Dear Harried,

For many years DK has been a fan of the regular columns by Kode Vicious in Communications of the ACM. The topics are not only timely, they're explained with wit and elegance. KV is not afraid to take unpopular views, and he savagely dissects lots of the insanity that tends to be spreading around.

So DK thinks you ought to try it out. In fact, there's even a better way now, because KV has gathered his columns together and extended them into a book. That book may be just what you crave.

About attention span, on the other hand, that's a tougher problem—especially for people of your generation. Consider spending some time on a desert island, with no access to the internet. Just go somewhere where the weather and accommodations are nice. Take a good technical book with you and lots of scratch paper and pencils and erasers.

A pedagogical book that's full of exercises with worked answers would be especially useful. In fact, if you happen to choose one of DK's own books, you might even find that it contains a quotation by KV himself.

Of course you should also take a copy of KV's book, to keep you grounded.

DK

Dear DK,

Somebody told me that you're a regular reader of Kode Vicious's column, which features answers to letters that he supposedly receives.

When I look closely at those letters, however, it seems to me that they are too perfect. Nobody ever sends me letters that are so well written and to the point.

Do you believe KV forges those letters, or are they actually real?

Skeptical Inquirer

Dear Skeptical,

Indeed, that's exactly the question that DK asked KV, when meeting him in person at the Hackers Conference some years ago. And KV shamelessly admitted to ghost-writing.

But if you think about it, you'll probably agree that the question-and-answer format is an ideal way to express ideas and to teach others. DK even bets that Plato himself ghost-wrote the "dialogues" that Socrates supposedly once had.

Guess what: That format is so effective, DK is now tempted to try it himself.

DK

Preface

What's the worst that could happen?

Famous last words

Welcome to an endeavor I never thought to undertake, the first book of Kode Vicious. In fact, I never thought I'd write a column for a magazine or that that column would run for more than 15 years and more than 100 articles, but life is full of strange twists and turns, especially when you don't duck quickly enough when a table full of your peers is looking for a victim...I mean volunteer!

"So now I'd like to throw out the worst idea of all time." With these words, from Wendy A. Kellogg, the idea that was to become Kode Vicious was born. "It should be someone from the board. Someone with an attitude problem. Someone bald." Back in the early days of *Queue*, I was the only bald board member, though at that point I'd already been shaving my head for a decade.

In February 2004 I was, along with the rest of the *Queue* editorial board, attending our monthly meeting where we get together and try to come up with interesting topics, and authors for *Queue*. It was the early days of the magazine, then in its fourth year, and though we had had several successful issues, we had no regular columnists. I had been invited to the board meetings by Eric Allman, and then written a couple of pieces for the magazine, and was working on co-authoring my first book, but I had never been a columnist, and although the idea seemed fun at the time, perhaps due to too much wine at dinner, I was at a loss as to how to make it actually work.

The original idea for KV, as he came to be known, was actually for a more *Miss Manners* style of column, based on the famous work of Judith Martin, who I had read with my mom when I was a kid. I would write the pieces, *in drag* as it were, and this seemed like an interesting challenge. The first name for KV was *Mother Code,* and I submitted two pieces to our editors based on this persona.

A bit of the character sketch from our meeting might give a better idea of where this was going at the time: Although Mother is never harsh in her advice or criticism, she is also firm in her beliefs. The image is of a strong, but flexible and kind, advice giver. She also has a signature line on every piece, something like "Don't forget to wipe your shoes" or "Remember to wear your galoshes" but that is related to our audience. Something like, "And remember, make sure your code builds before you check it in to the source tree."

In the end this all turned out to be unworkable for a couple of reasons. The most important reason why the original pieces didn't work is that it's very hard to write as someone you're not. Although one or two pieces might have been possible in a very different guise, it's far easier to

write as someone closer to your own persona than it is to write as someone completely different. Let's face it, Miss Manners I ain't.

I actually spent quite a while trying to come up with the persona I would use, including some obvious ones like "Code Confidential" and "Code Critic" as well as the embarrassing "Captain Safety," "Bug Basher," and "Lint Picker" before hitting on the word Vicious as a good one to use as a *nom de plume*. From there it was a quick romp through "Kid Vicious," "Code Vicious," and "Vicious Kode" to finally getting something that sounded right, "Kode Vicious."

With the new name came a new character sketch:

> @$$hole with a heart of gold. Always willing to teach, but is unwilling to teach those who are not willing to learn. Think Zen monk in a Sex Pistols T-shirt who you worry about bringing to dinner. Often uses nose tweaks to show the student the way or at least *a* way.

I was off and running. I rewrote the original *Mother Code* piece, "So Many Standards," about picking coding standards, and began my career as a columnist.

So, is the author behind Kode Vicious really a big loud jerk who throws co-workers out windows, flattens the tires of the annoying marketing guy, drinks heavily, and beats and berates his co-workers? The answer is both yes and no.

KV is a caricature, and people who know me and have worked with me can easily see how I can write the pieces that I do. Of course, KV is someone I might want to be, or turn into, from time to time, a Hyde to my Dr. Jekyll. Usually I want to be KV when I'm in one of *those* meetings where I take off my glasses, drop them loudly on the table, and run my hand over my bald head, thinking, "How can anyone be so stupid?!" If you are ever in a meeting with me and I do this, it's a tell: Whoever just spoke is a moron. The fact is that beating, or berating, stupid people never makes them any smarter, so instead I turn those thoughts into articles for KV, who can rant about things without winding up in jail and hopefully do a small amount of good at the same time.

It's odd to think about literary influences for KV, but as with any writer, I have several, not the least of whom was my mother, whom I wrote about in "Standards Advice"[1] and who was a hard@$$ and a harsh critic. My favorite authors have always been harsh, direct, and looking to mess with people, and if I'm honest, a lot of KV comes from wishing I was Hunter S. Thompson in the three truly great books he wrote: *Hell's Angels*, *Fear and Loathing in Las Vegas,* and *Fear and Loathing on the Campaign Trail '72*. The more salacious and surreal moments come from reading William S. Burroughs, and if you think it's *Naked Lunch*, I laugh, because that's a nice children's bedtime story compared to *The Wild Boys*.

Another direct influence has been *Queue* itself. Talking to our board and our guest experts for 15 years, the people who come to *Queue* meetings and help us frame our issues, reading and reviewing articles for the magazine, has been one of the most amazing learning experiences of my career. I have been lucky enough to have some truly amazing minds, slightly besotted by wine, violently pointing steak knives across the table at me and telling me just why some idea was either interesting or total bull$#!+.

1. https://queue.acm.org/detail.cfm?id=1687192

I have been asked on several occasions, including in one letter, whether I write both the questions and the answers. When I was originally writing the pieces, there were no letters, so I had to write the questions as well as the answers. At first this was quite difficult. I would be staring at the screen, hours away from deadline (I always submit my pieces on or just after the deadline), and I would have nothing besides "Dear KV" in my editor buffer. I then learned a good trick, which I have used ever since. If I run out of material, all I have to do is open up a piece of source code and read it. If the code is well-written, then I can write about what is good about the code, and if, as is often the case, the code is not well written, then I just have to wait for my blood to reach boiling and off I go.

In fact, I use both submitted letters and my own ideas in the column. Whenever something truly stupid happens in a piece of code, during a project, or on the news, I make note of it, if I think I can turn it into an article.

I also get e-mail that is put directly into the question section of the column, and ACM then rewards the letter writer with a *Queue* trinket, which I am sure they will cherish to their dying day. Which pieces are which? I'm not telling, and my editor doesn't know which is which either, so don't try bribing him at the next Turing Dinner.

Of course, KV would not be who he was without his editor, Jim Maurer, who has, for the past 15 years, read KV and then turned these insane ravings into something that ACM would not only publish, but publish in two magazines, *Queue* and *Communications of the ACM* (CACM). From time to time I read a really great line in the finished KV and I'm always intrigued, did I write that or did Jim? I will say that Jim has turned a lot of average bits of writing into something that is actually intelligent and enjoyable, and for this he has my deepest thanks.

I had been planning this book for a while, but it's hard to maintain the level of energy it takes to write as KV for more than about 1,500 words at a time, at least without substances that are only available by prescription, or, since I live in Brooklyn, just down the block. The book finally got written because of the very kinds words of the author of the preface, Don Knuth, who I had asked at one point to write a foreword and who surprised me at a conference with the words, "I've written a wonderful foreword for your book," at which point I knew that I had to finish this work, or the crushing guilt would kill me.

I've kept writing the columns in the meantime, of course, because my writing comes from anger, and anger is what I'm good at. Anger also leads to the dark side, and the dark side has cookies. The book, well, it clearly got done, the question is how long the rehab will take.

George Neville-Neil
aka KV
Brooklyn, NY
June 30, 2020

2

The time has come
The Walrus said
To speak of many things

"The Walrus and the Carpenter,"
Lewis Carroll

Koding Konundrums

Stepping away from the kode in front of us, we come to a slightly broader set of koding koncepts. Many people fail to understand that programming and software design aren't just about typing hundreds of lines into an editor or IDE and then pressing Run. There are concepts we must address no matter how large or small the system is that we're working with. As any system is built there are the problems of debugging, documenting, and testing the system as well as understanding challenges to the overall system performance, and these are some of the problems we turn to in this chapter.

2.1 Ode to the Method

The good thing about science is that it's true whether or not you believe in it.

Neil deGrasse Tyson

If computer science is truly a science, then clearly the scientific method can be applied to solve problems that present themselves in computers and their accompanying software. A topic rarely covered in computer science programs is how one might actually apply the scientific method to software engineering. If one is not solving a problem via something akin to the scientific method, then can they be said to be debugging by faith? Faith-based debugging has, thankfully, not taken off in the way agile and scrum have, but it is definitely a method I have seen applied by many people who really ought to know better. Treating software bugs as near supernatural occurrences is a frequent joke among koders, "Did you sacrifice a chicken?" being a common retort to someone saying that they simply cannot find a bug or get the printer to work. While there are koders who are so good that they can find the bug just by scanning a page of your kode, these are few and far between, and so the best programmers learn to apply what, in essence, is the scientific method to solving their problems. In the following response I lay out a very simple way to apply the method to fixing software bugs and being confident in the fix.

Dear KV,

I just started working for a new project lead who has an extremely annoying habit. Whenever I fix a bug and check in the fix to our code repo, she asks, "How do you know this is fixed?" or something like that, questioning every change I make to the system. It's as if she doesn't trust me to do my job. I always update our tests when I fix a bug, and that should be enough, don't you think? What does she want, a formal proof of correctness?

I Know Because I Know

Dear I Know,

Working on software is more than just knowing in your gut that the code is correct. In actuality, no part of working on software should be based on gut feelings, because, after all, software is supposed be a part of computer science, and science demands proof.

One of the problems I have with the current crop of bug-tracking systems (and trust me, this is only one of the problems I have with them) is that they don't do a good job of tracking the work you've done to fix a bug. Most bug trackers have many states a bug can go through: new, open, analyzed, fixed, resolved, closed, etc., but that's only part of the story of fixing a bug, or doing anything else with a program of any size.

A program is an expression of some sort of system that you, or a team, are implementing by writing it down as code. Because it's a system, you have to have some way of reasoning about that system. Many people will now leap up and yell, "Type systems!" and "Proofs!" and other things about which most working programmers have no idea and which they are not likely ever to come into contact with. There is, however, a simpler way of approaching this problem that does not depend on a fancy or esoteric programming language: use the scientific method.

When you approach a problem, you ought to do it in a way that mirrors the scientific method. You probably have an idea of what the problem is. Write that down as your theory. A theory explains some observable facts about the system. Based on your theory, you develop one or more hypotheses about the problem. A hypothesis is a testable idea for solving the problem. The nice thing about a hypothesis is that it is either true or false, which works well with our Boolean programmer brains: either/or, black or white, true or false, no "fifty shades of gray."

The key here is to write all of this down. When I was young, I never wrote things down because I thought I could keep them all in my head. But that was nonsense; I couldn't keep them all in my head, and I didn't know the ones I'd forgotten until my boss at the time asked me a question I couldn't answer. Few things suck as much as knowing that you've got a dumb look on your face in response to a question about something you're working on.

Eventually I developed a system of note taking that allowed me to make this a bit easier. When I have a theory about a problem, I create a note titled THEORY, and write down my idea. Under this, I write up all my tests (which I call TEST because, like any good programmer, I don't want to keep typing HYPOTHESIS). The note-taking system I currently use is Org mode in Emacs, which lets you create sequences that can be tied to hot keys, allowing you to change labels quickly. For bugs, I have labels called BUG, ANALYZED, PATCHED, —, and FIXED, while for hypotheses I have either PROVEN or DISPROVEN.

I always keep both the proven and disproven hypotheses. Why do I keep both? Because that way I know what I tried and what worked and what failed. This proves to be invaluable when you have a boss with OCD, or, as they like to be called, "detail oriented." By keeping both your successes and failures, you can always go back, say in three months when the code breaks in a disturbingly similar way to the bug you closed, and look at what you tested last time. Maybe one of those hypotheses will prove to be useful, or maybe they'll just remind you of the dumb things you tried, so you don't waste time trying them again. Whatever the case, you should store them, backed up, in some version-controlled way. Mine are in my personal source-code repo. You have your own repo, right? Right?!

KV

2.2 How Much + in C++?

My name is Ozymandias, King of Kings
Look on my Works, ye Mighty, and despair!

"Ozymandias,"
Percy Bysshe Shelley

The language wars are never ending, and in the following letter KV was perhaps baited into bashing one such language. To be honest I've always hated C++; I find it bloated and that it mostly produces intractable mounds of horse$#!+, as my grandmother used to say, which are hard to maintain and hard to tune. My response here is probably a bit tamer than my true feelings on C++, in part because I know that people will choose what they choose and also that we're all dealing with a mound of technical debt that we often didn't start with but that was thrust upon us on day one at work. That there are schools that still think that C++ is a good language to start teaching computer science with is, quite frankly, maddening, and I wish they would stop. I consider that particular practice to be mental health abuse. It would be better to give the students Python than C++, or, perhaps, an assembler.

In the years since this letter and response were first published we find that there are at least two new contenders in the compiled language space, Rust and Go. KV has only a passing acquaintance with both of these but is excited to see Rust trying to push into the embedded space, in particular. A language with better memory safety in systems that often lack virtual memory protections seems like it ought to be a good thing for every koder's mental health. The other upside with both Rust and Go is that they look somewhat familiar to those of us who already program in the Algol-like languages, so a jump from, say, C, C++, or Java to either of these languages isn't going to cause so much cognitive dissonance that you feel as if you'd put a small piece of blotter paper under your tongue before starting to work. As for either of these new languages as a teaching language, the jury is very definitely still out, and I still prefer we teach students with something interactive to start, such as Python.

For now let's see just how much + KV was able to find in C++.

Hello KV,

Since there was some debate in my company over this issue, I'm curious to see what do you believe: putting aside performance issues (which I think are relatively minor on modern PCs), when would you recommend using C++ for development, and when would you recommend C? Do you think it is always better to use C++?

My feeling is that unless your application is inherently object-oriented (e.g., user interfaces), C++ will tend to make the implementation worse instead of making it better (e.g., constructors and operators doing funny unexpected things, C++ experts trying to "use their expertise" and writing C++ code that is very efficient but extremely hard to read and even not portable, huge portability and performance issues when using templates, incomprehensible compiler/linker error messages, etc., etc.). I also think that while people can write bad C code (gotos out of macros was a nice one), typically people can write *awful* C++ code.

So, what do you think, where do you stand on this dispute?

Wondering How Much + There Is in ++

Dear Wondering,

Picking a language is something I've addressed before in other letters, but the C vs. C++ debate has raged as long as the two languages have been in existence, and really, it's getting a bit tiring. I mean, we all know that assembler is the language that all red-blooded programmers write in! Oh, no wait, that's not it.

I'm glad you ask this question, though, because it gives me license to rant about it and also to dispel a few myths.

The first and most obvious myth in your letter is that user interfaces are inherently object-oriented. While many introductory textbooks on object-oriented programming have user interfaces as their examples, this has a lot more to do with the fact that humans like pretty pictures. It is far easier to make a point graphically than with text. I have worked on object-oriented device drivers, which are about as far as you'll ever get from a user interface.

Another myth that your letter could promulgate is that C is not an object-oriented language. A good example of object-oriented software in C is the vnode filesystem interface in BSD Unix and other operating systems. So, if you want to write a piece of object-oriented software, you can certainly do it in C or C++, or assembler for that matter.

One final myth, which was actually dispelled by Donn Seely in "How Not To Write FORTRAN in any Language" (ACM Queue vol. 2, no. 9 - Dec/Jan 2004–2005), is that C++ leads to less understandable code than C. Over the past 20 years I have seen C code that was spaghetti and C++ code that was a joy to work on, and vice versa.

So, after all that myth bashing, what are we left with? Well, the things that are truly important in picking a language are:

1) What language is the largest percentage of the team experienced in?

   If you're working with a team and six out of eight of them are well-versed in C but only two know C++, then you're putting your project, and job, at risk in picking C++. Perhaps the two C++ koders can teach the C folks enough C++ to be effective, but it's unlikely. In order to estimate the amount of work necessary for a task you have to understand your tools. If you don't normally use a nail gun, then you're likely to take someone's toe off with it. Losing toes is bad as you need them for balance.

2) Does the application require any of the features of the language you're using?

   C and C++ are a lot alike as languages, i.e., in syntax, but they have different libraries of functions and different ways of working that may or may not be relevant to your application. Often real-time constraints require the use of C because of the control that can be gotten over the data types. If type safety is of paramount importance, then C++ is a better choice because that is a native part of the language that is not present in C.

3) Does the application require services from other applications or libraries that are hard to use or debug from one or the other language?

   Creating shim layers between your code and the libraries you depend on is just another way of adding useless, and probably buggy, code to your system. Shim layers should be avoided like in-laws. They're OK to talk about, and you might consider keeping them around for a week, but after that, out they go as so much excess, noisy baggage.

There are lots of other reasons to choose one language over another, but I suspect that the three listed should be enough for you and your team to come to some agreement, and you'll notice that none of them had to do with how easy it is to understand templates or how hard it is to debug with exceptions.

KV

2.3 Something Sleek and Modern

Move fast and break things.

Idiots' Mantra

Many letters to KV run along this vein of wanting to use a new language or technology to kode. As I point out here, what matters isn't newness but appropriateness to the job, and the use of good software engineering practices. Most of us in the koding business have a thing for the sleek, new, and modern. Many in our industry would probably be quite comfortable with the Futurist Manifesto, except for its unhealthy relationship to fascism. Written in 1909 by Filippo Tommaso Marinetti, nearly the entire manifesto could be twisted into the language of modern tech companies, "Move fast and break things," but honestly the manifesto was twisted enough to begin with.

As technologists many of us are attracted to new ideas, methods, and ways of working, especially if they promise to make our systems faster or able to do more with less. The fact is that being sleek and modern is no panacea for the ills of software. Good software requires careful thought, deliberate planning, and careful execution, as I point out here, and there has yet to be a silver bullet to the problems of developing good software.

Dear KV,

When I read your column, you sound to me like one of these guys who only kodes, as you would misspell it, in C or maybe C++. Many of us code in other languages such as PHP, Python, and Perl. How about writing about languages like those? Perhaps something written and designed after most of your readers born.

Where I work we provide a lot of web services, so we use a good deal of PHP in our work, with only a small amount of C and C++ doing the computationally intensive tasks, or things that have to be closer to the operating system. Do you have any advice for those of us who kode in other languages?

21st Century Kodern

Dear 21,

I'll have you know that I was born after C, but only just. I have no doubt that many of the readers of Queue cut their teeth on other languages, perhaps, though I shudder to think it BASIC, but I can only write about what people ask about. There have been letters from koders using languages other than C or C++, such as Linguistically Lost, who wrote to me a few months ago. If you had asked a concrete question, this would have made my job easier, but clearly that was not your goal. As to my misspellings, take it up with my editor, though I recommend you take a couple of self-defense classes first.

I have read plenty of PHP code in my time as well as Python, Perl, C, C++, TCL, Fortran, Lisp, COBOL, and others. The basic fact is that what separates good code from bad code has very little to do with the language itself. As someone recently pointed out in Queue, you should learn "How Not to Write FORTRAN in Any Language."

Good code is code that uses the dominant metaphors in a language in order to make it easy for other people to understand. All the languages I've mentioned in this reply have comments, and yet many people seem to either leave these out or misuse them completely. Since the 1980s it has been possible to write understandable variable and function names, and yet people still continue to use single letters, believing that those who look at the code after them will know what they mean.

In PHP you can write this:

```
function getn($data)
```

as easily as this:

```
// This function takes a string as input in the name_field
// The name must be a string starting with an alphabetic
// character, i.e. A..Z or a..z, and may not be more than 32
// characters in length. Only the first character must be
// alphabetic and all the following characters can be
// alphabetic or numeric i.e. 0..9
function get_name($name_field)
```

and yet people continue to write the first version. So, I don't care how modern you and your young friends are; if you apply the basics of writing code that is easy to understand the first time, some other poor koder has to read it. Once you're done with that, get back to me with some specific questions on PHP that will illuminate its sleek, modern feel, and we'll have something to talk about.

KV

2.4 What's in a Cache Miss?

People who are really serious about software should make their own hardware.

Alan Kay

Many people who develop software would like to pretend that hardware doesn't exist at all and that all their creations run on a perfectly imagined concept of a computer, but of course this is not actually the case. Software runs on hardware, and hardware is made up of bits that have to obey real-world constraints, such as the speed of light and entropy and other messy things that can interfere with our concepts of software.

While it might not be necessary to understand hardware at the level of the electrical engineers who build our chips, it is necessary at some level to understand how hardware affects the performance of software. No single change in computer architecture has had as profound an effect on overall system performance as that of introducing multiple levels of caching into CPUs. Most software performance is still predicated on a very old model of how CPUs execute software, but that model is long gone, except in perhaps the cheapest of low-end processors. The following letter and response tries to catch us up with the state of the art in which many of our performance problems lie hidden.

Dear KV,

I've been reading some pull requests from a developer who has recently been working in code that I also have to look at from time to time. The code he has been submitting is full of strange changes that he claims are optimizations. Instead of simply returning a value such as 1, 0, or -1 for error conditions, he allocates a variable and then increments or decrements it, and then jumps to the return statement. I haven't bothered to check whether or not this would save instructions, because I know from benchmarking the code that those instructions are not where the majority of the function spends its time. He has argued that any instruction we don't execute saves us time, and my point is that his code is confusing and hard to read. If he could show a 5 or 10 percent increase in speed, it might be worth considering, but he has not been able to show that in any type of test. I've blocked several of his commits, but I would prefer to have a usable argument against this type of optimization.

Pull the Other One

Dear Pull,

Saving instructions, how very 1990s of him. It's always nice when people pay attention to details, but sometimes they simply don't pay attention to the right ones. While KV would never encourage developers to waste instructions, given the state of modern software, it does seem like someone already has. KV would, as you did, come out on the side of legibility over the saving of a few instructions.

It seems that no matter what advances are made in languages and compilers, there are always programmers who think they're smarter than their tools, and sometimes they're right about that, but mostly they're not. Reading the output of the assembler and counting the instructions may be satisfying for some, but there had better be a lot more proof than that to justify obfuscating code. I can only imagine a module full of code that looks like this:

```
if (some condition) retval++; goto out: else retval–; goto out: ... out:
return(retval)
```

and, honestly, I don't really want to. Modern compilers, or even not so modern ones, play all the tricks that programmers used to have to play by hand: inlining, loop unrolling, and many others, and yet there are still some programmers who insist on fighting their own tools.

When the choice is between code clarity and minor optimizations, clarity must nearly always win. A lack of clarity is the source of bugs, and it's no good having code that's fast and wrong. First the code must be right, and then the code must perform; that is the priority that any sane programmer must obey. Insane programmers, well, they're best to be avoided. Eventually they wind up moving to a Central American nation, mixing their own drugs in bathtubs, and claiming they can unlock iPhones.

The other significant problem with the suggested code is that it violates a common coding idiom. All languages, including computer languages, have idioms, as pointed out at length in *The Practice of Programming* by Brian W. Kernighan and Rob Pike (Addison-Wesley Professional, 1999), which I recommended to readers more than a decade ago. Let's not think about the fact that that book is still relevant, and that I've been repeating myself every decade. No matter what you think of a computer language, you ought to respect its idioms for the same reason that one has to know idioms in a human language; they facilitate communication, which is the true purpose of all languages, programming or otherwise. A language idiom grows organically from the use of a language. Most C programmers, though not all, of course, will write an infinite loop in this way:

```
for (;;)
```

or as

```
while (1)
```

with an appropriate `break` statement somewhere inside to handle exiting the loop when there is an error. In fact, checking the *Practice of Programming* book, I find that this is mentioned early on (in section 1.3). For the return case, you mention it is common to return using a value such as 1, 0, or -1 unless the return encodes more than true, false, or error. Allocating a stack variable and incrementing or decrementing and adding a goto is not an idiom I've ever seen in code, anywhere, and now that you're on the case, I hope that I never have to.

Moving from this concrete bit of code to the abstract question of when it makes sense to allow some forms of code trickery into the mix really depends on several factors, but mostly on how much speedup can be derived from twisting the code a bit to match the underlying machine a bit more closely. After all, most of the hand optimizations you see in low-level code, in particular C and its bloated cousin C++, exist because the compiler cannot recognize a good way to map what the programmer wants to do onto the way the underlying machine actually works. Leaving aside the fact that most software engineers really don't know how a computer works, and leaving aside that what most of them were taught if they were taught about computers, hails from the 1970s and 1980s before superscalar processors and deep pipelines were a standard feature of CPUs, it is still possible to find ways to speed up by playing tricks on the compiler.

The tricks themselves aren't that important to this conversation; what's important is knowing how to measure their effects on the software. This is a difficult and complicated task. It turns out that simply counting instructions as your co-worker has done doesn't tell you very much about the runtime of the underlying code. In a modern CPU the most precious resource is no longer instructions, except in a very small number of compute-bound workloads. Modern systems don't choke on instructions; they drown in data. The cache effects of processing data far outweigh the overhead of an extra instruction or two,

or ten. A single cache miss is a 32-nanosecond penalty, or about 100 cycles on a 3-GHz processor. A simple MOV instruction, which puts a single, constant number into a CPU's register, takes one-quarter of a cycle, according to Agner Fog at the Technical University of Denmark.

```
http://www.agner.org/optimize/instruction_tables.pdf
```

That someone has gone so far as to document this for quite a large number of processors is staggering, and those interested in the performance of their optimizations might well lose themselves in that site generally.

```
http://www.agner.org
```

The point of the matter is that a single cache miss is more expensive than many instructions, so optimizing away a few instructions isn't really going to win your software any speed tests. To win speed tests you have to measure the system, see where the bottlenecks are, and clear them if you can. That, though, is a subject for another time.

KV

2.5 Code Spelunking

Fools ignore complexity; pragmatists suffer it; experts avoid it; geniuses remove it.

Alan Perlis

A career in software means a career spent reading and trying to understand other koder's kode. The term *code spelunking*, which I coined in this article, tries to give a feel for what this process is like, to wit, crawling around a very large, dangerous space in the dark with primitive tools and a very small lamp. Many years have passed since I wrote this piece, and yet the tools used to spelunk code have not really kept pace with the amount of code any one person might need to dig through; in fact, the amount of code one would need to read, and the number of computer linguistical boundaries one has to cross has multiplied significantly.

A tool that is now sorely needed but that seems to elude us is one that provides useful visualization of a large code base. Most of the current crop of code spelunking tools, be they standalone or part of an IDE, allow a programmer to skip through the code in order to dive down, or up, a call chain. Systems like Doxygen have a rudimentary ability to produce a visual call graph, but these are static, hard to navigate, and quickly fail at scale. If there is one tool that most koders would use every day, it is indeed this sort of tool, because what most of us do, day to day, is dive into a morass of multiplying functions and try not to drown. The following piece is a tour of tools that might keep you afloat.

Try to remember your first day at your first software job. Do you recall what you were asked to do, after the human resources people were done with you? Were you asked to write a piece of fresh code? Probably not. It is far more likely that you were asked to fix a bug, or several, and to try to understand a large, poorly documented collection of source code.

Of course, this doesn't just happen to new graduates; it happens to all of us whenever we start a new job or look at a new piece of code. With experience we all develop a set of techniques for working with large, unfamiliar source bases. This is what I call code spelunking.

Code spelunking is very different from other engineering practices because it is done long after the initial design and implementation of a system. It is a set of forensic techniques used after the crime has been committed.

There are several questions that code spelunkers need to ask, and tools are available to help them answer these questions. I will look at some of these tools, addressing their shortcomings and pointing out possible improvements.

Source bases are already large, and getting larger all the time. At the time of this writing, the Linux kernel, with support for 17 different processor architectures, is made up of 642 directories, 12,417 files, and more than 5 million lines of code. A complex network server such as Apache is made up of 28 directories and 471 files—encompassing over 158,000 lines of code—whereas an application such as the nvi editor contains 29 directories, 265 files, and over 77,000 lines of code. I believe that these examples are fairly honest representations of what people are confronted with when they start a job.

Of course, even larger systems exist for scientific, military, and financial applications, but those discussed here are more familiar and should help convey an instinctive feeling for the complexity involved in systems that we all come in contact with every day.

Unfortunately, the tools we have to work with often lag behind the code we are trying to explore. There are several reasons for this, but the primary one is that very few companies have been able to build a thriving business on tools. It is much easier to make money selling software that appeals to a broad audience, which software tools do not.

Static vs. dynamic. We can use two scales to compare code spelunking tools and techniques. The first one ranges from static analysis at one end to dynamic analysis at the other. In static analysis you're not observing a running program but are examining only the source code. A clear example is using the tools find, `grep`, and `wc` to give you a better idea of the size of the source code. A code review is an example of a static technique. You print the code, sit down at a table with it, and begin to read.

On the dynamic end of the scale are tools such as debuggers. Here you run the code on real data, using one program (the debugger) to examine another as it executes. Another dynamic tool is a software oscilloscope, which displays a multithreaded program as a series of horizontal time lines—like those on a real oscilloscope—to find deadlocks, priority inversions, and other bugs common to multithreaded programs. Oscilloscopes are used mostly in embedded systems.

Brute force vs. subtlety. The second type of scale measures the level of finesse that is applied to spelunking. At one extreme is a brute-force approach, which often consumes a lot of CPU time or which might generate a large amount of data. An example of brute force is attempting to find a bug by using `grep` to locate a message that prints out near to where the error occurs.

At the other end of the finesse scale is subtlety. A subtle approach to finding a string within a program involves a tool that builds a database of all the interesting components of a code base (e.g., function name, structure definitions, and the like). You then use that tool to generate a fresh database each time you update your source from the code repository. You would also use it when you want to know something about the source, as it already has the information you need at its virtual fingertips.

Plotting your method. You can use both of these scales to create a two-dimensional graph in which the x-axis represents finesse and the y-axis runs from static to dynamic. You can then plot tools and techniques on this graph so that they can be compared.

None of the terms used implies value judgments on the techniques. Sometimes a static, brute-force approach is just what you want. If it would take just as long (or longer) to set up a tool to do some subtle analysis, a brute-force approach makes sense. You don't often get points for style when code spelunking; people just want results.

One thing to keep in mind when code spelunking is the Scout motto, "Be prepared." A little effort spent up front getting your tools in place will always save time in the long run. After a while, all engineers develop a stable of tools that they use to solve problems. I have a few tools that I always keep on hand when code spelunking, and I make sure to prepare the ground with them whenever I start a project. These tools include Cscope and global, which I'll discuss in more detail later.

A very common scenario in code spelunking is attempting to fix a bug in an unfamiliar piece of code. Debugging is a highly focused task: You have a program, it runs, but not correctly. You must find out why it does this, where it does this, and then repair it. What's wrong with the program is usually your only known quantity. Finding the needle buried in the haystack is your job, so the first question must be, "Where does the program make a mistake?"

You can approach the problem in several ways. The approach you choose depends on the situation. If the program is a single file, you could probably find the bug through inspection, but as demonstrated, any truly useful application is much larger than a single file.

Let's take a theoretical example. Jack takes a job with the Whizzo Company, which produces WhizzoWatcher, a media player application that can play and decode many types of entertainment content. On his first day at work (after he has signed up for health insurance, the stock plan, and the 401K) Jack's boss e-mails him two bug reports to deal with.

The two bugs that Jack has just been assigned are as follows:

Bug 1: When WhizzoWatcher opens a file of type X, it immediately crashes with no output except a core file. Bug 2: While watching a long movie on DVD (*The Lord of the Rings, The Two Towers*) the audio sync is lost after about two hours. This does not happen at any particular frame; it varies. WhizzoWatcher 1.0 is a typically organic piece of software. Originally conceived as a "prototype," it wowed the VPs and investors and was immediately rushed into production over the objections of the engineers who had written it. It has little or no design documentation, and what exists is generally inaccurate and out of date. The only real source of information on the program is the code itself. Because this was a prototype, several pieces of open source software were integrated into the larger whole and were expected to be replaced when the "real system" was funded. The total system now consists of about 500 files that spread over 15 directories, some of which were written in-house and some of which were integrated.

Bug 1. This is the easiest bug for Jack to work on; the program crashes when it starts. He can run it in the debugger, and the offending line will be found at the next crash. In terms of code spelunking, he doesn't have to look at much of the code at first, although becoming generally familiar with the code base will help him in the long run.

Unfortunately, Jack finds that although the reason for the immediate crash is obvious, what caused it is not. A common cause of crashes in C code is dereferencing a null pointer. At this point in the debugger, Jack doesn't have the previous states of the program, only the state at the moment it crashed, which is actually a very small amount of data. A common technique is to visually inspect the code while stepping up the stack trace to see if some caller stomped on the pointer.

A debugger that could step backward, even over a small number of instructions, would be a boon in this situation. On entry into a function the debugger would create enough storage for all of the function's local variables and the application's global variables so that they could be recorded, statement by statement, throughout the function. When the debugger stopped (or the program crashed), it would be possible to step backward up to the start of the function to find out which statement caused the error. For now, Jack is left to read the code and hope to stumble across the real cause of the error.

Bug 2. Attacking Bug 2, where the code doesn't crash but only produces incorrect results, is more difficult because there isn't a trivial way to get a debugger to show you when to stop the program and inspect it. If Jack's debugger supports conditional breakpoints and watchpoints, then these are his next line of defense. A watchpoint or a conditional breakpoint tells the debugger to stop when something untoward happens to a variable, and allows Jack to inspect the code at a point closest to where the problem occurs.

Once Jack has found the problem, it's time to fix it. The key is to fix the bug without breaking anything else in the system. A thorough round of testing is one way to address this problem, but he'll feel more comfortable making a fix if he can find out more about what it affects in the system. Jack's next question ought to be, "Which routines call the routine I want to fix?"

Attempting to answer this question with the debugger will not work, because Jack cannot tell from the debugger all the sources that will call the offending routine. This is where a subtle, static approach bears fruit. The tool Cscope builds a database out of a chunk of code and allows him to do the following:

Find a C symbol

Find a global definition

Find functions called by a function

Find functions calling a function

Find a text string

Change a text string

Find an `egrep` pattern

Find a file

Find all files that `#include` this file

You will note that item 4, "Find functions calling a function," answers his question. If the routine he's about to fix modifies nonlocal variables or structures, he must next answer the question, "With which functions does my function share data?" This would never come up in a "properly designed program" (i.e., one written from scratch). Jack would never use a horde of global variables to control his program, because he knows what a nightmare it would be to maintain. Right?

Of course, this is code spelunking, so Jack is already past that point. Using a subtle tool such as Cscope again is tempting, but this turns out to be a case where brute force is his best hope. Generating a list of all the global references in a program (filename and line number) is certainly possible for a compiler, but none of them do this. Although this option would hardly be used in the creation of new code, it would make the process of debugging old code far easier. Jack will have to make do with a combination of find and `grep` to figure out where all these variables reside in the program.

Code spelunking isn't something you do only when debugging; it's how you perform a good code review, reverse-engineer a design document, and conduct a security audit.

During an age when many people use computers for financial and personal transactions, auditing code for security holes has (or should) become commonplace. In order to close these security holes, you have to know what the common attacks are, as well as which sections of the code are vulnerable to attack. Although the attacks are updated almost daily on the Bugtraq mailing list (`http://www.securityfocus.com`), what we're concerned with is finding them.

Consider the following example. Jill takes a job with a large bank that serves most of its customers electronically, over the Internet. On her first day her boss tells her that the bank is doing a security audit of its entire system, which is implemented on several different types of hardware and operating systems. One set of Unix servers handles incoming web traffic and then makes requests to a mainframe backend that actually manages the real money.

One possible security hole occurs when a program stores a user's private data in a way that makes it available to anyone with access to the machine. An example of this is storing a password as plain text in a file or registry key. If Jill had written the software or had access to the person who did, she could quickly find out where the program stores passwords simply by asking. Unfortunately, the person who wrote the login scripts was laid off when the bank moved its headquarters six months ago. Jill will have to find out how this is done without the author's help.

Unlike in the debugging case, few tools are available to tell Jill something as specific as, "Where does the program store data X?" Depending on the structure of the program, this could happen anywhere—and often does. Jill can tackle this problem with either a brute-force or a subtle approach. To do the former she would litter the code with debugging statements (i.e., `printfs` or language equivalent) to find out where the program stores the password. She probably has a few guesses to narrow the field from the entire program down to a few or a dozen locations, but this is still a good deal of work.

A subtler approach would be to run the program in the debugger, attempt to stop the program right after the user enters a password, and then follow the execution until the program performs the storage operation.

A typical way to attack a program is to give it bad input. To find these vulnerabilities, Jill must ask the question, "Where does the program read data from an untrustworthy source?" Most people would immediately think that any code dealing with network data would be vulnerable, and they would be right—in part. With the advent of networked file systems, the fact that the code is executing a `read()` statement does not imply that it is reading from a local (i.e., trustable) source.

Whereas Jack's debugging in my earlier example attempted to zero in on a problem, Jill's code audit is more of a "fan-out." She wants to see as much of the code as possible and understand how it interacts both with its own modules ("How is data passed around?") and with external entities ("Where do we read and write data?"). This can be an even more daunting task than finding a needle in a haystack and may be more like labeling all the individual straws. In this case, the most important thing for Jill to do is to find the most likely places that will cause problems—that is, the places that are executed most often.

There is a tool that, although not originally intended for this purpose, can help to focus Jill's efforts: a code profiler. For example, gprof was originally written to tell an engineer which routines within the program are using up all the CPU time and are, therefore, candidates for optimization. The program is run with a workload (let a user bang on it or have it service requests from the network), and then the output is analyzed. A profiler will tell Jill which routines are being called most often. These are obviously the routines to check first. There is no reason for Jill to pore over the large portion of the code that is called infrequently, while the routines called most often may have gaping holes.

Routines that pass bad arguments to system calls are another common security problem. This is most often exploited against network servers in an effort to gain control over a remote machine. Several commercial tools attempt to discover these kinds of problems. A quick and dirty approach is to use a system call tracer, such as ktrace or truss, to make a record of what system calls are executed and what their arguments are. This can give you a good idea of where possible errors lie.

Code spelunking is about asking questions. The challenge is to get your fingers around the right parts of the code and find the answers without having to look at every line (which is a near impossibility anyway). There is one tool I haven't yet mentioned in this article, and that's the one sitting inside your head. You can begin applying good engineering practices even though they weren't used to create the code you're spelunking.

Keeping a journal of notes on what you've found and how things work will help you to create, and keep in mind, a picture of how the software you're exploring works. Drawing pictures is also a great help; these should be stored electronically along with your notes. Good experiment design of the type you may have learned—and then promptly forgotten—in physics class is also helpful. Just beating on a piece of code until it works

is not an efficient way of figuring it out. Setting up an experiment to see why the code behaves a certain way may take a lot of thought, but usually very little code.

Finally, one of my favorite tools is the "stupid programmer trick." You get a colleague to look at the code, and then you attempt to explain to him or her what the code does. Nine times out of ten your colleague won't even need to say anything. Fairly quickly you realize what's going on, smack yourself on the forehead, say thank you, and go back to work. Through the process of systematically explaining the code out loud, you have figured out the problem.

There is no one tool that will make understanding large code bases easier, but I hope that I've shown you a few ways to approach this task. I suspect you'll find more on your own.

Tool Resources

Global

`http://www.gnu.org/software/global/`

This is the tool I apply to every source base that I can. Global is really a pair of tools: gtags and htags. The first, gtags, builds a database of interesting connections based on a source tree in C, C++, Java, or YACC. Once the database is built, you can use your editor (both Emacs and vi are supported) to jump around within the source. Want to know where the function you're calling is defined? Just jump to it. The second tool is htags, which takes the source code and the database generated by gtags and creates an HTML-browsable version of the source code in a subdirectory. This means that, even if you don't use Emacs or vi, you can easily jump around the source code finding interesting connections. Building the database is relatively speedy, even for large code bases, and should be done each time you update your sources from your source-code control system.

Exuberant Ctags

`http://ctags.sourceforge.net`

Works on dozens of languages: Ant, Asm, Asp, Awk, Basic, BETA, C, C++, C#, Cobol, DosBatch, Eiffel, Erlang, Flex, Fortran, HTML, Java, JavaScript, Lisp, Lua, Make, MatLab, OCaml, Pascal, Perl, PHP, Python, REXX, Ruby, Scheme, Sh, SLang, SML, SQL, Tcl, Tex, Vera, Verilog, VHDL, Vim, YACC. Very useful in a multilanguage environment.

Cscope

`http://cscope.sourceforge.net/`

Cscope was originally written at AT&T Bell Labs in the 1970s. It answers many questions, such as: Where is this symbol? Where is something globally defined? Where are all the functions called by this function? Where are all the functions that call this function? Like Global, it first builds a database from your source code. You can then use a command-line

tool, Emacs, or one of a few GUIs written to work with the system to get your questions answered.

gprof

`https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html`

This is the standard profiling tool on most open source Unix systems. Its output is a list of routines ordered by how often they were called and how much of the program's CPU time was spent executing them. This can be useful in figuring out where to start looking for security holes in a program.

ktrace

This is a standard tool on open source operating systems. The name stands for "kernel trace." It will give you a listing of all the system calls made by a program. The output includes the arguments and return values given to and received from the calls. You use ktrace by running a program under it and then dumping the output with kdump. It is available on all open source Unix operating systems.

DTrace

First developed on Solaris, but now available on FreeBSD, Linux, and Windows. The best reference on this tool is still the book by Brendan Gregg and Jim Mauro.[1]

Valgrind

`https://valgrind.org`

Useful for finding all kinds of memory leaks in C and C++ programs.

1. Brendan Gregg/Jim Mauro: *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*, 2011.

2.6 Input Validation

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilisation.

Gerald Weinberg

Of all the stupid things, and there are many in software and security, perhaps the most stupid is the continuing inability of koders to remember to validate their input. From cross-site scripting to SQL injection attacks to so many other things, not properly validating input leads to a whole host of problems on your hosts. Perhaps my favorite reference on this is from Randall Munroe's xkcd, in which he writes about a child named, in part, "Drop Tables"; see `https://xkcd.com/327/`. There are now libraries and guidelines galore on this topic, for every computer language you can imagine; all that is required is the will to use them.

Dear KV,

I work for a company the builds all kinds of different web applications. We do everything from blogs and news sites to mail and financial systems. It depends on what the customer wants.

Right now our biggest problem at work is the number of bugs we have that relate to input validation. These bugs are totally maddening because each time one of them is fixed some other problem pops up in the same code, and the checking code is getting very close to spaghetti. Is there any way out of this tangle without some mythical technology, like natural language understanding?

Input Invalid

Dear II,

You've come across one of the biggest programming problems since the day we stupidly let non-engineers, i.e., users, touch our nice toys. Of course, computers aren't really very useful if they don't do something for actual people, but it is a pain. Systems would be so much cleaner without people. Alas, user input is a fact of life, and one that we all have to work with every day. User input is also one of the biggest sources of security holes in software as any reader of the Bugtraq mailing list can tell you.

The first rule of handling user input is, "*Trust no one!*" in particular your users. Although I'm sure 90% of them are perfectly nice people who go to their religious shrine of choice at the appointed time every week, or whatever it is perfectly nice people do, I don't actually know any perfectly nice people, but I have heard about them; nevertheless, there are the usual minority of thieves, jerks, and just plain idiots who will look at your nice web form as a place to steal money, play tricks, and generally cause havoc. The rest of the people, the perfectly nice ones, whom I've never met, won't actually attack your system, they'll just use it in a way they think is logical, and if their logic and your logic do not match, kaboom. Kode Vicious hates kabooms; they mean late nights and complaints from my doctor about alcohol and caffeine intake. I can't help it if he's stingy with the prescription meds, but let's not get into that now.

The second rule is, "*Don't trust yourself!*" This is another way of saying that you should check your results to make sure you're not missing anything. Just because you sent something to the user does not mean that they didn't do something a bit odd to it before it came back to you. A quick example is a web form. If you depend on the data you sent in a web form to the user, you had better check the whole form, and not just the parts you expected the user to change with their browser. It's a simple trick to exploit an error in form submission code by sending a slightly changed form with proper user input.

It sounds, from your description, as if the system you're using was written using what is called a black list. A black list is a set of rules that says which things are bad. During the Cold War the United States maintained black lists to prevent people it didn't like from

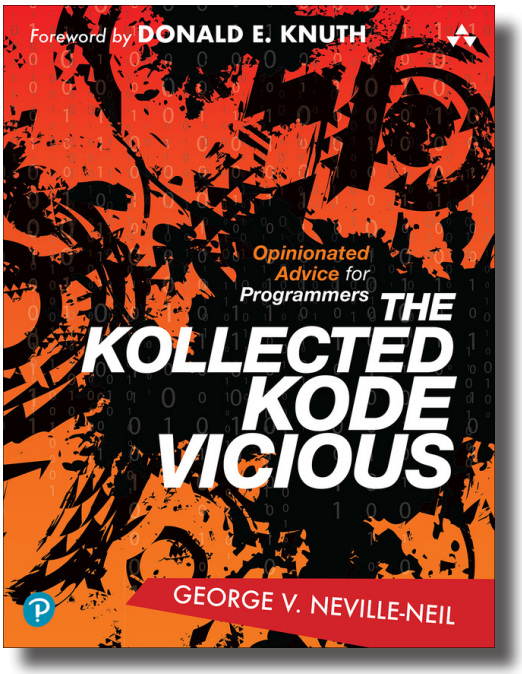
getting jobs. Your name is on the list, sorry, no job. In the same way, software uses black lists to say which types of operations, in this case user input, are bad. The problem with black lists is that they are hard to maintain. They start off simple enough, saying things like, "Do not accept input with URLs in them." But they quickly get out of hand, with lists of the names for "JavaScript," of which there are many, and different types of tags to check for, and, and, and… I hope you get the idea. It is better to use white lists where this is possible.

A white list, unsurprisingly, is the opposite of a black list. White lists only contain the things that are allowed, and are often very short. An example is "accept only ASCII alphabetic characters." White lists can be overly restrictive, but they have a distinct advantage over black lists in that the only time you have to change a white list is to make it more permissive. A black list is, by default, mostly permissive, with the few exceptions that are the entries in the list.

My recommendation is to switch to using white lists, and to be very restrictive in what can be given to you by the user. Initially this seems a bit draconian, but it's probably the best way to protect your code, both from users and from turning into spaghetti.

KV

Pragmatic, Bite-Sized Programming Advice from Koder-with-Attitude, Kode Vicious



ORDER & SAVE

Save 35% When You Order

from **informit.com/infoq/kode**
and enter the code **KODEVICIOUS**
during checkout

FREE US SHIPPING on print books

Major eBook Formats
Only InformIT offers PDF, EPUB, & MOBI together for one price

OTHER AVAILABILITY

Through O'Reilly Online Learning **(Safari)** subscription service

Booksellers and online retailers including Amazon/Kindle store and Barnes & Noble | bn.com

"For many years I have been a fan of the regular columns by Kode Vicious in Communications of the ACM. The topics are not only timely, theyre explained with wit and elegance.
—From the Foreword by Donald E. Knuth

Writing as Kode Vicious (KV), George V. Neville-Neil has spent more than 15 years sharing incisive advice and fierce insights for everyone who codes, works with code, or works with coders. Now, in The Kollected Kode Vicious, he's brought together his best essays and Socratic dialogues on the topic of building more effective computer systems. These columns have been among the most popular items published in ACMs Queue magazine, as well as Communications of the ACM, and KVs entertaining and perceptive explorations are supplemented here with new material that illuminates broader themes and addresses issues relevant to every software professional.

Neville-Neil cuts to the heart of the matter and offers practical takeaways for newcomers and veterans alike on the following topics:

- The Kode at Hand: What to do (or not to do) with a specific piece of code
- Koding Konundrums: Issues that surround code, such as testing and documentation
- Systems Design: Overall systems design topics, from abstraction and threads to security
- Machine to Machine: Distributed systems and computer networking
- Human to Human: Dealing with developers, managers, and other people

Each chapter brings together letters, responses, and advice that apply directly to day-to-day problems faced by those who work in or with computing systems. While the answers to the questions posed are always written with an eye towards humor, the advice given is deadly serious.

Pearson Addison-Wesley

**Discount code KODEVICIOUS confers a 35% discount off the list price on informit.com only. Discount may not be combined with any other offer, including the book + eBook "Best Value" bundle, and is subject to change.*

informIT® the trusted technology learning source