5 Famous Programming Quotes, Explained

You can become a better developer by understanding these timeless insights

Matthew MacDonald
Feb 12 · 8 min read ★

Illustration courtesy of Icons 8

To be a programmer is to sign yourself up for a life of constant learning. The fountain of *new* — new features, new languages, new tools, new frameworks —never stops gushing. But computer science is also a surprisingly traditional field that's grounded in time-tested principles. We've added object-oriented programming, modern hardware, and artificial intelligence. But despite these changes, many of the insights that were first articulated a generation ago still hold true today.

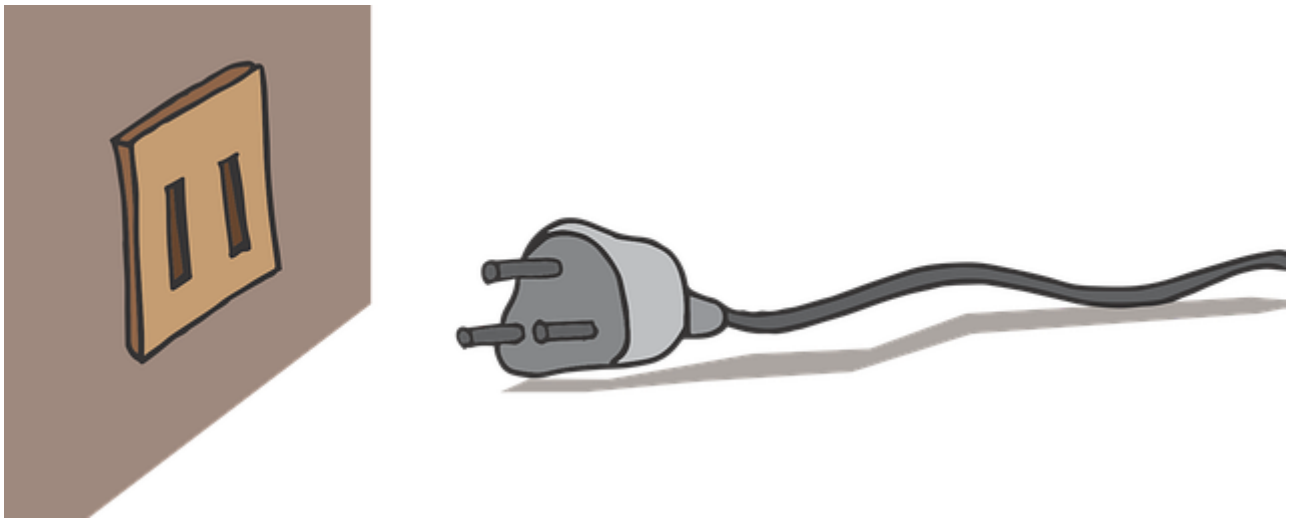
In this article, I've dissected a few of my favorite quotations about computer science. The only condition I set is that each quote must be at least twenty years old. Because while old technology quickly becomes useless, the ancient commandments of our programming ancestors have a lot more staying power.

1. On Indirection

> “All problems in computer science can be solved by another level of indirection.” — David Wheeler

Here’s an often-repeated compsci quote that few developers want to explain. But it’s one my *favorite* programming truths, because it strikes at the heart of what coding is all about.

The easiest way to start thinking about indirection is to imagine layers. For example, say you have a small project that needs to fit component A into component B:



All the pieces, none of the fit

Both components are standardized, so you can’t break them open and change the way they work. You could build a whole new component (`PlugTwoProngVariant`) but that’s a lot of work and unnecessary duplication. A better approach is to add a layer in between the two pieces — an adapter that fits into both components and translates between them.

Now, if indirection was just about adding a new layer to fit incompatible pieces together, it would be nice but narrowly useful. But the idea of solving problems by building *around* them is a concept that stretches from the bottom to the top of computing. You see it when you’re trying to fit a new data model to an old user interface. You see it when you’re trying to fit a legacy application to a new web service backend. You see it when you need to strap on higher-level features like logging and caching, or coordinate higher-level services like messaging and transactions. And at the top of the pyramid, you’ll get to rarefied topics like machine learning (when you can’t code the behavior you need, write another layer of code that will figure it out for you).

Plenty of people will tell you that coding is about writing explicit instructions in a language that even idiot computers can understand. But David Wheeler's quote reveals a better insight. Good programming is about climbing the ladder of abstraction to get to the most general solutions.

Bonus related quote:

Indirection is powerful, but there's a cost to complexity. People very rarely quote Wheeler's immediate follow-on remark about indirection:

> "But that usually will create another problem." — David Wheeler

That truth has kept programmers in business ever since.

2. On Simplicity

> "Simplicity is prerequisite for reliability." — Edsger Dijkstra

There's no shortage of wise programmers warning us not to overcomplicate our code. But few put the cost of complexity any clearer than Dutch computer science pioneer Edsger Dijkstra.

Here's the insight: You don't choose simplicity as a gift to the future. You don't do it because you're anticipating the chance to reuse your code, or because you want it to look cleaner at a code review, or because you want to make it easier to modify. (Although all these benefits are valuable!) You do it because simplicity is a *prerequisite*. Without simplicity, you can never have reliable code that you can trust to run a business or handle your data.

To accept Dijkstra's point, we need to redefine what "good code" means. It's not the shortest code. It's not necessarily the fastest code. It's definitely not the cleverest code. It's code that can be *trusted*.

Bonus related quote:

One of best ways to keep code simple is to remember that less is more. Dijkstra offers a metric to help us keep that in mind:

> "If we wish to count lines of code, we should not regard them as 'lines produced' but as 'lines spent.' "— Edsger Dijkstra

3. On Readability and Rewrites

> "It's harder to read code than to write it." — Joel Spolsky

At first glance, this quote by software legend and StackOverflow co-creator Joel Spolsky seems true but deceptively shallow. Yes, code can be dense, terse, and tediously long. And it's not just other people's code. If you look at your own work from a year ago, you'll probably need some time to sort through the logic you once knew intimately.

But Spolsky's insight comes with a twist. The danger of code you can't read isn't just the obvious (it's hard to change it and improve it). Instead, the greater danger is that complex code appears to be *worse* than it actually is. In fact, the burden of trying to understand someone else's already written code is so great that you might be tempted to make what Spolsky calls the worst possible mistake—deciding to rewrite that code from scratch.

It's not that rewrites can't improve the architecture of a system. They definitely can. But these improvements come at great expense. They reset the clock on testing and bug-fixing, two activities that take far longer than mere coding. Rewrites are tempting because they play on one of the most common biases in software development: We understate the effort to do things that are conceptually simple. It's why the final 5% of a project takes 50% of the time. Easy things can be surprisingly time-consuming! And nothing seems easier than solving a problem you've already solved.

So if you shouldn't rewrite everything to make it perfect, what's the better solution? The answer is to get every developer involved in constant bite-sized refactoring. This

gives you small, continuous code improvement — real rewards with minimal risks. You can improve readability on the way.

Bonus related quote:

If you're still in doubt about the importance of readability, Martin Fowler helps to put it in perspective:

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." — Martin Fowler

In other words, a programmer's job isn't just to make things work, but to make things make *sense*.

4. On Repetition

> "Don't repeat yourself. Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." — Andy Hunt and Dave Thomas

Every self-respecting programmer knows that repetition is the heart of much evil. If you're writing the same thing in different places, you're making extra work writing, testing, and debugging. Even worse, you're introducing the possibility for inconsistencies — for example, if one part of the code is updated but other, similar routines aren't brought into agreement. An inconsistent program is a program you can't trust, and a program you can't trust is no longer a viable solution.

A `GetTeamUniform()` method could have avoided this bug

However, code isn't the only place where repetition wreaks havoc. This version of the famous "don't repeat yourself" (DRY) rule expands the no-duplication principle to cover the other places where inconsistencies can hide. We're no longer talking about code duplication. We're also talking about repetition in a *system* — and a system has many different ways of encoding knowledge. They include:

- Code statements
- Code comments
- Developer or client documentation
- Data schemas (for example, database tables)
- Other specifications, like test plans, workflow documents, and build rules

All of these tiers can overlap with each other. And when they do, they risk introducing different versions of the same reality. For example, what happens if the documentation describes one way of working but the application follows another? Who has the ownership of the truth? What happens if the database tables don't match the data model in the code? Or the comments describe the operation of an algorithm that doesn't match its actual implementation? Every system needs a single, authoritative representation from which everything else derives.

Incidentally, competing versions of the truth isn't just a problem in small-scale projects or poorly designed code. One of the best examples erupted into the public with the battle between XHTML and HTML5. One camp argued that the specification was the official truth, and browsers needed to be corrected to follow it. The other faction claimed that browser *behavior* was the de facto standard, because that's what designers had in mind when they wrote web pages. In the end, the browser version of the truth won. From that point on, HTML5 was what browsers *did* — including the shortcuts they allowed and the errors they accepted.

Bonus related quote:

The possibility for code and comments to contradict each other has opened a heated debate about whether comments do more harm than good. The proponents of Extreme Programming treat them with open suspicion:

> “Code never lies; comments sometimes do.” — Ron Jeffries

5. On Hard Problems

> “There are only two hard things in computer science: cache invalidation and naming things.” — Phil Karlton

At first glance, this quotation seems like an amusing but ordinary programming joke. The contrast between one something that sounds difficult (cache invalidation) and something that sounds painless (naming things) is instantly relatable. Every programmer has invested hours of work fixing a ridiculously trivial issue, like a pair of parameters passed in the wrong order or an inconsistently capitalized variable (thanks JavaScript). As long as humans need to partner with machines to get things done, programming is going to be a mashup of high-level system planning and trivial typos.

But if you take a second look at Phil Karlton’s quote, there’s more to unpack. Naming things isn’t hard just because a programmer’s life is regularly ruined by tiny headaches. It’s also because the issue of naming is actually the edge of every programmer’s most important job: software *design*. In other words, how do you write code that’s clear, clean, and consistent?

There are plenty of ways to get naming wrong. We’ve all seen variables named after data types (`myString` , `obj`), abbreviations (`pc` for product catalog), some trivial implementation detail (`swappable_name` , `formUserInput`), or even nothing at all (`ret_value` , `tempArray`). It’s easy to fall into the trap of naming a variable based on what you’re doing with it at that moment rather than what it contains. And boolean values are particularly tricky — does `progress` set when progress starts, indicate that you need to display progress information in the UI, or flag something completely different?

With permission from CommitStrip.com

But variable names are just the beginning. Naming *classes* raises the question of how you divide code into independent parts. Naming public members shapes how you present the interface that allows one part of your application to interact with another. Locking down these names doesn't just describe what a piece of code can do, it determines what it will do.

Bonus related quote:

> "There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors." — Leon Bambrick

. . .

Like lists and want to grow your skills? Check out 5 Books That Can Help You Become a

Better Programmer. And for a once-a-month email with our best tech stories, subscribe to the Young Coder newsletter.

Programming Computer Science Design Patterns Software Development Gang Of Four