Editor: **Gerard J. Holzmann**
NASA/JPL
gholzmann@acm.org

Frequently Unanswered Questions

Gerard J. Holzmann

WHEN YOU'RE TRYING to find that bug that makes your code do something that defies logic, the best thing you can do to get unstuck is to explain to a colleague why the bug can't happen. Your colleague doesn't have to understand your code or do more than just wait for you to say, "Ah, never mind, I figured it out." I've been on both sides of these "discussions": as the developer trying to figure something out and as the listener waiting for that aha moment.

The key here is that you force yourself to explain to another human being the things that appear obvious to you, as clearly and concisely as you can. To do so, you have to find the essence of a problem, the key insights necessary to understand it, and why the code you wrote was designed to work that way.

The magical "please help me debug this" conversation is very different from the discussion you'd have if you were teaching someone to use your application. In the latter case, you'd likely concentrate on just the part the user needs to know, and you wouldn't spend much time on the internal details. You'd more likely focus on the "what" than on the "why" or "how" of your application.

This brings me to the question of how we generally write and document code. Just about every nontrivial tool or technique you want to use must be learned, and often you don't have the benefit of talking directly to the designers about the "why" or "how." You can only look at the manual that spells out all parts of the "what" in excruciating detail. That can be a problem, because if you don't first understand the "why" or "how" of a new tool, you might have considerable difficulty finding the answer to the mundane problems you might run into when trying to come up to speed. It can take a long time to develop the intuition that makes all the puzzle pieces fall in place before you can answer the low-level questions.

Once I start writing the manual for a tool I'm developing, I often go back to the code to restructure it so that explaining how it works becomes easier. Sometimes we take short-cuts in an implementation, creating exceptions to more general usage principles that apply only in special circumstances. You as the designer will have no trouble remembering those quirks and why they hide in the code, but things change when you have to explain them to someone else. More often than not, it's easier to go back to the code and remove the quirks so that nothing needs to be explained or remembered. Clearly, doing so improves the code and simplifies the documentation. The only reason it happens is that you make yourself explain things to a new user by writing the manual as clearly as possible. It's this process of explaining that helps reveal the dusty corners and the less-than-perfect logic you used in writing the first version of your code. The most assured trigger you have to clean it up is to force yourself to explain it.

Thinking in Phases

We often think of software development as an ideally streamlined process consisting of three phases: design, build, and test. If we want our code to be useful to anyone, though, we'll reluctantly have to acknowledge that we also need a documentation phase, which typically comes at the very end. Generally, there will be some iterations across these phases before we get everything right. Figure 1a illustrates the idealized process.

If writing clear documentation can affect code structure in the way I am suggesting, this classic software development model perhaps isn't the best way to produce good code.

Larger companies can afford to employ armies of people who are good at producing volumes of user documentation. Sadly, many of these professional documenters don't really understand the code they have to describe. Usually, this produces overwhelming amounts of poorly organized text that users must search to see if it happens to cover the questions they need answered. And, if I'm right, the lack of early documentation can also result in substandard code.

I was reminded of this phenomenon when I installed a new version of a popular static source code analyzer on my system. It ended up consuming an astounding 2 Gbytes of disk space, with close to 200 Mbytes of documentation. It's a fair bet that no user will read through all the documentation before using the tool—until he or she gets stuck, of course, and needs a specific question answered. Then, the goose chase will begin, to find the answer in the wealth of text. If the user is lucky, a text search will find it. More likely, though, the specific instance of the problem the user ran into won't be described anywhere and must be debugged with the help of someone who actually understands the code.

The Guessing Game

Not all applications come with megabytes of documentation. It's either feast or famine. Sometimes, the only documentation will be an online FAQ, which is perhaps best described as a modern version of the guessing game. It's similar to how a help line is typically organized. If you call a help line, you'll likely end up in the clutches of an algorithm that tries to guess why you're calling. "Press 1 if you're calling about X; press 2 if ...." You end up listening to all the guesses before you can make up your mind about the one that came the closest to your problem. And then you'll have to start all over again.

If your laptop fails to connect to the Internet, is that a hardware problem, a software problem, a network problem, or an authentication problem? Well, my guess is that you don't know, and that's probably why you're calling the help desk. But before you can speak to a human, you'll have to win the guessing game. If you guess wrong, you'll just have to call again and try a different response.

Write the Manual First

I suspect that the reason for the poor quality of most tool documentation is that it's written much too late in the software development cycle, and often by people who don't know much about the actual code. Perhaps, if we started producing the documentation first, especially the user manuals, before we start writing code, the quality of both the documentation and software would improve.

In addition, the quantity of the documentation would shrink to something that can actually be useful to ordinary mortals. If the documentation is written early, the "what" details aren't available yet, forcing the documenter to get the "why" and "how" right. The user will then more likely find a logical structure that provides the insight needed to resolve the conundrums that otherwise defy explanation.

Good documentation, like good writing, is typically short and to the point. Some of the most influential papers in science were remarkably short. For instance, Peter Higgs's

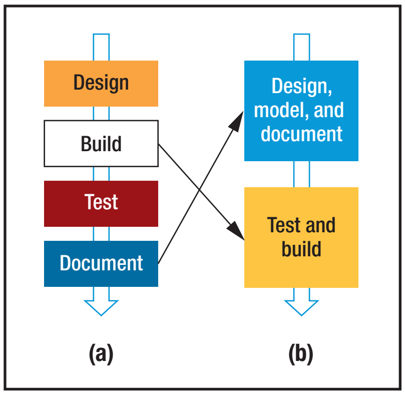

FIGURE 1. Two views of software development. (a) The typical, idealized view consists of four sequential phases, with documentation coming last. (b) The discipline of test-driven development says that you should write the test before you write the code. The structure of your code, and the logic of the design it's based on, will improve further if you document your application before you write either tests or code.

1964 paper postulating the existence of what we now call the Higgs boson was barely two pages.[1] In our own field, Gordon Moore's 1965 paper describing what we now call Moore's law was less than three pages,[2] and Edsger Dijkstra's famous "Go To Statement Considered Harmful" from 1968 was just a little over one page.[3]

If you started programming early enough, you probably learned C from Brian Kernighan and Dennis Ritchie's 1978 book *The C Programming Language*.[4] The main text of the book's first edition counted just 178 pages, plus an appendix with a full C reference manual of 40 pages. The book didn't start with a list of keywords; it started with the famous "hello world" program to ease readers into the language. Today, books seem to fight for shelf space with ever-increasing page counts. For example, a recent C++ textbook has 1,368 pages and a spine measuring close to two inches. The 65-page index to that book is larger than the full C reference manual from Kernighan and Ritchie's book. I know which type of book I'd rather learn from.

Test-Driven Development

Many have observed that the build and test phases shouldn't be separated as cleanly as Figure 1a seems to claim. In practice, the two phases will always overlap. A good case can further be made that test development should start even before any code is written. That is, the initial part of the test phase would precede the build phase. This approach is called *test-driven development*, with the mantra "Write the test before you write the code." When you develop code this way, initially all tests that have been defined will fail. But then slowly, one by one, as your code starts acquiring the functionality it's meant to have, the tests will start passing. When all tests pass, you're done.

In large software development projects, the design phase should result in a list of requirements the final code must satisfy. There's no reason why you couldn't develop the test suite at that point, which should eventually be able to show that each requirement was met. Documenting the requirements and tests up front will force you to be clear about the intended performance envelope of an application. That in turn can let you make an informed assessment of the risk that the application will find itself outside that performance envelope when conditions aren't favorable.

Clear requirements and clear documentation let us build high-level models of code that clearly describe the underlying design. This has inspired the approach of model-driven design, which is quickly gaining strength in industry. The biggest gain model-driven design brings, though, isn't the specification of a product itself but the ability to analyze it with powerful tools such as logic model checkers.

If we make documentation and modeling integral to design and make testing part of the build-and-code process, software development will start looking more like Figure 1b.

References

1. P.W. Higgs, "Broken Symmetries and the Masses of Gauge Bosons," *Physical Rev. Letters*, vol. 13, no. 16, 1964, pp. 508–509.
2. G.E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, 1965, pp. 114–117.
3. E.W. Dijkstra, "Go To Statement Considered Harmful," *Comm. ACM*, vol. 11, no. 3, 1968, pp. 147–148.
4. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, 1st ed., Prentice-Hall, 1978.

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.