developer.*
A Web Magazine for Software Developers

# Code Comprehension

By Daniel Read

I've been thinking a lot this week about code and documentation—more precisely, I've been thinking about code *as* documentation. The idea of "self-documenting code" is certainly not a new one. When, as a relatively inexperienced developer, I first read about self-documenting code in Steve McConnell's *Code Complete*, it was a revelation to me. My first real-world experience with programming was maintaining and debugging someone else's horrible code, and I knew intuitively from that trial by fire how important it is to make one's code clear and understandable. From that experience, I started concentrating on code-level comments as the key to making sure that code is properly documented and can be easily understood by others.

But McConnell's *Code Complete* really opened my eyes. Self-documenting code, he writes, is "the Holy Grail of legibility…Such code relies on good programming style to carry the greater part of the documentation burden. In well-written code, comments are the icing on the readability cake." (page 456) While reading the 450 pages leading up to this statement, I began to see how writing good quality code is really a matter of making thousands of tiny decisions along the way. Quality code does not happen by accident. It really takes true desire in the heart of the developer, because at each of those thousands of tiny decision points, there are so many opportunities for shortcuts, and so many opportunities to do something that reduces the quality of the code.

The key to achieving self-documenting code is what McConnell calls "good programming style." To achieve this "Holy Grail of legibility," it really takes constant attention, and an ongoing dialog with myself as I work. I don't know that I'll ever achieve it, but I'm always trying. I try to approach the writing of code as if I were making art. In my mind, the metaphor of the musician is most apt, since the musician must work within the mathematical confines of his medium, but at the same time can lend so much personal style to the end product.

Architecture is another useful analogy. The architect designing a building must obviously work within the confines of physics, engineering principles, and municipal building codes, but people pay huge premiums for the best architects, who infuse the building design with their unique stylistic statements.

But is self-documenting code enough? Clearly not, because the code itself can never capture the *intent* of the developer writing the code. The code documents the solution, but knowledge of the problem can only be inferred. This is where comments come in. Here is a quote from my description of The Principle of Comments (from an article called "Principled Programming":

  

> The code is the solution to the problem being solved. Ideally, the code should speak for itself. A person can read the code, and if it's good code, should be able to readily see what the code is doing and how it's doing it. However, what is lost over time is *what was in the mind of the developer who wrote the code*. In general, *this* is what needs to commented on. What was the *intent* of the developer? How is this code *intended* to be used? How does this code *intend* to solve the problem at hand? What was the idea behind the code? How does this code explicitly interrelate to other parts of the code? One of the greatest sins a developer can commit is to leave his code without his intent being clear to subsequent readers of the code.

What I try to capture most in my comments is to explain, at a high level, what my code is doing. I try to illustrate what my "scheme" is—how it all interconnects and works. I try to picture myself in the position of the maintenance developer, coming to my code for the first time, trying to figure out how it all works. What would he want to know about this code? I try to picture myself looking over the shoulder of that maintenance developer. How would I explain this code to him? What's not immediately obvious? What would his questions be? How would I explain to him *why* I did it this way? The answers to these questions are what I put in my "at the level of intent" comments.

There is a lot of controversy today over which methodology or "family" of methodologies is the best one (the truth is that different methodologies are better or worse in different circumstances), but the larger issues of process and methodology fall away when a developer is by herself, at the computer, writing code. The writing of code is a very personal act. It's just the coder and the code, alone at the computer. The opportunity exists for the developer (or pair of developers) to stamp his or her own unique signature and style onto the code, just as the musician or architect does.

However, while the desire to develop a truly advanced coding style is essential, it is not enough. The other essential part of the equation is knowledge. Can the musician produce beautiful music without knowledge of the rules of music? Can the architect design a beautiful building without knowledge of the laws of physics and the principles of structural engineering? Can we as developers produce beautiful code without the knowledge of what it takes to accomplish that?

In order to develop a style that leads to beautiful, self-documenting code (as if the two could be separated), in order to make the best choice in those thousands of tiny decisions—indeed, in order to even *recognize the opportunity* to make those decisions—we must have the knowledge of where the potential trouble spots are. We must know the techniques and conventions and standards. How can we choose the best data and routine names or use ideal layout techniques if we've never learned what the programmers of the last forty years have learned by trial and error? How can we properly modularize a system and make it tolerant to change if we are not intimate with the concepts of cohesion and coupling?

If one has that true desire to make one's code the best it can be, one must acquire the knowledge of what works, and what doesn't. One must learn the situational subtleties that

  

go into making the best choice in those thousands of tiny decisions that a developer makes when writing a system. Choice A might be the best choice in Situation 1, but in Situation 2, Choice B might be preferable. The rules and conventions are not necessarily absolutes. But if one does not know the rules and conventions, one cannot know when to adjust them to the situation. This is another way of phrasing the old adage, "First master the rules, then you can discern when to bend and break them."

(One must also master the peculiarities of the language being used, even if that language is not the developer's favorite. Each language is different, and has its own conventions for naming, layout, exception handling, etc. We should try to respect these differences, and not force the conventions of another language onto the one we are writing in today. Too many developers learn a style (good or bad) with their first language, and then impose that style on every other language they use in the future.)

If one desires to develop his or her own unique style, reading lots of great books, articles, and essays is an excellent first step. Make reading a continuing activity.

The next step would be practice, since the code editor and IDE is where theory gets put to the test. Strive to make each program better than the last one you wrote, and then move on to the next one.

Third, read and analyze other people's programs. If your team is not performing peer-based code reviews, suggest that you start. If you are learning on your own, download code from programming web sites, or look through the code that comes on the CD-ROM with a programming book. You will find plenty of bad code, and, if you're lucky, some exceptionally good code too.

Finally, if you are able, find a mentor. Identify someone around you that you look up to, whose opinion you respect, and who has the experience you would like to acquire. Ask that person if she will read your code and help you make it better. This can provide amazing leaps forward that might otherwise take months or years—or that might not ever happen at all.

###

Daniel Read is editor and publisher of the **developer.*** web magazine. He lives in Atlanta, GA, where he works as a software architect. He is currently at work on a book about software development crafted for a business audience.

  