

developer.***A Web Magazine for Software Developers**

Elevating Expressions

By Daniel Read

In response to a previous essay called “The Human Impact of Software,” a reader named Scott wrote to me and had this to say:

“In my opinion it is usually NOT the coder, but the rest of the software company who is at fault. The code and the coder are not the lynchpins of software development, but instead it is usually the managers and designers. You can't always blame the coder, and so your column is sort of biased against the coder.

“The managers of that company should have monitored the coder's progress, done code reviews, had good design docs, and made sure the code for THEIR business was good. It is THEIR business and therefore THEIR fault. If the coder sucks, then kick his @\$ out of the company and get another one. As for all the bugs that were found after the release, again this is management's fault for NOT having a proper testing process before release.

“You can't expect people to really care if their code is good or not. They are just hired to do a job and they will do it however they can to secure their paycheck. This is human nature. 10% of people do care. The rest don't and NEVER will. It is the same with all crafts in this world. Most people will do a poor job which will cause others stress and cost them lots of money and time. This is a terrible thing, but it is the job of the manager to make sure quality people are hired and well compensated.”

Scott's comments deserve a response, so I thought I would use the excellent opportunity as a basis for a follow-up essay.

I hear Scott making two major points, and I will hit them one at a time.

First, Scott makes the point that I placed all of the blame for the disastrous release described in the “The Human Impact of Software” on the developer who wrote the code. This is an understandable reaction, given that I did not make any mention of what mistakes management might have made, or what inadequacies might have existed in the overall development process. This is a failing of mine, and I should have balanced the column by making some mention of other factors that might have also influenced the failed release I described.

There were of course other factors at work, and yes, the management of the company has to accept ultimate responsibility for all of the problems caused by that release. It was, as Scott point's out, management's decision to release the software. Furthermore, there were probably deficiencies in the design and requirements process (though I was not at the

company during that time), and given the number of bugs that reached our customers, the testing process should have been more rigorous. Regular code reviews would also have been a stellar idea. So yes, other people share responsibility.

However, none of that absolves the developer of that horrible code. Horrible code is horrible code, and the only one who can write it is a developer. The designer did not write that bad code, nor did the managers, testers, or the person who gathered the requirements. The problems in that code were not related to process and management deficiencies, nor were they related to incomplete requirements, bad design, or an unreasonable schedule. When a developer writes bad code, any attempt to shift the blame for its failures to designers and managers is nothing less than a cop-out.

The point of the essay was not to focus on how managers and development processes can have an effect on the quality of software (and therefore the lives of real people)—of course they can. Rather, my point was to draw attention to the fact that there is a direct line between the abstraction embodied in our code and the reality of the people who will come into contact with that code. Methodologies and managers are beside the point—a distraction from the real issue.

Too often when we write code, we get caught up only in the exercise of the code itself and forget about the people who will have to use the software and maintain the code after we are long gone. We should write code as if we are going to compile it, hand it to the user, and that's it. We should strive to make it that good.

When I am working, I take this idea one step further: I pretend that I am going to have to sit there right next to the user every day and watch him use my application. Every time he has a problem understanding how it works, every time it crashes, every time a calculation is wrong, every time it takes too long, every time a cryptic error message or prompt comes up on the screen, he's going to turn to me, with a pained look on his face, and say "What are you trying to do to me?" In this way, I use my own ego and vanity to make sure that the user instead would turn to me and say "This is awesome!"

Another thing I do is picture another developer coming to my code with the job of maintaining it. I picture myself looking over that developer's shoulder, watching him trying to understand it and make some modifications. When I write code, I want to make sure that this developer would turn to me and say "Damn! This code is amazing. You have made my life as a maintenance developer so much easier. Thanks." These imagined scenarios may be silly, but they work for me.

Will we as developers make mistakes? Will we cause bugs? Will we take a shortcut here and there? Will we be forced by business realities to make compromises? Of course. I am not calling for absolute perfection. I live in the real world with everyone else. But in too many cases, the balance is shifted so far the other way that it is (and probably always has been) a huge problem for our profession. The number of bad developers out there is so high that it is an embarrassment to all of us.

Which brings us to Scott's second point: that 90% of all developers will always suck and won't ever care one bit about all this quality and craftsmanship jibber jabber. Perhaps Scott is right. Perhaps it is futile, naïve, or even arrogant to think that something like this essay is going to convert a bunch of people from poor developers to good ones. Perhaps as an industry we should just resign ourselves to the fact that 90% of developers will always suck and do everything we can to compensate for this fact—or just make sure that we only hire from that pool of top 10% people.

While I am a long-time fan of Sturgeon's Law, I am not quite this cynical when it comes to our burgeoning profession. I believe that we can—and must—raise the general level of quality and craftsmanship. The best way to do this in my opinion is to examine ourselves, do our best to improve, and set a good example for our fellow developers. Has anyone reading this not encountered code so bad it hurts the brain just to look at it? I've only been in this business eight years, but I have encountered code like this at *every single job I've worked at*.

I see the problem as two-fold: one, too many developers have not adopted the proven techniques and practices that lead to good code; and two, the "profession" of software development is so young and informal that most managers and business people do not respect it or even have a clue about it. In other words, the profession itself does not have yet the posture to "push back" at business to insist that code be written to a high standard of quality, and to insist that certain minimum processes be followed. Until some of us in that top 10% start standing up (in a nice way, of course) and saying, "No, I just won't do it that way," business will never get the message, and neither will our fellow developers.

Think of the "mature professions": accounting, law, medicine, electrical engineering, civil engineering, mechanical engineering. These professions have established and codified standards, practices, and ethics. These principles are taught in schools and enforced through peer pressure and even official sanction. Even though Sturgeon's Law still applies within any one of these professions (that is, 90% of accountants will suck when compared to the top 10% of accountants), the *general level* of quality and craftsmanship one finds in any one of these professions is going to be higher than what we find *generally* in the profession of software engineering.

I was inspired by Steve McConnell's 1999 book *After the Gold Rush: Creating a True Profession of Software Engineering* (Microsoft Press). This book argues that there is already underway an inevitable evolution of software development from a "craft," into what McConnell calls a "true profession" based on the principles of engineering. I will close this installment with a quote from *After the Gold Rush*:

"Arthur C. Clarke said that any sufficiently advanced technology is indistinguishable from magic. Software technology is sufficiently advanced, and the general public is mystified by it. The public does not understand the safety risks or financial risks posed by software products. As high priests of powerful magic, software developers need to use the technology wisely.

Engineering may be regarded as boring in some quarters but, boring or not, engineering is a better model for software than magic is. The engineering approach to design and construction has a track record of all but eliminating some of the most serious risks to public safety and of supporting some of the most elevating expressions of the human spirit. Concerned software developers have a responsibility to ensure that software rises to the level of engineering. In the short term, software engineering practice needs to be patched up enough to prevent further software disasters. In the long term, software engineering practice needs to be elevated to a level at which it can support the next generation of technological miracles.” (Page 63)

References

Steve McConnell, *After the Gold Rush: Creating a True Profession of Software Engineering*, (Microsoft Press, 1999)

###

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.