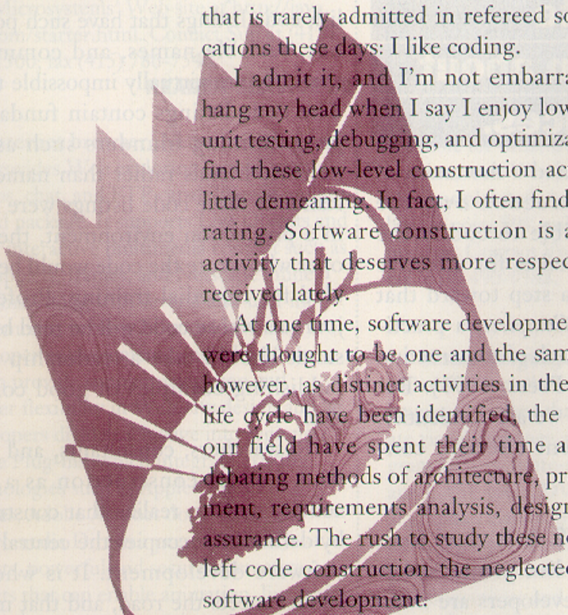
Who Cares About Software Construction?

Prospecting for progammer's gold.

Software practitioners are subjected to a barrage of advice about effective development practices. The search for effective practices — programmers' gold — can be almost as chancey as the search for the precious yellow metal itself. Some mediocre practices are overhyped and don't pan out; many valuable practices are buried under the hype heaped on other practices. This column aims to separate the gold from the ore by providing a practitioner's appraisal of past and present development practices. Future columns will take up practitioner-oriented topics ranging from "Whatever happened to information hiding?" to "The estimation story — defending unpopular estimates."
— *Steve McConnell*



I AM GOING TO ADMIT SOMETHING that is rarely admitted in refereed software publications these days: I like coding.

I admit it, and I'm not embarrassed. I don't hang my head when I say I enjoy low-level design, unit testing, debugging, and optimization. I do not find these low-level construction activities even a little demeaning. In fact, I often find them invigorating. Software construction is an important activity that deserves more respect than it has received lately.

At one time, software development and coding were thought to be one and the same. Over time, however, as distinct activities in the development life cycle have been identified, the best minds in our field have spent their time analyzing and debating methods of architecture, project management, requirements analysis, design, and quality assurance. The rush to study these newer areas has left code construction the neglected stepchild of software development.

Construction has also been neglected by researchers and writers because of the mistaken idea that, compared to other development activities, construction is a relatively mechanical process and presents little opportunity for improvement. Nothing could be further from the truth. Construction is not at all mechanical. Sometimes a formal architecture addresses only the top level of system decomposition, intentionally leaving design work for construction. Even program designs that are supposed to be detailed enough to provide for fairly mechanical coding will always have gaps — the coder usually designs part of the program, officially or otherwise.

The reduction in attention to code construction has been exacerbated by the treatment of coding as the dirtiest grunt work of development. An entry-level programmer in a large organization is typically assigned to code routines that have been specified and designed by someone higher up on the corporate ladder. After a few years, the programmer is promoted to architecture, requirements analysis, or project management. A few years after that, the former programmer may proclaim with pride that it has been years since he or she has written any code.

Code is often the only accurate description of the software available, so it is imperative that it be of the highest possible quality.

CENTRAL ROLE. The irony in this shift in focus is that construction is the only development activity that is guaranteed to be done. Right or wrong, you could assume requirements rather than analyzing them, you can shortchange architecture design, and you can abbreviate or skip system testing. But no matter how rushed or poorly planned your project is, you cannot skip construction. If there's going to be a program, there must be construction.

Because construction is the only activity that must be done, code is often the only accurate description of the software available, which makes it imperative that the source code be of the highest possible quality. Consistent application of detailed source-code techniques distinguishes a Rube Goldberg contraption from a polished, correct, and informative program. Such detailed techniques must be applied as the code is constructed — it is virtually impossible to retrofit the thousands of picky details that spell the difference between a maintainable program and a failure.

Continued on page 127

Editor:
Steve McConnell
Phantom Lake Engineering
PO Box 6922
Bellevue, WA 98008
smcconn@aol.com

Continued from page 128

Even when a program is developed using effective practices, construction typically makes up about 80 percent of the effort on small projects and 50 percent on medium projects. And, although the figures cited vary considerably, construction accounts for about 75 percent of the errors on small projects and 50 to 75 percent on medium and large projects. (This data is approximate for detailed design, coding, unit testing, and debugging. See, for example, *Programming Productivity* by Capers Jones, McGraw-Hill, 1986.)

Any activity that accounts for 50 to 75 percent of errors presents a clear opportunity for improvement.

Some commentators have suggested that, although construction errors account for a high percentage of total errors, these errors tend to be less expensive to fix than errors created during analysis and design. The implication is that construction errors are therefore less important.

Although it may be true that construction errors cost less to fix, this is misleading because the cost of *not* fixing them can be incredibly expensive. Gerald Weinberg reported 10 years ago that the three most expensive programming errors — each of which costs hundreds of millions of dollars — were one-line, coding-level mistakes ("Kill that Code!" *Infosystems*, Aug. 1983).

Little has changed since this report, as a perusal of any recent "Risks to the Public" section of *Software Engineering Notes* shows. Errors in single lines might be less expensive to fix than errors in analysis or design, but obviously this does not imply that detecting and correcting them should be a low priority.

OPPORTUNITY KNOCKS. What does this have to do with best practices?

Construction presents an unusually good opportunity to disseminate information about best practices in software engineering. Millions of programmers already doing construction can do it better. People without formal training have taught themselves to program in Pascal, Basic, C, and C++. Learning about better software construction allows these people to build on what they already know and gives them a foot in the door to other powerful development practices.

In 1990, the Computer Science and Technology Board stated that the biggest gains in software-development quality and productivity will come from codifying, unifying, and distributing existing knowledge about effective software-development practices ("Scaling Up: A Research Agenda for Software Engineering," *Communications of the ACM*, March 1990). The board concluded that software-engineering handbooks should play a key role in disseminating that knowledge.

Construction presents a good opportunity to disseminate information about best practices.

We have accumulated a mountain of research about construction over the years. It seems possible to codify the knowledge of what makes for effective construction. I took a step toward that end by writing *Code Complete*, a practitioner-oriented handbook of construction practices (Microsoft Press, 1993). But much work remains if we are to identify, validate, and document effective construction practices.

RETRAINING GURUS. It is clear from what I have seen on live projects that young, hotshot PC developers are making the same mistakes their elders made and learned to avoid 10 or 20 years ago. It is also clear from much of what has been written that the gray-haired software-engineering experts could learn some things from the young hotshots (after all, there are reasons why they're hotshots).

School districts put administrators back into the classrooms occasionally, and service companies put top executives back on customer-service duty. I have a dream in which prominent software-engineering experts sign up for construction roles on projects at modern companies like Microsoft, Borland, and Novell. During the last 10 years, an unbelievable amount has changed in the way that production software is developed, and a lot of those differences must be experienced on a live project to be appreciated. Maybe this dream can never be realized because it calls for too much individual financial sacrifice. Maybe it would seem like too much of a step backward for a software-engineering guru to take a construction job. But the industrywide benefit would be amazing. The insights that some of the wisest and brightest people in our field would gain from being back on the front lines of construction for awhile could invigorate the whole industry.

Publications like this one can help, too. Some professional journals still publish code listings that have such poor layout, variable names, and commenting that they are virtually impossible to read. The code listings contain fundamental programming blunders such as using numeric literals rather than named constants. If these code listings were created in a production environment, the developer who wrote the code would be scheduled for remedial training. Professional journals whose code will be read by thousands should take a leadership role in publishing examples of good construction.

Researchers, consultants, and writers who dismiss construction as a trivial activity need to realize that construction, by definition, occupies the central role in software development. It is where the rubber meets the road, and that makes it a uniquely productive area in which to focus our attention.

Construction is important and will remain so for the foreseeable future. The difference between good and bad construction can mean the difference between project success and failure. The difference between recognizing the key role that construction plays in software development and ignoring it can mean the difference between moving the industry forward and repeating the same old mistakes yet another time. ◆