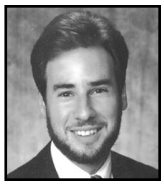

Steve McConnell

The Art, Science, and Engineering of Software Development

When interviewing candidates for programming jobs, one of my favorite interview questions is, "How would you characterize your approach to software development?" I give them examples such as carpenter, fire fighter, architect, artist, author, explorer, scientist, and archeologist, and I invite them to come up with their own answers. Some candidates try to second-guess what I want to hear; they usually tell me they see themselves as "scientists." Hot-shot coders tell me they see themselves as commandos or SWAT-team members. My favorite answer came from a candidate who said, "During software design, I'm an architect. While I'm designing the user interface, I'm an artist. During construction, I'm a craftsman. And during unit testing, I'm one mean son of a bitch!"

I like to pose this question because it gets at a fundamental issue in our field: What is the best way to think of software development? Is it science? Is it art? Is it craft? Is it something else entirely?

Software development *is* art. It *is* science. It is craft, fire fighting, archeology, and a host of other activities.

Two cultures or an ideal unmet?

We have a long tradition in the software field of debating whether computer programming is art or science. Thirty years ago, Donald Knuth began writing a seven-volume series, *The Art of Computer Programming*. The first three volumes stand at 2,200 pages, suggesting the full seven might amount to more than 5,000 pages. (If that's what the *art* of computer programming looks like, I'm not sure I want to see the *science*!)

People who advocate programming as art point to the aesthetic aspects of software development and argue that science does not allow for such inspiration and creative freedom. People who advocate programming as science point to many programs' high error rates and argue that such low reliability is intolerable—creative freedom be damned. In my view, both these views are incomplete and both ask the wrong question. Software development *is* art. It *is* science. It is craft, fire fighting, archeology, and a host of other activities. It is as many different things as there are different people programming. But the proper question is not "What *is* software development?" but rather "What *should* software development be?" In my opinion, the answer to that question is clear: Software development should be *engineering*. Is it? No. Should it be? Unquestionably, yes.

Beyond the buzzword

The dictionary definition of engineering is the application of scientific and mathematical principles toward practical ends. That is what most of us try to do, isn't it? We apply scientifically developed and mathematically defined algorithms, functional design methods, quality-assurance practices, and other practices to develop software products and services. As David Parnas points out, in other technical fields the engineering professions were invented and given legal standing so that customers could know who was qualified to build technical products ("Software Engineering: An Unconsummated Marriage," *Software Engineering Notes*, Nov. 1997). Software customers deserve no less.

Some people object to the idea that software development should be treated as engineering because they think "software engineering" is just a buzzword; they argue that no core body of knowledge can be identified as "software engineering." Thirty years ago when the first NATO conference on software engineering was held, this statement was undoubtedly true. The first paper on structured design had not yet been published. Neither had the first papers on inspections, measurement-based estimation, or the high performance variability among individual program

Continued on page 118

EDITOR: Steve McConnell • Construx Software Builders • stevemcc@construx.com

Continued from page 120

mers. Ten years would pass before the publication of the first books on structured design, software system specification, and metrics. Online systems were controversial, and the first readily available GUI interfaces were still more than a decade away. Whatever core body of knowledge could have been defined at that time would have been at least 50 percent obsolete within 10 years.

Today, "software engineering" is still thrown around as a buzzword more often than not. That's unfortunate. But the fact that the term is abused does not mean it has no legitimate meaning. Software development has come a long way in 30 years. We still do not have an absolutely stable core body of knowledge, and knowledge related to specific technologies will never be very stable, but we do have a body of knowledge that is stable enough to call software engineering. That core includes practices used in requirements development, functional design, code construction, integration, project estimation, cost–benefit trade-off analysis, and quality assurance of all the rest.

Many core elements have not yet been brought together in practically oriented textbooks or courses, and in that sense our body of knowledge is still fragmented and under construction. But the basic knowledge about how to perform each of these practices is available—in journal articles, conference papers, and seminars. The pioneers of software engineering have already blazed the trails and surveyed the land. Now the software engineering settlers need to get to work, turning the trails into roads and developing the rest of the education and accreditation infrastructure.

Each software program must strike a balance between functionality and reliability appropriate to its purpose.

Engineering trade-offs

Some people think that treating software development as engineering means we'll all have to use formal methods, which both common sense and experience tell us are overkill for many projects. Others say that commercial software is too dependent on changing market conditions to permit careful, time-consuming engineering. These objections are based upon narrow, and I think, mistaken, ideas of engineering. Engineering is the application of scientific principles toward *practical* ends. If the engineering isn't practical, it isn't good engineering.

Treating software as engineering makes clearer the idea that different development goals are appropriate for different projects. When a building is designed, the construction materials must suit the building's purpose. I can build a large equipment shed to store farming vehicles from thin, uninsulated sheet metal. I wouldn't build a house the same way. But even though the house is sturdier, warmer, and likely to last longer, we don't refer to the shed as having lower quality than the house. The shed has been designed appropriately for its intended purpose. If it had been built as robustly as a house, we might even criticize it for being "overengineered"—a judgment that the designers wasted resources in building it.

Similarly, each software program must strike a balance between functionality and reliability appropriate to its purpose. We don't want to pay $5,000 for a word processor, nor do we want one that crashes every 15 minutes. Thus, the shrink-wrap software industry is constantly optimizing an equation that includes the variables of time to market, reliability, functionality, and cost.

In most cases, these optimizations are business decisions rather than moral decisions. There is no inherent "right" or "wrong" quality level for software independent of the specific software package being created. The "right" materials depend on a specific building's purpose; the right reliability depends on the specific software's purpose. When safety is involved, the optimizations become moral decisions, but such calculations are not unique to software. Drug manufacturers weigh the benefits of their drugs against their side effects. Bridge designers and aeronautical engineers consider both safety and cost. Whether the calculations involve functionality, time, money, or human safety, striking the right balance in a deliberate, informed way is properly the domain of engineering—applying scientific principles toward practical ends. Many current software projects strike that balance poorly, or with little awareness that trade-offs even exist. An engineering approach can only help them.

Roles in software engineering

Another objection is that treating software development as engineering will take all the fun out of it. Software development in many quarters is a fast-and-loose craft, and many fast-and-loose craftsmen are making fantastic incomes without knowing much about effective development practices. Only a prophet could know how long the public will tolerate computer programmers who make doctors' salaries without comparable training and education, but I predict that one fallout of Year 2000 problems will be public insistence on regulating software developers.

All other engineering fields are self-accredited and self-regulated; software developers will either voluntarily swallow the medicine of software as engineering or we will have it forced down our throats. I know which way I prefer to take my medicine.

As roles in the field settle out, I won't be surprised to see greater coordination between software engineering and other engineering disciplines. Some stu-

dents will major in electrical engineering or aeronautical engineering with an emphasis in software. They will lead the software parts of their engineering projects, and their classmates who choose pure electrical engineering or aeronautical engineering will not. Some students will attain degrees in general software engineering.

A major distinction between software engineering and other kinds of engineering is that software is so labor intensive that a significant amount of engineering energy must be focused on *project* goals in addition to *product* goals—on the means to the ends as well as the ends themselves. Other kinds of engineers choose components to be used in the structures they build. Software engineers choose both the tools used to build the software and the components to be used within the software itself.

The fact that some software must be produced by trained, accredited engineers doesn't imply the same for all software. I can fix a broken railing on my stairs without a civil engineering degree. I can mix a gin and tonic without a chemical engineering degree. But can I build a second story on my house? No. My local government will require me to have an engineer sign the building plans. We should eventually see a similar stratification of software jobs into amateurs, skilled craftsmen, and engineers.

The fact that some software must be produced by trained, accredited engineers doesn't imply the same for all software.

THE RIGHT QUESTIONS

Once we stop asking the trick question "*Is* software development engineering?" and start asking the real question "*Should* software development be engineering?" we can start answering really important questions: What is software engineering's core body of knowledge? How should software engineers be trained? How should software engineers be certified? And, perhaps the hardest to answer, *How long will it take for all this to happen?* ❖