CHAPTER FOUR

Software Engineering, Not Computer Science

A scientist builds in order to learn; an engineer learns in order to build.

FRED BROOKS

When interviewing candidates for programming jobs, one of my favorite interview questions is, "How would you describe 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. When 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?

"Is" vs. "Should"

We have a long tradition in the software field of debating whether software development 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 ever 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. Both these views are incomplete and both ask the wrong question. Software development is art. It is science. It is craft, archeology, fire fighting, sociology, and a host of other activities. It is amateurish in some quarters, professional in others. It is as many different things as there are different people programming. But the proper question is not "What *is* software development currently?" but rather "What *should* professional software development be?" In my opinion, the answer to that question is clear: Professional software development should be engineering. Is it? No. But should it be? Unquestionably, yes.

Engineering vs. Science

With only about 40 percent of software developers holding computer science degrees and practically none holding degrees in software engineering, we shouldn't be surprised to find people confused about the difference between software engineering and computer science. The distinction between science and engineering in software is the same as the distinction in other fields.[1] Scientists learn what is true, how to test hypotheses, and how to extend knowledge in their field. Engineers learn what is true, what is useful, and how to apply well-understood knowledge to solve practical problems. Scientists must keep up to date with the latest research. Engineers must be familiar with knowledge that has already proven to be reliable and effective. If you are doing science, you can afford to be narrow and specialized. If you are doing engineering, you need a broad understanding of all the factors that affect the product you are designing. Scien-

tists don't have to be regulated because they are chiefly accountable to other scientists. Engineers do have to be regulated because they are chiefly accountable to the public. An undergraduate science education prepares students to continue their studies. An undergraduate engineering education prepares students to enter the workforce immediately after completing their studies.

Universities award computer science degrees, and they normally expect their computer science students to obtain software development jobs in which they will immediately begin solving real-world problems. Only a small fraction of computer science undergraduates go on to graduate school or research environments in which they are advancing the state of knowledge about software or computers.

This puts computer science students into a technological no-man's land. They are called scientists, but they are performing job functions that are traditionally performed by engineers, without the benefit of engineering training. The effect is roughly the same as it would be if you assigned a physics Ph.D. to design electrical equipment for commercial sale. The physicist might understand the electrical principles better than the engineers he is working with. But his experience in building equipment is in creating prototypes that are used to advance the state of knowledge in a laboratory. He does not have experience or training in designing rugged, economical equipment that provides practical solutions in real-world settings. We would expect the equipment designed by the physics Ph.D. to work, but perhaps to lack some of the robustness that would make it usable or safe outside a laboratory. Or the equipment might use materials in a way that's acceptable for a single prototype but extravagantly wasteful when units are manufactured by the thousands.

Situations resembling this simple physics example occur literally thousands of times each year in software. When workers educated as computer scientists begin working on production systems, they often design and build software that is too frail for production use, or that's unsafe. They focus narrowly and deeply on minor considerations to the exclusion of other factors that are more important. They might spend two days hand-tuning a sorting algorithm instead of two hours using a code library or

copying a suitable algorithm from a book. The typical computer science graduate typically needs several years of on-the-job training to accumulate enough practical knowledge to build minimally satisfactory production software without supervision. Without appropriate formal education, some software developers work their entire careers without acquiring this knowledge.

The lack of professional development isn't solely the software developer's failure. The software world has become a victim of its own success. The software job market has been growing faster than the educational infrastructure needed to support it, and so more than half the people holding software development jobs have been educated in subjects other than software. Employers can't require these software retreads to obtain the equivalent of an undergraduate engineering degree in their off hours. Even if they could, most of the courses available are in computer science, not software engineering. The educational infrastructure has fallen behind industry's needs.

Beyond the Buzzword

Some people think that "software engineering" is just a buzzword that means the same thing as "computer programming." Admittedly, "software engineering" has been misused. But a term can be abused and still have a legitimate meaning.

The dictionary definition of "engineering" is the application of scientific and mathematical principles toward practical ends. That is what most programmers try to do. We apply scientifically developed and mathematically defined algorithms, functional design methods, quality-assurance methods, 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.[2] Software customers deserve no less.

Some people think that treating software development as engineering means we'll all have to use formal methods—writing programs as mathe-

matical proofs. Common sense and experience tell us that that is overkill for many projects. Others object that commercial software is too dependent on changing market conditions to permit careful, time-consuming engineering.

These objections are based upon a narrow and mistaken idea of engineering. Engineering is the application of scientific principles toward *practical* ends. If the engineering isn't practical, it's bad engineering. Trying to apply formal methods to all software projects is as bad an idea as trying to apply code-and-fix development to all projects.

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 and warmer, we wouldn't refer to the shed as being inferior to the house in any way. The shed has been designed appropriately for its intended purpose. If it had been built the same way as a house, we might even criticize it for being "over-engineered"—a judgment that the designers wasted resources in building it and that it actually isn't well engineered.

In software, a well-run project can be managed to meet any of the following product objectives:

- Minimal defects
- Maximum user satisfaction
- Minimal response time
- Good maintainability
- Good extendibility
- High robustness
- High correctness

Each software project team should define the relative importance of each characteristic explicitly, and then the project team should conduct the project in a way that achieves its objectives.

Software projects are different from engineering projects that use physical materials. In other kinds of engineering, the cost of materials can contribute 50 percent or more of the total project cost. Some engineering companies report that they automatically regard projects with labor constituting more than 50 percent of project cost as high risk.[3] On a typical software project, labor costs can contribute almost 100 percent of the total project cost. Most engineering projects focus on optimizing *product* goals; design costs are relatively insignificant. Because labor cost makes up such a large part of total lifetime software costs, software projects need to focus more on optimizing *project* goals than other kinds of engineering do. So, in addition to working toward product objectives, a software team might also work to achieve any of the following project objectives:

- Short schedule
- Predictable delivery date
- Low cost
- Small team size
- Flexibility to make mid-project feature-set changes

Each software project must strike a balance among various project and product goals. We don't want to pay $5,000 for a word processor, nor do we want one that crashes every 15 minutes.

Which of these specific product and project characteristics a project team emphasizes does not determine whether a project is a true "software engineering" project. Some projects need to produce software with minimal defects and near-perfect correctness—software for medical equipment, avionics, anti-lock brakes, and so on. Most people would agree that these projects are an appropriate domain for full-blown software engineering. Other projects need to deliver their software with adequate reliability but with low costs and short schedules. Are these properly the domain of software engineering? One informal definition of engineering is "doing for a dime what anyone can do for a dollar." Lots of software projects today are doing for a dollar what any good software engineer

could do for a dime. Economical development is also the domain of software engineering.

Today's pervasive reliance on code-and-fix development—and the cost and schedule overruns that go with it—is not the result of a software engineering calculation, but of too little education and training in software engineering practices.

The Right Questions

Software development as it's commonly practiced today doesn't look much like engineering, but it could. Once we stop asking the wrong question—"What *is* software development currently?"—and start asking the right question—"*Should* professional software development be engineering?"—we can start answering the really interesting questions. What is software engineering's core body of knowledge? What needs to be done before professional software developers can use that knowledge? How big is the payback from practicing software development as an engineering discipline? What are appropriate standards of professional conduct for software developers? For software organizations? Should software developers be regulated? If so, to what extent? And, perhaps the most interesting question of all: What will the software industry look like after all these questions have been answered?

Notes

1. For much of this discussion, I am indebted to David L. Parnas, especially for his paper, "Software Engineering Programmes Are Not Computer Science Programmes," *IEEE Software,* November/December 1999.
2. Parnas, David L., "Software Engineering: An Unconsummated Marriage," *Software Engineering Notes,* November 1997.
3. Baines, Robin, "Across Disciplines: Risk, Design, Method, Process, and Tools," *IEEE Software,* July/August 1998, pp. 61–64.