Why Software Engineering Is Not B.S.

By Charles Connell

I teach software engineering, and I know what "real" computer scientists think about the subject: It is soft. It is not quantitative. It is little different from sociology (gasp), since it partly concerns the behavior of people in groups. Real computer scientists usually prefer topics such as cellular automata, undecidability, lambda calculus, probabilistic factoring, and queuing theory. These disciplines have satisfying mathematically provable results — and sometimes an equally satisfying proof that there is no proof.

Software engineering, on the other hand, is concerned with "creating high-quality software systems in an efficient and predictable manner" — without firm definitions of what *high-quality*, *efficient* and *predictable* mean. The results of the study of software engineering are heuristics for success that ought to work, most of the time, if nothing unusual happens. The methods of the field include empirical learning, policies and procedures, and (gasp again) people skills.

The feeling of many computer scientists toward software engineering is summed up by the following true story. I attended graduate school in computer science after working in the software industry for several years. I signed up for the only course on software engineering and looked forward to learning some innovative programming techniques to augment my practical experience. On the first day of class, the professor announced, "Software engineering is B.S.. There is nothing to teach about it. So we are going to study Unix internals instead." And that was that. I learned nothing about software engineering during four years at graduate school because the professor who taught the course believed the subject to be empty.

While this story is anecdotal, objective proof of software engineering's second-class status within computer science can be found in computer science's most prestigious publication: *Journal of the Association of Computing Machinery*. Over the last five years, there were no articles about software engineering in this important publication.

But software engineering is not B.S.. In fact, it may be more worthy of serious study than some traditional computer science topics. Software engineering is not B.S. for three reasons: software engineering is simply at an early stage of development; software engineering happens to be hard; and the gains to be had from any new knowledge about this subject are huge.

Software Engineering Is At An Early Stage

In the 16th century, chemistry could barely be considered a science at all. The best statement chemistry could make about the composition of the material world was that the universe is composed of four elements: earth, air, fire, and water. But this statement had no predictive power, so was nearly useless. Moreover, chemistry's primary goal for a long time had been to change an inexpensive substance into gold. But all the greatest minds in the field had failed to do so after 1000 years of effort. The sum of all the successes in the field was minute. At the time, we might reasonably have concluded that chemistry was a discipline not worth our time and

effort. Students would wisely have been cautioned to choose another field of study with better prospects, perhaps astrology or blood letting.

In fact, chemistry is one of the outstanding success stories in the human search for knowledge because interested scientists persevered through lean times and because a few key discoveries opened the doors to many more. Software engineering is in the same position. The field is young. The subject domain is vast and difficult to catalog. The goal (creating high-quality software in an efficient, predictable manner) is lofty. But new sciences always flail around with qualitative, short-lived results in their early years. The newness of a field does not mean it is less important or less worthy of our best study. The newness of software engineering, to the contrary, means it is ripe with opportunities for seminal discoveries. And with today's accelerated pace of discovery, we can expect software engineering will take fewer than 500 years to mature.

Software Engineering Is Hard

Progress in software engineering is slower than everyone connected to software development would like. By progress, I mean tools and methods that consistently are helpful when creating large, complex software systems. There are many tools and methods that help in one context, but gum up the works in another. (Frederick Brooks likens software development to a tar pit where great beasts thrash violently and quickly become entangled.) But why have other computer science topics shown relative mastery over their problem domains — analysis of algorithms and compiler design for example — while software engineering lags behind? The reason is software engineering is more difficult.

Analysis of algorithms, compiler design, programming language semantics, complexity theory, cryptography, and other core computer science disciplines concern topics that are constrained and numerable. The problems, and acceptable solutions to them, are defined precisely. This is one of the hallmarks of these subjects. Ill-defined problems or unclear solutions are rejected as being unacceptable. Problems, or avenues of inquiry, that lead nowhere or to vague results, are excluded from the research program. This practice of narrowing the problem statement, so the results are in an acceptable form, is practiced by many sciences and allows them to move forward.

The precision of traditional computer science has a drawback, however. The problems that are solved are those that are amenable to precise solutions. These problems are, by definition, tightly defined, with no mushy statements or an unacceptably high number of variables. In short, these problems are easier than real-world problems that aren't conveniently narrowed. Software engineering happens to be just such a real-world problem.

The relationship between core (mathematical) computer science and software engineering is similar to the relationship between basic physics and meteorology. While the physical principles related to weather formation (heating and cooling, evaporation and condensation, air pressure) are well understood, the science of weather forecasting is far less advanced. Meteorology concerns the complex interactions of many simple phenomena. Meteorologists now can make reasonably accurate predictions a few days into the future. Forecasts beyond that are shaky. This is the case even though the individual forces that create the weather can be calculated accurately

by college freshmen in Physics 101. Software engineering is similar. In isolation, the items that make up a large software system are quite simple. The individual algorithms, file formats, and parameter passing mechanisms of a complex system are likely trivial. Mixed together by the hundreds or thousands, however, they are extremely difficult to control and their interactions are hard to predict.

The problems to be solved by software engineers are defined by the real world — creating a specific piece of software to route medical records in a specific hospital, for example. In general, the problems cannot be constrained significantly by limiting the problem statement. We cannot make the medical records or the hospital go away. Software engineers face the same task as meteorologists — making sense of a vast number of interactions between simple events. This is a hard problem. Unlike weather forecasters, software engineers do have the luxury of changing some variables to effect a different outcome. (Hire more people; get a different compiler.)

This is not much comfort, though, when the interactions are so complicated that we don't know what to change.

None of this should be taken as a knock on traditional computer science topics. These topics are important and worthy of sustained investigation. I have spent many happy hours studying these subjects and wrote my master's thesis on one of them. But software engineering is, qualitatively and quantitatively, a harder challenge.

The Gains Will Be Huge

The amount of time, effort, and money spent on software development and software use is immense. Software is now central to, or becoming central to, nearly everything we do — money and banking, medical care, public transportation, government operations, human communication, education at every level, and so on. Human endeavors are impacted not just by explicit use of software (sending someone an Instant Message on AOL) but also by embedded software systems (stepping on a train with computerized instruments, or turning on a light connected to a computer-controlled power grid). Organizations of all sizes and types create software, often in hidden ways. Virtually every large company and organization develops software systems for internal use to help run their business. Many products, such as cell phones, that are not strictly software products contain significant software components. And, of course, companies in the software business expend all their time and effort planning, designing, and creating software.

Layered on top of the ubiquity of software is the fact that, so far, people are lousy at making it. Many studies have shown about half of all software projects are cancelled during creation or abandoned when finished. Cost and time overruns, terrible quality, non-acceptance by users, and obsolescence before completion are just a few of the problems that plague software development. For perspective, imagine our reaction if half of all housing starts failed or were unlivable when finished. We certainly would perceive a major crisis in the housing industry. This is precisely the current state of affairs for software development.

If the study of software engineering helps us improve, by even a small amount, our ability to create software, the entire field justifies its existence. Suppose some new findings in software

engineering improve the efficiency or quality of worldwide software development by just 10 percent. Given the pervasiveness of software, this would have profound economic and human impact. An improvement just in the way we create certain critical software systems, such as air traffic control or the Internet backbone, also would justify all the efforts expended in this field.

In other words, software engineering matters because software matters. Software controls significant portions of many human activities, and this centrality will grow. Yet we currently do a bad job of writing it. Any gain in the efficiency, predictability or quality with which we create software will have far-reaching effects on our lives. The study of software engineering is not the scientific dung heap. In fact, given the growing role of software, both explicit and embedded, in our world, it is hard to think of a more worthy field of inquiry.

Charles Connell is president of CHC-3 Consulting, teaches software engineering at Boston University, and writes frequently on computer topics. He can be reached at www.chc-3.com.