Why software developers should be licensed

Over the last few decades, software has been replacing mechanical, electrical and electronic components in traditional engineering products. It has also become a critical component in medical devices, chemical plants, buildings and aircraft. Software is regularly used to design critical products, the effectiveness and safety of which can depend on the correctness of that software.

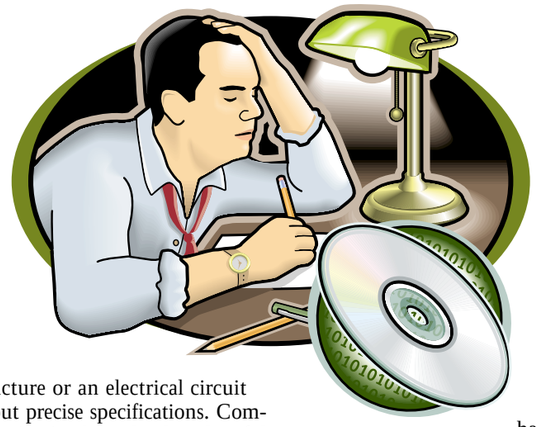
by David Parnas, PhD, P.Eng.

The ongoing debate about who is qualified to develop critical software deals with two distinct issues: First, under what circumstances should a software developer be required to be licensed or certified before taking responsibility for a software product? Second, what knowledge should be required of those who are licensed or certified to develop critical software? Recalling that responsible professionals must assure that their products are fit for their intended use, we assume that a qualified developer must be prepared to analyze both the software and the environment in which it functions.

What most engineers don't know about software

Many engineers assume that the primary requirement for software development is familiarity with the software tools that will be used in the project. They describe their competence by listing the languages, platforms and tools that they have used. However, there are also engineering principles and mathematical concepts that are important in software development. Here are some that are essential in developing critical software:

- *Discrete mathematics.* This plays much the same role in software development that differential and integral calculus and linear algebra play in traditional engineering. This area of mathematics is usually given short shrift in traditional engineering programs.
- *Programming using precise specifications.* The ability to read, write *and satisfy* precise specifications is the core of every engineering discipline. No engineer would develop an engine component,



a structure or an electrical circuit without precise specifications. Computer scientists have developed useful specification techniques, but they are not widely used. Traditional engineering specification techniques do not suffice for software. Consequently, most software is developed using vague specifications.

- *Structure, documentation and analysis of software.* Engineers in traditional disciplines are aware of the importance of documenting their work. Unfortunately, because programs are represented by strings of symbols, developers often regard software as "self-documenting" and are satisfied with ad hoc documents. Such documentation has repeatedly been found to be inadequate for the large programs that we use today. Computer scientists have identified some clear design and documentation principles for software "architecture." Some of this is taught in computer engineering programs, but often the treatment is shallow and the necessary mathematical tools are avoided. Engineers are also aware of the importance of careful mathematical analysis of any proposed design. Using mathematics, one can systematically determine critical characteristics of structures or circuits. Analogous techniques are available for software, but most traditionally educated engineers are not familiar with them.
- *Design and selection of computer algorithms and data structures.* Algorithms are the language-independent "core" of computer programs. There are important design and analysis methods for algorithms that are independent of the programming language. Because of time limitations, programming courses tend to spend most of their time on language grammar rather than algorithm design, and most engineers have a limited understanding of the analysis of algorithms.
- *Design and selection of programming languages.* Most engineers have learned a few programming languages, but few have been introduced to the wide variety of available languages, or the criteria to use when selecting one.
- *Designing concurrent and real-time software.* Methods for design and analysis of real-time and interactive software (including useful structuring, synchronization and scheduling techniques, as well as deadlock prevention methods) have been developed over the past 40 years, but most engineering programs don't have time to teach them.
- *Numerical computation.* Many engineers use software to perform numerical computations. Most know that the results are only approximate, but they assume that the precision is best improved by using higher precision arithmetic. They may use a numerically poor method when a more accurate method is available, and are unlikely to know how to estimate bounds on the error in their results.
- *Continuous and discrete optimization methods.* Developers are often required to write programs that "find the best" set of parameters for a design. Systematic optimization algorithms for both discrete and continuous spaces have been developed by mathematicians and computer scientists. Few engineers would have been exposed to these and many use ad hoc methods where better methods are available.
- *Performance analysis of computer systems (queuing theory and queuing networks).* Engineers who design structures know how to determine such attributes as the maximum permissible load. When we design a circuit, we can calculate safe power levels. But with software, we usually depend on "cut and try" approaches and experience surprise failures. There are well understood analytical and simulation approaches to estimate performance characteristics before building software, but these are rarely taught in traditional engineering programs.
- *Computer networks.* There is a large and interesting body of knowledge on how to build reliable networks with predictable performance, but most engineers have had no systematic exposure to the issues, problems or solutions.
- *Computer security.* With the advent of both local and large-scale computer networks, the problem of preventing unauthorized use or denial of service to authorized users has become serious. Logical, cryptographic and structural approaches to these problems have been developed, but are rarely taught in traditional engineering programs.
- *Design of human computer interfaces.* While some engineers have had exposure to engineering ergonomics, the computer has added many concerns that are not traditionally dealt with in such courses.

What many computer scientists don't know

Graduates of computer science (CS) programs might be expected to know much more about programming methods, programming languages, operating systems and software development tools than graduates of traditional engineering programs. However, there are two reasons for not assuming that a particular CS graduate is qualified to develop critical software products. First, many topics that could be very important when writing critical software, knowledge that engineers take for granted, are not generally taught in CS programs. Second, standardization and accreditation are foreign concepts in CS. There is incredible variation between CS programs and no topic is covered in depth by all of them.

Most computer science programs are designed assuming that what is important to graduates is an understanding of what goes on inside the computer. This allows graduates to "build the thing right," but it

(continued on next page)

(continued from previous page)

does not help them to "build the right thing." To build the right thing, a designer must also be able to analyze aspects of what goes on outside the computer. It is possible to get a computer science degree without taking a course in physics or chemistry. Analysis of the systems that interact with the computer is not discussed in most CS programs. Here are some gaps in the education of typical computer scientists:

- *Mathematics.* The use of math is essential for developing trustworthy software and precise documentation. Many of the bugs that we encounter in today's software could have been avoided if simple classical mathematics had been used during development. Mathematics is the core of CS. Many CS pioneers were mathematicians and many CS departments were spin-offs of mathematics departments. In the 1960s, applied mathematics, particularly numerical methods, was central to CS. However, CS programs have changed radically: One can now get a degree in CS without a single course on numerical methods, and earn a PhD in CS without learning calculus. Even in mathematically oriented CS programs, CS graduates may not learn linear algebra, differential equations, statistical methods and other mathematics taught to engineering students. In CS programs, emphasis has shifted from applied mathematics to discrete mathematics, emphasizing topics that would have once been called theoretical. Consequently, a CS graduate may not be qualified to work on many important applications. Even such basic tasks as finding a solution to a set of linear equations can be done badly by people who have not been taught how it should be done. Moreover, mathematics in CS programs is often taught independently of "practical programming" courses. Consequently, CS graduates often do not know how to use the mathematics that they know.
- *Engineering basics.* Computer science programs generally do not cover traditional control systems, computerized (digital) control systems, thermodynamics and heat transfer, or electricity and magnetic fields. Some CS programs do not require even one course in chemistry or physics. Computer science programs also omit such fundamental topics in communication as information theory, coding and treatment of noisy data.
- *Software science.* Listed earlier were a number of principles for software design that are not traditionally taught to engineers. Surprisingly, a CS graduate may also lack preparation in many of these areas. CS students have more freedom to choose what courses they take and CS instructors have greater leeway in what they teach than is usual in engineering. Most CS graduates learn surprisingly little in the areas discussed above and one can never be sure that they understand a particular topic.

Today's software professionals

Our newspapers are full of statistics indicating a severe shortage of software professionals. We also hear stories about the poor quality of software. Airplanes crash and patients die because of errors in software. Technicians and users spend hours on helplines to deal with many small "glitches" that make us all less productive.

Strangely, when discussing the quality of software and the quantity of professionals, we rarely discuss the qualifications of the professionals. But poorly qualified personnel are likely to develop poor quality software and there is no better job creation mechanism than a poor software developer. By creating software with many bugs, poor documentation and inadequate functionality, a "productive" but bad programmer can create several jobs a year.

Because there has been such a shortage of development personnel, many people have entered the profession through "side doors." In addition to CS graduates and traditionally educated engineers, we find mathematicians, physicists, philosophers, English majors and historians writing software. Their education is completely inadequate for the work they do. They rely on their intuition and a few technological short courses on particular tools.

What should be done?

Neither CS graduates nor professional engineers educated in the traditional disciplines can be assumed to be qualified to develop critical software. We need engineers who understand basic engineering principles, but who also have the necessary specialized knowledge to develop software intensive products. We also need other software specialists who will develop software tools, support systems and other products that may require a deeper knowledge of programming and less knowledge of basic engineering topics. Licensing and certification is essential in both cases.

Many traditionally educated engineers may (unknowingly) be practising outside of their area of competence because software engineering has not yet been widely recognized as a distinct discipline within engineering. Computer scientists may or may not have the required knowledge, depending on their interests and the interests of the professors who designed their education. Licensing and certification are needed to identify those who have the appropriate educational background. Licensing is essential for professionals who deal directly with the public or who are responsible for critical projects. Certification will be useful to those who will be part of a team and report to more senior professionals who are able to evaluate their work. ◆

David Parnas, PhD, P.Eng., is director of the software engineering program at McMaster University in Hamilton. He was a member of the PEO task group that developed criteria to enable PEO to license as professional engineers those individuals working in software but educated in another area of engineering. The task group also defined the areas of software development that require licensing to safeguard the public. For further information, see PEO's brochure *Licensing as a Professional Engineer, Answers to Frequently Asked Questions for Software Practitioners* at www.peo.on.ca. Printed copies of the brochure are also available by calling PEO at 416-224-1100 or 1-800-339-3716.

Viewpoint is a forum for opinion on current engineering issues. Ideas expressed do not necessarily reflect PEO opinion or policy, nor does the association assume responsibility for the oipnions expressed. Feedback from readers is invited.