DOI:10.1145/1831407.1831419

David L. Parnas

Inside Risks

Risks of Undisciplined Development

An illustration of the problems caused by a lack of discipline in software development and our failure to apply what is known in the field.

THE BRANCHES OF engineering (such as civil, electrical, and mechanical), are often referred to as disciplines for good reason. Associated with each specialty is a set of rules that specify:

- checks that must be made;
- properties that must be measured, calculated, or specified;
- documentation that must be provided;
- design review procedures;
- tests that must be carried out on the product; and
- product inspection and maintenance procedures.

Like all professional education, engineering education is designed to prepare students to meet the requirements of the authorities that regulate their chosen profession. Consequently, most graduates are taught they must carry out these procedures diligently and are warned they can be deemed guilty of negligence and lose the right to practice their profession if they do not.

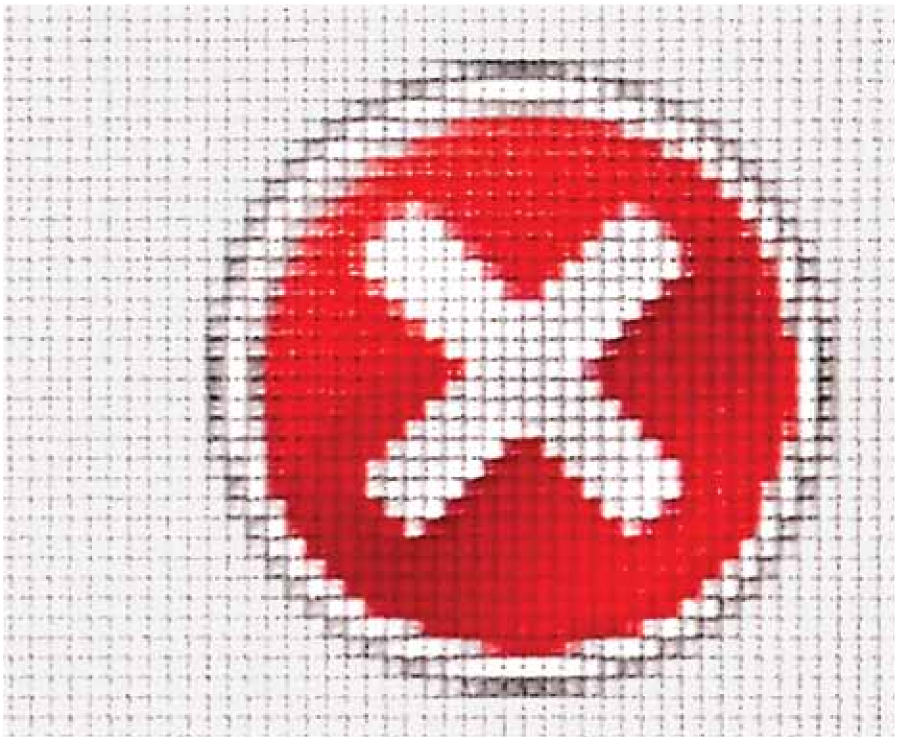
Because they are preparing students for a career that can last many decades, good engineering programs teach fundamental principles that will be valid and useful at the end of the graduate's career. Engineering procedures are based on science and mathematics; and graduates are expected to understand the reasons for the rules, not just blindly apply them.

These procedures are intended to assure that the engineer's product:

- will be fit for the use for which it is intended;
- will conform to precise stable standards;
- is robust enough to survive all foreseeable circumstances (including incorrect input); and
- is conservatively designed with appropriate allowance for a margin of error.

In some areas, for example building and road construction, the procedures are enforced by law. In other areas, and when engineers work in industry rather than selling their services directly to the public, employers rely on the professionalism of their employees. Professional engineers are expected to know what must be done and to follow the rules even when their employer wants them to take inappropriate shortcuts.

Anyone who observes engineers at work knows that exercising due diligence requires a lot of "dog work." The dull, but essential, work begins in the



PHOTOGRAPH BY BRANDON SHIGETA

design phase and continues through construction, testing, inspection, commissioning, and maintenance. Licensed engineers are given a unique seal and instructed to use it to signify the acceptability of design documents only after they are sure the required analysis has been completed by qualified persons.

Real-World Experience

Recent experiences reminded me that the activity we (euphemistically) call software engineering does not come close to deserving a place among the traditional engineering disciplines. Replacing an old computer with a newer model of the same brand revealed many careless design errors—errors that in all likelihood could have been avoided if the developers had followed a disciplined design process. None of the problems was safety critical, but the trouble caused was expensive and annoying for all parties.

My "adventure" began when the sales clerk scanned a bar code to initiate the process of creating a receipt and registering my extended warranty. There were three codes on the box; not surprisingly, the sales clerk scanned the wrong one. This is a common occurrence. The number scanned bore no resemblance to a computer serial number but was accepted by the software without any warning to the clerk. The nonsense number was duly printed as the serial number on my receipt. My extended warranty was registered to a nonexistent product. I was billed, and no problem was noted until I phoned the customer care line with a question. When I read the serial number from the receipt, I was told that I had purchased nothing and was not entitled to ask questions. After I found the correct number on the box, I was told that my computer was not yet in their system although a week had passed since the sale.

Computer science students are not taught to work in disciplined ways. In fact, the importance of disciplined analysis is hardly mentioned.

Correcting the problem required a trip back to the store and tricking the company computer by returning the nonexistent machine and buying it again. In the process, my name was entered incorrectly and I was unable to access the warranty information online. After repeatedly trying to correct their records, the help staff told me it could not be done.

A different problem arose when I used the migration assistant supplied with the new computer to transfer my data and programs to the new machine. Although the description of the migration assistant clearly states that incompatible applications will be moved to a special directory rather than installed, a common software package on the old machine, one that was not usable or needed on the new one, was installed anyway. A process began to consume CPU time at a high rate. Stopping that process required searching the Internet to find an installer for the obsolete product.

The next problem was an error message informing me that a device was connected to a USB 1.1 port and advising me to move it to a USB 2.0 port. My new computer did not have any 1.1 ports so I called the "care" line for advice. They had no list of error messages and could not guess, or find out, which application or component of their software would issue such a message or under what conditions it should be issued. They referred the problem to developers; I am still waiting for a return call.

These incidents are so petty and so commonplace that readers must wonder why I write about them. It is precisely because such events are commonplace, and so indicative of lack of discipline, that such stories should concern anyone who uses or creates software.

As early as the late 1950s, some compilers came with a complete list of error messages and descriptions of the conditions that caused them. Today, such lists cannot be found. Often, when reviewing a system, I will pick a random message or output symbol and ask, "When does that happen?" I never get a satisfactory answer.

There are methods of design and documentation that facilitate checking that a programmer has considered all possible cases (including such undesired events as incorrect input or the need to correct an earlier transaction) and provided appropriate mechanisms for responding to them. When such methods are used, people find serious errors in software that has been tested and used for years. When I talk or write about such methods, I am often told by colleagues, experienced students, and reviewers that, "Nobody does that." They are right—that's the problem!

Much of the fault lies with our teaching. Computer science students are not taught to work in disciplined ways. In fact, the importance of disciplined analysis is hardly mentioned. Of course, just telling students to be diligent is not enough. We need to:

- teach them what to do and *how* to do it—even in the first course;
- use those methods ourselves in *every* example we present;
- insist they use a disciplined approach in *every* assignment in *every* course where they write programs;
- check they have inspected and tested their programs diligently, and
- test their ability to check code systematically on examinations.

Many of us preach about the importance of determining the requirements a software product must satisfy, but we do not show students how to organize their work so they can systematically produce a requirements specification that removes all user-visible choices from the province of the programmer.

Some of us advise students to avoid dull work by automating it, but do not explain that this does not relieve an engineer of the responsibility to be sure the work was done correctly.

Innovation and Disciplined Design

It has become modish to talk about teaching creativity and innovation. We need to tell students that inventiveness is not a substitute for disciplined attention to the little details that make the difference between a product we like and a product we curse. Students need to be told how to create and use checklists more than they need to hear about the importance of creativity.

Even sophisticated and experienced purchasers do not demand the documentation that would be evidence of disciplined design and testing.

It is obviously important to give courses on picking the most efficient algorithms and to make sure that students graduate prepared to understand current technology and use new technology as it comes along, but neither substitutes for teaching them to be disciplined developers.

Disciplined design is both teachable and doable. It requires the use of the most basic logic, nothing as fancy as temporal logic or any of the best-known formal methods. Simple procedures can be remarkably effective at finding flaws and improving trustworthiness. Unfortunately, they are time-consuming and most decidedly not done by senior colleagues and competitors.

Disciplined software design requires three steps:

1. Determine and describe the set of possible inputs to the software.
2. Partition the input set in such a way that the inputs within each partition are all handled according to a simple rule.
3. State that rule.

Each of these steps requires careful review:

1. Those who know the application must confirm that no other inputs can ever occur.
2. Use basic logic to confirm that every input is in one—and only one—of the partitions.
3. Those who know the application, for example, those who will use the program, must confirm the stated rule is correct for every element of the partition.

These rules seem simple, but reality complicates them:

1. If the software has internal memory, the input space will comprise event sequences, not just current values. Characterizing the set of possible input sequences, including those that should not, but could, happen is difficult. It is very easy to overlook sequences that should not happen.
2. Function names may appear in the characterization of the input set. Verifying the correctness of the proposed partitioning requires knowing the properties of the functions named.
3. The rule describing the output value for some of the partitions may turn out to be complex. This is generally a sign that the partitioning must be revised, usually by refining a partition into two or more smaller partitions. The description of the required behavior for a partition should always be simple but this may imply having more partitions.

Similar "divide and conquer" approaches are available for inspection and testing.

While our failure to teach students to work in disciplined ways is the primary problem, the low standards of purchasers are also a contributing factor. We accept the many bugs we find when a product is first delivered, and the need for frequent error-correcting updates, as inevitable. Even sophisticated and experienced purchasers do not demand the documentation that would be evidence of disciplined design and testing.

We are caught in a catch-22 situation:

- Until customers demand evidence that the designers were qualified and disciplined, they will continue to get sloppy software.
- As long as there is no better software, we will buy sloppy software.
- As long as we buy sloppy software, developers will continue to use undisciplined development methods.
- As long as we fail to demand that developers use disciplined methods, we run the risk—nay, certainty—that we will continue to encounter software full of bugs. C

David L. Parnas (parnas@mcmaster.ca) is Professor Emeritus at McMaster University and the University of Limerick as well as President of Middle Road Software. He has been looking for, and teaching, better software development methods for more than 40 years. He is still looking!