**Why is Software Engineering Difficult?[[1]](#footnote-1)**

**Nancy G. Leveson**

Just as James Watt's invention of the first practical steam engine fueled the Industrial Revolution, the invention of the first practical computers about 50 years ago has drastically altered our society. The uniqueness and power of the digital computer over other machines stems from the fact that, for the first time, we have a general-purpose machine (Figure 2.1). We no longer need to build a mechanical or analog autopilot from scratch, for example, but simply to write down the ``design'' of an autopilot in the form of instructions or steps to accomplish the desired goals.

 These steps are then loaded into the computer, which, while executing the instructions, in effect *becomes* the special-purpose machine (the autopilot). If changes are needed, the instructions can be changed instead of building a different physical machine from scratch.



FIGURE 2.1 Software plus a general-purpose computer creates a special-purpose machine

 Machines that previously were physically impossible or impractical to build become feasible, and the design of a machine can be changed quickly without going through an entire retooling and manufacturing process. In essence, the manufacturing phase is eliminated from the lifecycle for these machines: The physical parts of the machine (namely the computer hardware) can be reused, leaving only the software design and verification phases.[[2]](#footnote-2) The design phase also has changed: Emphasis is placed only on the steps to be achieved without having to worry about how those steps will be realized physically.

 The advantages of computers have led to an explosive increase in their use, including their introduction into potentially dangerous systems. This appendix explores the role of computers in accidents, some of the myths related to their use, and why we seem to have so much difficulty with the engineering of software.

Communication problems are making efforts to rectify these deficiencies more difficult. System and software engineers have few common models or tools, and even their vocabulary is different. Computer science, by developing separately from engineering, has created its own technical vocabulary—sometimes the two groups use different names for the same things or they give the same name to different things.

Let’s start with some common misunderstandings about software among non-software professionals.

**Software Myths**

If there are problems, why are computers being used so widely? The basic reason is that computers provide a level of power, speed, and control not otherwise possible; they are also relatively light and small. Many other supposed advantages of using computers are myths, however, usually stemming from looking only at the short term. Understanding these myths is important if we are to make good decisions about using computers to control safety-critical processes.

*Myth 1*: *The cost of computers is lower than that of equivalent analog or electromechanical devices.*

*Reality*: This myth, like most myths, has some superficial truth: Microcomputer hardware is cheap relative to other electromechanical devices. However, the cost of writing and certifying highly reliable and safe software to make that microprocessor useful, together with the cost of maintaining that software without compromising reliability and safety, can be enormous.

The on-board Space Shuttle software, for example, while relatively simple and small (about 400,000 words) compared to more recent control systems, cost NASA approximately $100,000,000 a year to maintain \cite{319}.

 Designing an electromechanical system is usually much easier and cheaper, especially when standard designs can be used. Of course, software *can* be built cheaply, but then lifetime costs—including the cost of accidents and required changes when errors are found—increase and may become exorbitant.

 *Myth 2*. *Software is easy to change*.

*Reality*: Again, this myth is superficially true: Changes to software *are* easy to make. Unfortunately, making changes without introducing errors is extremely difficult. And, just as for hardware, the software must be completely re-verified and re-certified every time a change is made, at what may be an enormous cost. In addition, software quickly becomes more ``brittle'' as changes are made—the difficulty of making a change without introducing errors may increase over the lifetime of the software as changes accumulate.

*Myth 3*.} *Computers provide greater reliability than the devices they replace.*

*Reality*: Although true in theory—software does not ``fail'' in the sense this term usually implies in engineering—there is little evidence to show that erroneous behavior by software is not a significant problem in practice.

When systems were made only from electromechanical and human components, engineers always had to worry about mechanical failure and operator or maintenance error. Techniques were developed to reduce greatly (but not eliminate) random wear out failures and human errors and to mitigate their consequences. Those same techniques and others are used on computer hardware and are largely very effective in making them highly reliable.

Now that computers are being introduced into safety-critical, real-time systems, however, a new, completely abstract factor—software—has been added. Because software is pure design, there is no need to worry about the random wear out failures found in physical analog and electronic devices, but system behavior now can be significantly affected by software design errors and by maintenance and upgrade actions on the software.

Little hard data is available on the reliability of operational software, especially data that compares software reliability to the reliability of equivalent systems that do not use computers. What is available mixes too many types of software (such as game, business, and control software) to be useful.

A study by the British Royal Signals and Radar Establishment used commercially available tools to examine the number of errors in software written for some highly safety-critical systems \cite{340}. Up to 10 percent of the program modules or individual functions were shown to deviate from the original specification in one or more modes of operation. Discrepancies were found even in software that had undergone extensive checking using sophisticated test platforms. Many of the detected anomalies were too minor to have any perceptible effects—for example, a discrepancy of 1 part in 32,000 in a computation using 16-bit arithmetic. However, about 1 in 20 of the functions found to be faulty (that is, about 1 in 200 of all new modules) contained errors with direct and observable effects on the performance of the system being controlled. For example, potential overflows in integer arithmetic were detected that caused a change in the direction of deflection of an actuator—such as ``turn left'' when the correct action was ``turn right.''

On the surface, it seems that the solution to the software reliability problem is simply to get the software right. Although human error is a factor (because software is designed by humans), ample time *appears* to be available to use sophisticated techniques to eliminate any design errors before the software is used. However, accomplishing this goal has turned out to be harder than expected. Very few sophisticated software systems have been fielded that have not contained a significant number of errors.

These are not just ``teething'' problems that go away after the software is used for a while: Software-related errors usually occur over the entire system lifetime, sometimes after tens or hundreds of thousands of hours of use. The Therac-25 worked correctly thousands of times before the first known overdose, and the accidents were spread out over two and a half years \cite{333}.

The Space Shuttle software was used for about 30 years, and NASA invested an enormous amount of effort and resources in verifying and maintaining this software. Despite this effort, since the Shuttle started operation in 1980, 16 severity level 1 software errors[[3]](#footnote-3) were discovered in released software. Eight of those remained in code that was used in flights, but none was encountered during flight. An additional 12 errors of lower severity have been triggered during flight—none threatened the crew, three threatened the achievement of the mission, and nine were worked around. These problems occurred despite NASA having one of the most thorough and sophisticated software development and verification processes in existence.

Although reliability is easily augmented by the use of redundancy for random, hardware wear out failures, techniques that provide the equivalent reliability enhancement for software design errors have not been found. For various reasons (described in the next section), including the fact that eliminating failures caused by design errors is inherently much more difficult than predicting and eliminating wear out failures, highly effective techniques are unlikely to be found. Vendors often tout tools or approaches that will lead to ``zero-defect'' software, but these claims are more sales than science.

Even if the techniques did exist to produce perfect software, there is usually not enough time to accomplish this goal: The ideal conditions—including unlimited funds and time—seldom if ever exist \cite{117}. Instead, there are competing needs to reduce software costs (which already exceed hardware cost in many large systems). Time pressures also become severe, because the time required to develop software often controls the pace of the overall project.

But even if this myth were true—that is, computers are more reliable than other devices—this fact would not necessarily mean that they were *safer* than the devices they replace.

*Myth 4. Increasing software reliability will increase safety.*

*Reality*: Software reliability can be increased by removing software errors that are unrelated to system safety, thus increasing reliability while not increasing safety at all.

In addition, software reliability is defined as compliance with the requirements specification, while most safety-critical software errors can be traced to errors in the requirements—that is, to misunderstandings about what the software should do. Many software-related accidents have occurred while the software was doing exactly what it was intended to do—the software satisfied its specification and did not ``fail.'' Software can be correct and 100-percent reliable and still be responsible for serious accidents. (Examples are given in later chapters.) Safety is a system property, not a software property.

Safety and reliability, while partially overlapping, are not the same thing: Increased computer or software reliability does not necessarily result in increased system safety.

*Myth 5. Testing software or ``proving'' (using formal verification techniques) software correct can remove all the errors.*

*Reality*: The limitations of software testing are well known. Basically, the large number of states of most realistic software makes exhaustive testing impossible; only a relatively small part of the state space can be covered. Although research has resulted in improved testing techniques, no great breakthroughs are on the horizon, and mathematical arguments have been advanced for their impossibility \cite{176}.

The use of mathematical techniques to verify the consistency between the software instructions and the specifications is another way to gain assurance. Although not currently practical, mathematical verification of software is likely to be so in the future. Unfortunately, such verification will not solve all our problems. The process requires that the ``correct'' behavior of the software first be specified in a formal, mathematical language. This task is not easy and may turn out to be as difficult and error-prone as writing the code.

In addition, although the basic computations and algorithms are easily specified and verified, the most important errors may not lie in these aspects of the code. For example, many software-related accidents have involved overload. An instance occurred in England when a computer that was dispatching emergency ambulance services stopped working because it was unable to handle the number of calls it received \cite{341}. Such sophisticated timing problems are much more difficult, and perhaps impossible, to verify formally because they involve more than just the application software itself.

Most important, as stated earlier, practical experience and empirical studies \cite{305} have shown that most safety-related software errors can be traced to the requirements and not to coding errors (which tend to have less serious consequences in practice). Writing adequate software requirements is a difficult and unsolved problem. This book presents some techniques that may help, but the problem is far from solved and may remain unsolved for quite some time.

*Myth 6. Reusing software increases safety*.

*Reality*: Although reuse of proven software components can increase reliability, reuse has little or no effect on safety. In fact, reuse may actually *decrease* safety because of the complacency it engenders and because the specific hazards of the new system were not considered when the software was originally designed and constructed. Examples of safety problems arising from the reuse of software include the following:

* The Therac-20, parts of which were reused for the Therac-25, contained the same error responsible for at least two deaths in the Therac-25. The error had noserious consequences when encountered in the Therac-20; it resulted only in an occasional blown fuse and not in a massive overdose, and so was never detected and fixed (see Appendix A).
* Software used successfully for air traffic control for many years in the United States was reused in Great Britain with less success. The American developers had not worried about handling zero degrees longitude (since that was not relevant in the United States); as a result, the software basically folded England along the Greenwich Meridian, plopping Manchester down on top of Warwick \cite{14a}.
* Aviation software written for use in the northern hemisphere often creates problems when used in the southern hemisphere \cite{342}. In addition, software written for American F-16s has caused accidents when reused in Israeli aircraft flown over the Dead Sea, where the altitude is less than sea level.

Safety is not a property of the software itself, but rather a combination of the software design and the environment in which the software is used: It is application-, environment- and system-specific. Therefore, software that is safe in one system and environment may be unsafe in another. Reuse is *not*} a solution to the safety problem.

*Myth 7. Computers reduce risk over mechanical syste*ms.}

*Reality*: Computers have the *potential* to decrease risk, but not all uses of computers achieve this potential. Computers can automate tedious and potentially hazardous jobs such as spray painting and electric arc-welding, thus reducing the risk to workers in these particular jobs. However, other rguments that computers can reduce risk are debatable:

1. *Argument*: Computers allow finer control in that they can check parameters more often, perform complicated computations in real time, and take action quickly.

 *Counter-argument*: Computers *do* provide finer control computations in real time, and they *can* take action quickly. But finer control allows the process to be operated closer to its optimum, and the safety margins can be cut. The resulting systems have economic benefits, because they will, theoretically, shut down less often, and productivity may be increased by allowing more optimal control. However, any potential safety benefits of the finer control may be negated by the decrease in safety margins—perhaps without the concomitant attainment of the high software reliability on which the arguments for smaller safety margins were based. There is no way to know before extensive use outside the test environment whether high software reliability has been achieved.

1. *Argument*: Automated systems allow operators to work farther away from hazardous areas.

 *Counter-Argument*: Because of lack of familiarity with the hazards, more accidents may occur when operators *do*} have to enter hazardous areas. Assumptions that plants controlled by robots will not require operators to intervene physically are usually wrong and can lead to accidents. For example, a computer-controlled robot killed a worker in a plant that the designers had assumed would require a minimum of intervention—they did not include walkways for humans or standard safety devices such as audible warnings that the robot was in motion \cite{343}. After the plant was operational, the operators found that they needed to enter the hazardous areas 15 to 20 times a day to bail out the robots and maintain adequate productivity: The original assumption that all robots (and thus the plant) would be shut down before humans entered hazardous areas became impractical. The designers had overestimated the ability of the plant to work adequately without human intervention and had not foreseen changes that would be required to the planned operating procedures to meet productivity goals.

1. *Argument* : By eliminating operators, human errors are eliminated.

 *Counter-argument*: Operator errors are replaced by human design and maintenance errors: Thus, humans are not removed from the system, they are merely shifted to different jobs. It should not be too much of a surprise that human designers have been found to make the same types of errors as operators (see Chapter~\ref{human-error}).

 Moreover, as noted in Chapter~\ref{risk}, when humans are removed from direct contact with the system, they lose information that is necessary for correct decision making. Physically removing operators from the processes they supervise may simply lead to new types of errors and hazards.

1. *Argument*: Computers have the potential to provide better information to operators and thus to improve decision making.

 *Counter-argument*: While theoretically true, in reality this potential is very difficult to achieve. The subject is complex, and a detailed discussion is deferred until later. Briefly, computers make it easy to provide too much information to operators and to provide it in a form that is less usable for some purposes than traditional instrumentation.

1. *Argument*: Software does not fail.

 *Counter-argument*: This common belief is true only for a very narrow definition of ``failure.'' Later chapters propose precise definitions of relevant engineering terms and their application to software. The important point here is that computers can exhibit incorrect and hazardous behavior, whether we call this behavior failure or not.

 One of the results of substituting computers for mechanical devices is a reduced ability to predict failure modes. Most mechanical systems have a limited number of failure modes, and often they can be designed to fail in a safe way---for example, a valve can be designed to fail closed or a relay can be designed to fail with its contacts open. In comparison to software, the limited number of physical failure modes also simplifies (1) the analysis of a system for potentially unsafe behavior, (2) the process of assuring that the design is adequately safe, and (3) the elimination or control of hazards to make the system safer. The unpredictability of software behavior and the potentially large number of incorrect behaviors often preclude the same type of failure-mode analysis and fail-safe design.

In summary, computers have the potential to increase safety, and, surely, this potential will be realized in the future. But we cannot assume that we know enough now to accomplish this goal. In addition, any increased potential may not be realized if those building the systems use it to justify taking more risks.

Computers will not go away: Their use and importance in complex systems is only going to increase. Software engineers, often with little training or experience in safety engineering, are building software for safety-critical systems. At the same time, safety engineers are finding themselves faced with ensuring that computer-dominated control systems are safe. To achieve and ensure safety in these systems, software must be included in the system-safety activities, and the software must be specifically developed to be safe using the results of system hazard analysis. A goal of this book is to provide information and ideas about how to do this.

**Why Software Engineering is Difficult**

Why do we have so much trouble engineering software when, for the most part, the software is performing the same functions as the electromechanical devices it is replacing? Shouldn't the same engineering approaches apply since the same type of design errors can be made in both? Shouldn't they be equally hard or easy to construct?

Parnas \cite{318} and Shore \cite{317} have written excellent descriptions of the unique engineering problems in constructing complex software. Much of the following discussion comes from these two sources.

**Analog versus Discrete State Systems**.

In control systems, the computer is usually simulating the behavior of an analog controller. Although the software may be implementing the same functions previously performed by the analog device, the translation of the function from analog to digital form may introduce inaccuracies and complications. Continuous functions can be difficult to translate to discrete functions, and the discrete functions may be much more complex to specify.

In addition, the mathematics of continuous functions is well understood; mathematical analysis often can be used to predict the behavior of physical systems. The same type of analysis does not apply to discrete (software) systems. Software engineering has tried to use mathematical logic to replace continuous functions, but the large number of states and lack of regularity of most software result in extremely complex logical expressions. Moreover, factors such as time, finite-precision arithmetic, and concurrency are difficult to handle. There is progress, but it is very slow, and we are far from being able to handle even small software. Mathematical specifications or proofs of software properties may be the same size as the program, more difficult to construct, and often harder to understand than the program. They are therefore as prone to error as the code itself \cite{344}.

Physical continuity in analog systems also makes them easier to test than software. Physical systems usually work over fixed ranges, and they bend before they break. A small change in circumstances results in a small change in behavior: A few tests can be performed at discrete points in the data space, and continuity can be used to fill in the gaps. This approach does not work for software, which can behave in bizarre ways anywhere in the state space of inputs; the incorrect behavior need not be related in any way to normal behavior \cite{176}.

**The ``Curse of Flexibility.''**

A computer's behavior can be easily changed by changing its software. In principle, this feature is good—major changes can be made quickly and at seemingly low cost. In reality, the apparent low cost is deceptive, as discussed earlier, and the ease of change encourages major and frequent change, which often increases complexity rapidly and introduces errors.

Flexibility also encourages the redefinition of tasks late in the development process in order to overcome deficiencies found in other parts of the system. During development of the C-17, for example—a project that has run into great difficulties largely because of software problems—the software was changed to cope with structural design errors in the aircraft wings that were discovered during wind tunnel tests. This case is typical. As Shore says, ``*Software is the resting place of afterthoughts*'' \cite{317}.

With physical machinery, major design modifications are much more difficult to make than minor ones. The properties of the physical materials in which the design is embedded provide natural constraints on modification. The design of a computer application, on the other hand, is stored in electronic bits and presents no physical barriers to manipulation. Thus, while natural constraints enforce discipline on the design, construction, and modification of a physical machine, these constraints do not exist for software.

Shore explains this difference by comparing software with aircraft construction, where feasible designs are governed by mechanical limitations of the design materials and by the laws of erodynamics. In this way, nature imposes discipline on the design process, which helps to control complexity. In contrast, software has no corresponding physical limitations or natural laws, which makes it too easy to build enormously complex designs. The structure of the typical software system can make a Rube Goldberg design look elegant in comparison (see Figure~\ref{cartoon}). In reality, software is just as brittle as hardware, but the fact that software is logically brittle rather than physically brittle makes it more difficult to see how easily it can be broken and how little flexibility actually exists.

\epsfbox{cartoon.eps}

\caption{A simplified pencil sharpener. (From {\em Rube Goldberg vs. the Machine Age by Reuven L. Goldberg. Reproduced with permission of King Features)}

The myth of software flexibility also encourages premature construction, before we fully understand what we need to do. The software medium is so forgiving that it encourages us to begin working with it too soon. Although we often intend to go back and start again after the details are worked out, this iteration process rarely happens in practice, and design decisions made in prototypes and early design efforts usually remain unchanged. Few engineers would start to build an airplane before the designers had finished the detailed plans.

Another trap of software flexibility is the ease with which partial success is attained, often at the expense of unmanaged complexity. The untrained can achieve results that appear to be successful, but are really only partially successful: The software works correctly most of the time, but not all the time. Attempting to get a poorly designed, but partially successful, program to work all of the time is usually futile; once a program's complexity has become unmanageable, each change is as likely to hurt as to help. Each new feature may interfere with several old features, and each attempt to fix an error may create several more. Thus, although it is extremely difficult to build a large program that works correctly under all required conditions, it is easy to build one that works 90 percent of the time. Shore notes that it is difficult to build reliable aircraft too, but it is not particularly easy to build planes that fly 90 percent of the time.

Few people would dare to design an airplane without training or after having built only model airplanes, but there seem to be few such qualms about attempting to build complex software without appropriate knowledge and experience. Shore explains,

*Like airplane complexity, software complexity can be controlled by an appropriate design discipline. But to reap this benefit, people have to impose that discipline; nature won't do it. As the name implies, computer software exploits a ``soft'' medium, with intrinsic flexibility that is both its strength and its weakness. Offering so much freedom and so few constraints, computer software has all the advantages of free verse over sonnets; and all the disadvantages* \cite{317}.

Another type of discipline is also necessary—limiting the functionality of the software. This discipline may be the most difficult of all to impose. Theoretically, a large number of tasks can be accomplished with software, and distinguishing between what {\em can} be done and what {\em should} be done is very difficult. Software projects often run into trouble because they try to do too much and end up acco mplishing nothing. When we are limited to physical materials, the difficulty or even impossibility of building anything we might think about building limits what we attempt. The flexibility of software, however, encourages us to build much more complex systems than we have the ability to engineer correctly. A common lament on projects that are in trouble is ``*If we had just stopped with doing {\it x} and not tried to do more*\dots.'' McCormick notes that

*A project's specification rapidly becomes a wish list. Additions to the list encounter little or no resistance. We can always justify just one more feature, one more mode, one more gee-whiz capability. And don't worry, it'll be easy—after all, it’s just software. We can do anything.*

 *In one stroke we are free of nature's constraints. This freedom is software's main attraction, but unbounded freedom lies at the heart of all software difficulty* \cite{345}.

**Complexity and Invisible Interfaces**.

One way to deal with complexity is to break the complex object into pieces or modules. For very large programs, separating the program into modules can reduce individual component complexity. However, the large number of interfaces created introduce uncontrollable complexity into the design: The more small components there are, the more complex the interface becomes. Errors occur because the human mind is unable to fully comprehend the many conditions that can arise through the interactions of these components \cite{318}.

An interface between two programs is comprised of all the assumptions that the programs make about each other. Shore notes that such dependencies can be subtle and almost impossible to detect by studying the programs involved. For example, one program might work properly only if another program can be relied on to finish its job in a specific amount of time. When changes are made, the entire structure collapses.

Finding good software structures has proven to be surprisingly difficult \cite{318}. In the design of physical systems, like nuclear power plants or cars, the physical separation of the system functions provides a useful guide for effective decomposition into modules. Equally effective decompositions for software are hard to find. In addition, the relatively high cost of the connections between physical modules helps to keep interfaces simple. As Shore points out,

*Physical machines such as cars and airplanes are built by dividing the design problems into parts and building a separate unit for each part. The spatial separation of the resulting parts has several advantages: It limits their interactions, it makes their interactions relatively easy to trace, and it makes new interactions difficult to introduce. If I want to modify a car so that the loudness of its horn depends on the car's speed, it can be done, at least in principle. And if I want the car's air conditioner to adjust automatically according to the amount of weight present in the back seat, that too can be done—again in principle. But in practice such changes are hard to make, so they require careful design and detailed planning. The interfaces in hardware systems, from airplanes to computer circuits, tend to be simpler than those in software systems because physical constraints discourage complicated interfaces. The costs are immediate and obvious* \cite{317}.

In contrast, software has no physical connections, and logical connections are cheap and easy to introduce. Without physical constraints, complex interfaces are as easy to construct as simple ones, perhaps easier. Moreover, the interfaces between software components are often ``invisible'' or not obvious: It is easy to make anything depend on anything else. Again, discipline and training are required to control these problems, but when the software reaches a certain size (which is often found in control systems today), the complexity can overwhelm even the few tools we have to control it. McCormick suggests,

``*The underlying premise is suspect, namely that we really can build any system, no matter how complicated. The right tool, the right process will let us do anything, or so the salesmen assure us''* \cite{345}.

 Those waiting for tools to solve our problems are likely to be disappointed:

*Regrettably, humans can cope with very little complexity. Better tools and methods can help us with many of the rote aspects of system development; the tools and methods we use are valuable, even indispensable. But consultants and tool vendors often perpetuate a delusion, the delusion that we can cope with endless complexity, if only we would use a better tool or a different method.*

*Tools can only be an aid to judgment. Tools cannot substitute for the physical constraints encountered naturally in other disciplines. Without a harsh and uncaring nature forcing us to make hard choices, we tend to rationalize the complexity we see growing before us on each new project….Despite the best intentions of highly skilled people, each new increment of complexity seems entirely plausible on its own. We are willingly seduced.*

*I submit that the grand failures of big, software-intensive systems have been due primarily to this willing seduction. Post-mortem analysis of such projects routinely reveals a specification that grew in complexity until project cancellation. Natural constraints simply do not apply to software, and nobody knew when to say no. After all, it was only software* \cite{345}.

**Lack of Historical Usage Information**.

A final difficulty with software not found in hardware systems is that no historical usage information is available to allow measurement, evaluation, and improvement on standard designs. Software is almost always specially constructed, whereas physical systems benefit from the experience gained by the use of standard designs over long periods of time and in varied environments. Consider the difficulty that would ensue if every part of an airplane or car were completely changed for each new model or version and the entire design process started anew. That basically describes the situation for software.

To complicate matters further, the features that are most likely to change from one complex system design to another are exactly those that are most likely to be controlled by or embedded within software.

**The Reality We Face**

When systems were composed only of electromechanical and human components, engineers knew that random, wear out failures and human errors could be reduced and mitigated but never completely eliminated. They accepted the fact that they had to devise ways to build systems that were robust and safe despite random failures. Design errors, on the other hand, could be handled fairly well through testing and reuse of proven designs.

Because software has only design errors, the primary approach used to deal with reliability and safety problems has been simply to get the software correct. Theoretically, the possibility does exist for finding a set of techniques or methodology that will allow us to build perfect software. Much energy has been invested in looking for this methodology and less in finding ways to build software and systems that are robust and safe in the presence of software errors.

In reality, the time to create perfect software is never there, and perhaps it never can be. We may be seeking an impossible goal: software that is free of requirements and implementation flaws and that will always do what is required under all circumstances, no matter what changes occur to it or to the environment in which it operates.

Those who believe that the methodology exists that will allow us to construct such perfect software will find this book quite unsatisfactory. For those who have reached the conclusion that this goal is impossible to achieve—or at least not reachable now or in the immediate future—and that other solutions, perhaps adapted from those developed to cope with similar problems in hardware, are necessary, this book provided some clues as to what might be done.

1. © Nancy Leveson, August 2020, all rights reserved. This is a slightly edited chapter from Safeware, Addison-Wesley, 1995. That book has recently gone out of print. The new chapter will be part of a new Safeware II book. [↑](#footnote-ref-1)
2. Although duplication of software might be considered to be manufacturing, it is usually a relatively trivial process. [↑](#footnote-ref-2)
3. Shuttle flight software errors were categorized by the severity of their potential consequences, without regard to the likelihood of their occurrence: severity 1 errors were defined as errors that could produce a loss of the Shuttle or its crew; severity 2 errors could affect the Shuttle's ability to complete its mission objectives; severity 3 errors affected procedures for which alternatives, or workarounds, exist; severity 4 and 5 errors were minor coding or documentation errors. [↑](#footnote-ref-3)