Software Engineering: A Look Back and A Path to the Future

Nancy G. Leveson
University of Washington

December 14, 1996

Trying to predict the future of our field, as others have discovered, is risky: Our technology is changing so fast that the information necessary to make good predictions is simply not available. Instead, I thought I would look at the past and current state of software engineering and use this viewpoint to formulate some hypotheses about what the future *should* hold, that is, some of the paths I would like to see us take.

Software engineering has come a long way since the sixties and the first attempts to make our field into an engineering discipline. In fact, the first steps included the name itself, which reflected the goal of introducing engineering discipline into the software development process. Our achievements toward this goal include a greater understanding of the role of abstraction and separation of concerns in software engineering, the introduction of modularity and the notions of a software life cycle, process, measurement, abstract specifications and notations, etc.

Most of these ideas come directly from engineering, although they needed to be adapted to the unique problems that arise in working with different and more abstract materials. Although hardware engineers are also involved in design, they are guided and limited by the natural laws of the materials with which their designs must be implemented. Software *appears* not to have these same types of natural limits, but to be infinitely flexible and malleable. In reality, the limits exist but are simply less obvious and more related to limitations in human abilities than limitations in the physical world.

Thus the first fifty years may be characterized as our learning about the limits of our field, which are intimately bound up with the limits of complexity with which humans can cope. Our tools and techniques are used to assist us in dealing with this complexity, that is, to help make our systems intellectually

managable. We do this by imposing on the software development process the discipline that nature imposes on the hardware engineering process. We have been learning what types of discipline are necessary and how to best enforce them.

Besides engineering and management discipline, we have also been learning how to apply mathematical rigor and discipline to software development. To this end, many of the pioneers of our field have shown the relationship of software with mathematics and the use of mathematics in solving our problems. These achievements include the axiomatization of programming languages and data types, formal verification, and formal specification and analysis.

Although we have come a long way in building the engineering and mathematical foundations of software engineering and in improving our ability to build complex software, at the same time the problems we are attempting to solve have been getting more difficult: Man's reach always seems to exceed his grasp. The problems are also changing in their fundamental nature. The earlier emphasis on efficiency has shifted to an emphasis on correctness and utility as we become increasingly dependent on computers in applications where losses due to computer errors are potentially huge. Economic considerations have increased the emphasis on reuse and reusable components. And although our early days were filled with building new software, we are more and more consumed with the problems of maintaining and evolving existing software. In addition, as our systems grow bigger and require large teams of designers, we have started to examine the ways humans collaborate and to devise ways to assist them to work together effectively.

These same trends will continue in the next fifty years, with perhaps even less emphasis on coding and more on the other aspects of the software engineering process. But there will be new challenges and perhaps new approaches and directions that will be required to solve the problems of the next century. To address these challenges, we may need to shift our emphases and follow some new paths.

If our problems in building and interacting with complex systems are really rooted in intellectual managability and human limits in managing complexity, then we will need to stretch these limits to build ever more complex systems. But basic human ability is not changing. To successfully build and operate ever more complex systems, we will need to find ways to *augment* human ability, both in terms of system designers and system users. Achieving this goal, I believe, will require augmenting our engineering and mathematical foundations with ideas from cognitive psychology and the social sciences.

While our first 50 years have seen us develop our concepts of software as an engineered product and a mathematical object, less attention has been focused

on software as a human product and on computers as devices that interact with and assist humans (as opposed to replacing them). Software engineering is a problem-solving activity and software engineering techniques and tools are used to assist humans in this activity—the effectiveness of our tools could be greatly increased if we based their design on scientific knowledge about how humans solve problems. Our software products are also used or monitored by humans, and the way that our software is designed to interact with humans is a critical factor in whether the software is useful to or usable by them.

When creating new software engineering methods and tools, we often inadvertently enforce particular problem-solving strategies, often the one preferred by the designer of the method or tool. We need to learn more about human problem solving, particularly with respect to software engineering tasks, and give our students a better grounding in cognitive psychology. For example, psychologists have found that not only do problem-solving strategies vary among individuals, but individuals vary their strategies dynamically during a problem-solving activity. To design more effective and usable software engineering methods and tools, we need to ensure they do not limit or assume certain problem-solving strategies but instead support multiple strategies and allow for shifting among strategies during problem solving.

Our techniques and tools not only have an effect on our problem-solving ability, they also affect the errors we make while solving those problems. Thus, our tools and methods should also reflect human limitations and capabilities, which will require our learning more about human errors and limitations in performing software engineering tasks and in using our tools and products.

In addition to these new challenges in making our software engineering techniques more human-centered, important problems are starting to arise in designing human-software interfaces and interactions. In the engineering world, the challenges in building high-tech systems composed of humans and machines have necessitated augmenting traditional human factors approaches to consider the capabilities and limitations of the human element in complex systems. *Cognitive engineering* is a term that has come to denote the combination of ideas from systems engineering, cognitive psychology, and human factors to cope with these challenges. With computers playing more and more important roles in these systems, computer science and especially software engineering needs to be integrated with these other concerns.

I believe that many of the problems arising in our attempts to build complex systems are rooted in the lack of integration of software engineering, system engineering, and cognitive engineering. We need to build more bridges between these three disciplines. The problems in building complex systems today often arise in the *interfaces* between the components—where the com-

ponents may be hardware, software, or human. One example is the recent glass cockpit aircraft accidents where the events have been blamed on human error, but more properly reflect difficulties in the collateral design of the aircraft, the avionics systems, and the demands placed on the pilots. We need methodologies that ease coordinated design of the components and the interfaces and interactions between the components and that provide seamless transitions and mappings between the disciplines involved.

Another example of the important questions we need to tackle is the reasonableness of our goals in terms of replacing humans (such as pilots, nurses, factory workers) by computers. Aside from the moral and philosophical questions, there are technical ones: Have we oversold (albeit inadvertently) the ability of computers to replace human intelligence and ability? Often, we simply automate what can be automated while leaving humans with an assortment of miscellaneous tasks that may be harder to do correctly in isolation. At the same time, we ask humans to perform what are often impossible monitoring or backup tasks and then blame them when the inevitable accidents occur. Do we increase risk or simply change it by using computers to provide control of potentially dangerous systems rather than assisting humans in doing a better job of controlling them? The latter is more difficult because it requires a deep understanding of human capabilities and limitations, but will it get us farther in the end? These are some of the new issues I believe software engineers will have to confront. To solve them will require recognizing the important role of psychology in software engineering, augmenting our foundations with appropriate knowledge, and building links with cognitive engineering.

Our links with the social sciences also need to be strengthened. Truly understanding and advancing a technology requires understanding its history, scientific basis, and the cultural and social milieu in which it operates. Technology does not exist outside of the context of a human society:

> We pretend that technology, our technology, is something of a life force, a will, and a thrust of its own, on which we can blame all, with which we can explain all, and in the end by means of which we can excuse ourselves [T. Cuyler Young in *Man in Nature* edited by Louis D. Levine, Royal Ontario Museum, Toronto, 1975].

We need to place more emphasis on understanding the effects of the technology we create on the world. We have had a tremendous effect on human life and human society, but only a few computer scientists seem to be considering these effects to any degree. While caught up in the fervor and excitement of developing a new and revolutionary new technology with the potential to change the world in profound ways, we might be excused for concentrating

on the technical to the exclusion of the social. But we have now matured to the point where we need to start assuming responsibility for what we do. A basic precept in most engineering professional codes of conduct is that engineers shall hold paramount the safety, health, and welfare of the public in the performance of their professional duties. As a maturing field, we will need to develop our own standards and codes of professional conduct and more fully accept our responsibility for the uses and potential misuses of our inventions, for the effect we have on society and human life, and for our role in those events.

The history of software engineering has been one of coming to see that what originally seemed to be limitless actually does have limits, understanding the nature of those limits, and then searching for ways to expand them. To continue our progress, we will need to continue building our scientific knowledge about those limits and searching for new and different ways to stretch them.