THE ROLE OF SOFTWARE ENGINEERING IN SYSTEMS FOR DESIGN AND PROCESS CONTROL

Nobel Laureate Herbert A. Simon
Carnegie Mellon University

Nearly four years ago I spoke at the Second World Conference on Integrated Design and Process Technology in Austin; and now I am speaking to the Fifth World Conference, at which SDPS is joined by a sister society, the Software Engineering Society. On both occasions, I have relied upon technology — appearing not in material form but as mind, that is to say, not by physical transport but represented by disembodied words.

As I am going to take up my remarks where I left off on the final theme of my talk of four years ago, and as the computer on which I am drafting these remarks has probably never been turned off in the intervening years, this may represent one of the longest speeches in history. I don't know whether that is a record to be proud of or not, but it might make me eligible for the Guiness Book of Records.

1. The Current State of Affairs in Software

Since a few of you may have forgotten that final theme in my previous talk, let me recall it for you. It had to do with constructing revisable and understandable systems, with special reference to software systems. I quote from the published version,

When we train a person to perform a task, we know that if the task changes, the person can often learn to adapt to the change. A continuing problem with automated expert systems has been that they have typically had no such learning capabilities. When that is so, we can amortize the investment in expert software over only a short period of time, and then must build a new system or make a substantial investment to understand the old system sufficiently well that we can modify it — often a forbidding task. A major design requirement for complex systems in a changing world is that they be modifiable at an acceptable cost, either by human intervention or through learning processes incorporated into the systems themselves.

How easy or hard it is to modify a system depends, in turn, on the thought that has been given to flexibility in designing it — especially its modularity, so that modification of one component does not lead to unanticipated changes in the behavior of other components. If the system is modifiable, then we can aspire to give it the ability to modify itself in response to new specifications of its task or of the environment in which the task is performed. Ability to learn is another form of intelligence that we must design into our systems.

I regret to report (but you are already aware of it) that the problem of designing understandable and revisable systems has not been solved in the four years since my last conversation with you. It must be a very difficult problem, for it has been with us since the birth of modern computers, a half century ago, and at least along some dimensions, it is a more severe and damaging problem now than it was then.

Let me start from the beginning: Hardware and software are the matter and mind of computers, as body (including the brain) and mental processes (thoughtware) are the matter and mind of biological organisms. To work together, body and mind must be compatible with each other. But computer hardware engineering progresses almost independently of software engineering, with the gulf between them appearing to increase each year.

I will not try to hazard a guess as to why this is so. Perhaps it has something to do with the fact that hardware designers and manufacturers have imagined that if the matter were properly designed and maintained, the mind would take care of itself. In my consulting activities during the 1960s and 1970s, I found the executives of major computer companies quite willing to listen to gee-whiz stories about advances in artificial intelligence, but not willing to believe that computer intelligence might have important implications for the future of computers and their application to the real world. Certainly, at least until the personal computer made its presence felt, hardware was in the saddle, and software trotted behind, and it is clear that the necessary radical readjustments in that relation have not been accomplished up to the present date.

As early as 1955, nearly a half century ago, engineers at Westinghouse Electric had built and were using automatic programs (written in Fortran!) to design, wholly automatically, electric motors, transformers, and generators. The programs were capable of handling about 70% of the customer orders that couldn't be satisfied by off-the-shelf items. About fifteen years later these programs were no longer in use. Why? Because it was too costly to up-date them continually to take account of improvements in device design and — more ominous, to adapt to the constant change and growing complexity of the software programs, from assembly language upwards, in which the Fortran programs were embedded. It was easier and cheaper to assign this task of adaptation to human engineers than to automate it.

The situation today approaches the ridiculous. In defiance of the slogan of "upward compatibility," the new computers we are urged to acquire for the miracles they perform, are willing less and less to perform the humdrum task of running last year's programs, written for simpler machines. Ditto for the software systems, designed to run on the new machines, but not compatible with the old ones. The Y2K problem was just one pool of incidents in a whole sea of compatibility problems, and far from the most important one. I wonder if there is any software user in this room who does not find this year more overwhelming than last year in the burdens that are imposed by continually changing hardware and software.

Each day I receive e-mail with attachments that, even if they do not contain lethal viruses, are untranslatable unless I can find, somewhere on my campus, conversion software packages that happens to understand the particular obscure languages in which they are written. And if I were to attempt to keep myself up to date on all these "improvements," I would spend my entire life in such activities and have no time for my research — like some of our brightest and best hacking students. The Tower of Babel is being rebuilt, and rises story by story each day.

Matters would be quite tolerable if we relied entirely on our personal computers. Then we could simply reject hardware and software novelty unless it offered benefits that compensated for the time and money consumed in adopting it. We could continue to do good science with our ten-year-old computers. But e-mail, the Web, and groupware have deprived us of that luxury. However unimportant it may be to our own work to modernize our systems, modernize we must, if we are to remain in communication with our colleagues.

The problem is not entirely novel. I have already referred to the multiplicity of languages in the world (where, by the way, the rapidly increasing importance of international communication and collaboration is driving everyone, against their deepest inclinations, toward English). There is also the matter of the metric and English system of measures, with which the United States, at least, has not yet made its peace (even in its space program). And then there is the case of railroad gauges: are Western Europe and Russia still operating on tracks of different width? And has there been any reconciliation in Australia between the tracks of Victoria and New South Wales?

I suppose these examples might be taken as evidence that we can live without standardization, but the incompatibilities of computer software are much more costly than these others. Only the human language problem has comparable scope, and its impact is only now beginning to be felt because of rapid internationalization.

Languages, when largely restricted to separate social communities, have always been very conservative systems, changing at a leisurely pace that required little revision, during a lifetime, of knowledge that was acquired beginning a year or so after birth. And the problems of communication between language communities were mitigated by the ability of humans, when we put our minds to it, to more or less master several languages if we have occasion to use them over some substantial period of time.

No such extenuating circumstances are present in today's software situation, or in the prospects for tomorrow. We are incurring enormous costs for the human learning and the reprogramming required to accommodate to the high rate of system change — costs that are much larger than the costs of the equipment replacement, or the costs of developing the novel software that lies at the root of the problem. How much of your workweek are you and your colleagues now spending in learning about the new systems that flow in on you?

2. The Task for Software Developers

I would prefer, however, to talk about solutions than about problems. Paradoxically enough, if software novelty is the problem, novel software must be the solution. We have a whole agenda of tasks laid out before us in order to produce software that reaches the levels of understandability and revisability it will need to reach in order to play the role we are asking it to play in the worlds of business organization and all forms of communication. Let me first address the issue of understandability, then the issue of revisability, although we will see that the two are closely related.

2. 1. Understandability

At the outset, we must ask, "understandability to whom?" There are the things that systems designers must understand and things that systems users must understand. Equally important, there are things that the computers and their programs must understand. If a computer does not understand what "add" means (i.e., what to do when that symbol appears in its instruction register), then it will not do very good arithmetic. The boundaries of responsibility for understanding, between designers, users, and computers, are themselves constantly shifting.

Users. I will start with the users, for, above all, our aspiration is to be "user friendly." We began our user-friendliness with the User Manual. (I will be eternally grateful to the lucid manual of the IBM 701 computer in 1952.) When manuals no longer worked too well, after about 1990, we went over to informative monitor displays combined with on-line help messages.

Without detailing my charges, but appealing to your experiences, I will declare that shift a disaster. Operating a system with only monitor and on-line help, the difference between having or not having a genuine (human) systems expert at the adjoining desk is colossal. Our current systems simply cannot

carry on adequate conversations with humans about their processes, and unless we can find the knowledge in a nearby human head, even our own as the last resort, we are helpless. The advantage of the human head, of course, is that it can carry on a coherent conversation with you, something still beyond the competence of most computer dialogue systems.

2.1.1. Software engineering goal #1

To rethink the task of helping users operate with their software and diagnose its difficulties when they appear.

Here I have a few positive suggestions, one suggested by the history of the automobile. To make it universally usable without devoting our educational institutions wholly to driver training, we made as many things as possible both standard and automatic (starting the motor and shifting gears are the prime examples) and limited the forces drivers had to apply for braking and steering. We are now exploring a whole new round of automation, soon including steering and even perhaps driving in general. The idea is: "You don't have to train the driver for the things s/he doesn't have to do."

Another name for this cure is to simplify the system, often at the expense of making fewer alternatives available to the user. But simplification can save enormous costs in user time and effort. The idea that you can achieve such savings by standardization is not new. When consumers were offered a cheap, standardized, usable "Tin Lizzy," they flocked to it by the millions.

A related simplification uses a black-box design to relieve the user of the temptation of dealing with details. Systems programmers have always fought vigorously against black-boxing computer languages; they need, even today, to rethink what was gained and lost by going back to the assembly-language detail of C++. The problem with (many) systems programmers is that they spend all their time mastering a language and wonder why users shouldn't do the same.

Perhaps the most important direction of relief to the user is to automate the diagnosis and cure of malfunctions — a path that has been followed with considerable success in auto repair and increasingly in program verification. If the computer can find its own bugs, it doesn't have to know how to help the user find them. This domain, currently pursued with some vigor, seems to me a prime candidate for software R&D.

2. 1. 2. Software engineering goal #2

To rethink the task of enabling programmers to debug and update systems that have to run in highly complex software environments.

Programmers. Most of what I have said about users can be repeated for programmers, albeit at a somewhat different level of detail and sophistication. Here the difficulties and costs show up primarily in debugging and in the modification of programs to keep them contemporary with the domain on which they operate. (My earlier Westinghouse example is a case in point.) In the multi-layered programming worlds in which we live, one of the most challenging tasks is to reduce the cost of adapting and debugging higher-level languages when they are moved to different hardware and software environments. How common is Common Lisp, and what effort does it take to make and keep it common when you transport it to a new assembly language?

An effort along this line has been of much concern to me personally of late, as we tried to move a sizable program called CaMeRa — which made use of the Common Lisp capability for representing diagrams as bit maps, from a Macintosh environment to an IBM Windows environment. Perhaps this is no great task for a true aficionado, but it can baffle for months users who don't know the underlying environments well — and systems programmers, even experts in Lisp, who don't know the structure of

the higher levels of the Lisp source code. If I am simply exhibiting my ignorance, so be it, for my central point is the unacceptable cost of trading that ignorance for knowledge. We cannot afford to change our factories and offices wholly into schools.

If I were not afraid of being accused of fascist tendencies, I might propose even more forms of standardization to reduce the costs of making systems adaptable and compatible. As the Microsoft case teaches us, standardization can raise important and difficult problems of power, including monopoly power as well as other kinds. We could even worry about electronic ethnocide, to coin a phrase, of programming groups the world round no longer permitted to speak their own patois.

But we require one more goal of understandability: understandability to the computers themselves.

2. 1. 3. Software engineering task #3

To bring computers and computer programs to a higher level of understanding of their own languages and their own processes.

Understandability for computers. One response that one still hears in the year 2000, when one suggests improving the ability of computers to understand what they are doing, is that computers *can't* understand. I have already devoted many words, in many talks and papers, explaining how computers, in appropriate circumstances, do understand. The Turing Address that Al Newell and I gave in 1965 is a good place to find a summary of our viewpoint, so I won't repeat the argument here.

In fact, however, the best way to find out about computer understanding is to examine the numerous programs in the world today that, tested by the criteria used to test human understanding in the same situations, do have a very large capacity for understanding — usually limited (not wholly unlike human understanding) to particular domains. A few minutes ago, I explained what it means for a computer to understand the word "add." Beyond that, look at the program that steers a car on open highways at high speed, the robots that play soccer, Siklossy's 1968 program for learning language from pictures of scenes — the list is nearly endless and growing rapidly, especially as, through robotics, programs come into contact with real-world environments. When judging computer understanding, be sure that you apply the same test of understanding that you would apply to a person in the same situation, avoiding a double standard.

Let me be specific by talking about engines for searching the Web. A search engine that simply employs key words, however cleverly, does not understand, or need to. By developing such search engines, in all their varieties, we are able to increase the speed of our searches to high levels, and increase their yield. But the latter is a very mixed blessing indeed. The powerful engine simply returns to us a multitude of crudely chosen samples of the world's information and challenges us to sort out the small fraction that are relevant to our particular need, and in particular, the even smaller fraction that is so important that we can devote any of our time to it. We squander our own intelligence and time if we must do this essential sorting.

The many things that we promise ourselves from the Web are only going to be genuinely available to the degree that we instill intelligence in our search engines that will distinguish the interesting from the uninteresting, and more crucially, the vitally important from the merely interesting. Only then will these searches be worth the scarce human time devoted to using their product.

But the need for intelligence in computers goes far beyond searches of data bases. It becomes especially important when we ask how we can update our programs to keep pace with constantly changing knowledge and techniques. Human beings learn. However painfully they learn, they learn massively. Children and young adults typically learn an average of about ten words per day — their appearance, sound and meaning — throughout their first twenty years. (They learn a lot of other things

too, and learning words appears to occupy only a minute or two per word.) Because we learn, we can pass a lifetime in a world that changes substantially, year by year, although at the cost of devoting a substantial part of our time to learning.

For us to realize our aspirations for computers, they must acquire comparable capabilities for learning. Computer learning is not new; it harks back to Arthur Samuel's program that learned to play checkers, which was operative about 1956. That pioneering effort has been followed by many others, using a variety of learning devices, including discrimination nets (e.g., EPAM), neural networks of many sorts, Bayesian decision trees, statistical Markov processes, and adaptive production systems.

All of these procedures, and perhaps others, need to be in our repertory of learning schemes, and they need to be applied especially to the tasks of enabling programs to learn what they need to know in order to update themselves. Notice that a price is paid for this independence. Just as we human beings do not know what goes on in each other's heads, or how the knowledge of others is organized, computers that learn independently will be correspondingly opaque to their human programmers and users. As in the case of humans, whether that is a good or a bad thing depends on how effective the computers are in their learning.

There remains one factor in favor of machine learning that decisively tilts the balance in its favor: a computer program, however it was constructed or learned, whether we understand it or not, can always be transferred from one machine to others almost costlessly (provided, of course, that lower software levels are compatible). This is not true for human learning, which has to be repeated for each learner.

2. 2. Revisability

I am sure that, with your help, I could add extensively to the dimensions along which the ability of computers to understand what they are doing will contribute in major ways to their value. But in talking about computer learning we have already crossed the boundary between understandability and revisability, and I would like to devote my remaining time to the latter topic. The discussion will show, moreover, that the two topics are very closely related.

3. Complex Systems

Today, complexity is a word that is much in fashion. We have learned very well that many of the systems that we are trying to deal with in our contemporary science and engineering are very complex indeed. They are so complex that it is not obvious that the powerful tricks and procedures that served us for four centuries or more in the development of modern science and engineering will enable us to understand and deal with them. We are learning that we need a science of complex systems, and we are beginning to construct it.

One of the facts about complexity that has evoked considerable interest is that almost all of the successful complex systems we find in nature (structures like atoms and molecules, and systems like live organisms), as well as the most complex artifacts that we have learned to design and build, have a common, if very general, feature. They are hierarchical in nature, like successively nested sets of Chinese boxes, each box cradling a set of smaller boxes.

If we were physicists or chemists, we might prefer to describe them as molecules made up of peptides and amino acids and the like, these made up of atoms, these, in turn, of electrons, protons, and neutrons, and these made up of quarks. Maybe the meaning of this layering would be clearer to the group at this meeting if I said that they are like large computer programs built from successive layers of closed subroutines.

These systems tend to have one further property: the connections and interactions within each subsystem are more frequent and rapid than the interactions between subsystems. Systems with this

property we call *nearly decomposable,* and it is no accident that most complex systems, natural or contrived, are nearly decomposable. In particular, near-decomposability facilities their formation, and it greatly increases the rate at which they can evolve and improve their performance — hence survive in competition with other systems. Let me explain.

First, suppose we begin with some small systems that have the property that they can form into small stable clusters, and these, in turn, into clusters of larger size, and so on. Such an array will gradually produce systems of large size and complexity, and will do so much more frequently and rapidly than if the larger structure had to be assembled from the smallest components by a single, instantaneous miraculous act. Elaborated, this process can provide an explanation of the origin of complex molecules from their components, and even of viruses and bacteria from complex molecules, by more or less chance meetings of compatible components, even in the relatively "short" time available since the Big Bang.

Second, suppose we already had a somewhat complex system with specialized subparts (I will not pause to explain where the specialization came from), then, if these subparts were only lightly linked (say by their inputs and outputs), with independent internal processes, each of them can mutate into more efficient forms without damaging the effectiveness of the others. Each can evolve to higher fitness independently, without troublesome go-to's that cause malfunctioning in other parts of the system. In environments where there is natural selection, nearly decomposable systems will simply evolve much faster than systems that do not possess this property.

The verbal arguments I have just presented are easily translated into rigorous mathematical form, and, quite recently, computer simulations, using genetic algorithms, have demonstrated the superior rate at which nearly decomposable organisms gain fitness as compared with their competitors.

What does this mean for software engineering? It means that we need to take near decomposability as one of our central design constraints for systems that are to be understandable (to themselves or to us) and improvable by revision. This idea (first mathematized about 1956) already forecast the effectiveness of the venerable concept of structured programming.

Near-decomposability also casts considerable light on the difficulties encountered in attempts to build general-purpose parallel computers that placed no discipline on the kinds of interactions that can take place among components. In fact, the whole set of questions of when, where, and how parallelism is feasible and superior to seriality, needs to be reviewed in the light of the concept of near decomposability. That in itself is an important task that will keep many of us busy who are interested in fundamental and abstract questions associated with computation and computer software.

But I would not like to leave with you the impression that near-decomposability is only of interest to theorists. Under the rubric of "division of work," it is a central concern of all those who design organizations. Under the rubric of "specialization" it is a central concern of those who design production processes. It or its absence are fundamental properties of complex systems that should be of central concern to all of us who are concerned with designing or operating such systems.

4. Conclusion

Let me halt my account at this point. I have not tried to paint a picture of the software future, for two reasons. First, I don't know what that future will look like; I have no crystal ball.

Second, and more important, our task as scientists and designers is not to predict futures but to help in visualizing desirable futures, to create pictures that can teach us the characteristics of realistic, sustainable worlds, and can perhaps give us hints of the routes that might take us to one of these worlds. The sustainable world I see for software is a world that is understandable by people and computers, and that is revisable, that is, improvable; probably because it has been designed with near-decomposability in mind.