Only by acknowledging change as a constant in our industry can we successfully negotiate the challenges we face and ensure our continued survival in this computer age.

# Software's Future: Managing Evolution

**M.M. Lehman,** Imperial College, London

In his essay, Ed Yourdon expresses, justifies, and leaves unresolved two well-founded questions: What is the future of software? What does the future hold for the software professional? His prognosis is evasive, incomplete, and unsatisfying: the future will be good for some, not so for others.

Given Yourdon's extensive experience in the real world of computer usage, as proven by the problems he has observed, it is easy to see why he feels that software's future is uncertain. But he does not point to a solution to this uncertainty, nor does he indicate what can be done to achieve the best possible outcome for software professionals. More importantly, Yourdon's analysis does not indicate what should be done to ensure the security, well being, and survival of society, which depends increasingly on software.

For more than a decade now, there have been those in the software engineering community who have accepted that the need to continually change and evolve software is a fact—a fact addressed through activity that is planned, executed, and controlled by humans. Thus, the software development and maintenance processes, which I prefer to unify and call *software evolution*,[1] are key to managing computerization. In my view it is key to our survival in this computer age.


0740-7459/98/$10.00 © 1998 IEEE

## WHERE ARE WE?

Society is growing ever more dependent on computers and, therefore, on software. As more and more individual and organizational activities involve computers, both society's software dependency and the interdependency among users increases more than linearly. This creates internal and external pressures and should change the way software development is assessed, planned, and managed.

### Internal affairs

Applications are becoming ever more extensive and critical. Ever larger organizational units are partially, at least, subject to common control. Individual software systems are growing rapidly in functionality, behavioral complexity, size, and structural complexity. Software reflects organizational and individual behavior in ever greater detail.

As organizations increase their exploitation of computer technology, integration becomes the order of the day. Previously independent systems merge and are made to communicate with—and in some cases control—each other. Despite this, they must remain compatible with established processes and practices. Coupling—both within the system and between the system and users—is initially loose, but inevitably becomes tighter and more intimate with time. Moreover, in specifying new applications, seeking mergers, and integrating already operational systems, developers must impose bounds on both the applications and the extent and details (bells and whistles) of their operational domain. This results in restrictions that become the source of bottlenecks and irritants, both in themselves and because they result in a need for manual intervention when mechanized functions are insufficient or incomplete.

The consequence is that users experience frustration, growing ambition, and increasing interdependence in the operating environment that leads to pressure for change and unification. Such unification can never be complete. There is inevitable and irresistible pressure for change on an ever more extensive scale. To achieve this reliably, economically, and promptly requires deep insight by users and developers into the characteristics of the application in its operational domain and into the solutions to be employed and their wider impact.

### External forces

There are also exogenous pressures originating in the operational domain. Sometimes—as exemplified by Yourdon's discussion of the millennium problem—such pressures may be unavoidable once they arise. They could, however, have been avoided, if the activities I discussed later had been common practice.

Other external pressures result from policy decisions that take little if any note of the wider IT impact. The political decision to adopt the Euro currency is one such example. Other pressures include those resulting from technological changes whose exploitation requires software adaptation. One example is the extension of telephone number codes resulting from the explosion in telecommunication usage. Pressure to change can also come from flawed system design decisions or implementations, such as the European Space Agency's Arianne 5 flight 501 rocket disaster.[2]

Whatever the source of the problem, the result is the same: frustration, expense, even disaster, followed by an urgent need for software change. And if these were not enough, there is always the demand for more detail to make software more responsive to people's needs, to the changing domain, and to changing technological, legal, and economic requirements. The opportunities offered by design and implementation are equally limitless. Exploiting this multidimensional unboundedness is an inevitable consequence of computer usage. Facts of life such as these lead to an unending process of software change, enhancement, and evolution, the unending sequence of releases with which we are all familiar.

There is inevitable and irresistible pressure for change on an ever more extensive scale.

## SOFTWARE IN THE REAL WORLD

In structure, content, and functionality software systems are by far the most complex artifacts we have ever created. Software itself is a model of the application, its participants (human and mechanical), the operational domain, and activities in that domain. The entire universe constitutes the domain, even though many of its elements and properties may appear irrelevant in the context of the application as it is at the time of conception, development, and installation of the system. And this domain is essentially unbounded: it is, in fact, countably infinite in at least three dimensions.

### Danger of assumptions

The problem of apparently irrelevant detail subsequently becoming significant is beautifully illustrated by the case of the recently constructed CERN particle accelerator. The new machine had twice the diameter of its predecessor. It was put into operation using control and interpretation software transferred from its predecessor. The results were dismal. They varied from day to day, and even previously successful results could not be reproduced. It was only when someone noticed an apparent correlation between the variations and the moon's phases that realization struck: The previously justified assumption—that the moon was outside the selected

bounds of the operational domain and its gravitational pull was too small to be of consequence—turned out to be invalid when new circumstances (a larger accelerator) arose. There was no choice but to modify all the software to account for the new circumstance.

*E*-type software is defined as software that addresses some real-world application, problem, or activity in some real-world domain.[3] Such software is an artifact created by humans in finite time, is represented by finite sequences, is developed by creators who generally have only limited understanding of the intended application and its real-world domains, and is executed in a system with finite storage capacity. The software is, of necessity, both finite and incomplete. It is a finite and incomplete model of an unbounded application in an unbounded operational domain.

Software systems are static unless and until humans change them.

To further complicate matters, the executing program becomes part of the application and its domain. It must contain a model of itself, of its own operation. In a finite domain (model), that is logically impossible. This intrinsic limitation in the software's ability to take its own behavior in the operational domain into account is another source of imprecision and incompleteness. Every *E*-type program is essentially incomplete.

### Bridging the gap

There is, therefore, always a gap between the bounded system and the unbounded application in its unbounded domain. This gap is bridged by assumptions that are embedded in the software in various ways, in the choice of algorithms, values for parameters, sequencings, and so on. System selection, specification, design, and implementation also involve numerous assumptions. Some of these assumptions will be made explicitly, during requirements selection for example. Others will be implicit, a consequence of adopting a theory, designing an algorithm, selecting a procedure, defining an interface, setting limits, or even in deciding to use two digits to represent years (whence the millennium problem) and so on. The sources and nature of the assumptions are countless (I have estimated that a typical program has about one real-world assumption for every 10 lines of code). Some of these assumptions will remain valid throughout the system's life; others will be invalidated by subsequent changes in the application or its operational domain. Still others will fall somewhere in-between: valid in some circumstances, but leading to unacceptable results or behavior in others. Such invalidity generally remains undetected until a problem arises or a disaster such as Arianne 5 flight 501 occurs.[2]

### Management challenge

Software systems are static unless and until humans change them. Software cannot, of itself, adapt to external change. Only where humans anticipate change and incorporate a correct adaptation mechanism into the executable code can software be self-adaptive. Code can be made flexible only to the extent that programmers specifically recognize uncertainty or a possibility of change and incorporate the appropriate tolerance, responsiveness, and switchable new mechanisms into the system's logic and its textual implementation. One incorrect bit among tens of millions can cause misbehavior, failure, or, in extreme cases, disaster. But the universe and the operational domains therein are continually changing. Hence, the gap between the domain and its software model tends to grow. The gap can be kept satisfactorily small only by ceaseless maintenance.

The challenge to keep relative parity between a system and its operational domain, despite anticipated changes, increases as our dependence on computers grows. The penalty for failure increases as well. These observations are not new; they were first recognized in the late '60s.[4] However, given that the problem—the range, extent, and criticality of computer application—and the concomitant threat to humanity continue to grow, we must raise the question: In the light of the facts outlined here, how shall we manage computerization and the planning, development, and use of appropriate computer systems and their software?

## Recommendations: The way forward

From about 1989 onward, formal process modeling has been widely viewed as a key to managing the inevitable changes we face.[5,6] But, as evidenced by Yourdon's essay in this issue, it is widely recognized that neither modeling nor formalism have solved the problems.

Recently I suggested[7] that a phenomenon first recognized in 1972[8] could be invoked to explain the major difficulties encountered in producing and maintaining software and in improving the software process. The FEAST (feedback, evolution, and software technology) hypothesis formalizes this observation as follows:

> *The software process constitutes a multilevel, multiloop feedback system and must be treated as such if major progress in its planning, control, and improvement is to be achieved.*

Process is used here in its broadest sense to include any activity that influences its outcome. It encompasses, therefore, not only the activities of technical personnel but also those of, for example, management (from line managers to organizational execu-

tives), marketing and sales personnel, user support, and users. All have an impact on the process. All are sources of feedback information and subject to feedback control. All must be considered when developing process models—models that must include information flow paths and feedback mechanisms if they are to support process evaluation and improvement.

Recognition and treatment of the software process as a feedback system is, at present, the exception rather than the rule in the process and improvement communities. However, first results of the FEAST/I project[9-11] confirm my observations and conclusions from the 1970s,[12] suggesting once again that, among other things, mastering and exploiting the feedback phenomenon may be key to help solve the problems Yourdon outlines. But it must not be seen as the final solution to these problems, merely as a tool to help identify solutions.

Projects such as FEAST/1 are opening up new approaches to understanding what is involved in computerization and how you might cope more effectively with the challenges raised by the application of information technology. A brief, unordered list of some guiding principals for avoiding problems such as those Yourdon described follows. A more systematic and complete account must await another opportunity.

♦ When introducing or expanding computer support, its wider impact must be considered. It is not sufficient to simply consider an application's effectiveness or its economic benefit in a local domain.

♦ Application and domain boundaries must be identified from the start. Such decisions should be recorded in a structured fashion that also displays the recognized dependencies and relationships between them.

♦ It must be recognized and accepted that such bounds will change and expand with time, experience, and indigenous and exogenous change. Changeability is a necessary attribute of software architectures, designs, and implementations.

♦ Thus, as development proceeds, boundaries must be updated to reflect an inevitable increase in understanding, the implications of the emerging design and implementation, and the anticipated impact on and of the user.

♦ Bounds definitions must be reviewed regularly, both during and after development, to ensure continuing satisfaction as circumstances change.

♦ At all stages of definition, design, and development, attempts must be made to recognize, capture, and record assumptions, whether explicit or implicit, in design and implementation decisions, as must any dependencies and relationships between them. Such assumptions relate not only to technical and management matters, but to the reactions of users, to a program's impact on the operational domain, to economic and societal factors, and so on.

♦ Assumptions must be recorded (preferably in machine-processable form) in a structured fashion to simplify the inspection and identification of any that may have become of questionable validity.

♦ Both the bounds and assumptions records should contain indications of the likelihood of future application or domain changes, and of change drivers, to guide and limit the review process.

♦ Assumptions must be regularly reviewed both during implementation and after the system enters service to ensure continuing validity.

♦ In the same way that the embedded assumptions and code are updated so they remain consistent with the world, application models, the operational domain, and the solution prescriptions must also be updated.

♦ Whenever bounds change, assumptions must be reviewed.

♦ When considering or implementing tight or loose couplings between separate applications systems or system elements, their bounds and assumption lists must be reviewed in their joint contexts and domain-wide implications identified and acted upon.

♦ Proposed changes to a software system must be examined in relation to the existing bounds and assumption set to avoid incompatibility or other undesirable side effects.

♦ Proposed process changes, at whatever level, must be examined and assessed in relation to their global and local impact. This requires, among other things, knowledge and understanding of any associated feedback loops.

## Treatment of the software process as a feedback system is the exception rather than the rule.

♦ In introducing or modifying management controls, assessment of the global impact of the change must take the global process, its feedback mechanisms, and their global impact into account.

♦ Software architectures that minimize the interdependence of units (modules, components, subsystems, and so on) must be developed. You might, for example, develop system structures that are composed from software units defined by their behavior rather than assemble the system from procedure-oriented elements.[13,14]

♦ Wherever possible, system elements should serve the user directly rather than providing data for further processing by other elements. The concept is to build software systems by assembling functionally independent units, in much the way hardware systems are assembled.

♦ System safety and security must be addressed at the system level (both hardware and software). Physical systems can be made safe; E-type software is intrinsically uncertain and cannot be made safe.

Some of these practical suggestions may be directly adopted, others require research or development. Unquestionably, much work must be done to apply them systematically, economically, and, above all, reliably. But in all cases, I believe the underlying technology is within our grasp. Those who take these facts, attitudes, and principles to heart can expect continued prosperity in their development and the maintenance of satisfactory computer systems. Those who do not will likely face the worst of all times. The ideas I presented here, however briefly, indicate some answers to the question implicit in Yourdon's essay.

With its ever-increasing dependence on computers, the world's future will be bright or disastrous. It is the software engineering community's responsibility to ensure the former. I believe that the material presented contains seeds of a solution. ❖

## References


1. M.M. Lehman, V. Stenning, and W.M. Turski, "Another Look at Software Design Methodology," ICST DoC Res. Rep. 83/13, June 1983; see also *Software Eng. Notes*, Apr. 1984, pp. 38-53.
2. "Flight 501 Failure," *Arianne 501 Inquiry Board Report*, European Space Agency, Paris, 19 July 1996.
3. M.M. Lehman and L.A. Belady, *Program Evolution—Processes of Software Change*, Academic Press, San Diego, Calif., 1985, p. 522.
4. P. Naur and B. Randell, "Software Engineering—Report on a Conference," Scientific Affairs Division, NATO, Brussels, 1969, p. 231.
5. *Representing and Enacting the Software Process, Proc. 4th Int'l Proc. Workshop*, C. Tully, ed., IEEE Comp. Soc. Press, Los Alamitos, Calif., 1989.
6. L. Osterweil, "Software Processes are Software Too, Iteration in the Software Process," *Proc. 3rd Int'l Proc. Workshop*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1987, pp. 79-80.
7. M.M. Lehman, "Feedback in the Software Evolution Process," *Proc. Information and Software Technology, Special Issue on Software Maintenance*, Elsevier, Amsterdam, 1996, pp. 681-686.
8. L.A. Belady and M.M. Lehman, "An Introduction to Program Growth Dynamics," in *Statistical Computer Performance Evaluation*, W. Freiburger, ed., Academic Press, New York, 1972, pp. 503-511.
9. M.M. Lehman and V. Stenning, "FEAST/1—Feedback, Evolution, and Software Technology; Case for Support," (UK) Engineering and Physical Sciences Research Council Research Proposal, Nov. 1995/March 1996, p. 11; available from http://www-dse.doc.ic.ac.uk/~mml/.
10. M.M. Lehman et al., "Metrics and Laws of Software Evolution—The Nineties View," *Proc. Metrics '97*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1997.
11. M.M. Lehman, "Process Models—Where Next?," *Proc. ICSE 19*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1997, pp. 549-552.
12. L.A. Belady and M.M. Lehman, "An Introduction to Growth Dynamics," *Proc. Conf. on Statistical Computer Performance Evaluation*, Academic Press, 1972, pp. 503-511.
13. M.M. Lehman, "The Funnel, A Software Unit or Function Channel," IBM Patent Disclosure P08-76-002, 31 Dec. 1975, p. 8; copies available from the author.
14. W.M. Turski, "Specification as a Theory with Models in the Computer World and in the Real World," P. Henderson, ed., *System Design, Infotech State of the Art Report*, Infotech, London, 1981, pp. 363-377.


## About the Author


**Meir M. (Manny) Lehman** is an emeritus professor and senior research fellow in the Department of Computing at Imperial College in London. He has worked previously for IBM Research, Yorktown Heights, where he established and led a team to architect and design the IMP parallel processing system. He also investigated IBM's programming process, publishing a report on his findings, "The Programming Process," in 1969. Prior to his work at IBM, Lehman worked in the Scientific Department of the Israeli Ministry of Defense, where he led a small team in the design and construction of the SABRAC digital computer. His study of the evolution of OS/360 and other systems, along with his previous programming process study, led to recognition of the software process as a feedback system, his formulating the laws of software evolution, and the concepts of software process dynamics. He joined the staff of Imperial College in 1972. He has served as the Head of the Department of Computing and established and served as chair and director of Imperial Software Technology Ltd (IST).

Lehman received his BSc in mathematics from Imperial College, The University of London. He then began research at Imperial designing the ICCE II arithmetic unit for which he received a PhD and a DSc from the University of London. He was elected a fellow of the IEEE in 1985 and to the Royal Academy of Engineering in 1989.

Address questions about this article to Lehman at the Department of Computing, Imperial College of Science, Technology and Medicine, London SW7 2BZ; +44 171 594 8214; fax +44 171 594 8215 or 44 171 581 8024; mml@doc.ic.ac.uk; http://www-dse.doc.ic.ac.uk/~mml/.