On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle

M. M. Lehman

Imperial College of Science and Technology, London

The paper presents interpretations of some recently discovered laws of evolution and conservation in the large-program life cycle.

Program development and maintenance processes are managed and implemented by people; thus in the long term they could be expected to be unpredictable, dependant on the judgments, whims, and actions of programming process participants (e.g., managers, programmers, and product users). Yet, observed, measured, and modeled regularities suggest laws that are closer to biological laws or those of modern physics than to those currently formulated in other areas subject to human influence (e.g., economics and sociology).

After a brief discussion of the first four laws, to highlight underlying phenomena and natural attributes of the program evolution process, the paper concentrates on a fifth law and shows how, and why, this law represents a conservation phenomenon: the Conservation of Familiarity.

LARGE PROGRAMS

Various published papers ([1] and its bibliography), have discussed the characteristics and dynamics of evolution of large programs and the laws that have emerged from the studies of Belady, Lehman, and others over the past 7 years. The main objective of the present contribution is to discuss one specific aspect of these laws—conservation. However, some general introductory remarks are desirable.

In the first place we should stress that the discussion here is limited to large programs. Until recently these were defined as programs of which at least some part has been concurrently implemented and/or maintained by at least two separately managed groups [2]. Such programs will certainly display the characteristics of largeness [3]. They will, for example, inevitably have the property of variety; they will also be outside the intellectual grasp of any individual; above all, they will undergo a continuous process of maintenance and evolution, generally in somewhat undisciplined fashion.

The above definition is, however, not very satisfying. For one thing, programs not satisfying it may also display some or all of the other characteristics of largeness. Moreover, the definition tends to focus attention on management or sociological issues, whereas our fundamental concern is with programming methodology and the engineering of software. In particular, we should seek to recognize and learn to control the circumstances that lead to the ill effects so often associated with largeness. Will not the adoption of appropriate attitudes, algorithms, methodologies, and programming techniques yield large programs that are well disciplined? Such questions are being addressed via our current research, which includes attempts to formulate more acceptable and useful definitions. We expect to report the results in due course.

THE NATURE OF LAWS OF PROGRAM EVOLUTION

Their Place Within the Spectrum of Scientific Laws

The evolution of large programs, software systems, is clearly not a natural process governed by immutable laws of nature. Changes to a program are neither initiated nor occur spontaneously. People do the work: amend or emend the requirements, the specification, the code, the documentation; repair the system; improve and enhance it. They do this in response

Address correspondence to M. M. Lehman, Dept. of Computing and Control, Imperial College of Science and Technology, London SW7 2BZ.

© Elsevier North Holland, Inc., 1980

0164-1212/80/030213-09$02.25

to fault reports, user requests, business or legal requirements, managerial directives, or their own inspiration. Human thought and judgment play a decisive role in driving and executing the process that results from (and in) a seemingly continuous sequence of exogenous inputs.[1]

Thus, we should not expect to discover laws of program evolution that yield the precision and predictability of the laws of physics [4]. If laws governing large-program evolution can be formulated at all, they must certainly be weaker than those formulated in the biological sciences, where regularity stems from the collective behavior of cellular systems that, even though alive, are nonintelligent, i.e., are not influenced by conscious thought processes, at least at the level of human understanding. Program evolution should not even be expected to display the regularity that has been abstracted into laws in the social and economic sciences, e.g., the so-called Law of Supply and Demand. After all, the programming process is planned and controlled by an organizational and management structure that is sensitive and reactive to the demands, pressures, circumstances, and contingencies of each moment. Thus, superficially, it would seem reasonable to expect the program evolution process to be totally irregular, a reflection at each instance in time of the pressures of the moment.

One of the first and most surprising, yet most fundamental, results of our observations and of the resultant analysis of the dynamics of evolution of some eight programs ranging over a wide spectrum of implementation and usage environments, has been that this is not so. Regularities, trends, and patterns appear and dominate large-program evolution. The common features and patterns of behavior reflect common characteristics [3] from which laws can be deduced; laws that, within the spectrum outlined, lie closer to those that describe the time behavior of biological organisms than those that emerge from the study of socioeconomic systems. Moreover, these laws find very practical application. They provide a basis for large-program life cycle management tools, as well as insight and understanding for improvement of the programming process. As we increasingly rely on the laws for guidance in the development of programming methodology and on a software engineering discipline with its techniques and tools, it becomes vital to develop also a deeper understanding of these laws and the fundamental phenomena or truths that they reflect.

[1]We may regard the inputs as exogenous even though some of them will have been generated while, or as a result of, working on, or using, the system.

The Underlying Cause of Regularity

Once the phenomena have been recognized, the mechanisms underlying them are not difficult to understand. As a totally unintelligent machine, the computer executing a program impacts its environment in a way that is precisely and completely determined by the code in association with any input data. The code is unforgiving; there is no room for logical error or imprecision. Thus any deviation from the required semantic and syntactic structure creates a need for corrective action. Good intentions, hopes of correctness, wishful thinking, even managerial edict cannot change the semantics of the code as written or its effect when executed. Nor can they a posteriori affect the relationship between the desires, needs, and requirements of users and the program specification as presented to the programmers; nor that between the specification and its implementation; nor between any of these and operational circumstances—the real world.

Additionally, the program and its documentation in all their versions—the system—has a damping effect analogous to an ever-increasing mass. It is precisely the freedom to implement changes or additions required for obtaining desired program behavior which is increasingly constrained by existing accumulated code and documentation, past program application and behavior, acquired habits, and implementer and user practices.

The development and implementation of change and of any subsequent corrective action is strongly influenced by the fact that, in itself, program code is also not malleable. Internal coupling, interconnections, and dependencies cause even changes that superficially appear localized to impact and modify the semantic consequences of code elsewhere in the program. Thus when changes are made to the code, deviations from absolute correctness will occur and unexpected side effects will appear; these are very likely to lead to a need for further corrective action. The more intensive the pressure for change, the higher the rate of its implementation, the larger the group of people involved, and the more likely that maintenance must subsequently be diverted from progressive enhancement to repair and cleanup (i.e., on redesign, restructuring, and re-implementation).

We recognize these dependencies as feedback connections over the entire system and application processes and organizations. The resultant interplay of forces for change and expansion of the code on the one hand and the inertia of accumulated code, documentation, and habits on the other, and the interplay over the various processes and the organizational

structures implementing them are believed to be major factors in causing the observed regularity, determining its statistical characteristics and parameters.

These facts alone suffice to explain the consistency of the observations. Recognition of other factors merely strengthens our belief in the reality of the phenomena. In particular, large programs are, as a rule, created within large organizations and for large numbers of users; otherwise they could not be economically justified or maintained. But size causes inertia, and inertia smoothes behavior that might otherwise prove highly irregular. Moreover, the size and complexity of both the program and the application for which the program is intended mean that decisions take time (sometimes considerable) and large numbers of people to implement. Resultant delays provide exogenous pressures and endogenous opportunities for change. The overall circumstances and environment act as a filter, smoothing out the global consequences of individual decisions but, paradoxically, also adding the occasional random disturbance. They also act as an economic and social brake that inhibits or softens decisions that would have too drastic an impact. For example, large budgets can, in general, be neither suddenly terminated nor drastically increased; in practice they can only be changed by fractional amounts. Similarly, a work force cannot be instantaneously hired, retrained, relocated, or dismissed: at best a task force can be sent in, and can cause a local perturbation.

In summary, large-program creation and maintenance occur in an environment with many levels of arbitration, correction, smoothing, and feedback control. A large number of superficially independent (i.e., almost random) inputs are concurrently and successively superimposed to yield time behavior that may be statistically modeled (e.g., described by parameters that have normal distributions). Many, if not all, of the inputs arise from organizational checks and balances, from feedback often also involving the users of the system. The feedbacks in general ensure long-term stability; negative feedback dominates. The alternative, of course, would be instability and disintegration of the system. The existence of regularity, and therefore of laws abstracting that regularity, becomes reasonable and understandable.

The Gross Nature of the Laws

The detailed behavior of the programming process and of the system that is the object of process activity is the consequence of human decision and action. Specific individual events in the life cycle of the system, the system development and maintenance process, cannot therefore be predicted more precisely than can the specific acts of participating or interacting individuals [4]. Any laws can only relate to the gross (statistical) dynamics of large-program systems over a period of time, but as such they yield insight and understanding that should permit improvement of the programming process and advance the development of software engineering science and practice.

Feedback Consequences of Increasing Understanding of the Process

Increasing understanding of the dynamics of the large-program life cycle raises another problem: To what extent will the discovery and acquisition of knowledge and understanding of the laws that regulate the programming process, by an environment previously unaware of or insensitive to their existence, lead to changed behavior and thus invalidation of the laws? How will managerial awareness of and conscious reaction to the laws affect the very nature of these laws? Since they reflect the joint behavior of people, the laws are unlikely to be immutable. Surely they may be expected to change as understanding of system behavior increases [4].

Space does not permit us to address this question in detail. We merely assert that the present laws reflect deeply rooted aspects of human and organizational behavior. Associated with the mechanistic forces that define, control, and execute the automatic computational process, they are sufficiently fundamental to be treated as absolute, at least in our generation. As knowledge of them is permitted to impact the programming process, and as programming technology advances, they may require restatement or revision, or become irrelevant or obsolete: but for the time being, we must accept and learn to use them. To ignore them is foolish and costly.

THE LAWS

We now comment briefly on the laws summarized in Table 1, so as to expose some of the more fundamental truths that they reflect. These laws have been fully discussed in earlier publications ([1] and its bibliography).

1. The Law of Continuing Change

This first law reflects a phenomenon intrinsic to the very being of large programs. It arises, at least in part, from the fact that the world (in this case, the computing environment) undergoes continuing change.

Table 1. Five Laws of Program Evolution

1. **CONTINUING CHANGE**

   A program that is used and that, as an implementation of its specification, reflects some other reality, undergoes continuing change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the program with a recreated version.

2. **INCREASING COMPLEXITY**

   As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it.

3. **THE FUNDAMENTAL LAW** (OF PROGRAM EVOLUTION)

   Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances.

4. **CONSERVATION OF ORGANIZATION STABILITY** (INVARIANT WORK RATED)

   The global activity rate in a project supporting an evolving program is statistically invariant.

5. **CONSERVATION OF FAMILIARITY** (PERCEIVED COMPLEXITY)

   The release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

All programs are models of some part, aspect, or process of the world. They must therefore be changed to keep pace with the needs and the potential of a changing environment. If they are not, they become progressively less relevant, useful, and cost effective.

Of course all complex systems evolve. Living, social, and artificial systems [5] all respond to reactions and pressures from their environments by changes in operational pattern, function, and structure. Software is distinguished not by the fact that evolution occurs, but by the way in which it occurs.

The pressure for change with respect to any large program is felt almost daily. A widely held view is that the details of the desired change need "only" be written down and then applied without further real effort (or so it would seem) to all instances of the system. As a consequence, changes are superimposed (change upon change upon change) in a *current embodiment*. This contrasts strongly with normal industrial practice where conceptual changes are inputs to a redesign and recreation process that ultimately produces a *new instance* of the system. Moreover, any repairs to software are a departure from the original conceived design and/or implementation rather than the replacement of a worn-out part. In addition there is, in software, absolutely no decay or death process through which older parts of the system wear out and are replaced, or disintegrate and disappear out of the system. Removals, with replacements and additions, occur as a result of system-extraneous pressures and effort, and then *only* as the result of conscious and directed effort on the part of people.

The evolution of software differs from that of other systems in many other ways, but it is not our concern here to prove that software is different or to state in detail how it is different. We ask the reader to accept that difference and then to ponder the practical implications.

These implications are, we assert, strongly influenced by the fact of continuing evolution, recognized and formalized by the first law. The causes of continuing change are seen as stemming, at least in part, from the continuing evolution of the environments, in combination with the "soft" nature of programming technology. Hence changeability and all it implies must be accepted as a basic requirement for software systems. The degree to which it is achieved and maintained may make all the difference, in the development, application, and cost effectiveness of a system between success and failure, profitability and loss.

2. The Law of Increasing Complexity

Our second law may be seen as an analogue of the second law of thermodynamics. More correctly, both of these laws should perhaps be viewed as descriptions of instances of a still more fundamental natural phenomenon. In our case, the law is a consequence of the fact that a system is changed to improve its capabilities and to do so in a cost-effective manner. Specific change objectives develop from a consideration of factors that indicate immediate or measurable benefit. They are expressed in terms of performance targets, system resources required during execution, implementation resources, completion dates, fiscal objectives and constraints, and so on.

In cases with multiple objectives, it is generally impossible to fulfill all of them optimally. Hence the completed project and system must represent a compromise that results from judgments and decisions taken during the planning and implementation processes, often on the basis of time and group- or management-local optimization.

Structural maintenance is rarely mentioned in objectives. Being antiregressive [6], it yields no immediate or visible benefit but *merely (sic)* prevents deterioration. Thus structure, being excluded from stated project objectives, will inevitably suffer; each change will degrade the system a little more. The resultant accumulation of gradual degradation ultimately leads to the point where the system can no longer be cost effectively maintained and enhanced

unless and until redesign and cleanup or reimplementation is undertaken and successfully completed.

The law suggests that large-program structure must not only be created but must also be maintained if decay is to be avoided or, at least, postponed. Planning and control of the maintenance and change process should seek to ensure the most cost-effective balance between functional and structural maintenance based on the lifetime of the program. Models and tools are required to facilitate such balance.

3. The Fundamental Law of Large-Program Evolution

This was previously called *the Law of Statistically Smooth Growth* [7]. It expresses the observation already made above that large-program evolution does not simply reflect, at each instant and in each period, the decisions and actions of the people in the environment in which it is maintained and in which it is used. The law states that, at least in the current state of the art, there *exists* a dynamics whose characteristics are largely determined during the conception and early life of the system of the maintenance process and of the maintenance organization. The characteristics of this dynamics increasingly determines the gross trends of the maintenance and enhancement process. System, project and organizational *history* play an important role in the program evolution process, while feedback effects the additional, inherent, factor producing a self-stabilizing control process, itself evolving. Thus cyclic effects emerge, though not necessarily with pure periods.

This law is particularly important in guiding our understanding of the software creation and maintenance process. However, its tacit acceptance (for the time being) also helps the manager and the planner to remain realistic. We are not free to set and achieve arbitrary design, performance, and work targets [8]. Project constraints are at present not all under our control. Thus we must accept any limits they imply until they can be or have been changed. Moreover, the law implies that models of large-program evolution can be created and be exploited as planning and control tools.

4. The Law of Organizational Stability

This was previously referred to as the Law of Invariant Work Rate [7]. It reflects the fact that, in general, human organizations seek to achieve and maintain stability or stable growth. As suggested above, sudden substantial changes in such managerial parameters as staffing, budget allocations, manufacturing levels, and product types are avoided; as a rule, such changes are not even possible. A variety of managerial, union, and governmental checks, balances, and controls ensure smooth overall progress to the ever-changing, ever-distant objective of the organization (or its eventual collapse). In addition, the fourth law also reflects the organizational response to the limitation that, we shall show, underlies the fifth law.

Thus with hindsight it becomes clear that the discovery of an invariant activity measure (statistically invariant, as when its parameters are always normally distributed with constant mean and variance) could have been anticipated. What is not really understood is why, in large-program maintenance projects, measures of work input rate should be the quantities to display such invariance. However, the fact remains that for the systems observed, the count of modules changed (handled) or changes made per unit of time, as averaged over each release interval, has been statistically invariant over the period of observation. The limitations implied by this invariance can only be temporarily overcome. If they need to be overstepped, the consequences should be identified and must be accepted.

5. The Law of Conservation of Familiarity (Perceived Complexity)

In [7], this was referred to as the Law of Incremental Growth Limits. Its discovery was based on data from three systems, each of which was made available to users release by release. In each case the incremental growth of the program varied widely from one release to the next, but the average over a relatively large number of releases remained remarkably constant; that is, a high-growth release would tend to be followed by one with little or no growth, or even by system shrinkage; or two releases, each displaying near average growth, would be followed by one with only slight growth. Moreover, releases for which the net growth exceeded about twice the average proved to be minor disasters (or major ones, depending on the degree of excess) with poor performance, poor reliability, high fault rates, and cost and time overruns.

The evidence suggests that initial release quality is a nonlinear function of the incremental growth. From a more complete phenomenological analysis along the lines outlined below, we hypothesize that quality is expontentially related to the *release content*, that is, to the amount of change implemented in the release.

It should perhaps be added that at this time we know of no *precise* way of defining or measuring release content that takes into account the size, complexity, and interrelationships of system and code

changes, additions, and deletions. It is not even clear that a metric can be found. If it can, then such a universal measure must also be sensitive to the characteristics of the systems and the environments involved in or affected by the changes.

The absence of adequate definitions and measures is no reason for ignoring observed phenomena and their implications. The gradual clarification and evolution of concepts, definitions, and measures is fundamental to the very nature of the phenomenological approach we have adopted, an approach that is considered essential for significant progress in mastering the problems of software engineering. One first observes and measures some phenomenon, then seeks models, interpretations, and explanations in more fundamental terms; subsequently, one can seek measures and devise experiments that confirm, reject, modify, and/or extend the original hypotheses, interpretations, and explanations; and so on.

INTERPRETATION OF THE FIFTH LAW

Change and Refamiliarization

The phenomenon abstracted by the fifth law was detected at a very early stage of the evolution dynamics studies and was featured in the earliest models [9]. It has been applied as a planning and control parameter for a number of years. The explanation, however, has only recently become apparent. The release process has always been understood as fulfilling a stabilization role [9]. Once a large program is in general use, its code and documentation are normally in a state of flux. A fault is fixed locally; in other installations it is perhaps fixed differently or not at all. Minor or major changes and local adaptations are made. Code is changed without a corresponding change to documentation. Documentation is changed to correspond to observed behavior without a full and detailed analysis of the precise semantics of the code within the context of the total system under all possible environmental conditions. Only at the moment of release does there exist an authoritative version of the program, the code, and its documentation. Even this may include multiple versions of modules, say, for more or less clearly defined alternative situations.

Some time after the release of a program or program version, each designer, implementer, tester, salesman, and user that has been exposed to or worked with the system will have become thoroughly conversant with, at least, those of its attributes and characteristics that are considered at all relevant. The resultant familiarity will have bred some degree of relaxation, of ability to work with the program in order to accomplish specific objectives. The program will be manipulated without uncertainty or concern and used without (apparent) need for concentrated thought. External perception of a program's intrinsic complexity will be at a minimum. For people working consistently on or with the program, its perceived complexity may be said to approach zero.

As changes are introduced, as the new release is gradually created and becomes available, new and unfamiliar code appears. The program behaves differently in execution, in its interaction with and impact on the environment. Pagination in the previously familiar documentation has changed and any need for reference entails a major search. The system has become uncomfortably unfamiliar, the degree of unfamiliarity depending on the magnitude and extent of the change.

A major intellectual effort is now required by each person involved before any completely successful and cost-effective interaction with the new system can occur. The system has suddenly become strange. Its perceived complexity is high.

Even those who participated in the preparation of the new release will normally have been involved directly with only a small part of the changes, a small portion of the system. They too must now learn to understand the new system in its totality. Moreover, until the complete system is available, all acquisition of knowledge and understanding of the changes and of the new system must be based on reading of code and documentation text, or on partial execution of system components on test cases or system models. At least some part of system-internal interactions or dependencies will be absent in such an environment. Only with final integration of the new system does the full executable program become available. We suggest that when the release content exceeds some critical amount, only operational experience with the complete system can bring or restore the degree of knowledge and familiarity, the global viewpoint, that is essential for subsequent cost-effective maintenance, enhancement, and exploitation of the large program.

Thus, in general, at the moment of release or shortly before that time a major learning effort will begin that involves all those associated with the system, not just the users. All changes and additions must be identified, understood, and *experienced*, their significance appreciated within the operational context of the total system. Once this has been done, the old degree of comfort with the system will return and its perceived complexity once again will approach zero; the level of familiarity has been restored.

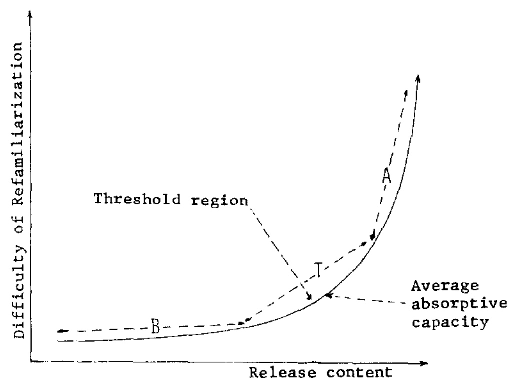
The amount of work that must be invested, the intellectual effort required to achieve this, depends among other factors on the attitudes of people, on the organization, and on the number, magnitude, and complexity of changes introduced. Because changes to the system interact with one another, because changes implemented in the same release must be understood in the context of all other changes being concurrently implemented as well as in the context of the unchanged parts of the system and of past and future applications, the relationship between the release content and the amount of intellectual effort needed to absorb the changes introduced by a new release fully is at least quadratic. But whatever the precise relationship between the difficulty of restoring familiarity with the program and the magnitude of the release content, it will be of the general form indicated in Figure 1.

The axes of the curve are not calibrated since at present neither concept nor suitable measures are well defined. Our concept of "difficulty" relates to that of Norden and of Putnam [10,11]. They, however, are concerned with difficulty of implementation, whereas our concern is with understanding the changes within the context of the total system and their implications with regard to its operational behavior. Although related, since one cannot (should not?) implement without understanding, the concepts are clearly not identical.

The Averaging of Ability Through Human Interactions

It must be left to the future to identify or define measures and provide an improved formulation of the fifth law. Meanwhile we must clarify the basic concepts and increase understanding of, at least, the phenomenology; thereby we shall provide a basis for ultimate formalization.

Figure 1. Difficulty-release-content relationship. A, above threshold: B, below threshold.



Everyone's ability to master a new or changed object is limited, though people clearly differ in their ability to absorb new knowledge (e.g., to achieve full understanding of the changed program). Thus the impact of changes will vary from person to person according to many factors that will include, but are not limited to, their learning ability and absorptive capacity. For a given large program with which many people are inevitably involved, the direct and indirect costs of familiarization (delays incurred, mistakes made, destructuring, etc.) relate to the average ability of all the people involved. This average will not change significantly with time or, with the detailed composition of the group.

In the implementation environment, for example, although the above-average person will regain mastery more quickly, make fewer mistakes, and achieve a temporary advantage (which might lead to promotion or transferal to another project), the below-average person will fall behind, perhaps lose contact, make more mistakes, and do more damage. This person may well be reassigned to a less demanding role, one with less impact, or even be fired. But the damage will have been done; others will have to do additional work to apply corrective action. Since hiring policies are related to the already established makeup of the project, the average capacity to understand will, at best, remain unchanged; more probably it will decline [12].

In the application and user environment, the capable individual will master the changes relatively quickly and carry on with assigned responsibilities, experience minimal perturbation, and cause no impact on others. The less capable, on the other hand, will have to discuss with others their difficulties in fully appreciating the changes. They will even misinterpret documentation or system behavior and report difficulties or faults that are, in fact, nonexistent. Such discussions or reports will cause delays or disruptions in the project, and may even lead to erroneous repair. Once again the presence of persons with greater than average difficulty in refamiliarization has a project impact that ranges beyond the immediate bound of responsibility of these individuals. It results in an *overall* slowdown in the return to normalcy after change, the time required being determined by *average* ability.

For different organizations, systems, structures, methodologies, and processes, the average level will, of course, be different. This implies that any models will contain exogenous variables. These variables point to a potential for improving the average absorp-

tion level, once the phenomenon, the organization, and the programming process are understood.

Conservation of Familiarity and Statistically Invariant Release Content

Given the above insights into (1) the increased difficulty of understanding changes and their implications as release content increases and (2) the mechanism of the slow down of both utilization and further evolution as system structure deteriorates, the number of faults increases, documentation lags, and performance declines, we are now in a position to appreciate the fifth law.

If the release content, the magnitude of change and/or the incremental growth, is less than some threshold region *T* (Figure 1), the integration and operational installation of the new system should be fairly straightforward. No major problems should be experienced in mastering the new release; it may well be that the change may be absorbed and familiarity restored without actual operational disturbance.

The very ease of the refamiliarization process in conjunction with the never-ending search for productivity growth will, however, create a managerial climate in which more ambitious releases that will challenge system capability and may well flout its natural parameters will be attempted. A pressure is created that tends to move subsequent releases from the *B* region (Figure 1) into or above the threshold region *T*.

When the release content lies in *T* (which may not be precisely delineable), quality, performance, completion, and installation problems are to be expected. Slippage and cost overrun will probably occur. A subsequent release may be required to clean up the system and restore it to a state that permits further cost-effective evolution. This experience will certainly not encourage management to demand an increase in release content. The next release will tend to be in the same threshold region or even below it.

Finally, if a release is attempted whose content exceeds that of the *T* region and moves into *A*, serious problems will be encountered. Slippage and cost overrun will occur unless plans take account of the greatly increased difficulties that will be experienced. If not properly planned, such an attempt may lead to the effective collapse of the system or, as we have observed in at least two instances, to an effect that we have termed *system fission*. Since only release of the system to end users and to the developers provides full exposure, even when adequate resources and time have been provided, such a release will still have to be followed by a restoration or clean-up release. This results in one or more successor releases in the *B* region of the characteristic curve of Figure 1.

It was the repeated observation of the above patterns of release behavior that suggested the analysis and led to the insights summarized in the preceding paragraphs. Our analysis suggests that the consequences of feedback in the process, in conjunction with the nonlinear characteristics indicated in Figure 1, lead (over several releases) to stabilization of release content in or just below the threshold region. We have not yet attempted to create an analytic model of this effect, but it should not prove too difficult to build and validate [13].

The fifth law abstracts both the observations and their interpretation including the emergence of invariant average incremental growth or release content. The latter is also a consequence of the additional exogenous pressure for accelerated functional growth of content that is characteristic of large-program applications and, in general, of organizational environments. Once again the law suggests that managers and planners take note of project and system invariances; when formulating plans, they must respect the limitations the invariances imply or accept the inevitable consequences.

FINAL COMMENTS

The first recognition of the laws discussed was based entirely on an examination and analysis of data from a variety of programs and systems, both large and not so large. To make the transition from phenomenology to science, however, the laws, once formulated, must be examined in their own right. The laws of large-program development and evolution are now beginning to be understood in this way. They are seen to express very basic attributes of computing, of the programming development, maintenance, and usage processes, of programs themselves, and of the organizations and environments in which these activities are carried out.

Once this interpretation of the laws in terms of more fundamental phenomena has been achieved, the old data must be reexamined and new information examined in the light of the laws *as understood*. Deviations must be explained and interpreted; contradictions may require reformulation and reinterpretation of a law, or even its rejection.

There is, of course, nothing new in these comments: they form the very basis of the scientific method. They are added here, however, to assert the belief that the laws as formulated have been substantiated by experience and by experimental data to the point where they can stand in their own right until

evidence and developing insight and understanding demand their change—or until we can so change system structure, process methodology and characteristics, and programmer and user practice and habits, that the laws as formulated no longer apply.

ACKNOWLEDGMENT

Although the author must bear full responsibility for the present text, the understanding, concepts, and interpretations here presented were developed over a period of years in close collaboration with L. A. Belady, F. M. Parr, and others. My grateful thanks and appreciation are due to all of them. More immediately I am deeply grateful to W. M. Turski for an extended interchange of ideas and for his detailed critique of the draft manuscript of this paper. The clarity of his thinking and expression and our mutual stimulation has resulted in a major advance in the formalization and understanding of the evolution dynamics concepts, though these advances are excluded from the present text except in that we have revised the wording of the laws to that presented in Table 1.

REFERENCES

1. M. M. Lehman, The Software Engineering Environment, in *Structured Software Development*, Infotech State of the Art Report, 1979, Vol. 2, pp. 147–163.
2. M. M. Lehman, Laws and Conservation in Large-Program Evolution, in *Second Software Life Cycle Management Workshop*, Atlanta, Georgia, 21–22 August, 1978, IEEE Cat. No. 78CH1390-4C, pp. 140–145.
3. L. A. Belady and M. M. Lehman, Characteristics of Large Systems, in *Research Directions in Software Technology*, Proceedings of the Conference on Research Direction in Software Technology, 19–12 October 1977, Brown University, Providence, Rhode Island. Sponsored by the Tri-Services Committee of the DOD. MIT Press, Cambridge, Massachusetts, Part 1, Chapter 3, pp. 106–142.
4. M. M. Lehman, Human Thought and Action as an Ingredient of System Behaviour, in *Encyclopaedia of Ignorance* (Duncan and Weston-Smith, eds.), Pergamon, New York, 1977, pp. 347–354.
5. H. A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts, 1969.
6. M. M. Lehman, Programs, Cities, Students—Limits to Growth? in *Programming Methodology* (D. Gries, ed.), first published as an inaugural lecture, 14 May 1974 and in the Imperial College of Science and Technology Inaugural Lecture Series, vol. 9, 1970–1974, pp. 211–229. Springer Verlag, New York, 1979, pp. 42–69.
7. M. M. Lehman, Laws of Program Evolution—Rules and Tools for Programming Management, in *Why Software Projects Fail*, Proceedings of Infotech State of the Art Conference, 9–11 April 1978, pp. 11/1–11/25.
8. F. P. Brooks, *The Mythical Man-Month—Essays on Software Engineering*, Addison-Wesley, Reading, Massachusetts, 1975.
9. L. A. Belady and M. M. Lehman, *Programming System Dynamics or the Meta-Dynamics of Systems in Maintenance and Growth*, IBM Research Report RC3546, September 1971, p. 30.
10. P. V. Norden, Project Life Cycle Modelling, in *Software Phenomenology—Working Papers of the (First) SLCM Workshop*, Airlie, Virginia, August 1977, Published by ISRAD/AIRMICS, Computer Systems Command, U.S. Army, Fort Belvoir, Virginia, December 1977, pp. 217–306.
11. L. H. Putnam, The Influence of the Time-Difficulty Factor in Large Scale Software Development, *loc. cit.*, pp. 307–312.
12. M. M. Lehman, *Mediocrity in Middle Management*, unpublished ms, 1969.
13. C. M. Woodside, *A Mathematical Model for the Evolution of Software*, Dept. of Computing and Control Research Report CCD 79155, April 1979.