Programs, Life Cycles, and Laws of Software Evolution

MEIR M. LEHMAN, SENIOR MEMBER, IEEE

Abstract—By classifying programs according to their relationship to the environment in which they are executed, the paper identifies the sources of evolutionary pressure on computer applications and programs and shows why this results in a process of never ending maintenance activity. The resultant life cycle processes are then briefly discussed. The paper then introduces laws of Program Evolution that have been formulated following quantitative studies of the evolution of a number of different systems. Finally an example is provided of the application of Evolution Dynamics models to program release planning.

I. Background

A. The Nature of the Problem

THE TOTAL U.S. expenditure on programming in 1977 is estimated to have exceeded $50 billion, and may have been as high as $100 billion. This figure, which represents more than 3 percent of the U.S. GNP for that year, is already an awesome figure. It has increased ever since in real terms and will continue to do so as the microprocessor finds ever wider application. Programming effectiveness is clearly a significant component of national economic health. Even small percentage improvements in productivity can make significant financial impact. The potential for saving is large.

Economic considerations are, however, not necessarily the main cause of widespread concern. As computers play an ever larger role in society and the life of the individual, it becomes more and more critical to be able to create and maintain effective, cost-effective, and timely software. For more than two decades, however, the programming fraternity, and through them the computer-user community, has faced serious problems in achieving this [1]. As the application of microprocessors extends ever deeper into the fabric of society the problems will be compounded unless very basic solutions are found and developed.

B. Programming

The early 1950's had been a pioneering period in programming. The sheer ecstasy of instructing a machine step by step to achieve automatic computation at speeds previously undreamed of, completely hid the intellectually unsatisfying aspects of programming; the lack of a *guiding theory* and *discipline*; the largely *hit* or *miss* nature of the process through which an acceptable program was finally achieved; the ever present uncertainty about the *accuracy*, even the *validity*, of the final result.

More immediately, the gradual penetration of the computer into the academic, industrial, and commercial worlds led to serious problems in the provision and upkeep of satisfactory programs. It also yielded new insights. Programming as then practiced required the breakdown of the problem to be solved into steps far more detailed than those in terms of which people thought about it and its solution. The manual generation of programs at this low level was tedious and error prone for those whose primary concern was the result; for whom programming was a means to an end and not an end in itself. This could not be the basis for widespread computer application.

Thus there was born the concept of *high-level*, problem-oriented, *languages* created to simplify the development of computer applications. These languages did not just raise the level of detail to which programmers had to develop their view of the automated problem-solving process. They also removed at least some of the burdens of procedural organization, resource allocation and scheduling, burdens which were further reduced through the development of operating systems and their associated job-control languages. Above all, however, the high-level language trend permitted a fundamental shift in attitude. To the discerning, at least, it became clear that it was not the programmer's main responsibility to instruct a machine by defining a step-by-step computational process. His task was to state an algorithm that correctly and unambiguously defines a mechanical procedure for obtaining a solution to a given problem [2], [3]. The transformation of this into executable and efficient code sequences could be more safely entrusted to automatic mechanisms. The objective of language design was to facilitate that task.

Languages had become a major tool in the hands of the programmer. Like all tools, they sought to reduce the manual effort of the worker and at the same time improve the quality of his work. They permitted and encouraged concentration on the intellectual tasks which are the real province of the human mind and skill. Thus, ever since, the search for better languages and for improving methodologies for their use, has continued [4].

There are those who believe that the development of *programming methodology*, high-level languages and associated concepts, is by far the most important step for successful computer usage. That may well be, but it is by no means sufficient. There exists a clear need for additional methodologies and tools, a need that arises primarily from program maintenance.

C. Program Maintenance

The sheer level of programming and programming-related activity makes its disciplining important. But a second statistic carries an equally significant message. Of the total U.S. expenditure for 1977, some 70 percent was spent on program *maintenance* and only about 30 percent on program *develop-*

Manuscript received February 27, 1980; revised May 22, 1980.
The author is with the Department of Computing, Imperial College of Science and Technology, 180 Queen's Gate, London SW7 2BZ, England.

0018-9219/80/0900-1060$00.75

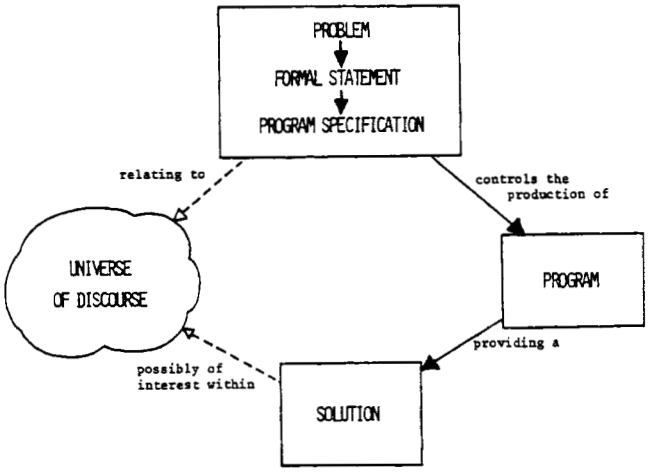

Fig. 1. *S*-programs.

ment. This ratio is generally accepted by the software community as characteristic of the state of the art.

Some clarification is, however, necessary. For software the term *maintenance* is generally used to describe *all* changes made to a program after its first installation. It therefore differs significantly from the more general concept that describes the *restoration* of a system or system component to its *former* state. Deterioration that has occurred as a result of usage or the passage of time, is corrected by repair or replacement. But software does not deteriorate spontaneously or by interaction with its operational environment. Programs do *not* suffer from wear, tear, corrosion, or pollution. They do not change unless and until *people* change them, and this is done whenever the current behavior of a program in execution is found to be wrong, inappropriate, or too restricted. *Repair* actually involves changes *away* from the previous implementation. Faults being corrected during maintenance can originate in any phase of the program *life cycle* (Section III).

Moreover, in hardware systems, major changes to a product are achieved by *redesign*, retooling, and the construction of a new model. With programs *improvements* and *adaptations* to a changing environment are achieved by alterations, deletions, and extensions to existing code. New capability, often not recognized during the earlier life of the system, is superimposed on an existing structure without redesign of the system as a whole.

Since the term software maintenance covers such a wide range of activities, the very high ratio of maintenance to development cost does not necessarily have to be deprecated. We shall, in fact, argue that the need for continuing change is *intrinsic* to the nature of computer usage. Thus the question raised by the high cost of maintenance is not exclusively how to control and reduce that cost by avoiding errors or by detecting them earlier in the development and usage cycle. The *unit cost of change* must initially be made as low as possible and its growth, as the system ages, minimized. Programs must be made more *alterable*, and the alterability *maintained* throughout their lifetime. The change process itself must be *planned* and *controlled*. Assessments of the economic viability of a program must include *total lifetime costs* and their life cycle *distribution*, and not be based exclusively on the initial development costs. We must be concerned with the cost and effectiveness of the life-cycle process itself and not just that of its product.

The opening paragraph highlighted the high cost of software and software maintenance. The economic benefit and potential of the application of computers is, however, so high that present expenditure levels may well be acceptable, at least for certain classes of programs. But we must be concerned with the fact that *performance*, *capability*, *quality in general*, cannot at present be designed and built into a program *ab initio*. Rather they are gradually achieved by evolutionary change and refinement. Moreover, when desirable changes are identified and authorized they can usually not be implemented on a time scale fixed by external need. *Responsiveness* is poor. And as mankind relies more and more on the software that controls the computers that in turn guide society, it becomes crucial that people control absolutely the programs and the processes by which they are produced, throughout the useful life of the program. To achieve this requires *insight*, *theory*, *models*, *methodologies*, *techniques*, *tools:* a *discipline*. That is what software engineering is all about [5]–[8].

II. Programs as Models

A. Programs

Program evolution dynamics [9 and its bibliography] and the laws [2], [3], [10], [11] discussed in the next section, have always been associated with a concept of *largeness*, implying a classification into large and nonlarge programs. Great difficulty has, however, been experienced in defining these classes. Recent discussions [12] have produced a more satisfying classification. This is based on a recognition of the fact that, at the very least, any program is *a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse*. The classification categorizes programs into three classes, S, P, and E. Since programs considered large by our previous definition will generally be of class P or E, the new classification represents a broadening and firming of the previous viewpoint.

B. S-Programs

S-programs are programs whose function is formally defined by and derivable from a *specification*. It is the programming form from which most advanced programming methodology and related techniques derive, and to which they directly relate. We shall suggest that as programming methodology evolves still further, all large programs (software systems) will be constructed as structures of *S*-programs.

A specific problem is stated: lowest common multiple of two integers; function evaluation in a specified domain; eight queens; dining philosophers; generation of a rectangle of a size within given limits on a specific type of visual display unit (VDU). Each such problem relates to its universe of discourse. It may also relate directly and primarily to the external world, but be completely defined, e.g., the *classical* travelling salesman problem.

As suggested by Fig. 1 the specification, as a formal definition of the problem, directs and controls the programmer in his creation of the program that defines the desired solution. Correct solution of the problem as *stated*, in terms of the programming language being used, becomes the programmer's sole concern. At most, questions of elegance or efficiency may also creep in.

The problem statement, the program and the solution when obtained may relate to an external world. But it is a casual, noncausal relationship. Even, when it exists we are free to

change our interest by redefining the problem. But then it has a *new program* for its solution. It may be possible and time-saving to derive the new program from the old. But it is a *different* program that defines a solution to a *different* problem.

When this view can be legitimately taken the resultant program is conceptually static. One may change it to improve its clarity or its elegance, to decrease resource usage when the program is executed, even to increase confidence in its correctness. But any such changes must not effect the mapping between input and output that the program defines and that it achieves in execution. Whenever program text has been changed or transformed [13], [14] it must be shown that either the input–output relationship remains unchanged, or that the new program satisfies a new specification defining a solution to a new problem. We return to the problem of correctness proving in Section II-E.

C. P-Programs

Consider a program to play chess. The program is completely specified by the rules of chess plus procedure rules. The latter must indicate how the program is to analyze the state of the game and determine possible moves. It must also provide a decision rule to select a next move. The procedure might, for example, be to form the tree of all games that may develop from any current state and adopt a minimax evaluation strategy to select the next move. Such a definition, while complete, is naive, since it is not implementable as an executing program. The tree structure at any given stage is simply too large, by many orders of magnitude, to be developed or to be scanned in feasible time. Thus the chess program must introduce approximation to achieve practicality, judged as it begins to be used, by its performance in actual games.

A further example of a problem that can be precisely formulated but whose solution must inevitably reflect an approximation of the real world is found in weather prediction. In theory, global weather can be modeled as accurately as desired by a set of hydrodynamic equations. In the actual world of weather prediction, approximate solutions of modified equations are compared with the weather patterns that occur. The results of such comparisons are interpreted and used to improve the technology of prediction, to yield ever more usable programs, whose outputs, however, always retain some degree of uncertainty.

Finally consider the travelling salesman problem as it arises in practice, for example from a desire to optimize continuously in some vaguely defined fashion, the travel schedule of salesmen picking up goods from warehouses and visiting clients. The required solution can be based on known approaches and solutions to the classical problem. But it must also involve considerations of cost, time, work schedules, timetables, value judgments, and even salesmens' idiosyncracies.

The problem statement can now, in general, no longer be precise. It is a model of an abstraction of a real-world situation, containing uncertainties, unknowns, arbitrary criteria, continuous variables. To some extent it must reflect the personal viewpoint of the analyst. Both the problem statement and its solution approximate the real-world situation.

Programs such as these are termed *P*-programs (real world *problem* solution). The process of creating such programs is modeled by Fig. 2 which shows the intrinsic feedback loop that is present in the *P*-situation. Despite the fact that the problem to be solved can be precisely defined, the acceptability of a solution is determined by the environment in which it is

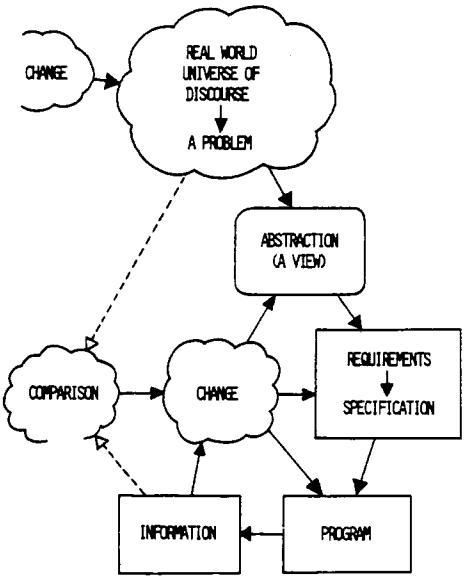

Fig. 2. *P*-programs.

embedded. The solution obtained will be evaluated by comparison with the real environment. That is, the critical difference between *S* and *P*-programs is expressed by the comparison cloud in Fig. 2. In *S*-programs, judgments about the correctness, and therefore the value, of the programs relate by definition *only to its specification*, the problem statement that the latter reflects. In *P*-programs, the concern is not centered on the problem statement but on the *value* and *validity* of the solution obtained *in its real-world context*. Differences between data derived from observation and from computation may cause changes in the world view, the problem perception, its formulation, the model, the program specification and/or the program implementation. Whatever the source of the difference, ultimately it causes the program, its documentation or both to be changed. And the effect or impact of such change cannot be eliminated by declaring the problem a *new* problem, for the real problem has always been as now perceived. It is the perception of users, analysts and/or programmers that has changed.

There is also another fact of life that needs to be considered. Dissatisfaction will arise not only because information received from the program is incomplete or incorrect, or because the original model was less than perfect. These are imperfections that can be overcome given time and care. But *the world too changes* and such changes result in additional pressure for change. Thus *P*-programs are very likely to undergo never-ending change or to become steadily less and less effective and cost effective.

D. E-Programs

The third class, *E*-programs, are inherently even more change prone. They are programs that mechanize a human or societal activity.

Consider again the travelling salesman problem but in a situation where several persons are continuously en route, carrying products that change rapidly in value as a function of both time and location, and with the pattern of demand also changing continuously. One will inevitably be tempted to see this situation as an application in which the system is to act as a continuous dispatcher, dynamically controlling the journeys and calls of each individual. The objective will be to maximize

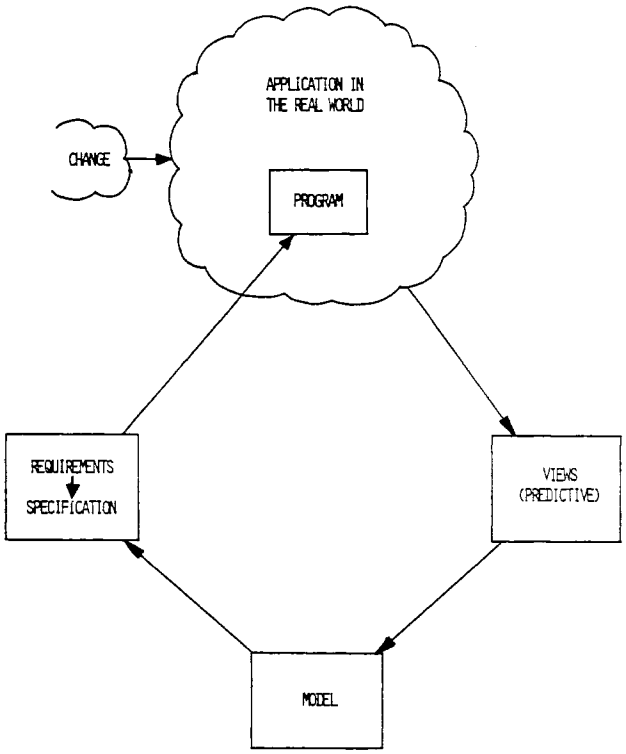

Fig. 3. *E*-programs-The basic cycle.

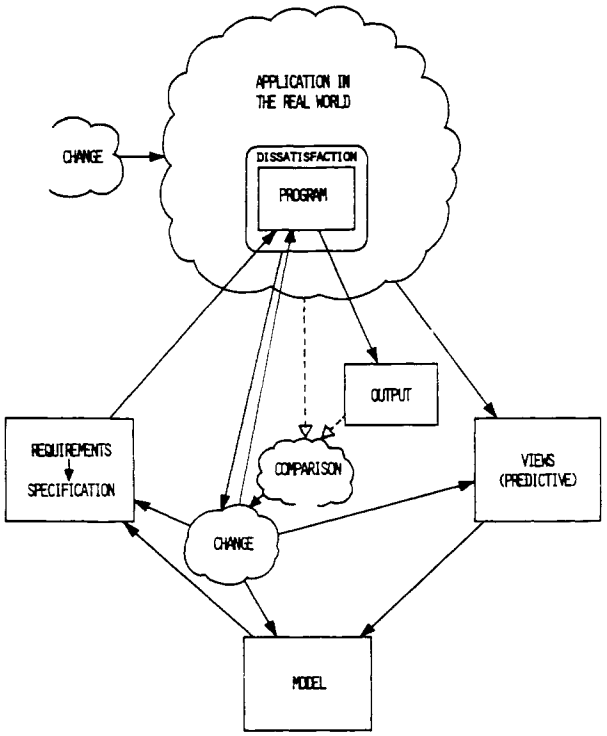

Fig. 4. *E*-programs.

profit, minimize loss, expedite deliveries, maintain customer satisfaction or achieve some optimum combination of the factors that are accepted as the criteria for success. How does this situation differ from that discussed in the previous sections?

The installation of the program together with its associated system–radio links to the salesmen, for example–change the very nature of the problem to be solved. *The program has become a part of the world it models*, it is *embedded* in it. Conceptually at least the program as a model contains elements that model itself, the consequences of its execution.

The situation is depicted in Figs. 3 and 4. Even without considering program execution and evaluation of its output in the operational environment, the *E*-situation contains an intrinsic feedback loop as in Fig. 3. Analysis of the application to determine requirements, specification, design, implementation now all involve extrapolation and prediction of the consequences of system introduction and the resultant potential for application and system evolution. This prediction must inevitably involve opinion and judgment. In general, several views of the situation will be combined to yield the model, the system specification and, ultimately, *a* program. Once the program is completed and begins to be used, questions of correctness, appropriateness and satisfaction arise as in Fig. 4 and inevitably lead to additional pressure for change.

Examples of *E*-programs abound: computer operating systems, air-traffic control, stock control. In all cases, the behavior of the application system, the demands on the user, and the support required will depend on program characteristics as experienced by the users. As they become familiar with a system whose design and attributes depend at least in part on user attitudes and practice before system installation, users will modify their behavior to minimize effort or maximize effectiveness. Inevitably this leads to pressure for system change. In addition, system exogenous pressures will also cause changes in the application environment within which the system operates and the program executes. New hardware will be introduced, traffic patterns and demand change, technology advance and society itself evolve. Moreover the nature and rate of this evolution will be markedly influenced by program characteristics, with a new release at intervals ranging from one month to two years, say. Unlike other artificial systems [15] where, relative to the life cycle of process participants, change is occasional, here it appears continually. The pressure for change is built in. It is intrinsic to the nature of computing systems and the way they are developed and used. *P* and *E* programs are clearly closely related. They differ from *S*-programs in that they represent a computer *application* in the real world. We shall refer to members of the union of the *P* and *E* classes as *A*-type programs.

E. Program Correctness

The first consequence of the SPE program classification is a clarification of the concepts of program correctness and program proving. The meaning, reality, and significance of these concepts have recently been examined at great length [16], [17]. Many of the viewpoints and differences expressed by the participants in that discussion become reconcilable or irrelevant under an adequate program classification scheme.

For the SPE scheme, the concept of verification takes on significantly different meanings for the *S* and the *A* classes. If a completely specified problem is computable, its specification may be taken as the starting point for the creation of an *S*-program. In principle a logically connected sequence of statements can always be found, that demonstrates the validity of the program as a solution of the specified problem. Detailed inspection of and reasoning about the code may itself produce the conviction that the program satisfies the specification completely. A true proof must satisfy the accepted standards of mathematics. Even when the correctness argument is

expressed in mathematical terms, a lengthy or complex chain of reasoning may be difficult to understand, the proof sequence may even contain an error. But this does not invalidate the concept of program correctness proving, merely this instance of its application.

We cannot discuss here the range of *S*-programs for which proving is a practical or a valuable technique, the range of applicability of constructive methods for simultaneous construction of a program and its proof [18], [19]; whether confidence in the validity of an *S*-program can always be increased by a proof. We simply note that since, by definition, the sole criterion of correctness of an *S*-program is the satisfaction of its specification, (correct) *S*-programs are always *provably* correct.

This is not purely a philosophical observation. Many important components of a large program, mathematical procedures for example, in conjunction with specified interface rules (calling and output), are certainly *S*-type. It becomes part of the design process to recognize such potential constituents during the partitioning process and to specify and implement them accordingly. In fact it will be postulated in the next section that an *A*-program may always be partitioned and structured so that *all* its elements are *S*-programs. If this is indeed true, no *individual* programmer should *ever* be permitted to begin programming until his task has been defined and delimited by a complete specification against which his completed program can be validated.

For an *E*-program as an entity on the other hand, validity depends on human assessment of its effectiveness in the intended application. Correctness and proof of correctness of the program as a whole are, in general, irrelevant in that a program may be formally correct but useless, or incorrect in that it does not satisfy some stated specification, yet quite usable, even satisfactory. Formal techniques of representation and proof have a place in the universe of *A*-programs but their role changes. It is the detailed *behavior* of the program *under operational conditions* that is of concern.

Parts of the program that can be completely specified should be demonstrably correct. But the environment cannot be completely described without abstraction and, therefore, approximation. Hence absolute correctness of the program as a whole is not the real issue. It is the *usability* of the program and the *relevance* of its output in a changing world that must be the main concern.

F. Program Structures and Structural Elements

The classification created above relates to program entities. Any such program will, in general, consist of many parts variously referred to as subsystems, components, modules, procedures, routines. The terms are, of course, not used synonymously but carry imputations of functional identity, level, size, and so on.

The literature discusses criteria [20] and techniques [21]–[23] for partitioning systems into such elements. Related design methodologies and techniques seek to achieve optimum assignment, in some sense, of element content and overall system structure. In the present context we consider only one aspect of partitioning using the term module for convenience. The discussion completes the presentation of the SPE classification and provides a link to other current methodological thinking [24].

Consider the end result of the design process for an *A*-program to be constructed of primitive elements we term modules. The analysis and partitioning process will identify some functional elements that can be fully specified and therefore developed as *S*-program modules. Any specification may of course be less than fully satisfactory. It may even prove to be wrong in relation to what the system purpose demands, in itself or in relation to the remainder of the design. For example the specification may not mention input validity checks, the specified output accuracy may be insufficient or the specified range of an input variable may be wrong. But each of these represents an omission from or an error in the *specification*. Thus it is rectified by first correcting the *specification* and then creating, by one means or another, a new program that satisfies the new specification.

The remainder of the system is required to implement functions that are at least partly heuristic or behavioral in nature and therefore define *A*-elements. Nevertheless, we suggest that it is *always* possible to continue the system partitioning process until *all* modules are implementable as *S*-programs. That is, any imprecision or uncertainty emanating from model reflections of incomplete world views will be implicit or, if recognized when the specification is formulated, explicit in the specification statement. The final modules will all be derived from and associated with precise specifications, which *for the moment*, may be treated as complete and correct.

The design may now be viewed and constructed as a data-flow structure with the inputs of one module being the outputs of others (unless emanating from outside the system). Each module will be defined as an abstract data type [25]–[27] defining, in turn, one or more input-to-output transformations. Module specifications include those of the individual interfaces, but for the system as a whole, the latter should, in some sense be standardized [28]. Moreover, given appropriate system and interface architecture and module design, each module could be implemented as a program running on its own microprocessor and the system implemented as a distributed system [9], [24], [28], [92]. The potential advantages for both execution (parallelism) and maintainability (localization of change) cannot be discussed here.

Many problems in connection with the design and construction of such systems need still to be solved. Adequate solutions will represent a major advance in the development of a process methodology (Section III-C). We observe, however, that the concepts presented follow directly from our brief analysis and classification of program types. Interestingly, the conclusions are completely compatible with those of the programming methodologists [24], [29], [30].

III. The Life Cycle

A. The General Case

The dynamic evolutionary nature of computer applications, of the software that implements them and of the process that produces both, has in recent years given rise to a concept of a program *life cycle* and to techniques for *life-cycle management*. The need for such management has, in fact, been recognized in far wider spheres, particularly by national defense agencies and other organizations concerned with the management of complex artificial systems. In pursuing their responsibilities, these must ensure continuing effectiveness of systems whose elements may involve many different and fast developing technologies. Often they must guarantee utterly reliable operation under harsh, hostile, and unforgiving conditions. The outcome is an ever increasing financial commitment. Only lifetime-orientated management techniques applied from project

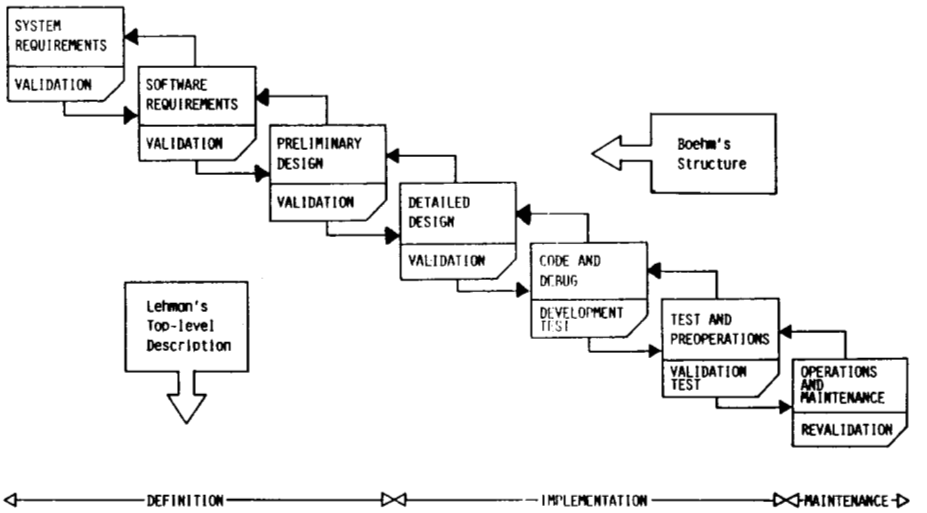

Fig. 5. The software life cycle according to Boehm.

initiation can permit the attainment of lifetime effectiveness and cost effectiveness.

The problems in the more general situation are essentially those we have already explored, except that the time interval between generations is perhaps an order of magnitude greater than in the case of pure software systems. In briefly examining the nature of the life cycle and its management in this section, we use the terminology of programming and software engineering. The reader will be able to generalize and to interpret the remarks in his own area of interest.

B. Software Life Cycles

In studying program evolution, repetitive phenomena that define a life cycle can be observed on different time scales representing various levels of abstraction. The highest level concerns *successive generations* of system sequences. Each generation is represented by a sequence of system releases. This level corresponds most closely to that found in the more general systems situation, with each generation having a life span of from, say, five to twenty years. Because of the relatively slow rate of change it is difficult for any individual to observe this evolution phenomenon, measure its dynamics and model it as a life-cycle process since in the relevant portion of his professional career he will not observe more than two or three generations. It might therefore be argued that this level should not be treated as an instance of the life-cycle phenomenon. The present author has, however, had at least one opportunity to examine program evolution at this level and to make meaningful and significant observations [31]. These indicated that much could be gained in cost effectiveness in the software industry if more attention were paid to the earlier creation of replacement generations, something that can be achieved effectively only if the appropriate predictive models are available.

The second level is concerned with a *sequence* of *releases*. The latter term is also appropriate when a concept of continuous release is followed, that is when each change is made, validated, and immediately installed in user instances of the system.

Fig. 5 shows one view [6] of the *sequence* of *activities* or life-cycle phases that constitute the lowest level, the development of an individual release, if it is assumed that "maintenance" in the seventh box refers to on-site *fixes* and repairs implemented as the system is used. If maintenance is taken to refer to permanent changes, effected through new releases by the system originator, then the structure becomes *recursive* with each maintenance phase comprised of all seven indicated phases. With this interpretation the single recursive model reflects the composite life-cycle structure of all the above levels.

The remainder of this paper is chiefly concerned with the intermediate level, the life cycle of a generation as represented by a sequence of releases. It is at this level that analysis in terms of the S and A classification is particularly relevant and enlightening.

C. Assembly Line Processes

An assembly line manufacturing process is possible when a system can be partitioned into subsystems that are simply coupled and without invisible links. Moreover, the process must be divisible into separate phases without significant feedback control over phases and with relatively little opportunity for tradeoff between them.

Unfortunately, present day programming is not like that. It is constituted of tightly coupled activities that interact in many ways. For example, at least some aspects of the specification and design processes are left over, usually implicitly, to the implementation (coding) phase. Fault detection through inspection [90] is not yet universal practice and by default is often delayed till a system integration or system testing phase. One of the main concerns of life-cycle process methodology research must be to develop techniques, tools, new system architectures (Section II-F) and programming support environments [32]–[34] that permit partitioning of the program development and maintenance process into separated activities.

D. The Significance of the Life-Cycle Concept

For assembly line processes the life-cycle concept is not, generally, of prime importance. For software and other highly complex systems it becomes critical if effectiveness, cost effectiveness, and long life are to be achieved. At each moment in time, a manager's concern concentrates on the successful completion of his current assignment. His success will be assessed by immediately observable product attributes, quality, cost, timeliness, and so on. It is his success in areas such as these that determine the furtherance of his career. Managerial strategy will inevitably be dominated by a desire to achieve maxi-

mum local payoff with visible short-term benefit. It will not often take into account long-term penalties, that cannot be precisely predicted and whose cost cannot be assessed. Top-level managerial pressure to apply life-cycle evaluation is therefore essential if a development and maintenance process is to be attained that continuously achieves, say, desired overall balance between the short- and long-term objectives of the organization. Neglect will inevitably result in a lifetime expenditure on the system that exceeds many times the assessed development cost on the basis of which the system or project was initially authorized.

To overcome long time lags and the high cost of software, one may also seek to extend the useful lifetime of a system. The decision to replace a system is taken when maintenance has become too expensive, reliability too low, change responsiveness too sluggish, performance unacceptable, functionality too limiting; in short, when it is economically more satisfactory to replace the system than to maintain it. But its expected life time to that point is determined primarily in its conception, design and initial implementation stages. Hence management planning and control during the formative period of system life, based on lifetime projections and assessment, can be critical in achieving long life software and lifetime cost effectiveness [1].

E. Life-Cycle Phases

1) The Major Activity Classes: At its grossest level a life cycle consists of three phases: definition, implementation and maintenance. As indicated in Fig. 5, these three phases correspond approximately to the activities described in the first three, the second three and the seventh box respectively of Boehm's model. In practice, however, many of these activities are overlapped, interwoven, and repeated iteratively.

2) System Definition: For *E*-class systems in particular, the development process begins with a pragmatic analysis leading into a systematic *systems analysis* to determine total system and program *requirements* [35]–[38]. The analysis must first establish the *real* need and objectives and may examine the manual techniques whereby the same purpose is currently achieved. Where appropriate, it may be based on mathematical or other formal analysis. Whatever the approach, it has now been recognized that the analysis must be *disciplined* and *structured* [29], [30], the term *structured analysis* now being widely used [9], [41], [42].

By their very nature initial requirements, being an expression of the user's view of his needs, are likely to include incompatibilities or even contradictions. Thus the analysis and the negotiation process by and between analysts and potential users that produces the final *requirements specification*, must identify a balanced set that, in some sense, provides the optimum compromise between conflicting desires.

The requirements set will be expressed in the concepts and language of the application and its users. It must then be transformed into a *technical specification*. The specification process [43], [44] must aim to produce a correct technical statement, *complete* in its coverage of the requirements and *consistent* in its definition of the implementation. It may include additional determinations or constraints that follow from a technical evaluation of the requirements in relation to what is feasible, available and appropriate in the judgment of the analyst and designer in agreement with the user.

It has long been the aim of computer scientists to provide formal languages for the expression of specifications so as to permit mechanical checking of completeness and consistency [45]–[49], [91], but a widely accepted language does not yet exist. Given a machinable specification it is conceptually possible to reduce it mechanically to executable [50] and even efficient [14] code but these technologies too are not yet ready for general exploitation.

Thus, for the time being, the specification process will be followed by a *design* phase [49], [51]. The prime objective of this activity is to identify and structure data, data transformation and data flow [23]. It must also achieve, in some defined sense, optimal partitioning of system function [20], select computational algorithms and procedures, and identify system components, and the relationships between them. It is now generally accepted that iterative *top-down* [52] analysis and partitioning processes are required to achieve *successive refinement* [21] of the system design to the point where the identified objects, procedures, and transformations can be directly implemented.

3) Implementation: Following the completion of the design, system *implementation* may begin. In practice, however, design and implementation overlap. Thus, as the hierarchical partitioning process proceeds, analysis of certain aspects of the system may be considered sufficient for implementation, while others require further analysis. In a software project, time always appears to be at a premium. A work force comprising many different abilities is available and must be kept busy. Thus, regrettably, implementation of subsystems, components, procedures, or modules will be initiated despite the fact that the overall, or even the local design, is not yet complete.

As the implementation proceeds code must be *validated* [53], [54]. Present day procedures concentrate primarily on *testing* [55], though in recent years increasing use has been made of design *walkthrough* and code *inspection* [90]. These latter procedures are intended to disclose both design and implementation errors before their consequences become hidden in the program code. The ratio of costs of removing a fault discovered in usage as against the cost of removing the same fault if discovered during the design or first implementation phase is sometimes two or three orders of magnitude. Clearly, it pays to find faults early in the process.

In any case, testing by means of program execution is carried out, generally bottom up, first at the unit (module or procedural) level, then functionally, component by component. As tested components become available they are then assembled into a system in an *integration* process and *system test* is initiated. Finally, after some degree of independent certification of system function and performance, the system is designated ready for *release*.

The above very brief summary has identified some of the activities that are typically undertaken in a system creation process. Individual activities as described may overlap, be iterated, merged, or not undertaken at all. Design of an element, for example, may be followed immediately by a test implementation and preliminary performance evaluation to ensure feasibility of a design before its implications spread to other parts of the system. Clearly, there should be a set of overall controlled procedures to take a concept from the first pragmatic evaluation of the potential of an application for mechanization to the final program product executing in defined hardware or software and hardware environment(s).

4) Maintenance: Once the system has been released, the maintenance process begins. Faults will be observed, reported, and corrected. If user progress is blocked because of a fault, a temporary bypass of the faulty code may be authorized. In other circumstances a temporary or permanent fix may be

applied in some or all user locations. The permanent repair or change to the program can then be held over for a new release of the system. In other cases, a permanent change will be prepared for immediate installation by all those running the system. The particular strategy adopted in any instance will depend on the nature and severity of the fault, the size and difficulty of the change required, the number and nature of program installations and user organizations, and so on. The aggregate strategy will have a profound impact on the rate of system complexity growth, on its life-cycle costs, and on its life expectancy.

The faults that are fixed in the maintenance process may be due to changes external to the system, incorrect or incomplete specification, design or implementation errors, hardware changes or to some combination of these. Since each user exposes the system in different ways, all installations do not experience all faults, nor do they automatically apply all manufacturer-supplied fixes or changes. On the other hand, installations having their own programming staff may very well develop and install localized changes or system modifications to suit their specific needs. These patches, insertions, or deletions may in turn cause new difficulties when further incremental changes are received from the manufacturer, or at a later date when a new release is received. The inevitable consequences of the maintenance process applied to systems installed for more than one user, is that the system drifts apart. Multiple versions of system elements develop to encompass the variations and combinations [56]. System *configuration* management becomes a major task. *Support environments* [33]–[35] that automatically collect and maintain total activity records become an essential tool in programming process management.

F. Life-Cycle Planning and Management

The preceding discussion, while presenting a simplified view of the life cycle, will have made clear the difficulty associated with cycle planning. In recent years this problem has received much attention [57], [58]. A variety of techniques have been developed to improve estimation of cost, time, and other resources required for software development and maintenance [59]–[64]. These techniques are based on extrapolation of past experience and tend to produce results in the nature of self-fulfilling prophecies. In general, it has not yet proved possible to develop techniques that estimate project requirements on the basis of objective measurement of such attributes as application complexity and size and the work required to create a satisfactory system. Techniques such as software science [65], [66] seek to do just this but to date lack substantiation [67] and interpretation. Major research and advances are required if software engineering is to become as manageable as are other engineering disciplines, though fundamentally the peculiar nature of software systems [28] will always leave software engineering in a class of its own.

IV. Laws of Program Evolution

A. Evolution

The analysis of Section II associated with the life-cycle description of Section III, has indicated that evolution is an intrinsic, feedback driven, property of software. The metasystem within which a program evolves contains many more feedback relationships than those identified above. Primitive instincts of survival and growth result in the evolution of stabilizing mechanisms in response to needs, events and changing objectives. The resulting pseudohierarchical structure of self-stabilizing systems includes the products, the processes, the environments and the organizations involved. The interactions between and within the various constituents, and the overall pattern of behavior must be understood if a program product and its usage are to be effectively planned and maintained.

The organizational and environmental feedback, links, focuses, and transmits the evolutionary pressure to yield the continuing change process. A similar situation holds, of course, for any human organized activity, any artificial system. But some significant differences are operative in the case of software. In the first instance there is no room in programming for imprecision, no malleability to accommodate uncertainty or error. Programming is a mathematical discipline. In relation to a *specific* objective, a program is either right or wrong. Once an instruction sequence has been fixed and unless and until it is manually changed, its behavior in execution on a given machine is determined solely by its inputs.

Secondly, a software system is soft. Changes can be implemented using a pencil, paper, and/or a keyboard. Moreover, once a change has been designed and implemented on a development system it can be applied mechanically to any number of instances of the same system without further significant physical or intellectual effort using only computing resources. Thus the temptation is to implement changes in the existing system, change upon change upon change, rather than to collect changes into groups and implement them in a totally new instance. As the number of superimposed changes increases, the system and the metasystem become more complex, stiffer, more resistant to change. The cost, the time required, and the probability of an erroneous or unsatisfactory change all increase.

Thirdly, the rate at which a program executes, the frequency of usage, usage interaction with the operating environment, economic and social dependence of external process on program execution, all cause deficiencies to be exposed. The resultant pressure for correction and improvement leads to a system rate of change with a time scale measured in days and months rather than in the years and decades that separate hardware generations.

B. Dynamics and Laws of Program Evolution

The resultant evolution of software appears to be driven and controlled by human decision, managerial edict, and programmer judgment. Yet as shown by extended studies [68]–[76], measures of its evolution display patterns, regularity and trends that suggest an underlying dynamics that may be modeled and used for planning, for process control, and for process improvement.

Once observed the reasons for this unexpected regularity is easily understood. Individual decisions in the life cycle of a software system generally *appear* localized in the system and in time. The considerations on which they are based *appear* independent. Managerial decisions are largely taken in relative isolation, concerned to achieve local control and optimization, concentrated on some aspect of the process, some phase of system evolution. But their aggregation, moderated by the many feedback relationships, produces overall systems response which is regular and often normally distributed.

In its early stages of development a system is more or less under the control of those involved in its analysis, design, and implementation. As it ages, those working on or with the system become increasingly constrained by earlier decisions, by existing code, by established practices and habits of users and

TABLE I
LAWS OF PROGRAM EVOLUTION

I. *Continuing Change*

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

II. *Increasing Complexity*

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

III. *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.

IV. *Conservation of Organizational Stability (Invariant Work Rate)*

During the active life of a program the global activity rate in a programming project is statistically invariant.

V. *Conservation of Familiarity (Perceived Complexity)*

During the active life of a program the release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

implementors alike. Local control remains with people. But process and system-internal links, dependencies, and interactions cause the global characteristics of system evolution to be determined by organization, process and system parameters. At the global level the metasystem dynamics have largely taken over.

Since the original observation [63], studies of program evolution have continued, based on measurements obtained from a variety of systems. Typical examples of the resultant models have been reported [69]-[72], [74], [76] including also one detailed example of their application to release planning [77].

It was repeated observation of phenomenologically similar behavior and the common interpretation of independent phenomena, that led to a set of five laws, that have themselves evolved as insight and understanding have increased. The laws, as currently formulated to include the new viewpoint emerging from the SPE classification, are given in Table I. Their early development can be followed in [9], [10], [72]. We note that the laws are abstractions of observed behavior based on statistical models. They have no meaning until a system, a project and the organizational metasystem are well established. More detailed discussion of their nature and of their technical and managerial implications will be found in [11], [77], [78] and [77], [79], [80], respectively.

The first law, *continuing change*, originally [3], [10], [79] expressed the universally observed fact that large programs are never completed. They just continue to evolve. Following our new insight, however, reference to *largeness* is now replaced by the phrase . . . "that reflect some other reality . . ."

The second law, *increasing complexity*, could be seen as an instance of the second law of thermodynamics. It would seem more reasonable to regard both as instances of some more fundamental natural truth. But from either viewpoint its message is clear.

The third law, the *fundamental law of program evolution*, is in the nature of an existence rule. It abstracts the observed fact that the number of decisions driving the process of evolution, the many feedback paths, the checks and balances of organizations, human interactions in the process, reactions to usage, the rigidity of program code, all combine to yield statistically regular behavior such as that observed and measured in the systems studied.

The fourth law, *conservation of organizational stability*, and the fifth, *conservation of familiarity*, represent instances of the observations whose generalization led to the third law. The fourth reflects the steadiness of multiloop self-stabilizing systems. It is believed to arise from organizational striving for stability. The managements of well-established organizations avoid dramatic change and particularly discontinuities in growth rates. Moreover, the number of people and the investments involved, the unions, the time delays in implementing decisions, all operate together to prevent sudden or drastic change. Wide fluctuations may in fact lead to instability and the breakup of an organization.

The reader may find it difficult to accept the implication that the work output of a project is independent of the amount of resources employed, though the same observation has also been recorded by others [81]. The underlying truth is that activities of the type considered, though initiated with minimal resources, rapidly attract more and more as commitment to the project, and therefore the consequences of success or failure, increase. Our observations as formalized in the fourth law imply that the resources that can be productively applied becomes limited as a software project ages. The magnitude of the limit depends on many factors including attributes of the total environment. But the pressure for success leads to investment to the point where it is exceeded. The project reaches the stage of resource saturation and further changes have no visible effect on real overall output.

While the fourth law springs from a pattern of organizational behavior, the fifth reflects the collective consequences of the characteristics of the many individuals within the organization. It is discussed at length in [11]. Suffice it to say here that the law arises from the nonlinear relationship between the magnitude of a system change and the intellectual effort and time required to absorb that change.

TABLE II
SYSTEM X STATISTICS

Release 19 Statistics	
Size	4800 Modules
	1.3 Assembly *M*-statements
Incremental growth	410 Modules
Modules changed[a]	2650 Modules
Fraction of modules changed	0.55
Release interval	275 Days
System Statistics	
Age	4.3 Years
Change rate	10.7 Modules/day
Average incremental growth	200 Modules/release
Maximum safe growth rate	400 Modules/release

Most Recent Releases

Release	15	16	17	18	19
Incremental growth (Δ Mod)	135	171	183	354	410
Fraction changed	0.33	0.43	0.48	0.50	0.56
Change rate	12.5	0.12	9.6	9.9	9.6
Interval (Days)	96	137	201	221	275
Old mods, Changed/ Mod	7.9	8.6	10.0	5.1	5.4

[a]Modules that are changed in any way in release $i+1$ relative to release i are counted as one changed module, independently of the number of changes or of their magnitude.

V. APPLIED DYNAMICS

A. Introduction

The previous sections have emphasized the phenomenological basis for the laws of program evolution, indicating how they are rooted in phenomena underlying the activity of programming itself.

The origin of the laws in individual and societal behavior makes their impact on the construction and maintenance of software more than just descriptions of the evolutionary process. The laws represent *principles* in software engineering. They are, however, clearly not immutable, as for example, are the laws of physics or chemistry. Since they arise from the habits and practices of people and organizations, their modification or change requires one to go outside the discipline of computer science into the realms of sociology, economics and management. The laws therefore form an environment within which the effectiveness of programming methodologies and management strategies and techniques can be evaluated, a backdrop against which better methodologies and techniques can be developed.

Their implications, technical and managerial, have been previously discussed in the literature [3], [9], [11], [79], [80]. In the present paper, we restrict the discussion to outlining an example of the application of evolution dynamics models to release planning.

B. A Case Study–System X

1) The System and its Characteristics: System X is a general purpose batch operating system running on a range of machines. The eighteenth release (R18) of the system is operational in some tens of installations running a variety of work loads. The nineteenth release (R19) is about to be shipped.

Table II and Fig. 6(a)–(g) present the system and release data and models available for the purposes of the present exercise. We cannot, however, provide here the details of statistical analysis and model validation [76], based on this data and that from other systems that gives us confidence in our conclusions and predictions.

Examining the system dynamics as implied by models derived from the data and as illustrated by the figures, Fig. 6(a) shows the continuing growth of the system (first law) albeit at a declining rate (demonstrably due to increasing difficulty of change, growing complexity–second law).

Fig. 6(b) indicates that as a function of release sequence number (RSN) the system growth (measured in modules) has been linear but with a superimposed ripple (a strong indicator of feedback stabilization).

Fig. 6(c) shows the net incremental growth per release (fifth law).

For system architectures such as that of system X, the fraction of system modules that are changed during a release may be taken as a gross indicator of system complexity. Fig. 6(d) shows that system X complexity, as measured in this way, shows an increasing trend (second law).

Fig. 6(e) is an example of the repeatedly observed constant average work rate (fourth law).

Fig. 6(f) illustrates how the average work rate achieved in individual releases, as measured by the rate of module change (changed modules per release interval day (m/d) oscillates, a period of high rate activity being followed by one or more in which the activity rate is much lower (third law).

Finally, Fig. 6(g) plots the release interval against release sequence number. It has been argued that release interval depends purely on management decision that is itself based on market considerations and technical aspects of the release content and environment. Data such as that of Fig. 6(g) indicates, however, that the feedback mechanisms that, amongst other process attributes, also control the release interval, while including human decision taking processes, are apparently not dominated by them. As a consequence, the release interval pattern is sufficiently regular to be modelable, and is statistically predictable once enough data points have been established.

2) The Problem: Already prior to the completion (and re-

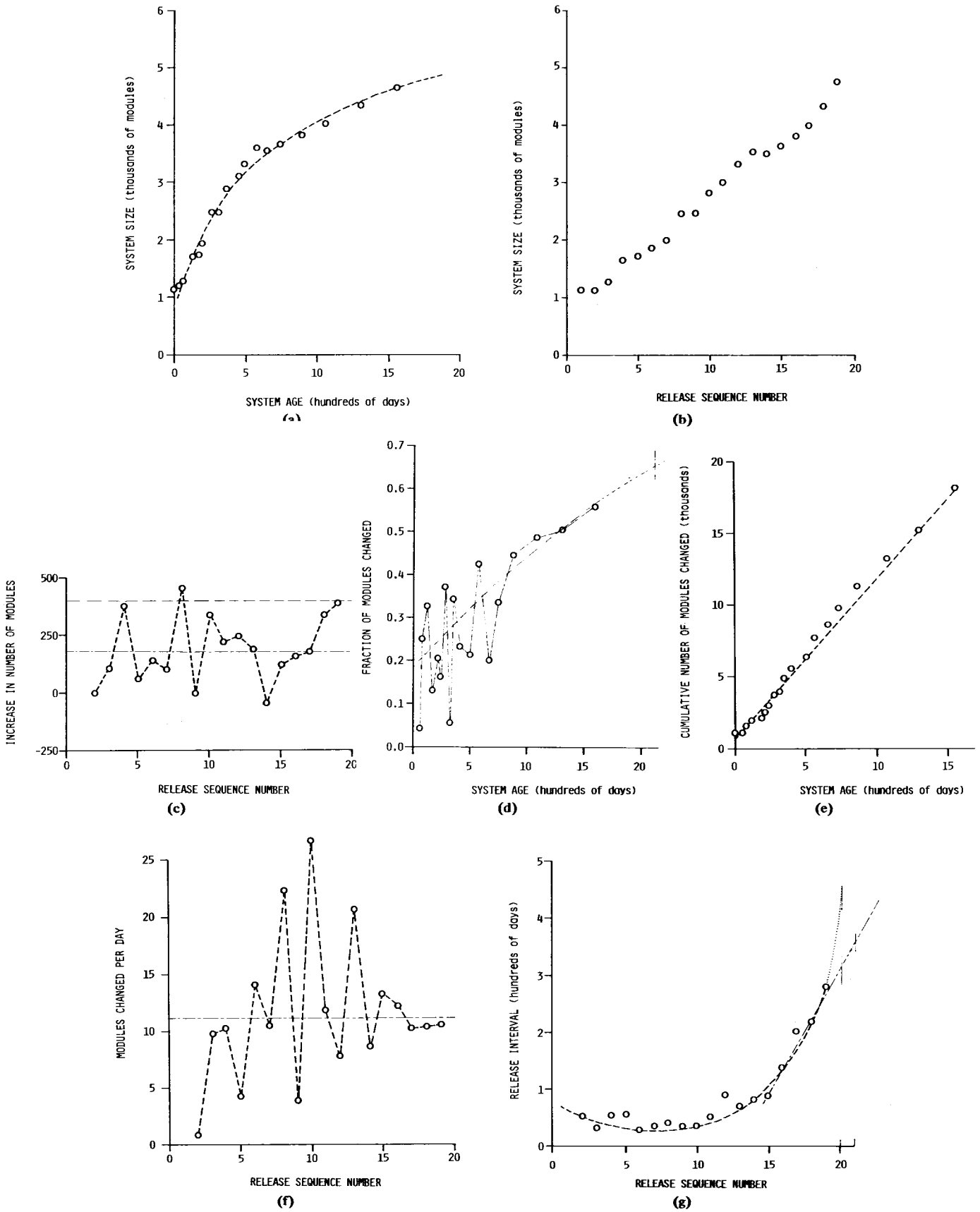

Fig. 6. System characteristics.

TABLE III
RELEASE 20 PLANNED CONTENT

Functional Enhancement

No.	Description	New Mods (NM)	Old Mods Chgd (OMC)	Mods Chgd (NM + OMC)	OMC/NM (IR)
1.	Identified faults (PRE. RLS. 19)	2	380	382	–
2.	Expected faults (RLS. 19)	0	600	600	–
3.	Interactive terminal support (ITS)	750	1783	2533	2.4
4.	Dynamic storage management (DSM)	170	1500	1670	8.8
5.	Remote job entry (RJE)	57	462	519	8.1
6.	New disk support (NDS)	17	124	141	7.3
7.	Batch scheduler improvements (BSI)	3	29	32	9.7
8.	File access system (FAS)	8	74	82	9.3
9.	Paper tape support (PTS)	12	80	92	6.7
10.	Performance improvements	2	157	159	–
		1021	5189	6210	

Detail of Interactive Terminal Support (ITS)

No.	Description	New Mods	Old Mods Chgd	OMC/NM
3A	Terminal support	444	1032	2.3
3B	Scheduling	127	293	2.3
3C	Telecom support	58	232	4.0
3D	Misc.	121	226	1.9
		750	1783	

lease) of R19, work has begun on a further version R20, whose main component is to be the addition of interactive access to complement current batch facilities. This new facility "ITS" together with other changes and additions summarized in Table III, are to be made available eighteen months after first customer installation of R19.

For each major planned functional change, the table lists the number of new modules to be added (NM), the number of R19 modules that are to be changed in the course of creating R20 (OMC), the total number of modules changed (NM + OMC), and the ratio of OMC to NM (the interconnectivity ratio (IR), an indicator of complexity). No modules are planned for removal in the creation of R20 hence the planned net system growth is 1021 modules.

Management has also accepted that a further release R21, will follow twelve months after R20, to include any leftovers from R20. It may also include additional changes for which a demand develops over the next two years. The current exercise is to endorse the overall plan, or if it can be shown to be defective, to prepare an alternative recommendation.

3) Process Dynamics:

a) Work rate: From Fig. 6(e) the work rate has averaged 10.4 m/d[1] over the lifetime of the system. Fig. 6(f) indicates that the maximum rate achieved so far has been 27 m/d. Evidence that cannot be detailed here reveals, however, that that data point is misleading and that a peak rate of about 20 m/d is a better indicator of the maximum achievable with current methodology and tools. Moreover, there is strong circumstantial evidence that releases achieved with such high work rates were extremely troublesome and had to be followed by considerable clean-up in a follow-up release, as also implied by Fig. 6(c). Thus, if R20 is planned so as to require a work rate in the region of 20 m/d, it would be wise to limit R21 to at most 10 m/d, the system average. If on the other hand, the process is further stabilized by working on R20 at near average rate, one could then, with a high degree of confidence, approach R21 with a higher work rate plan.

b) Incremental growth: The maintained average incremental growth for system X has been around 200 modules/release. Once again circumstantial evidence indicates that releases (for which, in this case, the growth rate (incremental growth per release) has exceeded twice the average) have slipped delivery dates, a poor quality record and a subsequent need for drastic corrective activity. Fig. 6(c) and Table II indicate that R19 will lie in this region and that R18 had high incremental growth. That is, R19, once released, is likely to prove a poor quality base. The first evidence emerges that maybe R20 should be a clean-up release.

c) Growth rate in modules for release: The same indication follows from Figs. 6(a) and (b) where the ripple periods are seen to be three, four, and five intervals, respectively over the first three cycles. In the fourth cycle, six intervals of increasing growth rate have passed with the R18–R19 growth the largest ever. Without even considering the planned growth to R20 (Point X), it seems apparent that a clean-up release is due.

4) R20 Plan Analysis:

a) Initial analysis: The first observation on the plan as summarized by Table III stems from the column (6) of IR

[1] Modules per day = $\frac{\text{number of modules changed in release}}{\text{release interval in days}}$

factors. It has not been calculated for items 1, 2, and 10 since these represent activities that only rarely require the provision of entirely new (nonreplacement) modules. For items 4–9 the ratio lies in the range 8.2 ± 1.5, a remarkably small range for widely varying functional changes. Yet the predicted ratio for ITS is only 2.4. One must ask whether it is reasonable to suppose that the code implementing an interactive facility is far more loosely coupled to the remaining system than, for example, a specialist facility such as paper tape support? Is it not far more likely that ITS has been inadequately designed; viewed perhaps as an independent facility that requires only loose coupling into the existing system? Thus, when it is integrated with the remainder of the system to form R20, will it not require many more changes to obtain correct and adequate performance? From the evidence before us, the question is undecidable. Experience based intuition, however, suggests that it is rather likely that the number of changes required elsewhere in the system has been underestimated. Thus a high-priority design reappraisal is appropriate. If the suspicion of incomplete planning proves to be correct, it would suggest delaying R20, so that the planning and design processes may be completed. An alternative strategy of delaying at least ITS to R21 should also be evaluated.

b) Number of modules to be changed: The situation may of course not be quite as bad as direct comparison of the present estimate of the ITS interconnection ratio (IR) with that of the other items, suggests. In view of the 750 new modules involved, its IR factor could not exceed 6.4 even if all 4800 modules of R19 were effected by the ITS addition. Such a 100 percent change is, in fact, very unlikely, but the IR factor of 2.4 remains very suspect.

Moreover, even with the low ratio for ITS the sum of the individual OMC estimates for the entire plan exceeds the number of modules in R19. This suggests a new situation. Multiple changes applied to the same module must have become a significant occurrence. Even ignoring the fact that even independent changes applied in the same release to the *same* module generally demand significantly more effort than similar changes applied to independent modules, the total effort and time required must clearly increase with both the number of changes implemented and the number of modules changed. The presently defined measure "modules *changed*" is inadequate. The new situation demands consideration of more sensitive measures such as "number of module *changes*" and "average number of changes per module."

These cannot be derived from the available data. We may, however, proceed by considering a model based on the data of Fig. 6(d). Extrapolating the fraction changed trend, reveals that R20 may be expected to require a change of, say, 64 percent, or 3725 changed modules.[2] Comparing this estimate with the total of 6210 obtained if the estimates for individual items are summed, it appears that the average number of changes to be applied to R19 modules according to the present plan is at least of order two. We have already observed that multiple changes cause additional complications. Hence any prognosis made under the implied assumption of single changes (or of a somewhat lower interconnection ratio) will lead to an optimistic assessment.

[2] *Historical Note:* In the system on which this example is based the release including the interactive facility ultimately involved some 58 percent of modules changed. Moreover the first release was significantly delayed, and was of limited quality and performance. More than 70 percent of its modules had subsequently to be changed again to attain an acceptable product. Our estimate is clearly good.

c) Rate of work: The current plan calls for R20 with its 3725 module changes to be available in 18 months, that is 548 days. This implies a change rate of less than 6.8 m/d. This relatively low rate, following a period of average rate activity suggests that *work rate* pressures are unlikely to prove a source of trouble, even with multiple changes to many of the modules.

d) Growth rate: In Figs. 6(a) and (b), we have indicated the position of R20 as per plan, with an X. Both modules indicate that the planned growth represents a major deviation from the previous history. Thus confirmation that the plan is realistic requires a demonstration that the special nature of the release, or changes in methodology, makes it reasonable to expect a significant change in the system dynamics. In the absence of such a demonstration, the suspicion that all is not well is strengthened.

e) Incremental growth: The current R20 plan calls for system growth of over 1000 modules. This figure which is five times the average and two and a half times the recommended maximum, must be interpreted as a danger signal.

We have already suggested that the low interconnection ratio for ITS suggests that the planners saw the new component as a stand alone mechanism that interfaces with the remainder of the system via a narrow and restricted interface. If this view proves justified, the large incremental growth need not be disturbing. But it seems reasonable to question it. With the architecture and structure that system X is known to have, such a relatively narrow interface is unlikely to be able to provide the communication and control bandwidth that safe, effective, and high capacity operation must demand. This is apparent from comparisons with, say, the paper tape or disk support changes or the RJE addition. The onus must be put onto the ITS designers to demonstrate the completeness of their analysis, design, and implementation.

Without such a demonstration one must conclude that the present plan is not technically viable. *Marketing* or other considerations may, of course, make it desirable to stay with the present plan even if this implies slipped delivery dates, poor and unreliable performance of the new release, limited facilities, and so on. But if such considerations force adoption of the plan, the implications must be noted, and corrective action planned. Ways and means will have to be created to enable users to cope with the resultant system and usage problems and the inevitable need for a major clean-up release. It might, for example, be wise to set up specialized customer support teams to assist in the installation, local adaptation and tuning of the system.

f) Release interval: Fig. 6(g) indicates two possible models for the prediction of the most likely (desirable?) release interval for R20 and R21. Linear extrapolation suggests a release period of under one year for each of the two releases. If this is valid, the apparent desire for a release after the 18 months is of itself unlikely to prove a source of problems. On the basis of evidence not reproduced here, however, the exponential extrapolation is likely to be more realistic and this yields an R20 release interval forecast of about 15 months and an R21 interval of some 3 years.

g) Recommendation–Summary: On the basis of the available data we have concluded that

1) to proceed with the plan as it stands is courting delivery and quality problems for R20;
2) a clean-up release appears due in any case;
3) failure to provide it will leave a weak base for the next

TABLE IV
MODIFIED RELEASE 20 CONTENT

Class	Reason	Items
Fault Repair	Clean-up of base	1, 2.
Hardware Support	Revenue Producing	6, 9.
Performance Improvement	Install–but do not announce. Will be available to counteract ITS performance deterioration in R21′	7, 10.
ITS Related Components	To receive early user exposure	3c, 5, 8.

TABLE V
MODIFIED RELEASE 20 STATISTICS (FROM TABLE III) IN ORDER OF PRIORITY

Item	New Mods.	Running Total	Changes	Running Total
1	2	2	382	382
2	0	2	600	982
6	17	19	141	1123
9	12	31	92	1215
7	3	34	32	1247
10	2	36	159	1406
8	8	44	82	1488
5	57	101	519	2007
3c	58	159	522	2529

release; at the very least the number of expected faults (Table III, item 2) is likely to prove an underestimate;

4) the absolute size of the ITS component and the related incremental system growth would represent a major challenge even on a clean base;
5) there are indications that the ITS aspect of the release design is incomplete;
6) change rate needs for R20 are not likely to prove a source of problems;
7) nor is the demand for attainment of a next release in eighteen months.

The following recommendations follow:

8) initiate immediately an intensive and detailed reexamination of the ITS design and its interaction with the remainder of System X;
9) from the integration records of R19 and by comparison with the records of earlier releases, make quality and error rate models and obtain a prognosis for R19 and an improved estimate for R20 correction activity; integration and error rate models have not been considered in the present paper but have been extensively studied by the present author and by others [85];
10) assess the business consequences of, on the one hand, a slippage of one or two years in the release of ITS and on the other, a poor quality, poor performance release with a slippage of, say, some months (due to acceptable work rate but excessive growth);
11) in the absence of positive indication of a potential for major deviations from previous dynamic characteristics or the existence of a genuine business need that is more pressing than the losses that could arise from a poor quality product, abandon the present plan;
12) instead redesign release 20 to yield R20′; a clean, well-structured, base on which to build an ITS release, R21′;
13) tentatively release intervals of 9 months and 15 months are proposed for R20′ and R21′, respectively;
14) R21′ should be a restricted release for installation in selected sites;
15) it would be followed after 1 year by a general release R22′.

h) Recommendations–Details: Assuming that the further investigation as per paragraphs 8 to 10 of Section V-B4g reinforces the conclusions reached, three releases would have to be defined. We outline here proposals for R20′ and R21′. The third, R22′ will be a clean-up but its content cannot be identified in detail until a feel for the performance and general quality of R21′ has developed. The detailed analysis is left as an exercise to the reader.

The inherent problem in the design of the ITS release is the fact that the component has a size almost twice the maximum recommended incremental growth. Moreover, with the possible exception of its telecommunications support (Table III, item 3c), none of the component subsystems would receive usage exposure in the absence of the others. Thus a clean ITS release cannot be achieved except by releasing the component in one fell swoop. Similarly, dynamic storage management (DSM) is exposed to user testing only when the ITS facility is operational. We may, however, investigate whether the telecommunication facility (3c) will be usable in conjunction with the RJE facility, item 5. If it is, there will be some advantage to be gained by releasing 3c and 5 before the remainder of ITS and DSM.

Strictly speaking, Fig. 6(c) suggests that R20′ should be a very low content release dedicated to system clean-up and restructuring. But the six preceding releases were achieved with average change rates and, from that point of view, did not stress the process. Thus, if R20′ is also an average rate release, it should not cause problems, and it would seem a low risk strategy to include R20′ in all those items as in Table IV, that will simplify the subsequent creation and integration of the excessively large ITS release.

The list, in priority order, of the new proposal shows a maximum incremental growth (159) well under average. It is a matter of some judgment and experience whether it would be wiser to delay item 3c with 58 new modules and item 5 with 57 to R21′ thereby achieving the very low content release mentioned above. With the information before the reader it is not possible to resolve this question since additional information, at the very least answers to the questions raised in Section V-B4g, would be required. However, the desire to minimize R21′ problems suggests the adoption of the complete plan as in Tables IV and V.

In assessing achievable release intervals for these releases, we base our estimates only on the module change count and change rate. The constraints on the present example do not permit the full analysis which would consider models based on Fig. 6(g), and take into account additional data. At 10 m/d change rate, implementation of the complete plan appears to require 253 days, say 9 months, whereas exclusion of 3c and 5 would reduce the predicted time required to some seven months. This recommendation cannot be taken further without more information of both a technical and a marketing nature, and an examination of other interval models. But the need for a clean base for R21′ suggests adoption of the

maximum acceptable release interval. R21′ will now include, at the very least, ITS (except 3c) and DSM. This involves at least 920 new modules, an excessive growth that cannot usefully be further split between two or more releases. Assuming a change fraction of, say, 70 percent (Fig. 6(d)), of a system that is expected to contain 5911 modules, we estimate a total of 4200 changed modules in the release, many with multiple changes. Since there will now have been seven near average change-rate releases, it seems possible to plan for a change rate of 15–20 m/d, yielding a potential release interval of under 9 months. That is, it would appear that, by adopting the new strategy, all of the original changes and additions could be achieved in about the same time, but much more reliably. More complete analysis, however, based on additional data, other models and taking into account the special nature of the releases might well lead to a recommendation to increase the combined release interval to, say, two years.

A further qualification must also be added. As proposed in the revised plan, R21′ will still be a release with excessive incremental growth and is therefore likely to yield significant problems. The additional fact that the evidence indicates incomplete planning, reinforces concern and expectation of trouble ahead. It is therefore also recommended that R21′ be announced as an experimental release for exposure to usage by selected users in a variety of environments. It would be followed after an interval of perhaps one year by an R22′, a cleaned up system, suitable for further evolution.

i) Final comments: The preceding section has presented a critique of a plan, and outlined an alternative which is believed technically more sound. The case considered is based on a real situation, though in the absence of complete information details have had to be invented. But the details are not important since the objective has been to demonstrate a methodology. Software planning can and should be based on process and system measures and models, obtained and maintained as a continuing process activity. Plans must be related to dynamic characteristics of the process and system, and to the statistics of change. By rooting the planning process in *facts*, *figures* and *models*, alternatives can be quantitatively compared, decisions can be related to reality and risks can be evaluated. Software planning must no longer be based solely on apparent business needs and market considerations; on management's local perspective and intuition.

VI. Conclusion

This paper rationalizes the widely held view, first expressed in Garmisch [82], that there is an urgent need for a discipline of software engineering. This should facilitate the cost-effective planning, design, construction, and maintenance of effective programs that provide, and then continue to provide, valid solutions to stated (possibly changing) problems, or satisfactory implementations of (possibly changing) computer applications.

Following a brief discussion of the nature of computer usage and of the programs, the paper introduced the new SPE classification that addresses the essential evolutionary nature of various types of programs and establishes the existence of a determining specification as the criterion for nonevolution.

In the subsequent discussion of the concepts, significance and phases of the program life cycle, no details of life-cycle planning and management models, as such, have been included. In particular, we have not here discussed cost, resource, and reliability models [83]–[85]. Approaches to process modeling based on continuous models [73], [75] have also not been included, nor has the vital topic of software complexity [86]–[89].

Recognizing the intrinsic nature of program change, the laws that appear to govern the dynamics of the evolution process were introduced. Among their other implications, the laws indicate that project plans must be related to dynamic characteristics of the process and system, and to the statistics of change. By rooting the planning process in facts, figures, and models, alternatives can be quantitatively compared, decisions can be related to reality and risks can be evaluated. Software planning must no longer be based solely on apparent business needs and market considerations; on management's local perspective and intuition. To illustrate this, we have included a brief example of the application of evolution dynamics models to release planning.

Many of the concepts and techniques presented in this paper could find wide applications outside the specific area of software systems, in other industries, and to social and economic systems. Unfortunately that theme cannot be pursued here.

Acknowledgment

First and foremost, thanks must be extended to L. A. Belady, a close collaborator for almost ten years. Many others, particularly colleagues and associates at ALMSA, IBM, Imperial College, and WG 2.3 have contributed through their comments, questions, critique, and original thoughts. All of them deserve and receive the author's grateful acknowledgments and thanks for their individual and collective contributions. The author would like to single out Prof. W. M. Turski for the major contribution he made on his recent visit to London. Also, sincere thanks to Dr. G. Benyon-Tinker, Dr. P. G. Harrison, and Dr. C. Jones for their detailed and constructive criticism of an early draft of this paper and R. Bailey for his artistic support. Finally, the author would like to acknowledge the constant support of his wife, without which neither the work itself nor this paper would have been possible.

References

[1] J. Goldberg, Ed., in *Proc. Symp. High Cost of Software* (Naval Post-grad. School, Monterey, CA). Menlo Park, CA: SRI, 1973, 138 pp.

[2] M. M. Lehman, "The environment of design methodology," Keynote Address, in *Proc. Symp. Formal Design Methodology*, T. A. Cox, Ed. (Cambridge, England), Apr. 1979. Harlow, England: STL Ltd., 1980, pp. 18–38.

[3] —, "The software engineering environment," Infotech State of the Art Rep., "Structured software development," P. J. L. Wallis, Ed., vol. 2, pp. 147–163, 1979.

[4] W. A. Wulf, "Languages and structured programs," in *Current Trends in Programming Methodology*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 33–60.

[5] L. A. Belady, Ed., *Proc. IEEE Special Issue on Software Engineering*, vol. 68, Sept. 1980.

[6] B. W. Boehm "Software engineering," *IEEE Trans. Comput.*, vol. C-25, pp. 1226–1241, Dec. 1976.

[7] W. M. Turski, *Computer Programming Methodology*. London, England: Heyden, 1978, 208 pp.

[8] B. W. Boehm, "Software engineering—As it is," in *Proc. 4th Int. Conf. Software Engineering* (Munich, Germany), pp. 11–21, Sept. 1979. (IEEE Cat. no. 79CH1479-5C.)

[9] L. A. Belady and M. M. Lehman, "Characteristics of large systems," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, part I, ch. 3, pp. 106–142 (sponsored by the Tri-Services Committee of DoD); and in *Proc. Conf. Research Directions in Software Technology* (Brown University, Providence, RI), Oct. 10–12, 1977.

[10] M. M. Lehman, "Programs, cities, students—Limits to growth?" Inaugural Lecture, May 14, 1974, *ICST Inaugural Lecture Series*,

vol. 9, pp. 211-229, 1970-1974; and in *Programming Methodology*, D. Gries, Ed. New York: Springer-Verlag, 1979, pp. 42-69.
[11] ——, "On understanding laws, evolution and conservation in the large program life-cycle," *J. Syst. Software*, vol. 1, no. 3, pp. 213-232, 1980.
[12] W. M. Turski, "Report on an SRC-sponsored visit to Imperial College," Dep. Computing, Imperial College of Science and Technology, Univ. of London, London, England, Oct. 1979, 2 pp.
[13] F. L. Bauer, H. Partsch, P. Pebber, and H. Wessner, "Notes on the project-CIP: An outline of a transformation system," TUM-INFO-7729, Tech. Univ. Munich, 67 pp, 1977.
[14] J. Darlington, "Programming transformation: An introduction and survey," *Comput. Bull.*, ser. 2, no. 22, pp. 22-24, Dec. 1979.
[15] H. A. Simon, *The Sciences of the Artificial*. Cambridge, MA: M.I.T. Press, 1969, 123 pp.
[16] R. A. Demillo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," *Commun. Ass. Comput. Mach.*, vol. 22, no. 5, pp. 271-280, May 1979; and no. 11, pp. 621-630, Nov. 1979.
[17] C. A. R. Hoare, "Review of a paper by Demillo, Lipton and Perlis: 'Social processes and proofs of theorems and programs,'" *ACM Comput. Rev.*, vol. 22, no. 8, rev. no. 34897, p. 324, Aug. 1979.
[18] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *Nordisk Tidsrift for Informations. Bahandling, Sweden*, vol. 8, pp. 174-186, 1969.
[19] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, no. 10, pp. 576-583, Oct. 1969.
[20] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, no. 12, pp. 1053-1058, Dec. 1972.
[21] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, no. 4, Apr. 1971, pp. 221-227.
[22] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. New York: Academic Press, 1972, pp. 1-81.
[23] M. A. Jackson, *Principles of Program Design*. London, England: Academic Press, 1975, 299 pp.
[24] N. Wirth, "The module: A system structuring facility in high-level programming languages," in *Proc. Symp. Programming Languages and Programming Methods* (Sydney, Austral.), J. Tobias, Ed. Lucas Hts., New South Wales: AAEC, 1979.
[25] B. Liskov and S. Zilles, "An introduction to formal specification of data abstraction," in *Current Trends in Programming Methodology*, vol. 1, *Software Specification and Design*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 1-32.
[26] C. Jones, *Software Development—A Rigorous Approach*, Englewood Cliffs, NJ: Prentice-Hall, 1980, 400 pp.
[27] M. Shaw, "The impact of abstraction concerns on modern programming languages," this issue, pp. 1119-1130.
[28] M. M. Lehman, "The funnel—A functional channel," Imperial College, Dep. Computing, Univ. of London, Res. Rep. 77/29, July 1977, 14 pp.; and *IBM Tech. Disclosure Bull.*, 1976.
[29] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. New York: Academic Press, 1972, 220 pp.
[30] R. C. Linger, and H. D. Mills, "On the development of large, reliable programs," in *Current Trends in Programming Methodology*, vol. 1, *Software Specification and Design*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 120-139.
[31] M. M. Lehman, "OS-VS2-MVS long range prognosis," Private communication, MML-104, 13 pp. Apr. 15, 1975.
[32] T. A. Dolotta and J. R. Mashey, "An introduction to the programmer's workbench," in *Proc. 2nd Int. Conf. Software Engineering* (San Francisco, CA) Oct. 1976. (IEEE Cat. no. 76CH-1125-4C, Oct. 1976, pp. 164-168.)
[33] A. F. Hutchings, R. W. McGuffin, A. E. Elliston, B. R. Trauter, and P. N. Westmacott, "CADES—Software engineering in Practice," *Proc. 4th Int. Conf. Software Engineering* (Munich, Germany), Sept. 1979. (IEEE Cat. no. 79CH-1479-5C, Sept. 1979, pp. 136-152.)
[34] J. N. Buxton, "Requirements for ADA programming support environment—STONEMAN," U.S. Dep. of Defense, Washington, DC, 44 pp., Feb. 1980.
[35] T. E. Bell, D. C. Bixler, and M. E. Dyer, "An extendable approach to computer-aided software requirements engineering," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 49-59, Jan. 1977.
[36] M. W. Alford, "Software requirements engineering methodology (SREM) at the age of two," in *Proc. COMPSAC 78*, pp. 332-339, Nov. 1978. (IEEE Cat. no. 78CH1338-3C.)
[37] K. Heninger, "Specifying requirements for complex systems: New techniques and their application," in *Proc. Specification of Reliable Software Conf.*, pp. 1-14, Mar. 1979. (IEEE Cat. no. 74CH1401-9C.)
[38] R. T. Yeh, and P. Zave, "Specifying software requirements," this issue, pp. 1077-1085.
[39] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.
[40] G. J. Myers, *Composite/Structured Design*. New York: Van Nostrand Reinhold, 1978, 134 pp.
[41] D. T. Ross, and K. E. Schoman, "Structuring analysis for requirements definition," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 6-15, Jan. 1977.
[42] ——, "Structured analysis (SA): A language for communicating ideas," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 16-33, Jan. 1977.
[43] T. Demarco, *Structured Analysis and System Specification*. New York: Yourdon Press, 1978, 352 pp.
[44] B. W. Liskov and V. Berzins, "An appraisal of program specifications," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, Part 2.1, ch. 7, pp. 106-142 (sponsored by the Tri-Services Committee of DoD); and in *Proc. Conf. Research Directions in Software Technology* (Brown University, Providence, RI), pp. 276-301, Oct. 10-12, 1977.
[45] J. N. Buxton, and E. Randell, Eds., "Software engineering techniques," Rep. Conf. sponsored by the NATO Science Committee (Rome, Italy), Oct. 1969. (Brussels, 164 pp, 1970.)
[46] P. Van Leer, "Top-down development using a program design language," *IBM Syst. J.*, vol. 15, no. 2, pp. 155-170, 1976.
[47] Teichroew and E. A. Hershey III, "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 41-48, Jan. 1977.
[48] R. P. Yeh, "Current trends in programming methodology," vol. 1, *Software Specification and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1977, 275 pp.
[49] T. A. Cox, Ed., *Proc. Symp. Formal Design Methodology* (Cambridge, England), 1977. Harlow, England: STL Ltd., 1980, 350 pp.
[50] F. W. Zurcher and B. Randell, "Iterative multi-level modelling—A methodology for computer system design," in *Proc. IFIP Congr. 1968* (Edinburgh, Scotland), pp. D138-142, Aug. 1968.
[51] L. Peters, "Software design engineering," this issue, pp. 1085-1093.
[52] G. H. Swaum, *Top-Down Structured Design Techniques*. New York: Petrocelli Books, 1978, 140 pp.
[53] E. Miller and W. E. Howden, Eds., "Tutorial: Software testing and validation technique," *IEEE Comput. Soc.*, 423 pp., 1978. (IEEE Cat. no. EHO-138-8.)
[54] J. B. Goodenough and L. M. Clement, "Software quality assurance testing and validation," this issue, pp. 1093-1098.
[55] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156-173, June 1975.
[56] L. A. Belady and P. M. Merlin, "Evolving parts and relations—A model of system families," IBM Res. Rep. RC6677, 14 pp, Aug. 1977.
[57] M. M. Lehman and L. H. Putnam, Eds., "Software phenomenology, working papers of the (first) software life cycle management workshop (Airlie, VA), Aug. 1977. Fort Belvoir, VA: ISRAD/AIRMICS, Computer Systems Command, U.S. Army, Dec. 1977, 682 pp.
[58] U. R. Basili, E. Ely, and D. Young, Eds., "Second software life-cycle management workshop, 21-22 Aug. 1978 (Atlanta, GA)," 220 pp., Dec. 1978. (IEEE Publ. no. 78CH1390-4C.)
[59] C. P. Felix and C. E. Walston, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54-73, 1977.
[60] L. H. Putnam and R. W. Wolverton, "Quantitative management—Software cost estimating," in *Proc. Comp. Soc. 77, IEEE Computer Software and Applications Conf. (Tutorial)*, 326 pp., Nov. 1977. (IEEE cat. no. EH0129-7.)
[61] L. H. Putnam, "The influence of the time-difficulty factor in large scale development" in *Proc. Software Phenomenology Working Papers of the (first) Software Life-cycle Management Workshop* (Airlie VA), Aug. 1977. Fort Belvoir, VA: ISRAD/AIRMICS, Computer Systems Command, U.S. Army, Dec. 1977, pp. 307-312.
[62] B. W. Boehm, and R. W. Wolverton, "Software cost modelling—Some lessons learned," in *Proc. 2nd Software Life-cycle Management Workshop*, Aug. 21-22, 1978 (Atlanta, GA) pp. 129-132, Dec. 1978. (IEEE Publ. no. 78CH1390-4C.)
[63] F. N. Parr, "An alternative to the Rayleigh curve model for software development effort," *IEEE Trans. Software Eng.*, vol. SE-6, May 1980, pp. 291-296.
[64] S. C. Aron, *The Program Development Process, Part II The Programming Team*. Reading, MA: Addison-Wesley, 1980.
[65] M. Halstead, *Elements of Software Science*. New York: Elsevier, 1977, 127 pp.
[66] A. Fitzsimmons and T. Love, "A review and evaluation of software science," *Computing Survey*, vol. 10, no. 1, pp. 3-18, Mar.

1978.
[67] D. B. Johnston and A. M. Lister, "Software science and student programs," *Software: Pract. and Exp.*, vol. 10, no. 2, pp. 159–1960, Feb. 1980.
[68] M. M. Lehman, "The programming process," IBM Res. Rep. RC2722, p. 47, Dec. 1969.
[69] L. A. Belady and M. M. Lehman, "Programming system dynamics or the meta-dynamics of systems in maintenance and growth," IBM Res. Rep. RC3516, 30 pp, Sept. 1971.
[71] M. M. Lehman, "Programming systems growth dynamics," infotech State of the Art Lectures, no. 18, "Computer reliability," State of the Art Lectures, no. 20, pp. 391–412, 1974.
[72] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 225–252, 1976.
[73] J. S. Riordan, "An evolution dynamics model" in *Proc. Software Phenomenology, Working Papers of the (first) Software Life-cycle Management Workshop* (Airlie, VA) Aug. 1977," ISRAD/AIRMICS, Computer Systems Command, U.S. Army, (Fort Belvoir, VA), pp. 339–360, Dec. 1977.
[74] J. K. Patterson and M. M. Lehman, "Preliminary CCSS systems analysis using evolution dynamics techniques," in *Proc. Software Phenomenology, Working Papers of the (first) Software Life-cycle Management Workshop* (Airlie, VA), Aug. 1977, ISRAD/AIRMICS, Computer Systems Command, U.S. Army, Fort Belvoir, VA, Dec. 1977, pp. 324–332.
[75] M. Woodside, "A mathematical model for the evolution of software," *J. Syst. Software*, vol. 1, no. 3, 1980.
[76] C. K. S. Chon Hok Yuen, "A Phenomenology of Program Maintenance and Evolution," Ph.D. dissertation, Dep. Computing, Imperial College of Science and Technology, Univ. of London, London, England, to be published.
[77] M. M. Lehman, "Programs, programming and the software life-cycle," CCD-ICST Res. Rep. 80/6, 48 pp. Apr. 1980.
[78] ——, "Human thought and action as an ingredient of system behavior," in *Encyclopaedia of Ignorance*, Duncan and Weston-Smith, Eds., Oxford, England: Pergamon Press, 1977, pp. 347–354.
[79] ——, "Laws of program evolution—Rules and tools for programming management," Infotech State of the Art Conf., "Why software projects fail," pp. 11/1–11/25, Apr. 9–11, 1978.
[80] M. M. Lehman and F. N. Parr, "Program evolution and its impact on software engineering," in *Proc. 2nd Int. Conf. Software Engineering* (San Francisco, CA), pp. 350–357, Oct. 1976. (IEEE Cat. no. 76CH1125-4C.)
[81] F. P. Brooks, *The Mythical Man-Month—Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1975, 195 pp.
[82] P. Naur and B. Randell, Eds., "Software engineering: Report on a conference sponsored by the NATO science Committee," (Garmisch, Germany), Oct. 7–11, 1968. Brussels, Belgium: Scientific Affairs Division, NATO, 1969, 231 pp.
[83] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proc. 2nd Int. Conf. Software Engineering* (San Francisco, CA), pp. 592–605, Oct. 1976. (IEEE Cat. no. 76CH1124-4C.)
[84] F. N. Parr and M. M. Lehman, "State of the art survey of software reliability," Dep. Computing, Imperial College, London, England, Res. Rep. 77/15, 102 pp.
[85] J. D. Musa, "The measurement and management of software reliability," this issue, pp. 1131–1143.
[86] T. J. McCabe, "A complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, pp. 308–320, Dec. 1976.
[87] M. M. Lehman, "Complexity and complexity change of a large applications program," ERO Research Proposal, 32 pp, Mar. 1977.
[88] L. A. Belady, "Software complexity," in *Proc. Software Phenomenology, Working Papers of the (first) Software Life Cycle Management Workshop* (Airlie VA) Aug. 1977. Fort Belvoir, VA: ISRAD/AIRMICS, Computer Systems Command, U.S. Army, Dec. 1977, pp. 371–384.
[89] E. T. Chen, "Program complexity and programmer productivity," *IEEE Trans. Software Eng.*, vol. SE-1. pp. 187–193, May 1978.
[90] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, 1976.
[91] C. B. Jones, "The role of formal specifications in software development," in *Proc. Infotech State of the Art Conf. on Life-cycle Management*, 1980.
[92] H. Kopetz, F. Lohnert, and W. Merker, "An outline of project MARS—maintainable real-time system," Technische Universitat, Berlin, Germany, Bericht 79-09, 19 pp, July 1979.