Rules and Tools for Software Evolution Planning and Management

Meir M. Lehman Juan F. Ramil

Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ
tel + 44 - 207 - 594 8214 fax 44 - 207 - 594 8215
{mml,ramil}@doc.ic.ac.uk
http://www.doc.ic.ac.uk/~mml/feast

Abstract

When first formulated in the early seventies, the *laws* of software evolution were, for a number of reasons, not widely accepted as relevant to software engineering practice. Over the years, however, they have gradually become recognised as providing useful inputs to understanding of the software process. Now eight in number, they have been supplemented by the software uncertainty principle and the FEAST (*F*eedback, *E*volution *A*nd *S*oftware *T*echnology) hypothesis.

Based on all these and on the further results of the FEAST research projects this paper develops and presents some fifty rules for application in software system process planning and management and indicates tools available or that could usefully be developed to support their application. The listing is structured according to the laws that encapsulate the observed phenomena and that lead to the recommendations. Each sub-list is preceded by a textual discussion providing at least some of the reasoning that has led to the recommended procedures. The references direct the interested reader to the literature that records observed behaviours, interpretations, models and metrics obtained from industrially evolved systems, and from which the recommendations were derived.

Keywords: assumptions, *E*-type software, FEAST, feedback, laws of software evolution, software management, process improvement, rules for process planning and management, software evolution.

1. Introduction

The software *evolution phenomenon* manifests itself in the common need, since the beginnings of electronic computing, for continuing maintenance and periodic upgrades of software used in real world applications. Though studied for over thirty years [Lehman 1969; Belady and Lehman 1972] (both papers included in [Lehman and Belady 1985]), the phenomenon and its economic and social importance has only recently been *widely* recognised and accepted as worthy of serious study [e.g., IWPSE 1998; Pfleeger 1998b].

One may consider and study software evolution from different points of view. The principal issue of concern may be with the ***what*** and the ***why*** of evolution. One focuses on the properties of the phenomenon, its causes and identification of the drivers underlying development and maintenance activity. This is the view adopted by several groups (including the present authors) [e.g., Cook *et al.* 2000; FEAST 2001; Godfrey and Tu 2000; Hsi and Potts 2000; Kemerer and Slaughter 1999; Shepperd 2000]. An alternative view is primarily concerned with the *how* of evolution, that is, with the methods, tools and technology to facilitate disciplined and efficient software change [e.g., IWPSE 1998; ISPSE 2000; FFSE 2001; IWPSE 2001]. Clearly the two views complement each other, though the latter is, by far, the more common. But to master the technology and to justify the deployment of good practice in industrial processes, understanding the *what* and *why* is essential [Lehman and Ramil 2000]. Such insight and understanding provides an opening for developing means for achieving the goals of effective evolution in a given development, application domain or specific technology, the *how* of evolution.

To appears in Annals of Software Engineering Vol. 11, 2001

This paper, a substantially revised version of an earlier contribution [Lehman 2000a]) summarises practical implications of the results of the FEAST (*F*eedback, *E*volution *A*nd *S*oftware *T*echnology) and earlier studies, their observations, models derived from them and their interpretation, as determined in extensive discussions amongst themselves and with collaborator personnel. It presents some fifty or so recommendations, by no means exhaustive, of software evolution *planning* and *management* and identifies some of their implications. Recommendations are only good to the extent that they can be reliably and cost-effectively implemented. Proposals for project planning and management tools are therefore also included though much development is required in that area.

Overall, the recommendations must, clearly, be applied intelligently. This requires insight into the relevant general observations, the data, the models and the interpretations on which the conclusions are based. This paper cannot provide such detail and is restricted to a general review of the relevant phenomenology. Where appropriate, references to additional sources of information possible modelling approaches and structures, for example, are provided.

Observations and rules relevant to software system evolution planning and management were first identified during studies of the evolution of OS/360-70 and other systems between 1968 [Lehman 1969] and 1985 [Lehman and Belady 1985]. They were first summarised in a 1978 paper [Lehman 1978]. More recently the FEAST/1 project (1996-1998) [Lehman and Stenning 1996], with the collaboration of ICL, Logica, MoD-DERA and Matra-BAe Dynamics, has been able to confirm, refine and extend the earlier results. This was made possible by analysis of data on the evolution of their respective systems, VME Kernel, the FW Banking Transaction system and a Matra-BAe defence system. Data on a real time Lucent Technologies system also became available for analysis during this time. FEAST/2 (1999 - 2001) [Lehman 1998b], which includes BT Labs as a collaborator, has advanced the conclusions and understanding of the evolution phenomenon reached in FEAST/1 and in earlier work. Broadly speaking, the long-term evolutionary behaviour of the *release-based*[1] systems studied was qualitatively equivalent, though varying in detail. This despite the very different application and implementation domains in which the systems were developed, evolved, operated and used, and to which the respective data sets related. Moreover, the results were broadly compatible with and supportive of those obtained in the earlier studies [Lehman and Belady 1985].

The systems studied so far have been evolved applying paradigms as used for the past thirty or so years. Extending the results obtained to the products of newer paradigms such as OO design and component-based systems and open source software requires further investigation. Thus, for example, the more significant role of *integration* may change evolutionary characteristics significantly. Hypotheses concerning the evolution phenomenology of component and COTS-intensive processes [Lehman and Ramil 1998] have been proposed but await empirical investigation. A proposal for such an investigation is part of a project now being planned. The results of a study of the evolution of Linux, an open source system, reveals commonalties with earlier evolution studies but also differences [Godfrey and Tu 2000]. This result provides an opportunity for the development of deeper insight by seeking understanding for the source of both commonalties and differences. That said, the many similarities observed in the long-term evolutionary behaviour of the systems studied over a period of rapidly evolving technology suggests that the observed behaviour and the rules derived therefrom are not primarily dependent on specific technologies, application areas or environments employed. It seems likely that they have more fundamental explanations shared by all of the systems. This suggests that it may be possible to extend the conclusions presented here to yield results of wide validity in the field of software, and ultimately more general, evolution.

In presenting the practical implications of the FEAST work to date, the paper follows the order

[1] In this paper the only software evolution paradigms considered are those based on the release of successive versions of the system to users

in which the *laws of software evolution* [Lehman 1974, 1978, 1997; Lehman and Belady 1985; Lehman *et al.* 1998] and other concepts and principles [Lehman and Belady 1985; Lehman 1989, 1990] were formulated. This ordering has, however, only historical significance. It is now recognised that the laws are not independent of one another. The dependencies and relationships between them are currently being investigated and formalised. An introductory description of that investigation is now available [Lehman and Ramil 2000]. The relationship is, unavoidably, reflected in this paper by a degree of overlap and repetitiveness between sections. In any event, its use in structuring this paper is a matter of convenience and has no other implications.

2. Laws of Software Evolution

The eight laws of software evolution, formulated over the 70s and 80s [Lehman 1974, 1978, 1980a,b; Lehman and Belay 1985], are listed in the Table below. They first emerged from a follow up of the 1969 study of the evolution of IBM OS/360 [Lehman 1969] and were strengthened by the results of other evolution studies in the seventies. Additional support came from an ICL study in the eighties [Kitchenham 1982]. The present wording incorporates minor modifications that reflect new insights gained over the years [Lehman and Belady 1985; Lehman 1994, 1997; Lehman *et al.* 1998; FEAST 2001]. Over the years, the laws have gradually become recognised as providing useful inputs to understanding of the software process [e.g., Pfleeger 1998a,b] and have found their place in a number of software engineering curricula. Analysis of the release-based data obtained from FEAST collaborators has indicated that six (I,II,III,V,VI and VIII) of the eight laws are, in general, consistent with the processes reflected by the data [Lehman *et al.* 1998; Lehman *et al.* 2000; FEAST 2001]. With regards to the remaining two laws (IV and VII), one (IV) is currently being investigated and the other (VII) was neither supported nor negated by available data.

No.	Brief Name	Law
I 1974	Continuing Change	An *E*-type system must be continually adapted else it becomes progressively less satisfactory in use
II 1974	Increasing Complexity	As an *E*-type system is evolved its complexity increases unless work is done to maintain or reduce it
III 1974	Self Regulation	*Global E*-type system evolution processes are self-regulating
IV 1978	Conservation of Organisational Stability	Average activity rate in an *E*-type process tends to remain constant over system lifetime or segments of that lifetime
V 1978	Conservation of Familiarity	In general, the average incremental growth (growth rate trend) of *E*-type systems tends to decline
VI 1991	Continuing Growth	The functional capability of *E*-type systems must be continually increased to maintain user satisfaction over the system lifetime
VII 1996	Declining Quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of an *E*-type system will appear to decline as it is evolved
VIII 1996	Feedback System (Recognised 1971, formulated 1996)	*E*-type evolution processes are multi-level, multi-loop, multi-agent *feedback* systems

Table. Current Statement of the Laws.

The laws have been subject to strong criticism from the moment of first formulation of laws I - III [Lehman 1974]. Expressed concern included the absence of precise definitions or statement of assumptions, their being based on a single and atypical data source (OS/360), an allegation that OS/360 represented unique (IBM) development and marketing domains and, more recently, that it reflected outmoded technology. Further criticism was based on an alleged lack of significant support for some of the laws when applying statistical tests [Lawrence 1982]. That author did not, however, address issues related to the detection of qualitative commonalties from quantitative data sets having the characteristics of the data being analysed, for example, how statistical tests could be applied or interpreted when applied to very small data sets. Finally, critics expressed concern about use of the term *laws* in relation to observations about phenomena directed, managed and reflecting human activity. This latter was countered by one of the present authors (mml) who noted that it was precisely such human involvement that justified use of the term. The term was selected just because each law encapsulates organisational and sociological factors that lie outside the realm of software engineering and the scope of software developers. From the perspective of the latter they must be accepted as laws. This reasoning is supported by the analogous application of the term to the *forces of supply and demand* in economic systems. These forces are "… so powerful that they constantly break through all barriers erected for their suppression … [Baumol 1967, p. 415]".

3. *S*- and *E*-type Program Classification

3.1. Definitions

The first published reference to this program scheme classification appears to have been in 1980 [Lehman 1980b]. Previous to that time [e.g., Lehman 1978] Lehman and Belady's work on program growth dynamics and software evolution had considered the latter as a characteristic of *large* programs. The common view was, and remains, that a program is *large* if, for example, it contained more than some arbitrary number, perhaps 100K lines of deliverable source code (KLOC). Lehman was critical of this view on the grounds of its arbitrariness. Instead he proposed a definition that classified a program as *large* if "… it was designed, or had been developed or maintained in a management structure involving at least two groups" [Lehman 1979]. That fact alone could, it was asserted, account, directly or indirectly, for many of the observed behavioural phenomena. Even with this definition, however, concerns about the suitability of *large* as the primary basis for studying software evolution remained. The classification scheme of three program types S, P, E followed [Lehman 1980b].

An *S*-type program may be defined as one whose acceptability on completion of development depends only on in its satisfying, in the mathematical sense, a formal specification. The designation *S* was chosen to indicate the definitive role that the *specification* plays in defining the required properties of the product. An alternative interpretation of the *S* is provided by Shari Lawrence Pfleeger in her discussion of the classification scheme [Pfleeger 1998a,b]. She implies that the designation *S* stands for *static* since, it stands in stark contrast to *E*-type systems, defined immediately below, that are essentially evolutionary, must undergo continual evolution to remain satisfactory.

E-type programs were originally defined as "… programs that mechanise a human or societal activity..." [Lehman 1980b]. In subsequent papers and presentations that informal definition was extended to include all programs that "… operate or address a problem or activity in the real world". Already in the 1980 paper it was shown that, to remain satisfactory *E*-type programs must, by their very nature, be continually changed and updated. They must be continually *evolved.* Hence the designation *E.*

A third type, *P* was initially defined in a way that made it intermediate between the other two. It soon became clear that, from many points of view, the *P*-type could be subsumed in one or other of the two. This type is, therefore, not considered further here. The interested reader may consult [Lehman 1980b] and subsequent publications [e.g., Lehman and Belady 1985].

It has long been realised that the definitions given above, and slightly refined alternatives that have been used, need to be made precise. This implies their formalisation. As mentioned elsewhere in this paper, a proposal to undertake the development of a formal theory of software evolution has been prepared [Lehman 2000b]. One of the very first tasks of that activity must be to develop formal definitions of the terms *S* and *E*-type as applied to software systems.

3.2. Basic Properties

The laws *apply*, in the first instance, to *E-type* programs and the associated *global* processes. The latter includes all activities involved in system evolution including but not limited to those undertaken by technical, management, marketing, user support personnel and users, etc [Lehman 1994]. As above indicated, type *E* refers to programs actively (and regularly) used to solve a problem or address an application in a real world domain. A key characteristic of such systems is that their *acceptability* depends on the results delivered to users and other stakeholders. *E*-type properties include expectations that, at least for the moment, stakeholders are *satisfied* with the system as is. Over and above *functionality*, factors such as *quality* (however defined), *behaviour* in execution, *performance*, ease of use, *changeability* and so on will also be of concern.

Evolution is intrinsic to *E*-type systems. They must be continually enhanced, adapted and fixed if they are to remain effective in a changing world and an evolving application environment. The study of *E*-type systems is important because the majority of systems upon which businesses and organisations rely for their operation, are of this type. The laws reflect properties termed elsewhere [Lehman and Ramil 2000] *behavioural invariants* of *E*-type products. Processes used to produce, update and adapt, that is, to *evolve* them as the application domain changes, are similarly termed type *E*. In this they differ from *S-type programs* where the *sole* criterion of acceptability is *correctness*, in the mathematical sense. Only that which is explicitly included in the specification or follows from what is so included is of concern in assessing (and accepting) the program and the results of its execution. An *S*-type program is completely defined by and is required to be *correct* with respect to a fixed and consistent *specification* [Lehman 1980b]. A property not so included may be present or absent, deliberately, by oversight or at the programmer's whim. We note that *S*-type specifications, and with them, the systems that implement them, may also evolve. For example, compilers are now, in general, formally specified, and so are of type *S*. They must be evolved when, for example, the language they implement changes.

3.3. Role of Assumptions

The importance of *S*-type programs lies in their being, by definition, mathematical objects about which one may reason. Thus they can be *verified*, that is, *proven* correct with respect to a program specification. A *correct* program possesses all the properties required to satisfy its specification. Other properties are implicitly declared *don't cares* by their omission from the specification. It is entirely determined by the specification. Once the specification has been fixed and verified, acceptance of an *S*-type program is independent of any *assumptions* [Lehman 1989, 1990; 1998a; Lehman and Ramil. 2000]. Note that such assumptions may be explicit or implicit conscious or unconscious, by commission or omission. At some later time the program may require fixing or enhancement as a result of an oversight in its preparation, changes in the purpose for which it is being executed, changes in the operational domain or some other reason. An updated specification, including changes to the assumption set, can then be prepared and a new program derived, perhaps by modification of the old.

At the detailed level of program development or evolution activity, the work is carried out by individuals. Assumptions they make during the course of their work become a part of the program properties even though not a part of the original specification and not, in general, documented. As the operational domain or other circumstances change some assumptions may become invalid and affect program behaviour in unpredictable and/or unacceptable fashion.

3.4. Role of S-type Programs in E-type Systems

The role of *S*-type programs in the programming process now becomes clear. For both *S*- and *E*-type programs, the unconsidered injection of assumptions may be minimised by providing a *precise* specification for each individually assigned task, that is, giving implementers *S*-type assignments. With this, one may minimise the risk that assumptions incompatible with the specification are embedded in the assignment and, also maximise awareness of implicit assumptions being imposed on the implementation. In doing so, *S*-type programs become the *bricks* from which larger programs and complex systems are built. A recognised set of assumptions, particularly such as relate to current or probable future states of the application and its domain of operation, will then be either explicit in the specification or stand as *don't cares* by omission. This permits the elemental parts of individual products to be assessed objectively by verification against their specification. At the same time one would wish, as far as possible, to assess the likely future validity of each as any of the domains in which its operationally embedded or executed may be expected to change. The better the record of assumptions, the more it is reviewed, the simpler will it be to maintain the programs valid in a changing world, the less likely it is to experience unexpected or strange behaviour in execution, the more confidence one can have in the operation of the program when integrated into a host system and later when operational in the real world. But as the system evolves, the specifications from which the *S*-type programs are derived, will inevitably have to be changed to satisfy the changing domains within which they work and to fulfil the current needs of the application they are serving. The elemental bricks, too, need to evolve. That comment is significant in the context of, for example, component-intensive and COTS software. As is becoming increasingly recognised [e.g. Lehman and Ramil 1998; GCSE 2000] components of *E*-type applications and systems must also evolve.

When a system is operational, any *S*-type bricks within it operate in the real world. In that context they will acquire *E*-type characteristics [Lehman *et al.* 1998]. The restriction of the laws to *E*-type systems does not, therefore, decrease their practical significance despite the fact that, in their original conception and creation, those elements that are of type *S* will have been specified. Both *S*- and *E*-type programs have a role to play in system development and evolution.

3.5. Implications of the Program Classification Scheme

Guidelines that follow from the preceding discussion are listed in this section, others under the headings that follow. Note that this list (and all others that follow) is to be considered randomly ordered. No implications, for example in terms of relative importance, are to be drawn from the position of any item.

a. all properties and attributes required to be possessed by software products created or modified by individual developers should be explicitly identified in a specification that then serves as the task definition
b. assumptions must be captured and recorded when incorporated into a program specification, a program or into their documentation. This is so even if, at the time they are adopted and embedded in them, they are compatible with the application domain
c. the (long-term) goal should be to express specifications formally
d. it must be a goal of every process to capture and retain assumptions underlying the specification, both those that form part of its inputs and those arising during the subsequent development process
e. when *validating* any specification and program a conscious effort must be made to identify, document and validate all assumptions implied by individual or combinations of statements, ensuring also their continued validity in the circumstances of the moment, that is the state of the application domain, of the execution domain and of the software system itself
f. it must be recognised and agreed by the assignor that whatever is not so included is left to the assignee who must ensure that the decision is not inconsistent with the specification and is either:
    - 1.documented in an *exclusion* document or
    - 2.formally approved and added to the specification and to all supporting and appropriate user documentation
g. tools to assist in the implementation of these recommendations and to support their systematic application should be developed and introduced into practice.

Practical implications of the laws and the tools suggested by them follow. Many of the items will appear self-evident. What is new is the unifying conceptual framework on which they are based.

4. First Law: Continuing Change

An E-type system must be continually adapted else it becomes progressively less satisfactory in use

The need for change reflects a need to *adapt* the system as the outside world, the domain being covered and the application and/or activity being supported or pursued, changes. Such exogenous changes are likely to invalidate assumptions made during system definition, development, validation, installation and application or render them unsatisfactory. The software reflecting such assumptions must then be adapted to restore their validity.

Every *E*-type system is a *model* (in the mathematical sense [Turski and Maibaum 1987]) of the application in its operational domain. Both the real world and every application have an unbounded number of attributes or properties, however defined. Being part of the real world, the operational domain in which the system operates is initially undefined and is, therefore, also intrinsically unbounded. Software systems, on the other hand, are essentially finite. Therefore, the process of abstraction and transformation defining and developing the application system and its software involves *assumptions* about, for example, what capabilities are to be included in the final program and what properties the latter should display in execution. This process of finitisation excludes all elements/attributes of the operational domain and the application not specifically included. As a model of the real world, the system is incomplete [Lehman 1976, 1989, 1990; Lehman and Belady 1985]. As briefly discussed above, some of the assumptions will be explicit, others implicit, some by inclusion, others by exclusion. They will be reflected in the system, by choice of theories and algorithms, code, lists, parameters, call sequencing, documentation and so on. Exclusions may be explicit or by omission and omissions are as real in impacting system operation as are inclusions. Thus every *E*-type system has embedded in it an infinite assumption set whose composition will determine the domain of valid application in terms of execution environments, time, function, geography, the detail of many levels of the implementation and so on.

It may be that the initial set of assumptions was *valid* in the sense that it defined a system that had all the required and desired properties. Any limitations it imposed and the system behaviour it assumed did not render the system unacceptable in operation at the time of its introduction. However, with the passage of time user experience increases, user needs and expectation change, new opportunities arise, applications expand in terms of numbers of users, details of usage etc, new needs and constraints arise in the operational domain and so on. Thus, a growing number of assumptions may become invalid. This is likely to lead to less than acceptably satisfactory performance in some sense and hence to requests for change. There will also be changes in the real world that impact the operational domain so requiring changes to the system to restore it to being an acceptable model of the operational domain. Taken together, these facts lead to the unending *maintenance* that has been the universal experience of computer users since the start of serious computer application by all regular computer users.

An upper limit for the number of changes implemented per release for example, to be *safe* (good potential for success) is likely to exist. As the number, magnitude and orthogonality to system architecture of changes implemented in a release increases, complexity, assumption and defect injection rates grow, probably more than linearly, with respect to that number.

There follows a partial listing of practical consequences of this unending need for change to every *E*-type system in continuing use to adapt it to changing stakeholder views and changing operational domains:

a. change validation must address the change itself, actual and potential interaction with the remainder of the system and *impact* [Bohner and Arnold 1996] on the remainder of the system
b. comprehensive documentation must be created and maintained up-to-date as changes accumulate
c. as the design and implementation of changes proceed, all aspects including, for example, the issue being addressed, the reasons why a particular implementation design/algorithm is being used, details of explicit assumptions, adopted and so on must be recorded in a way that will facilitate subsequent review
d. the assumption set must be reviewed as an integral part of release planning and periodically thereafter to detect any inconsistencies, domain and other changes that conflict with the existing set or that violate constraints

e. releases concentrating primarily on defect removal, performance enhancement and structural clean up will be required from time to time to maintain system viability. Strategies that have been applied in industry include daily stabilisation with internal release (the *synch-and-stabilise* process [Cusumano and Selby 1995]) and alternating change/growth and stabilisation releases [e.g. Woodside 1979]
f. models should be developed to determine the effect of system age and change rate on release stability and to provide an indication of limits to *safe* change rates. Such models may, for example, be based on numbers of changes per release over a sequence of releases or in real time. Another useful metric is the fraction of elements changed or *handled* per release or over a given time period. Metrics and models that may be used in this context are discussed elsewhere [Ramil *et al.* 2000]
g. it is beneficial to determine the number of distinct additions and changes to requirements and assumptions over constituent parts of the system per release or over some fixed time period to assess domain and system volatility. This can assist evolution release planning in a number of ways, for example, by pointing to system areas that are ripe for restructuring because of high defect rates or high volatility or where, to facilitate future change, extra care should be taken in change design and implementation.

5. Second Law: Growing Complexity

As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it

Increasing complexity arises because of the injection and the super-positioning of changes to achieve, for example, growth in functionality or satisfaction of the needs of changing operational domains. This leads to increasing internal interconnectivity and, hence, to deteriorating system structure, increasing disorder. Equally it results in increasing complexity of internal and external interfaces at all levels. These effects are amplified because, as the system ages, changes are more likely to be orthogonal to existing system structures. However, effective interaction with the system, whether as developer or user, requires one to *understand* it in its entirety, *to be comfortable* with it. As the system ages, as changes and additions to the system become ever more remote from the original concepts and structures, increasing effort and time will be required to understand and implement the changes, to validate and use the system, to ensure that the untouched portion of the system continues to operate as required. The original system architects or experts with in-depth knowledge of the software and of the application may have already left the organisation [Rajlich and Bennett 2000]. Changes and additions take longer to design and to implement, errors and the need for subsequent repair become more likely, comprehensive validation is more complex.

It follows that, for example, the number of *potential* connections and interactions between elements (objects, modules, subsystems, etc.) is proportional to the *square* of the number n of elements. Thus, as a system evolves, and with it the number of elements, the work required to ensure a correct and adequate interface between the new and the old, the potential for error and omission, the likelihood of incompatibility between assumptions, all tend to increase as n^2. The increasing remoteness of changes from the existing design will also contribute to increased inter-connectivity. All contribute to an increase in system complexity.

The growth in the difficulty of design, change and system validation, and hence in the effort and time required for system evolution, causes growth in costs and in the need for user support. This will, in general, tend to be accompanied by a decline in product quality (see section 10) and in the rate of evolution (see section 8), however defined and measured, unless additional compensatory work is undertaken. FEAST observations indicate directly that, in the long term, the average software growth rate measured in elements such as modules or their equivalent tends to decline as a function of the release sequence number as the system ages. The long term growth trends of the released-based systems so far studied in FEAST tend to follow *inverse square* trajectories (see [Turski 1996; FEAST 2001]).

Complexity control effort is largely *anti-regressive* [Lehman 1974], has no immediate value but constrains or reduces future increases in costs due to future increases in effort required or quality deterioration. The activity includes elimination of so called *dead* or *repetitive* code, re-structuring, documentation updating and so on. In general, these activities have minor or no impact in functionality, performance or other properties of the software in execution. In any event, *immediate* benefits are generally relatively small. Long-term impact is, however, likely to be significant; may, at some stage, make the difference between system survival and its demise or replacement. Anti-regressive work is exemplified by code *refactoring* activities [Fowler 1999].

Determining the level of effort for *anti-regressive* activity such as complexity control in a release or sequence of releases, what effort is to be applied, presents a major paradox. If the level is reduced or even abandoned to free resources for *progressive* [Baumol 1967] activity such as system enhancement and extension, system complexity is likely to increase, *productivity* and evolution rates to decline. This is likely to lead to stakeholder dissatisfaction, increases in future effort and cost and declines in system quality (see law VII). If, on the other hand, additional resources are provided for complexity control, resources for system enhancement and growth are likely to be reduced. Once again the system growth rate is likely to decline. In the absence of process improvement that is based on the principles examined in this paper, decline of evolution rate appears inevitable.

Based on the second law, which reflects these observations, and the measurement, modelling, analysis and other supporting evidence obtained over the years, the following observations and guidelines may be identified:

a. the many aspects of system complexity must be considered in process design, improvement and planning and, when possible, actively managed and controlled. They include but are not limited to:
    - 1. application and functional complexity – including that of the operational domain
    - 2. specification and requirements complexity
    - 3. architectural complexity
    - 4. design and implementation complexity
    - 5. structural complexity at many levels (subsystems, modules, objects, calling sequences, object usage, code, documentation, etc.)

b. there must be a conscious effort to control, and reduce complexity and its growth, wherever, possible, as modifications are made locally and in interfaces with the remainder of the system

c. complexity control is an integral part of the development and maintenance responsibility. Activity to address it must be considered whenever changes to a system or new releases are being planned

d. in planning release content for one or a series of releases the timing, degree and distribution of complexity control activity must be carefully considered. One must evaluate alternative complexity control strategies [e.g., Hops and Sherif 1995] and approaches [e.g., Fowler 1999] and select that most likely to help achieve corporate business goals or whatever else requires to be optimised

e. in general, it appears to be a sound strategy to alternate releases between those focussing primarily on complexity reduction and restructuring and those implementing major enhancement and adding new function or significant functional extension [e.g., Woodside 1979].

6. Third Law: Self Regulation

Global E-type system evolution processes are self regulating.

Patterns of incremental growth observed in FEAST and before can be explained in terms of the presence of self-regulatory feedback mechanisms (see section 11) [e.g., Lehman *et al.* 1998; FEAST 2001]. It is unlikely that these patterns are exclusively a reflection of conscious management control or their desire for *stability*[2]. These mechanisms should be identified to permit improvement of control. As a first step to their identification one may search for properties common to several projects or groups or for correlations between project or group characteristics such as size, age, application area, team size, organisational experience or behavioural patterns. One then may seek quantitative or behavioural invariants associated with each characteristic. To identify the feedback mechanisms and controls that play a role in performance self-stabilisation and to exploit them in future planning, management and process improvement the following steps will be helpful:

a. using measurement and modelling techniques as used, for example, in FEAST [Lehman *et al.* 2000; Ramil *et al.* 2000], determine typical patterns, trends, rates of growth and rates of change implementation of a number of projects within the organisation. To obtain meaningful results, data reflecting at least six to ten past releases in the systems studied was required

b. establish *baselines*, that is, typical values for process rates such as growth, defects, changes over the entire system, units changed, units added, units removed and so on. These may be counted per release or per unit time. Our experience has been that the former yields results that are more regular and interpretable because of the fact that some of the feedback mechanisms operate over releases. Initially however, and occasionally thereafter, results over release sequence number and over real time must be compared and appropriate conclusions drawn. Incremental values, that is the difference between values for successive releases or standard time intervals should also be determined, as should numbers of people working with the system in various capacities, person days, for example, in categories such as specification, design, implementation, testing, integration, customer support and so on. A third group of measures relates to quality factors. These can be expressed, for example, in pre-release and user reported defects, user take-up rates, installation time and effort, support effort, etc

c. new data that becomes available as time passes and as more releases are added, should be used to recalibrate and improve the models or to revalidate them and test their predictive power

d. analysis of FEAST/1 data, models and data patterns suggests that, in planning a new release or the content of a sequence of releases, the first step must be to determine which of three possible scenarios exists. Let m be the running average of the incremental growth of the system in going from release to release over a series of perhaps five or so releases and s the standard deviation of the incremental growth over the same interval. The scenarios may, for example, be differentiated by an indicator $m+2s$ that identifies a release plan as *safe*, *risky* or *very risky* according to the conditions listed below. Note that the rules are expressed for release based measures. For observations based on incremental growth per standard real time unit, analogous safe limits are likely to exist but will be a function of the interval between observations in a way that remains to be determined

- 1. The FEAST studies suggest a *safe* level for planned release content m (where m may decline as the system ages, see section 8). If the desired release content is less than or equal to m, growth at that rate may proceed with good potential for success. Growth at that rate over a sequence of releases is likely to be achievable

[2] Stability, as sought by management, means *planned* and *controlled* change, not constancy. Managers, including software managers, in general, abhor surprises or unexpected changes. They all desire, for example, constant workloads and increases in productivity; software managers, a steady decline in defects; senior management, a decline in costs and growth in profit and, for example, market share and sales.

- 2.The desired release content is greater than *m* but less than *m*+2*s*. The release is *risky*. It could succeed in terms of achieved functional content but serious *delivery delays* (or a perceived need to significantly increase the planned interval between releases) may result. Quality or other problems could arise. If pursued, it would be advisable to plan for a follow-on clean-up release. Even if not planned, such a small growth release is likely to be required. Note that *m+2s* has long been identified as a threshold value, for example, in statistical process control and monitoring [Box and Luceño 1997]
- 3.The desired release content is close to or greater than *m+2s*. A release with incremental growth of this magnitude is *very risky*. It is likely to cause major problems and evolution instability over one or more subsequent releases. At best, it is likely to require to be followed by a major clean up, that is, a *recovery release* in which the main emphasis is on defect fixing and *anti-regressive* work (see section 5)

e. an appropriate *evolutionary development* strategy [Gilb 1981] should be considered whenever the number of items on the *to be done* list for a release being planned would lead, if implemented in one release, to incremental growth in excess of the levels indicated above. It should prove appropriate whenever the size and/or complexity of the required addition is large. In that event, strategies to be considered include spreading the work over two or more releases, the *delivery* of the new functionality over two or more releases with mechanisms in place to return to older version if necessary, reinforcing the support group, preparing for the release by means of one or more clean up releases or, if the latter is not possible, preparing for a fast follow on release to rectify problems that are likely to appear. In either of the last two instances provision must be made for additional user support

f. it would seem that this law implies that the characteristics of the process are influenced by its system dynamics [Forrester 1961] as demonstrated by the self-stabilisation postulated in this law. That being so, the construction of system dynamic models could make an important contribution to better planning and management of the release process.

7. Fourth Law: Conservation of Organisational Stability

Average activity rate in an E-type process tends to remain constant over system lifetime or segments of that lifetime

The observations on which this law is based date back to the late seventies. Further data gathered in FEAST suggests that the activity rate (e.g., elements *changed, handled* or *handlings* per release or unit of time [Lehman and Belady 1985; Ramil *et al.* 2000]) tends to remain constant over periods or segments of system lifetime. Behaviour that is an otherwise smooth (e.g., constant average and variance) may, in the long term, be seen to display an abrupt change from time to time, breaking the overall trajectory into two or more segments. This is reflected in the most recent formulation of the fourth law as stated above.

This 'piecewise' behaviour of long-term evolutionary attributes is consistent with Boehm's statement that "...Once an organization has determined its desired level of maintenance for a software product ... the organization's social, economic, and political inertias will generally make it difficult to make significant changes in the level of effort or mode of operation. In some situations, major increases in demand from the investment segment (such as conversion) or significant increases or redirections in demand ... may cause instabilities or reorientations. But in most cases, the maintenance activity will settle into a fairly predictable equilibrium" [Boehm 1981, p. 546]. For example, Boehm recently referred to the activity triggered by the Y2K issue in many organisations as an anomaly to the original statement of the fourth law that did not account for segments in previous formulations. Other possible triggers of anomalies mentioned by Boehm are mergers, acquisitions, downsizing, and a move into e-commerce [Boehm 2000]. This view is consistent with that of Rajlich and Bennett [2000] who have discussed the presence of stages in software maintenance and evolution. In the present context the fourth law leads, amongst others, to the following recommendations:

a. process plans should allow for work rates that do not exceed the activity rate of the immediate past, unless appropriate adjustments to the process, policies and resource allocation, to support the new desired activity rate are considered
b. use activity rate metrics, such as elements *handled* per release [Lehman and Belady 1985; Ramil *et al.* 2000], to characterise process performance in association with appropriate *change impact analysis* tools [Bohner and Arnold 1996] and to provide a basis for work planning models [Ramil *et al.* 2000]
c. as discussed above, activity rate can, under some circumstances, be forced to change. Each segment of equilibrium may benefit from its individual, though in all likelihood not dissimilar, model for release planning and effort estimation, for example. This requires on-going monitoring for structural changes in the trends reflecting attributes of the product, process or related domains. Model re-calibration or adjustment during or after any such change will be required to ensure that they remain a valid tool. Tools for metric tracking and analysis that enable both monitoring of changes in evolutionary trends and detection of segments of equilibrium would be a desirable management aid
d. Rajlich and Bennett [2000] suggest that individual stages in product evolution will require application of *stage-specific* methods and tools to adequately perform software maintenance and evolution. This view has also managerial implications that cannot be discussed here. The interested reader is referred to the source [Rajlich and Bennett 2000].

8. Fifth Law: Conservation of Familiarity

In general, the average incremental growth (growth rate trend) of E-type systems tends to decline

With the partial exception of OS/360, and of intermediate periods or segments where the net growth rate appears to be increasing, a long term decline in incremental growth and growth rate appears to dominate the growth of all release-based systems studied. It might be thought that this could be due to a reduction in the demand for correction and change as the system ages but anecdotal evidence from the market place and from developers, for example, indicates otherwise. In general, there is always more work in the *waiting attention* queue than in progress or active planning. Other potential sources of declining growth rate include increasing mismatch of system structure and interfaces with the operational domain and the application addressed (see section 10), decreasing interest in the system, its move into a servicing stage in which only essential fixes are done or progressive system phase-out [Rajlich and Bennett 2000].

It is not appropriate to here speculate further on the source of the behaviour described by the fifth law. We restrict our comment to what has emerged so far from the FEAST studies. That analysis has suggested that the most likely source of the declining incremental growth rates observed is, primarily, increasing complexity as the system ages (section 5).

Given the growing complexity of the system, its workings and its functionality, achieving renewed familiarity after numerous changes, additions and removals, restoration of pre-change familiarity after change becomes increasingly difficult. This reasoning suggests that the rate of change and growth of the system be slowed down as it ages. This trend has been observed in nearly all the data studied to date but there is no evidence that the slow down is the result of such reasoning on the part of those involved. It is, instead, interpreted as a reflection of the feedback driven software process dynamics.

Further analysis, in phenomenological terms, of distributed mechanisms that control evolution rate together with models of related data, suggest the following guidelines for determining release content:

a. Collect, plot and model system growth and change data as a function of real time or release sequence numbers (*rsn*) to determine system evolution trends. Elements that may be counted include objects, lines of code (*locs*), modules, inputs and outputs, interconnections, subsystems, features, requirements, etc. As a start it is desirable to record several or even all of these measures so as to detect similarities and differences between the results obtained from the various measures and to identify those from which the clearest indications of evolutionary trends can be obtained. Once set up, further collection of such data is trivial. Procedures for their capture may already be a part of configuration management or other procedures. Once data is available, models that reflect historical growth trends may be derived and indicators derived

b. on the basis of the observations reported above in section 6, in planning further releases the following guide lines should be followed:
   - 1. seek to maintain incremental growth per release (or the growth rate in real time) at or about a level *m* (section 6). Such a running average will follow the long term system growth trend, that is, *m* will tend to decrease. Alternatively, *m* could be derived from the trend model(s) obtained following the recommendation (a) and periodically adjusted
   - 2. when the growth per release needs to be significantly greater than the level *m* indicated, seek to reduce it by, for example, spreading over two or more releases
   - 3. if limiting growth to the recommended level is difficult or not possible, plan and implement a *preparation release* that pre-cleans the system
   - 4. alternatively, allow for a longer release period to prepare to handle problems at integration, higher than normal defect injection and, hopefully, detection rates, some user discontent
   - 5. if the required release increment is near to or above some upper limit of safe growth (see section 6) the steps in 2 - 4 immediately above must be even more rigorously applied. Prepare to cope with and control a period of system instability, provide for a possible need for an increase in customer support and accept that a major *recovery release* may be required
   - 6. plan to periodically clean-up, re-structure or re-engineer the software and/or other measures to avoid the, otherwise likely to decline, system growth rate

c. develop automatic tools to help in collection, modelling and interpretation of the data as it builds up over a period of time to derive, for example, the dynamic trend patterns. A scripting language such as *Perl* [Wall *et al.* 1996] can be used to extract data from sources such as change-logs to estimate, amongst others, element growth and change rate. Adoption of a fixed standard format for change-log data will facilitate data extraction.

9. Sixth Law: Continuing Growth

The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime

This law must be distinguished from the first law which asserts 'Continuing Change'. It reflects the fact that all software, being finite, limits the functionality and other characteristics of the system (in extent and in detail) to a finite selection from an infinite set. The domain of operation has also infinite attributes, but the system can only be designed and validated, explicitly or implicitly, for satisfactory operation in some finite part of it. Sooner or later, excluded features, facilities and domain areas become bottlenecks or irritants in use. They need to be extended to fill the gap. The system needs to be evolved to satisfactorily support new situations and circumstances.

Though they have different causes and represent, in many ways, different circumstances, the steps to be taken so as to take cognisance of the sixth law do not, in principle, differ radically from those listed for the first law (section 4). There are, however, some differences due to the fact that the former is, primarily, concerned with functional and behavioural change whereas the latter leads, in general, directly to additions to the existing system and therefore to its growth. In practice, it may be difficult or inappropriate to associate[3] a given activity with either law.
Nevertheless, since the two laws are due to different phenomena they also are likely to lead different, though overlapping, recommendations. These are therefore listed together in section 4.

In general, the cleaner the architecture and structure of the system to be evolved the more likely is it that additions may be cleanly added with firewalls that permit only the exchange of appropriate information between old and new parts of the system. There must, however, be some penetration from the additions to the existing system. This will, in particular, be so when one considers the continued evolution of systems that were not, in the first instance, designed or structured for dynamic growth by the addition of new *components*. Sadly, the same remarks, limitations and consequent precautions, are likely to apply when one is dealing with systems that are component based or that made widespread use of COTS [Hybertson *et al.* 1997, Lehman and Ramil 1998] from the start. Future growth is inevitable and a sound architectural and structural base will reduce the effort that will inevitably be required when extending or re-engineering the system. Careful attention must be paid at all times, to the points made in section 4.

a. it appears, in general, to be a sound strategy to alternate releases between those concentrating primarily on defect removal, complexity reduction and minor enhancements and those that implement performance improvement, provide functional extension or add new function [e.g., Woodside 1979]. Incremental growth and other models provide indicators to help determine if and when this is appropriate.

10. Seventh Law: Declining Quality

Unless rigorously adapted to take into account changes in the operational environment, the quality of an E-type systems will appear to decline as it is evolved

This law follows directly from the first and sixth laws. As discussed in previous sections, an *E*-type system must undergo changes and additions to adapt and extend it if it is to remain satisfactory in use in a changing operational domain. Functionality must be changed and extended. To achieve this, new blocks of code, modules, components, subsystems are attached, new interactions and interfaces are created, sometimes one on top of the other. If such changes are not made, embedded assumptions become falsified, mismatch with the operational domains increases. As already observed the complexity of the system in terms of the interactions between its parts, and the potential for such interaction, all increase. Performance is likely to decline and the potential for defects will increase as earlier embedded assumptions are inadvertently violated and the potential for undesired interactions created. From the point of view of performance, behaviour and future system evolution, adaptation and growth effort increase. Growing complexity and mismatch with operational domains, declining performance, increasing numbers of defects, increasing difficulty of adaptation and growth will all cause stakeholder satisfaction to decline unless the work to maintain quality in every respect is undertaken.

There are many approaches to defining software quality [e.g., Boehm *et al.* 1978; Sommerville 2001]. The above lists causes of decline in terms of some of the more obvious sources and causes. There are many others. It is not proposed to discuss here possible viewpoints, the impact of circumstances or more formal definitions. The bottom line is that quality is a function of many factors whose relative significance will vary with circumstances. Users in the field will think of it in such terms as performance, reliability, functionality, adaptability. A CEO, at the other extreme, will be concerned with the contribution the system is making to corporate profitability, its market share, the corporate image, resources

[3] For a recent discussion on the topic of classification of activity into types see [Chapin et al. 2001].

required to support it, the support provided to the organisation in pursuing its business and so on.

In summary we observe that the underlying cause of the seventh law, the decline of software quality with age, appears to relate to a growth in complexity which must be associated with ageing. It follows that in addition to undertaking activity from time to time to reduce complexity, practices in architecture, design and implementation that reduce complexity or limit its growth should be pursued:

a. identify aspects of quality being of concern in relation to the business or task being addressed
b. quantify the aspects identified in (a) so that they become adequately controllable. Subject to being observed and measured in a consistent way, associated measures of quality can be defined for a system, project or organisation. Their value, preferably normalised, may then be tracked over releases or units of time and analysed to determine whether levels and trends are as required or desired. One may, for example, monitor the number of user generated defect reports per release. A fitted trend line (or other model) can then indicate whether the rate is increasing, declining or remaining steady. One may also observe oscillatory behaviour and test this to determine whether sequences are regular, randomly distributed or correlated to internal or external events. Time series modelling may be applicable to extract and encapsulate serial correlation [e.g., Humphrey and Singpurwalla 1991]. One may also seek relationships with other process and product measures such as the size of or the number of fixes in previous releases, subsystem or module size, testing effort and so on. When abundant metric data is available, and the process is sufficiently mature, models such as Bayesian nets may be useful to predict defect rates [Fenton and Neil 1999]. The above examples all relate to defect related aspects of quality. Other measures may be defined, collected and analysed in an analogous manner.
c. design changes and additions to the system in accordance with established principles such as information hiding, structured programming, elimination of pointers and GOTOs, and so on, to limit unwanted interactions between code sections and control those that are essential
d. devote some portion of evolution resources to complexity reduction of all sorts, restructuring and the removal of *dead* system elements, unnecessary replications, etc. Though primarily *antiregressive* [Lehman 1974], without immediate revenue benefit, such activities help ensure future changeability, potential for future reliable and cost effective evolution. Hence, in the long run, they contribute to profitability
e. train personnel to seek to capture and record assumptions, whether explicit or implicit, at all stages of the process in standard form and in a structure that will facilitate their being reviewed
f. review relevant portions of the assumption set at all stages of the evolution process to avoid design or implementation action that invalidates even one of them. Methods and tools to capture, store, retrieve and review them and their realisation, a non-trivial action, must be developed
g. monitor appropriate attributes to identify or predict a need for cleanup, restructuring or replacement of parts [e.g., Hops and Sherif 1995] or the whole software.

As already indicated, the definition, measurement, modelling and monitoring of software quality related characteristics is very dependent on application, organisation, product and process characteristics and goals. Interested readers that seek details of the various aspects of quality monitoring, modelling and control beyond those discussed in the present paper, are referred to the literature in this field [e.g., Boehm *et al.* 1978; Sommerville 2001].

11. Eighth Law: Feedback System

E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

The behaviour of complex feedback systems is not and cannot, in general, be described *directly* in terms of the local behaviour of its forward path activities and mechanisms. Feedback will constrain the ways that the process constituents interact with one another and will modify their individual, local, and collective, global, behaviour. According to the eighth law the software process is such a system. This observation must, therefore, be expected to apply. Thus, the contribution of any activity to the global process may be quite different from that suggested by its open loop characteristics. If the feedback nature of the software process is not taken into account when predicting its behaviour, unexpected, even counter-intuitive, results must be expected both locally and globally.

Consider, for example, the growth and stabilisation processes described by the first and third laws (sects. 4 & 6). Positive feedback conveys the desire for functional extension leading to pressure for *growth* and a need for continuing *adaptation* to exogenous changes. If the resultant pressure is excessive it may lead to instability. Management, exercising its responsibility to manage change and the rate of change will, in response to information received about progress, system quality and so on, induce *negative* feedback, in the form of directives and controls to limit change, contain its side effects and drive it in the desired direction. Stabilisation results. The FEAST and earlier studies have provided behavioural evidence to support this analysis and the eighth law.

The positive and negative feedback loops and control mechanisms of the global *E*-type process involve activities in the many domains, organisational, marketing, business, usage and so on, within which the process is embedded and evolution is pursued. It develops a dynamics that drives and constrains it. Many of the characteristics of this dynamics are rooted in and will be inherited from its history and the wider, global, domains. As a result there are significant limitations to the control that management can achieve in the process. The basic message of the eighth law is, therefore, that in the long term managers are not free to adopt any action considered appropriate from some specific business or other local point of view. Reasonable decision can, generally, be locally implemented. The long-term, global, consequences that follow, may not be what was intended or anticipated.

It follows that fully effective planning and management requires that one takes into account the dynamic characteristics of the process; the limitations and constraints it imposes, as outlined above and elsewhere [FEAST 2001]. To achieve this requires models that reflect the dynamic forces and behaviour. FEAST and other sources [e.g., JSS 1999; ProSim 2000] have made progress in such modelling but more, much of it interdisciplinary, is required to achieve a systematic, controlled and usable discipline for the design and management of global software processes. From the FEAST work it appears that feedback loops involving personnel outside the direct technical process may have a major impact on the process dynamics and, therefore, on the behaviour of the software evolution process. The interactions of, for example, maintenance, planning, user support, marketing and corporate personnel needs at least as much thought and planning as do technical software engineering and other low level issues and activities.

These observations lead to the following recommendations:

a. determine the organisational structures and domains within which the technical software development process operates including information flow, work flow and management control, both forward and feedback and monitor changes
b. in particular seek to identify the many informal communication links that are not a part of the formal management structure but play a continuing role in driving and directing the system evolution trajectory, and seek to establish their impact
c. model the global structure using, for example, system dynamics approaches [Forrester 1961], calibrate and apply sensitivity analysis to determine the influence and relative importance of the paths and controls
d. in planning and managing further work, use the models as simulators to help determine the implications of the influences that are implied by the analysis [e.g., Woodside 1979; Wernick

and Lehman 1998; Chatters *et al.* 1999; Kahen *et al.* 2000]

e. in assessing process effectiveness, use the models as outlined in (c) above to guide to identify interactions, improve planning and control strategies, evaluate alternatives and focus process changes on those activities likely to prove the most beneficial in terms of the organisational goals.

12. The FEAST Hypothesis[4]

To achieve major process improvement of E-type processes other than the most primitive, their global dynamics must be taken into account.

The FEAST hypothesis extends the eighth law by drawing explicit attention to the fact that one must take the feedback system properties of the complex global software process into account when seeking effective *process improvement*. The description of the process as *complex* is an understatement. It is a multi-level, multi-loop, multi-agent system. The loops may not even be fixed and need not be hierarchically structured. The implied level of complexity is compounded by the fact that the feedback mechanisms involve humans whose actions cannot be predicted with certainty [Lehman 1976]. Thus analysis of the global process, prediction of its behaviour and determination of the impact of feedback, are clearly not straightforward. One approach to such investigation uses simulation models [Forrester 1961; Vensim 1995]. FEAST/1 has made some progress in this regard and FEAST/2 is continuing this line of work [FEAST 2001]. A number of system dynamics models [Forrester 1961] using the Vensim tool [Vensim 1995] are now being calibrated and investigated [e.g., Kahen *et al.* 2000]. Many of their variables reflect organisational characteristics and it may be possible to derive generic versions. Modelling and analysis *methods* are already emerging [Ramil *et al.* 2000; Lehman *et al.* 2001]. Available evidence indicates the validity of the hypothesis. Much effort must, however, still be applied if full understanding of the role of feedback in software development and maintenance is to be achieved and fully exploited. In any event the recommendations made in the previous section may be extended as follows:

a. when seeking disciplined process improvement, use models as outlined in section 11 to guide the analysis of the global process, investigation of potential changes and evaluation of alternatives, focusing implementation on those changes likely to prove the most beneficial in terms of the organisational goals.

13. The Principle of Software Uncertainty

The real world outcome of any E-type software execution is inherently uncertain with the precise area of uncertainty also not knowable

This principle was first formulated in the late 80s [Lehman 1989, 1990]. It asserts that the outcome of the execution of an *E*-type program is not absolutely predictable. The likelihood that execution will not satisfy the stakeholders may be small but a *guarantee* of *satisfactory* results (from the point of view of the stakeholders, and of the operational domain) can never be given no matter how impeccable previous operation has been or whether the program *satisfies* a formal specification. This statement may sound alarmist or trivial (after all there can always be unpredictable hardware failure) but that is not the issue. By accepting the statement and taking appropriate steps even a small likelihood of unsatisfactory results may be further reduced.

There are several sources of software uncertainty [Lehman 1989, 1990]. The most immediate and one that can be at least partially addressed in process design and management (sect. 3, 4, 9, 10), relates to the assumptions reflected in every *E*-type program. Some will have been taken consciously and deliberately, for example to limit the geographical range of the operational domain to a specific region or to limit the scope of an air traffic control system. Others may be unconscious, for example ignoring the gravitational pull of the moon [CERN 1998] in setting up control software for a particle accelerator. Others may follow from implementation decisions taken without sufficient foresight such as adopting a two-digit representation for years in dates. These examples illustrate circumstances where errors can eventually arise when changes in the user or machine world, or in associated systems,

[4] One of a number of alternative statements given since its first formulation.

invalidates assumptions and their reflection in code or documentation.

As indicated in section 3 a real world domain has an uncountable number of properties. Once any part of that real world is excluded from the system specification or its implementation the number of assumptions also becomes infinite. Any one of this infinite set can become invalid, for example, by extension of the operational domain, by changes to the problem being solved or to the activity that the system implements or supports or by changes in the system domain under and with which the program operates. Uncertainty is therefore intrinsic since an invalid assumption can lead to behavioural change in the program. Awareness of that uncertainty can, however, reduce the threat of error or failure if it leads to systematic search for and early detection of invalidity through regular checking of the assumption set. The better the records of assumptions, the simpler they are to review and the greater the frequency with which they are reviewed the smaller the threat. Hence the recommendation in earlier sections to incorporate conscious capture, recording and review of assumptions of all types into the software and documentation processes.

The discussion on software uncertainty has focussed on *assumptions*. There are also other sources of uncertainty in system behaviour on execution but the likelihood of their contributing to system failure is small in relation to that stemming from invalid assumptions embedded in the code or documentation. They are, therefore, not further considered here.

As also implied in earlier recommendations, it follows that:

a. when developing a computer application and associated systems, estimate and document the likelihood of change in the various areas of the application domains and their spread through the system to simplify subsequent detection of assumptions that may have become invalid as a result of changes
b. seek to capture by all means, assumptions made in the course of program development or change
c. store the appropriate information in a structured form, related possibly to the likelihood of change as in (a), to facilitate to detect any that have become invalid in periodic review
d. assess the likelihood or expectation of change in the various categories of catalogued assumptions, and as reflected in the database structure to facilitate such review
e. review the assumptions database by categories as identified in (c), and as reflected in the database structure, at intervals guided by the expectation or likelihood of change or as triggered by events
f. develop and provide methods and tools to facilitate all of the above
g. when possible, separate validation and implementation teams to improve questioning and control of assumptions
h. provide for ready access by the evolution teams to all appropriate domain specialists with in-depth knowledge and understanding of the application domain.

Finally, and as already noted, the uncertainty principle is a consequence of the unboundedness of the operational and application domains of *E*-type systems and the fact that the totality of *known* assumptions embedded in it must be finite. However much understanding is achieved, however faithfully and completely the recommendations listed in the previous sections are followed, the results of execution of an *E*-type system must always be *uncertain*. There is no escape. Adherence to the recommendations will, however, ensure that unexpected behaviour and surprise failures can be reduced, if not completely avoided. In view of the increasing penetration of computers into all facets of human activity, organisational and individual, often in life, societal or economically critical applications, any reduction in the likelihood of failure is important.

14. Conclusions

It is appreciated that some of the above recommendations may be difficult and/or costly to implement but the potential long-term benefit of their implementation in terms of productivity and predictability, process effectiveness, and system, maintainability for whatever reason and by whatever means, makes their pursuit worthwhile. The selection and application of individual recommendations and their combination becomes a matter of management judgement in terms of their relative efficacy, cost-benefit evaluation and business objectives.

Determination of a product evolution strategy is a management responsibility that must take into account many technical and business related factors. Circumstances may, for example, arise where business or technical considerations suggest a release policy in the interests of the business as a whole that may have undesirable consequences for the system whose evolution is being planned. Global and local, *black-box* and *white-box* process models [FEAST 2001; Ramil *et al.* 2000] reflect, amongst others, the role and mutual impact on one another of the evolving system, the organisation, the system evolution and usage domains and the actors and activities in all these domains. Their systematic use, together with full understanding of the technical alternatives and their likely consequence in the short, medium and long term, will facilitate a well founded decision that optimises the overall organisational benefit, reducing risk and increasing the long term aggregate benefit.

Lengthy though it is, this paper gives at best an overview of the topic. The reader is advised not to apply the recommendations blindly. To appreciate, and apply them fully, requires understanding of the human and technical, usage and organisational, backgrounds that underlie the observations from which the conclusions were derived. For real progress, some understanding of the phenomenology is necessary. References have been provided and the reader is encouraged to explore these and to seek to understand the models and the reasoning that underlies them. Above all, note that this paper is based primarily on an on-going investigation by a single group. For further advances, many more people must become involved in the search for ever more effective approaches to software process management and, more generally, to the development and evolution of computerised systems and the software that is crucial to their satisfactory and safe performance. Moreover, many of the conclusions relate directly to the behaviour of people (implementers, managers and all stakeholders), and organisations, such studies demand interdisciplinary collaboration.

It is an open question, to which degree an evolution process planned and managed with full awareness of the laws will or will not ultimately conform to the behaviours implied by them. The present authors, have reason to believe that such awareness has not been part of the processes studied by the present group or those studied in earlier work. Whether the full or partial application of the recommendations in this paper will significantly affect the evolutionary behaviour will also need to be addressed.

Finally, it must be stressed that the evolution studies, source of the recommendations in this paper, has been based on the well-tested scientific method. The real world has been observed, patterns of behaviour identified, measured and quantified, observations modelled, hypotheses formulated, support sought and theories developed or planned. This process must, of course, be pursued iteratively. This would apply to any further study of the topic. The addition of a theoretical framework is now being planned [Lehman and Ramil 2000; Lehman 2000b]. Eventually such a theory might be broadened to comprise a theory of the software process or generalised to address a wider group of business processes. It is hoped that this effort will provide significant benefit to a world and a society relying ever more on computers and, therefore, on the software that provides their functionality.

Acknowledgements

Sincere thanks are due to our colleagues at Imperial College, Dr Goel Kahen and Ms Siew Lim, for their continuing support. Equally to Professors Perry and Turski, to our collaborators at BT Labs, DERA-MOD, ICL, Logica and Matra-BAe and to their respective staffs who have provided the data, process insight and constructive criticism. Thanks are also due to the anonymous referees for their comments. Finally to the UK Engineering and Physical Science Research Council and its reviewers for funding provided under grant numbers GR/K86008 (FEAST/1), GR/M44101 (FEAST/2), GR/L07437, GR/L96561 and GR/N02412 (SVFs).

References[5]

[5] Papers identified with a "*" have been reprinted in [Lehman and Belady 1985].

Baumol W.J. (1967), "Macro-Economics of Unbalanced Growth - The Anatomy of Urban Cities," *American Economics Review,* June, pp. 415-426.

*Belady L.A. and M.M. Lehman (1972), "An Introduction to Program Growth Dynamics," In *Statistical Computer Performance Evaluation*, W. Freiburger (ed.), Academic Press, NY, pp. 503-511.

Boehm B. (1981), *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ.

Boehm B. (2000), "Value-Based Feedback in Software/IT Systems," Joint Invited Keynote Presentation, *ProSim 2000* and *FEAST 2000 Workshops*, 12 July, Imperial College, London, UK, available from http://www.doc.ic.ac.uk/~mml/f2000/program#presentations

Boehm B., J.R. Brown, J.R. Kaspar, M. Lipow, C.J. MacCleod and M.J. Merritt (1978), C*haracteristics of Software Quality*, North Holland.

Bohner S.A. and R.S. Arnold, Eds. (1996), *Software Change Impact Analysis*, IEEE Computer Society Press, Los Alamitos, CA, 376 pps.

Box G. and A. Luceño (1997), *Statistical Control by Monitoring and Feedback Adjustment*, Wiley, 327 pps.

CERN (1998), "The Earth breathes on LEP and LHC", CERN Bulletin 09/98; 23 February 1998, http://bulletin.cern.ch/9809/art1/Text_E.html

Chapin N., J.E. Hale, K.M. Khan, J.F. Ramil and W.G. Tan (2001), "Types of Software Evolution and Software Maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, 13, 1, January-February, pp. 1-30.

Chatters B.W. *et al.* (1999), "Modelling a Software Evolution Process," In *Proceedings of ProSim'99, Software Process Modelling and Simulation Workshop*, Silver Falls, OR28□30 June,. A revised version as "Modelling a Long Term Software Evolution Process, "*Journal of Software Process - Improvement and Practice*, 5, 2/3, July 2000, pp. 95-102.

Cook S., H. Ji and R. Harrison (2000), "Software Evolution and Software Evolvability," working paper, University of Reading, August.

Cusumano M.A. and R.W. Selby (1995), *Microsoft Secrets*, The Free Press, NY, 512 pps.

FEAST (2001), Feedback, Evolution And Software Technology, Projects Web Site, Department of Computing, Imperial College, London, UK, http://www.doc.ic.ac.uk/~mml/feast/

Fenton N.E. and M. Neil (1999), *A Critique of Software Defect Prediction Models*, IEEE Transaction on Software Engineering, 25, 3.

FFSE (2001), *Intl. Session on Formal Foundations of Software Evolution*, FFSE 2000, Lisbon, 13 March, http://prog.vub.ac.be/poolresearch/FFSE/

Forrester J.W. (1961), *Industrial Dynamics*, MIT Press, Cambridge, Mass.

Fowler M. (1999), *Refactoring: Improving the Design of Code*, Addison-Wesley, New York.

GCSE (2000), *Int. Symp. on Generative and Component Based Software Engineering*, 9–12 October 2000, Erfurt, Germany, http://www.netobjectdays.org/node00/en/Authors/cfp-gcse.html

Gilb T. (1981), "Evolutionary Development," *ACM Software Engineering Notes*, April.

Godfrey M.W. and Q. Tu (2000), "Evolution in Open Source Software: A Case Study," In *Proceedings of the International Conference on Software Maintenance*, ICSM 2000, 11-14 October, San Jose, CA, pp. 131-142.

Hops J.M. and J.S. Sherif (1995), "Development and Application of Composite Complexity Models and a Relative Complexity Metric in a Software Maintenance Environment," *Journal of Systems and Software*, 31, 2, November, pp 157-169.

Hsi I. and C. Potts (2000), "Studying the Evolution and Enhancement of Software Features," In *Proceedings of the International Conference on Software Maintenance*, ICSM 2000, 11-14 October, San Jose, CA, pp. 143-151.

Humphrey W.S. and N.D. Singpurwalla (1991), "Predicting (Individual) Software Productivity," *IEEE Transactions on Software Engineering*, 17, 2, February, pp. 196-207.

Hybertson, D.W., D.T. Anh and W.M. Thomas (1997), "Maintenance of COTS-intensive Software Systems," *Journal of Software Maintenance: Research and Practice*, 9, pp. 203-216.

ISPSE (2000), *Pre-prints of the Intl. Symposium on the Principles of Software Evolution*, Kanazawa, Japan, 1-2 November, 2000

IWPSE (1998), *Proceedings of the Intl. Workshop on the Principles of Software Evolution*, co-located with the 20th Intl. Conference on Software Engineering, Kyoto, 20 - 21 April

IWPSE (2001), *Intl. Workshop on the Principles of Software Evolution*, Call for Participation, Vienna, 10 -12 Sept. 2001

JSS (1999), *The Journal of Systems and Software,* Special Issue on Software Process Simulation Modelling, 46, 2/3.

Kahen G., M.M. Lehman, J.F. Ramil and P.D. Wernick (2000), "Dynamic Modelling in the Investigation of Policies for *E*-type Software Evolution," ProSim 2000, Imperial College, London UK, 12-14 July, a revised version to appear in *Journal of Systems and Software*, 2001

Kemerer C.F. and S. Slaughter (1999), "An Empirical Approach to Studying Software Evolution," *IEEE Transaction on Software Engineering*, 25, 4, July/August, pp. 493-509.

Kitchenham B. (1982), "System Evolution Dynamics of VME/B," ICL Technical Journal, May, pp. 42-57.

Lawrence M.J. (1982), "An Examination of Evolution Dynamics," In *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, Japan, 13-16 September , IEEE Computer Society, pp. 188-196.

*Lehman M.M. (1969), "The Programming Process," IBM Research Report RC 2722, IBM Research Centre, Yorktown Heights, NY, September.

*Lehman M.M. (1974), "Programs, Cities, Students, Limits to Growth?," Inaugural Lecture, in *Imperial College of Science and Technology Inaugural Lecture Series*, 9, 1970, 1974, pp. 211-229. Also in D. Gries, Ed. (1978), *Programming Methodology*, Springer-Verlag, pp. 42-62.

Lehman M.M. (1976), "Human Thought and Action as an Ingredient of System Behaviour," Imperial College of Science Technology, CCD Research Report 76/12, July 1976, Also in Duncan R. and M. Weston-Smith, Eds. (1977) *Encyclopaedia of Ignorance*, Pergamon Press, Oxford, pp. 397-354.

*Lehman M.M. (1978), "Laws of Program Evolution—Rules and Tools for Programming Management," In *Proceedings Infotech State of the Art Conference, Why Software Projects Fail?*, April, pp. 11/1-11/25.

Lehman M.M. (1979), "The Environment of Design Methodology," Keynote Address, In *Proceedings Symposium on Formal Design Methodology*, Cox TA (ed.). Cambridge, UK, 9-12 April 1979, pp. 17-38, published by STL Ltd, Harlow, Essex, UK, 1980.

*Lehman M.M. (1980a), "On Understanding Laws, Evolution, and Conservation in the Large Program Life Cycle," *Journal of Systems and Software*, 1, 3, pp. 213-221.

*Lehman M.M. (1980b), "Programs, Life Cycles and Laws of Software Evolution," *In Proceedings of IEEE Special Issue on Software Engineering*, September, pp 1060 - 1076. With more detail as "Programs, Programming and the Software Life-Cycle," In *System Design, Infotech State of the Art*, Rep, Se 6, No 9, Pergamon Infotech Ltd, Maidenhead, 1981, pp 263-291.

Lehman M.M. (1989), "Uncertainty in Computer Application and its Control through the Engineering of Software," Journal of Software Maintenance: Research and Practice, 1, 1 September, pp. 3-27.

Lehman M.M. (1990), "Uncertainty in Computer Application," Technical Letter, *Communications of the ACM*, 33, 5, 584.

Lehman M.M. (1994), "Feedback in the Software Evolution Process," Keynote Address, *CSR Eleventh Annual Workshop on Software Evolution: Models and Metrics*, Dublin, Ireland, 7-9 Sept. and In *Information and Software Technology, special issue on Software Maintenance*, 38, 11, 1996, Elsevier, pp. 681-686.

Lehman M.M. (1997), "Laws of Software Evolution Revisited," *In Proceedings of*

EWSPT'96, Nancy, LNCS 1149, Springer-Verlag, pp. 108124.

Lehman M.M. (1998a), "The Future of Software - Managing Evolution," Invited Contribution, *IEEE Software,* 15, 1, January-February, pp. 40-44.

Lehman M.M. (1998b), "FEAST/2: Case for Support," Department of Computing, Imperial College, London, UK, July. Available from links at the FEAST project web site http://www.doc.ic.ac.uk/~mml/feast

Lehman M.M. (2000a), "Rules and Tools for Software Evolution Planning and Management," position paper, *FEAST 2000 Workshop*, Imperial College, London, UK, 10-12 July available from links at http://wwwdoc.ic.ac.uk/~mml/f2000

Lehman M.M. (2000b), "TheSE - An Approach to a Theory of Software Evolution," Project Proposal, Department of Computing, Imperial College, London, UK, December.

Lehman M.M. and L.A. Belady, Eds. (1985), *Program Evolution—Processes of Software Change*, Academic Press, London.

Lehman M.M., D.E. Perry, and J.F. Ramil (1998), "On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution," in *Proceedings of the Fifth International Metrics Symposium*, Metrics '98, Bethesda, Maryland, 20-21 November.

Lehman M.M. and J.F. Ramil (1998), "Implications of Laws of Software Evolution on Continuing Successful Use of COTS Software," Technical Report 98/8, Department of Computing, Imperial College, London, UK, incl. panel pos. statement, ICSM '98, Washington DC, 16-18 November. A revised version as "Software Evolution in the Age of Component Based Software Engineering," *IEE Proceedings - Software, Special Issue on Component Based Software Engineering,* 147, 6, pp. 249 - 255, December 2000

Lehman M.M. and J.F. Ramil (2000), "Towards a Theory of Software Evolution - And Its Practical Impact," Invited Lecture*, Pre-prints of the International Symposium on Principles of Software Evolution*, ISPSE 2000, Kanazawa, Japan, pp. 1 - 9, 1-2 November

Lehman M.M., J.F. Ramil and P.D. Wernick (2000), "Metrics-Based Process Modelling With Illustrations From The FEAST/1 Project," as chapter 10 in Bustard D, Kawalek P and Norris M (eds.). *Systems Modelling for Business Process Improvement*, Artech House, April.

Lehman M.M., J.F. Ramil and G. Kahen (2001), "Experiences with Behavioural Process Modelling in FEAST, and some of its Practical Implications", *Proceedings of the 8th European Workshop on Software Process Technology,* EWSPT-8, 19-21 June, Haus Bommerholz, Witten, Dortmund, Germany, LNCS, Springer Verlag, Berlin

Lehman M.M., and V. Stenning (1996), "FEAST/1: Case for Support," Project Proposal, Department of Computing, Imperial College, London, UK, March. Available from links at the FEAST project web site http://www.doc.ic.ac.uk/~mml/feast

Pfleeger S.L. (1998a), *Software Engineering - Theory and Practice*, Prentice-Hall.

Pfleeger S.L. (1998b), "The Nature of System Change," *IEEE Software*, 15, 3; May-June, pp. 87-90.

ProSim (2000), *Workshop on Software Process Simulation and Modelling*, 12-14 July 2000, Imperial College, London, UK, http://www.prosim.org

Rajlich V.T. and K.H. Bennett (2000), "A Staged Model for the Software Life Cycle," *Computer*, July, pp. 66-71.

Ramil J.F. and M.M. Lehman (2000), "Metrics of Software Evolution as Effort Predictors - A Case Study," In *Proceedings International Conference on Software Maintenance*, 11-14 October, San Jose, CA, pp. 163-172.

Ramil J.F., M.M. Lehman and G. Kahen (2000), "The FEAST Approach to Quantitative Process Modelling of Software Evolution Processes," In *Proceedings PROFES'2000 2nd International Conference on Product Focused Software Process Impovement*, Oulu, Finland, 20-22 June, LNCS 1840 Springer, Berlin, pp. 311-325.

Shepperd M. (2000), "Dynamic Models of Maintenance Behaviour," Workshop on *Empirical Studies of Software Maintenance*, WESS 2000, 14 October, San Jose CA, available from http://members.aol.com/_ht_a/geshome/wess2000/metricsandmodels.htm

Sommerville I. (2001), *Software Engineering*, Sixth Edition, Addison-Wesley & Pearson Education Limited, Harlow, UK. Chapter 24.

Turski W.M. (1996), "Reference Model for Smooth Growth of Software Systems," *IEEE Transactions on Software Engineering*, 22, 8, August.

Turski W.M. and T. Maibaum (1987), *The Specification of Computer Programs*, Addison Wesley, London.

Vensim (1995), *Vensim Reference Manual* (1995), Ver. 1.62, Ventana Systems Inc., Belmont, MA.

Wall L., T. Christiansen and R.L. Schwartz (1996), *Programming Perl*, O'Reilly & Associates, Sebastopol, CA, 645 pps.

Wernick P. and M.M. Lehman (1999), "Software Process Dynamic Modelling for FEAST/1," *Journal of Systems and Software*, 46, 193-201.

*Woodside C.M. (1980), "A Mathematical Model for the Evolution of Software," *Journal of Systems and Software*, 1, 4, October, pp. 337-345.