Rules and Tools for Software Evolution Planning and Management

Meir M Lehman
Department of Computing
Imperial College
London SW7 2BZ
tel: +44 (0) 20 7594 8214
fax: +44 (0) 20 7594 8215
mml@doc.ic.ac.uk
http://www-dse.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, they have gradually become recognised as providing useful inputs to understanding of the software process and have found their place in a number of software engineering curricula. Now eight in number, they have been supplemented by a Software Uncertainty Principle and a FEAST Hypothesis.

Based on all these and on the results of the recent FEAST/1 and current FEAST/2 research projects, this paper develops and presents some fifty rules for application in software system process planning and management and indicates tools available or to be developed to support their application. The listing is structured according to the laws that encapsulate the observed phenomena and that lead to the recommended procedure. Each sub-list is preceded by a textual discussion providing at least some of the justification for the recommended procedure. The text is fully referenced. This directs the interested reader to the literature that records observed behaviours, interpretations, models and metrics obtained from some three of the industrially evolved systems studied, and from which the recommendations were derived.

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

1 Introduction

This paper focuses on the practical implications of the results of the FEAST/1 and earlier studies, their observations, models derived therefrom, their interpretation and on conclusions reached by the FEAST/1 group in extensive discussions amongst themselves and with collaborator personnel. It also points to general principles of software evolution and their methodological implications. To support application of the latter, proposals for project planning and management tools are also included. The recommendations must, clearly, be applied intelligently, not blindly. This requires insight into the relevant general observations, the data, the models and the interpretations on which the conclusions are based. This relatively brief paper cannot provide such detail and is restricted to a general review of the relevant phenomenology. Where appropriate, references to additional sources and information will be given.

Observations and rules relevant to software system evolution planning and management [leh98d] were first identified during studies of the evolution of OS/360-70 and other systems between 1968 [leh69] and 1985 [leh78,85,woo79]. More recently, with the active collaboration of ICL, Logica and Matra BAe Dynamics, the FEAST/1 project [leh96b] (1996-1998) 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 two real time Lucent Technologies systems also became available for analysis during this time. Broadly speaking, the long-term evolutionary behaviour of these current release based systems 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. Such similarity in the long-term evolutionary behaviour of widely different systems over a period of rapidly evolving technology suggests that the observed behaviour the rules derived therefrom is not primarily due to the technologies employed. In the main these impact local behaviour. Global behaviour, as observed from outside the process, appears to be determined, at least in part, by other forces. Thus it should be possible to extend the conclusions to yield results of wide validity in the field of software, and ultimately more general, evolution.

The FEAST/2 project (1999 - 2001) [leh98b] expects, *inter alia*, to provide a firmer empirical base for the conclusions briefly summarised here. It also seeks to refine earlier results by identifying the sources of evolutionary trends and patterns and their implications on evolutionary behaviour. Such trends are unlikely to be limited to specific implementation, application areas or environments but to be more widely relevant.

As a practical guide, the paper follows the historical ordering and wording of the laws of software evolution and other principles [leh85,97]. Some overlap between sections is unavoidable if excessive cross-references are to be avoided. We trust that the paper is, nevertheless, useful and finds practical application.

The empirical evidence referred to in this paper is restricted to that obtained in the 70s studies as modified and extended during FEAST/1. Additional insight and information is being accumulated in FEAST/2 and will be made available in due course. The reader wishing for clarification, more detail or to see the data and analyses from which many of the conclusions and recommendations were derived can access the relevant literature via the FEAST web site [fea00].

2 Laws of Software Evolution

The eight laws of software evolution, formulated over the 70s and 80s [leh74,78,80], are listed in Table 1. The present listing incorporates minor modifications that reflect new insights gained during the FEAST/1 project [94,97,98c]. They emerged from a follow up of the 1969 study of the evolution of IBM OS/360 [leh69] as strengthened by the results of other evolution studies in the seventies. Additional support came from an ICL study in the eighties [kit82] but some criticism directed, primarily, at the inadequacy of the statistical support came from Lawrence [law82]. Over the years, the laws have gradually become recognised as providing useful inputs to understanding of the software process [pfl98] and have found their place in a number of software engineering curricula. Additional support for six of the eight laws has accumulated as the data obtained from its collaborators [leh98c,00a] was analysed in the FEAST/1 project [leh96b]. In the absence of relevant data, the remaining two laws were neither supported nor negated by the additional evidence acquired from the latest studies

No.	Brief Name	Law
I 1974	Continuing Change	*E*-type systems must be continually adapted else they become 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	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving *E*-type system tends to remain constant over product lifetime
V 1978	Conservation of Familiarity	In general, the incremental growth and long term growth rate of *E*-type systems tend 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 *E*-type systems will appear to be declining
VIII 1996	Feedback System (Recognised 1971, formulated 1996)	*E*-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

Table 1 - Current Statement of the Laws

In presenting some of the practical implications of the FEAST work to date, the structure of this paper follows the order in which the laws were formulated. It is, however, now recognised that since the laws are not independent they should not be linearly ordered. Thus the numbering of Table 1 has only historical significance. Its use in structuring this paper is a matter of convenience and has no other implication. The question of dependencies and relationships between the laws is currently the subject of further investigation and an introductory description of that investigation is now available [leh00b]. If successful it should put to rest the main criticisms to which the laws have been subjected over the years [law82]. These addressed the absence of precise definitions or statements of assumptions, their being based on data from a single, atypical, source (OS/360), the fact that OS/360 represented an outmoded software technology and the use of the term *laws* in relation to observations about phenomena directed, managed and reflecting human activity. This criticism was countered by the present author who noted that it was precisely such human involvement that justified use of the term, though the demand (when first formulated) for more wide spread evidence of their validity was fully justified. The term *law* was deliberately selected just because each 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.

Note that the laws apply, in the first instance, to *E*-type programs that is, for systems actively used and embedded in a real world domain [leh85]. Once such systems are operational, they are judged by the results they deliver. Their properties include loosely expressed expectations that, at least for the moment, stakeholders are *satisfied* with the system as is. In addition to functionality, factors such as *quality* (however defined), *behaviour* in execution, *performance*, ease of use, *changeability* and so on will be of concern. 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* [tur00]. A property not so included may be present or absent, deliberately, by oversight or at the programmer's whim.

The importance of *S*-type programs lies in their being, by definition, mathematical objects about which one may reason. Thus they can be verified, i.e., *proven* correct. 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. Once the specification has been fixed and verified, acceptance of the program is independent of any assumptions by the implementers; entirely determined by the specification. 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 an explicit or implicit assumption set can then be prepared and a new program derived, probably by modification of the old.

The role of *S*-type programs in the programming process follow from these properties. Now, the minds of individual programmers cannot be read and one cannot therefore, know what *assumptions* are made during the course of their work. Any such assumptions become a part of the program properties even though not a part of the original specification. As the operational domain or other circumstances change some assumptions will become invalid and affect program behaviour in unpredictable and/or unacceptable fashion. It is therefore important to prevent individuals or small groups from unconsciously incorporating assumptions into a program or its documentation. It is equally important to support the process of making and documenting conscious assumptions and to adapt the system as this become necessary.

The unconsidered injection of assumptions into both *S*- and *E*-type programs may be minimised by providing a *precise* and *complete* specification for each individually assigned task, that is, giving implementers only *S*-type assignments. With this approach, *S*-type programs constitute the essential *bricks* from which larger programs and complex systems are built. The main body of assumptions will then be explicit in the specification or implied by omissions from the specification. Thus the elemental parts of individual products can be assessed objectively by verification against their specification. However, the detection of implied assumptions is difficult, often happens unconsciously. Moreover, situation requiring the *de facto* introduction of new assumptions cannot be avoided. When this occurs, they must, as far as possible, be detected, captured, validated, formally approved and added to the specification. But some assumptions will slip through the net. 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[1].

Once integrated into a larger system, and when that system is operational, the *bricks* operate in the real world. In *that* context they display *E*-type characteristics. This is, of course, fine provided that records of changes to individual specifications record additions and changes to the assumption set that underlie them. The restriction of the laws to *E*-type systems, therefore, in no way decreases their practical significance. Both *S*- and *E*-type programs have a role to play in system development and evolution.

We now proceed to outline some of the practical implications of the laws and the tools that are suggested by them. Many of the items on the lists that follow will appear intuitively self-evident. What is new is the unifying conceptual framework on which they are based. This framework follows from observed behaviour, interpretation, inference and so on as discussed in greater detail in the FEAST literature [fea00]. Together these provide the basis for a comprehensive *theory of software evolution* and since evolution is intrinsic to all practical software, also to a *theory of the software process* such as that now being developed [leh00b].

---

[1] This is becoming more widely recognised and addressed, e.g., in the context of component-based software [gen00].

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

A full analysis of the meaning and implications of this classification requires more discussion than can be provided here but guidelines that follow from the preceding discussion are listed in this section, others under the headings that follow. It is appreciated that some of these recommendations may be difficult to implement but the potential long-term benefit of their implementation in terms of productivity, process effectiveness, with system predictability, maintainability, changeability makes their pursuit worthwhile. 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. The (long-term) goal should be to express specifications formally.
- c. 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.
- d. When *verifying* the completed specification a conscious effort must be made to identify and document all assumptions implied by individual or combinations of statements.
- e. 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.
- f. Tools to assist in the implementation of these recommendations and to support their systematic application should be developed and introduced into practice.

4 First Law: Continuing Change - *E-type systems must be continually adapted else they become progressively less satisfactory in use*

It has already been observed that every *E*-type system is a model of the application in its operational domain. Both the real world and every application have, potentially, an infinite number of attributes. Being part of the real world, the operational domain in which the system operates is initially undefined and is, therefore, also intrinsically unbounded. On the other hand, the software system and, for that matter, the entire application system, are essentially finite. Therefore, the process of abstraction and transformation to define and develop the application system and its software involves, *inter alia*, *assumptions* about what is to be included in the final program. This process of finitisation excludes perforce all other elements/attributes of the operational domain and the application. As a model of the real world, the system is incomplete [leh77,85,89,90]. 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, code, 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 complete and valid in the sense that its effect on system behaviour 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, system application expands in terms of numbers of users or details of usage and so on. Thus, a growing number of assumptions become invalid. This is likely to lead to less than acceptably satisfactory performance in some sense and hence to change requests. On top of that there will 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 commerce, industry and government, in fact all regular computer users.

There follows a partial listing of practical consequences of this unending need for change to every *E*-type system in continuing use to, *inter alia*, adapt it to changing operational domains.

- a. Comprehensive documentation must be created and updated to minimise the impact of growing complexity as changes are applied over the system lifetime, change upon change (see next section).
- 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. As the design of change proceeds, all aspects including, for example, the issue being addressed, the reasons why a particular implementation design/algorithm is being used, details of assumptions, explicit and implicit, adopted and so on must be recorded in a way that will facilitate regular subsequent review.
- d. The assumption set must be reviewed as an integral part of release planning and periodically to detect domain and other changes that conflict with the existing set, or violate constraints.
- e. The safe rate of change per release is constrained by the process dynamics. As the number, magnitude and orthogonality to system architecture of changes in a release increases, complexity and fault rate grow more than linearly. Successor releases focussing on fault fixing, performance enhancement and structural clean up will be necessary to maintain system viability. Models of, for example, patterns of incremental growth (see next bullet) and of numbers of changes per release over a sequence of releases, can provide an indication of limits to *safe* change rates. Another useful metric is the numbers of elements changed (i.e. *handled*) per release or over a given time period. Other change metrics have also been discussed [fea00].
- f. FEAST/1 and earlier *incremental growth models* (section 8), suggest that an excessive number of changes in a release has an adverse impact on release schedule, quality, and freedom of action for following releases. A more precise statement of the consequences remains to be determined.
- g. It appears, in general, to be a sound strategy to alternate releases between those concentrating primarily on fault fixes, complexity reduction and minor enhancements and those that implement performance improvement, provide functional extension or add new function [woo79]. Incremental growth and other models provide indicators to help determine if and when this is appropriate.
- h. Change validation must address the change itself, actual and potential interaction with the remainder of the system and impact on the remainder of the system.
- i. It is beneficial to determine the number of distinct additions and changes to *requirements* 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 fault rates or high functional volatility or where, to facilitate future change, extra care should be taken in change architecture 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*

One reason for complexity growth as a system evolves, the imposition of change upon change upon change, has been mentioned in the previous section. There are others. For example, the number of potential connections and interactions between elements (objects, modules, holons, sub-systems, 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. Moreover, with the passage of time, changes and additions that are made are likely to be ever more remote from the initial design concepts and architecture, so further increasing the inter-connectivity. Even if carefully controlled all these 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, tends to cause a growth in the need for user support and in costs. Such increases will, in general, tend to be accompanied by a decline in product quality and in the rate of evolution, however defined and measured, unless additional work is undertaken to compensate for this. FEAST/1 observations indicate directly that the average software growth rate measured in modules or their equivalent tends to decline as a function of the release sequence number as the system ages. The long term trend tends to follow an *inverse square* trajectory [tur96] with the *mae* (mean absolute error), that is the mean absolute difference, between the sizes of releases as predicted by the model and their actual size of order 6% (see section 6 and [fea00]).

Based on this law which reflects the observations, measurement, modelling, analysis and other supporting evidence obtained over the last thirty years, and most recently, in the FEAST/1 project, the following observations and guidelines may be identified:

- a. System complexity has many aspects. 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. Complexity control is an integral part of the development and maintenance responsibility. Such effort is largely *anti-regressive* [leh74]; *immediate* benefits are generally relatively small. Its long-term impact is, however, likely to be significant; may indeed, at some stage, make the difference between survival of the system and its replacement or demise. In planning release content for one or a series of releases the timing, degree and distribution of complexity control activity must be carefully considered.
- c. Determining the level of effort for *anti-regressive* activity such as complexity control in a release or sequence of releases and what effort is to be applied, presents a major paradox. If the level is reduced or even abandoned to free resources for *progressive* [bau67] 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 evolution rate will decline. In the absence of process improvement that is based on the principles examined in this paper, decline of evolution rate appears inevitable. Hence, one must evaluate alternatives and select the strategy most likely to help achieve corporate business goals or whatever else requires to be optimised.
- d. 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 [woo79].

6 Third Law: Self Regulation - *Global E-type system evolution processes are self regulating*

This law was first suggested by the growth patterns of OS/360 [leh74], confirmed by observations on three other systems in the 70's and most recently reconfirmed for the release-based systems studied in FEAST/1. The detailed growth *patterns* of the different systems differ but the *gross trends* are strikingly similar. In particular, inverse square models have yielded unexpectedly good fits to plots of system size S_i, (generally measured in modules or their equivalent) against release sequence number (rsn) i in all the release based systems studied under the FEAST projects [tur96, fea00]. The model takes the form $S_{i+1}=S_i+\hat{e}/S_i^2$, where $\hat{e}$ can be calculated, for example, as the mean of a sequence of e_i calculated from pairs of neighbouring size measures S_i and S_{i+1}. The predictive accuracy of the models, as measured by the *mae*, are of order 6%. This is a remarkable result considering that the size of individual releases is mainly determined by management focus on functional needs, since with the declining cost of storage, overall system growth is not, in general, consciously managed or constrained. An exception to that observation is illustrated by certain embedded systems where storage is limited physical considerations.

The FEAST projects have identified two exceptions to this surprisingly simple model. In the case of VME Kernel, the least square model can be fitted directly to the growth data as for the other systems studied. An improved fit is, however, obtained when the model is fitted separate segments spanning rsn 1 to 14 and rsn 15 to 29 respectively. The second exception is OS/360 where a *linear* model over rsn 1 to 19 gives a lower mae than an inverse square model. In that case, however, the system growth rate fluctuates wildly for a further 6 releases beyond rsn 19 before fission into VS1, VS2. In that region the inverse square growth model is totally inappropriate. Explanations of these exceptions have been proposed but confirmation and with it, determination of their wider implications, is now difficult to come by.

A common feature of the growth patterns of *all* the release based systems observed is a superimposed *ripple* [bel72,leh78,fea00]. These ripples are believed to reflect the action of stabilising mechanisms that yield the regulation referred too in the law. Feedback mechanisms that achieve such stabilisation have been proposed. Their identification in real industrial processes, exploration of the manner in which they achieve control such as process stabilisation and the behavioural implication on system evolution, was initiated in FEAST/1 and is being continued in FEAST/2 by means of system dynamics models [cha99]. Attempts to improve, even optimise, such mechanisms can follow.

The conclusion that feedback is a major source of the behaviour described by the third law is supported by the following reasoning. As for many business processes, one of the goals of outer loop mechanisms in the software process is *stability*[2] in technical, financial, organisational and sales terms. To achieve this goal technical, management, organisational, marketing, business, support and user processes and the humans who work in or influence or control them, apply negative (constraining) and positive (reinforcing) information controls guided by indicators of past and present performance, data for future direction and control. Their

[2] Stability, as used here, 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 bugs; senior management, a decline in costs and growth in sales.

actions are exemplified by, for example, progress and quality monitoring and control, checks, balances and control on resource usage. All these represent, primarily, feedback based mechanisms as described by the third law. The resultant information loops tend to be nested and intuition suggests that the outer ones will exert a major, possibly dominant, influence on the behaviour of the process as measured from outside the loop structure. Preliminary evidence from FEAST/1 supports this assertion [cha99]. Thus in describing global process behaviour (and, by implication, that of the system) as observed from the outside, the law is entirely compatible with the realities of the real world of business. It may be concluded that the mechanisms underlying the third law are strongly feedback related and the law is likely to be closely related to the eighth - Feedback - law (sects. 11 and A.11 [leh00b]).

In a complex, multi loop, system such as the software process, it is difficult to identify the sources of the feedback forces that influence individual behavioural characteristics [leh96a]. Thus, for example, stabilisation forces and mechanisms in software evolution planning and execution are not explicit, will often be unrecognised, may not be manageable. Many will arise from outside the process, from organisational, competitive or marketplace pressures, for example. Each may work for or against deviations from established levels, patterns and trajectories and lead to consequences that may well be counter-intuitive, particularly when decisions are taken in the absence of a global picture. Process changes may well have the intended local impact but a global consequence that is unexpected. In feedback systems, correct decisions can generally only be taken in the context of an *understanding* of the system as a whole. In the absence of appropriate global models provision must be made for the unexpected.

As discussed above, important process and product behavioural characteristics are probably determined to a significant degree by feedback mechanisms that are themselves not consciously controlled by management or otherwise. As a first step to their identification one should 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 self-stabilisation and to exploit them in future planning, management and process improvement the following steps should be helpful:

- a. Using measurement and modelling techniques as used, for example, in FEAST/1 [leh00a], determine typical patterns, trends, rates and rates of change of a number of projects within the organisation. To obtain meaningful results, systems that have had at least eight to ten releases are likely to be required.
- b. Establish *baselines*, that is, typical values for process rates such as growth, faults, 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, for reasons well understood, the former yields results that are more regular and interpretable. Initially however, and occasionally thereafter, results over real time and over release sequence number must be compared and appropriate conclusions drawn. Incremental values, that is the difference between values for successive time intervals should also be determined, as should numbers of people working with the system in various capacities, person days in categories such as specification, design, implementation, testing, integration, customer support and costs related to these activities. A third group of measures relates to quality factors. These can be expressed, for example, in pre-release and user reported faults, 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 mean of the incremental growth m_i of the system in going from release i to release $i+1$ and s the standard deviation of the incremental growth both over a series of some five or so releases or time intervals. The scenarios may, for example, be differentiated by an indicator $m+2s$ that identifies a release plan as *safe*, *risky* or *unsafe* 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 that a *safe* level for planned release content m_i is that it be less than or equal to m. If the condition is fulfilled growth at the desired rate may proceed.
  - 2. The desired release content is greater than m but less than $m+2s$. The release is *risky*. It could succeed in terms of achieved functional scope but serious delivery delays, 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 zero growth release is likely to be required. Note that $m+2s$ has long been identified as an alarm limit, for example, in statistical process control and monitoring [box97].

- 3. The desired release content is close to or greater than *m+2s*. A release with incremental growth of this magnitude is *unsafe*. 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 in which the main emphasis is on fault fixing and *anti-regressive* work such as the elimination of so called *dead code*, restructuring, documentation updating and so on.

- e. An appropriate *evolutionary development* strategy [gil81] 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.

7 Fourth Law: Conservation of Organisational Stability - *Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving E-type system tends to remain constant over product lifetime*

The observations on which this law is based date back to the late seventies. Further data gathered in FEAST/1 neither supports nor negates it. When first formulated, the feedback nature of the software process had already been identified [bel72,leh78,85]. Insight into the underlying process mechanisms and related phenomena supported the interpretation. Subsequent observations suggested, for example, that the average activity rate measured by the *change rate*, is *stationary*[3] but with changes of the mean and variance at a few two points during the observed life time of the released-based systems. The wider influence of feedback on the process was, however, not fully appreciated until recently. If further observations yield similar results and when more understanding is achieved, refinement of the law may be indicated. Those aspects of the law that relate to the role of feedback stabilisation and its implications are discussed below. Its management implications are not considered further in the present report..

8 Fifth Law: Conservation of Familiarity - *In general, the incremental growth and long term growth rate of E-type systems tend to decline*

With the exception of OS/360, the decline in incremental growth and growth rate has been observed in all the systems whose evolution has been 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 with age can be identified. Consider, for example, system maintenance over a series of releases. This requires a split of resources between fixing, enhancement and extension. For many reasons, as a system ages the need for fixes is likely to increase. If investment in system maintenance remains constant, this implies a drop in resources available for system growth and, hence, a possible source of a declining growth rate. Equally the budget allocation may be declining because it has, for example, been decided that it is no longer in the organisation’s interest to expand the system or because it is believed that increasing maintenance productivity, as personnel experience and system familiarity increases, permits the reduction of maintenance funding. In the case of VME Kernel, for example, it appears that over recent years a decrease in incremental growth has been accompanied by a *decrease* in the size of the development team, that is in resource. But the reason for this reduction is not yet known. In particular, it has not been possible to show that the decrease explains or even correlates to the declining growth. In general, the nature of the relationship between the system evolution rate and resources has not been systematically investigated, may well be counter-intuitive [bro75].[4]

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 study. That analysis has suggested that the

---

[3] Loosely defined, *stationarity* is a property of stochastic processes that display a constant average and variance [box97].

[4] FEAST/2 is now directly addressing the topic of cost estimation in the context of software evolution [ram00b]. The study should also advance understanding of the various drivers that, individually and jointly, influence growth and change rates.

most likely source of the declining incremental growth rates observed is, primarily, due to increasing complexity as the system ages. This 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. Changes and additions take longer to design and to implement, errors and the need for subsequent repair are more likely, comprehensive validation is more complex. These are some of the factors that may be causing the decline, identified by the fifth law, in the rate of, for example, system growth.

Though the law refers primarily to long term behaviour, regular short-term variations in incremental growth have also been observed. FEAST/2 is studying these. A related aspect of the investigation is the relationship between incremental growth which tends to reflect the addition of new functionality, and modification of existing software elements to reflect changes in the application, the domain or other parts of the system. In the VME and Lucent data, an apparent decrease in incremental growth appears to be accompanied by an increase in the number of elements changed per release and *vice versa* [leh98a]. This is probably due to the fact that as more resources are applied to clean up, repair and adaptation less are likely to be available for system extension. If confirmed, models derived from this relationship can provide additional planning aids.

In summary, both developers and users must be familiar with the system if they are to work on or use it effectively. 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. Common sense, therefore, dictates that the rate of change and growth of the system be slowed down as it ages and this trend has been observed in nearly all the data studied to date. The only exception has been the original OS/360 study where the growth trend appeared to remain constant to rsn 19. It is quite possible that the failure to slow down the growth rate led to the erratic behaviour of OS/360 following release rsn 20 and its ultimate break-up.

The fact that a reduction in growth rate as a system ages is likely to be beneficial has not, to date, been widely appreciated. Its widespread occurrence is, therefore, unlikely to be the result of deliberate management control. It is, rather, feedback from locally focussed validation and correction activities that are the likely causes. Nevertheless, since such behaviour results from local and sequential control, from correction not specifically aimed at managing growth, short term, incremental growth fluctuates. Such fluctuations reflect, for example, ambitious high content releases with significant (much larger than average) growth. The inevitable clean up and re-structuring has to follow. The overall result is stabilisation.

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 growth and change data as a function of real time or rsn to determine system evolution trends. The choice includes objects, lines of code (locs), modules (holons), inputs and outputs, 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.
- b. Develop automatic tools to interpret the data as it builds up over a period of time to derive, for example, the dynamic trend patterns. In FEAST/1 it has been shown how, using a scripting language such as Perl [wal96], unplanned records, such as change-logs, can be used as data sources to estimate, *inter alia*, element growth and change rate. A degree of discipline, such as the adoption of a fixed standard pattern for change-log data added to change-log preparation, should facilitate data extraction. Once data is available, derive models that reflect historical growth trends. Indicators may be computed by, for example fitting an inverse square trend line or some other appropriate curve.
- c. Once the models have stabilised (after perhaps some six or so releases of a system) the models should provide first estimates of the trends and patterns of growth and changes per release or time unit as determined by the system dynamics. These measures must be updated and the trend indicator recomputed or redisplayed at regular time intervals and/or for each subsequent release.

- d. On the basis of the observations reported above in section 6d, 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 the level *m* suggested by the trend model(s).
  - 2 when the need for growth per release needs to be significantly greater than *m*, seek to reduce it by, for example, spreading it over two or more releases.
  - 3 plan and implement a 'preparation release' that pre-cleans the system, if limiting growth to around *m* is difficult or not possible.
  - 4 alternatively, allow for a longer release period to prepare to handle problems at integration, a higher that normal fault report rate, some user discontent.
  - 5 if the required release increment is near to or above $m+2s$ the steps in 2 - 4 must be even more rigorously pursued. Prepare to cope with and control a period of system instability, provide for a possible need for more than average customer support and accept that, as outlined above, a major *recovery* release may be required.

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'. 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 change. 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.

The sixth law 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 a potentially infinite set. The domain of operation is also potentially infinite, 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. 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 a given change 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, apply when one is dealing with systems that were component based or that made widespread use of COTS [hyb97, leh98a] 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.

10 Seventh Law: Declining Quality - *The quality of E-type systems will appear to be declining unless they are rigorously adapted, as required, to take into account changes in the operational environment*

This law follows directly from the first and sixth laws. As briefly discussed in the previous section, to remain satisfactory in use in a changing operational domain, an *E*-type system must undergo changes and additions to adapt and extend it. Functionality must be changed and extended. To achieve this, new blocks of code are attached, new interactions and interfaces are created, one on top of the other. If such changes are not made, embedded assumptions become falsified, mismatch with the operational domains increases. Additions will tend to be increasingly remote from the established architecture, function and structure. All in all 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 faults 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 faults, increasing difficulty of adaptation and growth will all cause stakeholder satisfaction to decline. Each represents a factor in declining system quality.

There are many approaches to defining software quality. 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. To do so involves issues more adequately discussed and examined when a theory of software evolution [leh00b] is available. 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 required to support it, the support provided to the organisation in pursuing its business and so on.

Once identified as being of concern in relation to the business or task being addressed, aspects of quality must be quantified to be 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 fault reports per release to obtain a display of the fault rate trend with time. 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 correlations [hum91]. One may also seek relationships with other process and product measures such as the size of or the number of fixes in previous releases, sub-system or module size, testing effort and so on. When enough metric data is available, and the process is sufficiently mature, models such as Bayesian nets may be useful to predict defect rates [fen99]. The above examples all relate to fault related aspects of quality. Other measures may be defined, collected and analysed in an analogous manner.

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, i.e:

- a. 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.
- b. Devote some portion of evolution resources to complexity reduction of all sorts, restructuring and the removal of "dead" system elements. Though primarily *antiregressive*, without immediate revenue benefit, they help ensure future changeability, potential for future reliable and cost effective evolution. Hence, in the long run, they are profitable.
- c. 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.
- d. 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.
- e. Monitor appropriate system attributes to predict the need for cleanup, restructuring or replacement of parts or the whole.

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 vast literature in this field [e.g. boe78].

11 Eighth Law: Feedback System - *E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.*

This is the key law of the eight and underlies the behaviour encapsulated by the other seven. The relationships between them are currently being investigated. It is hoped to describe them in a formal theory that covers the observed phenomenology [leh00b].

The behaviour of feedback systems is not and cannot, in general, be described *directly* in terms of the aggregate 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[5] 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 as illustrated by the end life of OS/360-70 [leh85,98c]. The observed instability and final break up of that system was attributed to excessive positive feedback, arising from competitive market and user pressure for virtual memory and interactive operation. In any event 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/1 and earlier studies have provided behavioural evidence to support this analysis and the eighth law. FEAST/2 is continuing the investigations.

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 exert on the process. The basic message of the eighth law is, therefore, that in the long term managers are not absolutely free to adopt any action considered appropriate from some specific business or other 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 in FEAST publications [fea00]. To achieve this requires models that reflect the dynamic forces and behaviour. The FEAST project and other sources[6] 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, 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 [for61], 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 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.

[5] The global process includes all activities that impact system evolution including but not limited to those undertaken by technical, management, marketing, user support personnel and users, etc.)

[6] See, for example, the work done in the context of software process simulation [jss99,pro00].

12 The FEAST Hypothesis[7] - 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. 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. The consequences of human decision and action, of variable forces and of flow levels may be described by statistical distributions. It is an open question whether such quantitative models must be specific to each system, or can be generic, constructed and calibrated to be valid over a number of systems or organisations.

FEAST/1 has made some progress in this regard and FEAST/2 is continuing this line of work [fea00]. A number of system dynamics models [for61] using the Vensim tool [ven95] are now being calibrated and investigated [wer98,cha99]. Many of their variables reflect organisational characteristics and it may be possible to derive generic versions. Generic modelling and analysis methods are already emerging [ram00a]. 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 11b 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 Uncertainty Principle – *The real world outcome of any E-type software execution is inherently uncertain with the precise area of uncertainty also not knowable*

This principle, first formulated in the late 80s [leh89,90] as a stand alone observation, is now regarded as a direct consequence of the laws [leh00b]. It asserts that the outcome of the execution of an *E*-type program is not absolutely predictable. The likelihood of unsatisfactory execution may be small but a *guarantee* of satisfactory results can never be given no matter how impeccable previous operation has been. This statement may sound alarmist or trivial (after all there can always be unpredictable hardware failure) but that is not the issue. A proven fact is a fact and by accepting this and taking appropriate steps even a small likelihood may be further reduced.

There are several sources of software uncertainty [leh89,90]. 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 a traffic control system. Others may be unconscious, for example to ignore the gravitational pull of the moon in setting up control software for a particle accelerator[8]. 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, invalidates assumptions and their reflection in code or documentation.

As indicated in section 3 the real world domain is infinite. 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.

---

[7] One of a number of alternative statements formulated over the years.

[8] A major oversight in one of the world's most prestigious nuclear physics research centres.

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. 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

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 escaping. 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

Determination of a product evolution strategy is a management responsibility [sta97] that must take into account many 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 and white box (system dynamics) models [ram00a] reflect, *inter alia*, 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.

Over recent years there has been a move towards component-based software development and the use of COTS [hyb97]. Thus activities such as *integration* are becoming ever more important in terms of, for example, the effort they absorb or their frequency as a source of faults. Does this trend invalidate the laws, conclusions and rules summarised in this paper? It is still too early for a definitive answer though much will depend on the thoroughness, accuracy and completeness with which developers of such components document their assumptions, the various bounds of the operational validity [leh98a]. Preliminary assessment [leh98a] suggests that the laws will continue to be important in the context of component-intensive processes and products. Definitive conclusions must await a study of their evolutionary behaviour.

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. Extensive 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 ongoing 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 (technical and non-technical), management and organisations, such studies demand interdisciplinary collaboration.

Finally, it must be stressed that the study 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 and support sought. A theoretical framework is being developed [leh00b]. It is hoped that this effort will provide significant benefit to a world and a society relying ever more on computers.

15 Acknowledgements

My sincere thanks are due to my colleagues Juan F. Ramil, Dr Goel Kahen and Siew Lim for their continuing support, questioning and constructive criticism. Equally to Professors Perry and Turski, to our collaborators at BT Labs, DERA-MOD, ICL, Logica, Lucent and Matra-BAe and to their respective staffs who have provided the data, process insight and constructive criticism that has made the recent advances possible. Finally to EPSRC and its reviewers, for funding FEAST/1 (grant no. GR/K86008), FEAST/2 (grant no. GR/M44101) and the Senior Visiting Fellowships (GR/L07437 and GR/L96561).

16 References[9]

[bau67] Baumol WJ, *Macro-Economics of Unbalanced Growth - The Anatomy of Urban Cities*, Am. Econ. Rev. June 1967, pp. 415 - 426

[bel72] Belady LA and Lehman MM, *An Introduction to Program Growth Dynamics*, in Statistical Computer Performance Evaluation, W. Freiburger (ed.), Acad. Press, NY, 1972, pp. 503 - 511

[boe78] Boehm B, Brown JR, Kaspar JR, Lipow M, MacCleod CJ and Merritt MJ, *Characteristics of Software Quality*, North Holland, 1978

[box97] Box G and Luceño A, *Statistical Control by Monitoring and Feedback Adjustment*, Wiley, New York, 1997, 327p.

[bro75] Brooks FP, *The Mythical Man-Month*, Addison-Wesley, Reading, MA, first edition 1975, 20th Aniv. Edition 1995, 322p.(20th aniv. Ed. 1995)

[cha99] Chatters BW et al., *Modelling a Software Evolution Process*, in Proc. of ProSim'99, Softw. Proc. Modelling and Simulation Worksh., Silver Falls, OR, June 28–30, 1999. To appear as Modelling a Long Term Software Evolution Process, in Software Process - Improvement and Practice in 2000.

[fea00] *FEAST Projects Web Site*, Department of Computing, Imperial College, London, UK, http://www-dse.doc.ic.ac.uk/~mml/feast/.

[fen99] Fenton NE and Neil M, *A Critique of Software Defect Prediction Models*, 25(3) IEEE Trans. on Softw. Eng., 1999

[for61] Forrester JW, *Industrial Dynamics*, MIT Press, Cambridge, Mass., 1961

[gen00] *Call for Papers*, GCSE '2000, Int. Symp. on Generative and Component Based Softw. Eng, 9 – 12 Oct. 2000, Erfurt, Germany http://www.netobjectdays.org/node00/en/Authors/cfp-gcse.html

[gil81] Gilb T, *Evolutionary Development*, ACM Softw. Eng. Notes, April 1981

[hum91] Humphrey WS and Singpurwalla ND, *Predicting (Individual) Software Productivity*, IEEE Trans. on Softw. Eng., Vol. 17, No. 2, February 1991, pp. 196 - 207

[hyb97] Hybertson, DW, Anh DT and Thomas WM, *Maintenance of COTS-intensive Software Systems*, Softw. Maintenance: Res. and Pract., Vol. 9, 1997, 203 - 216

[jss99] The Journal of Systems and Software, *Special Issue on Software Process Simulation Modelling*, Vol. 46, No. 2/3, April 1999

[kit82] Kitchenham B, *System Evolution Dynamics of VME/B*, ICL Tech. J., May 1982, pp. 42-57

[law82] Lawrence MJ, *An Examination of Evolution Dynamics*, Proc. 6th Int. Conf. on Softw. Eng., Tokyo, Japan, 13 - 16 Sep. 1982., IEEE Comp. Soc. ord. n. 422, IEEE cat. n. 81CH1795-4, pp. 188 - 196

[leh69]* Lehman MM, *The Programming Process*, IBM Research Report RC 2722, IBM Research Centre, Yorktown Heights, NY, Sept. 1969

[leh74]* *id*, *Programs, Cities, Students, Limits to Growth?*, Inaugural Lecture, in Imperial College of Science and Technology Inaugural Lecture Series, Vol. 9, 1970, 1974, pp. 211-229. Also in Programming Methodology, (D. Gries. ed.), Springer Verlag, 1978, pp. 42-62.

[leh77] *id*, *Human Thought and Action as an Ingredient of System Behaviour*, Contribution to the Encyclopaedia of Ignorance, July 1976, Imp. Col of Sc. Tech., CCD Research Report 76/12 (Pergamon Press 1977) pp. 397-354

[leh78]* *id*, *Laws of Program Evolution—Rules and Tools for Programming Management*, Proc. Infotech State of the Art Conf., Why Software Projects Fail?, Apr. 1978, pp. 11/1-11/25.

[9] Papers identified with a "*" may be found in [leh85].

[leh80]* *id*, *On Understanding Laws, Evolution, and Conservation in the Large Program Life Cycle*, J. of Sys. and Softw., v. 1, n. 3, 1980, pp. 213-221.

[leh85] Lehman MM, and Belady LA, *Program Evolution—Processes of Software Change*, Academic Press, London, 1985.

[leh89] Lehman MM, *Uncertainty in Computer Application and its Control through the Engineering of Software*, J. of Softw. Maint., Res. and Pract., vol. 1, 1 Sept. 1989, pp. 3 - 27

[leh90] *id*, *Uncertainty in Computer Application*, Tech. Letter, CACM, vol. 33, n. 5, pp. 584, May 1990

[leh94] *id*, *Feedback in the Software Evolution Process*, Keynote Address, CSR Eleventh Annual Workshop on Software Evolution: Models and Metrics, Dublin, Ireland, Sept. 7-9, 1994, and in Information and Software Technology, special issue on Software Maintenance, Vol. 38, No. 11, 1996, Elsevier, 1996, pp. 681-686.

[leh96a] Lehman MM, Perry DE and Turski WM, *Why is it so Hard to Find Feedback Control in Software Processes?* Invited Talk, Proc. of the 19th Australasian Comp. Sc. Conf., Melbourne, Australia, Jan 31 - Feb 2 1996, pp. 107-115

[leh96b] Lehman MM, and Stenning V, *FEAST/1: Case for Support*, Department of Computing, Imperial College, London, UK, Mar. 1996. Available from links at the FEAST project web site [fea00].

[leh97] Lehman MM, *Laws of Software Evolution Revisited*, Proceedings of EWSPT'96, Nancy, LNCS 1149, Springer Verlag, 1997, pp. 108–124.

[leh98a] Lehman MM and Ramil JF, *Implications of Laws of Software Evolution on Continuing Successful Use of COTS Software*, ICSTM, Dept. of Comp., Tech. Rep. 98/8, incl. panel pos. statement., ICSM '98, Washington DC, 16 - 18 Nov. 1998

[leh98b] Lehman MM, *FEAST/2: Case for Support*, Department of Computing, Imperial College, London, UK, Jul. 1998. Available from links at the FEAST project web site [fea00].

[leh98c] Lehman MM, Perry DE, and Ramil JF, *On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution*, in Proceedings of the Fifth International Metrics Symposium, Metrics '98, Bethesda, Maryland, Nov. 20-21, 1998.

[leh98d] Lehman MM, *The Future of Software - Managing Evolution*, inv. contr., v.15, n.1, IEEE Softw., Jan-Feb 1998, pp. 40-44

[leh00a] Lehman MM, Ramil JF and Wernick PD, *Metrics-Based Process Modelling With Illustrations From The FEAST/1 Project*, to appear as chapter 10 in Bustard D, Kawalek P and Norris M (eds.). Systems Modelling for Business Process Improvement, Artech House, April 2000

[leh00b] Lehman MM, *Approach to a Theory of Software Process and Software Evolution*, Position Paper, FEAST 2000 Workshop, Imp. Col., 10 - 12 July 2000

[pfl98] Pfleeger SL, *The Nature of System Change*, IEEE-Softw. v.15, n.3; May-June 1998; pp. 87-90.

[pro00] *Call for Papers and Participation*, ProSim'2000, Software Process Simulation and Modelling, July 12 - 14, Imperial College, London, http://www.prosim.org

[ram00a] Ramil JF, Lehman MM and Kahen G, The FEAST *Approach to Quantitative Process Modelling of Software Evolution Processes*, to appear in Proc. PROFES'2000 2nd Int. Conf.. on Product Focused Softw. Proc. Impr., Oulu, Finland, June 20 - 22, 2000, LNCS Springer.

[ram00b] Ramil JF and Lehman MM, *Metrics of Software Evolution as Effort Predictors - A Case Study*, Proc. Int. Conf. on Software Maintenance, October 11-14, 2000, San Jose, CA

[tur96] Turski WM, *Reference Model for Smooth Growth of Software Systems*, IEEE Transactions on Software Engineering, Vol. 22, No. 8, Aug. 1996.

[tur00] Turski WM, *An Essay on Software Engineering at the Turn of the Century*, Proc. ETAPS 2000, Lect. Notes on Comp. Science, Mar. 2000

[ven95] *Vensim Reference Manual*, Ver. 1.62, Ventana Systems Inc., Belmont, MA, 1995.

[wal96] Wall L, et al, *Programming Perl*, O'Reilly & Associates, Sebastopol, CA, 645 pps. 1996

[wer98] Wernick P and Lehman MM, *Software Process Dynamic Modelling for FEAST/1*, Journal of Systems and Software, 46, 1999: 193 - 201.

[woo79]* Woodside CM, *A Mathematical Model for the Evolution of Software*, J. of Sys. and Softw. vol. 1, no. 4, Oct. 1980, pp. 337 - 345 and in [leh85], pp. 339 - 354