Rules and Tools for Software Evolution Planning and Management

M M Lehman
Department of Computing
Imperial College
London SW7 2PH
+44 (0)20 7594 8214
< mml@doc.ic.ac.uk >
http://www-dse.doc.ic.ac.uk/~mml

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. Supplemented by an Uncertainty Principle and a FEAST Hypothesis, the eight laws have even found their place in a number of software engineering curricula.
Based on all these and on the results of the recent FEAST/1 and current FEAST/2 research projects, this paper develops and presents over fifty rules for application in software system process planning, management and implementation, suggesting tools to be developed to support the application. The listing is ordered by the laws that encapsulate the observed phenomena and lead to the recommended procedure. Each sub-list is preceded by a textual discussion to outline the basis for justification of the recommended procedure. The text is fully referenced to direct the interested reader to literature that records the data, behaviours, models and interpretations more fully. These were obtained from studies of several industrially evolved systems 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 and earlier studies, their observations, models derived therefrom, their interpretation and on conclusions reached by the FEAST 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, initial 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 data, observations, behaviours, models and interpretations on which the conclusions are based. This relatively brief paper cannot provide all the detail, is restricted to a general review of the relevant phenomenology. Where appropriate, references to other sources and information will be given.

Observations and rules relevant to software system evolution planning and management [32] were first identified during studies of the evolution of OS/360-70 and other systems between 1968 [17] and 1985 [20,22,45]. More recently, with the active collaboration of ICL, Logica and Matra BAe Dynamics, the FEAST/1 project [27] (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 1970s studies. Such similarity in the long-term evolutionary behaviour of widely different systems and development domains over a period of rapidly evolving technology suggests that the observed behaviour and the rules derived therefrom is not primarily due to the technologies employed. 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 business and organisational evolution.

The FEAST/2 project (1999 - 2001) [30] expects, *inter alia*, to provide a firm empirical base for the conclusions recorded here. In refining earlier results, it identifies sources and patterns of evolutionary trends and indicates implications on evolutionary behaviour. Such trends are unlikely to be limited to specific implementations, application areas or environments , are expected to be more widely relevant.

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. For clarification, more detail or to see the data and analyses from which the conclusions and recommendations were derived the relevant literature can be accessed via the FEAST web site [7].

As a practical guide, the paper follows the historical ordering of the laws of software evolution and other principles [22,28] from which the rules were derived. Some overlap between sections is, however, inevitable if excessive cross-references are to be avoided. We trust that the paper is, nevertheless, useful and finds practical application.

2 Laws of Software Evolution

The eight laws of software evolution formulated over the 70s and 80s [18,20,21] are listed in Table 1. The statement of the first three laws [l18] emerged from a follow up of the 1969 study of the evolution of IBM OS/360 and its successor OS/370 [17]. These results were further strengthened during the seventies by those of other software system evolution studies. Additional support came from an ICL study in the eighties [15] but some criticism directed, primarily, at the inadequacy of the statistical support came from Lawrence [16]. The present listing incorporates minor modifications that reflect new insights gained during the FEAST/1 project [25,28,31].

Over the years, the laws have gradually become recognised as providing useful inputs to understanding of the software process [35] and have found their place in a number of university software engineering curricula. Additional support for six of the eight laws has accumulated as the data obtained from its collaborators [31,33] was modelled and analysed in the FEAST/1 project [27]. The remaining two laws were neither supported nor negated by the evidence acquired from the latest studies because of the absence of relevant data. They appear, however, to reflect a basic truth whose precise nature still needs clarification.

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

As indicated above, 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 regarded as 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 initial thoughts on that topic are now available [34]. If that development is successful it should put to rest the main criticisms to which the laws have been subjected over the years [16]. These addressed the absence of precise definitions and of statements of assumptions, their being based on data from a single, atypical source OS/360, the fact that that system represented an outmoded software technology. The use of the term *laws* in relation to observations about phenomena directed, managed and reflecting human activity was also questioned. Human decision was surely the principle determinant of much of the behaviour encapsulated in the laws. In response it was noted that though the demand for more wide spread evidence of their validity was undoubtedly justified, the final criticism was not valid. It was precisely human involvement that justified use of the term. The term *law* is appropriate and was selected 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 such.

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

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 [22]. 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* [41]. 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. By their omission from the specification other properties implicitly become "don't cares". 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 least as far as the original contract is concerned. At some later time the program may require fixing or enhancement as a result of an error in the specification, changes in the purpose for which the program is required, changes in the operational domain or some other reason. An updated specification, including changes to the embedded assumption set can then be prepared and a new program derived. This may be achieved by modification of the old program or by *ab initio* development of a new one.

The minds of individual programmers cannot be read. One can therefore, not know what *assumptions* are made during the course of their work. Any such assumptions become embedded in and part of the program even though not originally specified. As the operational domain or other circumstances change some assumptions will become invalid and affect program behaviour in unpredictable, possibly 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 necessary.

The role of *S*-type programs in the programming process follows. The unconscious, unconsidered, injection of assumptions into *S*- and *E*-type programs may be minimised by providing a *precise* and *complete* specification for each individually assigned task, that is, by assigning only *S*-type programs to individuals. The lowest level *bricks* from which larger programs and complex systems are built should be of type *S*. The main body of assumptions will then be explicit in the specification of these bricks, or implied by omissions from those specifications. Each elemental parts of the total system can be verified objectively against its specification.

The detection of implied, often unconscious, assumptions is, however, difficult and 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, recorded and added to the specification. But some assumptions will slip through the net.

The better the record of assumptions, the more it is updated, the simpler will it be to maintain the programs valid in a changing world, the less likely is it that the program displays unexpected or strange behaviour in execution, the more confidence can one 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 does not, therefore, decrease their practical significance. Both *S*- and *E*-type programs have a role to play in system development and evolution.

[1] This is becoming more widely recognised and addressed, e.g., in the context of component-based software [10].

4 Practical Implications of the Laws

We now 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 [7]. Together these provide the basis for a comprehensive *theory of software evolution* such as that now being developed [34].

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, in terms of relative importance for example, 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 the specification. Such ***key*** properties serve as task definitions[2].
b. The (long-term) goal should be to express specifications formally within the organisation's quality system.
c. A goal of every process must be to capture and retain assumptions underlying the specification, both those that form part of its inputs and those arising during the subsequent development process. This should occur during the initial development of specifications, all the implementation steps that follow and any subsequent revisions. Further opportunities arise during verification and/or validation of either.
d. The previous point represents an ideal. In practice modifications to the specification, assumptions that accompany them and independent assumptions may be recorded in user or other documentation. This is appropriate since users need to be informed of assumptions that affect their usage. To treat that as a replacement for recording in the system documentation cannot be considered best practice.
e. It must be recognised and agreed by all involved that whatever is not included in the specification is left to the judgement of the assignee or whoever he/she wishes to consult. Any such decision must be confirmed as not being inconsistent with the specification and must either
    1. be documented in an *exclusion* document or
    2. formally approved and added to the specification or to other 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.

[2] The extent to which other properties are included in the specification or left to develop as a by-product of the development process is a matter of expert judgement as determined by stakeholder experts. The many facets of this issue cannot be pursued here.

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

The real world has a potentially unbounded number of attributes. The operational domain and the applications pursued in it are part of the real world, are initially undefined and are also intrinsically unbounded. The *E*-type software system implementing or supporting the application and, for that matter, the wider total application system, on the other hand, are essentially finite. Thus as a model of an application in the real world, every *E*-type system is essentially incomplete [19,22,23,24]. A process of bounding, abstraction and transformation is required to specify and implement the system. This finitisation process excludes an unbounded number of attributes of the operational domain and the application from the system specification. Only exclusions specifically identified as such in the specification are determined. All others are left to the implementers' choice, conscious or unconscious and each such exclusion reflects an *assumption* about required program properties. Some of the assumptions will be explicit, others implicit, some by inclusion, others by exclusion, some recorded, some unrecorded. They will be reflected in the application implementation and/or documentation in a variety of ways, not discussed further here. Note that implicit exclusion from the system is as real in impacting system operation as are inclusions. In summary, every *E*-type system has embedded in it an unbounded 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 rendered the system *satisfactory* 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 application or its domain of operation so requiring changes to the system to restore it to being an acceptable model of that domain. Taken together, these facts lead to the first law of software evolution and its practical outcome in the unending maintenance that has been the universal experience of computer users since the start of serious computer application.

There follows a listing of some of the practical implications of this unending need for change to *E*-type systems to adapt them to their respective changing operational domains.

a. Comprehensive, structured, documentation must be created and maintained to record the growth in content, interdependencies and complexity (see second law) to facilitate control of that growth as changes are applied over the system lifetime, change upon change.
b. 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.
c. There must be a conscious effort to control, and reduce complexity and its growth as changes are made locally or in interfaces with the remainder of the system.
d. The assumption set must be reviewed as an integral part of release planning and periodically thereafter to detect domain and other changes that conflict with the existing set, or violate constraints. The widespread practice of Beta releases should be a help in this regard in relation, for example, to the detection of such changes or violations.
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 over a sequence of releases of, for example, patterns of incremental growth (see next para.) and of numbers of changes per release 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 [7].
f. FEAST/1 and earlier *incremental growth models* (section 9), 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 [45]. 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.

6 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 other reasons. 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 [40] with the *mae* (mean absolute error), that is the mean absolute difference, between release sizes as predicted by the model and their actual size of order 6% (see sect. 7 and [7]).

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 and other *anti-regressive* [18] activities are an integral part of the development and maintenance responsibility. The *immediate* benefits of such effort will, generally, be relatively small. The long-term impact is, however, likely to be significant; may indeed, at some stage, make the difference between survival of a system and its replacement or demise. Release planning for one or a series of releases must carefully consider the degree, timing and distribution of anti-regressive activity.

c. Effective control of complexity requires that, as part of the maintenance and evolution process, its growth be measured to determine when anti regressive activity as above should be initiated to reverse adverse trends.

d. 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 so as to free resources for *progressive* [1] activity such as system enhancement and extension, system complexity and future effort/cost are likely to increase, productivity, evolution rates, stakeholder dissatisfaction and system quality to decline (law VII). If additional resources are provided, resources for system enhancement and growth are likely to be reduced. Once again the system evolution rate and with it stakeholder satisfaction, 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 a strategy most likely to help achieve corporate business or other goals requiring to be optimised.

e The fraction of total activity that should, on average be devoted to anti-regressive work or how this should be divided between releases of any system is likely to vary as a function of factors such as organisational, project, process, application area and so on. The development of models for estimating requires R & D but there is reason to believe that generic parameterised models can eventually be developed. In the meantime it behoves software organisations to investigate this with reference to individual environments.

f Despite the absence of sound models, many strategies for optimising progressive/antiregressive work balance can be conceived. One might, for example, alternate between releases focussing primarily on each of the two classes[45]. Alternatively one might seek to set up a sequence along the lines of a new function release (beta?), followed by a fault fixing release followed by an internal release reducing complexity. Selection of a strategy has technical, cost and business implications and requires local exploration and decision but one can conceive of developing generic strategies.

7 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 [18], 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 [7,40]. 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 since the size of individual releases is mainly determined by management focus on functional needs. 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 embedded systems where storage is often limited by physical considerations.

The FEAST projects have identified two exceptions to this surprisingly simple growth 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 by 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. However, its growth rate fluctuated wildly for a further 6 releases before fission into VS1, VS2. Over that region both inverse square and linear growth models are 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.

As noted above, with some exceptions, size is not, in general, a managed system parameter. Several growth related factors come to mind. Complexity growth is almost certainly a constraining factor. Increasing capability that may follow growing maturity may help overcome this though current evidence, the widely observed inverse square rate for example, suggests that complexity growth dominates. Further investigation of this issue is clearly of major interest and ripe for further research investigation.

A common feature of the growth patterns of *all* the release based systems observed is a superimposed *ripple* [2,7,20]. 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 [6]. 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*[3] 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 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 [6]. 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 appears 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 (sect. 12). An investigation, in a wider context, of this relationship is about to be initiated [34].

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 [26]. 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. Process changes may have the intended local impact but unexpected global consequences. Each may work for or against deviations from established levels, patterns and trajectories but lead to counter-intuitive consequences, particularly when decisions are taken in the absence of a global picture. 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 not themselves 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. Applying measurement and modelling techniques such as those used in FEAST/1 [33], determine typical patterns, trends, rates and rates of change of a number of projects within the organisation. To obtain models with an mae of O(5%) data on at least six, possibly eight to ten, releases are likely to be required. Quantitatively, the rules quoted in this report apply to systems that have evolved over that number of releases. Qualitatively, however, the rules have more general implications.

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

[3] 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.

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. The rules are here expressed in release based units. For observations based on incremental growth per standard real time unit, analogous safe limits are likely to exist. They are likely to be a function of the interval between observation. More work is required to determine whether meaningful relationships can be identified, quantified and modelled for predictive purposes.

1. As briefly discussed above and in greater detail in the FEAST publications, those 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 safely.
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 wise to plan for a follow-on clean-up release. Even if not planned, a zero growth release may to be required. Note that $m+2s$ has long been identified as an alarm limit, for example, in statistical process monitoring/control [4].
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 instability over one or more subsequent releases. At best, it is likely to require to be followed by a major clean up release which concentrates on fault fixing, documentation updating and *anti-regressive* work such as restructuring, the elimination of *dead code* and other anti-regressive work.

e. The application of some form of *evolutionary development* approach [11] should be considered if the number of items on the "to be done" list for a release in planning would, if implemented in one release, lead to incremental growth in excess of the levels indicated above. Such a strategy would be appropriate whenever the size and/or complexity of the required addition is large. In that event, several alternatives may be considered. They include spreading the work over two or more releases, the *delivery* to users of the new functionality over two or more releases with mechanisms in place to return to older version if necessary, support group reinforcement, preceding the release with one or more clean up releases or 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.

8 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 [2,20,22]. Insight into the underlying process mechanisms and related phenomena supported the interpretation encapsulated in the law. Subsequent observations suggested, for example, that the average activity rate measured by the *change rate*, is *stationary*[4] but with changes of the mean and variance at one or more points during the observed life time of the released-based systems. The influence of feedback on the process has been apparent since the early 70s [22] but the full extent was not fully appreciated until recently. If further observations yield similar results and when more understanding is achieved, refinement of the law may be indicated. Aspects of the law relating to the role of feedback stabilisation and its implications are discussed below. Management implications are not considered further here except in so far that they are reflected in the rules.

9 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

[4] Loosely defined, *stationarity* is a property of stochastic processes that display a constant average and variance [4].

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. There are many reasons why fixes are likely to increase as a system ages. If investment in system maintenance remains constant, a drop in resources available for system growth and, hence, a declining growth rate is implied as system complexity increases. 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 rate has indeed 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 system evolution rates and resources has not been seriously investigated, may well be counter-intuitive [5].[5]

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 *most likely* principle source of the declining incremental growth rates observed is increasing complexity as the system ages. This arises because of the super-imposition of changes to achieve, for example, performance enhancement, 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 and complexity. Equally it results in increasing complexity of internal and external interfaces at all levels. These effects are aggravated by the fact that, as the system ages, changes are more likely to be orthogonal to existing system structures. Moreover, 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 implement the changes, to understand them in a changed context, to validate and use the system; in short to ensure that the untouched, the changed and the new portions of the system all operate as required. Thus 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. There are likely to be others.

Though the law refers primarily to long term behaviour, regular short-term variations in incremental growth have also been observed. These are now being studied in FEAST/2.

A related aspect of that 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* [29]. 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. That is, the decline is believed to be due to the system dynamics and not to human decision. Nevertheless, since such behaviour results from local and sequential control, from correction not specifically aimed at managing growth, the short term incremental growth rate fluctuates. Such fluctuations reflect, for example, ambitious high content releases dictated by user demand, perceived needs or competitive pressure. These are likely to lead to larger than average growth and growth rates.

[5] FEAST/2 is now directly addressing the topic of cost estimation in the context of software evolution [38]. The study should also advance understanding of the various drivers that, individually and jointly, influence growth and change rates.

Inevitably, clean up and re-structuring must follow. The overall effect is stabilisation.

This and further analysis, in phenomenological terms, of the 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 [43], records, such as change-logs, not conceived for the purpose of metric unplanned, can be used as data sources to estimate such properties 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, will facilitate data extraction. Once data is available, models that reflect historical growth trends can and should be derived. Planning and management indicators for future use may then be computed by, for example fitting an inverse square trend line or some other appropriate curve.

c. Once the *model* parameters have *stabilised* (FEAST observations indicated that this required some six 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. Somewhat more data points will be required to reach, confidence levels of, say 5%. Measures must be updated and the trend indicator recomputed or redisplayed for each subsequent release and/or at regular time intervals.

d. On the basis of the observations reported in section 7d, 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 the expected growth of a release appears to require growth significantly greater than *m*, seek to reduce it, for example, by spreading the additional functionality 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.

e. If the required release increment is near to or above *m*+2*s* 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.

10 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’ as discussed in section 5. 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 unbounded set. The domain of operation is also potentially unbounded, 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 included to fill the gap. The system needs to be evolved to satisfactorily support new situations and circumstances, that is new requirements.

Though they have different causes and represent, in many ways, different circumstances, the steps to be taken to take cognisance of the sixth law do not, in principle, differ radically from those listed for the first. There are, however, differences due to the fact that the former is, mainly, concerned with functional and behavioural change whereas the latter leads, in general, directly to additions to the existing system and to its growth. In practice, it may be difficult or inappropriate to associate a given change with either law. Nevertheless, since the two are due to different phenomena they also are likely to lead different, though overlapping, recommendations. This report, however, does not distinguish between the rules implied by one, the other or both. Recommendations are listed together in section 5.

Some further remarks are, however, appropriate in the context of the sixth law. 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 to control the exchange of 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 are component based or that make widespread use of COTS [13,29]. They too will evolve. A sound architectural and structural base and application of the rules in this report will reduce the effort that will inevitably be required when such systems are extended. The comments of this paragraph may be taken as an additional rule.

11 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 previous sections, 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 as well as behavioural detail must be changed and extended. To achieve this, new blocks of code are attached, new interactions and interfaces are created on top of one another. 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 faults potential will increase as embedded assumptions are inadvertently violated and the potential for undesired interactions created. From the point of view of performance, behaviour and evolution, adaptation and growth effort increase. Growing complexity, mismatch with operational domains, declining performance, increasing numbers of faults, increasing difficulty of adaptation and growth, all lead to a decline in stakeholder satisfaction. 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 ones. 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, on the one hand, organisational and stakeholder concerns and, on the other, issues more adequately discussed in terms of a theory of software evolution [34]. The bottom line is that quality is a function of many factors whose relative significance will vary with circumstances. Users in the field will be concerned with performance, reliability, functionality, adaptability. The concern of a CEO, at the other extreme, will be with the system's contribution to corporate profitability, market share and 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 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 [12]. 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 data is available, and the process is sufficiently mature, models such as Bayesian nets may be useful to predict defect rates [8]. 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, primarily, to a growth in complexity 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, the removal of "dead" system elements and so on. Though primarily *antiregressive*, without immediate revenue benefit, this will help ensure future changeability, potential for future reliable and cost effective evolution. Hence, in the long run, the investment is 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. Verify the validity of assumptions with users and/or other stakeholders.
e. Assess the impact if the assumption were to be invalid, for example what changes would need to be made to correct for the invalid assumption?
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 must be developed.
g. Monitor appropriate system attributes to predict the need for cleanup, restructuring or replacement of parts or the whole.

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

This is the key law and underlies the behaviour encapsulated by the other seven. It is hoped to develop a formal theory that covers and describes the observed phenomenology and the relationships between the laws [34].

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 constrains the ways that 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[6] 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. For sound software process planning, management and improvement the feedback nature of the process must be mastered.

Consider, for example, the growth and stabilisation processes described by the first and third laws (sects. 5 & 7). 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 [22,31]. The observed instability and final break up of that system was attributed to excessive positive feedback, arising from competitive market and user demand 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 side effects and drive it in the desired direction. Stabilisation results. FEAST/1 and earlier studies have provided behavioural evidence to support this analysis and the eighth law. FEAST/2 is continuing the investigation.

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 in wider, global, domains. As a result there are significant limitations that management can exert on control of the process. The basic message of the eighth law is, in fact, 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.

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 [7]. To achieve this requires models that reflect the dynamic forces and behaviour. The FEAST project and other sources[7] 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.

FEAST results suggest 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 evolution process. It follows that the interactions of maintenance, planning, marketing, user support, corporate personnel and others 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

[6] 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.)

[7] See, for example, work done in the context of software process simulation [14,36].

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 [9], calibrate and apply sensitivity analysis to determine the influence and relative importance of the paths and controls.

d In developing the models make certain to include all personnel and activities that feedback information or instructions that may be used to direct, control or change the process

e 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.

f 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.

13 The FEAST Hypothesis - 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. It is believed that in the long run parameterised generic models can be developed to fit a wide range of circumstances. Some classes of systems, e.g. safety critical, or organisations, e.g. multi-nationals could be expected to benefit from tailored models.

FEAST/1 has made some progress in this regard and FEAST/2 is continuing this line of work [7]. A number of system dynamics models [9] using the Vensim tool [42] are now being calibrated and investigated [6,44]. Many of their variables reflect organisational characteristics and this should facilitate their wider application. Early steps in the development of composite process models and related analysis methods are being explored [37].

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 12c 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.

14 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 [23,24] as a stand alone observation, is now regarded as a direct consequence of the laws [34]. 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, identifying the sources of uncertainty and taking appropriate steps even a small likelihood may be further reduced.

There are several sources of software uncertainty [23,24]. The most immediate and one that can be at least partially addressed in process design and management (sect. 3, 4, 10, 11), 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

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

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 unbounded. Once any part of that real world is excluded from the system specification or its implementation the number of assumptions also becomes unbounded. Any one of this unbounded set can become invalid, for example, by extension of the operational domain, by changes to the problem being solved or the activity supported by the system 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 more effectively assumptions are captured, the more carefully they are validated in terms of the current knowledge and in terms of the foreseeable future, the more complete and accessible 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 and enforce the 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 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.

Overall, 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 detection in periodic review of any assumptions that have become invalid.

d. Have all assumptions validated by individuals with the necessary objective and contextual knowledge

e. Review the assumptions database by categories as identified in c, and as reflected in the database structure, at intervals guided by the likelihood or expectation 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 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 will 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. And when they do occur the source of the problem can be relatively rapidly located and fixed. In view of the increasing penetration of computers into all facets of human activity, organisational and individual, and their societal or economically critical roles, any such reduction in the likelihood of failure is important, must not be ignored.

15 Component Based Development

Over recent years there has been a move towards component-based architectures and software development, reuse and the use of COTS [13]. One of the drivers of this trend is an expectation that increased use of such components will increase system development productivity and response time together with system quality, reliability and evolvability. Does this trend invalidate the laws, conclusions and rules summarised in this report? It is still too early for a definitive answer and much will depend on the thoroughness, accuracy and completeness with which developers of such components document their assumptions, the bounds of their operational validity. Preliminary assessment [29] suggests that the laws and rules will continue to be important in the context of component-intensive processes and products. Definitive conclusions must await study of the evolutionary behaviour of such components and of the growth and complexity trends of systems constructed from them.

16 Conclusions

Determination of product evolution strategies is a management responsibility [39] that takes 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 [37] 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 well founded decisions that optimises overall organisational benefits, reducing risk and increasing the long term aggregate benefit.

Lengthy though it is, this paper gives at best an overview of the topic. Its recommendations cannot be applied blindly. To appreciate, and apply them fully, requires understanding of human, technical, usage and organisational, backgrounds that underlie the observations from which they 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. 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, safe performance. The conclusions relate directly to the behaviour of technical and non-technical people, management and organisations. For completeness 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 [34]. It is hoped that this effort will provide significant benefit to a world and a society relying ever more on computers in general and their software in particular.

17 Acknowledgements

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. Special thanks are due to Mr Brian Chatters for his critical and constructive comments on this report. 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).

References[9]

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

[2] 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

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

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

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

[6] 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.

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

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

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

[10] Call for Papers, GCSE '2000, Int. Symp. on Generative and Component Based Softw. Eng, 9 – 12 Oct. 2000, Erfurt, Germany

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

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

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

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

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

[16] 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

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

[9] Papers identified with a "*" may be found in [22].

[18]* 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.

[19] id, Human Thought and Action as an Ingredient of System Behaviour, Contribution to the Encyclopaedia of Ignorance, July 1976, Imperial College of Science. Technology (and Medicine), CCD Research Report 76/12 (Pergamon Press 1977) pp. 397-354

[20]* 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.

[21]* 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.

[22] Lehman MM, and Belady LA, Program Evolution - Processes of Software Change, Acad. Press, London, 1985.

[23] 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

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

[25] 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.

[26] 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

[27] Lehman MM, and Stenning V, FEAST/1: Case for Support, Department of Computing, Imperial College, London, UK, Mar. 1996. Available from the FEAST web site [7].

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

[29] 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. Also as a. panel position. statement., ICSM '98, Washington DC, 16 - 18 Nov. 1998

[30] 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 [7].

[31] 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.

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

[33] 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

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

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

[36] Call for Papers and Participation, ProSim'2000, Software Process Simulation and Modelling, July 12 - 14, Imperial College, London,

[37] 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.

[38] 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

[39] Stark GE and Oman PW, Software Maintenance Management Strategies: Observations from the Field, Journal of Software Maintenance, vol. 9, n. 6, 1997, pp. 365 - 378

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

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

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

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

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

[45]* 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