Integrated Design and Process Technology, **IDPT-2003**
Printed in the United States of America, 2003
©2003 Society for Design and Process Science

Software Evolution – Background, Theory, Practice

Meir M Lehman
School of Computing
Middlesex University
Bounds Green Road
London N11 2NQ, U.K.
tel. +44-20-8411 4225 fax +44-20-8411 6411
mml@mdx.ac.uk
http://www.cs.mdx.ac.uk/staffpages/mml/

Juan F Ramil
Computing Dept.
Faculty of Maths and Computing
The Open University
Walton Hall, Milton Keynes MK7 6AA, U.K.
tel. +44-1908-65 4088 fax +44-1908-65 2140
j.f.ramil@open.ac.uk
http://mcs.open.ac.uk/jfr46

ABSTRACT

This paper presents a brief summary of a 35 years study of the software process and the software evolution phenomenon. It draws attention, *inter alia*, to the *SPE* program classification, a principle of software uncertainty and laws of software evolution. Recent studies have led to refinement of earlier conclusions and provided a basis for formation of a theory of software evolution. Management rules and guidelines derived during the empirical FEAST studies, which are candidate theorems in the proposed theory, are briefly outlined to demonstrate that the topic has practical as well as theoretical significance. Rather than in depth discussion, this paper provides an introductory overview intended to encourage wider study, research and development

Keywords: the software process, assumptions, software engineering, implementation and evolution, laws and theory of software evolution, the uncertainty principle, best practice, management rules and guidelines

INTRODUCTION – THE EARLY DAYS

The term *evolution* describes a class of phenomena observable in many different domains, concrete and abstract. It can involve entities or sets of entities such as natural species, societies, cities, artefacts, concepts, theories, ideas. If any of these undergo *continual progressive change* in one or more of their attributes they are said to *evolve* over time. Change is defined as *progressive* if it results in improvement in some sense. Such improvement may, but need not, include the emergence of new properties.

Often, the change will be driven by a need to *adapt* the individual entity, or a class as a whole, so as to maintain or improve its fitness within a changing environment or circumstances. The change may make the entity, or class of entities, more *useful* or meaningful, or increase its *value* in some other sense. It may also remove properties that are no longer of value or otherwise inappropriate. Radical or fundamental changes are, in general, not considered evolutionary changes. The latter are generally incremental and small relative to the entity or class as a whole, but exceptions may occur.

This *general* definition of evolution is also appropriate for real world, that is *E*-type [leh85], software evolution. The latter has been consistently experienced over many years as evidenced by observations and data acquired by the present authors, their associates and others. The software evolution phenomenon was first identified in the late 60s [leh69] though not termed *evolution* till later [leh74]. Its study was pursued intermittently during the 70s, with early results collected together in [leh85]. The work that led to its discovery and exploration had been seeded by a nine month 1968/9 study of the IBM programming process [leh69,85]. The outcome of that study, recently judged to be as relevant today as it was then, led to the Lehman-Belady collaboration [leh85]. It concentrated primarily on measuring and interpreting the growth of software systems and evolutionary trends in other evolutionary attributes using both *real* and *pseudo*-time (rsn or release sequence numbers [cox66]) measures.

The data that triggered the studies was obtained from IBM's OS/360-70 operating system. Data from other systems [leh78,80a,b,85] followed. These early studies concentrated on the evolutionary behaviour of what were then termed *large*[1] software systems and on the organisations that developed, maintained and evolved them [bel71,72, leh85]. The overall picture revealed a degree of *discipline* exemplified by similarities between growth trends of different systems. This suggested that underlying the detailed evolution of each specific system there is a common *phenomenon* that can be systematically studied and modelled. The resultant models could then be used to forecast future system growth and growth rates.

The software process is conceived, directed, planned, managed, implemented and controlled by humans. At each stage of the process their decisions are assumed to drive and direct the process and determine product properties. In general management decisions are different from one situation to another and are expected to be dominated by the pressures of the moment, with these divergence possibly amplified by the subjective component in every the decision making, Thus, evolutionary behaviours should vary significantly from application to application, organisation to organisation, system to system, time to time and release to release.

[1] The term *large* is, generally, used to describe software whose size in number of lines of code is greater than some arbitrary value. For reasons indicated in [leh79], it is more appropriate to define a large program as one developed by processes involving groups with *two or more* management levels.

©2003 Society for Design and Process Science

The discovery of high level similarities in the evolutionary patterns of software addressing a widening spectrum of applications, development, marketing and user organisations and the emergence of common phenomenological interpretations contradicted this expectation. Instead, it suggested that *similar underlying forces, drove evolutionary growth*. It was suggested that the system of systems formed by the evolving software and the organisations involved in or relating to its evolution and usage constitute and behave like a *feedback system*; more precisely a *self-stabilising* [bel76], multi-level, multi-loop multi-agent, feedback system [leh94].

The models developed were relatively simplistic, limited in particular, by data availability. However, the investigations were not circumscribed to modelling and analysis of growth data and the dynamics of growth. They also included, for example, search for conceptual and theoretical models reflecting understanding of the phenomenon and the forces driving it [e.g. bel72]. This led to significant advances in understanding of the phenomenology as encapsulated in *laws* of software evolution [leh85][2]. It also triggered a terminological change from *software growth dynamics* to *software evolution* [leh74].

THE SECOND WAVE

The early work outlined above went largely unnoticed by the mainline Computer Science and Software Engineering communities. Gradually, however, the phenomenon began to attract other investigators [e.g. kit82, law82]. A major conceptual advance came with formulation of the *software uncertainty principle* [leh89,90,02a], the FEAST (*F*eedback, *E*volution *A*nd *S*oftware *T*echnology) hypothesis and the FEAST projects [leh96b,98]. The overall results of these studies and many of the conclusions to which they led have been widely reported [e.g. leh01a,b,02a,b,c, website]. Their wider impact include implications to real world computer usage. In particular, the insight that followed is very relevant to the growing and active interest in software process improvement.

EVOLUTION: PHENOMENON AND ACTIVITY

It is now widely accepted that *software evolution* may be systematically studied. There are, however, two aspects to such study. What has been termed a *nounal* view of *evolution* [leh00a], focuses on the *nature* of evolution, its *causes*, *properties*, *characteristics*, *consequences*, *impact*, *management*, *control* and *exploitation*. One may also adopt a *verbal* view [leh00a] concerning oneself with providing and improving *means*, processes, activities, languages, methods, tools for example, whereby evolution is implemented.

These views are mutually supportive. Both are necessary and becoming increasingly important as society becomes ever more dependent on computers, and hence on software. As suggested by fig. 1, the need for continual change and adaptation of real world software in response to computer usage and changes in the applications and domains in which they are applied is inevitable and continual. The acts of developing, installing and using the computer system changes both the application and the domain within which it operates, which it influences, and, to some extent, controls.

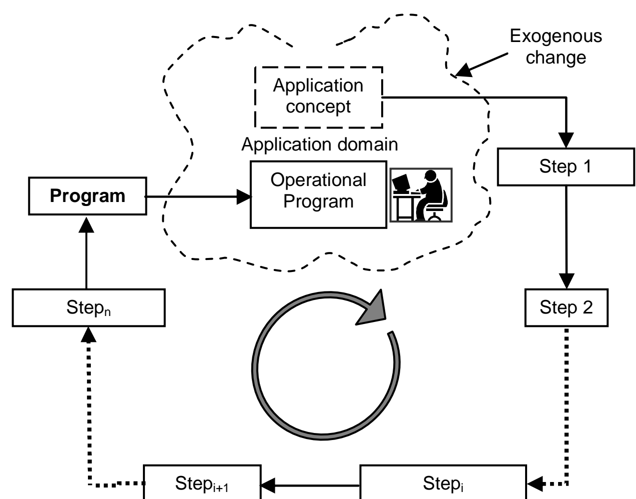

Figure 1 – Feedback: a driver of software evolution

System functionality and behaviour must keep pace with all changes. Defects must be fixed, parameters adjusted, functionality refined and extended, performance improved. The system must be adapted to accommodate operational extension, the need and desire for changes to existing features and for new capability.

As business operation, and organisational behaviour become ever more dependent on software, the consequences of a delay in keeping the system in tune with its changing purpose and domains range from frustration to disaster. Any improvement in *means* to support evolution will, in general, have an impact on quality, usability, timeliness, economic benefit, risk mitigation and so on, that is, on user satisfaction.

Increased *understanding* of the characteristics, nature and impact of evolution, on the other hand, will benefit the software process, evolution planning, process management and process improvement [leh01b]. Insight into the causes, properties and implications of software evolution will make planning, control, execution of process improvement more systematic and effective. It will indicate into the types of activities, methods and tools required, which are likely to be most beneficial, when and how they should be used and how they relate to one another. Studies of

[2] Termed *laws* instead of, for example, observations or hypotheses, because they reflect organisational, economic and social pressures leading to evolutionary behaviours which are largely independent of the individuals, organisations and domains involved in the evolution of the *E*-type systems studied [leh74].

©2003 Society for Design and Process Science

the two views of software evolution must move forward together. Progress in this direction will help development of a theoretical base and framework and accelerate that process.

The approach to the study by Lehman and his collaborators assumed the nounal interpretation. The behavioural similarities across a variety of software systems, has led to the construction of growth models of the same form derived from empirical data though model parameter values are specific to each system [leh97, website]. The identification of qualitatively similar [ram03a,b] but staged [ben00] growth patterns and the resultant significant improvement of model predictive power has further increased confidence in the reality of the phenomenon and in the models.

Fitting models to various attributes, combining data and other evidence from different studies, identification of differences, resolution of apparent incompatibilities, understanding the characteristics of individual stages, interpretation of observations [ram03a,b], all raise non-trivial issues requiring further investigation. Appreciation of these challenges has not, however, undermined confidence in the results. The qualitative commonalities in the evolution of so many *E*-type systems provide a solid basis for confidence.

LAWS OF *E*-TYPE SOFTWARE EVOLUTION

The laws already mentioned, reflect the observed evolutionary behaviour of large *E*-type[3] software systems and processes implementing their evolution. They are currently stated in natural language and encapsulate aspects of the common behaviour of many disparate systems. Evolutionary forces and constraints arise from human ambitions, competitive pressures, needs for sustained profitability of the organisations involved, the limited pool of human resources and expertise available for implementing evolution. Forces relating to technological aspects such as language and tool properties, usage and availability complement these. The former appear to have a far greater influence on the evolutionary behaviour described by the laws than the latter that exemplify the specifics of the technology. This conclusion, is, at first sight, counter-intuitive. It is believed to be one of the most significant new insights to emerge from the recent studies.

Over the years the laws have been refined and extended. The changes were driven by interpretation of models of additional data that had become available. A recent public discussion [icsm02] demonstrated broad consensus as to their continuing relevance. It is not the validity of the laws that still needs demonstration but their domain of relevance. As indicated above, it is the relevance of the laws to individual paradigms and how statement of the laws needs to be adjusted to widen their applicability which requires further behavioural data and study. More detail is available in the published literature [leh74,78,80a,b,96a,97, ram02,03a,b].

The laws of software evolution were individually identified, formulated and presented over a twenty-year period. Possible relationships between them were only casually considered. It was the formulation of the FEAST hypothesis and its restatement as the eighth, Feedback System, law that suggested that the feedback nature of the process could lie at the root of such relationships. That law was seen as the cornerstone of a theoretical framework and to better understanding of the inter-relationships between the laws.

Criticism of the laws and the alleged empirical support [law82, pir88, gdf00] has been based on various grounds. As a result of the FEAST studies some of these issues are now better understood [ram02] and can now be addressed and refuted [smi02, ram03a,b].

Amongst the concerns was the question whether laws could addressed a phenomenon, software evolution, whose activity is conducted by and dependent on human intellectual processes, decision taking and implementation. Critics also suggested that it was inappropriate to term the observations *laws*. However, as indicated in footnote 2, it was pointed out [leh74] that the statement of the laws reflect phenomena beyond the immediate control of those implementing system evolution. The statements emerge from observation of behavioural phenomena in the real world of software development and evolution and reflect the attitudes and behaviours of many groups and individuals engaged directly in software creation and evolution. They also relate to managers and, via feed forward and feedback, to other stakeholders in the end product. While locally significant, they appear to have, at most, a second order influence in the long term evolutionary trajectory of the system considered as a whole. That is, the activities of individuals directly involved in evolution activities are, in general, restricted to local areas of the evolving system. The drivers that underlie observed behaviours, as described for example by the laws, stem from group, organisational and societal behaviours and the multi-level, multi-loop, multi-agent feedback information and control network that links, aggregates and constrains them. The individuals involved in the process each have their area of experience and expertise. No matter the degree of experience and understanding of individuals and groups, no matter the methods and tools used, the scope and impact of action is constrained by complex relationships between them as aggregated, mitigated, constrained and directed by the feedback network structure of the evolution process, organisational *inertia* and the related *dynamics* of evolution.

Software engineers do not, in general, have the viewpoints, knowledge, experience or time to explore potential benefits of exploiting the feedback system properties of the software process. The causes of the behaviours and phenomena addressed by the statements will, in general lie outside their range of expertise. In general, they can do little or nothing to modify the behaviours implied by the laws. The behavioural

[3] The *SPE* application and software classification scheme is now well recognised and need not be discussed here. Further details may be found in the references cited above [e.g. leh85,02a].

©2003 Society for Design and Process Science

assertions relate to market and economic forces, societal, organisational and group behaviour and, at least for the moment, must be accepted as *laws*, as forces that must be accepted.

As knowledge and understanding of the phenomenon increases and software engineering education is extended, organisational and process improvements and new forms of technology may help overcome some of the behavioural limitations and constraints reflected by the laws. As paradigms evolve and new approaches adopted, evolutionary behaviour of software systems may change. The laws as now stated may then require modification. New ones may be added, some dropped. However, it has been reasoned [e.g. leh00b], that, in the long run, such adjustments are likely to be minor.

In summary, the specifics of the laws as presently stated may be questioned but the fact that it is meaningful to formulate and acknowledge such laws and that, in practice, they must be taken into account is now widely accepted [e.g. icsm02, bau03]. The phenomena addressed by them impact process planning, control and improvement. Their role in development of software evolution theory is likely to be significant.

AN APPROACH TO THEORY FORMATION

The approach to the study of software evolution taken in FEAST and earlier work was inspired by the traditional view of the scientific method. This involves empirical observation, measurement, hypothesis testing, phenomenological interpretation, further observation to confirm or reject the interpretation an so on. This approach includes Kelvin's much quoted statement that:

"…first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it. I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be." [kel].

Given the numbers that result from such measurement, one looks for patterns, regularities, and trends that provide inputs for development of preliminary hypotheses, phenomenological and mathematical models. At any point in time, available mathematical approaches may prove inadequate. Thus scientific advances are, often, accompanied by development of new mathematical concepts and tools. Given appropriate models, one searches for interpretations in the domain of interest, refines hypotheses and extends them individually or as a set. Further observation and, when possible, real or synthetic experiments may support or reject these. Validation or rejection follows. As the number of and confidence in hypotheses builds up, one looks for relationships and develops the seeds of a theory from the collection of observations and inferences. The latter constitute the seeds of the developing theory, driving an iterative search for new data, hypotheses, refinement and theory extension.

Application of this approach to the study of software evolution is, however, limited by the paucity of data that limits statistical analysis, interpretation of experiments and scaling up the results to industrial levels.

A THEORY OF SOFTWARE EVOLUTION

For many years now it has been observed that, apart from that provided by *programming methodology* as established by the work of WG 2.3 [ifip], software engineering has no solid theoretical base [e.g. nau68, leh85, ben00]. The former is vital in guiding the structure, implementation and underlying quality of the evolving software products, the evolvability of the resulting wider system. It lies at the heart of improving the *means* whereby evolution can be effectively achieved. But, though critical, programming methodology plays a relatively local part in the process. As indicated by FEAST results and also by other observers, long-term evolvability and the pattern of evolution is likely to be heavily dependent on more global mechanisms and subject to behaviours implied by the laws. The former include forward and feedback loops and mechanisms that involve players such as business executives, other stakeholders in the total evolution process, individual and organisational users, governments and economies. All influence disciplined and sustained improvement of the global software processes, system evolvability and the direction and rate of system evolution. A sound conceptual base with predictive and explanatory power, would contribute significantly to integration of these many influences, strengthen software engineering in general and guide improvement of software evolution processes.

The history of science reflects many different ways of achieving an empirically grounded theory applicable to a natural, artificial or hybrid phenomenon. One particular approach, relevant to the search for a software evolution theory, is that emanating from Carnap's work on theory formation [car66]. The envisaged theory is to be based on observation, hypotheses and assumptions. In particular, recent wide consensus expressed regarding relevance and potential value of the laws of software evolution[4] as well as the principle of software uncertainty suggests that one may start by considering these as *empirical generalisations* as defined by him. Additional conclusions of the nounal aspects of software evolution studies should lead to a fuller set to include other generalisations of repeated real world observations of *software evolution processes* and evolution of their *products*. These can be supplemented by generalisations about the domains in which software is developed, used and evolved. Together with insight

[4] For example, as expressed recently [icsm02].

©2003 Society for Design and Process Science

and *understanding* of the phenomenology reflected and *relationships* and *dependencies* identified, they provide a basis for the development of an encompassing theory. Confirmation obtained from the developing theory by further observation of the real world then provides partial validation, increasing confidence in the validity of the developing theory to the level of detail reached and over the domain in which the observations were made. Given such a partial theory one may then seek to formalise and extend it. This may involve refinement or rejection of previously formulated empirical generalisations, identification of additional ones, derivation of implications and formalisations to produce definitions, axioms and theorems. Iteration and continuing experimentation then permits the development of a fuller, but never complete, theory accepted as valid until shown to be inadequate or incorrect. The main activities involved in the proposed theory formation approach are depicted in figure 2.

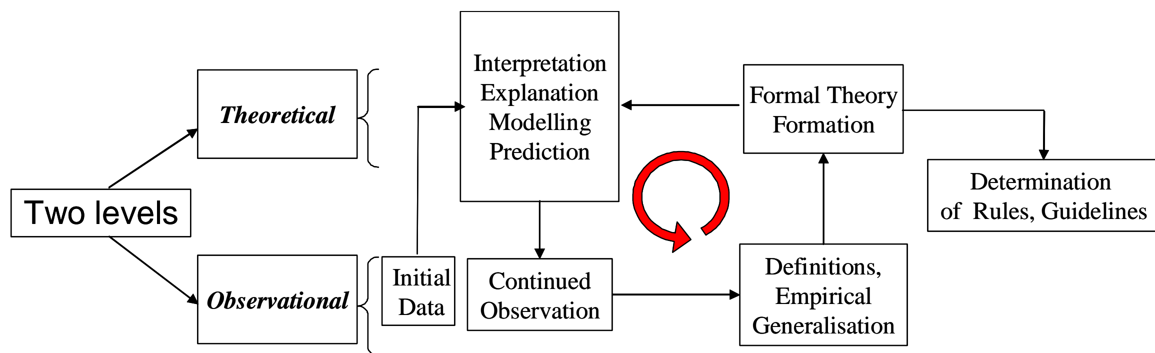

Figure 2 – An approach to the formation of a theory of software evolution

The initial step in implementation of this formation process requires formal definition of terms used, or their acceptance, at least temporarily, as undefinable, but this remains a challenge. All are candidate formalisable generalisations. Once begun, one can initiate selection and statement of axioms followed by the identification and proofs of theorems. An outline proof of the software uncertainty principle has been generated [leh01a] and will be used as an example of the approach. Its completion and formalisation awaits wider discussion, formal definition of the terms used in its statement and progress in the formation process.

Given an emerging theory, one may then formally develop its interpretations in the real world of industrial software development and derive practical implications for the planning, management, control and evaluation of system evolution, vital activities in a society ever more dependent on computers and their software. At present, improvement of these and other aspects of the software process is largely achieved informally, often intuitively. The research approach outlined above has the potential to merge the search for full *understanding* of the software evolution phenomenon and the development of effective, reliable, predictable and on time means for its achievement.

The above generalisations provide initial inputs for development of an empirical theory and its formation as a formal theory. Rooted in earlier findings, supplemented by the results of the FEAST projects, the efforts of other groups in Europe, North America and Japan, and the related insight and understanding achieved, development of a such a theory appears to be within reach [leh01a,c]. Its development could represent a first step in the development of a more general theory to support the discipline of Software Engineering. Moreover, software has long been regarded as the fruit fly (*Drosophila*) of artificial systems [sim69]. Thus such a theory could, in turn, provide an input to the development of a general theory of artificial systems [sim69] evolution. But, if at all feasible, that is many years, possibly decades, away from realisation.

EXAMPLE

Any approach to theory formation begins with formulation of a set of definitions, to be revised and extended as development proceeds. The latter is seeded by empirical *observation* of the phenomenon to be reflected. Interpretation leads to *assumptions* judged reasonable in relation to the domain being addressed. These become the source for the derivation of *inferences*. The initial set of definitions provides the base of an emerging theory that, eventually, one will seek to formalise, in part or completely. Together with the observations and assumptions they constitute axioms from which new and established implications may be formally derived as theorems. In the absence of such derivation the observations remain, at best, hypotheses. Practical application such as, for example, methods and guidelines for program development and management then be derived from all of these as corollaries.

The practical aspects of this approach to theory formation are illustrated by the lists that follow. These provide definitions, observations and implications that are believed to suffice for a formal proof of the principle of software uncertainty [leh90]. The derivation of such a proof must await more complete definition and formalisation and could not yet been undertaken.

©2003 Society for Design and Process Science

- The real world encompasses the entire universe and all happenings in it
- *E*-type operational *domains* and their attributes are, respectively, sub-domains and sub-sets of the real world and its attributes
- An *E*-type application addresses a problem or supports an activity in a specified *E*-type operational domain
- An *E*-type specification abstracts an *E*-type application, representing in a set of statements the recognised attributes required to define a solution in accord with the application needs or terms of reference
- An *E*-type program is a set of computer executable instructions defining a solution to an *E*-type application[5]
- A program is satisfactory as long as it is compatible with the solution required and the operational domains within which it is executed

List 1 – Provisional Definitions

- The real world has an unbounded number of attributes
- It is dynamic with attributes continually changing
- It may be partitioned in an unbounded number of ways into domains that, in general, each possess an unbounded number of attributes
- The designed and implemented attribute set of an *E*-type program, as distinct from the totality of attributes of such programs in execution, is necessarily bounded
- Attributes of an *E*-type application must be appropriately addressed in the implementing program to make the latter satisfactory

List 2 – Observations

Every *E*–type operational domain, though abstracted by a finite specification, has an unbounded number of attributes[6]

- By design and implementation *E*–type specifications and programs have a bounded number of attributes which reflect an unbounded number of assumptions (at least one per each unaddressed attribute)[7]
- Every *E*–type program is essentially incomplete and there will be attributes of the operational domain not addressed by it
- Assumptions about the operational domain reflected in *E*-type specifications and programs may become invalid as a consequence of changes in the real world so invalidating either one or both
- Though both are models of the same specification, an *E*–type program and its operational domain may be or may become incompatible
- *E*-type program execution entails a degree of uncertainty, sustained satisfaction cannot be guaranteed[8]
- Program evolution activity consists primarily of maintenance of the validity of the assumption set reflected in it
- Progressive change, that is evolution, of *E*-type programs is inevitable if satisfaction is to be maintained

List 3 – Inferences

[5] A program also is a model of an *E*-type specification.
[6] Some of which will change with usage and the passage of time.
[7] The assumptions issue is exemplified towards the end of the paper by three identified examples. The examples are the failure of the London Ambulance Service software, the Ariane 5 rocket disaster and the initial failure, during commissioning, of a new CERN accelerator.
[8] That is, sustained compatibility between the program and application it addresses, the domains within which it is executed.

PRINCIPLE OF SOFTWARE UNCERTAINTY

As already indicated, the proposed theory is not only of theoretical import. It, and theorems developed in it, should constitute a rich source of proposals for *improvement* of software development and evolution *processes* and for the derivation of *best practice* guidelines. This potential may be exemplified by pointing the reader to technical and management guidelines derived from the laws [leh01b] their practical implications and the principle of software uncertainty, as briefly discussed below. That analysis introduces, *inter alia*, the impact of assumptions on computer software, computer systems and their users. This topic is worthy of much wider attention than it has, with some exceptions [uch00], received. It provides just one, specific, example of the potential practical significance of the results of the studies based on the nounal interpretation of software evolution with its final outcome the formation of a conceptual framework encapsulated in a theory of software evolution. Mention of assumptions also draws attention to a neglected phenomenon with major practical consequences.

A central ingredient of an informal demonstration [leh01a] of the validity of the principle of software uncertainty, as illustrated here by the most recent outline of lists 1 – 3, is the observation that all *E*-type software has embedded within it reflections of an uncountable number of assumptions. These will have been adopted by commission or omission, consciously or unconsciously, be known or unknown documented or undocumented. Their presence and inevitability follows from the fact that any real world computer application and its operational domain each have a potentially uncountable number of properties. Having been developed by humans, with finite resources in finite time, the static software (as distinct from the software in execution) on the other hand has a finite number of attributes as determined in the design and implementation processes. As a finite model-like reflection of unbounded domains, *E*-type software is, essentially, incomplete. It reflects an unbounded number of assumptions [leh02a] generated by the abstraction process that determines system needs, requirements and a, partially explicit, partially implicit system specification from the initial real-world application concept and the subsequent design and implementation process.

Assumptions may relate to the application being addressed, to software functionality, application systems within which it executes, computer systems on which it runs, geographical, economic and societal domains on and in which it operates and which it supports, the processes by which it is produced, adapted and evolved and so on. Some will have been subjected to review. Others will be the consequences of decisions taken during system conception, specification, design, implementation, installation and operation. Others will be the result of overlooking or being ignorant of facts or situations that can affect the workings of the software,

©2003 Society for Design and Process Science

the results of execution or its impact on the operations or domains served[9] [10] [11]. Still others will have appeared entirely inconsequential or irrelevant at the time at which, implicitly or explicitly, they were adopted and embedded. Whatever the source, and even supposing that all those where validity has meaning, were valid at the time when relevant decisions were taken, they may become invalid as a consequence of changes in the domains within which the software executes, with which it interacts and which it supports. *E*-type system execution will, inevitably, be influenced to some degree by invalid assumptions. Given that the number of assumptions is uncountable, one cannot, *a priory*, know absolutely which are invalid and what the impact of such invalidity will be. The degree of satisfaction derived from execution of an *E*-type system cannot be predicted or guaranteed.

THE WIDER IMPACT OF ASSUMPTIONS

The above reasoning reflects the phenomenology underlying the principle of software uncertainty. Except for inferences 7 and 8 reasoning based on the lists 1 to 3 is, in fact, believed to suffice for a proof of the principle. This is not just of theoretical interest, but has important practical implications. Before addressing that issue, one point must be noted. In all the software systems recently examined that failed in development or use, it has been possible to show that the *underlying* cause of unsatisfactory operation or of failure was one or more implied assumptions that from the outset were unjustified, or that became invalid as a result of changes external to the software system (see footnotes to preceding section). There are good reasons to believe that this observation may, in fact, be generalised and applied to a high proportion of software, computer and computerisation project failures. They may be explained by assumptions about one aspect or another of the total computerisation process[12]. Clearly, assumptions play a critical role in the conception, birth, life and death of software systems. Hence, as many as possible must be identified, captured, questioned, confirmed and reviewed when adopted, as appropriate, thereafter. Moreover such justification must not concentrate on circumstances as they are then. The nature and direction of possible future changes, and their likelihood, must also be taken into account. Anything influencing viewpoints and decisions taken must be captured and stored in a way that will permit and trigger their review when external changes may have caused invalidity in the system. The frequency, times and extent of reviews, triggers for unscheduled reviews and so on will depend on the criticality of the application, volatility of the domains, the likelihood of domain changes, domain sensitivity to error and so on.

The real world is dynamic and many of its attributes are subject to continual change. Even conscious assumptions that were justified at the moment of adoption, can eventually become invalid. As for the unbounded unknowns who knows? Total management and control of assumptions, though the hope, is, in practice, not possible. Resource and time considerations limit the frequency and detail of assumption-database review to check for continuing validity, a need for correction. As in all engineering, compromises must be made, decisions taken and implemented. Given those cognitive and managerial constraints it must be accepted that an *E*-type system *cannot* be made or maintained absolutely valid and up-to-date.

But in current practice, industrial or otherwise conscious, explicit and continual attention to assumptions is the exception rather than the rule. For example, with one exception [uch00], the authors do not know of any specified *inspection* procedures at any process stage that calls for systematic questioning, recording of assumptions and questioning of their continued validity. We must do better than that. The management of assumptions over system lifetime must become an essential part of every software process to maximise the likelihood of sustained satisfactory operation of the software as it evolves.

The same is, of course, true of any engineering or other system development process. But at least one of the aspects in which software differs from physical and other systems relates directly to what may be termed the *assumption challenge*. Other than what is foreseen, pre-defined and precisely built into the system and its software, one cannot, in general and with current programming technology, build in code flexibility and tolerance limits that permit detection of a need to adapt the system to an external change and trigger design and implementation of the necessary fixes or changes. Concepts of absolute fit, forced fit, flexibility, tolerance and tolerance limits do not apply to software systems, except to the extent that specific needs can be foreseen procedures to address them built (coded) into the system. Logical statements, the basic constituents or bricks of the system have a precise and unambiguous meaning whose impact on system behaviour, in a specific context, is inflexible. However minor the adjustment, if it has not been foreseen it cannot be made except through human intervention. The consequences of a misfit may be irrelevant, minor, inconvenient or catastrophic. They cannot, in general, be predetermined.

Recognition of the role, largely inadvertent, played by assumptions in the exploitation of computers and in the lifecycle of *E*-type software explains a fact that has been recognised since computers came into common usage, the continuing, ubiquitous, need for, so called, software maintenance, updating and upgrading. Whenever used, computer systems must provide sound solutions to problems addressed and processes

---

[9] Consider, for example, the failure of the London Ambulance Service Computer Aided Despatch System in 1992 where ambulance drivers reactions and their inability to operate a complex interface whilst driving were overlooked in the requirements phase. There was an implicit assumption that they could [las].

[10] Another example is provided by the Ariane 5 rocket disaster which took place during its maiden flight on 4 June 1996 [ari5]

[11] A third example is the initial failure, during commissioning of the LEP accelerator in 1989 [cer98]

[12] An investigation of this hypothesis is being planned for the Software Forensic Centre at Middlesex University

©2003 Society for Design and Process Science

supported. It is stakeholder and, in particular, *user satisfaction* and *assumption set validity* that need to be *maintained*.

A software system does not, in general, adapt itself to changing situations or domains, though auto-update and upgrade mechanisms, triggered internally to download and apply software changes supplied from outside the system come close to this. In applications such as safety critical or defence systems, and to the extent permitted by economic considerations, developers can provide mechanisms which, by applying the necessary software changes, can accommodate future foreseeable external changes. But it is the domains within which software systems operate that change, often in unforeseen ways. Situations where a need and an appropriate mechanised auto-change can be anticipated, are the exception rather than the rule. Human intervention is required to maintain stakeholder satisfaction, by system *adaptation* to ensure continuing consistency that meets changing needs and desires of stakeholders and to continue to support the operational domains satisfactorily. That may be achieved by *changing* software, documentation and/or usage procedures, *evolving* the total system, not by restoring the software to its pristine beauty, as is the case when physical artefacts are maintained. In a strict sense software does not decay and, therefore, need not be maintained in the conventional sense as applied to physical artefacts. It must be *evolved* to meet new circumstances, and to remain satisfactory to its users, to the community it serves directly and to society at large.

The software uncertainty principle highlights the risks associated with reliance on software for critical real time control decisions – as in weapon systems for example. There might be, of course, reasons for doing so. But it must be understood and accepted that ignoring the inherent uncertainty that is involved and the implications of the evolutionary pressures and embedded assumptions that are intrinsic to computer systems poses challenging problems. Systems and systems of systems, that involve human activity, business (or other type of relevant) processes and software must be designed and operated as safely as possible, with software remaining the slave, not the master in the decision and implementation chains. Society ignores this fact of life at its peril.

PRACTICAL IMPLICATIONS

The text and listings that follow summarise some rules and tools for software evolution planning, management and control. Note that the items are not listed in any specific order. For more details and the derivation of the guidelines from empirical generalisations such as the laws, the reader may refer to earlier publications [leh01b]. Note, however, that progress has been made since these were published and the list provided here, though only illustrative, has been updated.

As an introduction, a general observation is important. Much of what is listed may appear self evident. Many of the items are already widely recognised and accepted as good practice. Originality is neither made nor implied for any item. Their collective listing does, however, demonstrate that the conceptual framework and the phenomenological interpretations on which it is based reflect the real world phenomenon as obtained from and described by real world observation and data. This agreement between the empirical framework and accepted good practice serves as encouraging support and strengthens the confidence in the validity of that framework. In other words, the originality does not lie in individual recommendations but in that they have been derived from and fit in with a common conceptual base and framework. It is likely that they will eventually become theorems in or follow from a theory, hopefully formal. A potential for this theory to be extended to address mentioned.

Assumptions Management

The first group of recommendations to be listed relate to assumptions, to the demonstrated fact that an unbounded number of these will be reflected in any *E*-type system. As discussed above, this empirical generalisation leads to the principle of software uncertainty. It also underlies the hypothesis that a primary source of software and software project misbehaviour or failure can ultimately be traced back to assumptions, explicit or implicit, conscious or unconscious, recorded or unrecorded, by commission or omission, that were never valid or, more likely, that have become invalid as a result of changes outside the software system.

- Identify, capture, structure, record and update all rationale, assumptions, decisions
- Institute periodic and event-triggered reviews and assessments to anticipate or identify any need for corrections to assumption set
- Review and revalidate whenever a change occurs/is made in program specification, design or implementation or occurs in operational domain
- Improve questioning of assumptions, for example, by using independent implementation, validation and inspection teams
- Make search for and questioning of assumptions in all inspections and validations a specific and required activity
- Where software operates in a rapidly changing domain or volatile environment complement detailed assumption review with re-writing of appropriate elements of the software
- Develop and use tool support for all of these activities

Evolution Management

The underlying theme of this paper is, of course, software evolution. Recommendations relating specifically to this include:

- Consistently assess and pursue anti-regressive activities [leh74] such as full documentation,

©2003 Society for Design and Process Science

complexity control and other refactoring [fow99] activities which, though having no immediate stakeholder impact, facilitate future evolvability
- Ensure that documentation includes identification and recording of assumptions
- Exploit metrics technology as below
- Assess the likelihood of functional and non-functional evolutionary trends in advance and review as part of release planning, taking system, domain volatility into account
- Involve application, domain specialists in assessment
- When validating, check interaction with, impact on unchanged parts of system and impact of assumptions
- Establish baselines of key measures over time - to support evolution planning and control

Release Management
- Observe safe change rate limits that indicate whether increments are safe, risky, unsafe as discussed in [leh01b] and that are related to standards recognised in statistical process control
- When large functional increments appear unavoidable, distribute across several releases as in Gilb's evolutionary development [gil81]
- If excessive functional increments are unavoidable, plan for follow on clean-up releases
- Follow established software engineering principles - e.g., information hiding – to minimise spread of change between system elements
- Assign appropriate resources to anti-regressive active, for example, to control complexity and its growth
- Consider the alternation of enhancement and extension with clean-up and restructuring releases
- When managing releases take the key role played by assumptions into account

To support all of the above activities one should make provision for:

Metrics and Modelling
- Develop tools to support data collection, modelling, and related activities
- For both the system and its parts, acquire, plot, model, interpret historical evolution data and determine trends, patterns, growth and their rates of change
- Progressively recalibrate and validate models as new data becomes available to reflect changes in the application, environments, process, evolution patterns
- Model and exploit the dynamics of global process to improve planning and process, identify interactions, assess and optimise policies, control strategies

FURTHER WORK

The early and more recent studies referred to in this paper concentrated on systems developed under industrial software process paradigms variants and extensions of the waterfall model [roy70]. With one exception [pir88], it was, however, not possible to investigate and relate *differences* in the details of the evolutionary patterns of individual systems to the domains in which they were developed or the type of applications in which they were used. Results from such investigation could make a major contribution to the software process improvement process. Some preliminary investigations of the evolutionary behaviour of open source software [pir88, gdf00,bau03] have been undertaken by others. But general validity of the observation that 'software evolution is driven by similar forces' in the context of newer paradigms such as object oriented, open source, agile programming and COTS-based development cannot be taken for granted. When sufficient data of such wider studies become available, results obtained to date will certainly need modification and, probably, extension. But theoretical reasoning (see for example [leh00b]) suggests that the underlying evolution phenomenon will persist. If and when this is confirmed the more extensive results will widen the scope of validity of the contemplated theory of software evolution.

In this paper, the discussion has mainly referred to the evolution of software as reflected in a *series of releases or upgrades*. Evolution that closely affects the continuing value, quality, cost and/or timeliness of software can and does, however, occur at many product and process levels [leh02c].

A fuller discussion of the levels, the role and the wider impact of evolution may be found in [leh02c]. Changes at any of these levels may have an impact on *E*-type software product properties or behaviour. Potential impact must be identified and considered during conception, specification, design, planning, management and execution of the processes that develop the products and maintain the software operationally satisfactory in a changing world. Hence, empirical study of the evolution phenomenon at all its levels is of considerable interest.

The extension of results of software-related investigations to the evolution of other artificial [sim69] systems or even, more broadly to other areas as exemplified in the introduction to this paper is likely to lead to further conclusions. But that remains to be demonstrated.

FINAL REMARK

Software evolvability, the ability, *inter alia*, for responsiveness and timely implementation of needed changes, will play an increasingly more critical role in ensuring the survival of a society ever more dependent on computers. This requires means to ensure timely adaptation of the ever more integrated computer systems to maintain compatibility between the sub-systems and the forever changing circumstances with, within and under which they operate. The goal here has been to convince the wider Computer Science and Software Engineering community that study of the software evolution phenomenon is of theoretical and practical importance and must be widely pursued.

©2003 Society for Design and Process Science

ACKNOWLEDGEMENTS

Sincere thanks are due to many colleagues and collaborators for sharing their insights, many useful discussions, and constructive criticism over more than thirty years of observation and investigation.

REFERENCES

[ari5] http://www.ima.umn.edu/~arnold/disasters/ariane5rep <as of July 2003>

[bau03] Bauer A and Pizka M, The Contribution of Free Software to Software Evolution, IWPSE 03, Amsterdam, 2 Sept, 2003

[bel71] Belady LA, Lehman MM, Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth, IBM Res. Rep., T. J. Watson Res. Centre, Yorktown Heights, NY 10598, RC 3546. Also as ch. 5 in [leh85]

[bel72] Belady LA, Lehman MM, An Introduction to Program Growth Dynamics, in W Freiburger (ed.), Statistical Computer Performance Evaluation, Academic Press, NY: 503 – 511. Also as ch. 6 in [leh85]

[bel76] Belady LA, Lehman MM, A Model of Large Program Development, IBM Sys J., vol 15, no. 3, 1976, pp. 225 – 252. Also as ch. 8 in [leh85]

[ben00] Bennett KH and Rajlich VT, Software Maintenance and Evolution: a Roadmap, in A. Finkelstein (ed.), The Future of Software Engineering, in conj. With ICSE 22, June 4-11, 2000 Limerick, Ireland

[car66] Carnap R, Philosophical Foundations of Physics, Basic Books Inc., 1966

[cer98] CERN Bulletin 09/98; 23 February 1998 http://bulletin.cern.ch/9809/art1/Text_E.html <as of July 2003>

[cox66] Cox DR and Lewis PAW, The Statistical Analysis of Series of Events, Methuen, London, 1966

[fow99] Fowler M; Refactoring: Improving the Design of Existing Code, Addison Wesley, NY, 1999, 461 pp

[gdf00] Godfrey MW, Tu Q, Evolution in Open Source Software: A Case Study, Proc. ICSM 2000, 11-14 Oct., San Jose, CA: 131 – 142

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

[icsm02] Madhavji NH, Introduction to the Panel Session Lehman's Laws of Software Evolution, in Context, Proc. ICSM 2002, Montreal, Canada: 66 – 66

[ifip] IFIP, Working Group 2.3 on Programming Methodology, http://www.ifip-tc2.org/ <as of October 2003>

[kel] Kelvin, WT, Popular Lectures and Addresses. 1891-1894

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

[las] http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las.html <as of July 2003>

[law82] Lawrence MJ, An Examination of Evolution Dynamics, Proc. ICSE 6, Tokyo, 13 - 16 Sep. 1982, pp 188 – 196

[leh69] Lehman MM, The Programming Process, IBM Res. Rep. RC 2722, Dec. 1969, 46 pp. Also as ch. 3 in [leh85]

[leh74] Lehman MM, Programs, Cities, Students, Limits to Growth?, Inaugural Lecture, in Imperial College of Science and Technology Inaugural Lecture Series, v. 9, 1970, 1974, pp. 211–229. Also in Programming Methodology, Gries D (ed.), Springer Verlag, 1978, pp. 42 – 62. Also in [leh85]

[leh78] Lehman MM, Laws of Program Evolution - Rules and Tools for Programming Management, Proc. of the Infotech State of the Art Conf., Why Software Projects Fail, Apr.: 11/1 – 11/25. Reprinted as ch. 12 in [leh85]

[leh79] Lehman MM, The Environment of Design Methodology, in Cox TA (ed.), Proc. Symp. on Formal Design Methodology, Cambridge, UK, 9-12 April 1979, pp. 17 – 18, pub. by STL Ltd, Harlow, Essex, UK, 1980

[leh80a] Lehman MM, On Understanding Laws, Evolution and Conservation in the Large Program Life-Cycle, J. Syst. and Softw., 1(3)

[leh80b] Lehman MM, Programs, Life Cycles and Laws of Software Evolution, Proc. IEEE Spec. Iss. on Software Eng, Sept.: 1060 – 1076. With more detail as "Programs, Programming and the Software Life-Cycle", in System Design, Infotech State of the Art, Rep, Se 6, No 9, Pergamon Infotech Ltd, Maidenhead, 1981: 263 – 291. Reprinted as ch. 19 in [leh85]

[leh85] Lehman MM and Belady LA, Program Evolution – Process of Software Change, Acad. Press, London, 1985

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

[leh90] Lehman MM, Uncertainty in Computer Application, Technical Letter, CACM, vol. 33, no. 5, pp. 584, May 1990

[leh94] Lehman MM, Feedback in the Software Evolution Process, Keynote Addr., CSR Eleventh Annual Workshop on Softw. Evolution: Models and Metrics, Dublin, 7-9th Sept. 1994, also in Information & Softw. Tech., sp. Iss. on Softw. Maintenance, Vol. 38, n. 11, 1996: 681 – 686

[leh96a] Lehman MM, Laws of Software Evolution Revisited, Proc. EWSPT'96, Nancy, October 1996, LNCS 1149, Springer Verlag, 1997: 108 – 124

[leh96b] Lehman MM and Stenning V, FEAST/1: Case for Support, ICSTM, DoC, EPSRC Proposal, Nov. 1995/March 1996, 11 pp

[leh97] Lehman MM., Perry DE, Ramil JF, Turski WM and Wernick P, Metrics and Laws of Software Evolution - The Nineties View, Proc. Metrics '97, Albuquerque, NM, 5 - 7 Nov. 1997: 20-32. Also as Chapter 17 in El Eman K. and Madhavji N.H. (eds.), Elements of Software Process Assessment and Improvement, IEEE CS Press, Los Alamitos, CA, 1999: 343 – 368

[leh98] Lehman MM, FEAST/2: Case for Support, DoC, Imp. Col., London, EPSRC Proposal, July 1998, 11 pp.

[leh00a] Lehman MM, Ramil JF and Kahen G, Evolution as a Noun and Evolution as a Verb, SOCE 2000 Workshop on Software and Organisation Co- evolution, Imp. Col., London, 12-13 Jul. 2000

[leh00b] Lehman MM and Ramil JF, Software Evolution Phenomenology and Component Based Software Engineering, IEE Proc. Softw., sp. issue on Component Based Software Engineering, v. 147, n. 6, Dec. 2000, pp. 249 - 255

[leh01a] Lehman MM and Ramil JF, An Approach to a Theory of Software Evolution, IWPSE 2001, Vienna, 10-11 Sept. 2001. Also in Proc. IWPSE 2001, IEEE CS Press, Los Alamitos, CA, 2002

©2003 Society for Design and Process Science

[leh01b] Lehman MM and Ramil JF, Rules and Tools for Software Evolution Planning and Management, Annals of Software Engineering, special issue on Software Management, 2001, vol. 11, pp. 15 – 44

[leh01c] Lehman MM, SETH – Approach to a Theory of Software Evolution, Case for Support, Part 2, DoC, Imperial College, London, September 2001 http://www.cs.mdx.ac.uk/staffpages/mml/seth_p2.pdf <as of July 2003>

[leh02a] Lehman MM and Ramil JF, Software Uncertainty, Soft-Ware 2002, 1st Intl. Conference on Computing in an Imperfect World, Belfast, North Ireland, 8-10 April 2002, In D Bustard, W Liu and R Sterritt (eds.), Soft-Ware 2002, LNCS 2311, 2002, pp. 174–190

[leh02b] Lehman MM and Ramil JF, An Overview of Some Lessons Learnt in FEAST, Proc. WESS'02, Montreal, 2nd Oct 2002

[leh02c] Lehman MM and Ramil JF, Software Evolution and Software Evolution Processes, Inv. Contr. to sp. iss. on Process-based Software Engineering, Annals of Softw. Eng. Nov. 2002 , vol. 14, pp. 275 – 309

[nau68] Naur P and Randell B (eds.), Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, January 1969, 231 pps

[pir88] Pirzada SS, An Statistical Examination of the Evolution of the Unix System, PhD Thesis, Dept. of Computing, Imperial College, London, 1988

[ram02] Ramil JF, Laws of Software Evolution and their Empirical Support, Invited Panel Statement, Proc. ICSM 02, Montreal, 3-6 Oct 2002: 71 – 71

[ram03a] Ramil JF, Continual Resource Estimation for Evolving Software, PhD Thesis, Dept. of Com., Imp. Col., London, January 2003

[ram03b] Ramil JF and Smith N, Qualitative Simulation of Models of Software Evolution, spec. issue on software process simulation modelling, Journal of Software Process, Improvement and Practice, to appear, 2003

[roy70] Royce WW, Managing the Development of Large Software Systems: Concepts and Techniques, Proc. WESCON, IEEE Computer Society Press, Los Alamitos, CA, 1970, reprinted in Proc. ICSE'87, Monterey, CA, March 30 - April 2, 1987

[sim69] Simon HA, The Sciences of the Artificial, 3rd. edition, The MIT Press, Cambridge, MA, 1996, 231 pp, first pub. 1969

[smi02] Smith N and Ramil JF, Qualitative Simulation of Software Evolution Processes, WESS' 02, Montreal, 2nd Oct 2002, pp. 41 – 47

[uch00] Uchitel S and Yankelevich D. Enhancing Architectural Mismatch Detection with Assumptions. Proc. of the 7th IEEE Int. Conf. on the Engineering of Computer Based Systems (ECBS 2000). Scotland, UK, April 2000

[website] http://www.cs.mdx.ac.uk/staffpages/mml/ <as of July 2003>