1

Software Evolution

Meir Lehman and Juan C. Fernández-Ramil

This chapter is a revised version of the paper by Lehman MM and Ramil JF, Software Evolution and Software Evolution Processes, Annals of Software Engineering, special issue on Software Process-based Software Engineering, vol. 14, 2002, pp. 275–309, with kind permission of Springer Science and Business Media.

1.1 Introduction

1.1.1 Evolution

Evolution describes a phenomenon that is widespread across many domains. Natural species, societies, cities, concepts, theories, ideas all evolve over time, each in its own context. The term reflects a process of *progressive*, for example beneficial, change in the attributes of the evolving entity or that of one or more of its constituent elements. What is accepted as progressive must be determined in each context.

It is also appropriate to apply the term evolution when long-term change trends are beneficial even though isolated or short sequences of changes may appear degenerative. Thus it may be regarded as the antithesis of decay. For example, an entity or a collection of entities may be said to be evolving if their value or fitness is increasing over time; Individually or collectively they are becoming more meaningful, more complete or more adapted to a changing environment.

In most situations, evolution results from concurrent changes in several, even many, of the properties of the evolving entity or collection of entities. Individual changes are generally small relative to the entity as a whole, but even then their impact may be significant. In areas such as software, many allegedly independent changes may be implemented in parallel. As changes occur as a part of the overall evolution, properties no longer appropriate may be removed or may disappear and new properties may emerge.

The evolution phenomena as observed in different domains vary widely. To distinguish between domains, one may start by classifying them according to their most evident characteristics. A study of common factors shared by subsets of their entities, distinctions between them and their individual evolutionary patterns may suggest specific relationships

Software Evolution and Feedback: Theory and Practice Nazim H. Madhavji, Juan C. Fernández-Ramil and Dewayne E. Perry

between evolution and other properties and indicate how individual patterns and trends are driven, directed and even controlled.

One could, perhaps, increase understanding of software evolution by studying instances of the phenomenon in other domains. The discussion here is, however, limited to the computing and software fields.

1.1.2 Interpretation of the Term Evolution in the Context of Software

The term *evolution in the context of software* may be interpreted in two distinct ways, discussed more fully in Chapter 16 [Lehman and Ramil 2001b]. The most widespread view is that the important evolution issues in software engineering are those that concern the *means* whereby it may be directed, implemented and controlled. Matters deserving attention and the investment of resources relate to *methods, tools* and *activities* whereby software and the systems it controls may be implemented from conception to realisation and usage, and then evolved to adapt it to changing operational environments. One is seeking continuing satisfactory execution with maximum confidence in the results at minimum cost and delay in a changing world.

Means include mechanisms and tools whereby evolution may be achieved according to plan in a systematic and controlled manner. The focus of this approach, termed the *verbal* approach, is on the *how* of software evolution. Work addressing these issues has been widely presented and discussed, for example, at a series of meetings titled Principles of Software Evolution (e.g. IWPSE 2004).

An alternative approach may also be taken. This less common, but equally, important view seeks an understanding of the *nature* of the evolution phenomenon, what drives it, its impact, and so on. It is a *nounal* view investigating the *what* and *why* of evolution. Far fewer investigators (e.g. Lehman *et al*. 1969–2002, Chong Hok Yuen 1981, Kemerer and Slaughter 1999, Antón and Potts 2001, Nanda and Madhavji 2002, Capiluppi *et al*. 2004) have adopted it. It is driven by the realisation that more insight into and better understanding of the evolution phenomenon must lead to improved methods and tools for its planning, management and implementation. It will, for example, help identify areas in which research effort is most likely to yield significant benefit. The need for *understanding* and its significance will become clearer when the nature of, at least, the industrial software evolution process as a multi-loop, multi-level, multi-agent feedback system (Lehman 1994) is appreciated. Failure to fully appreciate that fact and its consequences can result in unexpected, even anti-intuitive responses when software is executed and used.

There is a view that the term evolution should be restricted to software *change* (e.g. Mittermeir 2006). However, under this interpretation, important activities such as defect fixing, functional extension and restructuring would be implicitly excluded. Other authors have interpreted evolution as a stage in the operational lifetime of a software system, intermediate between initial implementation and a stage called *servicing* (Bennett and Rajlich 2000, Rajlich and Bennett 2000). These and still other interpretations are covered by the areas of evolution presented below. They are, therefore, not separately identified in the present chapter.

1.2 The Evolution of Large Software Systems

1.2.1 Early Work

As stated in Lehman's first law of software evolution, (Lehman 1974), it is now generally accepted (e.g. Bennett and Rajlich 2000, Pfleeger 2001, Cook *et al*. 2006) that

E-type[1] software must be continually adapted and changed if it is to remain satisfactory in use. Universal experience that software requires continual *maintenance* (as evolution was then termed) was first publicly discussed at the Garmisch Conference[2] (Naur and Randell 1968) and viewed as a matter of serious concern.

At about that time, Lehman reported on his study of the IBM programming process (Lehman 1969), though his report did not become generally available till much later (Lehman and Belady 1985). *Inter alia*, the report examined and modelled the continuing change process being applied to IBM's OS360-370 operating system. Preliminary models of that system's evolution were derived from measures and models of release properties. Refined versions of these were subsequently proposed as tools for planning, management and control of sequences of releases (Belady and Lehman 1972, Lehman 1974, 1980).

Recognition of the software process as a feedback system brought the realisation that the study of the process and its evolution must consider that fact, if more effective management and process improvement was to be achieved. This observation triggered an investigation of the phenomenon initially termed *Program Growth Dynamics* (Belady and Lehman 1972) and later *Program Evolution Dynamics* (Lehman 1974). The resultant study produced not only fundamental insights into the nature and properties of the software *process* but also into those of its products. Early studies concentrated on OS360-370 release data; later studies involved other systems (Lehman 1980, Lehman and Belady 1985). All in all, the results of these studies greatly increased understanding of the software evolution phenomenon and identified practices and tools for its support (Lehman 1980).

1.2.2 Large Programs

Lehman and Belady's early work on software growth dynamics and evolution concluded that evolution is intrinsic to *large* programs. This adjective has been variously interpreted as applying to programs ranging in size from 50 and 500 thousand of lines of code (Klocs). Subsequently, Lehman suggested that such an arbitrary boundary was not very useful in the evolution context. It appeared highly unlikely that one could identify, even approximately, a single bound over a spectrum of programs such that those on either side of the divide displayed different properties. Moreover, if size were a major factor in determining evolutionary properties, one would expect these to change for programs of different size.

Moreover, it was seen as unlikely that all would appear at around the same loc level, independently of, for example, application, organisational, managerial, process and computational factors. Any of these might relate to the emergence of disciplined evolutionary behaviour. As a result of considerations such as these, Lehman suggested that the observed phenomena were more likely to be linked to properties related to characteristics of software development, usage and application environments and processes or of their products. He, therefore proposed that a program should be termed *large* if '... it had been developed or maintained in a management structure involving at least two groups' (Lehman 1979), that is, subject to, at least, two levels of direct management. This property appeared sufficient to explain many of the observed evolution dynamics properties of the systems studied.

[1] Defined later in this chapter.

[2] See statement by H. R. Gillette in the Garmisch conference report, P. Naur and B. Randell eds. (1968), p. 111 in the original version.

This definition followed from the recognition of the fact that development by an individual or small group subject to the direct control of a single individual, is quite different to one in which there are two or more management levels. When a single manager is in day-to-day control the focus of goals and activities will be a matter of ongoing discussion and decision within the group, subject to a final approval by the manager. With two or more groups and managers at two or more levels of management, each level, each manager, each group will develop its individual goals, understanding, language, interpretations etc. Communication between the members of any individual group will tend to be continual and informal. Between groups and levels it will tend to be discontinuous and more formal. This will cause divergence of the terminologies, technologies, interpretations, goals, and so on as perceived and applied in and by the separate groups. Such divergence is clearly a major source of the 'large program problem' (Brooks 1975) and that problem, in turn, appears to be one of the drivers of software evolution. It must also be recognised that in cooperative multi-group activity, it is human nature for individual groups and their managers to seek to optimise their own immediate results, overlooking or ignoring the impact on other groups and on the overall, long-term consequence.

Furthermore, programs developed by the joint effort of multiple groups are functionally rich and structurally complex. Their effective development and use requires the application and integration of many skills and approaches and communication between participants. It was thought at one time (Lehman 1979) that the resultant activities will favour the emergence of evolutionary characteristics associated with programs that have traditionally been termed *large*. This further supported the above definition of *largeness*. However, the latter was still considered unsatisfying as a complete explanation for the intrinsic need for software evolution.

1.3 Program Classification

1.3.1 The SPE Program Classification Schema

Despite the revised definition, the concept of *largeness* appeared unsatisfactory as the fundamental basis for a study of software evolution. To address these concerns, a program classification scheme, not involving a concept of size was proposed. Initially, this defined programs of types *S*, *P* and *E* (Lehman 1980, 1982, Pfleeger 2001) as discussed below. The third is the most closely related to a discussion of software evolution. Though not a *defining* property of the phenomenon, it has been shown as inevitable for that class of program if users are to remain satisfied with the results of its use. Subsequently, it was realised that the classification is equally relevant to computer applications, application domains, application and computing systems and so on (Lehman 1991).

1.3.2 S-type Applications and Software

1.3.2.1 Definition

A program is defined as being of type *S* if it can be shown that it satisfies the necessary and sufficient condition that it is *correct* in the full mathematical sense relative to a pre-stated formal *specification* (Lehman 1980, 1982). Thus a demonstration, by means of *proof* for example, that it satisfies the specification (Hoare 1969, 1971), suffices for contractual completion and program acceptance. Where it is possible, for example with the exception of systems where decidability issues arise (Apt and Kozen 1986), demonstration of

correctness is also a matter of mathematical skill and the availability of appropriate tools. The proof demonstrates that program properties satisfy the specification *in its entirety*. Such verification suffices for program acceptability if the specification is satisfactory to intended users and meets their requirements. That is, the specification will have been validated and accepted. Completion of verification then justifies contractual acceptance of the program.

The definition assumes implicitly that a specification can be predetermined before development begins and that, once fixed, learning during the course of the subsequent process is restricted to determination of methods of solution and the choice of a *best* method (in the context of constraints applying in the solution domain). Implementation is driven purely by the implementers' knowledge, understanding and experience.

The designation *S* was applied to *S*-type systems to indicate the role played by the *specification* in determining product properties.

1.3.2.2 Validation

The above implies that verification with respect to the specification completes the *S*-type development process. If satisfaction of the specification by the final program product is (contractually) accepted as sufficient by both developer and client, verification leads directly to acceptance. Practical application of the *S*-type development process, however, requires that the specification is *valid* in the context of its intented use. Validation of a specification is, in general, nontrivial.

1.3.2.3 The *S*-type in a Changing Domain

Even if initially satisfactory, changes in the use of an *S*-type program or in its operational environments or circumstances can cause it to become unsatisfactory. In this event, the specification, the problem or both must be revised. By definition, this means that a *new* program based on a *new* specification is being implemented. However, the new derivation is likely to be based on previous versions of the specification and program, that is, the latter are modified rather than recreated. Conceptually, however, evolution of *S*-type programs is restricted to the initial development. It consists of a discrete sequence of processes each of which includes specification revision, program derivation and verification.

1.3.2.4 Formal Specification

Application of the *S*-type concept is limited to formally specifiable problems. It also requires that a procedure for computation of the solution is known or can be developed within budgetary and time constraints. In other words, there are four conditions that the *S*-type program must satisfy in order to be legitimately termed as such. First, the problem can be rigorously, that is, formally, stated. Second, the problem must be solvable algorithmically. Third, it must be feasible to prove that the program is correct with respect to the formal specification. Last, but not least, the specification must be complete, that is, final for the moment (see Section 1.3.2.3), in terms of the stakeholders' current requirements. It must explicitly state *all* functional and nonfunctional requirements of concern to the stakeholders and, in particular, clients and users. Nonfunctional requirements include the range and precision of variables, maximum storage space, execution time limits, and so on.

1.3.2.5 The *S*-type in Practice

S-type domains are exemplified by, though not restricted to, those in which it is required to compute values of mathematical functions or formally defined transformations as, for example, in program compilers or proof procedures. This is so because in these domains the development process may be followed in its purest form. Its use in other domains is more limited, but nevertheless retains both theoretical and practical importance. Its theoretical importance arises from the fact that the *S*-type represents an ideal, specification-driven, development process where the developers exercise maximum intellectual control on the program properties of interest. One example of its practical importance is briefly discussed in Section 1.3.3.2. In the vast majority of domains, however, the *S*-type program process cannot be implemented for a variety of reasons: The most common, the difficulty of creating a formal specification which is complete and final, in the sense implied above. It is then that the *E*-type discussed below becomes relevant.

1.3.3 E-type Applications and Software

1.3.3.1 Definition

Type *E* programs were originally defined as 'programs that mechanise a human or societal activity' (Lehman 1980). The definition was subsequently amended to include all programs that 'operate in or address a problem or activity of the real world'.

A key property of the type is that the system becomes an integral part of the domains within which it operates and that it addresses. It must reflect within itself all those properties of the domains that in any way affect the outcome of computations. Thus, to remain satisfactory as applications, domains and their properties change, *E*-type programs must be continually changed and updated. They must be *E*volved. Software evolution is a direct consequence and reflection of ongoing changes in a dynamic real world. Operating systems, databases, transaction systems, control systems are all instances of the type, even though they may include elements that are of type *S* in isolation.

1.3.3.2 *S*-Type Programs in the Real World

S-type elements can also contribute greatly to an *E*-type system despite the fact that it is addressing a real-world application. Given the appropriate circumstances, their use can provide important quality and evolvability benefits. Once embedded, they will, of course be subject to all the evolutionary pressures that the host system is subject to, even though shielded by other system elements. As the latter are changed to reflect an evolving application, changing application and operational domains (hard and soft) under which it operates, the *S*-type program will also require adaptation by changes to its specification and, possibly, its interfaces. One cannot always expect it to remain static, a matter that is particularly important in considering component-based architectures and the use of components typically termed *Commercial Off-The-Shelf* (COTS) (Lehman and Ramil 2000b).

1.3.3.3 Domain and System Bounds

The number of properties of an *E*-type application and of the domains in which it is developed, evolved, operated, executed and used is unbounded. Clearly they cannot be

explicitly identified, enumerated or uniquely defined. Hence, selection of those to be reflected in the system requires abstraction. Properties and behaviours considered irrelevant in the circumstances or to domains of interest will be discarded. Their exclusion may be explicit or implicit, conscious or unconscious, by commission or omission, recorded or unrecorded, momentarily valid or invalid. Those excluded will be unbounded in number since only a bounded number can be addressed and adopted. Moreover, each exclusion involves at least one assumption[3]. To complicate matters, the practical bounds of the many domains involved will, in general, be fuzzy and will change as knowledge and deeper understanding of the application, operational domains and acceptable solutions accumulate during development and as the intended application and the operational domains evolve. As discussed in the next section, feedback plays a central role in this process.

1.3.4 P-type Situations and Software

A further class, type *P*, was also defined (Lehman 1980). The type was conceived as addressing problems that appear to be fully specifiable but where the users' *concern* is with the correctness of the results of execution in the domains where they are to be *used* rather than being relative to a specification. Such programs will clearly satisfy the definition of one or other of the other types. Hence, in the context of the present discussion, their separate classification is redundant. However Cook *et al*. have recently proposed a redefinition of the type *P*, conceptually faithful to the initial description of the classification, but making the type *P* distinct from the other two types (Cook *et al*. 2006).

1.4 The Inevitability of Evolution

The *intrinsic* evolutionary nature of real-world computer usage (Lehman 1991) and, hence, of *E*-type software was recognised long ago (e.g. Lehman 1980, Lehman and Belady 1985, Lehman 1991, 1994). Continual correction, adaptation, enhancement and extension of any system operating in the real world was clearly necessary to ensure that it adequately reflected the state at the time of execution of all application and domain properties which influenced the real-world outcome of the problem being solved or the application being supported. It was also self evident that such change or evolution must be planned, directed and managed.

Information on the evolution of a variety of systems of differing sizes, from different application areas, developed in significantly different industrial organisations and with distinct user populations has been acquired over many years (e.g. Lehman and Parr 1976, Lehman and Belady 1985, FEAST 2001). From the very start the study demonstrated that software evolution is a phenomenon that can be observed, measured and analysed (Lehman 1980), with *feedback* playing a major role in determining the behaviour (Belady and Lehman 1972). A more complete picture and wider implications became clear over a longer period (Lehman 1994).

Figure 1.1 is the original example of supporting evidence showing a steady OS/360-370 growth trend with a superimposed ripple. The latter was interpreted as indicating feedback stabilisation and constituted the source of the suggestion that feedback plays a major role

[3] See Chapter 16 (Lehman and Ramil 2001b) in this book for a further discussion on the topic of assumptions.

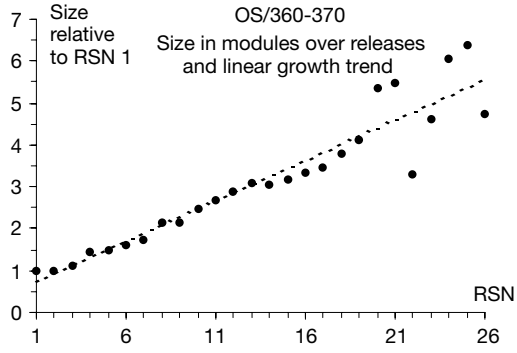

Figure 1.1 The growth of OS/360-370 over releases as a function of release sequence number (RSN)

in controlling software growth. The growth pattern following the release with sequence number 20 reinforced this conclusion being typical of the behaviour of a system[4] with excessive positive feedback. The excessive feedback here was reflected by a growth rate from RSN 20 to RSN 21, more than three times as great as any previously observed. Similar behaviour was also observed in the other systems studied (FEAST 2001), though with differences in detail.

All in all, the observations and measurements over the years on many systems confirm and advance the 1971 hypothesis (Belady and Lehman 1972) that in the long term '... the rate of growth of a system is self-regulatory, despite the fact that over the years many different causes control the selection of work implemented in each release, budgets vary, number of users reporting faults or desiring new function change, economic conditions vary and management attitudes towards system enhancement, frequency of releases and improving methodology and tool support all change'.

The feedback observation was formalised in the FEAST (*F*eedback, *E*volution *A*nd *S*oftware *T*echnology) hypothesis (Lehman 1994, FEAST 2001). This states that, in general and certainly for *mature*[5] processes, software evolution processes are multi-agent, multi-level, multi-loop feedback systems. They must be seen and treated as such if sustained improvement is to be achieved. Implications of the hypothesis have been discussed in a number of publications (FEAST 2001).

1.5 Levels of Software-Related Evolution

Evolution phenomena in software-related domains are not confined to programs and related artefacts such as specifications, designs and documentation. Applications, definitions, goals, paradigms, algorithms, languages, usage practices, the sub-processes and processes of software evolution and so on, also evolve. These evolving entities interact, impact and affect one another. If their evolution is to be disciplined, the respective evolution processes must be planned, driven and controlled. To be mastered, they must be understood and mastered individually and jointly.

[4] In general, a feedback system is a system in which the output modifies its input.

[5] For a discussion of the process *maturity* concept and its practical assessment see Paulk *et al*. (1993) and Zahran (1997).

In the first instance, however, one must focus on individual aspects. The consequences of interactions between the various levels of evolution require more insight than is presently available. It is mentioned here only in passing, even though it is a topic that requires further investigation.

Further discussion of software evolution is ordered by a simple classification scheme summarised below and discussed in more detail in the following sections:

I. The *development* process implements a new program or *software system* or applies *changes* to an existing system. On the basis of some identified need or desire, it yields a new artefact. The stimuli and feedback mechanisms that drive and direct this process yield gradual evolution of the application and its implementing system to adapt them to a changing environment with changing needs, opportunities and desires.

   At the start of an *E*-type system development, knowledge and understanding of the details of the application to be supported or the problem to be solved and of approaches and methods for their solution are often undefined, even arbitrary (Turski 1981). The relative benefits of alternatives often cannot be established except through trials. Results of the latter are unlikely to be comprehensive or conclusive. The development process is a learning process in many dimensions that includes both the *matter* being addressed and the *manner* in which it is addressed. Feedback from development, change experience and evaluation of results drive the evolution process.

II. At a somewhat higher level, consider a *sequence* of *versions, releases* or *upgrades* of a program or software system each of which is the output of such a process. These incorporate changes that rectify or remove defects, implement desired improvements or extensions to system functionality, performance, quality and so on. These are made available to users by means of what is commonly termed a *release* process (Basili *et al*. 1996).

   Generally intended to produce improvements to the program, the release process is often referred to as program *maintenance*. Over the years, however, it has been recognised that the term is inappropriate, even misleading, in the software context. After all, in other contexts, the term describes an activity that, in general, rectifies aging, wear, tear and other deterioration that has developed in an artefact. The purpose is to return the latter as closely as possible to a former, even pristine, state. But software as such is not subject to wear and tear. In itself, it does not deteriorate. The deterioration that software users and others sense is due to changes in its environment, in the purpose for which it was acquired, the properties of the application, those of the operational domains and the emergence of competitive products. Deterioration or misbehaviour can often be associated with assumptions implicitly or explicitly reflected in the software. These would have become invalid as a result of such external changes.

   Thus one must accept that in the software context, the term *maintenance* is incompatible with common usage. What happens with software is that it is changed or adapted to *maintain* it satisfactorily in changed domains and under new circumstances as judged by stakeholders such as users. Software is *evolved* to *maintain* embedded assumptions and its compatibility valid with respect to the world as it is now. Only in this sense, is the use of the term *maintenance* appropriate in the software context.

III. The areas supported by *E*-type software also evolve. *Activities* in these may range from pure computation to embedded computers to cooperative computer-supported integrated human-machine activity. We refer to such activities generically as *application* areas. Introduction to use of successive software versions by the user community as in II inevitably changes the activity supported. It also changes the operational domain. Changes may be driven and include needs, opportunities, functionality, procedures and so on. In general, they require further changes to the system to achieve satisfactory operation. Installation and operation of an *E*-type system, drives an unending process of joint system and application evolution.

IV. The *process* of software evolution also evolves. The term refers to the aggregate of all activities involved in implementing evolution in any of the above levels. It is variously estimated that between 60 and 95% of lifetime expenditure on a software system is incurred after first release (Pigoski 1996), that is, in area II evolution (can even exceed 95% in, for example, defence applications). Hence, there is good reason to improve the process of evolution, to achieve lower costs, improved quality and faster response to user needs for change and so on.

    Human dependence on computers and on the software that gives them functional and computational power is increasing at ever growing rates. Process improvement is also essential to reduce societal exposure to the consequences of high costs, computer malfunction and delays in adaptation to changing circumstances. All these and many other causes demand improvement of the means whereby evolution is achieved. And the improvement achieved must produce gains in areas such as quality, cost and response times in meeting the needs of the application areas and domains concerned. The process evolves, driven by experience and technological advances.

V. The software evolution process is a complex multi-loop feedback system[6]. Achieving full understanding and mastery of it remains a distant goal. *Modelling*, using a variety of approaches, is an essential tool for study, control and improvement of the process (Potts 1984). *Models* facilitate reasoning about it, exploration of alternatives and assessment of the impact of change, for example. As the process and understanding of it evolve, so must its models.

1.6 *Ab Initio* Implementation or Change

1.6.1 Process Steps

Ab initio implementation of a program or changes to an existing program is achieved by interacting individuals and teams in a series of discrete steps, using a variety of, generally computer-based tools. Their joint action over a period of weeks, months or even years produces the desired program or a new *version* or *release* of an existing program. The many steps or stages in such development differ widely. The first published model of the software process, the Waterfall model (Royce 1970) and its subsequent refinements (e.g. Boehm 1976, 1988), used terms such as *requirements development*, *specification*, *high-level design*, *detailed design*, *coding*, *unit test*, *integration*, *system test*, *documentation* and so on to describe these activities.

[6] In a multi-loop feedback system, the inputs are influenced by the outputs by many different routes or 'loops'.

Their execution is not purely sequential. Overlapping and iteration between steps in reaction to feedback or changes external to the system are inevitable as, for a variety of reasons, is repetition. Thus, execution of any step may reveal an error in an earlier step, suggest an improvement to the detailed design or reveal the impact of an underlying assumption that requires attention. The latter may relate to the application, a procedure being implemented, the current realisation, domain characteristics and so on.

Steps will, normally, operate at different conceptual and linguistic levels of abstraction and will require alternative transformation techniques. Their aggregated impact is that of a refinement process that systematically transforms an application concept into an operational software system. Program development was indeed recognised and termed *successive refinement* by Wirth (Wirth 1971). Thus even this process may be viewed as *evolutionary* because it progressively evolves the application concept to gradually produce the desired program. At the process level, it is conceptually equivalent to a process known as the *LST transformation*.

1.6.2 The LST Paradigm

The LST process, was described by its authors (Lehman *et al.* 1984) as a sequence of *transformation* steps driven by human creative and analytic power and moderated by developing experience, insight and understanding. At first sight, the paradigm may be considered abstract and remote from the complex reality of industrial software processes. This is, however, far from the truth. A brief description will suffice to clarify this in the context of the *practical* significance of the *SPE* classification (described in Section 1.3) and reveal some issues that emerge during *ab initio* software development.

LST views each step of the implementation process as the transformation of a specification into a model of that specification, in other words, of a design into an implementation. The transformation steps include verification, a demonstration that the relationship between the implemented output and the specification is *correct* in the strict mathematical sense. In this form, it is, therefore, only applicable to *S*-type applications where the formal specification, can be *complete* and, by definition, express *all* the properties the program is required to possess to be deemed satisfactory and acceptable. Only in this context is mathematical *correctness* meaningful and relevant.

The paradigm, however, also requires a process of *validation* – termed *beauty contest* in the LST paper – to complete each step. It is needed to confirm (or otherwise) at each stage of refinement that the process is heading towards a product that will satisfy the *purpose* for which it is being developed. The model fails validation if some weakness or defect is revealed, which implies that the final product is unlikely to be satisfactory in the context of the intended purpose. Unsatisfactory features may have arisen during transformation by the introduction of properties undesirable in the context of the intended purpose though not excluded by the current specification. Such features may even prove to be incompatible with the purpose, their nonexclusion by the specification reflecting an oversight or error in the latter.

The source of validation failure must be identified and rectified by modification of the specification. That is, the previous specification must be replaced by a new one[7]. When

[7] Though, in practice, it may be derived by modification of a previous version.

both verification and validation are successful, the new model becomes the specification of the next transformational refinement step and the process continues.

Verification is a powerful tool where applicable but can only be applied to completely and formally specified elements. It will be shown below that for programs operating in and addressing real-world applications in real-world domains, their properties cannot all be formally or completely specified. Hence the pure LST process cannot be used. Individually and collectively, however, these nonformalisable properties influence the computational process, its behaviour and its outputs and contribute to the level of user satisfaction and program quality. As already observed, it is the satisfaction with the results of program execution that concerns *E*-type users, not the correctness of the software. Without verification, validation becomes even more crucial. The process whereby they are implemented is, at best, a pseudo-LST process.

This distinction leads directly to a further observation relating to the use of component-based architectures, reuse and COTS. The benefits these are expected to yield implicitly assume that the elements are correct with respect to a stated specification. In a malleable, evolutionary *E*-type domain, *S*-type components must be maintained compatible with all of the domains in which they operate and are embedded (Lehman and Ramil 2000b); their specification must be continually updated. This is not straightforward. As Turski has affirmed '... the problem of adopting existing software to evolving specifications remains largely unsolved, perhaps is algorithmically not solvable in full generality ...' (Turski 2000). In the real world of constant change and evolving systems, reliance on the use of standardised components, reuse and COTS is difficult and hazardous, likely to negate the benefits of their alleged use.

1.6.3 Phenomenological Analysis of Real-World Computer Usage

Clearly, pseudo-LST process cannot be *guaranteed* to produce a program that is satisfactory whenever executed. This observation reflects the nature of the real world and of people. Satisfaction depends upon the state of the former and the needs, desires, reactions and judgements of the latter when *using* the results of execution. Relative to a world that is forever changing, formal specification and demonstration of correctness where applicable, is bound to the period at which the specification was developed and accepted. Behaviour considered satisfactory even yesterday may not meet the conditions, needs and desires of today. Later satisfaction cannot be *guaranteed* unless it is demonstrated that the definitions, values and assumptions underlying the formulation and correctness demonstration are still valid. *Testing* and other means of validation may increase *confidence* in the likelihood of satisfaction from subsequent execution. But even this is not absolute, As Dijkstra said 'Testing can only demonstrate the presence of defects, never their absence' (Dijkstra 1972b). In the real world of 'now' a claim of demonstrated correctness (even in its everyday sense) of an *E*-type program with respect to the specification as it was, is, at best, a statement about the likelihood of satisfaction from subsequent execution. Any assertion of *absolute* or *lasting*, satisfaction is meaningless.

1.6.4 Theoretical Underpinning

The above reasoning is phenomenological. Closer examination provides a basis for formalising its conclusions. Programs and their specifications are products of human activity.

As such, they are essentially *bounded* in themselves and in the number of real-world properties that they reflect. Real-world applications and domains are themselves unbounded in the number of their properties. Specifications and programs therefore, cannot reflect them in their entirety. Knowingly and unknowingly, an unbounded number of real-world properties are discarded during the abstraction that produces the specification and permeates the subsequent development process. Moreover, each abstraction involves at least one assumption. An unbounded number of assumptions are therefore reflected in any E-type system (and in each of its E-type elements). Moreover, assumptions reflected in the system may become invalid, for example, as discarded properties become relevant. Ignoring this possibility adds further assumptions. Admittedly, most of the assumptions embedded in the system will be and remain totally irrelevant, but some will inevitably become irritants very possibly error or other, misbehaviour. All program elements that reflect such assumptions will require rectification. However carefully and to whatever detail software specifications and their implementations are developed the time for which they remain valid will be limited.

Contractually, one may be able to protect the developers from responsibility for resultant failure to achieve satisfactory results. Users will, in general, be unaware of the fact that the program can only address foreseen changes that permit corrective procedures to be included in the software and/or usage procedures. Usage will be judged as satisfactory or otherwise on the basis of the results of execution but depends on the properties the program *has*, not those it *should* have to satisfy and reflect the *current* states of the application and the operational domains. Even in special cases where a real-world program has and is correct against a formal specification, the use of the term correctness of a bounded program relative to an unbounded domain is wrong. Formal *correctness* of a program or system has only limited value.

1.6.5 The Value of Formalisms and of Verification

Nevertheless, formalisms and specifications can play an important role in the development and evolution of E-type applications (van Lamsweerde 2000). Other than momentarily, systems, software or otherwise, cannot, in general, be better, than the foundations on which they are built. A demonstrably *correct* element does not provide any permanent indication that the system as a whole is valid or will be satisfactory to its users. Nor can correctness prove that a specification on which the demonstration is based is sufficient or correct to ensure satisfactory operation. But the greater the number of system elements that can be shown to be correct relative to a precise and complete specification, the greater the likelihood that the system will prove to be satisfactory, at least for a while. Demonstration, by whatever means, of the correctness of an element with respect to its specification can assist in the isolation, characterisation and minimisation of uncertainties and inconsistencies (Lehman 1989, 1990). It will then also assist systematic and controlled evolution of the system and its parts as and when required.

Some researchers have highlighted the need to accompany a formal specification with a precise, informal definition of its interpretation in the domains of interest (van Lamsweerde 2000). The systematic development and maintenance of these is a worthwhile activity in the context of E-type evolution. It is referred to briefly later in the next section when the role of assumptions is addressed.

1.6.6 Bounding

Abstraction is a bounding process. It determines the operational range of E-type systems. The bounds required for such systems are, generally, imprecise, even unclear and subject to change. Some of the boundaries will be well defined by prior practice or related experience, for example. Others are adopted on the basis of compromise or recognised constraints. Still others will be uncertain, undecidable or verging on the inconsistent. This situation may be explicitly acknowledged or remain unrecognised until exposed by chance or during system operation. Since applications and the domains to which they apply and in which they operate are dynamic, always changing, and E-type system (in particular) must be continually reviewed and, where necessary, changed to ensure continuing validity of execution results in all situations that may legitimately arise.

In the context of evolution, *fuzziness* of bounds arises from several sources. The first relates to the, in general, unlimited, number of potential application properties from which those to be implemented and supported must be selected. The detail of system functional and nonfunctional properties and system behaviour also cannot be uniquely determined. A limited set must be selected for implementation on the basis of the current state of knowledge and understanding, experience, managerial and legal directives and so on.

Precise bounds of the operational domains are, in general, equally undetermined. The uncertainty is overcome by provisionally selecting boundaries within which the system is to operate to provide satisfactory solutions at some level of precision and reliability, in some defined time frame, at acceptable cost. Inevitably however, once a system is operational, the need or desire to change or extend the area of validity, whether of domains or of behaviours, will inevitably arise. Without such changes, exclusions will become performance inhibitors, irritants and sources of system failure. In summary, the potential set of properties and capabilities to be included in a system is, in general, unbounded and not uniquely selectable. Even a set that appears reasonably complete may well exceed what can be accommodated within the resources and time allocated for system implementation. As implemented, system boundaries will be arbitrary, largely determined by many individual and group decision makers. Inevitably, the system will need to be continually evolved by modifying or extending domains it defines, and explicitly or implicitly assumes, so as to satisfy changing constraints, newly emerging needs or changed environmental circumstances.

But, unlike those of the domain, once developed and installed, system boundaries become solid, increasingly difficult and costly to change, interpret and respect, fault prone, slow to modify. A user requiring a facility not provided by the system may, in the first instance, use stand-alone software to satisfy individual or local needs. This may be followed by direct coupling of such software tightly to the system for greater convenience in cooperative execution. But problems such as additional execution overhead, time delays, performance and reliability penalties and sources of error will emerge, however the desired or required function is invoked and the results of execution passed to the main system. Omissions become onerous; a source of performance inhibitors and user dissatisfaction. A request for system extension will eventually follow.

The history of automatic computation is rich with examples of functions first developed and exploited as stand-alone application software, migrating inwards to become, at least conceptually, part of an operating or run time system and ultimately integrated into

some larger application system or, at the other extreme, into hardware (chips). This is exemplified by the history of language and graphics support. The evolving computing system is an *expanding universe* with an inward drift of function from the domains to the core of the system. The drift is driven by feedback about the effectiveness, strengths, weaknesses, precision, convenience and potential of the system as recognised during its use and the application of results.

1.6.7 The Consequence: Continual System Evolution

Properties such as those mentioned make implementation and use of an *E*-type system a learning experience. Its evolution is driven, in part, by the ongoing experiences of those that interact with or use the results of execution directly or indirectly, of those who observe, experience or are affected by its use as well as those who develop or maintain it. The system must reflect any and all properties and behaviours of the application being implemented or supported, the domains in which the application is being executed, pursued and supported, and everything that affects the results of execution. It must be a *model-like reflection*[8] of the application and its many operational domains.

However as repeatedly observed, the latter are unbounded in the number of their properties. They, therefore, cannot be known entirely by humans during the conscious and unconscious abstraction and reification decisions that occur from conception onwards. The learning resulting from development, use and evolution plays a decisive role in the changes that must be implemented throughout its lifetime in the nature and pattern of its inevitable evolution.

Evolution of *E*-type applications, systems, software and usage practices is clearly intrinsic to computer usage. Serious software suppliers and users experience the phenomenon as a continuing need to acquire successive versions, releases and upgrades of used software to ensure that the system maintains its validity, applicability, viability and value in an ever-changing world. Development and adaptation of such systems cannot be covered by an exhaustive and complete theory if only because of human involvement in the applications, the partially arbitrary nature of procedures in business, manufacturing, government, the service sector and so on, and the potential unboundedness of the domain boundaries (Turski 1981). Inherently, therefore, the software evolution process is, at least to some extent, *ad hoc*.

1.6.8 Summary

In summary, every *E*-type program is a bounded, discrete and static reflection of an unbounded, dynamic application and its operational domain. The boundaries and

[8] In accepted mathematical usage the term *model* is valid when formally describing, for example, a required relationship between a program *specification*, the *application*, operational *domains* to which it relates and the *program* derived from it. The specification is derived from application and domain statements by an abstraction process. The program is, in turn, derived from the specification by reification. The program, application and domains will, however, possess additional properties. These must not be incompatible with the specification but are not necessarily compatible with one another. The program is, therefore, *not* a model of the application and its domains. The term *model-like reflection* is used here to convey the relationships which do exist. Software maintenance may then be viewed as 'maintaining reflective validity between the program and application' as the latter and its operational domains evolve.

other attributes of the latter are first determined in initial planning and adjusted, during development, by technology, time, business and operational considerations and constraints. Some are determined explicitly in processes such as requirements analysis and specification, others as a result of explicit or implicit assumptions adopted and embedded in the system during the evolution process. Fixing the detailed properties of human/system interfaces or interactions between people and the operational system must include trial and error. The fine design detail cannot be based on either one-off observation and requirements elicitation or on intuition, conjecture or statistics alone. It arises from continuing human experience, judgement and decision by development staff, users and so on. Development changes perception and understanding of the application itself, of facilities that may be offered, of how incompatibilities may be resolved, what requirements should be satisfied by the solution, possible solutions, and so on. In combination, such considerations drive the process onwards, by experience and learning-based feedback, to its final goal, a satisfactory operational system.

1.6.9 Principle of Software Uncertainty

The preceding discussions have shown how the processes of abstraction and bounding each generate a bounded number of assumptions that are reflected in the specifications and programs. The latter are a subset of the unbounded number of assumptions made, implicitly or explicitly, during the above processes and that relate, *inter alia*, to the states and behaviours of the various domains addressed by the program and within which it operates. The real world is dynamic, always changing and the rate of change is also likely to be significantly affected by the development, installation and use of the computing system. Inevitably, members of this bounded, embedded, assumption set will become invalid as a result of changes in the real world. This principle follows that every E-type system is likely to reflect a number of invalid assumptions. Since they are, in general, not identified, the consequences in execution are not known. Hence the outcome of every E-type program or system execution is uncertain. This observation has been formalised in a Principle of Software Uncertainty. It has been discussed in several papers (e.g. Lehman 1989, 1990), more recently as an example of potential theorems in the development of a theory of software evolution (Lehman and Ramil 2000a, 2001b).

1.7 Software Systems Evolution

1.7.1 Early Work

The decision whether, when and how to upgrade a system will be taken by the organisation owning a product though often forced on them by others, clientele, for example. Their considerations will involve many factors: business, economic, technical and even social. Each version or release that emerges from the evolution process which implements their decision is an adaptation, improvement (in some sense) and/or extension of the system, and represents one element of the ongoing evolution. The sequence of releases transforms the system away from one satisfying the original concept to one that successively supports the ever-changing and emerging circumstances, needs and opportunities in a dynamic world. If conditions to support evolution do not exist, then the system will gradually lapse into uselessness as a widening gap develops between the real world as mirrored by the program and the real world as it now is (First Law of Software Evolution, Lehman, 1974).

Recognition of software evolution, its identification as a disciplined phenomenon and its subsequent study was triggered by a 1968/1969 report entitled *The Programming Process* (Lehman 1969). *Inter alia*, the study examined empirical data on the growth of the IBM OS/360-370 operating system. As analysed in a number of papers since then, it concluded that system evolution, as measured, for example, by growth in size over successive releases, displayed regularity that was unlikely to have been primarily determined by human management decision. Instead, the regularity appears to be due to feedback via many different routes. The empirical data that first suggested this conclusion was illustrated and was briefly discussed in Section 1.4, Figure 1.1. The figure plots system size measured in numbers of modules – a surrogate for the functional power of the system – against *release sequence number* (RSN) up to and including the period of instability preceding its break-up into the VS/1 and VS/2 systems.

The growth trend of OS/360-370, when plotted over releases, was close to linear[9] up to RSN 20. A superimposed ripple suggested self-stabilisation around that trend, *self* because no indication could be found that management sought linear growth. In fact, there was no evidence that growth considerations played any part in defining individual release content.

This stabilisation phenomenon provided the first empirical evidence that feedback was playing a role in determining the growth rate of functional power or other attributes of evolving systems. The conclusion was strengthened (Belady and Lehman 1972) by the post RSN 21 instability. By the same reasoning, this was attributed to excessive *positive* feedback, as reflected in the excessive incremental growth[10] from RSN 20 to RSN 21.

1.7.2 FEAST

Follow-on studies in the 1970s and 1980s (Lehman and Belady 1985) produced further evidence of similar evolutionary behaviour and led eventually to eight *Laws of Software Evolution* that encapsulated these invariants (Lehman 1974, 1978, 1980, Lehman *et al*. 1997). Following formulation of the FEAST hypothesis (Lehman 1994) successive studies, FEAST/1 and/2, were undertaken to further explore the evolution phenomenon (FEAST 2001). Figure 1.2 provides just one example of the similarity between the observation of the growth of OS/360-370 (Figure 1.1) and observations some 30 years later. Discussion of those results can be found in some of the publications listed on the FEAST web pages (FEAST 2001).

Attention should also be drawn to some of the differences in the evolution patterns of the systems studied. For example, five of the FEAST systems display *declining* growth rate trends appropriately modelled by an inverse square model of the form $S_{i+1} = S_i + E/S_i^2$ where S_i is the predicted size of the release with sequence number 'i', with size measured in appropriate units and E is a model parameter as determined from data on the growth history of the system (Turski 1996). Moreover, for all five of these systems, the precision of the trend model was increased by breaking up the growth data and by estimating the model over two or more sequential segments. The recovery of the growth rate, at break points such as that visible in Figure 1.2 may be assumed to indicate improvements in the evolution process or restructuring of the evolving system. In fact, the figure appears to

[9] As noted later, this early result was subsequently refined but this does not affect the basic reasoning.

[10] Three-and-a-half times as great as the previously largest growth increment.

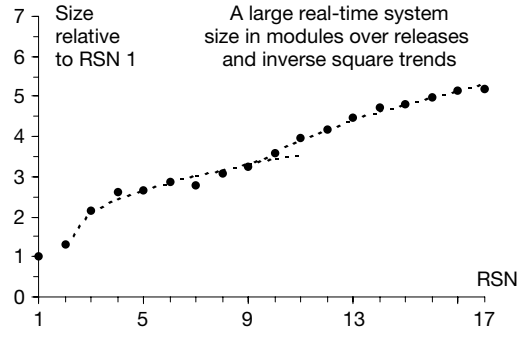

Figure 1.2 Growth trend of one of the systems studied in the FEAST projects (dots) with inverse square models fitted to two individual segments (dashes). The start of segment 1 at RSN 2 provides a slightly better fit than a model fitted starting at RSN 1

provide empirical support for the evolutionary stages concept (Bennett and Rajlich 2000, Rajlich and Bennett 2000).

These five systems exemplify release level evolution. Phenomenological reasoning as summarised above suggests that, in principle, similar behaviour is to be expected from *all* real-world software systems. The data studied in FEAST was, however, all obtained from systems developed and evolved using variations or extensions of the classical waterfall process paradigm. Use of newer approaches, *object oriented, open source, agile* and *extreme programming*, *component-based architecture*, and *implementation* introduces new situations. But the all-pervading influences of factors such as the role of learning, feedback, environmental changes, the impact of computer integration and usage on needs and usage patterns and the consequences of assumptions in a dynamic real world reflected in the software suggest that paradigm changes will, at most, have an impact on the detail of the software evolution phenomenon. The evolution of specific software under the newer approaches is currently a topic of study and some results have been reported [e.g. Godfrey and Tu 2000, Lehman and Ramil 2000b, Succi *et al*. 2001, Bauer and Pizka 2003, Capiluppi *et al*. 2004]. A discussion of current results cannot be included here.

The sixth system studied under FEAST involved *ab initio development* of a defence system. Moreover, in that system, the size of the executable code was externally constrained by the memory capacity of the computer used in the application and alternative metrics were unavailable. Hence it was concluded that the comparison of this system to the other five could not contribute to the present study. It is mentioned here for the sake of completeness.

1.7.3 The Growth Trend

The observed inverse square growth trend is consistent with a hypothesis that declining growth rate may be attributed, at least in part, to growing complexity of the evolving system and application as change is applied upon change. The growth in complexity may, of course, be compensated by growing familiarity with the system, improved training, expertise, documentation and tools, by re-engineering, system restructuring, *refactoring* (Fowler 1999) and, more generally, *anti-regressive* activity (Lehman 1974). System dynamics models (Forrester 1961) reproducing this phenomenon suggest that sufficient

anti-regressive activity can yield close to linear growth (following an initial increasing growth rate as briefly discussed below) (Lehman *et al*. 2002).

The results of these investigations have been widely reported (FEAST 2001). The systems studied were industrially evolved systems stemming from different development organisations, addressing different application areas in differing operational environments and of widely different sizes. The conclusions suggested some relatively minor modification of earlier overall results and strengthened conviction in the universality of the phenomenon of *E*-type software evolution. As stated by the first (continuing change) and sixth (continuing growth) laws, such systems must be continually adapted, changed and extended, that is, evolved, if they are to remain of value to users and profitable to the organisations in charge of their evolution.

More recently it has been realised that the inverse square model, while valid over an extended period of the system life cycle, or over segments, is not the last word. Re-examination of existing data and its interpretation indicates that growth rates at the start of a development or at the initiation of a new growth segment are increasing, even approaching the exponential. If, as appears likely, this conclusion is sustained then it is more appropriate to replace the segmented inverse square growth model with an S-curve, initial increasing rate that gradually approaches linearity and then decreases into, possibly, inverse square growth.

1.7.4 Evolution Drivers

Observations and insights that suggested the laws of software evolution support the observation that feedback plays a major role in driving and controlling release processes (Lehman and Belady 1985, chap. 16; Lehman 1991). Sources of feedback include *defect reports* from the field, *domain changes* due to installation, operation and use of the system, changing user *needs*, new *opportunities*, advances in technology, even the economic climate. At a more abstract level, experience changes user *perception*, *understanding*, underlying application *detail*, system *concepts*, *abstractions* and *assumptions*. A need and demand for change emerges. The always-emerging needs are conveyed back to suppliers and demands action on their part. But the response can rarely be immediate since it requires informed selection and approval that requires technical, business and, economic judgements with moderation of the needs and priorities of many different users. As is to be expected from a feedback system, the resultant delays cause further distortion of the evolution process.

The information required to support this process propagates along paths involving human interpretation, judgement and decision; hence there are significant delays. All involved are liable to have an impact on the information and on feedback characteristics. Many will contribute to the change process and not all are developers or user communities exploiting insight gained from their usage and experience. But in all cases information is the principal driver, with the characteristics of the feedback path influencing its significance: that is, process-internal feedback paths are relatively short and involve people who are experts in the application, the development process and the target system. Their feedback is based on individual interpretation. In control-theoretic terms it can be interpreted as low-level amplification, delay, noise and distortion. But long external *user*- and *business-based* loops are likely to be primary determinants of *release dynamics* characteristics.

1.7.5 Relationship Between the Above Levels of Evolution

Sections 1.6 and 1.7 of this chapter each addresses one area of software evolution. Section 1.6 covers the activity of *development* of an entire system *ab initio* or of a change to an existing system. Section 1.7 addresses the continual adaptation of a developed system to changing circumstances, needs and ambitions. The two areas are related: The second area also requires planning, development, specification, design and implementation of desired changes and additions. Such implementation will then involve evolution activity such as the one considered in the first area. But as briefly stated in Section 1.1.1, relative to the system as a whole, the amount of change in any one release of a software system is generally small even though locally many individual elements or components may be changed or replaced by newly developed or acquired alternatives.

1.7.6 Evolutionary Development

Attention may also be drawn to development approaches that constitute an amalgamation of the two above areas. As an example, consider Gilb's Evolutionary Development approach (Gilb 1981, 1988). In this approach, *ab initio* development and fielding of complex (in some sense) systems in a sequence of releases each involving a new component or chunk of functionality. In this way, the complexity of the task undertaken in any release interval is greatly reduced. Moreover, by fielding the '*in development*' system to users, the latter becomes progressively exposed to a system of increasing functionality and power. Learning and reaction, that is, user feedback, can be taken into account well before development is completed. Hence the degree and complexity of validation and of rework may be reduced. Regression testing and revalidation, on the other hand, is likely to have to be increased.

Application of the approach depends on being able to architect the system so that constituent parts may be interconnected, part by part, to yield a sequence of viable systems of increasing functionality and power. The parts are developed, installed and, ideally, introduced into use in a predetermined order. The latter is, however, very likely to require modification as a result of, for example, unanticipated difficulties in completing some elements, a need for redesign, introduction of new requirements, domain changes and so on.

The system is not evolved continuously but by leaps and bounds. Constituent parts are progressively exposed to system internal interactions and to usage. Hence, some interface errors and undesirable or incorrect internal interactions will be detected sooner than would be the case if real-world operation were to await completion of the entire system. On the other hand, any benefit from this may be reduced or even reversed as development later in the evolution process takes note of changes in the operational domains, reflects these in the current design and implementation activity but fails to adjust older part of the system. That is wrong of course, but very likely to occur.

It is likely that where a system structure can be decomposed to yield a viable process and a usable system at each stage of the development, the approach can provide clear net benefit. It has been industrially applied in practice but we are not aware of any empirical assessment of its effectiveness in relation to more conventional development approaches. It must, however, be accepted that a major problem in real-world system development is that of uncertainty and risk associated with fixing the properties of the system. Related to

this is the lack of a theoretical framework to guide selection of system properties during requirements analysis, specification and design. Many decisions are, therefore, arbitrary and not fully validated or rejected until the system has been fielded and is in regular use. It is not now clear how effectively evolutionary development addresses these issues although potentially it might well be more effective in this respect than the more classical approaches. Detailed assessment of the approach is required to determine its dependency on the nature of the application, development and other environments and what, if any, changes are required to ensure maximum benefit from the approach.

1.8 Evolution of the Application and Its Domain

Continuing evolution is not confined to the software or even to a wider *system* within which the software may be embedded. It is inherent in the very nature of computer *application*. This is illustrated by a study of long-term *feature* evolution in the telephone industry (Antón and Potts 2001). The activity that software supports and the problems solved also evolve. Such evolution is, in part, driven by human aspiration for improvement and growth. But more subtle forces are also at play. The very installation and use of the system changes both the activity being supported and the domains within which it is pursued. When installed and operational, the output of the process that evolved the software changes the attributes of the application and the domains that defined the process in the first place. As illustrated by Figure 1.3, the development process in association with the application and the operational domains as defined and bounded, clearly constitute a feedback loop. Depending on the manner in which and the degree to which changes impact use of the system and loop characteristics such as amplification, attenuation and delays, the overall feedback at this level can be negative or positive, leading to stabilisation, continuous controlled growth and/or instability.

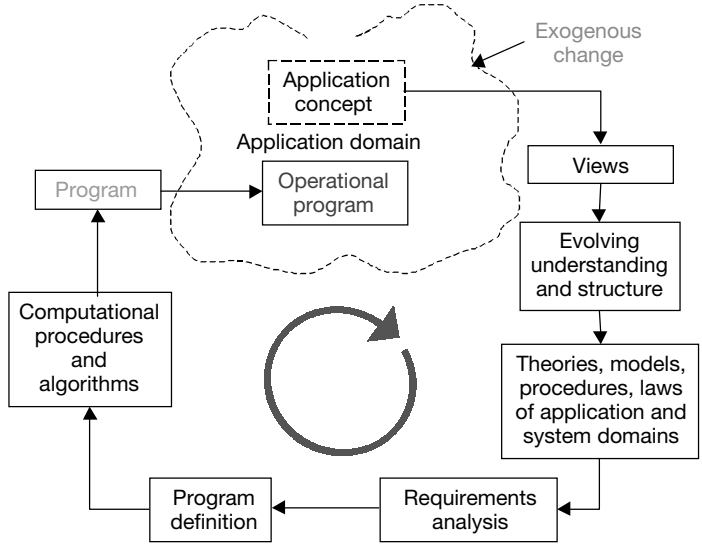

Figure 1.3 Evolution of the application in its domains as an iterative feedback system. Process steps are illustrative. Internal process loops are not shown

In many instances, however, the phenomenon of application evolution is more complex than indicated in the preceding paragraph. In particular, it may not be self-contained but a phenomenon of *co-evolution*. As government, business and other organisations make ever greater use of computers for administration, internal and external communication, marketing, security, technical activity and so on, the various applications become inextricably interdependent, sharing and exchanging data, invoking services from one another, making demands on common manpower resources and budgets. The inescapable trend is towards the integration of services, internal and external, with the goal, for example, of minimising the need for human involvement in information handling and communication, avoidance of delays and errors and increases in safety and security. And such integration is seen as needing to gradually extend to clients' systems, their customers and suppliers and service organisations, banks for example.

With this scenario, the rate at which an organisation can grow and be adapted to changing conditions and advancing technology depends on the rate at which it can evolve the software systems that support its activities. More generally, in the world of today, and even more of tomorrow, organisations will become interdependent. This will happen, whatever their activity or sphere of operation, however disparate the domains within which they operate, the activities they pursue, the technologies they employ and the computer software which links, coordinates and ties all together. All co-evolve, each one advancing only at a rate that can be accommodated by the others. And those rates depend not only on the various entities involved but also on the processes pursued and the extent to which these can be improved. Software is at the very heart of this co-evolution. Change to any element almost inevitably implies software changes elsewhere.

1.9 Process Evolution

1.9.1 Software Processes as Systems

Software processes are the aggregate of all activities in developing or evolving software and of the relationships between them. If correctly executed, they transform an application concept into a satisfactory operational system. Improvement of the process is achieved by improvement of its inputs, its parts and of their interactions. The parts themselves implement and support technical, operational and managerial activity. At some level, process steps can be seen as elements in a successive transformation paradigm (e.g. LST as in Section 1.6.2).

But enactment of a software process requires a wide variety of interacting activities and entities. Many of these are outside the core transformational steps but are nevertheless, needed to address fuzziness in the application concept, to enable the orderly interaction of many stakeholders and to ensure that the required outcome is achieved within relevant quality, schedule and economic constraints.

1.9.2 Process Improvement

Over the past decade, computers and the software that gives them their functional capability, have penetrated ever more deeply into the very fabric of society, individually and collectively. The world at large has become more and more dependent on the *timely* availability of *satisfactorily operating* software with the reliability and at a *cost* that is commensurate with the *value* that the software is to yield on *execution*. But, as repeatedly

observed, *E*-type software must be adapted and extended as the world changes to yield satisfactory results whenever or wherever, within the accepted and supported bounds, the system is executed. Errors or delays in this continuing process can yield significant cost and/or performance penalties due to incorrect or unacceptable behaviour. They can even constrain or throttle organisations limited by out-of-date capabilities and *legacy* software. The extent, number and severity of problems experienced is certainly, at least in part, related to the nature and quality of the process by which the software is developed, maintained and evolved.

As variously practised today, that process is far from perfect, expensive, the source of many delays and with its products displaying major defects and deficiencies. The need for improvement is widely accepted. Major investment is being made in developing and applying software *process improvement* techniques (Zahran 1997). The methods used have been formalised, developed and applied using paradigms such as SPICE (El Eman *et al*. 1997), Bootstrap (Kuvaja *et al*. 1994) ISO 9000 and its derivatives, CMM (Paulk *et al*. 1993) and more recently CMMi (Ahern *et al*. 2001). All are being explored and applied the world over.

A major element of this search for improvement involves the development of new programming paradigms and languages. These include Object Orientation, Component-Based Architecture, Java, UML, Agile and Extreme. These new technologies involve significant changes in approach and/or development practices to earlier practice. They also cross-fertilise one another and, in turn, suggest or demand changes to or extensions of the processes in which they feature. Hence, software evolution processes also evolve. In the absence of a comprehensive scientific framework for software technology, such evolution is primarily driven and directed by experience, emerging insight, inventiveness and feedback.

1.9.3 The Theoretical Approach

Process improvement may be based on theory or be empirical. The first approach is exemplified by the work of WG 2.3 (Gries 1978). That group has been meeting formally since 1971 as an IFIP working group, to discuss its members' views and work on various aspects of programming methodology. The approach is bottom–up, based on both fundamental thinking about the nature and goals of the basic program development task and how it is or could be approached by individuals seeking solutions of a problem. The group's many positive results extended earlier work by their members. These included Dijkstra's much quoted observation that 'GOTOs are considered harmful' (Dijkstra 1968b), the concepts and procedures of structured programming (Dijkstra 1972a), the concepts of program correctness proving (Dijkstra 1968a, Hoare 1969, 1971) and successive refinement (Wirth 1971). The approach has provided basic concepts of modern programming methods, but relate in the first instance to *S*-type programs. As a result they are most significant at the heart of programming process improvement. They provide, for example, a basis for individual programmer practice (Humphrey 1997) that seeks to develop defect-free code.

In summary, the wider importance of the theoretical approach relates in the main to the use of *S*-type elements to implement and evolve *E*-type systems. The resultant methods and techniques are primarily relevant to the development of individual elements within such systems. Any demonstration of correctness is limited by the fact that, in

the total system context, the individual element must be, and be maintained, correct in the context of an intrinsically incomplete specification. The value of this, if achieved, is unquestionable. The use of precise specifications with correct implementations at any level provides value. But as the application, its domains and the system in which elements are embedded and integrated evolve some, at least, will have to be adapted to the changing environment in which they operate. It will become increasingly difficult to maintain them correctly. The use of formal methods wherever possible is, at most, a partial answer to maintaining an *E*-type system satisfactorily.

1.9.4 Evolving Specifications

The theory-based approach has also identified another fundamental software evolution problem, the consequences of evolution at the specification level. As already stated in Section 1.6.2, '... the problem of adopting existing software to evolving specifications remains largely unsolved, perhaps is algorithmically not solvable in full generality ...' (Turski 2000). More generally, an open problem in program implementation relates to the achievement of evolutionary approaches, in which, for example, unforeseen changes and updates to an *existing* specification can be cheaply and safely reflected in an *existing* model of that specification, including the operational program. That problem too may not be solvable in full generality.

1.9.5 The Empirical Approach

The empirical approach must be seen as being parallel to and in support of the theoretical approach. It is essential if the methods and techniques developed in relation to the latter are to make a significant contribution to the evolution of large program systems. Empiricism in the software evolution area is exemplified by Lehman's early work (Lehman and Belady 1985), the work of the FEAST group (Lehman *et al*. 1994–2002, FEAST 2001) and that of Kemerer (Kemerer and Slaughter 1999). All these exploit observation, measurement, modelling and interpretation of actual industrially developed and evolved software systems. This permits the development of black and white box (e.g. system dynamics) models. Reasoning about the findings leads to the gradual development of an empirical theory, and this in turn must be tied in with the low-level approach.

It is not possible to discuss the findings of these empirical studies further here and the interested are referred to the referenced literature. It is, however, worthy of note that the eight laws of Software Evolution, as outlined briefly below, are a direct outcome of such empirical observation and interpretation over a period of some 30 years. The observations brought together here provide a basis for development of a formal Theory of Software Evolution (Lehman 2000, Lehman and Ramil 2000a, 2001b), indeed they constitute an informal, for the moment partial, theory. They also lead to practical rules for software release planning, management and control (Lehman and Ramil 2001a).

1.9.6 Laws of Software Evolution

The, currently eight, Laws, as listed below (Figure 1.4), were formulated in the decade following the mid-seventies. They were derived from direct observation and measurement of the evolution of a number and variety of systems. As such, they were viewed as reflecting specific, largely individual, behaviour and regarded as independent of one

No.	Name	Statement
1	Continuing Change	An *E*-type system must be continually adapted, else it becomes progressively less satisfactory in use
2	Increasing Complexity	As an *E*-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity
3	Self Regulation	Global *E*-type system evolution is feedback regulated
4	Conservation of Organisational Stability	The work rate of an organisation evolving an *E*-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime
5	Conservation of Familiarity	In general, the incremental growth (growth rate trend) of *E*-type systems is constrained by the need to maintain familiarity
6	Continuing Growth	The functional capability of *E*-type systems must be continually enhanced to maintain user satisfaction over system lifetime
7	Declining Quality	Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an *E*-type system will appear to be declining
8	Feedback System	*E*-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

Figure 1.4 The Laws of Software Evolution

another. Relationships between them, though not ruled out, were not investigated. However, following formulation of the observed feedback phenomenon as formalised in the eighth law[11], the likelihood of a structured relationship rooted in that law was accepted and awaits development as an integral part of the formal development of a Theory of Software Evolution. More complete discussion of the laws may be found in a number of publications (Lehman 1974, 1978, 1980, Lehman and Belady 1985, Lehman *et al.* 1997, Lehman and Ramil 2001a).

1.9.7 The Ephemeral Process

Any instance of the process is *transient*, *ephemeral*. Once executed, it is gone forever. It will normally have been pre-planned in outline, detail being filled in as progress is made. But unanticipated circumstances and conditions are the norm; budgets and schedules change, new requirements, functional or performance problems arise. All these, and many more, lead to process adjustments, adaptations and, though to be avoided, changes *on the*

[11] It is of interest to note that this phenomenon was second only to recognition of continual evolution as a phenomenon and was already referred to in 1972 (Belady and Lehman 1972).

fly. Triggered by observation of results or consequences of past activity or by perception of what lies ahead, such unplanned changes are often the result of crisis action and local reaction. They may result in a change to planned process activity or a need to backtrack or iterate. Thus they tend to be error prone, hence undesirable. In any event there is a complex mixture of feedback and *feed forward* of information based on individual and collective interpretation, intellectual judgement and decision by humans that determines how to proceed. The greater part of this is based on what is perceived, what is experienced, what is anticipated and challenges that arise.

Absolute predictability is not possible when people are involved in a process. Some degree of freedom must exist, otherwise their activity could and would be mechanised. The freedom relates to what is done, how it is done, what is not done and by what it is replaced. The potential for process definition and pre-planning is limited in extent, level of detail, precision and repeatability. It can only be enforced at a comparatively coarse level of granularity. Enforcement of a process specified at a high level of detail may appear desirable in specific circumstances as in life-critical medical or aerospace applications. But it must be accepted that rigid enforcement and application may itself result in problems, defect injection, inadequate treatment of unforeseen circumstances, high cost or serious time delays. Any of these can result from, for example, a misunderstanding of aspects of the situation, incorrect anticipation of the future or delays while authorisation to deviate is obtained.

Most development environments are subject to strong resource, schedule, budget and other constraints. Reliance on a process that can and will be carried out as planned is likely to prove naive in the extreme. Even in a single project, the process will evolve dynamically *in vitro*, as well as *in vivo* through pre-planning.

1.10 Process Model Evolution

1.10.1 The Nature of the Software Process

Real-world processes are very complex. As a multi-level, multi-loop, multi-agent feedback system with many of its mechanisms involving unpredictable decision taking by humans, the process is likely to display the nonintuitive or even anti-intuitive behaviour observed in feedback systems (Forrester 1961). Understanding how they act and interact requires models that reflect feedback mechanisms that can be used to validate them by observation and measurement of real-world properties and events.

1.10.2 Process Models

Models are used in many different areas and many different ways to facilitate and advance understanding of a phenomenon, activity or process. They are indeed essential as vehicles for communication and reasoning, providing, for example, means for systematic and disciplined examination, evaluation, comparison and improvement. As simulators or enactment tools they permit preliminary measurement, exploration and evaluation of proposed changes (Tully 1989). In all these areas, their role can be greatly enhanced if they are formal. In the absence of a formal representation, results are difficult to obtain or validate using theoretical reasoning.

As the applications they reflect become larger, more complex and more integrated, models must be evolved to remain of continuing value. This is particularly the case

for models of the software process. With feedback producing pressure for continuing change and direct human involvement, the full *consequences* of introduction and use of computers and software is essentially unpredictable. It has already been stressed that software must remain compatible with the human reactions and volatile applications and domains that it addresses and in which it operates. The result is continual pressure for change to applications, interfaces, domains and processes involved and to the application of changes to any of these. As a consequence, software processes and models of them must also be changed and evolved to cope with the demands made of them, for the role they can play in maintaining and improving the reliability, timeliness, cost effectiveness and, above all, reliability of the process product. One must also consider that there are inherent limitations in modelling processes involving human action and decisions in a dynamic world (Lehman 1977).

1.10.3 Software Process Models

In considering the role and potential of models in the software process, one must take the dynamic, feedback-dependant nature of the latter into account. Models must reflect its structure. This fact was briefly described and explored, in a number of papers in the 1970s and 1980s (Lehman and Belady 1985) using very simple models. This early work came to the forefront with the first International Process Workshop (Potts 1984) and its successors. Later instances of this series were dominated by discussions of *process programming* and process program models. Interest in this approach was triggered by Osterweil's keynote address at ICSE 9 in 1987. Serious questions about it were, however, raised by Lehman in his response to that talk. (Lehman 1987). Sometime later *behavioural, system dynamics* (Forrester 1961) and other types of process models, were proposed by Abdel-Hamid and Madnick for the management of software development (Abdel-Hamid and Madnick 1991).

1.10.4 Process Improvement

At the highest level of abstraction, process improvement relates to a variety of performance and product quality, more generally *reliability*, factors as well as those relating to cost and elapsed time, more generally *productivity* factors. These include the reduction of time to detect, analyse and correct defects of any sort and the reliable release of a correction to users. An overall goal must be the reduction in the total number and frequency of justified defect reports, and their rate of submission once the system is in use. In this context, behavioural process models that address these concerns can be useful (e.g. Lehman *et al.* 2002).

Process evolution to achieve such improvement proceeds slowly and is implemented in incremental steps. Such steps will tend to be implemented locally as, for example, in the insertion of a new activity between two existing steps, or code inspection between coding and testing. In other instances the new activity will be out of the main line of development seeking out and verifying, for example, program elements that have been formally specified. As in the two instances cited, such an improvement may be of great local significance with both inspection and verification adding to the underlying quality of the system being developed. As such it makes a definite contribution to the overall value of the system, one that must be welcomed. But in terms of contribution to the overall

quality or other attributes of the system as perceived by system beneficiaries, users and other stakeholders, this low-level activity will be taken for granted as it would in any other engineering discipline. It does not directly provide the visible benefit that stakeholders expect in a system. Programming standards and software validation techniques have, in general, advanced to the point where code stability and quality is largely taken for granted.

Incremental improvement at the process step level, for example, has, in general, little global impact on stakeholders in general, and individual and organisational users in particular.

Unless process changes take the multi-level feedback structure of the process into account, any benefit is likely to be overlooked, an illustration of the anti-intuitive behaviour of such systems. Multi-level loop structures tend to be largely hierarchical but may also involve loops, cutting across level boundaries. Whatever the case, one will have loops within loops; more generally, feedback-generated responses that control and moderate the behaviour and output of other loop mechanisms. Every process locality with its feedback loops will lie within others that drive and control the more extensive process and its interaction with the operational domains. If feedback is negative, encompassing loops will attenuate, even suppress, the effects of inner, more local changes. The potential impact of improvements, whatever their significance in the isolation of their process neighbourhood, will be small outside. Positive feedback, while having the potential to amplify some process property, may ultimately cause instability in the behaviour or a property of the domain it controls. The instability of OS/360-370 growth illustrated occurring after RSN 20 as illustrated in Section 1.4, Figure 1.1 represents an example of this effect. To constitute a visible and measurable global improvement and to provide benefit to the stakeholders and user communities, a process improvement must have visible, preferably measurable, impact outside the programming process. Only such impact, has meaning and value in the real world (Lehman 1994).

Models can be used to develop and evaluate proposed software process improvements. However, if they are to be of value in their reflection of the likely properties of the process after change and in the consequences of its implementation, they may well be more difficult, costly and error prone to implement than is the process change itself. It is not sufficient that the model reflects the change with sufficient precision. A framework must be provided to provide a realistic environment for its validation, assessment of the proposed change and evaluation. In addition, one must provide mechanisms to adjust, and in some sense optimise, the change. And in any event, the full global consequences of a process change are not straightforward to predict or to evaluate in the presence of feedback, whether before implementation or after. Process models have a role in and a contribution to make to software engineering but these are likely to be rather limited, barring some major advances in process modelling and the use of models.

One final note on process model evolution must be made. However exploited, the information that drives improvement is garnered from observation and previous experience. Model evolution is also feedback driven. The flow will be from within the organisation, from other software developers and from process experts and practitioners (Lehman 1991). Disciplined and directed effort in process improvement is typified by the work

of the Software Engineering Institute at Carnegie Mellon University (Humphrey 1989). Their work does not explicitly focus on process models or feedback direction and control. But those, in essence, are among the issues addressed and exploited.

1.10.5 Links Between Process and Process Model Evolution

What is the nature of the linkage between evolution of a process and that of its model? Where impetus for change comes from a need to adapt a process to specific conditions or circumstances, model evolution is a consequence of process evolution. The likely consequence or benefit of a process change may possibly be assessed by implementing, exploring and comparing alternative changes through model enactment before incorporating the selected change in the process. Evaluation of changes to the model can drive process change. Where this is not done, changes made to the process, whether premeditated or on the fly must be reflected in a change to the model if the latter is to retain its validity and value. If, on the other hand, the pressure for evolution comes from recognition of a need for improvement, the process model can play a seminal role. It may then influence the design and evaluation of the change before implementation. Such evaluation must, however, include the benefits that would result and the investment required for their implementation.

1.11 Relationships Between Levels

1.11.1 The Software/Software Process Contrast

There are clearly interactions between the various aspects and levels of evolution of individual roles and evolution patterns as discussed. But this must not be interpreted as indicating that there are similarities between their evolutionary behaviour. The reverse is, in fact, the case. Software process evolution, for example, clearly differs significantly from that of the software itself. Similarly, there is a fundamental difference in the relationships between a software process and its models on the one hand and between *E*-type software and the problem or application processes of which the software is a model-like reflection on the other.

Wherein lie the differences? *E*-type software is concerned with some application process and the application of program execution to the real world (Lehman 1991). Given that operational domain, one develops and evolves systems to be used by a changing population of (largely anonymous) people and organisations with differing degrees of understanding, skill and experience. The concern will, in general, be with user community behaviour. Only in exceptional instances can code make provision for individual misuse, and that only if such misuse can and has been anticipated.

An essential ingredient of successful software design is, therefore, insulation of the system from user behaviour. Computer applications evolve, *inter alia*, in response to the changes in their software and in the domains. This is so, even though the former may have been inspired by observation of real-world processes influenced or controlled by its execution. There is directed interaction from the software to the application. Though a model-like reflection of the application, it is often the software that forces evolution. Software changes drive application changes while co-evolving with it.

1.11.2 The Software Process/Process Model Contrast

In direct contrast, when software development processes and the models that describe them are considered, the focus of concern is the process even though a model was the source of evolutionary change. A proposed change and its consequences may indeed be explored by use of a model and be evaluated by its enactment. The process may even be guided to some extend by a model-based support environment (Taylor *et al*. 1988). Nevertheless, the real concern remains with the process *in execution*. Humans interpret specifications, process directives, choose directions, take decisions, follow and apply methods. The proof of the pudding lies in the eating. The process model is a broad-brush tool to permit reasoning about the process but the consequences of process execution depend on the processes as executed through the specific actions of individuals. It is the dependability, quality, ease of use, timeliness and robustness of the process, which is of direct concern.

Process models are incomplete; at best a high-level guide to the process. They do not and cannot provide a precise and complete representation of the process actually followed. If, mistakenly, they are accepted as precise and complete they become straightjackets. They constitute a constraint in domains where the unexpected and unanticipated is a daily occurrence. This must be contrasted with executable software. Once acknowledged, the software is relied upon to provide a precise, detailed, complete representation of the actuality required or desired. Software *defines the process of computation* completely. The language that determines it possesses a formal semantics with no ambiguity. Where the process cannot be predetermined, alternatives must be identified and automated, tests devised to select that which is to be followed. The process definition is absolute in the context of that language[12].

Process models, on the other hand are, as already observed, a *partial* reflection of the desired process. It is the *product* of that process that is of concern. Changes to the model are incidental to the ultimate purpose and interest in pursuing the process. They describe changes, proposed or implemented; concepts to be translated into reality by people. They are evaluated in terms of their *impact* on the process *in execution*. A process change may be conceived and incorporated in a model. The acid test comes with the *execution* of an instance of the *process*. Determination of its improvement or deterioration, success or failure, is judged on the basis of *product* attributes.

There are also other significant differences. For example, process quality, productivity and cost concerns relate to the process, not its model. For software, the reverse is the case. Quality, productivity and cost concerns as visualised by the software engineer relate to the software as a reflection of the application in its domain, not to the application itself. Concern about such factors does arise but these must, in the first place, be addressed by application experts. Deficiencies will, in general, be overcome, in the first instance, by changes to system requirements and specification, to be reflected in changes to future versions of the software.

Consider, finally, the time relationship between model and process changes and the nature of the feedback loops that convey the interactions. For the process the keyword is *immediacy,* whereas for software there is, in general, significant relative delay in feedback. One could go on listing the differences. The analysis as given suffices to indicate

[12] The use of, for example, statistical tests, random number generators or other random choices is no exception to this general rule. It is also predetermined, though the path taken cannot be predicted except in a probabilistic sense.

that the thesis (Osterweil 1987) that 'software processes are software too' must not be taken literally.

1.12 Conclusions

The brief discussion of evolutionary development at the end of Section 1.7 indicates that the classification of areas of evolution proposed in this chapter is not as precise as one might have hoped for. There are other examples and in introducing one more in these concluding remarks, a more general point can be made. Luqi's Evolution by Rapid Prototyping (Luqi 1989) also combines views from areas of *ab initio* development and release-based evolution. This suggests that there might be advantages in simultaneously addressing these, and indeed, other areas described in this chapter. In particular, it must be recognised that the lowest level of evolution as outlined is used to implement the evolution of the individual entities in the other areas. Thus, while compartmentalisation has very clear benefits as an aid to understanding, it remains arbitrary to some extent. It is certain that in industrial situations, for example, evolution over several levels will occur concurrently. Consideration and management of each and of the interactions between them must be coordinated to ensure maximum benefit.

The objective of this chapter has been to expose the wider and crucial role of evolution and feedback in a number of domains related to software. Only recently has serious thought been given to this topic and firm conclusions must await further directed and intensive study. Though not sufficiently structured, the analysis presented here constitutes an outline Theory of Software Evolution (Lehman and Ramil 2000a, 2001b). Formal development and presentation of such a theory should not be long delayed.

In summary, feedback drive and control plays a major, critical and unavoidable role in software technology. The characteristics of individual phenomena are functions of the properties of the feedback loops. As a phenomenon, evolution occurs at different levels in the computing and software domains. There is still much to be learned in this area and the nature, impact and control of evolution at all levels must become a major focus of future research and development.

1.13 Acknowledgments

Many thanks are due to industrial collaborators and academic colleagues for many discussions, particularly during two EPSRC supported FEAST projects (1996–2001). Over the years these have helped to prune, sharpen and extend the concepts and ideas presented.

References

References indicated with an "*" were reprinted in Lehman and Belady, 1985.

T.K. Abdel-Hamid and S.E. Madnick (1991), *Software Project Dynamics – An Integrated Approach*, Prentice Hall, Englewood Cliffs, NJ, 264.

D. Ahern, A. Clouse and R. Turner (2001), *CMMi Distilled – An Introduction to Multi-discipline Process Improvement, SEI Series in Software Engineering*, Addison-Wesley, Reading, MA.

A. Antón and C. Potts (2001), Functional Paleontology: System Evolution as the User Sees It, *23rd International Conference on Software Engineering*, Toronto, Canada, 12–19 May, pp. 421–430.

K.R. Apt and D. Kozen (1986), Limits for Automatic Program Verification of Finite-State Concurrent Systems, *Inf. Process. Lett.*, vol. 22, no. 6, pp. 307–309.

V.R. Basili, L. Briand, S. Condon, W. Melo and J. Valett (1996), Understanding and Predicting the Process of Software Maintenance Releases, *18th International Conference on Software Engineering*, Berlin, Germany, March 25–29.

A. Bauer and M. Pizka (2003), The Contribution of Free Software to Software Evolution, *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, Helsinki, Finland, Sept. 2003.

*L.A. Belady and M.M. Lehman (1972), An Introduction to Growth Dynamics, in W. Freiburger (ed.), *Statistical Computer Performance Evaluation*, Academic Press, New York, pp. 503–511.

K.H. Bennett and V.T. Rajlich (2000), Software Maintenance and Evolution: A Roadmap, in A. Finkelstein (ed.), *The Future of Software Engineering*, ACM Order Nr. 592000-1, June 4–11, ICSE, Limerick, Ireland, pp. 75–87.

B.W. Boehm (1976), Software Engineering, *IEEE Trans. Comput.*, vol. C-25, no. 12, pp. 1226–1241.

B.W. Boehm (1988), A Spiral Model of Software Development and Enhancement, *Computer*, vol. 21, May 1988, pp. 61–72.

F. Brooks (1975), *The Mythical Man-Month*, Addison-Wesley, Reading, MA.

A. Capiluppi, M. Morisio and J.F. Ramil (2004), The Evolution of Source Folder Structure in actively evolved Open Source Systems, *Metrics 2004 Symposium*, Chicago, Ill.

C.K.S. Chong Hok Yuen (1981), *Phenomenology of Program Maintenance and Evolution*, PhD thesis, Department of Computing, Imperial College.

S. Cook, R. Harrison, M.M. Lehman and P. Wernick (2006), Evolution in Software Systems: Foundations of the SPE Classification Scheme, *J. Softw. Maint. Evol.*, vol. 18, no. 1, pp. 1–35.

E.W. Dijkstra (1968a), A Constructive Approach to the Problem of Program Correctness, *BIT*, vol. 8, no. 3, pp. 174–186.

E.W. Dijkstra (1968b), GOTO Statement Considered Harmful, Letter to the Editor, *Commun. ACM*, vol. 11, no. 11, Nov. 1968, pp. 147–148.

E.W. Dijkstra (1972a), Notes on Structured Programming, in O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare (eds.), *Structured Programming*, Academic Press, pp. 1–82.

E.W. Dijkstra (1972b), The Humble Programmer, ACM Turing Award Lecture, *Commun. ACM*, vol. 15, no. 10, Oct. 1972, pp. 859–866.

K. El Eman, J.N. Drouin and W. Melo (1997), *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE Computer Society Press, Los Alamitos, CA, p. 450.

FEAST (2001), *Feedback, Evolution And Software Technology*, http://www.doc.ic.ac.uk/~mml/feast/ <as of Oct. 2001> See also http://www.cs.mdx.ac.uk/staffpages/mml <as of Feb 2004>.

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

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

T. Gilb (1981), Evolutionary Development, *ACM Softw. Eng. Notes*, vol. 6, no. 2, April, 1981, p. 17.

T. Gilb (1988), *Principles of Software Engineering Management*, Addison-Wesley, Wokingham, United Kingdom.

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

D. Gries (1978), *Programming Methodology – A Collection of Articles by Members of IFIP WG2.3*, Springer-Verlag, New York, p. 437.

C.A.R. Hoare (1969), An Axiomatic Basis for Computer Programming, *Commun. ACM*, vol. 12, no. 10, pp. 576–583.

C.A.R. Hoare (1971), Proof of a Program FIND, *Commun. ACM*, vol. 14, no. 1, pp. 39–45.

W.S. Humphrey (1989), *Managing the Software Process*, Addison-Wesley, Reading, MA.

W.S. Humphrey (1997), *Introduction to the Personal Software Process(SM)*, Addison-Wesley, Reading, MA.

IWPSE. (2004), *Proceedings International Workshop on Principles of Software Evolution*, Kyoto, Japan, 6–7 Sept. http://iwpse04.wakayama-u.ac.jp/ <as of July 2004>.

C.F. Kemerer and S. Slaughter (1999), An Empirical Approach to Studying Software Evolution, *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, July/August 1999, pp. 493–509.

S. Kuvaja, P. Koch, L. Mila, A. Krzanik, S. Bicego and G. Saukkonen (1994), *Software Process Assessment and Improvement – The Bootstrap Approach*, Blackwell.

*M.M. Lehman (1969), *The Programming Process*, IBM Research Report RC2722M, IBM Research Center, Yorktown Heights, New York.

*M.M. Lehman (1974), Programs, Cities, Students – Limits to Growth, *Imp. Coll. Inaug. Lect. Ser.*, vol. 9, 1970–1974, pp. 211–229; also in Gries, 1978.

*M.M. Lehman (1977), Human Thought and Action as an Ingredient of System Behaviour, in R. Duncan and M. Weston Smith (eds.), *Encyclopedia of Ignorance*, Pergamon Press, Oxford, England.

*M.M. Lehman (1978), Laws of Program Evolution–Rules and Tools for Programming Management, *Proceedings of the Infotech State of the Art Conference, Why Software Projects Fail*, London, England, April 9–11, 1978, pp. 1V1–lV25.

M.M. Lehman (1979), The Environment of Design Methodology, in T.A Cox (ed.), *Proceedings of Symposium on Formal Design Methodology*, Cambridge, UK, Apr. 9–12, 1979, pp. 17–38; STL Ltd, Harlow, Essex, 1980.

*M.M. Lehman (1980), Program Life Cycles and Laws of Software Evolution, *Proc. lEEE Spec. Iss. on Softw. Eng.*, vol. 68, no. 9, Sept. 1980, pp. 1060–1076.

*M.M. Lehman (1982), *Program Evolution, Symposium on Empirical Foundations of Computer and Information Sciences, 1982, Japan Information Center of Science and Technology*, published in J. Info. Proc. and Management, 1984, Pergamon Press, reprinted as chapter 2 in (Lehman and Belady 1985).

M.M. Lehman (1987), Process Models, Process Programs, Programming Support, *Invited Response to a Keynote Address by Lee Osterweil, Proceedings of the Ninth International Conference on Software Engineering*, Monterey, CA, March 30–April 2, pp. 14–16.

M.M. Lehman (1989), Uncertainty in Computer Application and its Control Through the Engineering of Software, *J. Softw. Maint. Res. Pract.*, vol. 1, no. 1, pp. 3–27.

M.M. Lehman (1990), Uncertainty in Computer Application, *Commun. ACM*, vol. 33, no. 5, pp. 584–586.

M.M. Lehman (1991), Software Engineering, the Software Process and Their Support, *IEE Softw. Eng. J.* Special Issue on Software Environ Factories, vol. 6, no. 5, pp. 243–258.

M.M. Lehman (1994), Feedback in the Software Evolution Process, *CSR Eleventh Annual Workshop on Software Evolution: Models and Metrics*, 7–9 Sept. 1994, Workshop Proceedings, Information and Software Technology, Special Issue on Software Maintenance, Elsevier, Dublin, NC, 1996, pp. 681–686.

M.M. Lehman (2000), *These – Towards a Theory of Software Evolution*, EPSRC Proposal, Case for Support Part 2, Department of Computing, ICSTM, 11 Dec.

M.M. Lehman and F.N. Parr (1976), Program Evolution and its Impact on Software Engineering, *Proceedings of the 2nd ICSE*, San Francisco, pp. 350–357.

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

M.M. Lehman and J.F. Ramil (2000a), Towards a Theory of Software Evolution – And its Practical Impact, in Katayama T, Tamai T and Yonezaki N, (eds.), invited talk, *Proceedings ISPSE 2000, Kanazawa, Japan*, IEEE Computer Society Press, Los Alamitos, CA, pp. 2–11.

M.M. Lehman and J.F. Ramil (2000b), Software Evolution in the Age of Component Based Software Engineering, *IEE Softw.*, special issue on Component Based Software Engineering, vol. 147, no. 6, pp. 249–255; earlier version as Tech. Rep. 98/8, Imperial College, London, June 1998.

M.M. Lehman and J.F. Ramil (2001a), Rules and Tools for Software Evolution Planning and Management, *Ann. Softw. Eng. Spec. Issue Softw. Manage.*, vol. 11, pp. 15–44.

M.M. Lehman and J.F. Ramil (2001b), *An Approach to a Theory of Software Evolution*, IWPSE 2001. A revised version as *Background and Approach to Development of a Theory of Software Evolution*.

M.M. Lehman, G. Kahen and J.F. Ramil (2002), Behavioural Modelling of Long-lived Evolution Processes: Some Issues and an Example, *J. Softw. Maint. Res. Pract.*, vol. 14, no. 5, pp. 335–351.

M.M. Lehman, Stenning V. and Turski W.M. (1984), Another Look at Software Design Methodology, *ACM SigSoft. Softw. Eng. Notes*, vol. 9, no. 2, pp. 38–53.

M.M. Lehman, D.E. Perry, J.F. Ramil, W.M. Turski and P. Wernick (1997), Metrics and Laws of Software Evolution – The Nineties View, *Proceedings of the 4th International Symposium on Software Metrics, Metrics 97*, Albuquerque, New Mexico, pp. 20–32; Also in K. El Eman and N.H. Madhavji (eds.) (1999), *Elements of Software Process Assessment and Improvement*, IEEE Computer Society Press, pp. 343–368.

Luqi (1989), Software Evolution through Rapid Prototyping, *IEEE Comput.*, vol. 22, no. 5, pp. 13–25.

R.T. Mittermeir (2006), *Facets of Software Evolution*.

V. Nanda and N.H. Madhavji (2002), The Impact of Environmental Evolution on Requirements Changes, *Proceedings International Conference on Software Maintenance*, Montreal, Canada, pp. 452–461.

P. Naur and B. Randell (1968), *Software Engineering – Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany, Scientific Affairs Division; NATO, Brussels, Belgium, 1969, http://homepages.cs.ncl.ac.uk/brian.randell/NATO/ <as of July 2004>.

L. Osterweil (1987), Software Processes are Software Too, *Proceedings of the 9th International Conference on **Software** Engineering*, IEEE Computer Society Press, Monterey, CA, Pub. 767, pp. 2–13.

M.C. Paulk, B. Curtis, M.B. Chrissis and C. Weber (1993), *Capability Maturity Model for Software*, Version 1.1. Technical Report CMU/SEI-93-TR-24, Software Engineering Institute.

S.L. Pfleeger (2001), *Software Engineering – Theory and Practice*, 2nd Ed, Prentice Hall, Upper Saddle River, NJ, pp. 659.

T.M. Pigoski (1996), *Practical Software Maintenance*, Wiley, p. 384.

C. Potts (ed.), (1984), *Proceedings of the Software Process Workshop*, IEEE Computer Society Press, Egham, Surrey, Feb., Order No. 587.

V.T. Rajlich and K.H. Bennett (2000), A Staged Model for the Software Life Cycle, *Computer*, vol. 33, no. 7, July 2000, pp. 66–71.

W. W. Royce (1970), Managing the Development of Large Software Systems, *Proceedings of IEEE Westcon*, Los Angeles, CA, pp. 1–9.

G. Succi, J. Paulson and A. Eberlein (2001), Preliminary Results from an Empirical Study on the Growth of Open Source and Commercial Software Products, *EDSER-3 Wkshop, Co-located with ICSE 2001*, May 14–15, Toronto, Canada.

R.N. Taylor, F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf and M.Young (1988), *Foundations for the Arcadia Environment Architecture*, SIGSOFT Software Engineering Notes, ACM Press, New York, vol. 13, no. 5, pp. 1–13.

C. Tully (1989), Representing and Enacting the Software Process, *Proceedings of the 4th International Software Process Workshop*, ACM SlGSOFT Software Engineering Notes, ACM Press, June 1989.

W.M. Turski (1981), Specification as a Theory with Models in the Computer World and in the Real World, *Infotech State Art Rep.* vol. 9, no. 6, pp. 363–377.

W.M. Turski (1996), A Reference Model for the Smooth Growth of Software Systems, *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 599–600.

W.M. Turski (2000), An Essay on Software Engineering at the Turn of the Century, in T. Maibaum (ed.), *Fundamental Approaches to Software Engineering, Proceedings of the Third International Conference FASE 2000. LNCS 1783*, Springer-Verlag, Berlin, Germany, pp. 1–20.

A. van Lamsweerde 2000, Formal Specification: a Roadmap, in A. Finkelstein (ed.), *22nd International Conference on software Engineering, The Future of Software Engineering*, ACM Press, Limerick, Ireland, Order No. 592000-1, pp. 149–159.

N. Wirth (1971), Program Development by Stepwise Refinement, *Commun. ACM*, vol. 14, no. 4, pp. 221–222.

S. Zahran (1997), *Software Process Improvement – Practical Guidelines for Business Success, SEI Series in Software Engineering*, Addison-Wesley, Harlow, England.