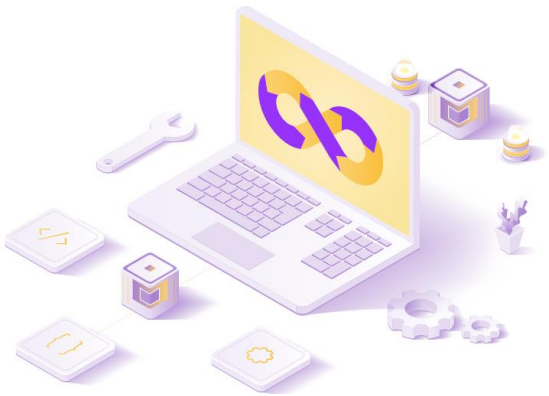
University of Buea,
College of Technology (COT), Department of Computer Engineering - 2023/2024 Academic Year

CEC418: Software Construction and Evolution

Enjoy Your Software Construction Ride!!



Unit 2

Unit 2: Software Evolution

2.1 Introduction

- ⇨ Large software systems are subject to changes in response to the dynamic nature of the environment where they are operating.
- ⇨ When a system is flexible and adaptable to such constant environment changes it may not be readily discovered.
- ⇨ This calls for a maintenance program accomplished in a series of system releases/versions.
- ⇨ Each release is a new version of the system with known error corrected and incorporating new or updated system facilities.
- ⇨ Software evolution plays an ever-increasing role in software development.
- ⇨ Programmers rarely build software from scratch but often spend more time in modifying existing software to provide new features to customers and fix defects in existing software.
- ⇨ Evolving software systems is often a time-consuming and error-prone process.
- ⇨ In fact, it is reported that 90% of the cost of a typical software system is incurred during the maintenance phase and a primary focus in software engineering involves issues relating to upgrading, migrating, and evolving existing software systems.
- ⇨ The term, software evolution dates back to 1976 when Belady and Lehman first coined this term.
- ⇨ Software evolution refers to the dynamic behaviour of software systems, as they are maintained and enhanced over their lifetimes.
- ⇨ Initially, Lehman and Belady's work on program growth dynamics and software evolution considered program evolution dynamics as an intrinsic characteristic of large programs.
- ⇨ Software evolution is particularly important as systems in organisations become longer-lived.
- ⇨ A key notion behind this seminal work by Belady and Lehman is the concept of ***software system entropy***.
- ⇨ The term ***entropy***, with a formal definition in physics relating to the amount of energy in a closed thermodynamic system is used to broadly represent a measure of the cost required to change a system or correct its natural disorder.
- ⇨ As such, this term has had significant appeal to software engineering researchers, since it suggests a set of reasons for software maintenance.
- ⇨ Law of program evolution have been derived out of experimental observation of a number of systems such as large operating system(s).

(1) Continuing change: A program that is being exercised in a real-world environment either continues to change or becomes less and less useful in that environment (i.e. no system is completely static, the only exception is the embedded system).

(2) Increasing complexity: As an evolving program change, its structure becomes more complex unless active efforts are made to avoid this phenomenon.

(3) Program Evolution: This is a self-regulating process and measurements of system attributes such as size, time between releases, number of reported errors etc. reviews statistically significant trend and invariances.

(4) Conservation of organisational stability over the rate of development of that program is approximately constant and independent of the resources devoted to system development.

(5) Conservation of familiarity: Over the lifetime of a system, the incremental system change in each release is approximately constant.

⇨ Later, many researchers have systematically studied software evolution by measuring concrete metrics about software over time.

⇨ Now that we accept the fact that software systems go through a continuing life cycle of evolution after the initial phase of requirement engineering, design, analysis, testing and validation, we describe an important aspect of software evolution|software changes in this unit.

2.2 Concepts and Principles

⇨ **Three** categories of software changes have been initially identified: corrective, adaptive, and perfective.

⇨ These categories were updated later and ISO/IEC14764 instead presents four types of changes: corrective, adaptive, perfective, and preventive.

2.2.1 Corrective Change

⇨ Corrective change refers software modifications initiated by software defects.

⇨ A defect can result from *design errors*, *logic errors*, and *coding errors*.

1. ***Design Errors***: software design does not fully align with the requirements specification. The faulty design leads to a software system that either incompletely or incorrectly implements the requested computational functionality.
2. ***Logic Errors***: a program behaves abnormally by terminating unexpectedly or producing wrong outputs. The abnormal behaviours are mainly due to flaws in software functionality implementations.

3. ***Coding Errors***: although a program can function well, it takes excessively high runtime or memory overhead before responding to user requests. Such failures may be caused by loose coding, or the absence of reasonable checks on computations performed.

2.2.2 Adaptive Change

- ⇨ Adaptive change is a change introduced to accommodate any modifications in the environment of a software product.
- ⇨ The term environment here refers to the totality of all conditions that influence the software product, including business rules, government policies, and software and hardware operating systems.
- ⇨ For example, when a library or platform developer may evolve its APIs, the corresponding adaptation may be required for client applications to handle such environment change.
- ⇨ As another example, when porting a mobile application from Android to iOS, mobile developers need to apply adaptive changes to translate the code from Java to Swift, so that the software is still compileable and executable on the new platform.

2.2.3 Perfective Change

- ⇨ Perfective change is the change undertaken to expand the existing requirements of a system.
- ⇨ When a software product becomes useful, users always expect to use it in new scenarios beyond the scope for which it was initially developed.
- ⇨ Such requirement expansion causes changes to either enhance existing system functionality or to add new features.
- ⇨ For instance, an image processing system is originally developed to process JPEG files, and later goes through a series of perfective changes to handle other formats, such as PNG and SVG.
- ⇨ The nature and characteristics of new feature additions is not necessarily easy to define and in fact understudied for that reason.

2.2.4 Preventive Change

- ⇨ Preventive change is the change applied to prevent malfunctions or to improve the maintainability of software.
- ⇨ According to Lehman's laws of software evolution, the long-term effect of corrective, adaptive, and perfective changes is deteriorating the software structure, while increasing entropy.
- ⇨ Preventive changes are usually applied to address the problems.
- ⇨ For instance, after developers fix some bugs and implement new features in an existing software product, the complexity of source code can increase to an unmanageable level.

- Through code refactoring|a series of behavior-preserving changes, developers can reduce code complexity, and increase the readability, reusability, and maintainability of software.

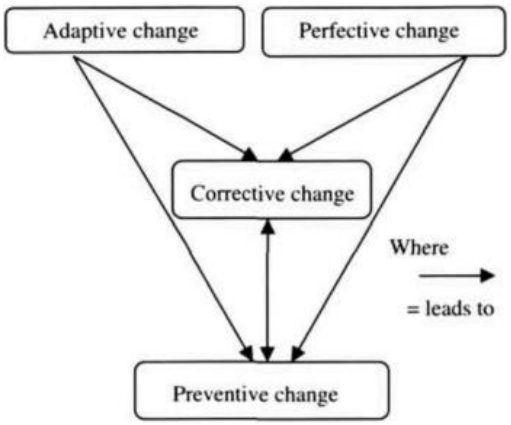


Figure 2.1: Potential relation between software changes.

- Figure 2.1 presents the potential relationships between different types of changes.
- Specifically, both adaptive changes and perfective changes may lead to the other two types of changes, because developers may introduce bugs or worsen code structures when adapting software to new environments or implementing new features.

2.3 Software Evolution Empirical Studies

- The characteristics of corrective, adaptive, perfective, and preventative changes are presented in this section using empirical studies and the process and techniques for updating software, respectively as presented in Figure 2.2.
- Next, regardless of change types, automation could reduce the manual effort of updating software.
- Therefore, we shall also discuss the topic of automated program transformation and interactive editing techniques for reducing repetitive edits.

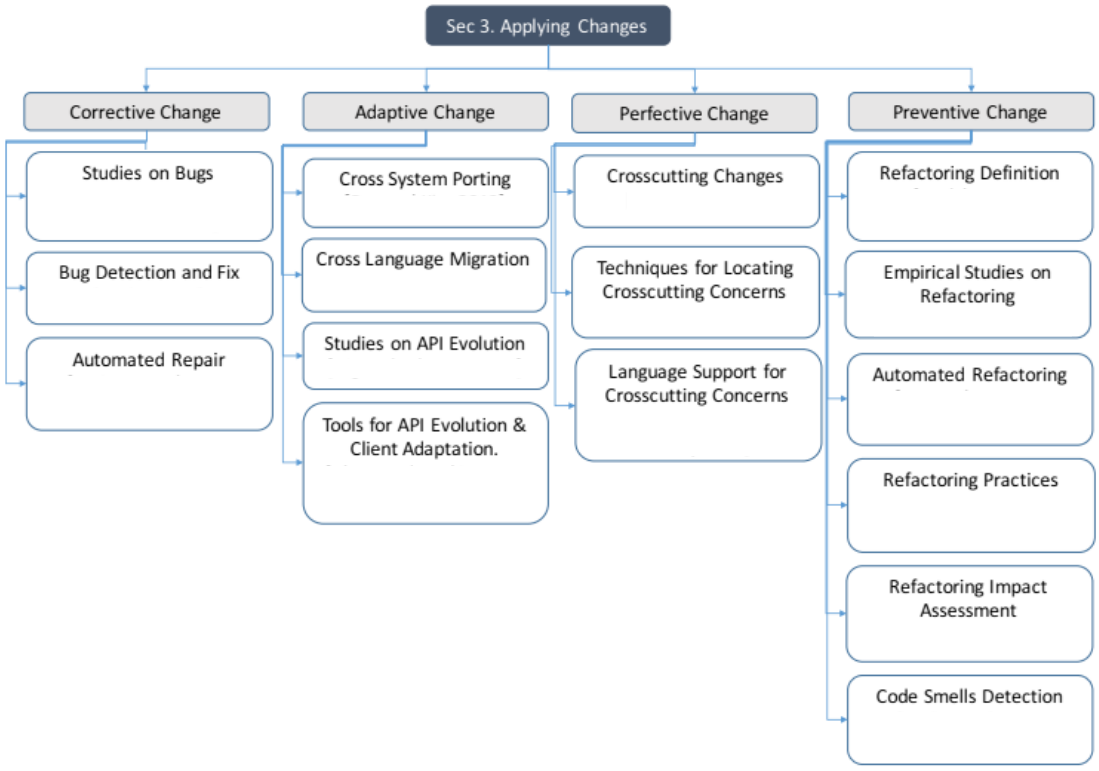


Figure 2.2: Applying Changes Categorised by Change Type.

2.4 Software SPE Program Classification

- ⇨ The SPE program classification scheme has been discussed many times and it identifies programs of types S, P and E, respectively.
- ⇨ It is the E-type that is of particular relevance in the context of evolution since such evolution is inevitable for a program of the E-type as long as it is in regular use.
- ⇨ S-type programs implement solutions to problems that can be completely and unambiguously specified, for which, in theory at least, a program implementation can be proven correct with respect to the specification.
- ⇨ In terms of the definition, issues of programming elegance and efficiency are not of concern.
- ⇨ S-type programs solve problems that are fully defined in an abstract and closed domain.
- ⇨ Examples include calculation of the lowest common multiple of two integers and the evaluation of certain mathematical expressions.
- ⇨ From the definition of S-type programs, the sole criteria for accepting satisfactory completion of such a program is that the completed product satisfies the specification that has been accepted as reflecting the behavioural properties expected from program execution.

- The specification constitutes the contract between supplier and stakeholder(s) (e.g., a representative of the prospective users).
- The definition presupposes existence of an appropriate formal specification, accepted by stakeholders as completely defining the problem to be solved or the need to be met.
- Whether, in some sense, the results of execution are useful, whether they provide a solution to the problem, will be of concern to both users and producers.
- However, once the specification has been contractually accepted and the product has been shown to satisfy it, the contract has, by definition, been fulfilled.
- If, nevertheless, results do not live up to expectations, that is, if the program properties need to be redefined, rectification requires a new, revised, specification to be drawn up and a new program to be developed.
- Depending on the details of changes required, such a new version may be developed from scratch or obtained by modification of that rejected.
- These views of S-type programs imply that the specification expresses all the properties that the program is required to possess to be deemed satisfactory or acceptable.
- The designation S was chosen to indicate the decisive role that the ***Specification*** plays in determining required product properties.
- An alternative interpretation, suggesting that S stands for ***static***, a property which, distinguishes it from the intrinsically evolutionary type E.
- Type E was originally defined as ". . . a program that mechanises a human or societal activity . . . ".
- This description was subsequently extended to include all programs that ". . . operate or address a problem or activity in the real world . . . ".
- Programs of this type are intrinsically evolutionary.
- They must be evolved as long as they remain in active use if they are to remain satisfactory to stakeholders; hence the designation E.
- The need and demand for continued change, correction, adaptation, enhancement and extension cannot be avoided if such software is to remain operationally satisfactory.
- It has long been recognised that any statement about the absolute correctness of an E-type program is meaningless in the context of E-type applications. Why?
- E-type specification and program are necessarily finite, but mirror applications and domains having an unbounded number of attributes.

- Moreover, some of the real-world attributes upon which successful execution may depend are subject to unpredictable change.
- But neither the specification nor the program can reflect such changes without (considerable) delay and then only with a risk that the program change introduced to correct the external change is incomplete or in error, in some way.
- The program will, however, be judged as satisfactory or otherwise in terms of the domain as it is each instance during execution.
- E-type programs are therefore characterised by an inherent uncertainty of the continual validity of the results of their execution when operating in a real-world domain, as stated in a Principle of Software Uncertainty.
- To make the classification as inclusive as possible a third type, type P, was also defined in the original schema.
- Subsequent reasoning suggested that as defined, in general, P-type programs satisfied the definition of one of the other two types.
- In particular, P-type programs will, if used in the real world, acquire E-type properties, for example, the intrinsic need to be continually evolved.
- Hence, they are not further considered here.

2.5 Areas of Software Related Evolution – Summary

- Evolution in the wider software related domain it is not confined to sequences of programs and artefacts of the programming process such as specifications, designs, and documentation.
- Other entities involved in software development and maintenance, such as programming and system development paradigms, languages, object definitions, usage domains and practices, applications and the very processes of software evolution themselves also evolve.
- These various evolution processes interact with and impact on one another. They overlap to some extent.
- To truly master software evolution one must understand and master the evolutionary properties of all entities individually and collectively.
- Ideally, and to the extent that this can be done, their evolution must be jointly planned, and controlled.
- In the first instance, however, one must focus on individual aspects.
- Consideration of the consequences of interactions between them can then follow, so are mentioned here only in passing.

- ⇨ Classification of areas of evolution in software and software related domains, including software technology, can be based on various alternative schemas.
- ⇨ The list that follows represents just one of these:

I. A basic common level at which the evolution phenomenon can be identified is in the implementation of programs and software systems from initial statement of an application concept to the final, released, installed and operational code and supporting documentation. Implementation of a set of changes and/or enhancements to an existing system may be viewed as a subarea of such ab initio development. The entire area and its processes, generally referred to as ***development***, relies heavily on feedback-based information and instruction employing both formal and informal mechanisms.

At the start of an E-type system development, for example, knowledge and understanding of the details of the application to be supported or the problem to be solved as well as approaches and methods of their solution are understood only in outline, are in many respects arbitrary. The relative benefits of alternative, more detailed objectives and approaches can often not be established except through trial and error and even the results of these are unlikely to be comprehensive or conclusive. The development process is a learning process in several dimensions. These include both the matter that is being addressed and the manner in which this is done. Such experiences-based feedback involving evaluation of past results leads to an evolutionary process. Software development (and change) activities typify the major constituents of area I in the present schema.

II. At the next level up, consider a sequence of versions, releases or upgrades of a program or software system. These incorporate changes that may rectify or remove defects, make provision for alternative operational environments, implement desired improvements or extensions to system functionality, performance, quality and so on. They are made available to users in the form of a patch or service package, a release, an upgrade or new version by means of what is generally termed a release process. Basically, all these changes are regarded by stakeholders as improvements to the program in one sense or another. Otherwise it would be unlikely that new releases of the same product would be purchased, installed and used. As already observed, this process is widely referred to as ***program maintenance***. Many people over the years have, however, recognised that the term maintenance is inappropriate, even misleading, in the software context. After all, as used in other areas, maintenance describes an activity seeking, in general, to rectify wear, tear or other deterioration that has developed in, say, a physical artefact such as a car or an aircraft. The purpose is to return the latter as nearly as possible to its original initial, pristine state. But software as such

is not subject to wear and tear. If one excludes hardware failure, it does not deteriorate per se. The "deterioration" that users sense arises from, for example, changes in the purpose for which the software was acquired, the changing properties of the operational domain, advancing technology, desire for business growth or the emergence of competitive products. Some part of the need for change will be a consequence of assumptions, that have become invalid as a result of external changes but are, nevertheless still implicitly or explicitly reflected in the software. There are, however, two contexts in which the term maintenance is valid in the software context. In evolving the system, one maintains the validity of the assumption set reflected in the software and its documentation, and its satisfactory behaviour as judged by the stakeholders, users for example. In short, software is evolved to maintain the validity of its embedded assumption set, its behaviour under execution, the satisfaction of its stakeholders and its compatibility with the world as it now is or as expected to be.

The preceding discussion has made a specific distinction between areas I and II.

The first covers the development of an entire system ab initio (or of a change to an operational version of a system) whereas the second addresses adaptation and enhancement of a developed system over a sequence of releases. Changes and additions implementing the release also require specification, design and implementation. Thus, area II involves area I activity and evolution.

III. E-type software supports E-type applications and the latter must also evolve. Applications span a wide application spectrum from pure computation to co-operative, communication-based, computer supported human/organisation/machine activity.

Introduction into use of successive software versions by the user community inevitably changes the application, that is, the activity supported by the software. It also changes the operational domain. It generally opens up new opportunities for functional and performance enhancement, increased effectiveness and efficiency, and cost reduction. New standards of satisfactory operation are recognised and changes to the system are initiated to achieve them. Installation and operation of an E-type system, drives an unending process of system and application evolution.

As an application evolves so, inevitably, do the processes and domains within which it operates or is pursued. Installation and operation of a new or improved system will also impact other processes and domains with which the application, its processes and its operational domain interacts or is associated. Evolution of an element of what constitutes a wider, global, system ripples out and spreads throughout that wider context and its encompassing domains.

Evolution of the processes and domains referred to in the previous paragraph may be regarded as subareas of area III or as areas in their own right. For simplicity, the former view is assumed here.

IV. The process of software evolution refers, in general, to the aggregate of all activities that implement one or other of the above levels of evolution. It is variously estimated that between 60 and 80% of the resource applied to evolve a software system over its lifetime is incurred after the first release. It may reach 95% or more in defence-related applications where every system improvement is likely to trigger a search for countermeasures of one sort (defensive) or another (offensive). Similar reactions occur also in business applications as a result of competitive pressures on both suppliers and users.

Societal dependence on computers and on the software that provides society with functional and computational power is increasing at ever increasing rates. This leads to sound reliability, economic and logistic reasons to improve the process to achieve lower costs, better functionality and performance, improved quality, faster response to user needs for change and so on. The software evolution process must be improved to reduce human exposure to the consequences of high costs, computer malfunction and delays in adaptation to changing circumstances. All demand improvement of the means whereby evolution is achieved. And such improvement must address the needs of each application area and domain. The process evolves, driven by experience and the changing needs and opportunities of a dynamic, continually changing, real world, not because software as such deteriorates.

V. Achieving full understanding and mastery of the software evolution process remains a distant goal. Process modelling, using a variety of approaches, is a useful tool for its study, control and improvement. Use of these techniques is likely to be useful because of the complex nature of the process feedback system, structure and mechanisms. Given the continuous and intimate involvement of human implementers and managers, the models facilitate reasoning, for example, about the process and its product, the exploration of alternatives and change impact assessment. The process evolves. So, must models of it.

2.6 Area I: Software Ab Initio (Changes to an Existing Program) Implementation

- ⇨ Ab initio implementation of a program or changes to an existing program requires execution of a series of discrete, iterated, and often overlapping steps by interacting individuals and teams, using a variety of, in general, computer-based tools.
- ⇨ Their joint action over a period of weeks, months or 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 from application area to application area and from organisation to organisation.
- ⇨ The Waterfall model, probably the first published model of the software process, and its subsequent refinements, identifies various process steps.
- ⇨ In current terminology, these are described by terms such as requirements elicitation and analysis, specification, high level design, detailed design, coding, unit verification and validation (V&V), integration, system V&V, documentation and so on.
- ⇨ Execution of these steps is not purely sequential. Execution of any step, for example, may reveal an error in one or more earlier steps or may suggest an improvement to the detailed design or a correction to underlying assumptions associated with them.
- ⇨ The process may also be described as one of successive transformation.
- ⇨ It is driven by human analytic and creative power as influenced and modified by developing insight and understanding.
- ⇨ Information garnered from later steps leads to iteration over earlier steps.
- ⇨ Changes in the external world that must be reflected in the system, also serve as drivers.
- ⇨ Successive steps are likely to operate at many different conceptual and linguistic levels of abstraction and require different transformational techniques.
- ⇨ The aggregated effect is, effectively, a process of successive refinement that transforms the original application concept into its operational implementation, the software system.
- ⇨ It's in this sense that the use of the term development is legitimate.
- ⇨ A high-level view of what is going on suggests, however, that the process must also be recognised as evolutionary since the transformational steps are elements in a successive-transformation paradigm that satisfies the definition of evolution, given in earlier
- ⇨ Evolution is seen as the continued progressive change in the properties or characteristics of some material or abstract entity or of a set of entities.
- ⇨ They add or modify functional and computational detail in a reification process as described in the LST (**L**ehman, **S**tenning, **T**urski) paradigm presented in Figure 2.3.
- ⇨ That view may be appearing too high a level of abstraction, too remote from the complexity of the industrial software process to be relevant in the present context.
- ⇨ It is, however, briefly discussed here because it provides insight into the nature of the software development process and helps illustrate issues that emerge during software ab initio implementation.

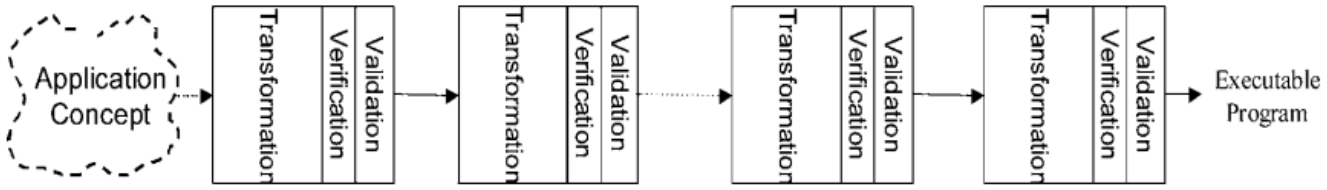


Figure 2.3: LST Program Implementation Paradigm.

2.6.1 The LST Paradigm

- ⇨ The LST paradigm identifies each step in the implementation process as the transformation of a specification into a model of that specification, as depicted in Figure 2.3.
- ⇨ Note that as shown, the structure shown applies only to S-type development and even their failure of the verification or validation activities as briefly discussed below can lead to iteration around or between individual steps.
- ⇨ Alternatively, the paradigm may be described as the transformation of a design into an implementation.
- ⇨ Where S-type development is possible, the transformation step is not finalised until its output has been verified as being correct in the strict mathematical sense where correctness is a precise relationship between input (i.e., the specification) and output (i.e., the implementation).
- ⇨ That is, the transformation process must, if faithfully followed, eventually lead to a contractually satisfactory program since, by definition the specification expresses all the properties that the program is required to possess.
- ⇨ For E-type applications, on the other hand, there can, inherently, be no guarantee that the transformation process will yield a satisfactory program.
- ⇨ That is so, even in the unlikely event that a formal specification together with means for demonstrating the correctness relationship are available for the system as a whole and/or each of the system elements at some level of detail.
- ⇨ Intrinsically, E-type programs address applications for which the specification cannot, at each moment in time, reflect fully and unambiguously all the properties that the program must display to remain satisfactory.
- ⇨ Hence, even though parts, or even the whole, of an E-type program may be shown to satisfy a formal specification, use of the term correctness in the E-type application domain is meaningless.
- ⇨ At best, validation by means such as, for example, testing under appropriate conditions can be used to increase confidence that the process output will prove satisfactory when executed in the real world.
- ⇨ Validation can, however, not be absolute.

- ⇨ As briefly stated above, the LST paradigm requires demonstration that the model output of each step satisfies its specification.
- ⇨ Even in the S-type context, each step also includes validation – termed beauty contest in the LST paper.
- ⇨ This serves the purpose of confirming (or otherwise) that, to the stage of refinement reached, the current model is likely to lead to a product that will satisfy the purpose for which it is being developed.
- ⇨ Validation failure implies a weakness, suggesting that the final product may be unsatisfactory in the context of the intended purpose and domain.
- ⇨ The problem may have arisen from changes in the purpose or the domain of the intended application since the specification was agreed.
- ⇨ Other potential triggers of validation failure are oversights in generating it or from introduction by the transformation process of properties that are considered unacceptable in terms of the intended purpose though they are neither excluded by the specification nor incompatible with it.
- ⇨ Whatever the source of failure, the source of the unacceptable properties must be identified and rectified by changing one or more of the transformation procedures, their inputs or the specification.

2.6.2 The S-Type Software

- ⇨ By definition, S-type software is, contractually, acceptable if it has been shown to satisfy its specification.
- ⇨ Its properties will reflect the specification in its entirety and verification is sufficient to determine its contractual acceptability.
- ⇨ This implies that the completed specification is believed to be complete as far as the intended purpose of the program is concerned; that the problem to be addressed is fully understood and unchanging.
- ⇨ Thereafter, it is primarily the knowledge, understanding and experience of the implementers that drives the implementation process.
- ⇨ Learning during the course of that process is largely restricted to determination of methods of solution, or of identification and selection of the best method, in the context of constraints encountered in the solution domain.
- ⇨ Feedback in S-type program implementation is restricted. It may well be present at a low level of development such as requirements design, coding or documentation.
- ⇨ It is unlikely to dominate the implementation process.

- ⇨ Though verification is sufficient for S-type programs from a contractual point of view, as briefly indicated above, business considerations require its validation since it determines the likely acceptability of the final product.
- ⇨ Technically it shifts the responsibility for product acceptance to the client since, contractually, its implementation was terminated by satisfactory completion of verification, based on acceptance of the specification by the client.
- ⇨ If the latter proves deficient, refinement is required, the development process is abandoned and a new one based on a new specification is initiated.
- ⇨ In practice, that new process may well take advantage of the earlier one.
- ⇨ Nevertheless, conceptually the development from an initial concept and the derived specification, consists of a series of discontinuous open loop processes rather than a feedback-driven continuous evolutionary process.
- ⇨ For S-type programs too, evolution occurs but over a sequence or generations of instances not, as for type E, over the lifecycle of a single, changing, instance.

2.6.3 E-Type Software

- ⇨ In the case of E-type systems the problem to be solved relates to the real world.
- ⇨ Thus, an application (or change to an application) to be developed and the domain (or change of domain) within which it is to be solved are not, in general, clearly and uniquely defined.
- ⇨ There will always be fuzzy aspects. If they are recognised, they may be firmed up as knowledge and deeper understanding of the problem or application, of the operational domain and of acceptable solutions.
- ⇨ This normally occurs during development as managers, developers and, possibly, clients take arbitrary decisions.
- ⇨ In this situation, feedback plays a crucial role.
- ⇨ Parts of the multi-dimensional domain boundary of an E-type system will be well defined by, for example, prior practice or related experience. Its operational range will, therefore, be determined.
- ⇨ Other parts of the boundary are adopted on the basis of compromise or recognised constraints. Still others will be uncertain, undecidable, possibly introducing inconsistencies.
- ⇨ This situation may be explicitly acknowledged or remain unrecognised until exposed by chance or during system operation.
- ⇨ The application domains have unclear, fuzzy, possibly fluctuating, boundaries that must be continually reviewed.

- ⇨ In relation to evolution, any initial fuzziness in development involves at least two separate and distinct aspects.
- ⇨ The first relates to the intrinsic unbounded nature of any application and its operational domain.
- ⇨ Initially the latter is neither precisely defined nor bounded in extent and in detail. Such uncertainty is resolved by a bounding process that determines the domain over which the application is to be valid, used and supported, or to which it is to be adapted to provide a satisfactory solution in some defined time frame at acceptable cost.
- ⇨ However, once the system is in operation, a need or desire to extend the area of validity to regions of the domain or to detail previously excluded is very likely to arise.
- ⇨ If not properly addressed, the latter, in particular, will become irritants and performance inhibitors.
- ⇨ Feeding back the need for modification or domain extension to the implementers and requiring them to satisfy newly emerging needs, changing constraints or changed environmental circumstances exerts pressure for system evolution.
- ⇨ The second concern relates to the number of functional and other properties to be implemented in a system.
- ⇨ As anyone with experience in systems analysis, specification and design knows, the list of properties and function that could be included in a system is potentially unbounded.
- ⇨ It is always in excess of what can be accommodated within the resources and time allocated for system implementation.
- ⇨ Thus, from the point of view of potential coverage, the boundaries of the final system are arbitrary.
- ⇨ But, unlike those of the domain, once developed and installed they become solid, determined, at any moment in time, by the installed software and hardware.
- ⇨ A user requiring a facility not included within this boundary may, in the first instance, use stand-alone software to provide the required facility.
- ⇨ It may be possible to couple such software tightly to the system for greater convenience in co-operative execution.
- ⇨ But, however the additional function is invoked and the results of execution passed to the main system, additional execution overhead, time delays, performance and reliability penalties and sources of error are incurred.
- ⇨ The omissions become onerous, a source of performance inhibitors and user dissatisfaction.
- ⇨ Continuing demands and proposals for system extension constitute the inevitable result.

- ⇨ The history of automatic computation is rich with examples of function first developed and exploited as standalone 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.
- ⇨ In some instances, the migration continues until the function is implemented in hardware (chips) as exemplified in many present-day systems by language and graphics support.
- ⇨ The evolving computing system may be seen as an expanding universe with an inward drift of function from the domain to the core of the system.
- ⇨ The process is driven by feedback about the strengths, weaknesses, effectiveness and potential of the system as recognised during development and use of the system or its outputs.
- ⇨ E-type programs and systems are the entities of ultimate concern of the software industry and of the technologies it uses.
- ⇨ So is the process of system evolution over versions, releases and upgrades. As already observed, the latter maintains program system applicability and viability, its value in a changing world.
- ⇨ E-type program development and adaptation is not amenable to coverage by a complete and exhaustive theory.
- ⇨ In part this is due to human involvement in the applications. Systems reflect and embed human viewpoints, policies and values that are subject to change as organisations and society evolve.
- ⇨ Policies and values, when formalised involve a degree of arbitrariness and approximation.
- ⇨ It is also related to the arbitrariness of decision and procedures in business, manufacturing, government, public and service sectors and so on. Successive refinement and adjustment of all these is a source for continual program evolution.
- ⇨ Finally, the fact E-type development cannot be covered by an exhaustive theory also relates to the actual or potential imprecision of the operational domain boundaries already mentioned.
- ⇨ Implementation and use of the systems are thus essentially a learning experience.
- ⇨ The system is intrinsically evolutionary.
- ⇨ Any program is a bounded, discrete and static reflection of an unbounded, dynamic application domain.
- ⇨ The boundaries and other attributes of the latter are intrinsically fuzzy but are progressively firmed up during development, installation, usage and evolution by operational, economic, time, and technology considerations and constraints, the striving for growth of human individuals and organisations and so on.

- ⇨ Some boundaries are determined explicitly in processes such as application proposals, requirements analysis and specification; others arise during the actual design and development activity and involve explicit or implicit assumptions adopted and embedded in the system during the evolution process.
- ⇨ Fixing detailed properties, such as those of human/system interfaces or interactions between people and the operational system will include trial and error.
- ⇨ Even in applications where there is long and rich experience in the procedures and interactions, it must be understood that introduction of a computer system or even replacement of such a system previously used, introduces new human reactions, new problems, new twists that are likely to affect the requirements.
- ⇨ Implementation of fine detail cannot be based on previous experience, one-off observation, elicitation of requirements, intuition, conjecture or statistics alone.
- ⇨ It arises from continuing human experience, the application of judgement and decision by development staff and users.
- ⇨ Most development activity changes perception and understanding of the application, 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 to its final goal by experience and learning based feedback to produce an operational system that is satisfactory by criteria considered appropriate during development and on its completion and acceptance.
- ⇨ Subsequent evolution over versions or releases is, however, inevitable for maintenance of that satisfaction.

2.6.4 Software Component-Based Implementations

- ⇨ The LST paradigm described above and distinction between the nature of the evolution process in the context of S- and E-type systems may appear abstract, without practical implications. That is certainly not so.
- ⇨ An increasing trend to the use of component-based implementation, reuse and COTS will make their practical significance more widely appreciated. Why?
- ⇨ Because these approaches will be based, at least conceptually, on elements, that must, implicitly and in isolation, at least, have been assumed to be of type S, that is, unambiguously specified, fully visibly.
- ⇨ In practice, however, to remain compatible with and correctly implement their intended functionality in the system within which they are embedded and integrated, to permit system adaptability to evolving

needs and a changing environment such components require components to have the malleability, flexibility and evolutionary characteristics of E-type systems.

⇨ Whether COTS or in-house reuse, components must also evolve.

2.7 Software Area II: Software System Evolution

⇨ The relationship of E-type software to the real world may be described as a model-like reflection.

⇨ In the accepted mathematical meaning of the term model, both the application in its real-world operational domain and the program implementation are models of a common specification obtained by abstraction and reification from a vision and understanding of the application.

⇨ These relationships are depicted in Figure 2.4.

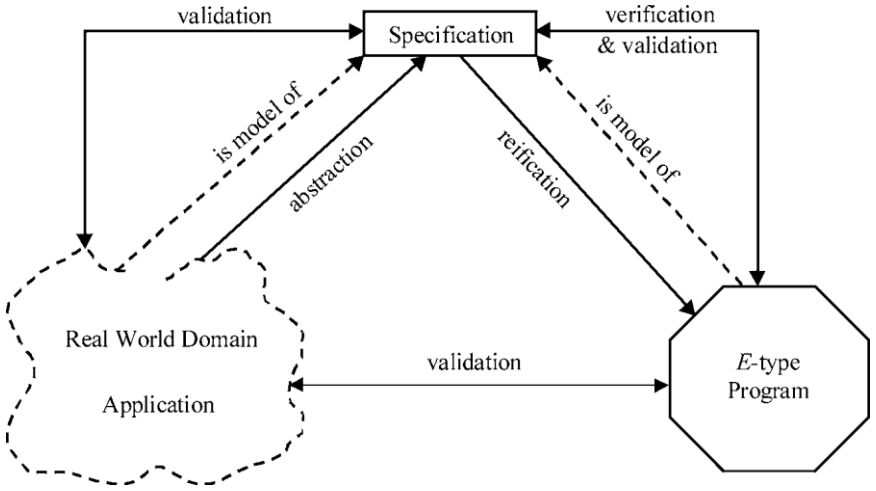


Figure 2.4: Relationship Between the Domain, Specification and Program.

⇨ Program and application may, and normally will, also possess additional properties that are not incompatible with the specification.

⇨ The term model-like reflection is used to convey the fact that in addition to reflecting all properties identified by its specification, a program must remain compatible with the real-world application in its operational domain.

⇨ Hence, as implied above, software maintenance may be defined as maintenance of compatibility (e.g., consistency) between a program and the various domains within which it is developed and used.

⇨ Continuing maintenance reconciles the system with its application, operational domains and stakeholders' views on a continuing basis as they continually change.

⇨ Area II evolution seeks to achieve and maintain compatibility between them over a sequence of versions, upgrades or releases.

⇨ Each of these culminates in a release process that makes the evolving system available to users.

- ⇨ As repeatedly indicated, evolution is intrinsic to E-type software, the systems and organisations they execute in and with and the applications they address.
- ⇨ Given the appropriate economic, social and technical conditions, all co-evolve over a succession of states that improve and extend the operational effectiveness in one way or another.
- ⇨ Each version of the software represents one step in an ongoing, complex, evolution process, one instance of evolution process output.
- ⇨ The sequence of releases transforms the system away from one satisfying the original concept to one that successively supports changing circumstances, needs and opportunities in a changing world.
- ⇨ If conditions to support such evolution do not exist, then the program will gradually lapse into uselessness because of the widening gap that develops between the real world as mirrored by the program and the real world as it now.
- ⇨ As mentioned above and described in a number of papers since then, the study of software system evolution emerged from a wider software process study.
- ⇨ Inter alia, the original study examined empirical data on the growth of the IBM OS/360 operating system.
- ⇨ It concluded that system evolution, as measured for example by growth in size over successive releases, displayed a remarkable degree of stability.
- ⇨ This was unlikely to have been the consequences of planning and decisions by process participants or actively sought by them but was more likely to be the consequence of dynamic forces associated with the feedback nature of the software process.
- ⇨ The empirical data that first suggested this conclusion is exemplified by Figure 2.3 above.
- ⇨ This plots system size measured in number of modules – a surrogate for the functional power of the system – against release sequence number (RSN).
- ⇨ This relatively stable growth trend was ended by a period of instability that preceded breakup of the system into separate systems, VS/1 and VS/2.
- ⇨ When plotted over releases, the long-term growth trend of OS/360 up to RSN 20, as displayed in Figure 2.3, was close to linear. The superimposed ripple suggests self-stabilisation around that trend.
- ⇨ It is described as self -stabilisation because no indication could be found that management sought linear growth, that, in fact, growth control considerations played any conscious part in defining individual release content.

- ⇨ The latter was a consequence of the aggregation of individual management considerations and decisions; based on many inputs from many sources and coordinated, to a greater or lesser extent, by release management.
- ⇨ The stabilisation suggested by the ripple must reflect the consequence of the organisational integration of all these inputs in and via a complex, multi-loop feedback structure mediated by managers, policies, established and ad hoc practices and other mechanisms.
- ⇨ The stabilisation phenomenon triggered the first realisation that feedback might be playing a role in determining the pattern of growth in functional power of an evolving system, and other attributes, of system evolution.

2.8 Software Area III: Evolution of the Application in its Domain

- ⇨ Continual evolution is not confined to the software or even to a wider application system within which the software may be embedded.
- ⇨ It is inherent in the very nature of computer application. The activity that software supports and the problems it solves, also evolve.
- ⇨ Such evolution is, in part, driven by human aspiration for improvement and growth. But more subtle forces are also at play.
- ⇨ Installation and use of the system changes both the activity being supported and the domain within which it is pursued.
- ⇨ When installed and operational, the output of the process that developed the software ab initio or evolved an existing system changes the attributes of the application and the domain that defined the process in the first place.
- ⇨ In association with the application and the operational domain as defined and bounded, the development process, as outlined in the discussion of area I, clearly constitutes a feedback loop as illustrated in Figure 2.5.
- ⇨ Depending on the manner in and degree to which changes impact use of the system and on loop characteristics such as the amplification/attenuation and delays, the overall feedback at this level can be negative or positive resulting in stable growth or instability.
- ⇨ 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 part of the phenomenon and process of co-evolution as explained above and elsewhere.

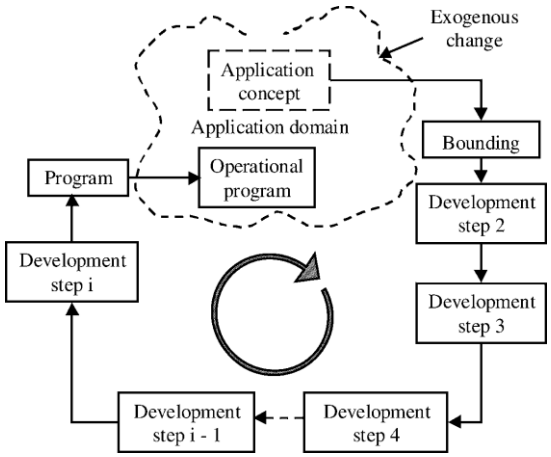


Figure 2.5: Evolution of the application in its domain as an iterative feedback system. Internal process loops not shown.

- ⇨ 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, exchanging and sharing data, invoking services from one another, making demands on common budgets.
- ⇨ The inescapable trend is towards the integration of all internal services, with the goal, for example, of minimising the need for human involvement in information handling and communication, to avoid delays and errors and to increase security.
- ⇨ And such integration is seen as needing to gradually extend to client's systems, their customers, suppliers and other external organisations.
- ⇨ With this scenario, the rate at which an organisation can change, grow and be adapted to changing conditions, new opportunities, competitive challenges and advancing technology increasingly 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 so tomorrow, organisations, whatever their activity or sphere of operation, the domains within which they operate, the activities they pursue, the technologies they employ and the computer software which links, co-ordinates and ties all together will be interdependent.
- ⇨ All must 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 co-evolution, the means whereby it is achieved.
- ⇨ Change to any constituent element of the global system will almost invariably imply software change.