2

TAXONOMY OF SOFTWARE MAINTENANCE AND EVOLUTION

Evolution is not a force but a process. Not a cause but a law.

—John Morley

2.1 GENERAL IDEA

In the early 1970s, the term "maintenance" was used to refer to tasks for making intentional modifications to the existing software at IBM. Those who performed maintenance tasks had not carried out the software development work. The idea behind having a different set of personnel to carry out maintenance work was to free the development engineers from support activities. The aforementioned model continued to influence the activities that are collectively known as "software maintenance." In circa 1972, in his landmark article "The Maintenance 'Iceberg'," R. G. Canning [1] used the iceberg metaphor to describe the enormous mass of potential problems facing practitioners of software maintenance. Practitioners took a narrow view of maintenance as correcting errors and expanding the functionalities of the system. In other words, maintenance consisted of two kinds of activities: correcting errors and enhancing functionalities of the software. Hence, maintenance can be inappropriately seen as a continuation of software development with an extra input—the existing software system [2].

The ISO/IEC 14764 standard [3] defines software maintenance as "… the totality of activities required to provide cost-effective support to a software system. Activities

Software Evolution and Maintenance: A Practitioner's Approach, First Edition.
Priyadarshi Tripathy and Kshirasagar Naik.

are performed during the pre-delivery stage as well as the post-delivery stage" (p. 4). Post-delivery activities include changing software, providing training, and operating a help desk. Pre-delivery activities include planning for post-delivery operations. During the development process, maintainability is specified, reviewed, and controlled. If this is done successfully, the maintainability of the software will improve during the post-delivery stage. The standard further defines software maintainability as "… the capability of the software product to be modified. Modification may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specification" (p. 3).

A major difference exists between software maintenance and software development: maintenance is event driven, whereas development is requirements driven [4]. A process for software development begins with the objective of designing and implementing a system to deliver certain functional and nonfunctional requirements. On the other hand, a maintenance task is scheduled in response to an event. Reception of a *change request* from a customer is a kind of event that can trigger software maintenance. Similarly, recognition of the needs to fix a set of bugs is considered another kind of event. Events originate from both the customers and from within the developed organization. Generally, the inputs that invoke maintenance activities are unscheduled events; execution of the actual maintenance activities might be scheduled according to a plan, but the events that initiate maintenance activities occur randomly. A maintenance activity accepts some existing artifacts as inputs and generates some new and/or modified artifacts.

Now we further explain the idea of a maintenance activity taking in an input and producing an output. In general, an investigation activity is the first activity in a maintenance process. In an investigation activity, a maintenance engineer evaluates the nature of the events, say, a change request (CR). Finding the impact of executing the CR is an example of investigation activity. Upon completion of the first activity, the organization decides whether or not to proceed with the modification activity.

In the following subsections, we explain maintenance activities from three viewpoints:

- Intention-based classification of software maintenance activities;
- Activity-based classification of software maintenance activities; and
- Evidence-based classification of software maintenance activities.

2.1.1 Intention-Based Classification of Software Maintenance

In the intention-based classification, one categorizes maintenance activities into four groups based on what we intend to achieve with those activities [5–7]. Based on the Standard for Software Engineering–Software Maintenance, ISO/IEC 14764 [3], the four categories of maintenance activities are corrective, adaptive, perfective, and preventive as explained in the following.

Corrective maintenance. The purpose of corrective maintenance is to correct failures: processing failures and performance failures. A program producing a wrong output is an example of processing failure. Similarly, a program not being able to

meet real-time requirements is an example of performance failure. The process of corrective maintenance includes isolation and correction of defective elements in the software. The software product is repaired to satisfy requirements. There is a variety of situations that can be described as corrective maintenance such as correcting a program that aborts or produces incorrect results. Basically, corrective maintenance is a *reactive* process, which means that corrective maintenance is performed after detecting defects with the system.

Adaptive maintenance. The purpose of adaptive maintenance is to enable the system to adapt to changes in its data environment or processing environment. This process modifies the software to properly interface with a changing or changed environment. Adaptive maintenance includes system changes, additions, deletions, modifications, extensions, and enhancements to meet the evolving needs of the environment in which the system must operate. Some generic examples are: (i) changing the system to support new hardware configuration; (ii) converting the system from batch to online operation; and (iii) changing the system to be compatible with other applications. A more concrete example is: an application software on a smartphone can be enhanced to support WiFi-based communication in addition to its present Third Generation (3G) cellular communication.

Perfective maintenance. The purpose of perfective maintenance is to make a variety of improvements, namely, user experience, processing efficiency, and maintainability. For example, the program outputs can be made more readable for better user experience; the program can be modified to make it faster, thereby increasing the processing efficiency; and the program can be restructured to improve its readability, thereby increasing its maintainability. In general, activities for perfective maintenance include restructuring of the code, creating and updating documentations, and tuning the system to improve performance. It is also called "maintenance for the sake of maintenance" or "reengineering" [8].

Preventive maintenance. The purpose of preventive maintenance is to prevent problems from occurring by modifying software products. Basically, one should look ahead, identify future risks and unknown problems, and take actions so that those problems do not occur. For example, good programming styles can reduce the impact of change, thereby reducing the number of failures [9]. Therefore, the program can be restructured to achieve good styles to make later program comprehension easier. Preventive maintenance is very often performed on safety critical and high available software systems [10–13]. The concept of "software rejuvenation" is a preventive maintenance measure to prevent, or at least postpone, the occurrences of failures due to continuously running the software system. *Software rejuvenation* is a proactive fault management technique aimed at cleaning up the system internal state to prevent the occurrence of more severe crash in the future. It involves occasionally terminating an application or a system, cleaning its internal state, and restarting it. Rejuvenation may increase the downtime of the application; however, it prevents the occurrence of more severe and costly failures. In a safety critical environment, the necessity of performing preventive maintenance is evident from the example of control software for Patriot missile: "On 21 February, the office sent out a warning that 'very long running time' could affect the targeting accuracy. The troops were not told, however, how many

hours 'very long' was or that it would help to switch the computer off and on again after 8 hours." (p. 1347 of Ref. [14]). The purpose of software maintenance activities of preventive maintenance of a safety critical software system is to eliminate hazard or reduce their associated risk to an acceptable level. Note that a *hazard* is a state of a system or a physical situation which, when combined with certain environment conditions, could lead to an accident. A hazard is a prerequisite for an accident or mishap.

2.1.2 Activity-Based Classification of Software Maintenance

In the intention-based classification of maintenance activities, the intention of an activity depends upon the reason for the change [7]. On the other hand, Kitchenham et al. [4] organize maintenance modification activities based on the maintenance activity. The authors classify the maintenance modification activities into two categories: corrections and enhancements.

- *Corrections.* Activities in this category are designed to fix defects in the system, where a defect is a discrepancy between the expected behavior and the actual behavior of the system.
- *Enhancements.* Activities in this category are designed to effect changes to the system. The changes to the system do not necessarily modify the behavior of the system. This category of activities is further divided into three subcategories as follows:
  - enhancement activities that modify some of the existing requirements implemented by the system;
  - enhancement activities that add new system requirements; and
  - enhancement activities that modify the implementation without changing the requirements implemented by the system.

Now one can find a mapping between Swanson's terminology and Kitchenham's terminology. Enhancement activities which are necessary to change the implementations of existing requirements are similar to Swanson's idea of perfective maintenance. Enhancement activities, which add new requirements to a system, are similar to the idea of adaptive maintenance. Enhancement activities which do not impact requirements but merely affect the system implementation appear to be similar to preventive maintenance.

2.1.3 Evidence-Based Classification of Software Maintenance

The intention-based classification of maintenance activities was further refined by Chapin et al. [15]. The objectives of the classification are as follows:

- base the classification on objective evidence that can be measured from observations and comparisons of software before and after modifications;
- set the coarseness of the classification to truly reflect a representative mix of observed activities;

Human resource. Human resource includes personnel from the maintenance and client organizations. Maintenance organization personnel include managers and engineers, whereas client organization personnel include customers and users. The management negotiates with the customers to find out the SLA, scheduling of requirement enhancements, and cost.

In general, maintenance tasks are perceived to be less challenging, and, hence, less well rewarded than original work. Often maintenance tasks are partly assigned to newly recruited programmers, which has a significant impact on productivity and quality. A novice programmer may introduce new defects while resolving an incident because of an absence in understanding the whole system. Normally, more skilled maintenance personnel produce more and better quality works.

Separation between development staff and maintenance staff impacts the maintenance process. On one hand, there is no real separation between maintenance and development. In such a scenario, the product undergoes continual evolution. The developers incorporate maintenance activities into a continuing process for planned enhancements. In this case, the tools and procedures are the same for both the development and maintenance activities. On the other hand, there are maintenance organizations which operate with minimal interactions with the development departments. Occasionally, the maintenance group may not be located in the same organization that produced the software. In such a scenario, maintenance engineers may need specially designed tools. Maintenance managers need to focus on the aforementioned issues when signing SLAs. Finally, the following user and customer issues affect maintenance:

- *Size.* The size of the customer base and the number of licenses they hold affect the amount of effort needed to support a system.
- *Variability.* High variability in the customer base impacts the scope of maintenance tasks.
- *Common goals.* The extent to which the users and the customer have common goals affects the SLAs. Ultimately, customers fund maintenance activities. If the customers do not have a good understanding of the requirements of the actual users, some SLAs may not be appropriate to the end users.

2.3 EVOLUTION OF SOFTWARE SYSTEMS

The term *evolution* was used by Mark I. Halpern in circa 1965 to define the dynamic growth of software [20]. It attracted more attention in the 1980s after Meir M. Lehman proposed eight broad principles about how certain types of software systems evolve [21–24]. Bennett and Rajlich [25] researched the term “software evolution,” but found no widely accepted definition of the term. However, some researchers and

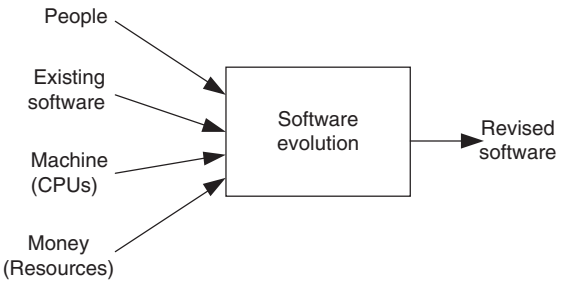

FIGURE 2.4 Inputs and outputs of software evolution. From Reference 26. © 1988 John Wiley & Sons

practitioners used the term software evolution as a substitute for the term software maintenance. Lowell Jay Arthur distinguished the two terms as follows:

- *Maintenance* means preserving software from decline or failure.
- *Evolution* means a continuously changing software from a worse state to a better state (p. 1 of Ref. [26]). Software evolution is like a computer program, with inputs, processes, and outputs (p. 246 of Ref. [26]) (See Figure 2.4).

Keith H. Bennett and Jie Xu [27] use "maintenance" for all post-delivery support, whereas they use "evolution" to refer to perfective modifications—modifications triggered by changes in requirements. Ned Chapin defines software evolution as:

> "the applications of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version, where the time period between versions may last from less than a minute to decades, together with the associated quality assurance activities and processes, and with the management of the activities and processes" (p. 21 of Ref. [15]).

The majority of software maintenance changes are concerned with evolutions triggered by user requests for changes in the requirements. The following are the key properties of software evolution as desired by the stakeholders:

- Changes are accomplished quickly and in a cost-effective manner.
- The reliability of the software should not be degraded by those changes.
- The maintainability of the system should not degrade. Otherwise, future changes will be more expensive to carry out.

Software evolution is studied with two broad, complementary approaches, namely, *explanatory* and *process improvement*, and those describe the *what* and *how* aspects, respectively, of software evolution.

- *Explanatory (what/why).* This approach attempts to explain the causes of software evolution, the processes used, and the effects of software evolution. The

explanatory approach studies evolution from a *scientific* view point. In this approach, the *nature* of the evolution phenomenon is studied, and one strives to understand its driving factors and impacts.

- *Process improvement (how).* This approach attempts to manage the effects of software evolution by developing better methods and tools, namely, design, maintenance, refactoring, and reengineering. The process improvement approach studies evolution from an *engineering* view point. It focuses on the more pragmatic aspects that assist the developers in their daily routines. Therefore, methods, tools, and activities that provide the means to direct, implement, and control software evolution are at the core of the process improvement approach.

2.3.1 SPE Taxonomy

The abbreviation SPE refers to S (Specified), P (Problem), and E (Evolving) programs. In circa 1980, Meir M. Lehman [24] proposed an SPE classification scheme to explain the ways in which programs vary in their evolutionary characteristics. The classification scheme is characterized by: (i) how a program interacts with its environment and (ii) the degree to which the environment and the underlying problem that the program addresses can change. He observed a key difference between software developed to meet a fixed set of requirements and software developed to solve a real-world problem which changes with time. The observation leads to the identification of types S (Specified), P (Problem), and E (Evolving) programs. In the following, we explain the SPE concepts in detail.

S-type programs: S-type programs have the following characteristics:

- All the nonfunctional and functional program properties that are important to its stakeholders are *formally* and *completely* defined.
- Correctness of the program with respect to its formal specification is the *only* criterion of the acceptability of the solution to its stakeholders.

A formal definition of the problem is viewed as the specification of the program. S-type programs solve problems that are fully defined in abstract and closed ways. Examples of S-type programs include calculation of the lowest common multiple of two integers and to perform matrix addition, multiplication, and inversion [28]. The problem is completely defined, and there are one or more correct solutions to the problem as stated. The solution is well known, so the developer is concerned not with the correctness of the solution but with the correctness of the implementation of the solution. As illustrated in Figure 2.5, the specification directs and controls the programmers in creating the program that defines the desired solution. The problem statement, the program, and the solution may relate to a real world—and the real world can change. However, if the real world changes, the original problem turns into a completely *new problem* that must be respecified. But then it has a *new program* to provide a solution. It may be possible and time saving to derive a new program from

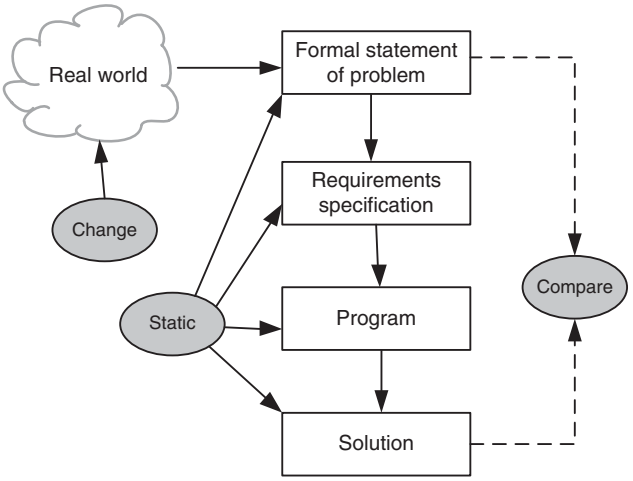

FIGURE 2.5 S-type programs

the old one, but it is a *different* program that defines a solution to a *different* problem. The program remains almost the same in the sense that it does not accommodate changes in the problem that generates it [29]. In the real world, S-type systems are rare. However, it is an important concept that evolution of software does not occur under some conditions.

P-type programs: With many real problems, the system outputs are accurate to a constrained level of precision. The concept of correctness is difficult to define in those programs. Therefore, approximate solutions are developed for pragmatic reasons. Numerical problems, except computations with integers and rational numbers, are resolved through approximations. For example, consider a program to play chess. Since the rules of chess are completely defined, the problem can be completely specified. At each step of the game a solution might involve calculating the various moves and their impacts to determine the next best move. However, complete implementation of such a solution may not be possible, because the number of moves is too large to be evaluated in a given time duration. Therefore, one must develop an approximate solution that is more practical while being acceptable.

In order to develop this type of solution, we describe the problem in an abstract way and write the requirement specification accordingly. A program developed this way is of P-type because it is based on a practical abstraction of the problem, instead of relying on a completely defined specification. Even though an exact solution may exist, the solution produced by a P-type program is tampered by the environment in which it must be produced. The solution of a P-type program is accepted if the program outcomes make sense to the stakeholder(s) in the world in which the problem is embedded. As illustrated in Figure 2.6, P-type programs are more dynamic than S-type programs. P-type programs are likely to change in an incremental fashion. If the output of the solution is unacceptable, then the problem abstraction may

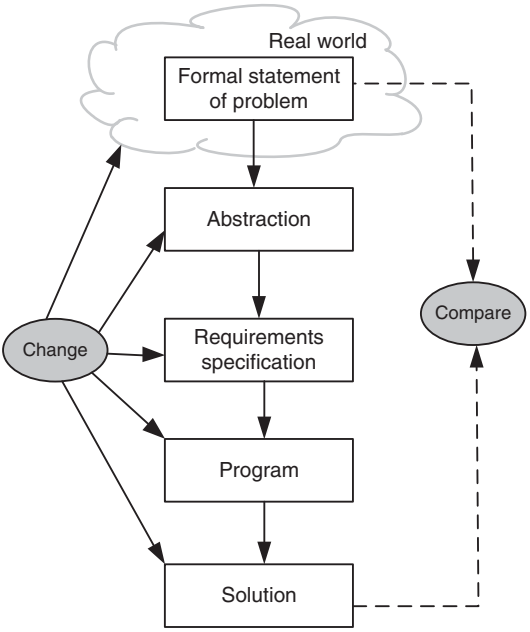

FIGURE 2.6 P-type programs

be changed and the requirements modified to make the new solution more realistic. Note that the program resulting from the changes cannot be considered a new solution to a new problem. Rather, it is a modification of the old solution to better fit the existing problem. In addition, the real world may change, hence the problem changes.

E-type programs: An E-type program is one that is embedded in the real world and it changes as the world does. These programs mechanize a human or society activity, make simplifying assumptions, and interface with the external world by requiring or providing services. An E-type system is to be regularly adapted to: (i) stay true to its domain of application; (ii) remain compatible with its executing environment; and (iii) meet the goals and expectations of its stakeholders [30].

Figure 2.7 illustrates the dependence of an E-type program on its environment and the consequent changeability. The acceptance of an E-type program entirely depends upon the stakeholders' opinion and judgment of the solution. Their descriptions cannot be completely formalized to permit the demonstration of correctness, and their operational domains are potentially unbounded. The first characteristic of an E-type program is that the outcome of executing the program is not definitely predictable. Therefore, for E-type programs, the concept of correctness is left up to the stakeholders. That is, the criterion of acceptability is the stakeholders' satisfaction with each program execution [31]. An E-type program's second characteristic

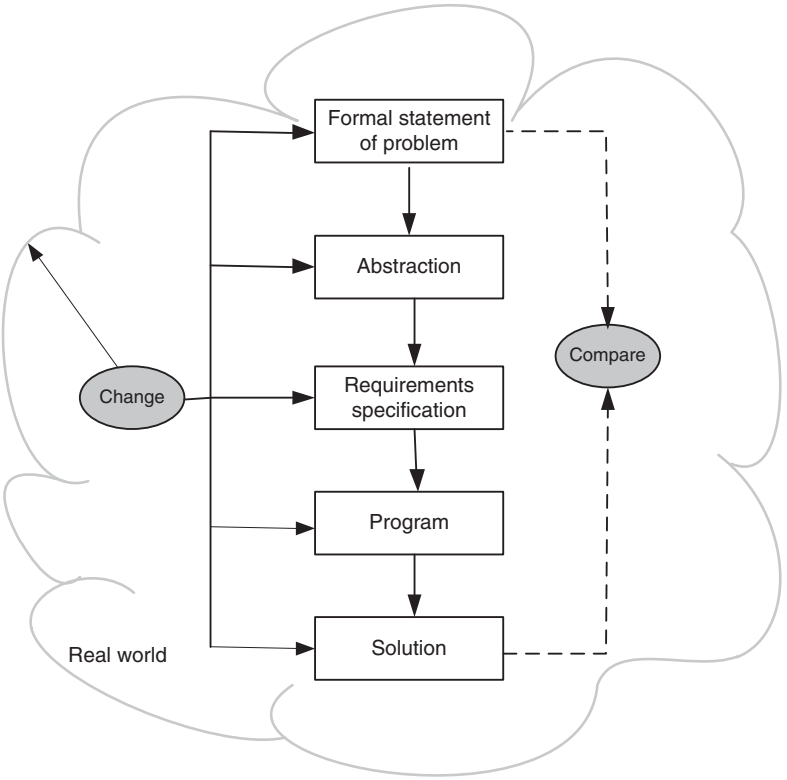

FIGURE 2.7 E-type programs

is that program execution changes its operational domain, and the evolution process is viewed as a feedback system [32]. Figure 2.8 [33] succinctly illustrates the feedback process.

2.3.2 Laws of Software Evolution

Lehman and his colleagues have postulated eight "laws" over 20 years starting from the mid-1970s to explain some key observations about the evolution of E-type software systems [34, 35]. The laws themselves have evolved from three in 1974 to eight by 1997, as listed in Table 2.5. The eight laws are the results of empirical studies of the evolution of large-scale proprietary software—also called closed source software (CSS)—in a variety of corporate settings. The laws primarily relate to perfective maintenance. These laws are largely based on the concept of feedback existing in the software environment. Their description of the phenomena are intertwined, and the laws are not to be studied separately. The numbering of the laws has no significance apart from the sequence of their development.

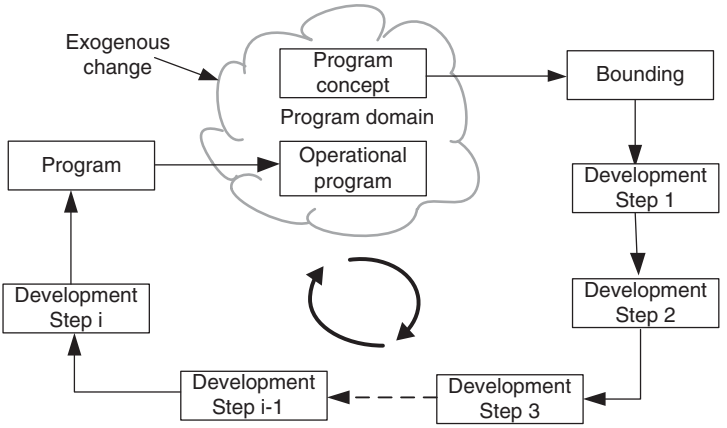

FIGURE 2.8 E-type programs with feedback. From Reference 33. © 2006 John Wiley & Sons

TABLE 2.5 Laws of Software Evolution

Names of the Laws	Brief Descriptions
I. Continuing change (1974)	E-type programs must be continually adapted, else they become progressively less satisfactory.
II. Increasing complexity (1974)	As an E-type program evolves, its complexity increases unless work is done to maintain or reduce it.
III. Self-regulation (1974)	The evolution process of E-type programs is self-regulating, with the time distribution of measures of processes and products being close to normal.
IV. Conservation of organizational stability (1978)	The average effective global activity rate in an evolving E-type program is invariant over the product's lifetime.
V. Conservation of familiarity (1978)	The average content of successive releases is constant during the life cycle of an evolving E-type program.
VI. Continuing growth (1991)	To maintain user satisfaction with the program over its lifetime, the functional content of an E-type program must be continually increased.
VII. Declining quality (1996)	An E-type program is perceived by its stakeholders to have declining quality if it is not maintained and adapted to its environment.
VIII. Feedback system (1971–1996)	The evolution processes in E-type programs constitute multi-agent, multi-level, multi-loop feedback systems.

Source: Adapted from Lehman et al. [34]. ©1997 IEEE.

Lehman's laws were not meant to be used in a mathematical sense, as, say, Newton's laws are used in physics. Rather, those were intended to capture stable, long-term knowledge about the common features of changing software systems, in the same sense social scientists use laws to characterize general principles applying to some classes of social situations [30]. The term "laws" was used because the observed phenomena were beyond the influence of managers and developers. The laws were an attempt to study the nature of software evolutions and the evolutionary trajectory likely taken by software.

First Law *Continuing change: E-type programs must be continually adapted, else they become progressively less satisfactory.* Many assumptions are embedded in an E-type program. A subset of those assumptions may be *complete* and *valid* at the initial release of the product; that is, the program performed satisfactorily even if not all assumptions were satisfied. As users continue to use a system over time, they gain more experience, and their needs and expectations grow. As the application's environment changes in terms of the number of sophisticated users, a growing number of assumptions become *invalid*. Consequently, new requirements and new CRs will emerge. In addition, changes in the real world will occur and the application will be impacted, requiring changes to be made to the program to restore it to an acceptable model. When the updated and modified program is reintroduced into the operational domain, it continues to satisfy user needs for a while; next, more changes occur in the operation environment, additional user needs are identified, and additional CRs are made. As a result, the evolution process moves into a vicious cycle.

Second Law *Increasing complexity: As an E-type program evolves, its complexity increases unless work is done to maintain or reduce it.* As the program evolves, its complexity grows because of the imposition of changes after changes on the program. In order to incorporate new changes, more objects, modules, and sub-systems are added to the system. As a consequence, there is much increase in: (i) the effort expended to ensure an adequate and correct interface between the old and new elements; (ii) the number of errors and omissions; and (iii) the possibility of inconsistency in their assumptions. Such increases lead to a decline in the product quality and in the evolution rate, unless additional work is performed to arrest the decline. The only way to avoid this from happening is to invest in preventive maintenance, where one spends time to improve the structure of the software without adding to its functionality.

Third Law *Self-regulation: The evolution process of E-type programs is self-regulating, with the time distribution of measures of processes and products being close to normal.* This law states that large programs have a dynamics of their own; attributes such as size, time between releases, and the number of reported faults are approximately invariant from release to release because of fundamental structural and organizational factors. In an industrial setup E-type programs are designed and coded by a team of experts working in a larger context comprising a variety of management entities, namely, finance, business, human resource, sales, marketing, support, and user process. The various groups within the large organization apply constraining information controls and reinforcing information controls influenced

by past and present performance indicators. Their actions control, check, and balance the resource usage, which is a kind of feedback-driven growth and stabilization mechanism. This establishes a self-controlled dynamic system whose process and product parameters are normally distributed as a result of a huge number of largely independent implementation and managerial decisions [36].

Fourth Law *Conservation of organizational stability: The average effective global activity rate in an evolving E-type program is invariant over the product's lifetime.* This law suggests that most large software projects work in a "stationary" state, which means that changes in resources or staffing have small effects on long-term evolution of the software. To a certain extent management certainly do control resource allocation and planning of activities. However, as suggested by the third law, program evolution is essentially independent of management decisions. In some instances, as indicated by Brooks [37], situations may arise where additional resources may reduce the effective rate of productivity output due to higher communication overhead or decrease in process quality. In reality, activities during the life cycle of a system are not exclusively decided by management but by a wide spectrum of controls and feedback inputs [38].

Fifth Law *Conservation of familiarity: The average content of successive releases is constant during the life cycle of an evolving E-type program.* As an E-type system evolves, both developers and users must try to develop mastery of its content and behavior. Thus, after every major release, established familiarity with the application and the system in general is counterbalanced by a decline in the detail knowledge and mastery of the system. This would be expected to produce a temporary slow down in the growth rate of the system as it is recognized that the system must be cleaned up to simplify the process of re-familiarization. In practice, adding new features to a program invariably introduces new program faults due to unfamiliarity with the new functionality and the new operational environment. The more changes are made in a new release, the more faults will be introduced. The law suggests that one should not include a large number of features in a new release without taking into account the need for fixing the newly introduced faults. Conservation of familiarity implies that maintenance engineers need to have the same high level of understanding of a new release even if more functionalities have been added to it.

Sixth Law *Continuing growth: To maintain user satisfaction with the program over its lifetime, the functional content of an E-type program must be continually increased.* It is useful to note that programs exhibit finite behaviors, which implies that they have limited properties relative to the potential of the application domain. Properties excluded by the limitedness of the programs eventually become a source of performance constraints, errors, and irritation. To eliminate all those negative attributes, it is needed to make the system grow.

It is important to distinguish this law from the first law which focuses on "Continuing Change." The first law captures the fact that an E-type software's operational domain undergoes continual changes. Those changes are partly driven by installation and operation of the system and partly by other forces; an example of other forces is human desire for improvement and perfection. These two laws—the first and

the sixth—reflect distinct phenomena and different mechanisms. When phenomena are observed, it is often difficult to determine which of the two laws underlies the observation.

Seventh Law *Declining quality: An E-type program is perceived by its stakeholders to have declining quality if it is not maintained and adapted to its environment.* This law directly follows from the first and the sixth laws. An E-Type program must undergo changes in the forms of adaptations and extensions to remain satisfactory in a changing operational domain. Those changes are very likely to degrade the performance and will potentially inject more faults into the evolving program. In addition, the complexity (e.g., the cyclomatic measure) of the program in terms of interactions between its components increases, and the program structure deteriorates. The term for this increase in complexity over time is called *entropy*. The average rate at which software entropy increases is about 1–3 per calendar year [17]. There is significant decline in stakeholder satisfaction because of growing entropy, declining performance, increasing number of faults, and mismatch of operational domains. The aforementioned factors also cause a decline in software quality from the user's perspective [39]. The decline of software quality over time is related to the growth in entropy associated with software product aging [18] or code decay [8]. Therefore, it is important to continually undertake preventive measures to reduce the entropy by improving the software's overall architecture, high-level and low-level design, and coding.

Eighth Law *Feedback system: The evolution processes in E-type programs constitute multi-agent, multi-level, multi-loop feedback systems.* Several laws of software revolution refer to the role of information feedback in the life cycles of software. The eighth law is based on the observation that evolution process of the E-type software constitutes a multi-level, multi-loop, multi-agent feedback system: (i) multi-loop means that it is an iterative process; (ii) multi-level refers to the fact that it occurs in more than one aspect of the software and its documentation; and (iii) a multi-agent software system is a computational system where software agents cooperate and compete to achieve some individual or collective tasks. Feedback will determine and constrain the manner in which the software agents communicate among themselves to change their behavior [40].

Remark: There are two types of aging in software life cycles: software process execution aging and software product aging. The first one manifests in degradation in performance or transient failures in continuously running the software system. The second one manifests in degradation of quality of software code and documentation due to frequent changes. The following aging-related *symptoms* in software were identified by Visaggio [41]:

- *Pollution.* Pollution means that there are many modules or components in a system which are not used in the delivery of the business functions of the system.
- *Embedded knowledge.* Embedded knowledge is the knowledge about the application domain that has been spread throughout the program such that the knowledge cannot be precisely gathered from the documentation.

- *Poor lexicon.* Poor lexicon means that the component identifiers have little lexical meaning or are incompatible with the commonly understood meaning of the components that they identify.
- *Coupling.* Coupling means that the programs and their components are linked by an elaborate network of control flows and data flows.

Remark: The code is said to have decayed if it is very difficult to change it, as reflected by the following three key responses: (i) the cost of the change, which is effective only on the personnel cost for the developers who implement it; (ii) the calendar or clock time to make the changes; and (iii) the quality of the changed software. It is important to note that code decay is antithesis of evolution in the sense that while the evolution process is intended to make the code better, changes are generally degenerative thereby leading to code decay.

2.3.3 Empirical Studies

Empirical studies are aimed at acquiring knowledge about the effectiveness of processes, methods, techniques, and tools used in software development and maintenance. Similarly, the laws of software evolution are prime candidates for empirical studies, because we want to know to what extent they hold. In circa 1976, Belady and Lehman [22] studied 20 releases of the OS/360 operating system. The results of their study led them to postulate five laws of software evolution: continuing change, increasing complexity, self-regulation, conservation of organizational stability, and conservation of familiarity. Those laws were further developed in an article published in 1980 [24]. Yuen [36, 42] further studied their five laws of evolution. He re-examined three different systems from Belady and Lehman [22] and several other systems and examined a variety of dependent variables. The number and percentage of modules handled are examples of dependent variables. After re-examining the data from previous studies, he observed that the characteristics observed for OS/360 did not necessarily hold for other systems. Specifically, the first two laws were supported, while the remaining three laws were not. Yuen, a collaborator of Lehman, notes that these three laws are more based upon those of human organizations involved in the maintenance process rather than the properties of the software itself.

Later, in a project entitled FEAST (Feedback, Evolution, And Software Technology), Lehman and his colleagues studied evolution of releases from four CSS systems: (i) two operating systems (OS/360 of IBM and VME OS of ICL); (ii) one financial system (Logica's FW banking transaction system); and (iii) a real-time telecommunication system (Lucent Technologies). Their results are summarized as a set of growth curves, as described by Lehman, Perry, and Ramil [43]. The studies suggest that during the maintenance process a system tracks a growth curve that can be approximated either as linear or inverse square [44]. The inverse square model represents the growth phenomena as an inverse square of the continuing effort. Those trends increase the confidence of validity of the following six laws: Continuing change (I), Increasing complexity (II), Self-regulation (III), Conservation of familiarity (V),