1

BASIC CONCEPTS AND PRELIMINARIES

Another flaw in the human character is that everybody wants to build and nobody wants to do maintenance.

—Kurt Vonnegut, Jr.

1.1 EVOLUTION VERSUS MAINTENANCE

In 1965, Mark Halpern introduced the concept of *software evolution* to describe the growth characteristics of software [1]. Later, the term "evolution" in the context of application software was widely used. The concept further attracted the attentions of researchers after Belady and Lehman published a set of principles determining evolution of software systems [2, 3]. The principles were very general in nature. In his landmark article entitled "The Maintenance 'Iceberg'," R. G. Canning compared software maintenance to an "iceberg" to emphasize the fact that software developers and maintenance personnel face a large number of problems [4]. A few years later, in 1976, Swanson introduced the term "maintenance" by grouping the maintenance activities into three basic categories: corrective, adaptive, and perfective [5]. In the early 1970s, IBM called them "maintenance engineers" or "maintainers" who had been making intentional modifications to running code that they had not developed themselves. The main reason for using nondevelopment personnel in maintenance work was to free up the software development engineers or programmers from support

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

activities [6]. In this book, we will use maintainer, maintenance engineer, developer, and programmer interchangeably.

Bennett and Rajlich [7] researched the term "software evolution" and found that there is no widely accepted definition of the term. In addition, some researchers and practitioners used the phrases "software evolution" and "software maintenance" interchangeably. However, key semantic differences exist between the two. The two are distinguished as follows:

- The concept of *software maintenance* means preventing software from failing to deliver the intended functionalities by means of bug fixing.
- The concept of *software evolution* means a continual change from a lesser, simpler, or worse state to a higher or better state ([8], p. 1).

Bennett and Xu [9] made further distinctions between the two as follows:

- All support activities carried out *after* delivery of software are put under the category of *maintenance*.
- All activities carried out to effect changes in requirements are put under the category of *evolution*.

In general, maintenance and evolution are generally differentiated as follows [10]:

- Maintenance of software systems primarily means fixing bugs but preserving their functionalities. Maintenance tasks are very much planned. For example, bug fixing must be done and it is a planned activity. In addition to the planned activities, unplanned activities are also necessitated. For example, a new usage of the system may emerge. Generally, maintenance does not involve making major changes to the architecture of the system. In other words, maintenance means keeping an installed system running with no change to its design [11].
- Evolution of software systems means creating new but related designs from existing ones. The objectives include supporting new functionalities, making the system perform better, and making the system run on a different operating system. Basically, as time passes, the stakeholders develop more knowledge about the system. Therefore, the system evolves in several ways. As time passes, not only new usages emerge, but also the users become more knowledgeable. As Mehdi Jazayeri observed: "Over time what evolves is not the software but our knowledge about a particular type of software" ([12], p. 3).

While we are on the topic of maintenance, it is useful to glance at the maintenance of physical systems. Maintenance of physical systems often requires replacing broken and worn-out parts. For example, owners replace the worn-out tires and broken lamps of their cars. Similarly, a malfunctioning memory card is replaced with a good one. On the other hand, software maintenance is different than hardware maintenance. In hardware maintenance, a system or a component is returned to its original good state. On the other hand, in software maintenance, a software system is moved from its original erroneous state to an expected good state [13]. Software maintenance comprises all activities associated with the process of changing software for the purposes of:

- fixing bugs; and/or
- improving the design of the system so that future changes to the system are less expensive.

1.1.1 Software Evolution

Although the phrase "software evolution" had been used previously by other researchers, fundamental work in the field of software evolution was done by Lehman and his collaborators. Based on empirical studies [2, 14], Lehman and his collaborators formulated some observations and they introduced them as *laws of evolution*. The "laws" themselves have "evolved" from *three* in 1974 to *eight* by 1997 [15, 16]. Those laws are the results of studies of the evolution of large-scale proprietary or closed source software (CSS) systems. The laws concern a category of software systems called *E-type* systems. The eight laws are briefly explained as follows:

1. *Continuing change*. Unless a system is continually modified to satisfy emerging needs of users, the system becomes increasingly less useful.
2. *Increasing complexity*. Unless additional work is done to explicitly reduce the complexity of a system, the system will become increasingly more complex due to maintenance-related changes.
3. *Self-regulation*. The evolution process is self-regulating in the sense that the measures of products and processes, that are produced during the evolution, follow close to *normal* distributions.
4. *Conservation of organizational stability*. The average effective global activity rate on an evolving system is almost constant throughout the lifetime of the system. In other words, the average amount of additional effort needed to produce a new release is almost the same.
5. *Conservation of familiarity*. As a system evolves all kinds of personnel, namely, developers and users, for example, must gain a desired level of understanding of the system's content and behavior to realize satisfactory evolution. A large incremental growth in a release reduces that understanding. Therefore, the average incremental growth in an evolving system remains almost the same.
6. *Continuing growth*. As time passes, the functional content of a system is continually increased to satisfy user needs.
7. *Declining quality*. Unless the design of a system is diligently fine-tuned and adapted to new operational environments, the system's qualities will be perceived as declining over the lifetime of the system.
8. *Feedback system*. The system's evolution process involves multi-loop, multi-agent, multi-level feedback among different kinds of activities. Developers must recognize those complex interactions in order to continually evolve an existing system to deliver more functionalities and higher levels of qualities.

In circa 1988, Pirzada [17] was the first one to study the differences between the evolution of the Unix operating system developed by Bell Laboratories and the systems studied by Lehman and Belady [18]. Pirzada argued that the differences in academic and industrial software development could lead to differences in the evolutionary pattern. In circa 2000, after a gap of 12 years, empirical study of evolution of free and open source software (FOSS) was conducted by Godfrey and Tu [19]. The authors provided the trend of growth of the popular FOSS operating system Linux during 1994–1999. They showed the growth rate to be super-linear that is greater than linear. Robles et al. [20] later replicated the study of Godfrey and Tu and concluded that Lehman's laws Nos. 3, 4, and 5 do not hold for large-scale FOSS systems such as Linux. These studies reveal the changing nature of both software and software development processes. Lehman's studies mostly examined proprietary, monolithic systems developed by a team of developers within a company, whereas FOSS systems and their developments follow a different evolution paradigm.

Remark: FOSS is available to all with relaxed or nonexistent copyrights. FOSS is commonly used as a synonym for free software even though "free" and "open" have different semantics. The term "free" means the freedom to modify and redistribute the system under the terms of the original agreement, while "open" means accessibility to the source code.

1.1.2 Software Maintenance

More likely than not, there are defects in delivered software applications, because defect removal and quality control processes are not perfect. Therefore, maintenance is needed to repair those defects in released software. E. Burton Swanson [5] initially defined three categories of software maintenance activities, namely, *corrective*, *adaptive*, and *perfective*. Those definitions were later incorporated into the standard software engineering–software life cycle processes–Maintenance [21] and introduced a fourth category called *preventive* maintenance. The reader may note that some researchers and developers view preventive maintenance as a subset of perfective maintenance.

Swanson's classification of maintenance activities is intention based because the maintenance activities reflect the intents of the developer to carry out specific maintenance tasks on the system. In the intention-based classification of maintenance activities, the intention of an activity depends upon the motivations for the change. An alternative way of classifying modifications to software is to simply categorize the modifications in terms of activities performed [22]:

- *Activities to make corrections.* If there are discrepancies between the expected behavior of a system and the actual behavior, then some activities are performed to eliminate or reduce the discrepancies.
- *Activities to make enhancements.* A number of activities are performed to implement a change to the system, thereby changing the behavior or implementation

of the system. This category of activities is further refined into three subcategories:

– enhancements that modify existing requirements;
– enhancements that create new requirements; and
– enhancements that modify the implementation without changing the requirements.

Chapin et al. [6] expanded the typology of Swanson into an evidence-based classification of 12 different types of software maintenance: training, consultive, evaluative, reformative, updative, groomative, preventive, performance, adaptive, reductive, corrective, and enhancive. The three objectives for classifying the types of software maintenance are as follows:

- It is more informative to classify maintenance tasks based on objective evidence that can be verified with observations and/or comparisons of software before and after modifications. This does not require accessing the knowledge of the personnel who originally developed the system.
- The granularity of the proposed classification can be made to accurately reflect the actual mix of activities observed in the practice of software maintenance and evolution.
- The classification groups are independent of hardware platform, operating system choice, design methodology, implementation language, organizational practices, and the availability of the personnel doing the original development.

Maintenance of COTS-Based Systems Many present-day software systems are built from components previously developed for other systems or to be reused in many systems. In this approach, new components are developed by combining commercial off-the-shelf (COTS) components, custom-built (in-house) components, and open source software components. The components are obtained from a variety of sources and maintained by different vendors, possibly from different countries [23]. The motivations for performing software maintenance are the same for both component-based software systems (CBS) and custom-built software systems. However, there are noticeable differences between the activities in the two approaches. The major sources of the differences are as follows [24, 25]:

- *Skills of system maintenance teams.* Maintenance of CBS requires specialized skills to monitor and integrate COTS products. Those skills are different than the skills required to perform the more traditional maintenance functions: analyze and modify source code developed in-house. Maintainers view a CBS as a group of black-box components, and not as a compiled set of source code modules, thereby requiring a different set of maintenance skills. The differences in skills are neither pros nor cons, but it is important that the differences are taken into consideration for planning, staffing, and training.

- *Infrastructure and organization.* Running a support group for in-house products is necessary to manage a large product. This additional cost may be shared with other projects.
- *COTS maintenance cost.* This cost includes the costs of purchasing components, licensing components, upgrading components, and training maintenance personnel. From the perspective of a system's life cycle, much cost is shifted from in-house development to license and maintenance fees, thereby increasing the overall maintenance cost.
- *Larger user community.* COTS users are part of a broad community of users, and the community of users can be considered as a resource, which is a positive factor. However, being part of a community means having less control over changes and improvements to COTS products.
- *Modernization.* In general, vendors of COTS components keep pace with changing technology and continually update the components. As a result, the system does not become obsolete. However, the flip side is that the costs and risks of making changes keep increasing even if the application does not require any changes. In general, control over the evolution and maintenance of significant portions of the system is relinquished to third-party COTS developers. Those third-party developers may be motivated to pursue their own commercial self-interest. In addition, the third-party vendors control not only the nature of maintenance to be done on the products, but also when it is to be done. Therefore reliance on third-party products impacts both the type and timing of the maintenance performed by COTS-based developers. In a nutshell, unfortunately, upgrades to products are necessitated by technology and vendor economics.
- *Split maintenance function.* A COTS product is maintained by its vendor, whereas the overall system that uses the COTS product is maintained by the system's host organization. As a result, multiple, independent maintenance teams exist. The advantage of COTS-based development is that the system maintainers receive additional support from the COTS vendors. On the other hand, the drawback of the approach is that the different COTS pieces need tighter coordination, and the product vendors may stray in all directions with respect to functionality and standard.
- *More complex planning.* If a system depends upon multiple technologies and COTS products, the unpredictability and risk of change become high, and planning becomes complicated because coordination among a large number of vendors is more difficult.

1.2 SOFTWARE EVOLUTION MODELS AND PROCESSES

There is much confusion about the terms "software maintenance" and "software evolution." The confusion is partly due to a lack of attention paid to models for sustaining software systems and partly due to considering maintenance to be another activity in software development. For example, consider the classical Waterfall model for software development proposed by Winston Royce in circa 1970 [26]. The final