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

phase of the Waterfall model is known as maintenance, which implies that software maintenance is a part of software development. In this regard it is worth quoting Norman Schneidewind [27]: "The traditional view of the software life cycle has done a disservice to maintenance by depicting it solely as a single step at the end of the cycle" (p. 304). Therefore, software maintenance should have its own software maintenance life cycle (SMLC) model [28]. A number of SMLC models with some variations are available in literature [8, 29–35]. Three common features of the SMLC models found in the literature are:

- understanding the code;
- modifying the code; and
- revalidating the code.

Other models view software development as *iterative* processes and based on the idea of *change mini-cycle* [7, 36–39] as explained in the following:

- *Iterative models.* The iterative models share the ideas that a complete set of requirements for a system cannot be completely understood, or the developers do not know how to build the full system. Therefore, systems are constructed in builds, each of which is a refinement of requirements of the previous build. A build is refined by considering feedback from users [40]. One may note that maintenance and evolution activities do not exist as distinct phases. Rather, they are closely intertwined.
- *Change mini-cycle models.* First proposed by Yau et al. [36] in the late 1970s, these models were recently re-visited by Bennet et al. [7] and Mens [41] among others. These models consist of five major phases: change request, analyze and plan change, implement change, verify and validate, and documentation change. In this process model, several important activities were identified, such as program comprehension, impact analysis, refactoring, and change propagation.

A different kind of software evolution model, called *staged model of maintenance and evolution*, has been proposed by Rajlich and Bennett [42]. The model is descriptive in nature, and its primary objective is to improve the understanding of how long-lived software evolves. The model considers four distinct, sequential stages of the lifetime of a system, as explained below:

1. *Initial development.* When the initial version of the system is produced, detailed knowledge about the system is fresh. Before delivery of the system, it undergoes many changes. Eventually, a system architecture emerges and soon it stabilizes.
2. *Evolution.* After the initial stability, it is easy to perform simple changes to the system. Significant changes involve higher cost and higher risk. In the period immediately following the initial delivery, knowledge about the system is still almost fresh in the minds of the developers. It is possible that the development team as a whole does not exist, because many original developers have taken up new responsibilities in the organization and some might have left

the organization. In general, for many systems, their lifespan are spent in this stage, because the systems continue to be of importance to the organizations.

3. *Servicing.* When the knowledge about the system has significantly decreased, the developers mainly focus on maintenance tasks, such as fixing bugs, whereas architectural changes are rarely effected. The developers do not consider the system to be a key asset. In this stage, the effects of changes are very difficult to predict. Moreover, the costs and risks of making changes are very significant.
4. *Phaseout.* When even minimal servicing of a system is not an option, the system enters its very final stage. The organization decides to replace the system for various reasons: (i) it is too expensive to maintain the system; or (ii) there is a newer solution available. Therefore, the organization develops an exit strategy to move from the current system to a new system. Moving from an existing, difficult-to-maintain system to a modern solution system has its own challenges involving wrapping and data migration. After the new system keeps running satisfactorily, sometimes in parallel with the old system, the old system is finally completely shut down.

Software Maintenance Standards A well-defined process for software maintenance can be observed and measured, and thus improved. In addition, adoption of processes allows the dissemination of effective work practices more quickly than gaining personal experience. Process centric software maintenance is more of an engineering activity, with predictable time and effort constraints, and less of an art. Therefore, software maintenance standards have been formulated by ISO and IEEE. The maintenance standard document from ISO is called ISO/IEC 14764 [21] which is a part of the standard document ISO/IEC 12207 [43] for life cycle processes. The maintenance standard document from IEEE is called IEEE/EIA 1219 [44].

Both the standards describe processes for managing and executing activities for maintenance. The IEEE/EIA 1219 standard organizes the maintenance process in seven phases: problem identification, analysis, design, implementation, system test, acceptance test, and delivery. As a quick summary, the standard identifies the different phases and the sequence of their executions. Next, for each phase, the standard identifies the input and output deliverables, the supporting processes and the related activities, and a set of evaluation metrics. Both the standards, namely ISO/IEC 14764 and IEEE/EIA 1219, use the same terminology to describe software maintenance, with a little difference in their depictions. An iterative process has been described in ISO/IEC 14764 to manage and execute maintenance activities. The activities comprising the maintenance process are:

- process implementation;
- problem and modification analysis;
- modification implementation;
- maintenance review/acceptance;
- migration; and
- retirement.

Each of the aforementioned activities is made up of tasks described with specific inputs, outputs, and actions.

Software Configuration Management Configuration management (CM) is the discipline of managing changes in large systems. The goal of CM is to manage and control the various extensions, adaptations, and corrections that are applied to a system over its lifetime. It handles the control of all products/configuration items and changes to those items. Software configuration management (SCM) is the configuration management applied to software systems. SCM is the means by which the process of software evolution is managed. SCM has been defined in the IEEE 1042 standard [45] as "software configuration management (SCM) is the discipline of managing and controlling change in the evolution of software systems." SCM provides a framework for managing changes in a controlled manner. The purpose of SCM is to reduce communication errors among personnel working on different aspects of the software project by providing a central repository of information about the project and a set of agreed upon procedures for coping with changes. It ensures that the released software is not contaminated by uncontrolled or unapproved changes. Early SCM tools had limited capabilities in terms of functionality and applicability. However, modern SCM systems provide advanced capabilities through which many different artifacts are managed. For example, modern SCM systems support their users in building an executable program out of its versioned source files. Moreover, it must be possible to regenerate old versions of the software system. In general, an SCM system has four different elements, each element addressing a distinct user need as follows [46, 47]:

- *Identification of software configurations.* This includes the definitions of the different artifacts, their baselines or milestones, and the changes to the artifacts.
- *Control of software configurations.* This element is about controlling the ways artifacts or configurations are altered with the necessary technical and administrative support.
- *Auditing software configurations.* This element is about making the current status of the software system in its life cycle visible to management and determining whether or not the baselines meet their requirements.
- *Accounting software configuration status.* This element is about providing an administrative history of how the software system has been altered, by recording the activities necessitated by the other three SCM elements.

1.3 REENGINEERING

Hongji Yang and Martin Ward [48] defined software evolution as "… the process of conducting continuous software reengineering" (p. 23). Reengineering implies a single cycle of taking an existing system and generating from it a new system, whereas evolution can go forever. In other words, to a large extent, software evolution can

be seen as repeated software reengineering. Reengineering is done to transform an existing "lesser or simpler" system into a new "better" system. Chikofsky and Cross II [49] define reengineering as "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

Therefore, reengineering includes some kind of reverse engineering activities to design an abstract view of a given system. The new abstract view is restructured, and forward engineering activities are performed to implement the system in its new form. The aforementioned process is captured by Jacobson and Lindstörm [50] with the following expression:

$$\text{Reengineering} = \text{Reverse engineering} + \Delta + \text{Forward engineering}.$$

Let us analyze the right-hand portion of the above equation. The first element "reverse engineering" is the activity of defining a more abstract and easier to understand representation of the system. For example, the input to the reverse engineering process is the source code of the system, and the output is the system architecture. The core of reverse engineering is the process of examination of the system, and it is not a process of change. Therefore it does not involve changing the software under examination. The third element "forward engineering" is the traditional process of moving from a high-level abstraction and logical, implementation-independent design to the physical implementation of the system. The second element "Δ" captures alterations performed to the original system.

While performing reverse engineering on a large system, tools and methodologies are generally not stable. Therefore, a high-level organizational paradigm enables repetitions of processes so that maintenance engineers learn about the system. Benedusi et al. [51] have proposed a repeatable paradigm, called Goals/Models/Tools, that describes reverse engineering in three successive stages, namely, Goals, Models, and Tools.

Goals. In this phase, one analyzes the motivations for setting up the process to identify the information needs of the process and the abstractions to be produced.

Models. In this phase, one analyzes the abstractions to construct representation models that capture the information needed for their production.

Tools. In this phase, software tools are defined, acquired, enhanced, integrated, or constructed to: (i) execute the Models phase and (ii) transform the program models into the abstractions identified in the Goals phase.

It is important to note that fact-finding and information gathering from the source code are keys to the Goal/Models/Tools paradigm. In order to extract information that is not explicitly available in source code, automated analysis techniques, such as *lexical analysis, syntactic analysis, control flow analysis, data flow analysis,* and *program slicing* are used to facilitate reverse engineering.

The increased use of data mining techniques in support systems have given rise to an interest in data reverse engineering (DRE) technology. DRE tackles the question of what information is stored and how this information can be used in a different context. DRE is defined by Peter Aiken as "the use of structured techniques to reconstitute the data assets of an existing system" [52]. The two vital aspects of the DRE process are to: (i) recover *data assets* that are useful or valuable and (ii) *reconstitute* the recovered data assets to make them more useful. Therefore, DRE can be regarded as adding value to the existing data assets, making it easier for organizations to conduct business efficiently and effectively.

1.4 LEGACY SYSTEMS

A legacy software system is an old program that continues to be used because it still meets the users' needs, in spite of the availability of newer technology or more efficient methods of performing the task. More often than not, a legacy system includes outdated procedures or terminology, and it is very difficult for new developers to understand the system. Organizations continue to use legacy systems because those are vital to them and the systems significantly resist modification and evolution to meet new and constantly changing business requirements [53, 54]. A legacy system falls in the *Phase out* stage of the software evolution model of Rajlich and Bennet described earlier. Organizations in business for a long time generally possess a sizable number of legacy systems. To manage legacy systems, a number of options are available. Some commonly chosen options are as follows [55, 56]:

- *Freeze.* An organization decides to perform no further work on a legacy system. This implies that either the services of the system are no longer needed or a new system completely replaces a legacy system.
- *Outsource.* An organization may decide that supporting legacy software—or for that matter any software—is not its core business. As an alternative, it may outsource the support service to a specialist organization.
- *Carry on maintenance.* In this approach, the organization continues to maintain the system for another period of time, despite all the difficulties in doing so.
- *Discard and redevelop.* In this approach, the application is redeveloped once again from scratch, using new hardware and software platforms, new software architecture and databases, and modern tools. When the new system is available, the legacy system is simply discarded.
- *Wrap.* In this approach, a legacy system is wrapped around with a new software layer, thereby hiding the unwanted complexity of the existing data, individual programs, application systems, and interfaces. The old system performs the actual computations, but users interact with the system in better ways. The notion of "wrapper" was first introduced by Dietrich et al. at IBM in the late 1980s [57]. Wrapping is a black-box reengineering task, because only the legacy interface is analyzed while ignoring the system's internals. A wrapper does

not directly modify the source code, but it indirectly modifies the software functionality of the legacy component. Wrapping lets organizations reuse well-tested components that they trust and leverage their massive investments in the system. As a result, the lifetime of the legacy system is increased. Many researchers have proposed techniques for wrapping legacy systems [58–60].

- *Migrate.* In this approach, an operational legacy system is moved to a new hardware and/or software platform, while still retaining the legacy system's functionality. The idea is to minimize any disruption to the existing business environment.

  Migration is the best alternative if wrapping is unsuitable and redevelopment is not acceptable due to substantial risk. Migration involves changes to the legacy system, including restructuring the system and enhancing the functionality of the system. However, it retains the basic functionality of the existing system without having to completely redevelop it. Migration projects require careful planning for smooth execution. Harry M. Sneed [61] suggested five steps for a good plan: project justification, portfolio analysis, cost estimation, cost-benefit analysis, and contracting. Project justification is the first step in any planning. Justifying the project requires analysis of the existing products, the maintenance process, and the business value of the applications. Portfolio analysis prioritizes applications to be reengineered according to their business value and technical quality. Cost estimation gives us an idea about the cost of the migration project. Cost-benefit analysis tells us the costs of the migration project and the expected returns. Contracting entails the identification of tasks and the distribution of efforts. Given the scale, complexity, and risk of failure of migration projects, a well-defined, easily implementable, detailed approach is essential to their success. Several migration approaches can be found in the literature: *Cold Turkey*, *Database First*, *Database Last*, *Composite Database*, *Chicken Little*, *Butterfly*, and *Iterative* [62–64].

1.5 IMPACT ANALYSIS

Impact analysis is the task of identifying portions of the software that can potentially be affected if a proposed change to the system is effected. The outcome of impact analysis can be used when planning for changes, making changes, and tracking the effects of changes in order to localize the sources of new faults. Impact analysis techniques can be categorized into two classes as follows [65]:

- *Traceability analysis.* In this approach, the high-level artifacts, such as requirements, design, code, and test cases related to the feature to be changed, are identified. A model of associations among artifacts, such that each artifact links to other artifacts, is constructed. This helps in locating the corresponding portions of the design, code, and test cases that need to be maintained.
- *Dependency analysis.* Dependency analysis attempts to assess the effects of a change on the semantic dependencies between program entities. This is achieved

by identifying the syntactic dependencies that may signal the presence of such semantic dependencies [66]. The two dependency-based impact analysis techniques are [67]: call graph-based analysis and dependency graph-based analysis. Dependency analysis is also known as source code analysis.

The following two additional notions are found to be keys to understanding impact analysis:

- *Ripple effect analysis.* Ripple effect analysis emphasizes the tracing repercussions in source code when the code is changed. It measures the impact of a change to a particular module on the rest of the program [68]. Impact can be stated in terms of the problems being created for the rest of the program because of the change. Analysis of ripple effect can provide information regarding what changes are occurring and where they are occurring. Measurement of ripple effect can provide knowledge about the system as a whole through its evolution: (i) the amount of increase or decrease of its complexity since the previous version; (ii) the levels of complexity of individual parts of a system in relation to other parts of the system; and (iii) the effect that a new module has on the complexity of a system as a whole when it is added.
- *Change propagation.* Change propagation activities ensure that a change made in one component is propagated properly throughout the entire system [69–71]. Misunderstanding, lack of experience, and unexpected dependencies are some reasons for failing to propagate changes throughout the development and maintenance cycles of source code. If a change is not propagated correctly, the project risks the introduction of new interface defects [72].

1.6 REFACTORING

Refactoring means performing changes to the structure of software to make it easier to comprehend and cheaper to make subsequent changes without changing the observable behavior of the system. A similar idea for non-object-oriented systems is called restructuring. Refactoring is achieved through removal of duplicate code, simplification of code, and moving code to a different class, among others. Without continual refactoring, the internal structure of software will eventually deform beyond comprehension, due to periodic maintenance. Therefore, regular refactoring helps the system to retain its basic structure [73]. In an agile software methodology, such as eXtreme Programming (XP), refactoring is continuously applied to: (i) make the architecture of the software stable; (ii) render the code readable; and (iii) make the tasks of integrating new functionalities into the system flexible.

An important characteristic of refactoring is that it must preserve the "observable behavior" of the system. Preservation of the observable behavior is verified by ensuring that all the tests passing before refactoring must pass after refactoring. Regression testing is used to ensure that the system did not deviate from the original

system during refactoring. Refactoring does not normally involve code transformation to implement new requirements. Rather, it can be performed without adding new requirements to the existing system. Another aspect of refactoring is to enhance the internal structure of the system. In addition, the concept of program restructuring can be applied to transform legacy code into a more structured form and migrate it to a different programming language. That is, restructuring and refactoring can be used to reengineer software systems.

Refactoring techniques put emphasis on the development of a list of basic refactorings, which can be combined to form complex refactorings [74, 75]. The original list of basic refactorings contained transformations on object-oriented code: (i) add a class, method, or attribute; (ii) rename a class, method, or attribute; (iii) move an attribute or method up or down the hierarchy; (iv) remove a class, method, or attribute; and (v) extract chunks of code into separate methods. Most complex refactoring scenarios require small code changes for the refactorings to work correctly. Primitive refactorings are rarely used in isolation.

1.7 PROGRAM COMPREHENSION

The purpose of program comprehension is to understand an existing software system for planning, designing, coding, and testing changes. T. A. Corbi [76] observed in 1989 that *program comprehension* accounts for 50% of the total effort expended throughout the life cycle of a software system. Therefore, good understanding of the software is key to raising its quality by means of maintenance at a lower cost. In terms of concrete activities, program comprehension involves building mental models of an underlying system at different levels of abstractions, varying from low-level models of the code to very high-level models of the underlying application domain [77]. Mental models have been studied by cognitive scientists to understand how human beings know, perceive, make decisions, and construct behavior in a real world [78, 79]. In the domain of program comprehension, a mental model describes a programmer's mental representation of the program to be comprehended.

Program comprehension involves constructing a mental model of the program by applying various cognitive processes. A key step in developing mental models is generating hypotheses, or conjectures, and investigating their validity. Hypotheses are a way for a programmer to understand code in an incremental manner. After some understanding of the code, the programmer forms a hypothesis and verifies it by reading code. Verification of hypothesis results in either accepting the hypothesis or rejecting it. Sometimes, a hypothesis may not be completely correct because of incomplete understanding of code by the programmer. By continuously formulating new hypotheses and verifying them, the programmer understands more and more code and in increasing details.

One can apply several strategies to arrive at meaningful hypotheses, such as bottom–up, top–down, and opportunistic combinations of the two. A bottom–up strategy works by beginning with the code, whereas a top–down strategy operates

by working from a high-level goal. A strategy is formulated by identifying actions to achieve a goal. Strategies guide two mechanisms, namely, chunking and cross-referencing to produce higher-level abstraction structures. Chunking creates new, higher-level abstraction structures from lower-level structures. Cross-referencing means being able to make links between elements of different abstraction levels. This helps in building a mental model of the program under study at different levels of abstractions. In general, understanding a program involves a knowledge base, which represents the expertise and background knowledge a programmer brings to the table, a mental model, and an assimilation process [80]. The assimilation process guides the programmer to look at certain pieces of information, such as a code segment or a comment, and move forward/backward while reading the code. The assimilation process can work in three ways: top–down, bottom–up, and opportunistic.

1.8 SOFTWARE REUSE

The 1968 NATO (North Atlantic Treaty Organization) conference on software engineering is viewed to have germinated the ever growing field of software engineering [81]. The conference focused on *software crises*—the problem of building large, reliable software systems in a controlled way. In that conference, the term *software crisis* was coined for the first time. Even in the first forum on software systems, *software reuse* was pronounced as a means for tackling software crisis. The idea of software reuse was first introduced by Dough McIlroy in a seminal paper [82] in 1968. He proposed to realize reuse by means of library components and automated ways for customizing components to varying degrees of robustness and precision.

Other significant early reuse research developments include David Parnas's idea of program families [83] and Jim Neighbors' introduction of the concepts of domain analysis [84]. A program family is a set of programs whose common properties are so extensive that it becomes advantageous to study the common properties of these programs before analyzing individual differences. On the other hand, domain analysis is an activity of identifying objects and operations of a class of similar systems in a particular problem domain.

Simply stated, software reuse means using existing software knowledge or artifacts during the development of a new system. Reusable assets can include both artifacts and software knowledge. Note that reuse is not constrained to source code fragments. Rather, Capers Jones identified four broad types of artifacts for reuse [85]:

- *Data reuse.* This involves a standardization of data formats. Reusable functions imply a standard data interchange format. Therefore, one of the critical precursors to full reusable software is that of reusable data.
- *Architectural reuse.* This consists of standardizing a set of design and programming conventions dealing with the logical organization of software. The goal is to define a complete set of functional elements which will be needed to create new systems from standard components.

- *Design reuse.* This deals with the reuse of abstract designs that do not include implementation details. These are then implemented specifically to fit the application requirements.
- *Program reuse.* This deals with reusing running code. The software units that are reused may be of different sizes. The whole of software system may be reused by incorporating it without change into other system (COTS product reuse).

Reusability is a property of software assets, which indicates the degree to which the software can be reused. For a software component to be reusable, it must exhibit the following characteristics: high cohesion, low coupling, adaptability, understandability, reliability, and portability. Those characteristics encourage the component's reuse in similar situations. There are two advantages of reusing previously written code [86–88]:

- *Better quality.* If previously tested modules are reused in a new software project, the reused modules are likely to have less faults than new modules. This reduces the overall failure rate of the new software.
- *Increased productivity.* Organizations can save time and other resources by reusing operational modules, thereby increasing their overall productivity. However, the quantum of increase depends upon the size and complexity of the components being reused and the overall size and complexity of the new software which reuses those components. The development cost of any software project is only about 40% of the total cost over its entire life cycle [89]. Significant maintenance benefit also results from reusing quality software. The empirical study conducted by Stephen R. Schach shows that the cost savings during maintenance, as a consequence of reuse, are nearly twice the corresponding savings during development [90].

1.9 OUTLINE OF THE BOOK

Having given the aforementioned brief introduction to software evolution and maintenance, now we provide an outline of the remaining chapters. Each chapter focuses on a specific topic in software evolution and maintenance, and it explains the topic by covering the technical, process, model, and/or practical aspects of the topic. Consequently, the reader gains a broad understanding of the main concepts in software evolution and maintenance.

In Chapter 2 we explain three major maintenance classification schemes based on intention, activity, and evidence. Then we describe Lehman's classification of properties of closed source software (CSS) of type S (Specified), P (Problem), and E (Evolving). The eight laws of software evolution for the E-type CSS system including empirical studies and its practical implications have been introduced. We discuss the origin of FOSS movement and the differences between CSS and

FOSS systems with respect to: (i) team structure; (ii) processes; (iii) releases; and (iv) global factors. In addition, we discuss the empirical research results about the Linux FOSS system to study the laws of evolution, originally proposed for CSS systems. We conclude this chapter with a brief discussion on maintenance of component off-the-shelf-based systems.

Chapter 3 introduces three types of evolution models, namely, reuse-oriented, staged, and change mini-cycle. Next, we discuss the IEEE/EIA 1219 and the ISO IEC 14764 maintenance processes. We explain a framework to make a plan for SCM to control software evolution processes. We close this chapter with a presentation of a state transition model to track the individual change requests as those flow through the organization.

Chapter 4 introduces the concepts of software reengineering based on three basic principles: abstraction, refinement, and alteration. We discuss five basic reengineering approaches: big bang, incremental, partial, iterative, and evolutionary. Next, we discuss two specific models for software reengineering: source code reengineering reference model and phase reengineering model. With the reengineering approaches and models in place, we introduce the concepts and objectives of reverse engineering with an introduction to the Goals/Models/Tools paradigm that divides a process for reverse engineering into three successive phases: Goals, Models, and Tools. In addition, we examine some low-level reverse engineering tasks such as decompilers and disassemblers. DRE for data-oriented applications is explained toward the end of the chapter.

Chapter 5 identifies the problems an organization faces in dealing with legacy information systems and discusses six viable solutions to the problems: freeze, out source, carry on, discard and redevelop, wrap, and migrate. We study four types of wrapping techniques in detail: database wrapper, system service wrapper, application wrapper, and function wrapper. In addition, we discuss five different levels of encapsulations: process level, transaction level, program level, module level, and procedural level. Next, we focus our attention on migration of information systems. First we discuss the migration issues, followed by 13 steps for migration planning to minimize the risk of modernization effort. We discuss seven available migration approaches: Cold Turkey, Database First, Database Last, Composite Database, Chicken Little, Butterfly, and Iterative.

Chapter 6 presents the fundamentals of impact analysis, including the related concepts of ripple effect and change propagation. The reader learns the strengths and limitations of impact analysis techniques. We have selected topics to provide a foundation for enduring value of impact analysis and change propagation.

In Chapter 7, we introduce to the reader different refactoring activities. Different formalisms and techniques to support these activities have been discussed. In addition, we discuss the initial work on software restructuring, such as elimination-of-goto, system sandwich, localization and information hiding, and clustering approaches.

Chapter 8 considers the issues and solutions that underpin program understanding during maintenance. We discuss different models proposed by different researchers. In addition, the concept of protocol analysis is introduced to the readers. The chapter ends with a brief discussion of visualization for software comprehension.

In Chapter 9, we introduce the readers to reuse and domain engineering. Software reuse has the potential to reduce the maintenance cost more than development cost of software projects. We present a taxonomy of reuse, followed by a detailed description of domain and application engineering concepts, including real domain engineering approaches: DARE, FAST, and Koala. Finally, we discuss maturity and cost models associated with reuse.

In the glossary section we have defined all the keywords that have been used in the book. The reader will find about 10 practice questions at the end of each chapter. A carefully chosen list of references is given at the end of each chapter for those who are more curious about the details of some of the topics. Finally, each of the following chapters contains a section on further reading. The further reading section provides pointers to more advanced materials concerning the topics of the chapter.

REFERENCES

[1] M. I. Halpern. 1965. Machine independence: its technology and economics. *Communications of the ACM*, 8(12), 782–785.

[2] L. A. Belady and M. M. Lehman. 1976. A model of large program development. *IBM Systems Journal*, 15(1), 225–252.

[3] M. M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, September, 1060–1076.

[4] R. G. Canning. 1972. The maintenance "iceberg". *EDP Analyzer*, 10(10), 1–14.

[5] E. B. Swanson. 1976. *The Dimensions of Maintenance*. Proceedings of the 2nd International Conference on Software Engineering (ICSE), October 1976, San Francisco, CA. IEEE Computer Society Press, Los Alamitos, CA. pp. 492–497.

[6] N. Chapin, J. F. Hale, K. M. Khan, J. F. Ramil, and W. G. Tan. 2001. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13, 3–30.

[7] K. H. Bennett and V. T. Rajlich. 2000, *Software Maintenance and Evolution: A Roadmap. ICSE, The Future of Software Engineering*, June 2000, Limerick, Ireland. ACM, New York. pp. 73–87.

[8] L. J. Arthur. 1988. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons.

[9] K. H. Bennett and J. Xu. 2003. *Software Services and Software Maintenance*. Proceedings of the 7th European Conference on Software Maintenance and Reengineering, March 2003, Benevento, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 3–12.

[10] M. W. Godfrey and D. M. German. 2008. *The Past, Present, and Future of Software Evolution*. Proceedings of the 2008 Frontiers of Software Maintenance (FoSM), October 2008, Beijing, China. IEEE Computer Society Press, Los Alamitos, CA. pp. 129–138.

[11] D. L. Parnas. 1994. *Software Aging*. Proceedings of 16th International Conference on Software Engineering, May 1994, Sorrento, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 279–287.

[12] M. Jazayeri. 2005. *Species Evolve, Individuals Age*. Proceedings of 8th International Workshop on Principles of Software Evolution (IWPSE), September 2005, Lisbon, Portugal. IEEE Computer Society Press, Los Alamitos, CA. pp. 3–9.

[13] P. Stachour and D. C. Brown. 2009. You don't know jack about software maintenance. *Communications of the ACM*, 52(11), 54–58.

[14] M. M. Lehman, D. E. Perry, and J. F. Ramil. 1998. *On Evidence Supporting the Feast Hypothesis and the Laws of Software Evolution*. Proceedings of the 5th International Software Metrics Symposium (Metrics), November 1998. IEEE Computer Society Press, Los Alamitos, CA. pp. 84–88.

[15] M. M. Lehman and J. F. Ramil. 2006. Software evolution. In: *Software Evolution and Feedback* (Eds N. H. Madhavvji, J. F. Ramil, and D. Perry). John Wiley, West Sussex, England.

[16] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. 1997. *Metrics and Laws of Software Evolution—The Nineties View*. Proceedings of the 4th International Symposium on Software Metrics (Metrics 97), November 1997. IEEE Computer Society Press, Los Alamitos, CA, pp. 20–32.

[17] S. S. Pirzada. 1988. A statistical examination of the evolution of the Unix system. PhD Thesis, Department of Computing, Imperical College, London, England.

[18] M. M. Lehman and L. A. Belady. 1985. *Program Evolution: Processess of Software Change*. Academic Press, London.

[19] M. W. Godfrey and Q. Tu. 2000. *Evolution in Open Source Software: A Case Study*. Proceedings of the International Conference on Software Maintenance, October 2000. IEEE Computer Society Press, Los Alamitos, CA, pp. 131–142.

[20] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz. 2005. *Evolution and Growth in Large Libre Software Projects*. Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE), September 2005, Lisbon, Portugal. IEEE Computer Society Press, Los Alamitos, CA. pp. 165–174.

[21] ISO/IEC 14764:2006 and IEEE Std 14764-2006. 2006. *Software Engineering–Software Life Cycle Processes–Maintenance*. Geneva, Switzerland.

[22] B. A. Kitchenham, G. H. Travassos, A. N. Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. 1999. Towards an ontologyy of software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 11, 365–389.

[23] G. Ramesh and R. Bhattiprolu. 2006. *Software Maintenance*. Tata McGraw-Hill, New Delhi.

[24] M. Vigder and A. Kark. 2006. *Maintaining Cots-Based Systems: Start with Design*. Proceedings of the 5th International Conference on Commercial-Off-The-Shelf (COTS)-Based Software Systems, February 2006, Orlando, Florida. IEEE Computer Society Press, Los Alamitos, CA. pp. 11–18.

[25] D. Hybertson, A. Ta, and W. Thomas. 1997. Maintenance of cots-intensive software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 9, 203–216.

[26] W. W. Royce. 1970. *Managing the Development of Large Software System: Concepts and Techniques*. Proceeding of IEEE WESCON, August 1970, pp. 1–9, Republished in ICSE, Monterey, CA, 1987, pp. 328–338.

[27] N. Schneidewind. 1987. The state of software maintenance. *IEEE Transactions on Software Engineering*, March, 303–309.

[28] N. Chapin. 1988. *Software Maintenance Life Cycle*. Proceedings of the International Conference on Software Maintenance (ICSM), October 1988, Phoenix, Arizona. IEEE Computer Society Press, Los Alamitos, CA, pp. 6–13.

[29] W. K. Sharpley. 1977. *Software Maintenance Planning for Embedded Computer Systems*. Proceedings of the IEEE COMPSAC, November 1977, IEEE Computer Society Press, Los Alamitos, CA, pp. 520–526.

[30] G. Parikh. 1982. The world of software manitenance. In: *Techniques of Program and System Maintenance* (Ed. G. Parikh), pp. 9-13. Little, Brown and Company, Boston, MA.

[31] J. Martin and C. L. McClure. 1983. *Software Maintenance: The Problem and Its Solution*. Prentice-Hall, Englewood Cliffs, NJ.

[32] S. Chen, K. G. Heisler, W. T. Tsai, X. Chen, and E. Leung. 1990. A model for assembly program maintenance. *Journal of Software Maintenance: Research and Practice*, March, pp. 3–32.

[33] D. R. Harjani and J. P. Queille. 1992. *A Process Model for the Maintenance of Large Space Systems Software*. Proceedings of the International Conference on Software Maintenance (ICSM), November 1992, Orlando, FL. IEEE Computer Society Press, Los Alamitos, CA. pp. 127–136.

[34] S. S. Yau, R. A. Nicholi, J. Tsai, and S. Liu. 1988. An integrated life-cycle model for software maintenance. *IEEE Transactions on Software Engineering*, August, pp. 1128–1144.

[35] S. S. Yau and I. S. Collofello. 1980. Some stability measures for software maintenance. *IEEE Transactions on Software Engineering*, November, pp. 545–552.

[36] S. S. Yau, J. S. Collofello, and T. MacGregor. 1978. *Ripple Effect Analysis of Software Maintenance*. COMPSAC, Chicago, Illinois. IEEE Computer Society Press, Piscataway, NJ. pp. 60–65.

[37] B. W. Boehm. 1988. A spiral model of software development and maintenance. *IEEE Computer*, May, pp. 61–72.

[38] V. R. Basili. 1990. Viewing maintenance as reuse-oriented software development. *IEEE Software*, January, pp. 19–25.

[39] R. G. Martin. 2002. *Agile Software Development: Principles, Patterns, and Practices*. Prentice-Hall.

[40] T. Gilb. 1988. *Principles of Software Engineering Management*. Addison-Wesley, Reading, MA.

[41] T. Mens. 2008. Introduction and roadmap: history and challenges of software evolution. In: *Software Evolution* (Eds. T. Mens and S. Demeyer). Springer Verlag, Berlin.

[42] V. T. Rajlich and K. H. Bennett. 2000. A staged model for the software life cycle. *IEEE Computer*, July, pp. 2–8.

[43] ISO/IEC 12207:2006 and IEEE Std 12207-2006. 2008. *System and Software Engineering–Software Life Cycle Processes*. Geneva, Switzerland.

[44] IEEE Standard 1219-1998. 1998. *Standard for Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA.

[45] IEEE Std 1042-1987. 1988. *IEEE Guide to Software Configuration Management*. IEEE, Inc., New York, NY.

[46] K. Narayanaswamy and W. Scacchi. 1987. Maintaining configuration of evolving software systems. *IEEE Transactions of Software Engineering*, March, 13(3), 324–334.

[47] D. Leblang. 1994. The CM challenge: configuration management that works. In: *Configuration Management*, Chapter 1 (Ed. W. F. Tichy). John Wiley, Chichester.

[48] H. Yang and M. Ward. 2003. *Successful Evolution of Software Systems*. Artech House, Boston, MA.

[49] E. J. Chikofsky and J. H. Cross II. 1990. Reverse engineering and design recovery. *IEEE Software*, January, pp. 13–17.

[50] I. Jacobson and F. Lindstrȍm. 1991. *Re-engineering of Old Systems to an Object-oriented Architecture*. Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications, October 1991. ACM Press, New York, NY, pp. 340–350.

[51] P. Benedusi, A. Cimitile, and U. De Carlini. 1992. Reverse engineering processes, design document production, and structure charts. *Journal of Systems Software*, 19, 225–245.

[52] P. Aiken. 1996. *Data Reverse Engineering: Staying the Legacy Dragon*. McGraw-Hill, Boston, New York, NY.

[53] K. H. Bennett. 1995. Legacy systems: coping with success. *IEEE Software*, January, pp. 19–23.

[54] M. Brodie and M. Stonebraker. 1995. *Migrating Legacy Systems*. Morgan Kaufmann, San Mateo, CA.

[55] A. Cimitile, H. Mủller, and R. Klosch (Eds.) 1997. *Pulling Together*. Proceedings of the International Conference on Software Engineering, Workshop on Migration Strategies for Legacy Systems, Available as Technical Report TUV-1841-97-06 from Technical University University of Vienna, Vienna, Austria.

[56] K. Bennett, M. Ramage, and M. Munro. 1999. *Decision Model for Legacy Systems*. IEEE Proceedings on Software, June, pp. 153–159.

[57] W. C. Dietrich Jr., L. R. Nackman, and F. Gracer. 1989. Saving a legacy with objects. *Proceedings of the 1989 ACM OOPSLA Conference on Object-Oriented Programming*, 24(10), 77–83. ACM SIGPLAN Notices, ACM, New York, NY.

[58] H. M. Sneed. 1996. *Encapsulating Legacy Software for Use in Client/Server Systems*. 3rd Working Conference on Reverse Engineering, Washington, DC. IEEE Computer Society Press, Los Alamitos, CA. pp. 104–119.

[59] S. Comella-Dorda, K. Wallnau, R. C. Seacord, and J. Robert. 2000. *A Survey of Black-box Modernization Approaches for Information Systems*. Proceedings of the International Conference on Software Maintenance, October, 2000, San Jose, CA. IEEE Computer Society Press, Los Alamitos, CA. pp. 173–183.

[60] F. P. Coyle. 2000. Legacy integration—changing perpectives. *IEEE Software*, March/April, 37–41.

[61] H. M. Sneed. 1995. Planning the reengineeirng of legacy systems. *IEEE Software*, January, pp. 24–34.

[62] J. Bisbal, D. Lawless, B. Wu, J. Grimson, V. Wade, R. Richardson, and D. O'Sullivan. 1997. A survey of research into legacy system migration. Technical Report TCD-CS-1997-01, Computer Science Department, Trinity College, Dublin, January, pp. 39.

[63] M. Battaglia, G. Savoia, and J. Favaro. 1998. *Renaissance: A Method to Migrate from Legacy to Immortal Software Systems*. Proceedings of Second Euromicro Conference on Software Maintenance and Reengineering, 1998, Florence, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 197–200.

[64] A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio. 2003. Iterative reengineering of legacy systems. *IEEE Transactions on Software Engineering*, March, 225–241.

[65] S. A. Bohner and R. S. Arnold. 1996. An introduction to software change impact analysis. In: *Software Change Impact Analysis* (Eds. S. A. Bohner and R. S. Arnold). IEEE Computer Society Press, Los Alamitos, CA.

[66] A. Podgurski and L. Clrke. 1990. A formal model of program dependencies and its implications for software testing, debugging, and maintenance. *IEEE Transactions of Software Engineering*, September, 16(9), 965–979.

[67] M. J. Harrold and B. Malloy. 1993. A unified interprocedural program representation for maintenance environment. *IEEE Transactions of Software Engineering*, 19(6), 584–593.

[68] S. Black. 2008. Deriving an approximation algorithm for automatic computation of ripple effect measures. *Information and Software Technology*, 50, 723–736.

[69] V. Rajlich. 1997. *A Model for Change Propagation Based on Graph Rewriting*. Proceedings of the International Conference on Software Maintenance (ICSM), October 1997, Bari, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 84–91.

[70] A. E. Hassan and R. C. Holt. 2004. *Predicting Change Propagation in Software Systems*. Proceedings of the International Conference on Software Maintenance (ICSM), October 2004, Chicago, USA. IEEE Computer Society Press, Los Alamitos, CA. pp. 284–293.

[71] N. Ibrahim, W. M. N. Kadir, and S. Deris. 2008. *Comparative Evaluation of Change Propagation Approaches Towards Resilient Software Evolution*. Proceedings of the Third International Conference on Software Engineering Advances, pp. 198–204.

[72] D. E. Perry and W. M. Evangelist. 1987. *An Empirical Study of Software Interface Faults—An Update*. Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences, January 1987, Volume II, pp. 113–126.

[73] M. Fowler. 1999. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley.

[74] W. F. Opdyke. 1992. Refactoring: A program restructuring aid in designing object-oriented application framework. PhD thesis, University of Illinois at Urbana-Champaign.

[75] S. Demeyer. 2008. Object-oriented reengineering. In: *Software Evolution* (Eds. T. Mens and S. Demeyer). Springer Verlag, Berlin.

[76] T. A. Corbi. 1989. Program understanding: challenge for the 1990s. *IBM Systems Journal*, 28(2), pp. 294–306.

[77] H. A. Mủller. 1996. *Understanding Software Systems Using Reverse Engineering Technologies: Research and Practice*. Department of Computer Science, University of Victoria. Available at http://www.rigi.csc.uvic.ca/uvicrevtut/uvicrevtut.html.

[78] P. N. Johnson-laird. 1983. *Mental Model*. Harvard University Press, Cambridge, MA.

[79] K. J. W. Craik. 1943. *The Nature of Explanation*. Cambridge University Press, Cambridge, UK.

[80] S. Letovsky. 1986. *Cognitive Processes in Program Comprehension*. Proceedings of the First Workshop in Empirical Studies of Programmers, pp. 58–79.

[81] P. Nauer, B. Randell, and J. N. Buxton (Eds). 1969. Software engineering. Report on a Conference by the NATO Science Committee, NOATO Scientific Affairs Division, Brussels, Belgium, Available through Petrocelli-Charter, New York.

[82] M. D. McIlroy. 1969. *Mass Produced Software Components*. Proceedings of Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering (Eds. P. Naur, B. Randell, and J. N. Buxton), pp. 138–155. Petrocelli-Charter, New York, NY.

[83] D. L. Parnas. 1976. On the design and development of program families. *IEEE Transactions of Software Engineering*, 2(1), 1–9.

[84] J. M. Neighbors. 1980. Software construction using components. Technical Report 160, Department of Information and Computer Sciences, University of California, Irvine.

[85] T. C. Jones. 1984. Reusability in programming: a survey of the state of the art. *IEEE Transactions of Software Engineering*, 10(5), 488–494.

[86] J. E. Gaffney and T. A. Durek. 1989. Software reuse - key to enhanced productivity: some quantitative models. *Information and Software Technology*, 31(5), 258–267.

[87] R. D. Banker and R. J. Kauffman. 1991. Reuse and productivity in integrated computer-aided software engineering: an emprical study. *MIS Quarterly*, 15(3), 374–401.

[88] V. R. Basili, L. C. Brand, and W. L. Melo. 1996. Machine independence: Its technology and economics. *Communications of the ACM*, 39(10), pp. 104–116.

[89] Gartner Group Inc. 1991. Software engineering strategies. Strategic Analysis Report, Stamford, CT, April.

[90] S. R. Schach. 1994. The economic impact of software reuse on maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, July/August, 6(4), pp. 185–196.

EXERCISES

1. Discuss the differences between software evolution and software maintenance.

2. Explain why a software system which is used in a real-world environment must be changed to not become progressively less useful.

3. What are some characteristics of maintaining software as opposed to new software systems?

4. You are asked to make a change to a system that leaves its functional specification unchanged but affects the design and source code of the system. This can be any of the four types of maintenance mentioned earlier except one. Identify the exception and justify your answer.

(a) Corrective maintenance

(b) Adaptive maintenance

(c) Perfective maintenance

(d) Preventive maintenance

5. Discuss the major differences between COTS-based software development and traditional in-house software development activities.

6. One of the key sources of risks in COTS-based development is the reliance on one or more third-party software vendors. However, this dependence can also present new challenges for the evolution of such systems. Which of the following evolution challenges can be directly attributed to reliance on the vendor?

(a) Lack of control over when errors in components are fixed.

(b) Number and complexity of inter-component interfaces.

(c) Diversity of inter-component interfaces.

(d) Lack of experience and tools for evolving COTS-based systems.

7. What are the objectives of SCM?

8. A feature of any complex change to an existing software system is that it is likely to introduce new defects, even if the aim of the change is to remove defects. When considering whether or not to implement a change request, should this feature be considered as a cost, benefit, or risk associated with the change request?

9. System A is a mission critical legacy system that captures and stores detailed data on product sales. Data from system A must be regularly extracted and loaded into a new system (B), which is to be used to help managers understand the changes in sales patterns from week to week. Initial estimates suggest that the data for 1 week can be extracted and transformed in around 3 hours. What migration frequency would you choose for this new application?

(a) Migrate on update.

(b) Migrate daily, every evening at 2.00 A.M.

(c) Migrate weekly, every Sunday evening at 2.00 A.M.

(d) Migrate monthly, on the last Sunday of every month at 2.00 A.M.

10. What are some of the risks of not doing an impact analysis before effecting a change?

11. What actions can be taken to minimize the impact of fixing defects?

12. What problems do maintainers face when rewriting or reengineering a piece of code? What are the causes of those problems?

13. Explain the term hypotheses in the context of program understanding.

14. What benefits can be derived from reusing software?