4

REENGINEERING

Neither situation nor people can be altered by the interference of an outsider. If they are to be altered, that alteration must come from within.

—Phyllis Bottome

4.1 GENERAL IDEA

Reengineering is the examination, analysis, and restructuring of an existing software system to reconstitute it in a new form and the subsequent implementation of the new form. The goal of reengineering is to:

- understand the existing software system artifacts, namely, specification, design, implementation, and documentation; and
- improve the functionality and quality attributes of the system. Some examples of quality attributes are evolvability, performance, and reusability.

Fundamentally, a new system is generated from an operational system, such that the target system possesses better quality factors. The desired software quality factors include reliability, correctness, integrity, efficiency, maintainability, usability, flexibility, testability, interoperability, reusability, and portability [1]. In other words, reengineering is done to convert an existing "bad" system into a "good" system [2]. Of course there are risks involved in this transformation, and the primary risks are: (i) the target system may not have the same functionality as the existing system; (ii) the target system may be of lower quality than the original system; and (iii) the benefits are not realized in the required time frame [3]. Software systems are reengineered by keeping one or more of the following four general objectives in mind [4]:

- Improving maintainability
- Migrating to a new technology
- Improving quality
- Preparing for functional enhancement.

Improving maintainability. Let us revisit Lehman's second law, namely, *increasing complexity*: "As an E-type program evolves, its complexity increases unless work is done to maintain or reduce it." As systems grow and evolve, the cost of maintenance increases because changes become difficult and time consuming. Consequently, it is unrealistic to avoid reengineering in the long run. The system is redesigned with more explicit interfaces and more relevant functional modules. In addition, both external and internal documentations are made up-to-date. All those activities lead to better maintainability of the system.

Migrating to a new technology. Lehman's first law, namely, *continuing change*, states that "E-type programs must be continually adapted, else they become progressively less satisfactory." A program is continually adapted to make it compatible with its operating environment. In the fast-paced information technology industry, new—and sometimes cheaper—execution platforms include new features, which quickly make the current system outdated, and maybe more expensive. Compatibility of the newer system with the old one is likely to be an issue, because vendors have less motivation to provide support for older parts—both software and hardware—that become incompatible and more expensive. Moreover, as systems evolve, expertise of employees migrates to newer technologies, with fewer staff to maintain the old system. Consequently, organizations with perfectly working software that continues to meet their business needs are forced to migrate to a modern execution platform that includes newer hardware, operating system, and/or language.

Improving quality. Lehman's seventh law, namely *declining quality*, states that "Stakeholders will perceive an E-type program to have declining quality unless it is rigorously maintained and adapted to its environment." As time passes, users make increasingly more change requests to modify the system. Each change causes "ripple effects," implying that one change causes more problems to be fixed. As the system is continually modified as a result of maintenance activities, the reliability of the software gradually decreases to an unacceptable level. Therefore, the system must be reengineered to achieve greater reliability.

Preparation for functional enhancement. Lehman's sixth law, namely, *continuing growth*, states that "The functionality of an E-type program must be continually increased to maintain user satisfaction with the program over its lifetime." This law reflects the fact that all programs, being finite, limit the functionalities to a finite selection from a potentially unbounded set. Properties excluded by the bounds eventually become a source of performance limitations, dissatisfaction, and error. These properties in terms of functionalities must be implemented in the application to satisfy the stakeholders. In general, reengineering is not performed to support more functionalities of a system; rather, as a preparatory step to enhance functionalities, a system is reengineered. For example, if the objective is to transform programs from a procedural to an object-oriented form to distribute them in a client-server architecture, then, at the same time, plan to reduce the maintenance costs by using a language such that the system will be more amenable to changes.

4.2 REENGINEERING CONCEPTS

A good comprehension of the software development processes is useful in making a plan for reengineering. Several concepts applied during software development are key to reengineering of software. For example, *abstraction* and *refinement* are key concepts used in software development, and both the concepts are equally useful in reengineering. It may be recalled that abstraction enables software maintenance personnel to reduce the complexity of understanding a system by: (i) focusing on the more significant information about the system and (ii) hiding the irrelevant details at the moment. On the other hand, refinement is the reverse of abstraction. The principles of abstraction and refinement are explained as follows [5]:

Principle of abstraction. The level of abstraction of the representation of a system can be gradually increased by successively replacing the details with abstract information. By means of abstraction one can produce a view that focuses on selected system characteristics by hiding information about other characteristics.

Principle of refinement. The level of abstraction of the representation of the system is gradually decreased by successively replacing some aspects of the system with more details.

A new software is created by going downward from the top, highest level of abstraction to the bottom, lowest level. This downward movement is known as *forward engineering*. Forward engineering follows a sequence of activities: formulating concepts about the system to identifying requirements to designing the system to implementing the design. On the other hand, the upward movement through the layers of abstractions is called *reverse engineering*. Reverse engineering of software systems is a process comprising the following steps: (i) analyze the software to determine its components and the relationships among the components and (ii) represent the system at a higher level of abstraction or in another form [6]. Some examples of reverse engineering are: (i) decompilation, in which object code is translated into a high-level program; (ii) architecture extraction, in which the design of a program is derived; (iii) document generation, in which information is produced from, say, source code, for better understanding of the program; and (iv) software visualization, in which some aspect of a program is depicted in an abstract way.

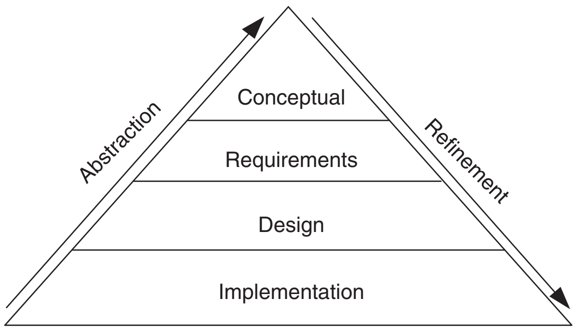

FIGURE 4.1 Levels of abstraction and refinement. From Reference 5. © 1992 IEEE

The concepts of abstraction and refinement are used to create models of software development as sequences of phases, where the phases map to specific levels of abstraction or refinement, as shown in Figure 4.1. The four levels, namely, conceptual, requirements, design, and implementation, are described one by one below:

- *Conceptual level.* At the highest level of abstraction, the software is described in terms of very high-level concepts and its reason for existence (*why*?). In other words, this level addresses the "why" aspects of the system, by answering the question: "Why does the system exist?"
- *Requirements level.* At this level, functional characteristics (*what*?) of the system are described at a high level, while leaving the details out. In other words, this level addresses the "what" aspects of the system by answering the question: "What does the system do?"
- *Design level.* At the design-refinement level, system characteristics (*what and how*?), namely, major components, the architectural style putting the components together, the interfaces among the components, algorithms, major internal data structures, and databases are described in detail. In other words, this level addresses more of "what" and "how" aspects of the system by answering the questions: (i) "What are the characteristics of the system?" and (ii) "How is the system going to possess the characteristics to deliver the functionalities?"
- *Implementation level.* This is the lowest level of abstraction in the hierarchy. At this level, the system is described at a very low level in terms of implementation details in a high-level language. In other words, this level addresses "how" exactly the system is implemented.

In summary, the refinement process can be represented as *why*? → *what*? → *what and how?* → *how*? and the abstraction process can be represented as *how*? → *what and how?* → *what*? → *why*?

A concept, a requirement, a design, and an implementation of a program usually denote different levels of abstraction. Moving from one level to another involves a process of crossing levels of abstraction. Usually a specification is more abstract than its implementation; therefore, the cycle of abstraction and refinement can be represented as follows: *concrete* → *more abstract* → *abstract* → *highly abstract* →

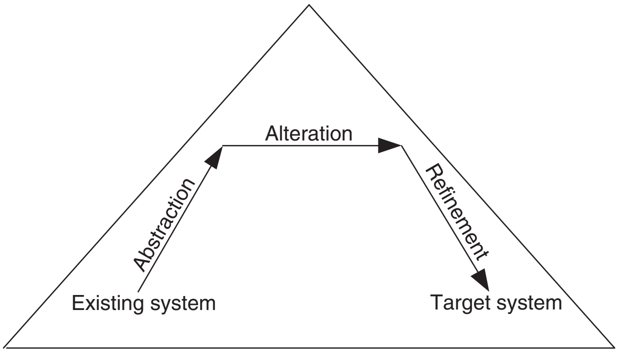

FIGURE 4.2 Conceptual basis for the reengineering process. From Reference 5. © 1992 IEEE

abstract → *less abstract* → *concrete*. Abstraction and refinement are important concepts, and these are useful in reengineering as well as in forward engineering.

In addition to the two principles of abstraction and refinement, an optional principle called *alteration* underlies many reengineering methods. The principle of alteration is defined as follows:

Principle of alteration. The making of some changes to a system representation is known as alteration. Alteration does not involve any change to the degree of abstraction, and it does not involve modification, deletion, and addition of information.

Figure 4.2 shows the use of the three fundamental principles to explain reengineering characteristics. An important conceptual basis for the reengineering process is the sequence {abstraction, alteration, and refinement}. Reengineering principles are represented by means of arrows. Specifically, *abstraction* is represented by an up arrow, *alteration* is represented by a horizontal arrow, and *refinement* by a down arrow. The arrows depicting refinement and abstraction are slanted, thereby indicating the increase and decrease, respectively, of system information. It may be noted that alteration is nonessential for reengineering. Generally, the path from abstraction to refinement is via alteration. Alteration is guided by reengineering strategies as discussed in Section 4.3.2.

Another term closely related to "alteration" is *restructuring*, which was introduced in Chapter 3 and discussed in Chapter 7. In reengineering context, the term "restructuring" is defined as the transformation from one representation form to another at the same relative abstract level while preserving the subject system's external behavior [6]. Restructuring is often used as a form of preventive maintenance.

4.3 A GENERAL MODEL FOR SOFTWARE REENGINEERING

The reengineering process accepts as input the existing code of a system and produces the code of the renovated system. On the one hand, the reengineering process may be as straightforward as translating with a tool the source code from the given language to source code in another language. For example, a program written in BASIC can be translated into a new program in C. On the contrary, the reengineering process may be very complex as explained below:

- recreate a design from the existing source code;
- find the requirements of the system being reengineered;
- compare the existing requirements with the new ones;
- remove those requirements that are not needed in the renovated system;
- make a new design of the desired system; and
- code the new system.

Founded on the different levels of abstractions used in the development of software, Figure 4.3, originally proposed by Eric J. Byrne [5], depicts the processes for all abstraction levels of reengineering. This model suggests that reengineering is a sequence of three activities—reverse engineering, re-design, and forward engineering—strongly founded in three principles, namely, abstraction, alteration, and refinement, respectively.

A visual metaphor called *horseshoe*, as depicted in Figure 4.4, was developed by Kazman et al. [7] to describe a three-step architectural reengineering process. Three distinct segments of the horseshoe are the left side, the top part, and the right side. Those three parts denote the three steps of the reengineering process. The first step, represented on the left side, aims at extracting the architecture from the source code by using the abstraction principle. The second step, represented on the top, involves architecture transformation toward the target architecture by using the alteration principle. Finally, the third step, represented on the right side, involves the generation of the new architecture by means of refinement. One can look at the horseshoe bottom-up to notice how reengineering progresses at different levels of abstraction: source code, functional model, and architectural design.

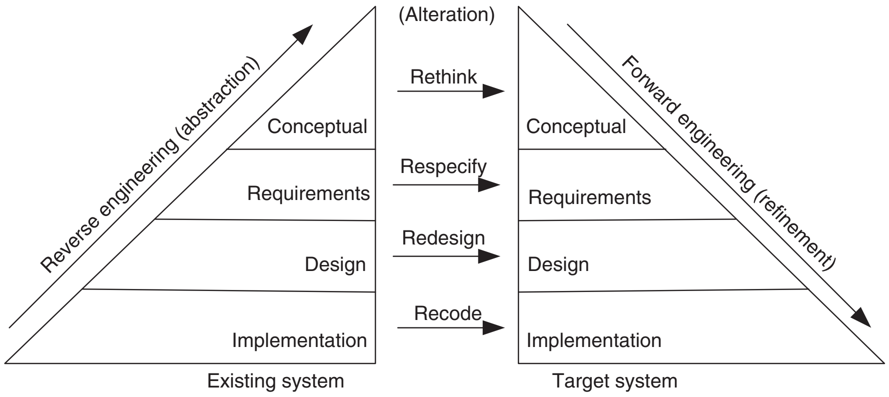

FIGURE 4.3 General model of software reengineering. From Reference 5. © 1992 IEEE

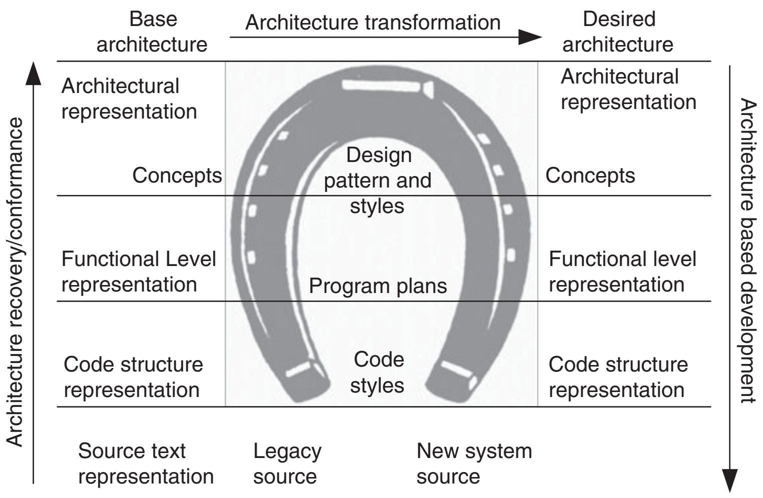
Base architecture
Architecture transformation
Desired architecture
Architecture recovery/conformance
Architectural representation
Concepts
Design pattern and styles
Architectural representation
Concepts
Architecture based development
Functional Level representation
Program plans
Functional level representation
Code structure representation
Code styles
Code structure representation
Source text representation
Legacy source
New system source

FIGURE 4.4 Horseshoe model of reengineering. From Reference 7. © 1998 IEEE

Now, we are in a position to revisit three definitions of reengineering.

- The definition by Chikofsky and Cross II [6]: Software reengineering is the analysis and alteration of an operational system to represent it in a new form and to obtain a new implementation from the new form. Here, a new form means a representation at a higher level of abstraction.
- The definition by Byrne [5]: Reengineering of a software system is a process for creating a new software from an existing software so that the new system is better than the original system in some ways.
- The definition by Arnold [3]: Reengineering of a software system is an activity that: (i) improves the comprehension of the software system or (ii) raises the quality levels of the software, namely, performance, reusability, and maintainability.

In summary, it is evident that reengineering entails: (i) the creation of a more abstract view of the system by means of some reverse engineering activities; (ii) the restructuring of the abstract view; and (iii) implementation of the system in a new form by means of forward engineering activities. This process is formally captured by Jacobson and Lindstörm [8] with the following expression:

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

Referring to the right-hand side of the above equation, the first element, namely, “reverse engineering,” is an activity to create an easier to understand and more abstract form of the system. The third element, namely, “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 made to the original system. Two major dimensions of alteration are change in functionality and change in implementation technique. A change in functionality comes from a change in the business rules [9]. Thus, a modification of the business rules results in modifications of the system. Moreover, change of functionality does not affect how the system is implemented, that is, how forward engineering is carried out. Next, concerning a change of implementation technique, an end-user of a system never knows if the system is implemented in an object-oriented language or a procedural language. Often, reengineering is associated with the introduction of a new development technology, for example, model-driven engineering [10]. A variant of reengineering in which the transformation is driven by a major technology change is called *migration*. Migration of legacy information systems (LIS) is discussed in Chapter 5.

Another common term used by practitioners of reengineering is *rehosting*. Rehosting means reengineering of source code without addition or reduction of features in the transformed targeted source code [11]. Rehosting is most effective when the user is satisfied with the system's functionality, but looks for better qualities of the system. Examples of better qualities are improved efficiency of execution and reduced maintenance costs. To modify a system's characteristics, alteration is performed at an abstraction level with much details about the characteristics. For example, if there is a need to translate the source code of a system to a new programming language, there is no need to perform reverse engineering. Rather, alteration, that is recoding, is done at the source code level. However, at a higher level of abstraction, say, architecture design, reverse engineering is involved and the amount of alterations to be done is increased. Similarly, to respecify requirements by identifying the functional characteristics of the system, reverse engineering techniques are applied to the source code.

4.3.1 Types of Changes

The model in Figure 4.3 suggests that an existing system can be reengineered by following one of four paths. The selection of a specific path for reengineering depends partly on the characteristics of the system to be modified. For a given characteristic to be altered, the abstraction level of the information about that characteristics plays a key role in path selection. Based on the type of changes required, system characteristics are divided into groups: rethink, respecify, redesign, and recode. It is worth noting that, on the one hand, modifications performed to characteristics within one group do not cause any changes at a higher level of abstraction. On the other hand, modifications within a particular group do result in modifications to lower levels of abstraction. For instance, one requirement is reflected in many design components, and one design component is realized by a block of source code. Hence, a small change in a design component may require several modifications to the code. However, the change to the design component should not influence the requirements. Next, the characteristics of each group are discussed in what follows.

Recode. Implementation characteristics of the source program are changed by recoding it. Source code level changes are performed by means of rephrasing and program translation. In the latter approach, a program is transformed into a program in a different language. On the other hand, rephrasing keeps the program in the same language [12].

Examples of translation scenarios are *compilation*, *decompilation*, and *migration*. By means of compilation, one transforms a program written in a high-level language into assembly or machine code. Decompilation is a form of transformation in which high-level source code is discovered from an executable program. In migration, a program is transformed into a program in another language while retaining the program's abstraction level. The language of the new program need not be completely different than the original program's language; rather, it can be a variation of the first language.

Examples of rephrasing scenarios are *normalization*, *optimization*, *refactoring*, and *renovation*. Normalization reduces a program to a program in a sublanguage, that is to a subset of the language, with the purpose of decreasing its syntactic complexity. Elimination of GOTO and module flattening in a program are examples of program normalization. Optimization is a transformation that improves the execution time or space performance of a program. Refactoring is a transformation that improves the design of a program by means of restructuring to better understand the new program.

Redesign. The design characteristics of the software are altered by redesigning the system. Common changes to the software design include: (i) restructuring the architecture; (ii) modifying the data model of the system; and (iii) replacing a procedure or an algorithm with a more efficient one.

Respecify. This means changing the requirement characteristics of the system in two ways: (i) change the form of the requirements and (ii) change the scope of the requirements. The former refers to changing only the form of existing requirements, that is, taking the informal requirements expressed in a natural language and generating a formal specification in a formal description language, such as the Specification and Description Language (SDL) or Unified Modeling Language (UML). The latter type of changes includes such changes as adding new requirements, deleting some existing requirements, and altering some existing requirements.

Rethink. Rethinking a system means manipulating the concepts embodied in an existing system to create a system that operates in a different problem domain. It involves changing the conceptual characteristics of the system, and it can lead to the system being changed in a fundamental way. Moving from the development of an ordinary cellular phone to the development of smartphone system is an example of Rethink.

4.3.2 Software Reengineering Strategies

Three strategies that specify the basic steps of reengineering are rewrite, rework, and replace. The three strategies are founded on three fundamental principles in software engineering, namely, abstraction, alteration, and refinement. The rewrite strategy is based on the principle of alteration. The rework strategy is based on the principles of abstraction, alteration, and refinement. Finally, the replace strategy is based on the principles of abstraction and refinement.

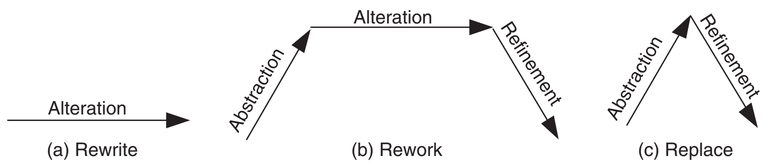

FIGURE 4.5 Conceptual basis for reengineering strategies. From Reference 5. © 1992 IEEE

Rewrite strategy. This strategy reflects the principle of alteration. By means of alteration, an operational system is transformed into a new system while preserving the abstraction level of the original system. For example, the Fortran code of a system can be rewritten in the C language. The rewrite strategy has been further explained in Figure 4.5a.

Rework strategy. The rework strategy applies all the three principles. First, by means of the principle of abstraction, obtain a system representation with less details than what is available at a given level. For example, one can create an abstraction of source code in the form of a high-level design. Next, the reconstructed system model is transformed into the target system representation, by means of alteration, without changing the abstraction level. Finally, by means of refinement, an appropriate new system representation is created at a lower level of abstraction. The main ideas in rework are illustrated in Figure 4.5b. Now, let us consider an example originally given by Byrne [5]. Let the goal of a reengineering project be to restructure the control flow of a program. Specifically, we want to replace the unstructured control flow constructs, namely GOTOs, with more commonly used structured constructs, say, a "for" loop. A classical, rework strategy-based approach to doing that is as follows:

- Application of abstraction: By parsing the code, generate a control flow graph (CFG) for the given system.
- Application of alteration: Apply a restructuring algorithm to the CFG to produce a structured CFG.
- Application of refinement: Translate the new, structured CFG back into the original programming language.

Replace strategy. The replace strategy applies two principles, namely, abstraction and refinement. To change a certain characteristic of a system: (i) the system is reconstructed at a higher level of abstraction by hiding the details of the characteristic and (ii) a suitable representation for the target system is generated at a lower level of abstraction by applying refinement. Figure 4.5c succinctly represents the replace strategy. Let us reconsider the GOTO example given by Byrne [5]. By means of abstraction, a program is represented at a higher level without using control flow concepts. For instance, a module's behavior can be described by its net effect, with no mention of control flow. Next, by means of refinement, the system is represented at a lower level of abstraction with a new structured control flow. In summary, the original unstructured control flow is replaced with a structured control flow.

4.3.3 Reengineering Variations

Three reengineering strategies and four broad types of changes were discussed in the preceding sections: (i) rewrite, rework, and replace are the three reengineering strategies and (ii) rethink, respecify, redesign, and recode are the four types of changes. The reengineering strategies and the change types can be combined to create different process variations. Three process factors cause variability in reengineering processes:

- the level of abstraction of the system representation under consideration;
- the kind of change to be made: rethink, respecify, redesign, and recode; and
- the reengineering strategy to be applied: rewrite, rework, and replace.

Possible variations in reengineering processes have been identified in Table 4.1. The table is interpreted by asking questions of the following type: If *A* is the abstraction level of the representation of the system to be reengineered and the plan is to make a *B* type of change, can I use strategy *C*? The table shows 30 reengineering

TABLE 4.1 Reengineering Process Variations

		Reengineering Strategy		
Starting Abstraction Level	Type Change	Rewrite	Rework	Replace
Implementation level	Recode	Yes	Yes	Yes
	Redesign	Bad	Yes	Yes
	Respecify	Bad	Yes	Yes
	Rethink	Bad	Yes*	Yes*
Design level	Recode	No	No	No
	Redesign	Yes	Yes	Yes
	Respecify	Bad	Yes	Yes
	Rethink	Bad	Yes*	Yes*
Requirement level	Recode	No	No	No
	Redesign	No	No	No
	Respecify	Yes	Yes	Yes
	Rethink	Bad	Yes*	Yes*
Conceptual level	Recode	No	No	No
	Redesign	No	No	No
	Respecify	No	No	No
	Rethink	Yes	Yes*	Yes*

Source: From Reference 5. © 1992 IEEE.
Yes—One can produce a target system.
Yes*—Same as Yes, but the starting degree of abstraction is lower than the uppermost degree of abstraction within the conceptual abstraction level.
No—One cannot start at abstraction level *A*, make *B* type of changes by using strategy *C*, because the starting abstraction level is higher than the abstraction level required by the particular type of change.
Bad—A target system can be created, but the likelihood of achieving a good result is low.

process variations. Out of the 30 variations, 24 variations are likely to produce acceptable solutions.

4.4 REENGINEERING PROCESS

An ordered set of activities designed to perform a specific task is called a *process*. For ease of understanding and communication, processes are described by means of process models. For example, in the software development domain, the Waterfall process model is widely used in developing well-understood software systems. Process models are used to comprehend, evaluate, reason about, and improve processes. Intuitively, process models are described by means of important relationships among data objects, human roles, activities, and tools. In this section, we discuss two process models for software reengineering.

Similarly, by understanding and following a process model for software reengineering, one can achieve improvements in how software is reengineered. The process of reengineering a large software system is a complex endeavor. For ease of performing reengineering, the process can be specialized in many ways by developing several variations. In a reengineering process, the concept of *approach* impacts the overall process structure. If a particular process model requires fine-tuning for certain project goals, those approaches need to be clearly understood. Five major approaches will be explained in the following subsections.

4.4.1 Reengineering Approaches

There are five basic approaches to reengineering software systems. Each approach advocates a different path to perform reengineering [13, 14]. Several considerations are made while selecting a particular reengineering approach:

- objectives of the project;
- availability of resources;
- the present state of the system being reengineered; and
- the risks in the reengineering project.

The five approaches are different in two aspects: (i) the extent of reengineering performed and (ii) the rate of substitution of the operational system with the new one. The five approaches have their own risks and benefits. In the following, we introduce the five basic approaches to software reengineering one by one.

Big Bang approach. The "Big Bang" approach replaces the whole system at once. Once a reengineering effort is initiated, it is continued until all the objectives of the project are achieved and the target system is constructed. This approach is generally used if reengineering cannot be done in parts. For example, if there is a need to move to a different system architecture, then all components affected by such a move must be changed at once. The consequent advantage is that the system is brought into its new environment all at once. On the other hand, the disadvantage of Big Bang is that the reengineering project becomes a monolithic task, which may not be desirable in all situations. In addition, the Big Bang approach consumes too much resources at once for large systems and takes a long stretch of time before the new system is visible.

Incremental approach. As the name indicates, by means of this approach a system is reengineered gradually, one step closer to the target system at a time. Thus, for a large system, several new interim versions are produced and released. Successive interim versions satisfy increasingly more project goals than their preceding versions. The desired system is said to be generated after all the project goals are achieved. The advantages of this approach are as follows: (i) locating errors becomes easier, because one can clearly identify the newly added components and (ii) it becomes easy for the customer to notice progress, because interim versions are released. The incremental approach incurs a lower risk than the "Big Bang" approach due to the fact that as a component is reengineered, the risks associated with the corresponding code can be identified and monitored. The disadvantages of the incremental approach are as follows: (i) with multiple interim versions and their careful version controls, the incremental approach takes much longer to complete; and (ii) even if there is a need, the entire architecture of the system cannot be changed.

Partial approach. In this approach, only a part of the system is reengineered and then it is integrated with the nonengineered portion of the system. One must decide whether to use a "Big Bang" approach or an "Incremental" approach for the portion to be reengineered. The following three steps are followed in the partial approach:

- In the first step, the existing system is partitioned into two parts: one part is identified to be reengineered and the remaining part to be not reengineered.
- In the second step, reengineering work is performed using either the "Big Bang" or the "Incremental" approach.
- In the third step, the two parts, namely, the not-to-be-reengineered part and the reengineered part of the system, are integrated to make up the new system.

The afore-described partial approach has the advantage of reducing the scope of reengineering to a level that best matches an organization's current need and desire to spend a certain amount of resources. A reduced scope implies that the selected portions of a system to be modified are those that are urgently in need of reengineering. A reduced scope of reengineering takes less time and costs less. A disadvantage of the partial approach is that modifications are not performed to the interface between the portion modified and the portion not modified.

Iterative approach. The reengineering process is applied on the source code of a few procedures at a time, with each reengineering operation lasting for a short time. This process is repeatedly executed on different components in different stages. During the execution of the process, ensure that the four types of components can