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

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

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.

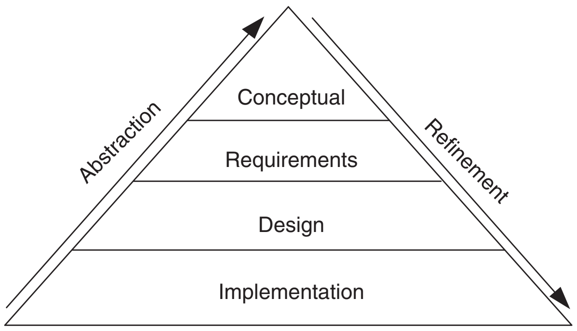

FIGURE 4.1 Levels of abstraction and refinement. From Reference 5. © 1992 IEEE

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* →

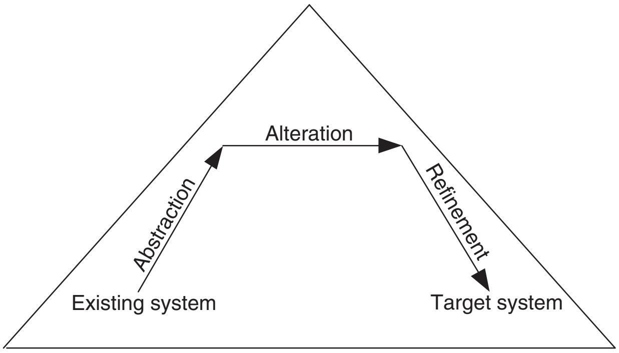

FIGURE 4.2 Conceptual basis for the reengineering process. From Reference 5. © 1992 IEEE

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.

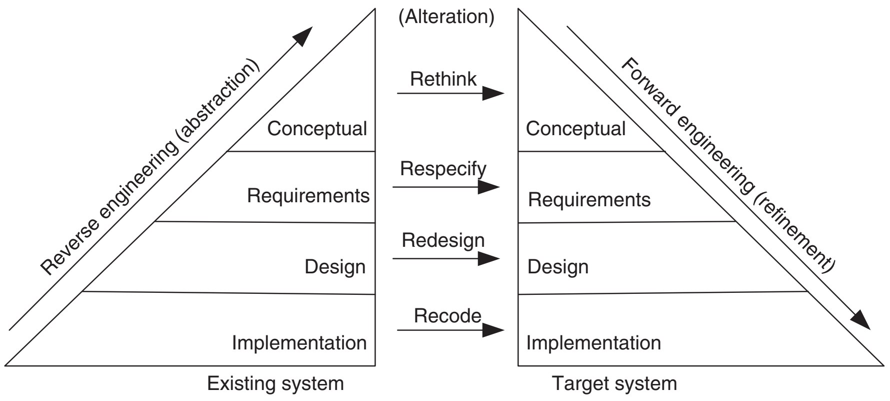

FIGURE 4.3 General model of software reengineering. From Reference 5. © 1992 IEEE

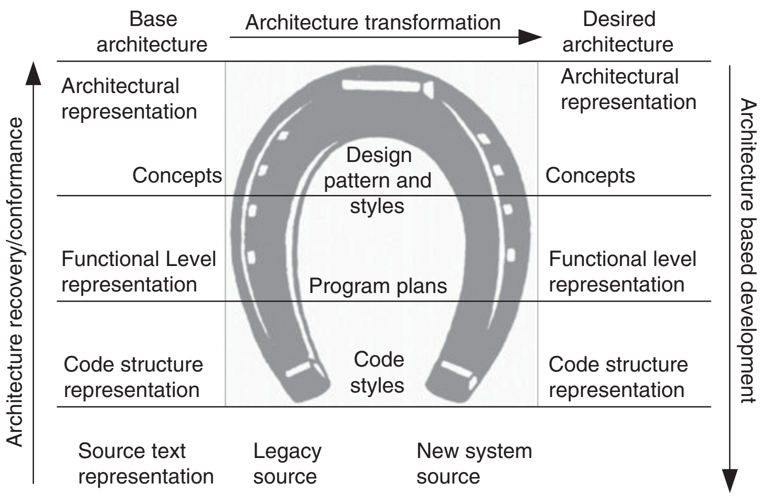
Base architecture
Architecture transformation
Desired architecture
Architectural representation
Concepts
Design pattern and styles
Architectural representation
Concepts
Functional Level representation
Program plans
Functional level representation
Code structure representation
Code styles
Code structure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

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.

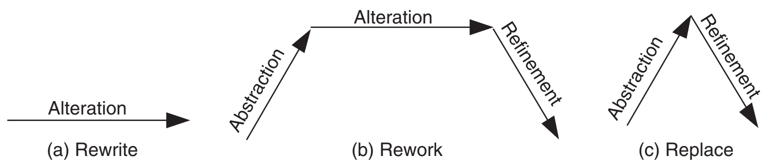

FIGURE 4.5 Conceptual basis for reengineering strategies. From Reference 5. © 1992 IEEE

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 coexist: old components not reengineered, components currently being reengineered, components already reengineered, and new components added to the system. Their coexistence is necessary for the operational continuity of the system. There are two advantages of the iterative reengineering process: (i) it guarantees the continued operation of the system during the execution of the reengineering process and (ii) the maintainers' and the users' familiarities with the system are preserved. The disadvantage of this approach is the need to keep track of the four types of components during the reengineering process. In addition, both the old and the newly reengineered components need to be maintained.

Evolutionary approach. Similar to the ”Incremental” approach, in the ”Evolutionary” approach components of the original system are substituted with reengineered components. However, in this approach, the existing components are grouped by functions and reengineered into new components. Software engineers focus their reengineering efforts on identifying functional objects irrespective of the locations of those components within the current system. As a result, the new system is built with functionally cohesive components as needed. There are two advantages of the “Evolutionary” approach: (i) the resulting design is more cohesive and (ii) the scope of individual components is reduced. As a result, the “Evolutionary” approach works well to convert an operational system into an object-oriented system. A major disadvantage of the approach is as follows: all the functions with much similarities must be first identified throughout the operational system; next, those functions are refined as one unit in the new system.

4.4.2 Source Code Reengineering Reference Model

The Source Code Reengineering Reference Model (SCORE/RM) is useful in understanding the process of reengineering of software. The model was proposed by Colbrook, Smythe, and Darlison [15]. The framework, depicted in Figure 4.6, consists of four kinds of elements: function, documentation, repository database, and metrication. The function element is divided into eight layers, namely, encapsulation, transformation, normalization, interpretation, abstraction, causation, regeneration, and certification. The eight layers provide a detailed approach to (i) rationalizing the system to be reengineered by removing redundant data and altering the control flow; (ii) comprehending the software's requirements; and (iii) reconstructing the software according to established practices. The top six of the eight layers shown in Figure 4.6 constitute a process for reverse engineering, and the bottom three layers constitute a process for forward engineering. Both the processes include causation, because it represents the derivation of requirements specification for the software.

Improvements in the software as a result of reengineering are quantified by means of the metrication element. The metrication element is described in terms of the relevant software metrics before executing a layer and after executing the same layer. The specification, constraints, and implementation details of both the old and the new versions of the software are described in the documentation element. The repository database is the information store for the entire reengineering process, containing the following kinds of information: metrication, documentation, and both the old and the

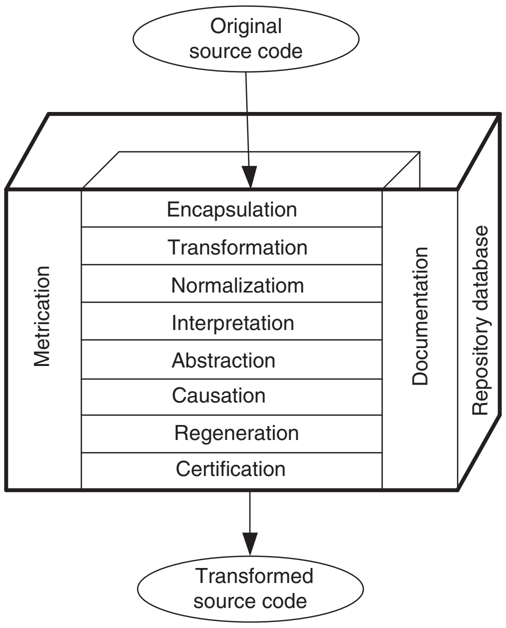

FIGURE 4.6 Source code reengineering reference model. From Reference 15. © 1990 IEEE

new source codes. The interfaces among the elements are shown in Figure 4.7. For simplicity, any layer is referred to as (*N*)-layer, while its next lower and next higher layers are referred to as (*N* − 1)-layer and the (*N* + 1)-layer, respectively. The three types of interfaces are explained as follows:

- Metrication/function: (*N*)-MF—the structures describing the metrics and their values.

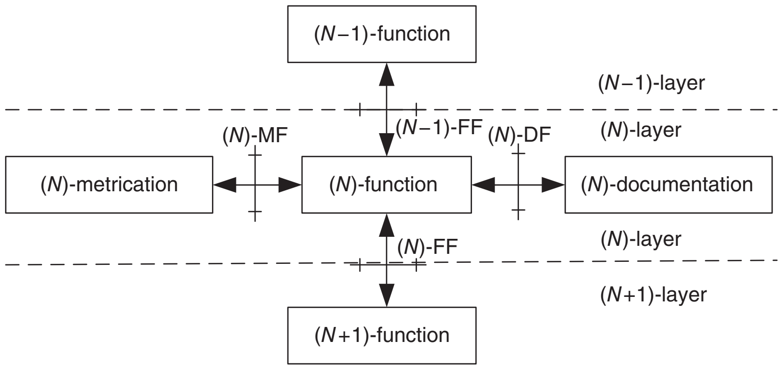

FIGURE 4.7 The interface nomenclature. From Reference 15. © 1990 IEEE. "(*N*)-" represents the *N*th layer

- Documentation/function: (*N*)-DF—the structures describing the documentation.
- Function/function: (*N*)-FF—the representation structures for source code passed between the layers.

The functions of the individual layers are discussed in the following.

Encapsulation: This is the first layer of reverse engineering. In this layer, a reference baseline is created from the original source code. The goal of the reference baseline is to uniquely identify a version of a software and to facilitate its reengineering. The following functions are expected of this layer:

- *Configuration management.* The changes to the software undergoing maintenance are recorded by following a well-documented and defined procedure for later use in the new source code. This step requires strong support from upper management by allocating resources.
- *Analysis.* The portions of the software requiring reengineering are evaluated. In addition, cost models for the tangible benefits are put in place.
- *Parsing.* The source code of the system to be reengineered is translated into an intermediate language (IL). The IL can have several dialects, depending upon the relationship between the languages for the new code and the original code. All the reengineering algorithms act upon the IL representation of the source code.
- *Test generation.* This refers to the design of certification tests and their results for the original source code. Certification tests are basically acceptance tests to be used as baseline tests. The “correctness” of the newly derived software will be evaluated by means of the baseline tests.

Transformation: To make the code structured, its control flow is changed. This layer performs the following functions:

- *Rationalization of control flow.* The control flow is altered to make code structured.
- *Isolation.* All the external interfaces and referenced software are identified.
- *Procedural granularity.* This refers to the sizing of the procedures, by using the ideas of high cohesion and low coupling.

Normalization: In this stage data and their structures are scrutinized by means of the following functions:

- *Data reduction.* Duplicate data are eliminated. To be consistent with the requirements of the program, databases are modified.
- *Data representation.* The life histories of the data entities are now generated. The life histories describe how data are changed and reveal which control structures act on the data.

Interpretation: The process of deriving the meaning of a piece of software is started in this layer. The interpretation layer performs the following functions:

- *Functionalization.* This is additional rationalization of the data and control structure of the code, which (i) eliminates global variables and/or (ii) introduces recursion and polymorphic functions.
- *Program reading.* This means annotating the source code with logical comments.

Abstraction: The annotated and rationalized source code is examined by means of abstractions to identify the underlying object hierarchies. The abstraction layer performs the following functions:

- *Object identification.* The main idea in object identification is (i) separate the data operators and (ii) group those data operators with the data they manipulate.
- *Object interpretation.* Application domain meanings are mapped to the objects identified above. It is the different implementations of those objects that produce differences between the renovated code and the original code.

Causation: This layer performs the following functions:

- *Specification of actions.* This refers to the services provided to the user.
- *Specification of constraints.* This refers to the limitations within which the software correctly operates.
- *Modification of specification.* The specification is extended and/or reduced to accurately reflect the user's requirements.

Regeneration: Regeneration means reimplementing the source code using the requirements and the functional specifications. The layer performs the following functions:

- *Generation of design.* This refers to the production and documentation of the detailed design.
- *Generation of code.* This means generating new code by reusing portions of the original code and using standard libraries.
- *Test generation.* New tests are generated to perform unit and integration tests on the source code developed and reused.

Certification: The newly generated software is analyzed to establish that it is (i) operating correctly; (ii) performing the specified requirements; and (iii) consistent with the original code. The layer performs the following functions:

- *Validation* and *Verification.* The new system is tested to show its correctness.

- *Conformance.* Tests are performed to show that the renovated source code performs at the minimum all those functionalities that were performed by the original source code. It is not known what form the equivalence relationship must take, particularly when modification of the specification is likely to have occurred during reengineering.

4.4.3 Phase Reengineering Model

The phase model of software reengineering was originally proposed by Byrne and Gustafson [14] in circa 1992. The model comprises five phases: analysis and planning, renovation, target system testing, redocumentation, and acceptance testing and system transition, as depicted in Figure 4.8. The labels on the arcs denote the possible information that flows from the tail entities of the arcs to the head entities. A major process activity is represented by each phase. Tasks represent a phase's activities, and tasks can be further decomposed to reveal the detailed methodologies.

Analysis and planning. The first phase of the model is analysis and planning. Analysis addresses three technical and one economic issue. The first technical issue concerns the present state of the system to be reengineered and understanding its properties. The second technical issue concerns the identification of the need for the system to be reengineered. The third technical issue concerns the specification of the characteristics of the new system to be produced. Specifically, one identifies (i) the characteristics of the system that are needed to modified and (ii) the functionalities of the system that are needed to be modified. The identified modifications are analyzed and eventually integrated into the system. In addition, understand the expected characteristics of the new system to plan the required work.

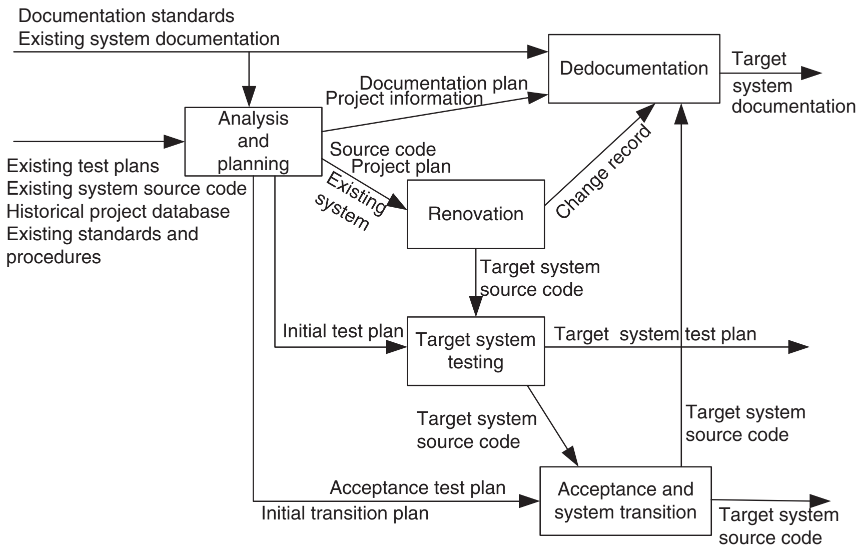

FIGURE 4.8 Software reengineering process phases. From Reference 14. © 1992 IEEE

TABLE 4.2 Tasks—Analysis and Planning Phase

Task	Description
Implementation motivations and objectives	List the motivations for reengineering the system. List the objectives to be achieved.
Analyze environment	Identify the differences between the existing and the target environments. Differences can influence system changes.
Collect inventory	Form a baseline for knowledge about the operational system by locating all program files, documents, test plans, and history of maintenance.
Analyze implementation	Analyze the source code and record the details of the code.
Define approach	Choose an approach to reengineer the system.
Define project procedures and standards	Procedures outline how to perform reviews and report problems. Standards describe the acceptable formats of the outputs of processes.
Identify resources	Determine what resources are going to be used; ensure that resources are ready to be used.
Identify tools	Determine and obtain tools to be used in the reengineering project.
Data conversion planning	Make a plan to effect changes to databases and files.
Test planning	Identify test objectives and test procedures, and evaluate the existing test plan. Design new tests if there is a need.
Define acceptance criteria	By means of negotiations with the customers, identify acceptance criteria for the target system.
Documentation planning	Evaluate the existing documentation. Develop a plan to redocument the target system.
Plan system transition	Develop an end-of-project plan to put the new system into operation and phase out the old one.
Estimation	Estimate the resource requirements of the project: effort, cost, duration, and staffing.
Define organizational structure	Identify personnel for the project, and develop a project organization.
Scheduling	Develop a schedule, including dependencies, for project phases and tasks.

Source: From Reference 14. © 1992 IEEE.

The economic issue concerns a cost and benefit analysis of the reengineering project. The economics of reengineering must compare with the costs, benefits, and risks of developing a new system as well as the costs and risks of maintaining an old system [16]. Planning includes (i) understanding the scope of the work; (ii) identifying the required resources; (iii) identifying the tasks and milestones; (iv) estimating the required effort; and (v) preparing a schedule. The tasks to be performed in this phase are listed in Table 4.2.

Renovation. An operational system is modified into the target system in the renovation phase. Two main aspects of a system are considered in this phase: (i) representation of the system and (ii) representation of external data. In general, the former

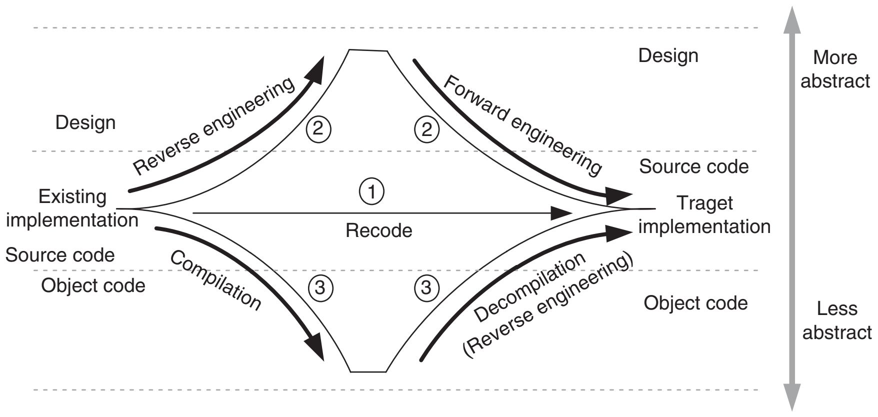

FIGURE 4.9 Replacement strategies for recoding

refers to source code, but it may include the design model and the requirement specification of the existing system. On the other hand, the latter refers to the database and/or data files used by the system. Often the external data are reengineered, and it is known as *data reengineering*. Data reengineering has been discussed in Section 4.8.

An operational system can be renovated in many ways, depending upon the objectives of the project, the approach followed, and the starting representation of the system. It may be noted that the starting representation can be source code, design, or requirements. Table 4.1 in Section 4.3.3 recommends several alternatives to renovate a system. Selection of a specific renovation approach is a management decision.

Let us consider an example of a project in which the objective is to recode the system from Fortran to C. Figure 4.9 shows the three possible replacement strategies. First, to perform source-to-source translation, program migration is used. Program migration accepts the source code for the system to be reengineered as input and produces new source code as output for the target system. Second, a high-level design is constructed from the operational source code, say, in Fortran, and the resulting design is reimplemented in the target language, C in this case. Finally, a mix of compilation and decompilation is used to obtain the system implementation in C: (i) compile the Fortran code to obtain object code and (ii) decompile the object code to obtain a C version of the program. For all the three approaches, the end effects are the same, but the tasks to be executed are different for each of the three replacement strategies.

Target system testing. In this phase for system testing, faults are detected in the target system. Those faults might have been introduced during reengineering. Fault detection is performed by applying the target system test plan on the target system. The same testing strategies, techniques, methods, and tools that are used in software development are used during reengineering. For example, apply the existing system-level test cases to both the existing and the new systems. Assuming that the two systems have identical requirements, the test results from both the scenarios must be the same.

Redocumentation. In the redocumentation phase, documentations are rewritten, updated, and/or replaced to reflect the target system. Documents are revised according to the redocumentation plan. The two major tasks within this phase are (i) analyze new source code and (ii) create documentation. Documents requiring revision are requirement specification, design documentation, a report justifying the design decisions, assumptions made in the implementation, configuration, user and reference manuals, on-line help, and the document describing the differences between the existing and the target systems. Different documents require different redocumentation tasks. A task for redocumentation comprises detailed subtasks to make a plan, actually update the document, and review the document.

Acceptance and system transition. In this final phase, the reengineered system is evaluated by performing acceptance testing. Acceptance criteria should already have been established in the beginning of the project. Should the reengineered system pass those tests, preparation begins to transition to the new system. On the other hand, if the reengineered system fails some tests, the faults must be fixed; in some cases, those faults are fixed after the target system is deployed. Upon completion of the acceptance tests, the reengineered system is made operational, and the old system is put out of service. System transition is guided by the prior developed transition plan.

4.5 CODE REVERSE ENGINEERING

Reverse engineering was first applied in electrical engineering to produce schematics from an electrical circuit. It was defined as the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system [17]. In the context of software engineering, Chikofsky and Cross II [6] defined *reverse engineering* as a process to (i) identify the components of an operational software; (ii) identify the relationships among those components; and (iii) represent the system at a higher level of abstraction or in another form. In other words, by means of reverse engineering one derives information from the existing software artifacts and transforms it into abstract models to be easily understood by maintenance personnel. The factors necessitating the need for reverse engineering are as follows [18]:

- The original programmers have left the organization.
- The language of implementation has become obsolete, and the system needs to be migrated to a newer one.
- There is insufficient documentation of the system.
- The business relies on software, which many cannot understand.
- The company acquired the system as part of a larger acquisition and lacks access to all the source code.
- The system requires adaptations and/or enhancements.
- The software does not operate as expected.

The above factors imply that a combination of both high-level and low-level reverse engineering steps need to be applied. High-level reverse engineering means creating

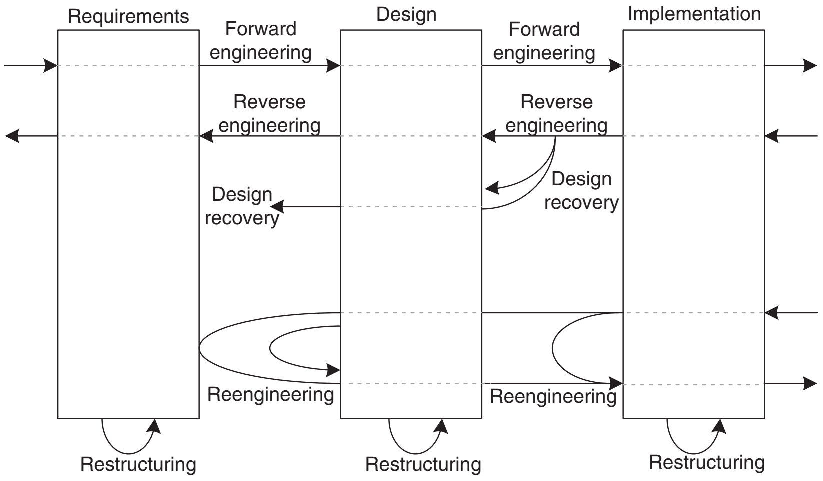

FIGURE 4.10 Relationship between reengineering and reverse engineering. From Reference 6. © 1990 IEEE

abstractions of source code in the form of design, architecture, and/or documentation. Low-level reverse engineering, discussed in Section 4.7, means creating source code from object code or assembly code.

Reverse engineering is performed to achieve two key objectives: *redocumentation of artifacts* and *design recovery*. The former aims at revising the current description of components or generating alternative views at the same abstraction level. Examples of redocumentation are pretty printing and drawing CFGs. On the other hand, the latter creates design abstractions from code, expert knowledge, and existing documentation [19]. In design recovery the domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher-level abstractions beyond those obtained directly by examining the system itself. The relationship between forward engineering, reengineering, and reverse engineering is shown in Figure 4.10.

Although difficulties faced by software maintenance personnel gave rise to the idea of software reverse engineering, it can be used to solve problems in related areas as well. Six objectives of reverse engineering, as identified by Chikofsky and Cross II [6], are generating alternative views, recovering lost information, synthesizing higher levels of abstractions, detecting side effects, facilitating reuse, and coping with complexity. If source code is the only reliable representation of a system, following the IEEE Standard for Software Maintenance [20], one can perform reverse engineering on the system to understand it. Reverse engineering has been effectively applied in the following problem areas:

- redocumenting programs [21];
- identifying reusable assets [22–25];
- discovering design architectures [7, 26–30];

- recovering design patterns [31, 32];
- building traceability between code and documentation [33–35];
- finding objects in procedural programs [36];
- deriving conceptual data models [37–40];
- detecting duplications and clones [41–44];
- cleaning up code smells [45];
- aspect-oriented software development [46];
- computing change impact [47];
- transforming binary code into source code [48];
- redesigning user interfaces [49–51];
- parallelizing largely sequential programs [52];
- translating a program to another language [53, 54];
- migrating data [55];
- extracting business rules [9, 56, 57];
- wrapping legacy code [58];
- auditing security and vulnerability [59, 60]; and
- extracting protocols of network applications [61].

Six key steps in reverse engineering, as documented in the IEEE Standard for Software Maintenance [20], are:

- partition source code into units;
- describe the meanings of those units and identify the functional units;
- create the input and output schematics of the units identified before;
- describe the connected units;
- describe the system application; and
- create an internal structure of the system.

The first three of the six steps involve local analysis, because those are performed at the unit level. On the other hand, the remaining three steps involve global analysis, because those steps are performed at the system level. A high-level organizational paradigm is found to be useful while setting up a reverse engineering process, as advocated by Benedusi et al. [21, 62]. The high-level paradigm plays two roles: (i) define a framework to use the available methods and tools and (ii) allow the process to be repetitive. They propose a paradigm, namely, *Goals/Models/Tools*, which partitions a process for reverse engineering into three ordered stages: Goals, Models, and Tools.

Next, the three phases are explained one by one.

Goals. In this phase, the reasons for setting up a process for reverse engineering are identified and analyzed. Analyses are performed to identify the information needs of the process and the abstractions to be created by the process. The team setting up the process first acquires a good understanding of the forward engineering activities and the environment where the products of the reverse engineering process will be used. Results of the aforementioned comprehension are used to accurately identify (i) the information to be generated and (ii) the formalisms to be used to represent the information. For example, the design documents to be generated from source code are as follows:

- Low-level documents give both an overview and detailed descriptions of individual modules; detailed descriptions include the structures of the modules in terms of control flow and data structures.
- High-level documents give (i) a general description of the software product and (ii) a detailed description of its structuring in terms of modules, their interconnections, and the flow of information between modules.

Models. In this phase, the abstractions identified in the Goals stage are analyzed to create representation models. Representation models include information required for the generation of abstractions. Activities in this phase are:

- identify the kinds of documents to be generated;
- to produce those documents, identify the information and their relations to be derived from source code;
- define the models to be used to represent the information and their relationships extracted from source code; and
- to produce the desired documents from those models, define the abstraction algorithm for reverse engineering.

The important properties of a reverse engineering model are expressive power, language independence, compactness, richness of information content, granularity, and support for information-preserving transformation.

Tools. In this phase, tools needed for reverse engineering are identified, acquired, and/or developed in-house. Those tools are grouped into two categories: (i) tools to extract information and generate program representations according to the identified models and (ii) tools to extract information and produce the required documents. Extraction tools generally work on source code to reconstruct design documents. Therefore, those tools are ineffective in producing inputs for an abstraction process aiming to produce high-level design documents.

4.6 TECHNIQUES USED FOR REVERSE ENGINEERING

Fact-finding and information gathering from the source code are the keys to the Goal/Models/Tools paradigm. In order to extract information which is not explicitly available in source code, automated analysis techniques are used. The well-known analysis techniques that facilitate reverse engineering are *lexical analysis, syntactic* *analysis, control flow analysis, data flow analysis, program slicing, visualization,* and *program metrics*. In the following sections, we explain these techniques one by one.

4.6.1 Lexical Analysis

Lexical analysis is the process of decomposing the sequence of characters in the source code into its constituent lexical units. Various useful representations of program information are enabled by lexical analysis. Perhaps the most widely used program information is the cross reference listing. A program performing lexical analysis is called a lexical analyzer, and it is a part of a programming language's compiler. Typically it uses rules describing lexical program structures that are expressed in a mathematical notation called regular expressions. Modern lexical analyzers are automatically built using tools called lexical analyzer generators, namely, lex and flex (fast lexical analyzer) [63].

4.6.2 Syntactic Analysis

The next most complex form of automated program analysis is syntactic in nature. Compilers and other tools such as interpreters determine the expressions, statements, and modules of a program. Syntactic analysis is performed by a parser. Here, too, the requisite language properties are expressed in a mathematical formalism called context-free grammars. Usually, these grammars are described in a notation called Backus–Naur Form (BNF). In the BNF notation, the various program parts are defined by rules in terms of their constituents. Similar to syntactic analyzers, parsers can be automatically constructed from a description of the programmatical properties of a programming language. YACC is one of the most commonly used parsing tools [63].

Two types of representations are used to hold the results of syntactic analysis: *parse tree* and *abstract syntax tree*. The former is the more primitive one of the two. It is similar to the parsing diagrams used to show how a natural language sentence is broken up into its constituents. However, a parse tree contains details unrelated to actual program meaning, such as the punctuation, whose role is to direct the parsing process. For instance, grouping parentheses are implicit in the tree structure, which can be pruned from the parse tree. Removal of those extraneous details produces a structure called an *Abstract Syntax Tree* (AST). An AST contains just those details that relate to the actual meaning of a program. Because an AST is a tree, nodes of the tree can be visited in a pre-set manner, such as depth-first order, and the information contained in the node is delivered to the analyzer. Many tools have been based on the AST concept; to understand a program, an analyst makes a query in terms of the node types. The query is interpreted by a tree walker to deliver the requested information.

4.6.3 Control Flow Analysis

After determining the structure of a program, control flow analysis (CFA) can be performed on it [64]. The two kinds of CFA are *intraprocedural analysis* and *interprocedural analysis*. The former shows the order in which statements are executed within a subprogram, whereas the latter shows the calling relationship among program units. Intraprocedural analysis is performed by generating CFGs of subprograms. The idea of *basic blocks* is central to constructing a CFG. A basic block is a maximal sequence of program statements such that execution enters at the top of the block and leaves only at the bottom via a conditional or an unconditional branch statement. A basic block is represented with one node in the CFG, and an arc indicates possible flow of control from one node to another. A CFG can directly be constructed from an AST by walking the tree to determine basic blocks and then connecting the blocks with control flow arcs. A CFG shows an abstract view of the ways in which a subprogram can execute.

Interprocedural analysis is performed by constructing a call graph [65, 66]. Calling relationships between subroutines in a program are represented as a call graph which is basically a directed graph. Specifically, a procedure in the source code is represented by a node in the graph, and the edge from node f to g indicates that procedure f calls procedure g. Call graphs can be *static* or *dynamic*. A dynamic call graph is an execution trace of the program. Thus a dynamic call graph is exact, but it only describes one run of the program. On the other hand, a static call graph represents every possible run of the program.

4.6.4 Data Flow Analysis

Although CFA is useful, many questions cannot be answered by means of CFA. For example, CFA cannot answer the question: Which program statements are likely to be impacted by the execution of a given assignment statement? To answer this kind of questions, an understanding of definitions (def) of variables and references (uses) of variables is required. Normally, if a variable appears on the left-hand side of an assignment statement, then the variable is said to be defined. On the contrary, if a variable appears on the right-hand side of an assignment statement, then it is said to be referenced in that statement.

Data flow analysis (DFA) concerns how values of defined variables flow through and are used in a program [67]. CFA can detect the possibility of loops, whereas DFA can determine data flow anomalies [68]. One example of data flow anomaly is that an undefined variable is referenced. Another example of data flow anomaly is that a variable is successively defined without being referenced in between. DFA enables the identification of code that can never execute, variables that might not be defined before they are used, and statements that might have to be altered when a bug is fixed.

4.6.5 Program Slicing

Originally introduced by Mark Weiser, program slicing has served as the basis of numerous tools [69]. The slice of a program for a given variable at a given line of code is the portion of the program that gives a value to the variable at that point. Therefore, if one determines during debugging that the value of a variable at a specific

```
[1]     int i;
[2]     int sum = 0;
[3]     int product = 1;
[4]     for(i = 0; ((i < N) && (i % 2 = 0)); i++) {
[5]        sum = sum + i;
[6]        product = product * i;
        }
[7]     printf("Sum = ", sum);
[8]     printf("Product = ", product);
```

FIGURE 4.11 A block of code to compute the sum and product of all the even integers in the range $[0, N)$ for $N \geq 3$

line is incorrect, one may look at the corresponding program slice to find the faulty code. In Weiser's definition, a slicing criterion of a program P is $S < p; v >$ where p is a program point and v is a subset of variables in P. A *program slice* is a portion of a program with an execution behavior identical to the initial program with respect to a given criterion but may have a reduced size.

Weiser introduced the concept of *backward slice* [69]. A backward slice with respect to a variable v and a given point p comprises all instructions and predicates which affect the value of v at point p. Backward slices answer the question "What program components might effect a selected computation?" The dual of backward slicing is *forward slicing*, and it was proposed by Binkley et al. [70]. With respect to a variable v and a point p in a program, a forward slide comprises all the instructions and predicates which may depend on the value of v at p. Note that the statements in a forward program slice execute *after* the slicing criterion. Forward slicing answers the question "What program components might be effected by a selected computation?" [71].

As an example, let us consider the program shown in Figure 4.11 which is a block of code in C. The *backward slice*, given the slicing criterion $S <$ [7]; sum $>$ is shown in Figure 4.12. For the slicing criterion $S <$ [3]; product $>$, the forward program slice has been shown in Figure 4.13.

Besides detecting defects, program slicing is also used to extract business rules [57] and in refactoring which is discussed in Chapter 7. Tip's article [72] provides a comprehensive survey of program slicing techniques and their applications.

```
[1]     int i;
[2]     int sum = 0;
[4]     for(i = 0; ((i < N) && (i % 2 = 0)); i++) {
[5]        sum = sum + i;
        }
[7]     printf("Sum = ", sum);
```

FIGURE 4.12 The backward slice of code obtained from Figure 4.11 by using the criterion $S <$ [7]; sum $>$

```
[3]     int product = 1;
[4]     for(i = 0; ((i < N) && (i % 2 = 0)); i++) {
[6]        product = product * i;
      }
[8]     printf("Product = ", product);
```

FIGURE 4.13 The forward slice of code obtained from Figure 4.11 by using the criterion $S <$ [3]; product $>$

4.6.6 Visualization

Software visualization is a useful strategy to enable a user to better understand software systems. In this strategy, a software system is represented by means of a visual object to gain some insight into how the system has been structured. The visual representation of a software system impacts the effectiveness of the code analysis or design recovery techniques.

Two important notions of designing software visualization using 3D graphics and virtual reality technology are *visualizations* and *representations*. These two notions are introduced by Young and Munro [73] in order to evaluate the effectiveness of 3D software visualizations. The authors described a collection of desirable properties of both the concepts when building visualization software, which are summarized in this section.

- *Representation.* This is the depiction of a single component by means of graphical and other media.
- *Visualization.* It is a configuration of an interrelated set of individual representations-related information making up a higher-level component.

Essentially, graphical symbols are used to represent components. In a call graph, for example, the nodes and arcs are the representations, whereas the graph itself is the visualization. For effective software visualization, one needs to consider the properties and structure of the symbols used in software representation and visualization.

When creating a representation or evaluating its effectiveness, following key attributes are considered:

- *Individuality.* The individuality property means that different types of components should have different looking representations.
- *Distinctive appearance.* Even in a large visualization compacted into a small space, two representations should be quickly identifiable as being either dissimilar or identical. It may be necessary to increase the visual complexity of a representation in order to have an easily distinguishable appearance among a large number of representations.
- *High information content.* A representation should display some information about the corresponding component type. However, increased information content increases the visual complexity of the representation.
- *Low visual complexity.* Low visual complexity enhances the user's understanding of the representation, and it leads to better performance of the visualization

system. A representation should be easy to understand in addition to appearing *distinctive*. The possibility of a trade-off exists between viewing a system with a large number of simple representations and viewing the same system with fewer but complex representations.

- *Scalability of visual complexity.* It should be relatively easy to increase or reduce the amount of information presented to the user, depending upon the use context. Scalability enables the display of:
  - a small number of components by giving their maximum information; and
  - an overview of a collection of a large number of components, by revealing less about the components.
- *Flexibility for integration into visualizations.* Representations contain both extrinsic and intrinsic dimensions which can be used to encode information about components for quick visual understanding. On the one hand, position and motion are examples of extrinsic dimensions. On the other hand, shape, color, size, and angular velocity are examples of intrinsic dimensions. A representation with many intrinsic dimensions is rendered less flexible for adoption in a visualization.
- *Suitability for automation.* The ability to automate processes for generating representations and visualizations is key to the widespread adoption of visualization models in reengineering. Therefore, representations need to be designed with the objective of being usable in an automation process.

The following requirements are taken into account while designing a visualization:

- *Simple navigation.* Visualizations are designed with their users in mind. Visualizations are structured with added features to enable users to easily navigate through the visualization. Navigations are made simple so that users quickly become familiar with the details of the system.
- *High information content.* While not overwhelming the viewer, a visualization should show as much details as possible. A balance between high information content and low visual complexity needs to be retained.
- *Low visual complexity.* The complexity of the information presented in a visualization impacts the structural complexity of the visualization. By means of abstraction, the visual complexity can be reduced. For example, too much details about components need not be presented.
- *Varying levels of detail.* Users should have the option to view a variety of details, in the form of hierarchy, information content, and type of information. For example, initially, users expect to see the visualized system in its entirety. Gradually the user will begin exploring specific areas of interest. Visualization should support the user to view increasing levels of details as the user learns more about the components.
- *Resilience to change.* Visualizations need to be robust against small changes to the information content of the visualization. For example, small changes in information content should not lead to major changes in the structure of

visualization. Large changes to visualizations as a result of minor changes to the information content will simply require the user to spend more time on learning the structure once again.

- *Effective visual metaphors.* Metaphors enable users to easily understand software systems in terms of concepts already seen in everyday life. For example, in the city metaphor: (i) an object-oriented software system is represented as a collection of elements of a city and (ii) the city is traversed to enable the user to gain an understanding of the sense of locality. The user interacts with the city model to understand the program.
- *Friendly user interface.* Interactions with the visualizations should be intuitive without introducing undue overheads.
- *Integration with other information sources.* The elements of a software system and the elements of a visualization are different in look and details. Visualizations depict systems in a form which is very different than the elements that they represent. For example, a hexagon may represent a certain module. Therefore, being able to correlate the elements of a visualization with the original information is very important. For example, a red hexagon on a visualization, representing a module, can be linked to the module's actual source code.
- *Good use of interactions.* A highly interactive visualization is key to sustaining user interests in the system being comprehended. Those interactions should give the user several ways to understand the system: top view of the system, component view, interactions among components, and details of individual components.
- *Suitability for automation.* A visualization framework must be amenable to automation. Without automation, it is of no practical use.

4.6.7 Program Metrics

To understand and control the overall software engineering process, program metrics are applied. Table 4.3 summarizes the commonly used program metrics. The early program metric research focused on *complexity metrics*, and one of the most widely used complexity metrics is cyclomatic complexity [74]. The concept of function point (FP) was introduced in late 1970s by Albrecht [75] as an alternative metrics based on simple source line count. The aim of FP is to measure the amount of functionality delivered by a program. Intuitively, the more functionality a program has, the larger is the FP count. Based on a module's *fan-in* and *fan-out* information flow characteristics, Henry and Kafura [76] define a complexity metric, $C_p = (\textit{fan-in} \times \textit{fan-out})^2$. Fan-in and fan-out have been explained in Table 4.3. A large fan-in and a large fan-out may be symptoms of a poor design.

In the 1990s, large-scale adoption of object-oriented (OO) programming techniques gave rise to some OO design metrics known as Chidamber and Kemerer (CK) metric suite [77]. Six performance metrics are found in the CK metric suite as follows:

- Weighted methods per class (WMC)—This is the number of methods implemented within a given class.

TABLE 4.3 Commonly Used Software Metrics

Metric	Description
Lines of code (LOC)	The number of lines of executable code
Global variable (GV)	The number of global variables
Cyclomatic complexity (CC)	The number of linearly independent paths in a program unit is given by the cyclomatic complexity metric [74].
Read coupling	The number of global variables read by a program unit
Write coupling	The number of global variables updated by a program unit
Address coupling	The number of global variables whose addresses are extracted by a program unit but do not involve read/write coupling
Fan-in	The number of other functions calling a given function in a module
Fan-out	The number of other functions being called from a given function in a module
Halstead complexity (HC)	It is defined as effort: $E = D * V$, where: Difficulty: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$; Volume: $V = N \times \log_2 n$ Program length: $N = N_1 + N_2$; Program vocabulary: $n = n_1 + n_2$ n_1 = the number of distinct operators n_2 = the number of distinct operands N_1 = the total number of operators N_2 = the total number of operands
Function points	It is a unit of measurement to express the amount of business functionality an information system provides to a user [75]. Function points are a measure of the size of computer applications and the projects that build them

- Response for a class (RFC)—This is the number of methods that can potentially be executed in response to a message being received by an object of a given class. It is the number of methods implemented within a class plus the number of methods accessible to an object class due to inheritance.
- Lack of cohesion in methods (LCOM)—For each attribute in a given class, calculate the percentage of the methods in the class using that attributes. Next, compute the average of all those percentages, and subtract the average from 100%.
- Coupling between object class (CBO)—This is the number of distinct noninheritance-related classes on which a given class is coupled. A class is said to be coupled to another class if it uses methods or attributes of the other class.
- Depth of inheritance tree (DIT)—This is the length of the longest path from a given class to the root in the inheritance hierarchy.
- Number of children (NOC)—This is the number of classes that directly inherit from a given class.

The Chidamber–Kemerer metrics has been investigated in the context of fault proneness. Basili et al. [78] studied the fault proneness in software programs using eight student projects. They observed that the WMC, CBO, DIT, NOC, and RFC metrics were correlated with defects, while the LCOM metric was not correlated with defects. Kontogiannis et al. [42] developed techniques to detect clones in programs using five kinds of metrics:

- fan-out;
- the ratio of the total count of input and output variables to the fan-out;
- cyclomatic complexity;
- function points metric; and
- Henry and Kafura information-flow metric.

In their framework, the basic assumption is that if code fragments c_1 and c_2 are similar under a set of features measured by metric M, then metric values $M(c_1)$ and $M(c_2)$ will also be close. Similarity is gauged by Euclidean distance defined in the five-dimensional space of the above measures. Metric-based approach has been also applied to finding duplicated web pages or finding clones in web documents [79, 80].

4.7 DECOMPILATION VERSUS REVERSE ENGINEERING

A decompiler takes an executable binary file and attempts to produce readable high-level language source code from it. The output will, in general, not be the same as the original source code, and may not even be in the same language. Actual recovery of the original source code is not really possible. The decompiler does not provide the original programmers' annotations that provide vital instructions as to the functioning of the software. There are significant elements in most high-level languages that are just omitted during the compilation process, and it is impossible to recover those elements. Usually, there will be few, if any, comments or meaningful variable names, except for library function names. During compilation, one may enable some assertion code in the debug builds and disable it in the retail builds. Decompilation techniques were originally developed during the 1960s and the 1970s by Maurice Halstead, which form the basis for today's decompilers [81]. Disassemblers are programs that take a program's executable binary as input and generate text files that contain the assembly language code for the entire program or parts of it. Disassembly is a processor-specific process, but some disassemblers support multiple CPU architectures. The user requires a good knowledge of the specific machine's assembly language, and the output is voluminous for nontrivial programs.

Decompilation, or disassembly, is a reverse engineering process, since it creates representations of the system at a higher level of abstraction [6]. However, traditional reverse engineering from source code entails the recognition of "goals," or "plans," which must be known in advance [82]. For example, the "goal" can be to increase the overall comprehension of the program for maintenance.

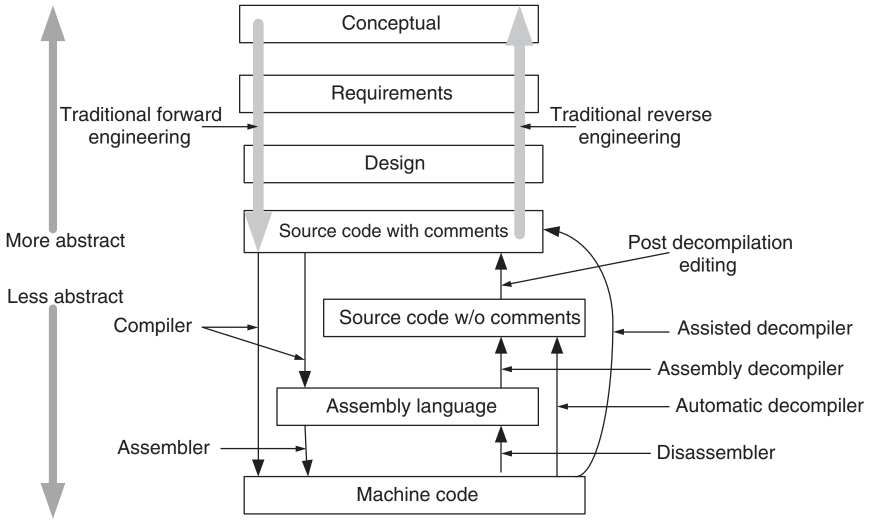

FIGURE 4.14 Relationship between decompilation and traditional reengineering. From Reference 83. © 2007

Decompilation provides the basis for comprehension, maintenance, and new development, with source code, but any high-level comprehension is provided by the reader [83]. Decompilation provides a limited kind of program comprehension. The relationship between decompilation and the traditional reengineering model of Byrne (see Figure 4.3) is depicted in Figure 4.14. As one can see, decompilation stops where architecture abstraction starts. Decompilation is certainly reverse engineering, since it is increasing the level of abstraction. However, compilation is not considered part of the forward engineering, since it is an automatic step.

Initially, decompilers aided program migration from one machine to another. As decompilation capabilities have increased, a wide range of potential applications emerged. Examples of new applications are recovery of lost source code, error correction, security testing, learning algorithms, interoperability with other programs, and recovery of someone else's source code (to determine an algorithm for example) [59]. However, not all uses of decompilers are legal uses. Most of the applications must be examined from the patent and/or copyright infringement point of view. It should be noted that it is never going to be possible to accurately predict beforehand whether or not a particular decompilation is going to be considered legal [84, 85]. License agreements may also bind the user to operate the program in a certain way and to avoid using decompilation or disassembly techniques on that program. It is recommended to seek legal counsel before starting any low-level reverse engineering project.

4.8 DATA REVERSE ENGINEERING

There has been considerable effort to develop concepts and methods to reengineer *data-oriented applications* [86, 87]. A data-oriented application is centered around a set of permanent files or a database. As a persistent data structure is the central part of the data-oriented applications, most approaches focus on database *schema analysis* [37, 39, 40, 88] (data reverse engineering (DRE)) and/or *schema translation and redesign* (data forward engineering) [39, 89, 90]. In addition, the procedural portions of *data-oriented applications* have to be adapted to the newly redesigned schema in order to complete the reengineering task.

Remark: A persistent data structure always preserves the previous version of itself when it is modified. A simple example of persistent data structure is the singly linked list. A data structure is partially persistent if all versions can be accessed but only the newest version can be modified.

DRE is defined as "the use of structured techniques to reconstitute the data assets of an existing system" [91, 92]. By means of structured techniques, existing situations are analyzed and models are constructed prior to developing the new system. The discipline in structured techniques makes the process of DRE economically viable. No doubt, data are valuable assets in all organizations. Student information, patient history, and billing addresses of customers are examples of data assets. The two vital aspects of a DRE process are (i) recover *data assets* that are valuable and (ii) *reconstitute* the recovered data assets to make them more useful. In other words, DRE increases the value of the existing data assets, making it more attractive for organizations to conduct business efficiently and effectively. In practice, the purpose of DRE is as follows [38, 93]:

- *Knowledge acquisition.* Knowledge acquisition is a method of learning. It includes elicitation, collection, analysis, modeling, and validation of information for software projects. The need for knowledge acquisition is pivotal to reverse engineering of *data-oriented applications*. The data portion—be it flat files or a relational database—must be clearly understood in a reverse engineering process.
- *Tentative requirements.* DRE of an operational system can identify the tentative requirements of the replacement system. DRE ensures that the functionality of the current system is not forgotten or overlooked.
- *Documentation.* DRE improves the documentation of existing systems, especially when the original developers are no longer available for advice. Maintenance of legacy software is assisted by the new documentation.
- *Integration.* DRE facilitates integration of applications, because (i) a logical model of encompassed software is a prerequisite for integration and (ii) a logical model of encompassed software presents a plausible model of how the program will function in certain environmental conditions.
- *Data administration.* As data are increasingly used as information, the data owner must be able to perform data administration easily and pragmatically.

DRE allows companies to manage data correctly and efficiently. Data administration, also known as data resource management, is an organizational function working in the areas of information systems that plans, organizes, describes, and controls data resources.

- *Data conversion.* One needs to understand the logical connection between the old database and the new one before converting the old data. Data conversion is the migration of the data instance from the old database to the new one.
- *Software assessment.* DRE represents one of the evaluation criteria for a software product, because DRE can be used to assess the database management system (DBMS) schema of the software [94]. A relational database contains a catalog that describes the various elements in the system. The catalog divides the database into sub-databases known as schemas. Within each schema are database objects—tables, views, and privileges. The catalog itself is a set of tables with its own schema name. Tables in the catalog cannot be modified directly; rather, those are modified indirectly with schema statements. The quality of the database design is an indicator of the quality of the software as a whole. Reverse engineering provides an unusual source of insight.
- *Quality assessment.* The overall quality of a software system can be assessed with DRE techniques, because a flawed design of a persistent data structure is likely to lead to faults in the software system. From data structure analysis, companies can decide whether or not to purchase and maintain COTS components.
- *Component reuse.* The concept of reuse has increasingly become popular amongst software engineers. DRE tools and techniques offer the opportunity to access and extract software components. Quite often these components need to be modified one way or another before they can be reused.

The complexity of reverse engineering *data-oriented applications* can be reduced because one can reverse engineer databases almost independent of the code. DRE deals only with data components of the *data-oriented applications*. Reverse engineering of a *data-oriented application*, including its user interface, begins with DRE. Recovering the specifications, that is the *conceptual schema* in database realm, of such applications is known as database reverse engineering (DBRE) [40]. A DBRE process facilitates understanding and redocumenting an application's database and files. The baselines for a generic DBRE methodology was proposed by Hainaut et al. [87], as discussed in the following.

By means of a DBRE process, one can recreate the complete logical and conceptual schemas of a database physical schema. The *conceptual schema* is an abstract, implementation-independent description of the stored data. Entity-relationship (ER) diagrams are commonly used as a schema notation. A conceptual schema, described with an ER diagram, comprises entities, relationships among entities, attributes of entities, and various constraints to capture an application's concepts and structures. On the one hand, a *logical schema* describes the data structures in concrete forms as those are implemented by the data manager. For example, the logical schema of a relational database precisely describes its tables, keys, and the data constraints. On the other hand, the *physical schema* of a database implements the logical schema by describing the physical constructs, namely, indices, and parameters, namely, page size and buffer management policy. Undoubtedly, deep understanding of the forward design process is needed to reverse engineer a database. The forward design process of a database comprises three basic phases as follows:

- *Conceptual phase*. In this phase, user requirements are gathered, studied, and formalized into a conceptual schema. The phase of conceptual schema has no impact on reverse engineering.
- *Logical phase*. In this phase, the conceptual schema is expressed as a simple model, which is suitable for optimization reasoning. Independent of the target DBMS, the model can be optimized. Next, it is translated according to the target model. The model can be further optimized according to data management system (DMS)-dependent rules.
- *Physical phase*. Now the logical schema is described in the data description language (DDL) of the DMS and the host programming language. The views needed by the application programs are expressed partly in DDL and partly in the host language.

A DBRE process is based on backward execution of the logical phase and the physical phase, beginning with the results of the physical phase. The process is divided into two main phases, namely, *data structure extraction* and *data structure conceptualization*. The two phases relate to the recovery of two different schemas: (i) the first one retrieves the present structure of data from their DDL/host language representation and (ii) the second one retrieves a conceptual schema that describes the semantics underlying the existing data structures. Figure 4.15 shows the general architecture of a reference DBRE methodology.

4.8.1 Data Structure Extraction

The complete DMS schema, including the structures and constraints, are recovered in this phase. In general, database systems provide a description of this schema in some machine processable form. Schemas are a rich starting point to begin a DBRE process. Schemas can be refined through further analysis of other components of the application such as subschema, screen and report layouts, procedures, and documentation. The problem is much more complex for standard files, for which no computerized description of their structure exists in most cases. For example, three standard files provided by Unix are Standard In, Standard Out, and Standard Error. Every running program under Unix starts with three standard opened files. By default, all inputs will come from the keyboard and all outputs (both normal and error output) will go to the screen. Each source program must be analyzed in order to detect partial structure of the files. For most real-world applications, *data structure elicitation* techniques are used to discover the constraints and structures

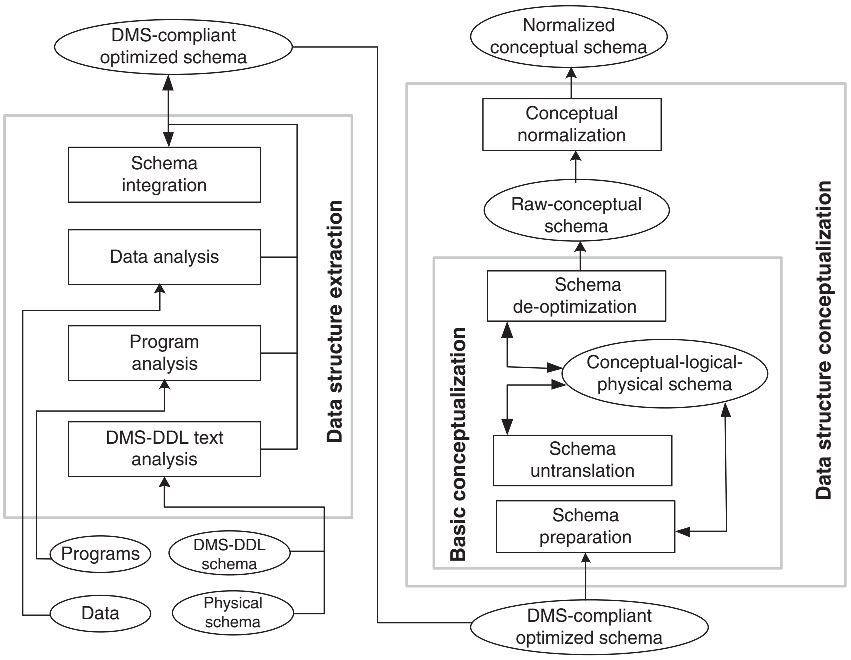

FIGURE 4.15 General architecture of the DBRE methodology. From Reference 95. © 1997 IEEE

that were implicitly incorporated into the implementation [40]. In this methodology, data structures are extracted by means of the following main processes:

DMS–DDL text analysis. Data structure declaration statements in a given DDL, found in the schema scripts and application programs, are analyzed to produce an approximate logical schema.

Program analysis. This means analyzing the source code in order to detect integrity constraints and evidences of additional data structures.

Data analysis. This means analyzing the files and databases to (i) identify data structures and their properties, namely, unique fields and functional dependencies in files and (ii) test hypothesis such as "could this field be a foreign key to this file?"

Schema integration. The analyst is generally presented with several schemas while processing more than one information source. Each of those multiple schemas offers a partial view of the data objects. All those partial views are reflected on the final logical schema via a process for *schema integration*.

4.8.2 Data Structure Conceptualization

In this phase, one detects and transforms or discards redundancies, DMS-dependent constructs, technical optimization, and nonconceptual structures. The phase comprises two sub-phases: *basic conceptualization* and *conceptual normalization*, as explained in the following.

Basic conceptualization. In this sub-phase, relevant semantic concepts are extracted from an underlying logical schema, by solving two different problems requiring very different methods and reasoning: *schema untranslation* and *schema de-optimization*. It is important to note that first the schema is made ready by cleaning it before tackling those two problems.

- *Making the schema ready.* First, the original schema might be using some concrete constructs, such as files and access keys, which might have been useful in the data structure extraction phase, but now can be eliminated. Second, some names can be translated to more meaningful names. Third, some parts of the schema might be restructured before trying to interpret them.
- *Schema untranslation.* The existing logical schema is a technical translation of the initial conceptual constructs. Untranslation means identifying the traces of those translations, and replacing them by their original conceptual constructs.
- *Schema de-optimization.* An optimized schema is generally more difficult to understand. Therefore, in a logical schema, it is useful to identify constructs included to perform optimization and replace those constructs.

Conceptual normalization. The basic conceptual schema is restructured for it to have the desired qualities, namely, simplicity, readability, minimality, extensibility, and expressiveness. Examples of conceptual normalization are (i) replace some entity types by relationship types; (ii) replace some entity types by attributes; (iii) make the *is–a* relation explicit; and (iv) standardize the names.

4.9 REVERSE ENGINEERING TOOLS

In this section, we analyze a number of commercial and academic reverse engineering tools. Each of these tools has one or more distinct features compared to others. We put emphasis on the purpose of each tool and the functionality that it supports. It is important to note that software reverse engineering is a complex process that tools can only support, not completely automate. There is a need for a good deal of human intervention with any reverse engineering project. The tools can provide a new view of the product, as shown in Figure 4.16. The basic structure of reverse engineering tools, as outlined by Chikofsky and Cross II, is as follows:

- The software system to be reverse engineered is analyzed.
- The results of the analysis are stored in an information base.
- View composers use the information base to produce alternative views of the system.

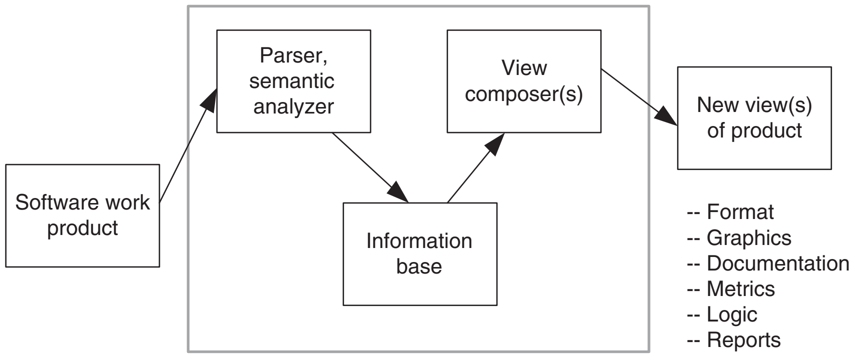

FIGURE 4.16 Basic structure of reverse engineering tools. From Reference 6. © 1990 IEEE

Ada SDA (System Dependency Analyzer) [96] is a tool that supports analysis and migration of Ada programs. It parses Ada code into an Object-Oriented Abstract Structural Type (OO-AST) and performs simple analysis. The analysis is focused on two areas, namely, portability check and architectural analysis. Portability check is carried out by searching for non-Ada routines, such as X/Motif in the program which are treated as *cliché*. The idea of *cliché* is explained after this paragraph. Architectural analysis involves determining what subunits and packages are used and the compilation dependency order.

Remark: A cliché is a frequently occurring pattern in a program, for example, data structures, algorithms, or domain-specific patterns. A *plan* is a representation of a cliché. Plan recognition involves recognizing cliché using plan. The recognition of plan is a symbolic pattern matching process [97]. To locate such patterns, what is needed is a search mechanism that is closer to the mental model of the software engineer. Program plans are abstract representations of source code fragments. Comparison methods are used to help recognize instances of program plans in a subject system. This process involves pattern matching at the programming language semantic level.

CodeCrawler [98] is a language-independent reverse engineering tool which combines metrics and software visualization [99]. CodeCrawler is written in Visualworks Smalltalk and it runs on every major platform. It uses two-dimensional displays to visualize software. Nodes represent software entities or abstraction of them, while the edges represent relationships between those entities. This tool combines the capability of showing software entities and their relationships with the capability of visualizing software metrics using *polymetric views*. CodeCrawler has been successfully used to reverse engineer several large industrial software systems.

DMS (Design Maintenance System) toolkit developed by Semantic Design, Inc. [100], is composed of a set of tools for carrying out reengineering of medium- or large-scale software systems. The capacities include analysis, porting, translation, interface changes, or other massive regular change and/or domain-specific program generation. This toolkit is the first implementation of the DMS, a software engineering environment that supports the incremental construction and maintenance of large application systems, driven by semantics and captured designs [101].

FermaT is the generic name for a set of tools designed by Software Migration Ltd. [102], specifically to support assembler code comprehension, maintenance, and migration. Based upon a unique IL called Wide Spectrum Language (WSL) and program transformation technique, FermaT is able to capture and manipulate the entire functionality of an assembler program, thereby significantly improving maintenance productivity and reducing cost [103]. These tools can be used to transform assembly language to C or COBOL.

GXL (Graph eXchange Language) [104] is an XML-based format for sharing data between tools. GXL represents typed, attributed, directed, ordered graphs which are extended to represent hyper-graphs and hierarchical graphs. An advantage of GXL is that one can exchange instance graphs together with their corresponding schema information in a uniform format. The language allows software reengineers to combine single-purpose tools especially for parsing, source code extraction, architecture recovery, data flow analysis, pointer analysis, program slicing, query techniques, source code visualization, object recovery, restructuring, refactoring, and remodularization into a single powerful reengineering workbench [105].

IDA Pro Disassembler and Debugger by Hex-Rays [106] is a powerful disassembler that supports more than 50 different processor architectures, including IA-32, IA-64 (Itanium), and AMD64. IDA also supports a variety of executable file formats such as PE (Portable Executable used in Windows), ELF (Executable and Linking Format, used in Linux), and even XBE, which is used on Microsoft's Xbox. IDA is a remarkably flexible product, providing highly detailed disassembly, along with a plethora of side features that assist with the reversing task. IDA is capable of producing a useful flowchart for a given function. In addition, it can produce call graphs that visually illustrate the flow of code of a loaded program. The graph can show internal subroutines and the links between every one of those subroutines. IDA also has several little features that make it very convenient to use, such as highlighting all instances of the currently selected operand. This makes it much easier to read disassembled listings and gain an understanding of how data flow within the code.

Hex-Rays Decompiler is a commercial decompiler plug-in for IDA Pro [107]. This plug-in adds a decompiler view to the other views available with the interactive disassembler. It converts executable programs into a human readable C-like pseudo code text. The output is not designed for recompilation; rather, it is used only for more rapid comprehension of what the function is doing. The output includes compound conditional operators (|| and &&), loops (for, while, and break), and function parameters and returns.

Imagix 4D [108] is useful in understanding legacy C, C++, and Java software. It enables users to quickly check software at multiple levels: high-level architecture, classes, and function dependencies. Users can explore their software's control structures, data usage, and inheritance. The tool is also useful in creating design documentation automatically.

IRAP (Input–Output Reengineering and Program Crafting) is a data reengineering tool developed by Spectra Research [109] that provides a semi-automated approach to recraft legacy software into an Intranet/Internet-enabled application without compromising program computational integrity. In particular, the input–output mechanism in a legacy software is migrated to the Intranet/Internet platform, and business rules are extracted from legacy software and reimplemented into an Intranet/Internet-enabled application. Currently, Spectra Research supports reengineering of file-base FORTRAN 77 and ANSI standard C programs.

JAD (JAva Decompiler) is a Java decompiler written in C++ [110]. JAD can be used for (i) recovering lost source code; (ii) exploring the sources of Java runtime libraries; and (iii) disassembling bytecode. Since Java files are compiled to bytecode, which is interpreted by a Virtual Machine, and not compiled to machine code like C programs, Java decompilers can decompile Java programs into compilable source code with near-perfect accuracy. JAD fails to decompile some nested loops and has difficulty in decompiling inline commands and inner functions.

ManSART [111] is a tool to recover the architecture of a given software system. Working primarily on source code, the tool semi-automatically constructs software structure by abstracting functionalities of the system.

McCabe IQ [112] is capable of predicting some key issues in maintaining large and complex business software applications: (i) locate error-prone sections of code and (ii) identify the risk of system failure. It also provides a rich graphical environment for analyzing the architectures of systems. In addition, it provides robust enterprise reporting, advanced reengineering capabilities, change analysis, and secure web-enabled test data collection. McCabe IQ is platform independent and supports a variety of languages such as C, C#, C++, Perl, PL1, COBOL, JAVA, and VB.NET.

PBS (Portable Bookshelf) is an implementation of the web-based concept called *Software Bookshelf* [113] for the presentation navigation of information representing large software systems. The *PBS Toolkit* [114] is a set of tools for the generation of a PBS Bookshelf, which captures, organizes, manages, and delivers comprehensive information about software systems. It provides an integrated suite of code analysis and visualization capabilities intended for software maintenance, reengineering, and migration. One accesses the Bookshelf via a set of web pages. A separate page is assigned to each subsystem of the target software, and the page hierarchy reflects the system decomposition. Because of the open nature of the structure of the Bookshelf, developers can directly access the source code and documentation by navigating the file system.

RE-Analyzer [115] is an automated, reverse engineering system providing a high level of integration with a computer-aided software engineering (CASE) tool developed at IBM. The RE-analyzer automatically reverse engineers source code into graphic and textual representations within a CASE tool supporting structured analysis methodology [116]. That is, it transforms source code into a set of data flow diagrams, state transition diagrams, and entity-relationship data models within the design database of a CASE tool. Since the resulting representations can be browsed and modified within the CASE tool environment, a broad range of software engineering activities is supported, including program understanding, reengineering, and redocumentation.

Reengineering Assistant (RA) aims to provide an interactive environment where software maintainers can reverse engineer source code into a higher abstraction level of representation. The earliest version of RA is called Maintenance Assistant (MA), which was first developed for program analysis purpose at the Centre for Software Maintenance, Durham University in 1989. By means of a unified reengineering framework, a rigorous approach to reverse engineering has been proposed by using a WSL [103]. Sound rules for creating abstractions have been developed within the framework, and a tool has been built for automation.

Rigi [117, 118] is a software tool for comprehending large software systems. Software comprehension is achieved by performing reverse engineering on the given system. The tool (i) extracts artifacts from the information space of the given system; (ii) organizes the artifacts to construct a model at a higher abstraction level; (iii) graphically presents the model; (iv) supports both automatic and user-defined clustering of source artifacts; and (v) supports hierarchically embedded views of different relations among the source artifacts. To understand Java code, a reverse engineering environment called Shimba [119] has been coupled with Rigi. Shimba can analyze the static and dynamic aspects of the system. A directed graph is used to show dependencies among static software artifacts extracted from Java byte code.

SEELA is a reverse engineering tool developed by Tuval Software Industries [120] to support the documentation and maintenance of structured source code. The tool performs top-down display of source code to enhance the readability of programs. Additional features available in the tool are a structure editor, a browser, and a source code documentation generator. It works with a variety of programming languages, including C, FORTRAN, Ada, and Cobol. The main motivation for designing SEELA was to bridge the gap between a system's design description and the source code.

4.10 SUMMARY

In this chapter, we introduced the concepts, processes, and techniques of software reengineering including the risks associated with it. In addition, four general software reengineering benefits were discussed: (i) improving maintainability; (ii) migrating to a newer technology; (iii) improving quality; and (iv) preparing for functional enhancement.

Next, we introduced the concept of software reengineering based on three basic principles: abstraction, refinement, and alteration. Then, we discussed a general model for software engineering proposed by Eric J. Byrne [5]. The model suggests that reengineering is a sequence of three major activities: reverse engineering, re-design, and forward engineering. The aforementioned sequence is founded in abstraction, alteration, and refinement. In addition, we discussed three reengineering strategies: rewrite, rework, and replace. These reengineering strategies specify the basic steps for reengineering methods. The rewrite strategy is founded on the principle of alteration; the rework strategy on the principles of abstraction, alteration, and refinement; and the replace strategy on the principles of abstraction and refinement. Different variations of the reengineering process based on three strategies—rewrite, rework, and replace—and four types of changes—re-think, re-design, re-specify, and re-code—were enumerated in a tabular form and critically analyzed.

We explained five basic reengineering approaches: big bang, incremental, partial, iterative, and evolutionary. Each approach advocates a path for reaching the same goal of producing the target system. A specific reengineering approach is chosen for a given project after careful considerations of the following key factors: (i) objectives of the project; (ii) resources available for the project; (iii) the current state of the system to be reengineered; and (iv) risks in executing the project. We described two reengineering models: (i) source code reengineering reference model (SCORE/RM) and (ii) phase reengineering model. On the one hand, the reference model is divided into eight layers providing a detailed approach to rationalizing the software, understanding its requirements and functions, and reconstructing the software system following established practices. On the other hand, the phase reengineering model comprises five stages: planning and analysis, renovation, target system testing, redocumentation, and acceptance testing and system transition. Tasks within the phases are further decomposed to express detailed methodologies.

With the reengineering approaches and models in place, we introduced the concepts and objectives of *reverse engineering*. Then we introduced the Goals/Models/Tools paradigm that divides a reverse engineering process into three ordered stages: Goals, Models, and Tools:

Goals. In the *Goals* stage, the motivations for setting up of a process for reverse engineering are analyzed. The goals of the analysis activity are to identify (i) the information that the process will need and (ii) the abstractions to be created by the process.

Models. In this stage, the abstractions identified in the first stage are analyzed to create representation models. Representation models include information needed for the generation of abstractions.

Tools. In this phase, one defines, acquires, and/or develops the software tools required to produce documents by means of reverse engineering. The tools are used in (i) executing the *Models* stage and (ii) transforming the program models into the abstraction models identified in the *Goals* stage.

Fact-finding and information gathering from source code are the keys to the Goal/Models/Tools paradigm. In order to extract information that is not explicitly available from source code, automated analysis techniques, such as *lexical analysis, syntactic analysis, control flow analysis, data flow analysis, program slicing, visualization*, and *metrics* are used to facilitate reverse engineering. We examined low-level reverse engineering such as decompilers and disassemblers. The relationship between decompilation and the traditional reengineering model was also discussed.

Next, we discussed DRE for *data-oriented applications*. In *data-oriented applications*, the complexity can be broken down by considering that the files or database can be reverse engineered almost independently of the procedural parts. DRE deals only with data components of the *data-oriented applications*. DRE is a process for understanding and redocumenting the files and/or database of an application. We discussed a generic methodology based on the work of Jean-Luc Hainaut and his colleagues.

Finally, we concluded this chapter with a description of several commercial and academic reverse engineering tools. During our discussion, we emphasized on the purpose of each tool and the functionality it supports.

LITERATURE REVIEW

The edited book by Robert S. Arnold entitled *Software Reengineering*, IEEE Computer Society Press, Los Alamitos, CA, 1993 is a collection of 52 selected articles. These articles will further enhance the reader's understanding of software reengineering technology. A wide variety of software engineering concepts, tools and techniques, case studies, risks, and benefits are presented in the book. The book includes an excellent annotated bibliography of software reengineering that were published before circa 1993.

Reverse engineering is an active area of research in computer science. Researchers and practitioners, who are interested in knowing more about this subject, are recommended to study the following two articles:

H. Müller, J. Jahnke, D. Smith, M. Storey, S. Tilly and K. Wong. 2000. *Reverse Engineering: A Roadmap*, International Conference on Software Engineering – Future of Software Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 47–60.

G. Canfora and M. Di Penta. 2007. *New Frontiers of Reverse Engineering*, International Conference on Software Engineering – Future of Software Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 326–341.

The first article above summarizes the reverse engineering research during the 1980s and the 1990s. The authors describe the research activities in data and code reverse engineering. In addition, they discuss research results for developing and evaluating tools for reverse engineering. The second article presents the main achievements in the area of reverse engineering, such as program analysis, architecture and design recovery, and visualization. In addition, the authors discuss future research directions in reverse engineering.

Program transformation has applications in many areas of software engineering including compilation, optimization, refactoring, software renovation, and reverse engineering. Program transformation is the act of changing one program into an equivalent program. The aim of program transformation is to increase programmer productivity by automating programming tasks, thus enabling programming at a higher level of abstraction, and increasing maintainability and re-usability. The article by Eelco Visser [12] gives a taxonomy of the application areas of program transformations, discusses considerations to be made in the implementation of program transformation systems, especially focusing on the specification of transformation strategies. A slightly different approach to program translation was proposed by Richard Waters, known as *program translation via abstraction and reimplementation* [53]. It is a two-step method: (i) the source program is analyzed in order to produce a high-level, language-independent description and (ii) the reimplementation process transforms the abstract description obtained in the first step into a program in the target language.

We discussed the Goals/Models/Tools paradigm proposed by Benedusi et al. [21] when setting up a reverse engineering process. In circa 2004, Spencer Rugaber and Kurt Stirewalt (Model-driven reverse engineering, *IEEE Software*, July/August, 2004, 45–52) proposed a model-driven reverse engineering (MDRE) process. MDRE uses a formal specification language called SLANG and automatic code generation to reverse the reverse engineering process. To illustrate the process, the authors use a numerical analysis application of finding the roots of nonlinear equations.

Clone detection in software systems is an active area of research within program analysis. Several interesting techniques have been developed besides metric-based techniques [42], such as AST-based and token-based techniques. The reader is recommended to study the survey article on this subject by Chanchal Kumar Roy and James R. Cordy (*A Survey on Software Clone Detection Research*, Technical Report No. 2007-541, School of Computing, Queen's University at Kingston, Ontario, Canada). Moreover, contrary to the common wisdom, C. Kasper and M. W. Godfrey (*'Cloning Considered Harmful' Considered harmful*, Proceedings of the 2006 Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, 2006. pp. 9–18) argue that clones are not necessarily harmful, especially so if maintainers are aware of clones.

Those interested in knowing more about low-level reverse engineering may refer to the excellent book by Eldad Eilam entitled *Reversing: Secrets of Reverse Engineering* [59]. The author discussed several interesting topics related to reverse engineering such as (i) reverse engineering on the .NET platform, (ii) how to reverse engineer an operating system, (iii) how to decipher an undocumented file format, (iv) copy protect and digital rights management technologies, and (iv) theory and principle behind decompilers. Regarding the history of decompilers, the readers are recommended to visit the decompilation wiki page (http://www.program-transformation.org/Transform/DeCompilation).

Due to the use of persistent data structures in a large number of software systems, concepts and techniques in DRE have gained much attention. DRE evolves through two communities: database community and software engineering community [93]. DRE comprises two major activities, namely, data analysis and conceptual abstraction. The former tries to recover a structurally complete and semantically annotated logical data model, whereas the latter aims to map the logical data model to an equivalent conceptual design. The interested readers are recommended to study the articles by Jean-Luc Hainaut and his colleagues [40, 87, 88, 95]. In addition, the book by Jörg P. Wadsack entitled *Data-oriented Reengineering*, Sudwestdeutcher Verlag fur Hochschulschriften(SVH) AG, 2009 will further enhance the reader's understanding of data reengineering technology. The book presents a wide variety of reengineering approaches, concepts, tools and techniques, risks, and benefits. It discusses the processes that combine tools for reengineering the data as well as the applications.

The question of the extent to which low-level reverse engineering (decompilation or disassembly) can legitimately be carried out without the consent of the owner of copyright of the computer program has been the subject of debate and uncertainty for many years. The issue arises because any form of reverse engineering involves intermediate copying of the software being analyzed. Copyright protects the form of expression of ideas or information (but not the ideas or information in itself), conferring certain exclusive rights on the author or creator. A more contentious issue is whether exceptions to the copyright owner's exclusive rights should be recognized in order to permit copying which occurs in the course of reverse engineering of computer software. The article by Cristina Cifuentes and Anne Fitzgerald (The legal status of reverse engineering of computer software, *Annals of Software Engineering*, Vol. 9, 2000, pp. 337–351) summarize the state of the law in the United States, the European Union, and Australia in relation to exceptions to copyright infringement which permit reverse engineering. The authors also discussed the landmark cases of *Sega Enterprises Ltd versus Accolade, Inc.* (*Sega*) and *Atari Games Corp. versus Nintendo of America, Inc.* (*Atari*).

REFERENCES

[1] J. A. McCall, P. K. Richards, and G. F. Walters. 1977. *Factors in Software Quality*. Vol. 1, ADA 049014, National Technical Information Service, Springfield, VA.

[2] D. Yu. 1991. A view on three r's (3rs): reuse, re-engineering, and reverse-engineering. *ACM SIGSOFT Software Engineering Notes*, 16(3), 69.

[3] R. S. Arnold. 1993. A road map guide to software reengineering technology In: *Software Reengineering* (Ed R. S. Arnold). IEEE Computer Society Press, Los Alamitos, CA.

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

[5] E. J. Byrne. 1992. *A Conceptual Foundation for Software Reengineering*. Proceedings of the International Conference on Software Maintenance, November 1992, Orlando, Florida. IEEE Computer Society Press, Los Alamitos, CA. pp. 226–235.

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

[7] R. Kazman, S. Woods, and J. Carrière. 1998. *Requirements for Integrating Software Architecture and Reengineering Models: Corum ii*. Proceedings of Working Conference on Reverse Engineering (WCRE), Washington, DC. IEEE Computer Society Press, Los Alamitos, CA. pp. 154–163.

[8] I. Jacobson and F. Lindström. 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.

[9] J. Shao and C. Pound. 1999. Extracting business rules from information systems. *BT Technology Journal*, 17(4), 179–186.

[10] D. C. Schmidt. 2006. Model-driven engineering. *IEEE Computer*, February, 25–31.

[11] J. Manzella and B. Mutafelija. 1992. *Concept of the Re-engineering Life-cycle*. Proceedings of the International Conference on System Integration, June 1992, Morristown, NJ. IEEE Computer Society Press, Los Alamitos, CA. pp. 566–570.

[12] E. Visser. 2005. A survey of strategies in rule-based program transformation. *Journal of Symbolic Computation*, 40(1), 831–873.

[13] L. H. Rosenberg and L. E. Hyatt. 1996. Software re-engineering. *SAYC-TR-95-1001*. Available at http://satc.gsfc.nasa.gov/support/reengrpt.PDF (accessed October 1996).

[14] E. J. Byrne and D. A. Gustafson. 1992. *A Software Re-engineering Process Model*. Proceedings of the Sixteenth Annual International Conference on Computer Software and Applications (COMPSAC), September 1992. IEEE Computer Society Press, Los Alamitos, CA. pp. 25–30.

[15] A. Colbrook, C. Smythe, and A. Darlison. 1990. *Data Abstraction in a Software Re-engineering Reference Model*. Proceedings of the International Conference on Software Maintenance, November 1990, San Diego, CA. IEEE Computer Society Press, Los Alamitos, CA. pp. 2–11.

[16] H. M. Sneed. 1991. Economics of software re-engineering. *Journal of Software Maintenance: Research and Practice*, September, 163–182.

[17] M. G. Rekoff Jr. 1985. On reverse engineering. *IEEE Transactions on Systems, Man, and Cybernetics*, March–April, 244–252.

[18] C. Cifuentes. 2001. Reverse engineering and the computing profession. *IEEE Computer*, December, 166–168.

[19] T. J. Biggerstaff. 1989. Design recovery for maintenance and reuse. *Computer*, July, 36–49.

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

[21] 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.

[22] G. Canfora, A. Cimitile, and M. Munro. 1994. Reverse engineering and reuse engineering. *Journal of Software Maintenance and Evolution: Research and Practice*, 6(2), 53–72.

[23] B. Cheng and J. Jeng. 1997. Reusing analogous components. *IEEE Transactions on Knowledge and Data Engineering*, 9(2), 341–349.

[24] E. Burd and M. Munro. 1998. A method for the identification of reusable units through reengineering. *The Journal of Systems and Software*, 44(2), 121–134.

[25] G. Canfora, A. De Lucia, and M. Munro. An integrated environment for reuse reengineering C code. *The Journal of Systems and Software*, 42(2), 153–164.

[26] P. T. Breuer and K. Lano. 1991. Creating specification from code: reverse engineering techniques. *Software Maintenance: Research and Practice*, 3, 145–162.

[27] A. Lakhotia. 1997. A unified framework for expressing software subsystem classification technique. *The Journal od Systems and Software*, 36(3), 211–231.

[28] L. J. Holtzblatt, R. L. Piazza, H. B. Reubenstein, S. N. Roberts, and D. R. Harris. 1997. Design recovery for distributed systems. *IEEE Transactions on Software Engineering*, 23(7), 461–472.

[29] D. Jerding and S. Rugaber. 1997. *Using Visualization for Architectural Localization and Extraction*. 4th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 56–65.

[30] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. 2006. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7), 454–466.

[31] G. Antoniol, G. Casazza, M. Di Penta, and R Fiutem. 2001. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2), 181–196.

[32] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. 2006. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11), 896–909.

[33] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. 2002. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10), 970–983.

[34] G. Ebner and H. Kaindl. 2002. Tracing all around in reengineering. *IEEE Software*, May/June, pp. 70–77.

[35] A. Marcus and J. I. Maletic. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon. IEEE Computer Society Press, Los Alamitos, CA. pp. 125–135.

[36] G. Canfora, A. Cimitile, and M. Munro. 1996. An improved algorithm for identifying objects in code. *Software – Practice and Experience*, 26(1), 25–48.

[37] W. J. Premerlani and M. R. Blaha. 1994. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5), 42–49.

[38] M. R. Blaha. 1997. *Dimension of Database Reverse Engineering*. 4th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 176–183.

[39] J. H. Jahnke and J. Wadsack. 1999. *Integration of Analysis and Redesign Activities in Information System Reengineering*. Proceedings of the 3rd European Conference on Software Maintenance and Reengineering, October 1999. IEEE Computer Society Press, Los Alamitos, CA. pp. 160–168.

[40] J. Hainaut, J. Henrard, D. Roland, V. Englebert, and J. Hick. 1996. *Structure Elicitation in Database Reverse Engineering*. 3th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 131–139.

[41] B. S. Baker. 1995. *On Finding Duplication and Near-duplication in Large Software Systems*. Proceedings of the Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 86–95.

[42] K. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein. 1996. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3, 77–108.

[43] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7), 654–670.

[44] R. Koschke. 2008. Identifying and removing software clones. In: *Software Evolution* (Eds T. Mens and S. Demeyer), pp. 15–36. Springer-Verlag, Berlin.

[45] E. van Emden and L. Moonen. 2002. *Java Quality Assurance by Detecting code Smells*. 9th Working Conference on Reverse Engineering, Richmond, VA. IEEE Computer Society Press, Los Alamitos, CA. pp. 97–107.

[46] M. Marin, A. van Deursen, and L. Moonen. 2004. *Identifying Aspects Using Fan-in Analysis*. Proceedings of the 11th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 132–141.

[47] R. S. Arnold and S. A. Bohner. 1993. Impact analysis—towards a framework for comparison. In: *Software Chnage Impact Analysis* (Eds S. A. Bohner and R. S. Arnold), pp. 34–43. IEEE Computer Society Press, Los Alamitos, CA.

[48] C. Cifuentes and K. J. Gough. 1995. Decompilation of binary programs. *Software – Practice and Experience*, 25(7), 811–829.

[49] E. Merlo, Y. Gagné P, J. F. Girard, K. Kontogiannis, L. J. Hendren, P. Panangaden, and R. de Mori. 1995. Reengineering user interfaces. *IEEE Software*, January, 64–73.

[50] C. Plaisant, A. Rose, B. Shneiderman, and A. Vanniamparampil. 1997. Low-effort, high-payoff user interface reengineering. *IEEE Software*, July–August, 66–72.

[51] H. M. Sneed and S. H. Sneed. 2006. Reverse engineering of system interfaces: a report from the field. 13th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 125–133.

[52] S. Bhansali, J. R. Hegemeister, C. S. Raghavendra, and H. Sivaraman. 1994. Parallelizing sequential programs by algorithm-level transformation. Proceedings of the 3rd Workshop on Program Comprehension, Washington, DC. IEEE Computer Society Press, Los Alamitos, CA. pp. 100–107.

[53] R. C. Waters. 1988. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8), 1207–1228.

[54] E. J. Byrne. 1991. Software reverse engineering: a case study. *Software – Practice and Experience*, 21(2), 1349–1364.

[55] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. 2000. Decomposing legacy programs: a first step towards migrating to client-server platforms. *Journal of Systems and Software*, 54(2), 99–110.

[56] I. Jacobson, M. Ericsson, and A. Jacobson. 1995. *The Object Advantage – Business Process Re-engineering with Object Technology*. Addison-Wesley, Reading, MA.

[57] H. Huang, W. T. Tsai, S. Bhattacharya, X. Chen, Y. Wang, and J. Sun. 1998. Business rule extraction techniques for cobol programs. *Journal of Software Maintenance Research and Practice*, 10(1), 3–35.

[58] H. M. Sneed. 2000. Encapsulation of legal software: a technique for reusing legacy software components. *Annals Software Engineering*, 9, 293–313.

[59] E. Eilam. 2005. *Reversing: Secrets of Reverse Engineering*. Wiley, Indianapolis, IN.

[60] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. 2003. *Characterizing the 'Security Vulnerability Likelihood' of Software Function*. Proceedings, Conference on Software Maintenance. IEEE Computer Society Press, Los Alamitos, CA. pp. 266–275.

[61] M. Shevertalov and S. Mancoridis. 2007. *A Reverse Engineering Tool for Extracting Protocols of Networked Applications*. 14th Working Conference on Reverse Engineering, Vancouver, BC. IEEE Computer Society Press, Los Alamitos, CA. pp. 229–238.

[62] P. Benedusi, A. Cimitile, and U. De Carlini. 1989. *A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams*. Proceedings of the International Conference on Software Maintenance, October 1989, Miami, FL. IEEE Computer Society Press, Los Alamitos, CA. pp. 180–189.

[63] D. Brown, J. Levine, and T. Mason. 1995. *lex & yacc*, 2nd Edition. O'Reilly Media Inc., Sebastopol, CA.

[64] M. S. Hecht. 1977. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY.

[65] B. G. Ryder. 1979. Constructing the call graph of program. *IEEE Transactions on Software Engineering*, 5(3), 216–226.

[66] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. 1990. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4), 483–487.

[67] L. J. Osterweil and L. D. Fosdick. 1976. Dave – a validation, error detection, and documentation system for fortran programs. *Software – Practice and Experience*, October/December, 473–486.

[68] L. D. Fosdick and L. J. Osterweil. 1976. Data flow analysis in software reliability. *Computing Surveys*, September, 305–330.

[69] M. Weiser. 1984. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), 352–357.

[70] D. Binkley, S. Horwitz, and T. Reps. 1995. Program integration for language with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1), 3–35.

[71] K. Gallagher and D. Binkley. 2008. *Program Slicing*. Proceedings of the 2008 Frontiers of Software Maintenance (FoSM), September 2008, Beijing, China. IEEE Computer Society Press, Los Alamitos, CA. pp. 58–67.

[72] F. Tip. 1995. A survey of program slicing techniques. *Journal of Programming Languages*, 3, 121–189.

[73] P. Young and M. Munro. 1998. *Visualising Software in Virtual Reality*. Proceedings of International Workshop on Working Conference on Program Comprehensive. IEEE Computer Society Press, Los Alamitos, CA. pp. 19–26.

[74] T. J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, December, 308–320.

[75] A. J. Albrecht. 1979. *Measuring Application Development Productivity*. Process Joint SHARE/GUIDE/IBM Application Development Symposium, October 1979. Reprinted in 1981 in *Programming Productivity: Issues for the Eighties* (Ed. Capers Jones), No. EHO 186-7, pp. 34–43. Computer Society Press.

[76] S. M. Henry and D. G. Kafura. 1981. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5), 510–518.

[77] S. R. Chidamber and C. F. Kemerer. 1994. A metric suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6), 476–493.

[78] V. R. Basili, L. C. Briand, and W. L. Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10), 751–761.

[79] G. A. Di Lucca, M. Di Penta, A. R. Fasolino, and P. Granato. 2001. *Clone Analysis in the Web Era: An Approach to Identify Cloned Web Pages*. Proceedings of 7th IEEE Workshop on Empirical Studies of Software Maintenance, November 2001, Florence, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 107–113.

[80] G. A. Di Lucca, M. Di Penta, and A. R. Fasolino. 2002. *An Approach to Identify Duplicated Web Pages*. Proceedings of 26th International Computer Software and Applications Conference, August 2002, Oxford, England. IEEE Computer Society Press, Los Alamitos, CA. pp. 481–486.

[81] M. Halstead. 1962. *Machine Independent Computer Programming*. Spartan Books, Washington, DC.

[82] I. D. Baxter and M. Mehlich. 1997. Reverse Engineering Is Reverse Forward Engineering. 4th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 104–113.

[83] M. J. Van Emmerik. 2007. Static single assignment for decompilation. Ph.D. Thesis, School of Information Technology and Electrical Engineering, The University Queensland, Australia. Available at http://www.program-transformation.org/Transform/DecompilationAndReverseEngineering (accessed June 17, 2014).

[84] P. Samuelson and S. Scotchmer. 2002. The law and economics of reverse engineering. *The Yale Law Journal*, 111, 1575–1663.

[85] D. N. Pruitt. 2006. Beyond fair use: the right to contract around copyright protection of reverse engineering in the software industry. *Journal of Intellectual Property, Chicago-Kent College of Law*, 6(1), 66–86.

[86] J. A. Ricketts, J. C. DelMonaco, and M. W. Weeks. 1989. *Data Reengineering for Application Systems*. Proceedings of the International Conference on Software Maintenance, November 1989. IEEE Computer Society Press, Los Alamitos, CA. pp. 174–179.

[87] J. Hainaut, M. Chandelon, C. Tonneau, and M. Joris. 1993. *Contribution to a Theory of Database Reverse Engineering*. Proceedings of Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 161–170.

[88] J. Hainaut, V. Englebert, J. Henrard, J. Hick, and D. Roland. 1995. *Requirements for Information System Reverse Engineering Support*. 2nd Working Conference on Reverse EngineeringToronto, ON. IEEE Computer Society Press, Los Alamitos, CA. pp. 136–145.

[89] J. Fong. 1997. Converting relational to object-oriented databases. *ACM SIGMOD Record*, 26(1), 53–58.

[90] S. Ramanathan and J. Hodges. 1997. Extraction of object-oriented structures from existing relational databases. *ACM SIGMOD Record*, 26(1), 59–64.

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

[92] P. Aiken. 1998. Structured design. *IBM Systems Journal*, 37(2), 246–269.

[93] K. H. Davis and P. H. Aiken. 2000. *Data Reverse Engineering: A Historical Survey*. 7th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, pp. 70–78.

[94] M. Blaha. 1998. *On Reverse Engineering of Vendor Database*. 5th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, pp. 183–190.

[95] J. Hainaut, J. Hick, J. Henrard, D. Roland, and V. Englebert. 1997. *Knowledge Transfer in Database Reverse Engineering: A Supporting Case Study*. 4th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA. pp. 194–203.

[96] G. Baratta-Perez, R. L. Conn, C. A. Finnell, and T. J. Walsh. 1994. Ada system dependency analyzer tool. *IEEE Computer*, February, 49–55.

[97] L. Wills and C. Rich. 1990. Recognizing a program's design: a graph-parsing approach. *IEEE Software*, January, 82–89.

[98] M. Lanza. 2007. Available at http://smallwiki.unibe.ch/codecrawler/. *CodeCrawler* (accessed August 2007).

[99] M. Lanza and S. Ducasse. 2003. Polymetric views – a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, Septemebr, 782–995.

[100] Semantic Design Inc. 1995–2008. *The DMS Software Reengineering Toolkit*. Available at http://www.semdesigns.com/products/dms/dmstoolkit.htm (accessed October 1995–2008).

[101] I. Baxter, P. Pidgeon, and M. Mehlich. 2004. Dms: *Program Transformations for Practical Scalable Software Evolution*. Proceedings of the 26th International Conference on Software Engineering, Scotland, UK. IEEE Computer Society Press, Los Alamitos, CA. pp. 625–634.

[102] Software Migration Ltd. 2007. *The FermaT Workbench*. Available at http://www.smltd.com/solutions.htm (accessed May 2007).

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

[104] R. C. Holt, A. Schủrr, S. E. Sim, and A. Winter. 2006. Gxl: a graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2), 149–170.

[105] R. Holt, A. Schủrr, S. E. Sim, and A. Winter. 2002. *Graph Exchange Language Tools Version 1.0*. Available at http:/www.gupro.de/gxl/tools/tools.html (accessed July 2002).

[106] Hex-rays. 2007. *The IDA Pro Disassembler and Debugger*. Available at http://www.hex-rays.com/idapro (accessed February 2007).

[107] Hex-rays. 2008. *Hex-Rays Decompiler*. Available at http://www.hex-rays.com/decom piler.shtml. (accessed February 2008).

[108] Imagix corporation. 2007. *Reverse Engineering Quality Metrics, and Documentation Tool*. Available at http://www.imagix.com/products/products.html. (accessed October 2007).

[109] Spectra Research. 2008. *Software I/O Reengineering*. Available at http://www.spectra-research.com/inner/reeng.htm (accessed October 2008).

[110] P. Kouznetsov. 1997–2001. *Jad – The Fast JAva Decompiler*. Available at http://www.kpdus.com/jad.html (accessed June 17, 2014).

[111] M. P. Chase, S. M. Christey, D. R. Harris, and A. S. Yeh. 1998. *Managing Recovered Functions and Structure of Legacy Software Components*. 5th Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, pp. 79–88.

[112] McCabe Software. 2008. *McCabe IQ Enterprise Edition*. Available at http://www.mccabe.com/iq_enterprise.htm (accessed October 2008).

[113] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. 1997. The software bookself. *IBM Systems Journal*, 36(4), 564–593.

[114] R. Holt. 1997. *Software Bookself: Overview and Construction*. Pbs Bookself. Available at http://www.swag.uwaterloo.ca/pbs (accessed March, 1997).

[115] A. B. O'Hare and E. W. Troan. 1994. Re-analyzer: from source code to structured analysis. *IBM Systems Journal*, 33(1), 110–130.

[116] E. Yourdon. 1989. *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, NJ, pp. 417–423.

[117] H. A. Mủller. 1998. *Rigi User's Manual Version 5.4.4.* Available at http://www.rigi.csc.uvic.ca/pages/download.html.

[118] K. Wong, S. Tilley, H. A. Mủller, and M. D. Storey. 1995. Structural redocumentation: a case study. *IEEE Software*, January, pp. 46–54.

[119] T. Systå, K. Koskimies, and H. Mủller. 2001. Shimba—an environment for reverse engineering java software system. *Software Practice and Experience*, 31(4), 371–394.

[120] Tuval Software Industries. 2008. *SEELA: A Code Documentation Tool.* Available at http://www.speechover.com/index_files/about.htm (accessed October 2008).

EXERCISES

1. The potential reengineering risk areas are identified as follows: (i) process; (ii) reverse engineering; (iii) forward engineering; (iv) personal; (v) tools; and (vi) strategy. Explain these risk areas in detail.

2. Explain the principles of abstraction, refinement, and alteration in the context of software reengineering.

3. What are the differences among reengineering, reverse engineering, forward engineering, and rehosting?

4. Explain the differences between program translation and rephrasing. Discuss different types of program translation and rephrasing.

5. Discuss the differences between refactoring and renovation.

6. Explain the rework and replace reengineering strategies by giving concrete examples.

7. Discuss the pros and cons of different reengineering approaches.

8. Discuss the purpose of reverse engineering.

9. Explain the Goals/Models/Tools paradigm of software reverse engineering.

10. Explain the concepts of representation and visualization. Can a graphical object be both a representation within one context and a visualization within another? Explain it with an example.

11. Draw a CFG for the following sample code. Determine the cyclomatic complexity of the graph.

```
(a) sum_of_all_positive_numbers(a, num_of_entries, sum)
(b) sum = 0
(c) init = 1
(d)     while(init <= num_of_entries)
(e)         if a[init] > 0
```

```
(f)          sum = sum + a[init]
          endif
(g)       init = init + 1
        endwhile
(h)  end sum_of_all_positive_numbers
```

12. Obtain slices separately on *nw*, *nc*, *nl*, *inword*, and *c* of the following word count program.

```
[1]   #define  YES  1
[2]   #define  NO   0
[3]   main()
[4]   {
[5]      int c, nl, nw, nc, inword;
[6]      inword = NO;
[7]      nl = 0;
[8]      nw = 0;
[9]      nc = 0;
[10]     c = getchar();
[11]     while (c! = EOF) {
[12]         nc = nc + 1;
[13]         if (c == '\n')
[14]            nl = nl + 1;
[15]         if (c == ' '||c == '\n')||c == '\t')
[16]            inword = NO;
[17]         else if (inword == NO) {
[18]            inword = YES;
[19]            nw = nw + 1;
[20]         }
[21]         c = getchar();
[22]     }
[23]     printf("%d\n", nl);
[24]     printf("%d\n", nw);
[25]     printf("%d\n", nc);
[26]  }
```

13. Explain plan recognition.

14. Explain why decompilation is a reverse engineering process, whereas compilation is not a forward engineering process. Discuss the applications of decompilers. What technique can be used to avoid decompilation?

15. Discuss the purpose of DRE.