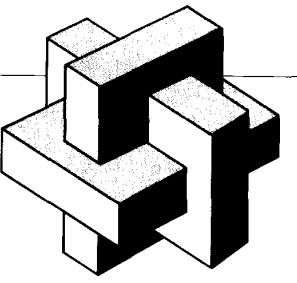

Reverse Engineering and Design Recovery: A Taxonomy

Elliot J. Chikofsky, *Index Technology Corp. and Northeastern University*
James H. Cross II, *Auburn University*

Reverse engineering is evolving as a major link in the software life cycle, but its growth is hampered by confusion over terminology. This article defines key terms.

The availability of computer-aided systems-engineering environments has redefined how many organizations approach system development. To meet their true potential, CASE environments are being applied to the problems of maintaining and enhancing existing systems. The key lies in applying reverse-engineering approaches to software systems. However, an impediment to success is the considerable confusion over the terminology used in both technical and marketplace discussions.

It is in the reverse-engineering arena, where the software maintenance and development communities meet, that various terms for technologies to analyze and understand existing systems have been frequently misused or applied in conflicting ways.

In this article, we define and relate six terms: forward engineering, reverse engineering, redocumentation, design recovery, restructuring, and reengineering. Our objective is not to create new terms but to rationalize the terms already in use. The resulting definitions apply to the underlying engineering processes, regardless of the degree of automation applied.

Hardware origins

The term "reverse engineering" has its origin in the analysis of hardware — where the practice of deciphering designs from finished products is commonplace. Reverse engineering is regularly applied to improve your own products, as well as to analyze a competitor's products or those of an adversary in a military or national-security situation.

In a landmark paper on the topic, M.G. Rekoff defines reverse engineering as "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system."[1] He describes such a process

0740-7459/90/0100/0013/$01.00 © 1990 IEEE

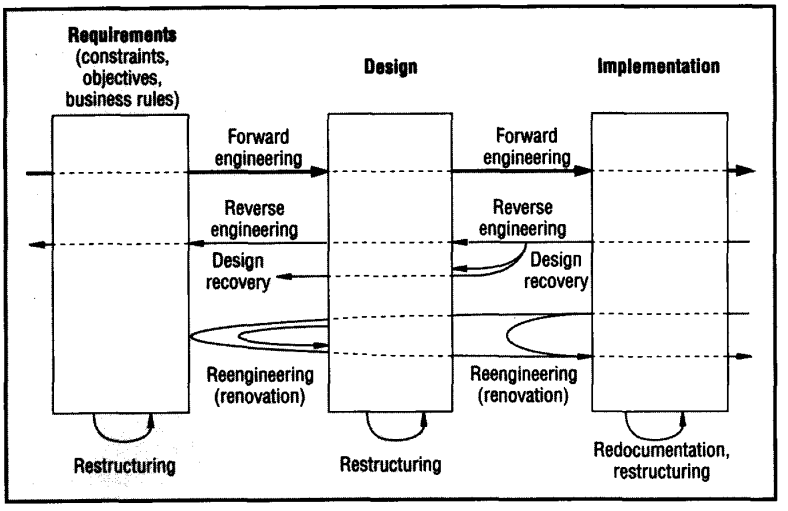

Figure 1. Relationship between terms. Reverse engineering and related processes are transformations between or within abstraction levels, represented here in terms of life-cycle phases.

as being conducted by someone other than the developer, "without the benefit of any of the original drawings ... for the purpose of making a clone of the original hardware system...."

In applying these concepts to software systems, we find that many of these approaches apply to gaining a basic understanding of a system and its structure. However, while the hardware objective traditionally is to duplicate the system, the software objective is most often to gain a sufficient design-level understanding to aid maintenance, strengthen enhancement, or support replacement.

Software maintenance

The ANSI definition of software maintenance is the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment," according to ANSI/IEEE Std 729-1983.

Usually, the system's maintainers were not its designers, so they must expend many resources to examine and learn about the system. Reverse-engineering tools can facilitate this practice. In this context, reverse engineering is the part of the maintenance process that helps you understand the system so you can make appropriate changes. Restructuring and reverse engineering also fall within the global definition of software maintenance. However, each of these three processes also has a place within the contexts of building new systems and evolutionary development.

Life cycles and abstractions

To adequately describe the notion of software forward and reverse engineering, we must first clarify three dependent concepts: the existence of a life-cycle model, the presence of a subject system, and the identification of abstraction levels.

We assume that an orderly life-cycle model exists for the software-development process. The model may be represented as the traditional waterfall, as a spiral, or in some other form that generally can be represented as a directed graph. While we expect there to be iteration within stages of the life cycle, and perhaps even recursion, its general directed-graph nature lets us sensibly define forward (downward) and backward (upward) activities.

The subject system may be a single program or code fragment, or it may be a complex set of interacting programs, job-control instructions, signal interfaces, and data files. In forward engineering, the subject system is the result of the development process. It may not yet exist, or its existing components may not yet be united to form a system. In reverse engineering, the subject system is generally the starting point of the exercise.

In a life-cycle model, the early stages deal with more general, implementation-independent concepts; later stages emphasize implementation details. The transition of increasing detail through the forward progress of the life cycle maps well to the concept of abstraction levels. Earlier stages of systems planning and requirements definition involve expressing higher level abstractions of the system being designed when compared to the implementation itself.

These abstractions are more closely related to the business rules of the enterprise. They are often expressed in user terminology that has a one-to-many relationship to specific features of the finished system. In the same sense, a blueprint is a higher level abstraction of the building it represents, and it may document only one of the many models (electrical, water, heating/ventilation/air conditioning, and egress) that must come together.

It is important to distinguish between *levels* of abstraction, a concept that crosses conceptual stages of design, and *degrees* of abstraction within a single stage. Spanning life-cycle phases involves a transition from higher abstraction levels in early stages to lower abstraction levels in later stages. While you can represent information in any life-cycle stage in detailed form (lower degree of abstraction) or in more summarized or global forms (higher degree of abstraction), these definitions emphasize the concept of *levels* of abstraction between life-cycle phases.

Definitions

For simplicity, we describe key terms using only three identified life-cycle stages with clearly different abstraction levels, as Figure 1 shows:

- requirements (specification of the problem being solved, including objectives, constraints, and business rules),
- design (specification of the solution), and
- implementation (coding, testing, and delivery of the operational system).

Forward engineering. Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

While it may seem unnecessary — in view of the long-standing use of design and development terminology — to introduce a new term, the adjective "forward"

has come to be used where it is necessary to distinguish this process from reverse engineering. Forward engineering follows a sequence of going from requirements through designing its implementation.

Reverse engineering. Reverse engineering is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.

Reverse engineering generally involves extracting design artifacts and building or synthesizing abstractions that are less implementation-dependent. While reverse engineering often involves an existing functional system as its subject, this is *not* a requirement. You can perform reverse engineering starting from any level of abstraction or at any stage of the life cycle.

Reverse engineering in and of itself does *not* involve changing the subject system or creating a new system based on the reverse-engineered subject system. It is a process of *examination*, not a process of change or replication.

In spanning the life-cycle stages, reverse engineering covers a broad range starting from the existing implementation, recapturing or recreating the design, and deciphering the requirements actually implemented by the subject system.

There are many subareas of reverse engineering. Two subareas that are widely referred to are redocumentation and design recovery.

Redocumentation. Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views (for example, dataflow, data structure, and control flow) intended for a human audience.

Redocumentation is the simplest and oldest form of reverse engineering, and many consider it to be an unintrusive, weak form of restructuring. The "re-" prefix implies that the intent is to recover documentation about the subject system that existed or should have existed.

Some common tools used to perform redocumentation are pretty printers (which display a code listing in an improved form), diagram generators (which create diagrams directly from code, reflecting control flow or code structure), and cross-reference listing generators. A key goal of these tools is to provide easier ways to visualize relationships among program components so you can recognize and follow paths clearly.

Design recovery. Design recovery is a subset of reverse engineering in which 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.

Reverse engineering in and of itself does not involve changing the subject system. It is a process of examination, not change or replication.

Design recovery is distinguished by the sources and span of information it should handle. According to Ted Biggerstaff: "Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains ... Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software-engineering representations or code."[2]

Restructuring. Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics).

A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. The term "restructuring" came into popular use from the code-to-code transform that recasts a program from an unstructured ("spaghetti") form to a structured (goto-less) form. However, the term has a broader meaning that recognizes the application of similar transformations and recasting techniques in reshaping data models, design plans, and requirements structures. Data normalization, for example, is a data-to-data restructuring transform to improve a logical data model in the database design process.

Many types of restructuring can be performed with a knowledge of structural form but without an understanding of meaning. For example, you can convert a set of If statements into a Case structure, or vice versa, without knowing the program's purpose or anything about its problem domain.

While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system. Restructuring is often used as a form of preventive maintenance to improve the physical state of the subject system with respect to some preferred standard. It may also involve adjusting the subject system to meet new environmental constraints that do not involve reassessment at higher abstraction levels.

Reengineering. Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring. This may include modifications with respect to new requirements not met by the original system. For exam-

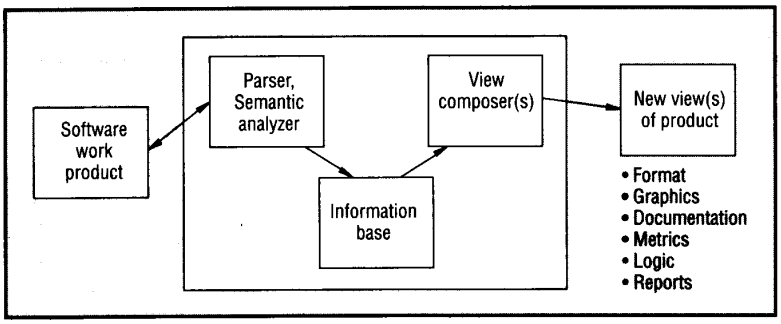

Figure 2. Model of tools architecture. Most tools for reverse engineering, restructuring, and reengineering use the same basic architecture. The new views on the right may themselves be software work products, which are shown on the left. (Model provided by Robert Arnold of the Software Productivity Consortium.)

ple, during the reengineering of information-management systems, an organization generally reassesses how the system implements high-level business rules and makes modifications to conform to changes in the business for the future.

There is some confusion of terms, particularly between reengineering and restructuring. The IBM user group Guide, for example, defines "application reengineering" as "the process of modifying the internal mechanisms of a system or program or the data structures of a system without changing the functionality (system capabilities as perceived by the user). In other words, it is altering the *how* without affecting the *what*."[3] This is closest to our definition of restructuring. However, two paragraphs later, the same publication says, "It is rare that an application is reengineered without additional functionality being added." This supports our more general definition of reengineering.

While reengineering involves both forward engineering and reverse engineering, it is *not* a supertype of the two. Reengineering uses the forward- and reverse-engineering technologies available, but to date it has not been the principal driver of their progress. Both technologies are evolving rapidly, independent of their application within reengineering.

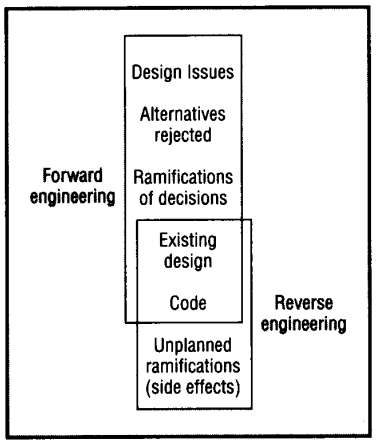

Figure 3. Differences between viewpoints. Although reverse engineering can help capture lost information, some types of information are not shared between forward- and reverse-engineering processes. However, reverse engineering can provide observations that are unobtainable in forward engineering.

Objectives

What are we trying to accomplish with reverse engineering? The primary purpose of reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development. Beyond the definitions above, there are six key objectives that will guide its direction as the technology matures:

- Cope with complexity. We must develop methods to better deal with the shear volume and complexity of systems. A key to controlling these attributes is automated support. Reverse-engineering methods and tools, combined with CASE environments, will provide a way to extract relevant information so decision makers can control the process and the product in systems evolution. Figure 2 shows a model of the structure of most tools for reverse engineering, reengineering, and restructuring.
- Generate alternate views. Graphical representations have long been accepted as comprehension aids. However, creating and maintaining them continues to be a bottleneck in the process. Reverse-engineering tools facilitate the generation or regeneration of graphical representations from other forms. While many designers work from a single, primary perspective (like dataflow diagrams), reverse-engineering tools can generate additional views from other perspectives (like control-flow diagrams, structure charts, and entity-relationship diagrams) to aid the review and verification process. You can also create alternate forms of nongraphical representations with reverse-engineering tools to form an important part of system documentation.
- Recover lost information. The continuing evolution of large, long-lived systems leads to lost information about the system design. Modifications are frequently not reflected in documentation, particularly at a higher level than the code itself. While it is no substitute for preserving design history in the first place, reverse engineering — particularly design recovery — is our way to salvage whatever we can from the existing systems. It lets us get a handle on systems when we don't understand what they do or how their individual programs interact as a system.
- Detect side effects. Both haphazard initial design and successive modifications can lead to unintended ramifications and side effects that impede a system's performance in subtle ways. As Figure 3 shows, reverse engineering can provide observations beyond those we can obtain with a forward-engineering perspective, and it can help detect anomalies and problems before users report them as bugs.
- Synthesize higher abstractions. Reverse engineering requires methods and techniques for creating alternate views that transcend to higher abstraction levels. There is debate in the software community as to how completely the process can be automated. Clearly, expert-system technology will play a major role in achieving the full potential of generating high-level abstractions.
- Facilitate reuse. A significant issue in the movement toward software reusability is the large body of existing software assets. Reverse engineering can help detect candidates for reusable software components from present systems.

Economics

The cost of understanding software, while rarely seen as a direct cost, is nonetheless very real. It is manifested in the time required to comprehend software, which includes the time lost to misunderstanding. By reducing the time required to grasp the essence of software artifacts in each life-cycle phase, reverse engineering may greatly reduce the overall cost of software.

In commenting on this article, Walt Scacchi of the University of Southern California made the following important observations: "Many claim that conventional software maintenance practices account for 50 to 90 percent of total life-cycle costs. Software reverse-engineering technologies are targeted to the problems that give rise to such a disproportionate distribution of software costs. Thus, if reverse engineering succeeds, the total system expense may be reduced/mitigated, or greater value may be added to current efforts, both of which represent desirable outcomes, especially if one quantifies the level of dollars spent. Reverse engineering may need to only realize a small impact to generate sizable savings."

Scacchi also pointed out that "software forward engineering and reverse engineering are *not* separate concerns, and thus should be viewed as opportunity for convergence and complement, as well as an expansion of the repertoire of tools and techniques that should be available to the modern software engineer. I, for one, believe that the next generation of software-engineering technologies will be applicable in both the forward and reverse directions. Such a view also may therefore imply yet another channel for getting advanced software-environment/CASE technologies into more people's hands — sell them on reverse engineering (based on current software-maintenance cost patterns) as a way to then introduce better forward engineering tools and techniques."

We have tried to provide a framework for examining reverse-engineering technologies by synthesizing the basic definitions of related terms and identifying common objectives.

Reverse engineering is rapidly becoming a recognized and important component of future CASE environments. Because the entire life cycle is naturally an iterative activity, reverse-engineering tools can provide a major link in the overall process of development and maintenance. As these tools mature, they will be applied to artifacts in all phases of the life cycle. They will be a permanent part of the process, ultimately used to verify all completed systems against their intended designs, even with fully automated generation.

Reverse engineering, used with evolving software development technologies, will provide significant incremental enhancements to our productivity. ❖

Acknowledgments

We acknowledge the special contributions of these individuals to the synthesis of this taxonomy and the rationalization of conflicting terminology: Walt Scacchi of the University of Southern California, Norm Schneidewind of the Naval Postgraduate School, Jim Fulton of Boeing Computer Services, Bob Arnold of the Software Productivity Consortium, Shawn Bohner of Contel Technology Center, Philip Hausler and Mark Pleszkoch of IBM and the University of Maryland at Baltimore County, Linore Cleveland of IBM, Diane Mularz of Mitre, Paul Oman of University of Idaho, John Munson and Norman Wilde of the University of West Florida, and the participants in directed discussions at the 1989 Conference on Software Maintenance and the 1988 and 1989 International Workshops on CASE.

References

1. M.G. Rekoff Jr., "On Reverse Engineering," *IEEE Trans. Systems, Man, and Cybernetics*, March-April 1985, pp. 244-252.
2. T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *Computer*, July 1989, pp. 36-49.
3. "Application Reengineering," Guide Pub. GPP-208, Guide Int'l Corp., Chicago, 1989.

Elliot J. Chikofsky is director of research and technology at Index Technology Corp. and a lecturer in industrial engineering and information systems at Northeastern University.

Chikofsky is an associate editor-in-chief of *IEEE Software*, vice chairman for membership of the Computer Society's Technical Committee on Software Engineering, president of the International Workshop on CASE, and author of a book on CASE in the Technology Series for IEEE Computer Society Press. He is a senior member of the IEEE.

James H. Cross II is an assistant professor of computer science and engineering at Auburn University. His research interests include design methodology, development environments, reverse engineering, visualization, and testing. He is secretary of the IEEE Computer Society Publications Board.

Cross received a BS in mathematics from the University of Houston, an MS in mathematics from Sam Houston State University, and a PhD in computer science from Texas A&M University. He is a member of the ACM and IEEE Computer Society.

Address questions about this article to Chikofsky at Index Technology, 1 Main St., Cambridge, MA 02142 or to Cross at Computer Science and Engineering Dept., 107 Dunstan Hall, Auburn University, Auburn, AL 36849.