CHAPTER

36

MAINTENANCE AND REENGINEERING

Regardless of its application domain, its size, or its complexity, computer software will evolve over time. Change drives this process. For computer software, change occurs when errors are corrected, when the software is adapted to a new environment, when the customer requests new features or functions, and when the application is reengineered to provide benefit in a modern context. Over the past 40 years, Manny Lehman [e.g., Leh97a] and his colleagues have performed detailed analyses of industry-grade software and systems in an effort to develop a *unified theory for software evolution.* The

QUICK LOOK

What is it? Consider any technology product that has served you well. You use it regularly, but it's getting old. It breaks too often, takes longer to repair than you'd like, and no longer represents the newest technology. What to do? For a time, you try to fix it, patch it, even extend its functionality. That's called maintenance. But maintenance becomes increasingly difficult as the years pass. There comes a time when you'll need to rebuild it. You'll create a product with added functionality, better performance and reliability, and improved maintainability. That's what we call reengineering.

Who does it? At an organizational level, maintenance is performed by support staff that are part of the software engineering organization. Reengineering is performed by business specialists (often consulting companies), and at the software level, reengineering is performed by software engineers.

Why is it important? We live in a rapidly changing world. The demands on business functions and the information technology that supports them are changing at a pace that puts enormous competitive pressure on every commercial organization. That's why software must be maintained continually, and at the appropriate time, reengineered to keep pace.

What are the steps? Maintenance corrects defects, adapts the software to meet a changing environment, and enhances functionality to meet the evolving needs of customers. At a strategic level, business process reengineering (BPR) defines business goals, identifies and evaluates existing business processes, and creates revised business processes that better meet current goals. Software reengineering encompasses inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. The intent of these activities is to create versions of existing programs that exhibit higher quality and better maintainability.

What is the work product? A variety of maintenance and reengineering work products (e.g., use cases, analysis and design models, test procedures) are produced. The final output is upgraded software.

How do I ensure that I've done it right? Use the same SQA practices that are applied in every software engineering process—technical reviews assess the analysis and design models; specialized reviews consider business applicability and compatibility; and testing is applied to uncover errors in content, functionality, and interoperability.

details of this work are beyond the scope of this book, but the underlying laws that have been derived are worthy of note [Leh97b]:

The Law of Continuing Change (1974): Software that has been implemented in a real-world computing context and will therefore evolve over time (called *E-type systems*) must be continually adapted else they become progressively less satisfactory.

The Law of Increasing Complexity (1974): As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

The Law of Self Regulation (1974): The E-type system evolution process is self-regulating with distribution of product and process measures close to normal.

The Law of Conservation of Organizational Stability (1980): The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

The Law of Conservation of Familiarity (1980): As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

The Law of Continuing Growth (1980): The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

The Law of Declining Quality (1996): The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The Feedback System Law (1996): E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

? Why do legacy systems evolve as time passes?

The laws that Lehman and his colleagues have defined are an inherent part of a software engineer's reality. In this chapter, we'll discuss the challenge of software maintenance and the reengineering activities that are required to extend the effective life of legacy systems.

36.1 Software Maintenance

It begins almost immediately. Software is released to end users, and within days, bug reports filter back to the software engineering organization. Within weeks, one class of users indicates that the software must be changed so that it can accommodate the special needs of their environment. And within months, another corporate group that wanted nothing to do with the software when it was released now recognizes that it may provide unexpected benefit. They'll need a few enhancements to make it work in their world.

The challenge of software maintenance has begun. You're faced with a growing queue of bug fixes, adaptation requests, and outright enhancements that must be planned, scheduled, and ultimately accomplished. Before long, the queue has

grown long, and the work it implies threatens to overwhelm the available resources. As time passes, your organization finds that it's spending more money and time maintaining existing programs than it is engineering new applications. In fact, it's not unusual for a software organization to expend as much as 60 to 70 percent of all resources on software maintenance.

You may ask why so much maintenance is required and why so much effort is expended. Osborne and Chikofsky [Osb90] provide a partial answer:

> Much of the software we depend on today is on average 10 to 15 years old. Even when these programs were created using the best design and coding techniques known at the time [and most were not], they were created when program size and storage space were principle concerns. They were then migrated to new platforms, adjusted for changes in machine and operating system technology and enhanced to meet new user needs—all without enough regard to overall architecture. The result is the poorly designed structures, poor coding, poor logic, and poor documentation of the software systems we are now called on to keep running . . .

Another reason for the software maintenance problem is the mobility of software people. It is likely that the software team (or person) that did the original work is no longer around. Worse, other generations of software people have modified the system and moved on. And today, there may be no one left who has any direct knowledge of the legacy system.

As we noted in Chapter 29, the ubiquitous nature of change underlies all software work. Change is inevitable when computer-based systems are built; therefore, you must develop mechanisms for evaluating, controlling, and making modifications.

Quote:

"Program maintainability and program understandability are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain.

Gerald Berns

Throughout this book, we've emphasized the importance of understanding the problem (analysis) and developing a well-structured solution (design). In fact, Part 2 of the book is dedicated to the mechanics of these software engineering actions, and Part 3 focuses on the techniques required to be sure you've done them correctly. Both analysis and design lead to an important software characteristic that we call maintainability. In essence, *maintainability* is a qualitative indication[1] of the ease with which existing software can be corrected, adapted, or enhanced. Much of what software engineering is about is building systems that exhibit high maintainability.

But what is maintainability? Maintainable software exhibits effective modularity (Chapter 12). It makes use of design patterns (Chapter 16) that allow ease of understanding. It has been constructed using well-defined coding standards and conventions, leading to source code that is self-documenting and understandable. It has undergone a variety of quality assurance techniques (Part 3 of this book) that have uncovered potential maintenance problems before the software is released. It has been created by software engineers who recognize that they

1 There are many quantitative measures that provide an indirect indication of maintainability (e.g., [Sch99], [SEI02]).

may not be around when changes must be made. Therefore, the design and implementation of the software must "assist" the person who is making the change.

36.2 Software Supportability

In order to effectively support industry-grade software, your organization (or its designee) must be capable of making the corrections, adaptations, and enhancements that are part of the maintenance activity. But in addition, the organization must provide other important support activities that include ongoing operational support, end-user support, and reengineering activities over the complete life cycle of the software. A reasonable definition of *software supportability* is

> . . . the capability of supporting a software system over its whole product life. This implies satisfying any necessary needs or requirements, but also the provision of equipment, support infrastructure, additional software, facilities, manpower, or any other resource required to maintain the software operational and capable of satisfying its function. [ISO08]

In essence, supportability is one of many quality factors that should be considered during the analysis and design actions that are part of the software process. It should be addressed as part of the requirements model (or specification) and considered as the design evolves and construction commences.

WebRef

A wide array of downloadable documents on software supportability can be found at **www.software-supportability.org/Downloads.html**.

For example, the need to "antibug" software at the component and code level has been discussed previously in the book. The software should contain facilities to assist support personnel when a defect is encountered in the operational environment (and make no mistake, defects *will* be encountered). In addition, support personnel should have access to a database that contains records of all defects that have already been encountered—their characteristics, cause, and cure. This will enable support personnel to examine "similar" defects and may provide a means for more rapid diagnosis and correction.

Although defects encountered in an application are a critical support issue, supportability also demands that resources be provided to support day-to-day end-user issues. The job of end-user support personnel is to answer user queries about the installation, operation, and use of the application.

36.3 Reengineering

In a seminal article written for the *Harvard Business Review,* Michael Hammer [Ham90] laid the foundation for a revolution in management thinking about business processes and computing:

> It is time to stop paving the cow paths. Instead of embedding outdated processes in silicon and software, we should obliterate them and start over. We should "reengineer" our businesses: use the power of modern information technology to radically

redesign our business processes in order to achieve dramatic improvements in their performance.

Quote:

"To face tomorrow with the thought of using the methods of yesterday is to envision life at a standstill."

James Bell

The hype associated with reengineering waned, but the process itself continues in companies large and small. The nexus between business reengineering and software engineering lies in a "system view."

As managers work to modify business rules to achieve greater effectiveness and competitiveness, software must keep pace. In some cases, this means the creation of major new computer-based systems.[2] But in many others, it means the modification or rebuilding of existing applications.

In the sections that follow, we examine reengineering in a top-down manner, beginning with a brief overview of business process reengineering and proceeding to a more detailed discussion of the technical activities that occur when software is reengineered.

36.4 Business Process Reengineering

KEY POINT

BPR often results in new software functionality, whereas software reengineering works to replace existing software functionality with better, more maintainable software.

Business process reengineering (BPR) extends far beyond the scope of information technologies and software engineering. Among the many definitions (most somewhat abstract) that have been suggested for BPR is one published in *Fortune* magazine [Ste93]: "[T]he search for, and the implementation of, radical change in business process to achieve breakthrough results." But how is the search conducted, and how is the implementation achieved? More important, how can we ensure that the "radical change" suggested will in fact lead to "breakthrough results" instead of organizational chaos?

36.4.1 Business Processes

ADVICE

As a software engineer, your work occurs at the bottom of this hierarchy. Be sure, however, that someone has given serious thought to the level above. If this hasn't been done, your work is at risk.

A business process is "a set of logically related tasks performed to achieve a defined business outcome" [Dav90]. Within the business process, people, equipment, material resources, and business procedures are combined to produce a specified result. Examples of business processes include designing a new product, purchasing services and supplies, hiring a new employee, and paying suppliers. Each demands a set of tasks, and each draws on diverse resources within the business.

Every business process has a defined customer—a person or group that receives the outcome (e.g., an idea, a report, a design, a service, a product). In addition, business processes cross organizational boundaries. They require that different organizational groups participate in the "logically related tasks" that define the process.

2 The explosion of Web and mobile applications and systems is indicative of this trend.

Every system is actually a hierarchy of subsystems. A business is no exception. The overall business is segmented in the following manner:

The business → business systems → business processes → business subprocesses

Each business system (also called *business function*) is composed of one or more business processes, and each business process is defined by a set of subprocesses.

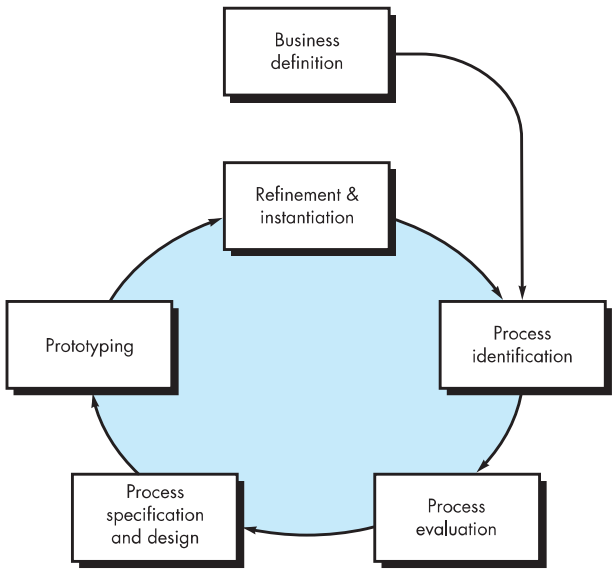
BPR can be applied at any level of the hierarchy, but as the scope of BPR broadens (i.e., as we move upward in the hierarchy), the risks associated with BPR grow dramatically. For this reason, most BPR efforts focus on individual processes or subprocesses.

36.4.2 A BPR Model

Like most engineering activities, business process reengineering is iterative. Business goals and the processes that achieve them must be adapted to a changing business environment. For this reason, there is no start and end to BPR—it is an evolutionary process. A model for business process reengineering is depicted in Figure 36.1. The model defines six activities:

1. **Business definition.** Business goals are identified within the context of four key drivers: cost reduction, time reduction, quality improvement, and personnel development and empowerment. Goals may be defined at the business level or for a specific component of the business.
2. **Process identification.** Processes that are critical to achieving the goals defined in the business definition are identified. They may then be ranked

FIGURE 36.1
A BPR model



by importance, by need for change, or in any other way that is appropriate for the reengineering activity.

Quote:

"As soon as we are shown something old in a new thing, we are pacified."

F. W. Nietzsche

3. **Process evaluation.** The existing process is thoroughly analyzed and measured. Process tasks are identified; the costs and time consumed by process tasks are noted; and quality/performance problems are isolated.
4. **Process specification and design.** Based on information obtained during the first three BPR activities, use cases (Chapters 8 and 9) are prepared for each process that is to be redesigned. Within the context of BPR, use cases identify a scenario that delivers some outcome to a customer. With the use case as the specification of the process, a new set of tasks are designed for the process.
5. **Prototyping.** A redesigned business process must be prototyped before it is fully integrated into the business. This activity "tests" the process so that refinements can be made.
6. **Refinement and instantiation.** Based on feedback from the prototype, the business process is refined and then instantiated within a business system.

These BPR activities are sometimes used in conjunction with workflow analysis tools. The intent of these tools is to build a model of existing workflow in an effort to better analyze existing processes.

SOFTWARE TOOLS

Business Process Reengineering (BPR)

Objective: The objective of BPR tools is to support the analysis and assessment of existing business processes and the specification and design of new ones.

Mechanics: Tools mechanics vary. In general, BPR tools allow a business analyst to model existing business processes in an effort to assess workflow inefficiencies or functional problems. Once existing problems are identified, tools allow the analysis to prototype and/or simulate revised business processes.

Representative Tools:[3]

ExtendSim, developed by ImagineThat (**www.imaginethatinc.com**), is a simulation tool for modeling existing processes and exploring new ones. Extend provides comprehensive what-if capability that enables a business analysis to explore different process scenarios.

Metastrom BPM, developed by OpenText (**http://bps.opentext.com/**), provides business process management support for both manual and automated processes.

IceTools, developed by Blue Ice (**http://www.icetools.com/home.html**), is a collection of BPR templates for Microsoft Office and Microsoft Visio.

OMNIBUS, developed by Kovair (**http://www.kovair.com/**), is one of many tools that enable an organization to model process workflow (in this case, IT workflow).

ProcessMaker, open-source workflow suite developed by Colosa (**http://www.processmaker.com/**), incorporates a suite of tools for workflow modeling, simulation, and scheduling.

A useful list of BPR tool links can be found at **www.opfro.org/index.html?Components/Producers/Tools/BusinessProcessReengineeringTools.html~Contents**.

3 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

36.5 Software Reengineering

The scenario is all too common: An application has served the business needs of a company for 10 or 15 years. During that time it has been corrected, adapted, and enhanced many times. People approached this work with the best intentions, but good software engineering practices were always shunted to the side (due to the urgency of other matters). Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. What to do?

Unmaintainable software is not a new problem. In fact, the broadening emphasis on software reengineering has been spawned by software maintenance problems that have been building for almost half a century.

36.5.1 A Software Reengineering Process Model

Reengineering takes time, it costs significant amounts of money, and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these reasons, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb information technology resources for many years. That's why every organization needs a pragmatic strategy for software reengineering.

WebRef

An excellent source of information on software reengineering can be found at **reengineer.org**.

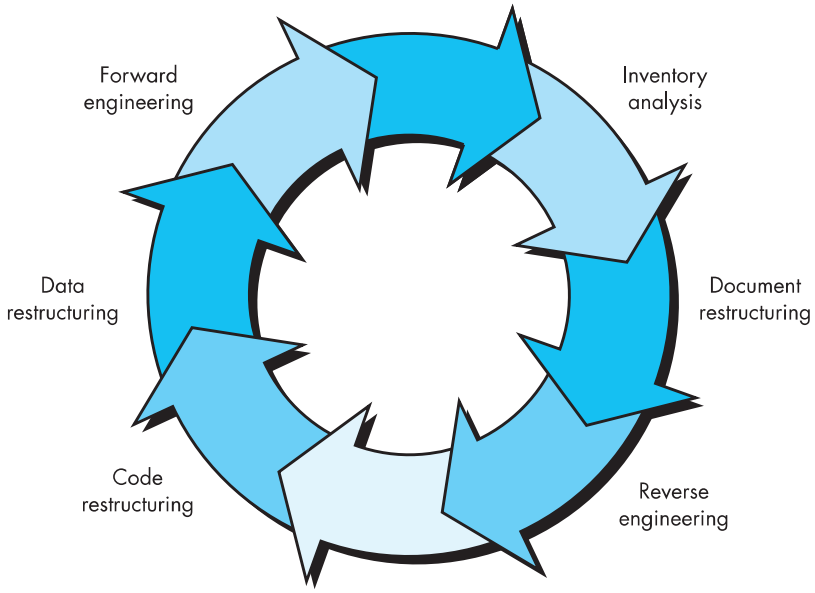
A workable strategy is encompassed in a reengineering process model. We'll discuss the model later in this section, but first, some basic principles.

Reengineering is a rebuilding activity. To better understand it, consider an analogous activity: the rebuilding of a house. Consider the following situation. You've purchased a house in another state. You've never actually seen the property, but you acquired it at an amazingly low price, with the warning that it might have to be completely rebuilt. How would you proceed?

- Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you (or a professional inspector) would create a list of criteria so that your inspection would be systematic.
- Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding (at much lower cost and in much less time).
- Before you start rebuilding be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
- If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.

FIGURE 36.2

A software reengineering process model



- If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.

Although these principles focus on the rebuilding of a house, they apply equally well to the reengineering of computer-based systems and applications.

To implement these principles, you can use a software reengineering process model that defines six activities, shown in Figure 36.2. In some cases, these activities occur in a linear sequence, but this is not always the case. For example, it may be that reverse engineering (understanding the internal workings of a program) may have to occur before document restructuring can commence.

36.5.2 Software Reengineering Activities

ADVICE

If time and resources are in short supply, you might consider applying the Pareto principle to the software that is to be reengineered. Apply the reengineering process to the 20 percent of the software that accounts for 80 percent of the problems.

The reengineering paradigm shown in Figure 36.2 is a cyclical model. This means that each of the activities presented as a part of the paradigm may be revisited. For any particular cycle, the process can terminate after any one of these activities.

Inventory analysis. Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability and supportability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

It is important to note that the inventory should be revisited on a regular basis. The status of applications (e.g., business criticality) can change as a function of time, and as a result, priorities for reengineering will shift.

Document restructuring. Weak documentation is the trademark of many legacy systems. But what can you do about it? What are your options? In some cases, creating documentation when none exists is simply too costly. If the software works, let it be! In other cases, some documentation must be created, but only when changes are made. If a modification occurs, document it. Finally, there are situations in which a critical system must be fully documented, but even here, documents should achieve an essential minimum. Your software organization must choose the documentation option that is most appropriate for each case.

Create only as much documentation as you need to understand the software, not one page more.

Reverse engineering. Reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of *design recovery*. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Code restructuring. The most common type of reengineering (actually, the use of the term *reengineering* is questionable in this case) is *code restructuring*.[4] Some legacy systems have a relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted and code is then restructured (this can be done automatically) or even rewritten in a more modern programming language. The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

Data restructuring. A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, information architecture has more to do with the long-term viability of a program than the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data restructuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

When data structure is weak (e.g., flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered.

4 Code restructuring has some of the elements of "refactoring," a redesign concept introduced in Chapter 12 and discussed elsewhere in this book.

Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

Forward engineering. In an ideal world, applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an "engine" will appear, but vendors have introduced tools that provide a limited subset of these capabilities that addresses specific application domains (e.g., applications that are implemented using a specific database system). More important, these reengineering tools are becoming increasingly more sophisticated.

Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software recreates the function of the existing system and also adds new functions and/or improves overall performance.

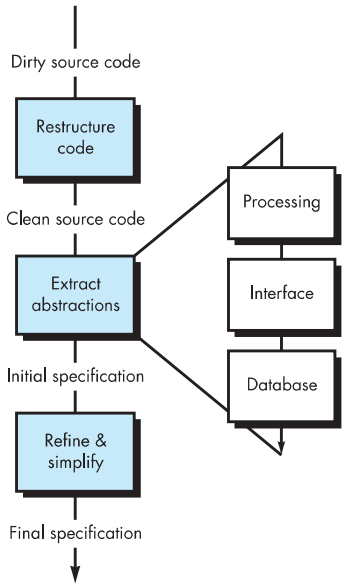
36.6 Reverse Engineering

Reverse engineering conjures an image of the "magic slot." You feed a haphazardly designed, undocumented source file into the slot and out the other end comes a complete design description (and full documentation) for the computer program. Unfortunately, the magic slot doesn't exist. Reverse engineering can extract design information from source code, but the abstraction level, the completeness of the documentation, the degree to which tools and a human analyst work together, and the directionality of the process are highly variable.

The *abstraction level* of a reverse engineering process and the tools used to effect it refers to the sophistication of the design information that can be extracted from source code. Ideally, the abstraction level should be as high as possible. That is, the reverse engineering process should be capable of deriving procedural design representations (a low-level abstraction), program and data structure information (a somewhat higher level of abstraction), object models, data and/or control flow models (a relatively high level of abstraction), and data models (a high level of abstraction). As the abstraction level increases, you are provided with information that will allow easier understanding of the program.

The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple architectural design representations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models.

FIGURE 36.3
The reverse engineering process



KEY POINT
Three reverse engineering issues must be addressed: abstraction level, completeness, and directionality.

Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. *Interactivity* refers to the degree to which the human is "integrated" with automated tools to create an effective reverse engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.

If the *directionality* of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.

WebRef
Useful resources for "design recovery and program understanding" can be found at **http://www.softpanorama.net/SE/reverse_engineering_links.shtml.**

The reverse engineering process is represented in Figure 36.3. Before reverse engineering activities can commence, unstructured ("dirty") source code is restructured (Section 36.5.1) so that it contains only the structured programming constructs.[5] This makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.

The core of reverse engineering is an activity called *extract abstractions*. You must evaluate the old program and from the (often undocumented) source code, develop a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

5 Code can be restructured using a *restructuring engine*—a tool that restructures source code.

36.6.1 Reverse Engineering to Understand Data

ADVICE

In some cases, the first reengineering activity attempts to construct a UML class diagram.

Reverse engineering of data occurs at different levels of abstraction and is often the first reengineering task. At the program level, internal program data structures must often be reverse engineered as part of an overall reengineering effort. At the system level, global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems). Reverse engineering of the current global data structures sets the stage for the introduction of a new systemwide database.

ADVICE

The approach to reverse engineering for data for conventional software follows an analogous path: (1) build a data model, (2) identify attributes of data objects, and (3) define relationships.

Internal data structures. Reverse engineering techniques for internal program data focus on the definition of classes of objects. This is accomplished by examining the program code with the intent of grouping related program variables. In many cases, the data organization within the code identifies abstract data types. For example, record structures, files, lists, and other data structures often provide an initial indicator of classes.

Database structure. Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database schema into another requires an understanding of existing objects and their relationships.

The following steps [Pre94] may be used to define the existing data model as a precursor to reengineering a new database model: (1) build an initial object model, (2) determine candidate keys (the attributes are examined to determine whether they are used to point to another record or table; those that serve as pointers become candidate keys), (3) refine the tentative classes, (4) define generalizations, and (5) discover associations using techniques that are analogous to the CRC approach. Once information defined in the preceding steps is known, a series of transformations [Pre94] can be applied to map the old database structure into a new database structure.

Quote:

"There exists a passion for comprehension, just as there exists a passion for music. That passion is rather common in children, but gets lost in most people later on."

Albert Einstein

36.6.2 Reverse Engineering to Understand Processing

Reverse engineering to understand processing begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

The overall functionality of the entire application must be understood before more detailed reverse engineering work occurs. This establishes a context for further analysis and provides insight into interoperability issues among applications within a larger system. Each of the programs that make up the system represents a functional abstraction at a high level of detail. A block diagram, representing the interaction between these functional abstractions, is created. Each

component performs some subfunction and represents a defined procedural abstraction. A processing narrative for each component is developed. In some situations, system, program, and component specifications already exist. When this is the case, the specifications are reviewed for conformance to existing code.[6]

Things become more complex when the code inside a component is considered. You should look for sections of code that represent generic procedural patterns. In almost every component, a section of code prepares data for processing (within the module), a different section of code does the processing, and another section of code prepares the results of processing for export from the component. Within each of these sections, you can encounter smaller patterns; for example, data validation and bounds checking often occur within the section of code that prepares data for processing.

For large systems, reverse engineering is generally accomplished using a semi-automated approach. Automated tools can be used to help you understand the semantics of existing code. The output of this process is then passed to restructuring and forward engineering tools to complete the reengineering process.

36.6.3 Reverse Engineering User Interfaces

Sophisticated GUIs have become de rigueur for computer-based products and systems of every type. Therefore, the redevelopment of user interfaces has become one of the most common types of reengineering activity. But before a user interface can be rebuilt, reverse engineering should occur.

To fully understand an existing user interface, the structure and behavior of the interface must be specified. Merlo and his colleagues [Mer93] suggest three basic questions that must be answered as reverse engineering of the UI commences:

How do I understand the working of an existing user interface?

- What are the basic actions (e.g., keystrokes and mouse clicks) that the interface must process?
- What is a compact description of the behavioral response of the system to these actions?
- What is meant by a "replacement," or more precisely, what concept of equivalence of interfaces is relevant here?

Behavioral modeling notation (Chapter 11) can provide a means for developing answers to the first two questions. Much of the information necessary to create a behavioral model can be obtained by observing the external manifestation of the existing interface. But additional information necessary to create the behavioral model must be extracted from the code.

It is important to note that a replacement GUI may not mirror the old interface exactly (in fact, it may be radically different). It is often worthwhile to develop a new interaction metaphor. For example, an old UI requests that a user

6 Often, specifications written early in the life history of a program are never updated. As changes are made, the code no longer conforms to the specification.

provide a scale factor (ranging from 1 to 10) to shrink or magnify a graphical image. A reengineered GUI might use a touch-screen slide bar to accomplish the same function.

SOFTWARE TOOLS

Reverse Engineering

Objective: To help software engineers understand the internal design structure of complex programs.

Mechanics: In most cases, reverse engineering tools accept source code as input and produce a variety of structural, procedural, data, and behavioral design representations.

Representative Tools:[7]

Imagix 4D, developed by Imagix (**www.imagix.com**), "helps software developers understand complex or legacy C and C++ software" by reverse engineering and documenting source code.

Understand, developed by Scientific Toolworks (**www.scitools.com**), parse Ada, Fortran, C, C++, C#, PHP, HTML, JavaScript, Python, and Java "to reverse-engineer, automatically document, calculate code metrics, and help you understand, navigate and maintain source code."

A list of reverse engineering tools can be found at **http://www.eclipse.org/gmt/modisco/relatedProjects.php**.

36.7 RESTRUCTURING

Software restructuring modifies source code and/or data in an effort to make it amenable to future changes. In general, restructuring does not modify the overall program architecture. It tends to focus on the design details of individual modules and on local data structures defined within modules. If the restructuring effort extends beyond module boundaries and encompasses the software architecture, restructuring becomes forward engineering (Section 36.8).

Restructuring occurs when the basic architecture of an application is solid, even though technical internals need work. It is initiated when major parts of the software are serviceable and only a subset of all modules and data need extensive modification.[8]

Although code restructuring can alleviate immediate problems associated with debugging or small changes, it is not reengineering. Real benefit is achieved only when data and architecture are restructured.

36.7.1 Code Restructuring

Code restructuring is performed to yield a design that produces the same function but with higher quality than the original program. In general, code restructuring techniques (e.g., Warnier's logical simplification techniques [War74]) model program logic using Boolean algebra and then apply a series of transformation rules that yield restructured logic. The objective is to take "spaghetti-bowl" code and derive a procedural design that conforms to the structured programming philosophy (Chapter 19).

7 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

8 It is sometimes difficult to make a distinction between extensive restructuring and redevelopment. Both are reengineering.

Other restructuring techniques have also been proposed for use with reengineering tools. A resource exchange diagram maps each program module and the resources (data types, procedures, and variables) that are exchanged between it and other modules. By creating representations of resource flow, the program architecture can be restructured to achieve minimum coupling among modules.

36.7.2 Data Restructuring

Before data restructuring can begin, a reverse engineering activity called *analysis of source code* should be conducted. All programming language statements that contain data definitions, file descriptions, I/O, and interface descriptions are evaluated. The intent is to extract data items and objects, to get information on data flow, and to understand the existing data structures that have been implemented. This activity is sometimes called *data analysis*.

Once data analysis has been completed, *data redesign* commences. In its simplest form, a *data record standardization* step clarifies data definitions to achieve consistency among data item names or physical record formats within an existing data structure or file format. Another form of redesign, called *data name rationalization,* ensures that all data naming conventions conform to local standards and that aliases are eliminated as data flow through the system.

When restructuring moves beyond standardization and rationalization, physical modifications to existing data structures are made to make the data design more effective. This may mean a translation from one file format to another, or in some cases, translation from one type of database to another.

SOFTWARE TOOLS

Software Restructuring

Objective: The objective of restructuring tools is to transform older unstructured computer software into modern programming languages and design structures.

Mechanics: In general, source code is input and transformed into a better structured program. In some cases, the transformation occurs within the same programming language. In other cases, an older programming language is transformed into a more modern language.

Representative Tools[9]

DMS Software Reengineering Toolkit, developed by Semantic Design (**www.semdesigns.com**), provide a variety of restructuring capabilities for COBOL, C/C++, Java, Fortran 90, and VHDL.

Clone Doctor, developed by Semantic Designs (**www.semdesigns.com**), analyzes and transforms programs written in C, C++, Java, or COBOL or any other text-based computer language.

plusFORT, developed by Polyhedron (**www.polyhedron.com**), is a suite of FORTRAN tools that contains capabilities for restructuring poorly designed FORTRAN programs into the modern FORTRAN or C standard.

Pointers to a variety of reengineering and reverse engineering tools can be found at **http://www.comp.lancs.ac.uk/projects/renaissance/RenaissanceWeb/Reengineering/Tools.html** and **http://www.fujaba.de/projects.html**.

9 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

36.8 Forward Engineering

A program with control flow that is the graphic equivalent of a bowl of spaghetti, with "modules" that are 2,000 statements long, with few meaningful comment lines in 290,000 source statements and no other documentation must be modified to accommodate changing user requirements. You have the following options:

? What options exist when we're faced with a poorly designed and implemented program?

1. You can struggle through modification after modification, fighting the ad hoc design and tangled source code to implement the necessary changes.
2. You can attempt to understand the broader inner workings of the program in an effort to make modifications more effectively.
3. You can redesign, recode, and test those portions of the software that require modification, applying a software engineering approach to all revised segments.
4. You can completely redesign, recode, and test the program, using reengineering tools to assist us in understanding the current design.

There is no single "correct" option. Circumstances may dictate the first option even if the others are more desirable.

Rather than waiting until a maintenance request is received, the development or support organization uses the results of inventory analysis to select a program that (1) will remain in use for a preselected number of years, (2) is currently being used successfully, and (3) is likely to undergo major modification or enhancement in the near future. Then, option 2, 3, or 4 is applied.

Reengineering is a lot like getting your teeth cleaned. You can think of a thousand reasons to delay it, and you'll get away with procrastinating for quite a while. But eventually, your delaying tactics will come back to cause pain.

At first glance, the suggestion that you redevelop a large program when a working version already exists may seem quite extravagant. Before passing judgment, consider the following arguments. The cost to maintain one line of source code may be 20 to 40 times the cost of initial development of that line. In addition, redesign of the software architecture (program and/or data structure), using modern design concepts, can greatly facilitate future maintenance. Because a prototype of the software already exists, development productivity should be much higher than average. The user now has experience with the software. Therefore, new requirements and the direction of change can be ascertained with greater ease. Automated tools for reengineering will facilitate some parts of the job. And finally, a complete software configuration (documents, programs, and data) will exist upon completion of preventive maintenance.

A large in-house software developer (e.g., a business systems software development group for a large consumer products company) may have 500 to 2,000 production programs within its domain of responsibility. These programs can be ranked by importance and then reviewed as candidates for forward engineering.

In most cases, forward engineering does not simply create a modern equivalent of an older program. Rather, new user and technology requirements are integrated into the reengineering effort. The redeveloped program extends the capabilities of the older application.

36.8.1 Forward Engineering for Client-Server Architectures

Over the past few decades, centralized computing resources (including software) have been distributed among many client platforms. Although a variety of different distributed environments can be designed, the typical centralized application that has been reengineered into a client-server architecture has the following features: application functionality migrates to each client computer, new GUI interfaces are implemented at the client sites, database functions are allocated to the server, specialized functionality (e.g., compute-intensive analysis) may remain at the server site, and new communications, security, archiving, and control requirements must be established at both the client and server sites. It is important to note that the migration from centralized computing to client-server computing requires both business and software reengineering.

In some cases, migration to a client-server architecture should be approached not as reengineering, but as a new development effort. Reengineering enters the picture only when specific functionality from the old system is to be integrated into the new architecture.

Reengineering for client-server applications begins with a thorough analysis of the business environment that encompasses the existing mainframe. Three layers of abstraction can be identified. The *database sits* at the foundation of a client-server architecture and manages transactions and queries from server applications. Yet these transactions and queries must be controlled within the context of a set of business rules (defined by an existing or reengineered business process). Client applications provide targeted functionality to the user community.

The functions of the existing database management system and the data architecture of the existing database must be reverse engineered as a precursor to the redesign of the database foundation layer. The client-server database is reengineered to ensure that transactions are executed in a consistent manner, that all updates are performed only by authorized users, that core business rules are enforced (e.g., before a vendor record is deleted, the server ensures that no related accounts payable, contracts, or communications exist for that vendor), that queries can be accommodated efficiently, and that full archiving capability has been established.

The business rules layer represents software resident at both the client and the server. This software performs control and coordination tasks to ensure that transactions and queries between the client application and the database conform to the established business process.

The client applications layer implements business functions that are required by specific groups of end users. In many instances, an older centralized application is segmented into a number of smaller, reengineered desktop applications.

Communication among the desktop applications (when necessary) is controlled by the business rules layer.

A comprehensive discussion of client-server software design and reengineering is best left to books dedicated to the subject. If you have further interest, see [Van02], [Cou00], or [Orf99].

36.8.2 Forward Engineering for Object-Oriented Architectures

Object-oriented software engineering has become the development paradigm of choice for many software organizations. But what about existing applications that were developed using conventional methods? In some cases, the answer is to leave such applications "as is." In others, older applications must be reengineered so that they can be easily integrated into large, object-oriented systems.

Reengineering conventional software into an object-oriented implementation uses many of the same techniques discussed in Part 2 of this book. First, the existing software is reverse engineered so that appropriate data, functional, and behavioral models can be created. If the reengineered system extends the functionality or behavior of the original application, use cases (Chapters 8 and 9) are created. The data models created during reverse engineering are then used in conjunction with CRC modeling (Chapter 10) to establish the basis for the definition of classes. Class hierarchies, object-relationship models, object-behavior models, and subsystems are defined, and object-oriented design commences.

As object-oriented forward engineering progresses from analysis to design, a CBSE process model (Chapter 10) can be invoked. If the existing application resides within a domain that is already populated by many object-oriented applications, it is likely that a robust component library exists and can be used during forward engineering.

Quote:

"You can pay a little now, or you can pay a lot more later."

Sign in an auto dealership suggesting a tune-up

For those classes that must be engineered from scratch, it may be possible to reuse algorithms and data structures from the existing conventional application. However, these must be redesigned to conform to the object-oriented architecture.

36.9 The Economics of Reengineering

In a perfect world, every unmaintainable program would be retired immediately, to be replaced by high-quality, reengineered applications developed using modern software engineering practices. But we live in a world of limited resources. Reengineering drains resources that can be used for other business purposes. Therefore, before an organization attempts to reengineer an existing application, it should perform a cost-benefit analysis.

A cost-benefit analysis model for reengineering has been proposed by Sneed [Sne95]. Nine parameters are defined:

P_1 = Current annual maintenance cost for an application
P_2 = Current annual operations cost for an application
P_3 = Current annual business value of an application
P_4 = Predicted annual maintenance cost after reengineering
P_5 = Predicted annual operations cost after reengineering
P_6 = Predicted annual business value after reengineering
P_7 = Estimated reengineering costs
P_8 = Estimated reengineering calendar time
P_9 = Reengineering risk factor ($P_9 = 1.0$ is nominal)
L = Expected life of the system

The cost associated with continuing maintenance of a candidate application (i.e., reengineering is not performed) can be defined as

$$C_{\text{maint}} = [P_3 - (P_1 + P_2)] \times \text{L} \tag{36.1}$$

The costs associated with reengineering are defined using the following relationship:

$$C_{\text{reeng}} = P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9) \tag{36.2}$$

Using the costs presented in Equations (36.1) and (36.2), the overall benefit of reengineering can be computed as

$$\text{Cost benefit} = C_{\text{reeng}} - C_{\text{maint}} \tag{36.3}$$

The cost-benefit analysis presented in these equations can be performed for all high-priority applications identified during inventory analysis (Section 36.4.2). Those applications that show the highest cost-benefit can be targeted for reengineering, while work on others can be postponed until resources are available.

36.10 Summary

Software maintenance and support are ongoing activities that occur throughout the life cycle of an application. During these activities, defects are corrected, applications are adapted to a changing operational or business environment, enhancements are implemented at the request of stakeholders, and users are supported as they integrate an application into their personal or business workflow.

Reengineering occurs at two different levels of abstraction. At the business level, reengineering focuses on the business process with the intent of making changes to improve competitiveness in some area of the business. At the software level, reengineering examines information systems and applications with the intent of restructuring or reconstructing them so that they exhibit higher quality.

Business process reengineering defines business goals; identifies and evaluates existing business processes (in the context of defined goals); specifies and designs revised processes; and prototypes, refines, and instantiates them within a business. BPR has a focus that extends beyond software. The result of BPR is often the definition of ways in which information technologies can better support the business.

Software reengineering encompasses a series of activities that include inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. The intent of these activities is to create versions of existing programs that exhibit higher quality and better maintainability—programs that will be viable well into the twenty-first century.

The cost-benefit of reengineering can be determined quantitatively. The cost of the status quo, that is, the cost associated with ongoing support and maintenance of an existing application, is compared to the projected costs of reengineering and the resultant reduction in maintenance and support costs. In almost every case in which a program has a long life and currently exhibits poor maintainability or supportability, reengineering represents a cost-effective business strategy.

Problems and Points to Ponder

36.1. Consider any job that you've held in the last five years. Describe the business process in which you played a part. Use the BPR model described in Section 36.4.2 to recommend changes to the process in an effort to make it more efficient.

36.2. Do some research on the efficacy of business process reengineering. Present pro and con arguments for this approach.

36.3. Your instructor will select one of the programs that everyone in the class has developed during this course. Exchange your program randomly with someone else in the class. Do not explain or walk through the program. Now, implement an enhancement (specified by your instructor) in the program you have received.

a. Perform all software engineering tasks including a brief walkthrough (but not with the author of the program).
b. Keep careful track of all errors encountered during testing.
c. Discuss your experiences in class.

36.4. Explore the inventory analysis checklist presented at the SEPA website and attempt to develop a quantitative software rating system that could be applied to existing programs in an effort to pick candidate programs for reengineering. Your system should extend beyond the economic analysis presented in Section 36.9.

36.5. Suggest alternatives to paper and ink or conventional electronic documentation that could serve as the basis for document restructuring. (Hint: Think of new descriptive technologies that could be used to communicate the intent of the software.)

36.6. Some people believe that artificial intelligence technology will increase the abstraction level of the reverse engineering process. Do some research on this subject (i.e., the use of AI for reverse engineering), and write a brief paper that takes a stand on this point.

36.7. Why is completeness difficult to achieve as abstraction level increases?

36.8. Why must interactivity increase if completeness is to increase?

36.9. Using information obtained via the Web, present characteristics of three reverse engineering tools to your class.

36.10. There is a subtle difference between restructuring and forward engineering. What is it?

36.11. Research the literature and/or Internet sources to find one or more papers that discuss case studies of mainframe to client-server reengineering. Present a summary.

36.12. How would you determine P_4 through P_7 in the cost-benefit model presented in Section 36.9?

Further Readings and Information Sources

It is ironic that software maintenance and support represent the most costly activities in the life of an application, and yet, fewer books have been written about maintenance and support than any other major software engineering topics. Among recent additions to the literature are books by Reifer (*Software Maintenance Success Recipes,* Auerbach, 2011), Jarzabek (*Effective Software Maintenance and Evolution,* Auerbach, 2007), Grubb and Takang (*Software Maintenance: Concepts and Practice,* World Scientific Publishing Co., 2nd ed., 2003), and Pigoski (*Practical Software Maintenance,* Wiley, 1996). These books cover basic maintenance and support practices and present useful management guidance. Maintenance techniques that focus on client-server environments are discussed by Schneberger (*Client/Server Software Maintenance,* McGraw-Hill, 1997). Current research in "software evolution" is presented in an anthology edited by Mens and Demeyer (*Software Evolution,* Springer, 2008).

Like many hot topics in the business community, the hype surrounding business process reengineering has given way to a more pragmatic view of the subject. Hammer and Champy (*Reengineering the Corporation*, HarperBusiness, revised edition, 2003) precipitated early interest with their best-selling book. Other books by Jacka and Keller (*Business Process Mapping: Improving Customer Satisfaction,* 2nd ed., Wiley, 2009), Sharp and McDermott (*Workflow Modeling,* 2nd ed., Artech House, 2008), Andersen (*Business Process Improvement Toolbox,* 2nd ed., American Society for Quality, 2007), Smith and Fingar [*Business Process Management (BPM): The Third Wave,* Meghan-Kiffer Press, 2003], and Harrington et al. (*Business Process Improvement Workbook,* McGraw-Hill, 1997) present case studies and detailed guidelines for BPR.

Abfalter (*Software Reengineering,* VDM Verlag, 2008), Fong (*Information Systems Reengineering and Integration,* Springer, 2006) describes database conversion techniques, reverse engineering, and forward engineering as they are applied for major information systems. Nierstrasz and his colleagues (*Object Oriented Reengineering Patterns,* Square Bracket Associates, 2009) provides a patterns-based view of how to refactor and/or reengineer OO systems. Secord and his colleagues (*Modernizing Legacy Systems,* Addison-Wesley, 2003), Ulrich (*Legacy Systems: Transformation Strategies*, Prentice Hall, 2002), Valenti (*Successful Software Reengineering*, IRM Press, 2002), and Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software,* Fitzroy Dearborn Publishers, 1999) focus on strategies and practices for reengineering at a technical level. Miller (*Reengineering Legacy Software Systems*, Digital Press, 1998) "provides a framework for keeping application systems synchronized with business strategies and technology changes."

Cameron (*Reengineering Business for Success in the Internet Age,* Computer Technology Research, 2000) and Umar (*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies,* Prentice Hall, 1997) provide worthwhile guidance for organizations that want to transform legacy systems into a Web-based environment. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems,* Prentice Hall, 1996) discusses the bridge between BPR and information technology. Aiken (*Data Reverse Engineering,* McGraw-Hill, 1996) discusses how to reclaim, reorganize, and reuse organizational data. Arnold (*Software Reengineering,* IEEE Computer Society Press, 1993) has put together an excellent anthology of early papers that focus on software reengineering technologies.

A wide variety of information sources on software reengineering is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.